Objets historiques et annotations pour les environnements logiciels

Rubby Casallas

THESE

présentée par

Rubby Casallas-Gutiérrez

pour obtenir le titre de

Docteur de l'Université Joseph Fourier

Grenoble I

(arrêtés ministériels du 5 juillet 1984 et du 30 mars 1992)

spécialité : INFORMATIQUE

Objets historiques et annotations pour les environnements logiciels

Soutenue le 24 mai de 1996 devant le jury composé de :

Président :	Michel Adiba
Rapporteurs :	M. Claude Godart
	M. Flavio Oquendo
Examinateur :	Mme. Christine Collet
Directeur de la thèse :	M. Jacky Estublier

Thèse préparée au sein du Laboratoire Logiciels, Systèmes et Réseaux de Grenoble

Je tiens à remercier,

M. Flavio Oquendo, professeur à l'Université de Savoie, d'avoir accepté de rapporter cette thèse, pour le temps qu'il m'a consacré, ainsi que pour ses commentaires éclairés qui m'ont permis d'améliorer ce manuscrit.

M. Claude Godart, professeur à l'Université Henri Poincaré de Nancy, qui a bien voulu être rapporteur de mon travail.

M. Michel Adiba, professeur à l'Université Joseph Fourier, de me faire l'honneur de présider mon jury de thèse.

Mme. Christine Collet, maître de conférences à l'Université Joseph Fourier, pour sa participation à ce jury, pour sa lecture constructive et ses conseils. Je tiens à la remercier tout particulièrement pour l'intérêt qu'elle m'a toujours témoigné au cours de ces dernières années.

M. Jacky Estublier, Directeur de Recherches au CNRS, de m'avoir donné l'opportunité, en m'accueillant dans son équipe, de participer au projet Adèle. Ce fut pour moi une expérience très enrichissante qui m'a permis de découvrir le génie logiciel sous un nouveau jour... *in-the-large*. Je le remercie également pour ses conseils, ses encouragements, sa disponibilité et finalement de m'avoir laissé bénéficier de son *savoir-faire*. Je lui suis très reconnaissante, je lui dois beaucoup.

Pierre Girard (pour sa lecture, sa re-lecture, sa re-re-lecture, ...), Miguel Santana (qui a courageusement essuyé les premiers plâtres), Claudia Jiménez-Domínguez (pour ses remarques et conseils), Claudia Roncancio (pour être toujours là), Jérôme Gensel (pour son français mais aussi pour son anglais), à tous pour leur gentillesse, pour leur intérêt, pour leur *patience*,... et pour leur(s) lecture(s), évidemment ! Ce manuscrit leur doit beaucoup.

M. Pierre Claude Scholl et Mlle. Marie Christine Fauvet pour leur intérêt pour mon travail, leur écoute et les références bibliographiques sur les bases de données temporelles.

Tous les membres de l'équipe Adèle y compris l'équipe *technique*, Christophe et Jean. Je remercie tout particulièrement Jean pour avoir *vachement* enrichi mon vocabulaire français.

Je voudrais remercier spécialement Jean-Marie Favre pour m'avoir fait partager son savoir et sa culture à travers ces *bavardages*, au demeurant bien innocents pour un profane. Je lui suis aussi reconnaissant de m'avoir fait profiter de sa joie et sa bonne humeur.

Ma famille et mes amis pour leur soutien et leur compréhension pendant ces années.

A "El Mono"

Résumé

Dans un environnement guidé par les procédés de fabrication de logiciel (EGPFL), la gestion de l'information est un problème complexe qui doit concilier deux besoins : gérer le produit logiciel et gérer les procédés de fabrication. Outre la grande quantité d'entités diverses et fortement interdépendantes, la gestion du produit doit prendre en compte l'aspect évolutif et les facteurs de variation du logiciel, ainsi que la nature coopérative des activités de fabrication des logiciels. La gestion des procédés recouvre la modélisation, l'exécution, l'évaluation et la supervision des procédés. Diverses informations doivent alors être prises en compte : la trace d'exécution des procédés, les événements survenus dans l'environnement et les mesures de qualité.

Nous proposons les *objets historiques annotés* pour gérer l'information d'un EGPFL. L'objet historique constitue la notion de base d'un modèle à objets *historique* permettant de représenter à la fois les entités logicielles et leur évolution. La notion d'annotation vient, quant à elle, enrichir ce modèle pour permettre d'introduire des informations qui dénotent des faits (notes, mesures, observations, etc) pouvant être ponctuellement associés aux entités de l'EGPFL. Un langage de requêtes est défini afin d'accéder aux différentes informations. Grâce à ce langage, l'EGPFL dispose d'un service puissant pour rassembler, à partir de la base d'objets, les diverses informations nécessaires à l'évaluation et au contrôle des procédés de fabrication.

Nous proposons également d'exploiter les possibilités offertes par notre modèle pour définir des événements et, éventuellement, en conserver un historique. Les événements permettent d'identifier des situations liant des informations provenant aussi bien de l'état courant que des états passés de l'EGPFL. C'est pourquoi la définition d'un événement peut comporter des conditions exprimées dans le langage de requêtes. L'emploi d'annotations permet d'enregistrer les occurrences d'événements, ainsi qu'une partie de l'état du système. Une implantation du modèle est proposée dans le système Adèle.

Abstract

In software process-centered environments (SPCE), information management is a complex problem which must conciliate two requirements: product management and process management. Besides the large quantity of diverse and highly interdependent entities, product management must take into account evolution aspects and variation factors of software, as well as, the cooperative nature of software activities. Process management encompasses modelling, enactment, performance and quality assessment processes. Various information types must be taken into account: trace of process execution, events raised into the environment and quality measures.

We propose *Annotated Historical Objects* as a support to represent the information of an SPCE. Historical Object is the basic notion of our historical object model which allows to model software artifacts and their evolution. Annotation notion enhances the model by allowing information to be introduced for denoting facts (notes, observations, measures, etc.) which can be associated to others entities in the system. A *navigational and historical* query language has been defined to access the various data. Thanks to this language, the SPCE provides a powerful service to collect from the object database the information needed to evaluate and to control software processes.

In addition, we propose to use the possibilities offered by this model to define complex events and possibly, to keep its history. Events can identify situations which involve current and past states of the system. For that, event definition can include conditions expressed in the query language. Annotations allow to record event occurrences and system states. An implementation is proposed into the Adele system.

Tabla de matières

Introduction

L'objectif des organisations fabricant des logiciels est de produire de façon systématique et prévisible des logiciels de bonne qualité. A cet effet, le meilleur atout de ce type d'organisation reste son *savoir-faire* ; c'est-à-dire les procédés de fabrication de logiciels (*software processes)* qu'elle met en oeuvre.

En effet, la *maturité* de toute organisation est en relation directe avec la capacité de ses procédés à produire les résultats prévues. Pour les fabricants de logiciels, acquérir une telle *maturité* est une mission difficile qui peut prendre beaucoup de temps et de ressources [Hum94]. La difficulté réside dans le fait que la plupart des activités constituant les procédés de fabrication de logiciel ne sont pas automatisables (*human-intensive*) et, par conséquent, non déterministes car les interactions entre humains, ainsi qu'entre humains et outils, sont très variables et souvent imprévisibles [CFFS92].

Ces dernières années, les procédés de fabrication de logiciels ont fait l'objet d'une attention grandissante dans le domaine du génie logiciel. Il est désormais admis que la qualité des logiciels peut être améliorée si les procédés de fabrication sont eux-mêmes améliorés. Les fabricants de logiciels commencent aussi à admettre que l'évaluation, et donc la *mesure,* est une tâche indispensable pour obtenir une amélioration systématique des procédés de fabrication [LHR95].

Les recherches autour de procédés logiciels ont abouti, d'une part, à de nombreuses études et propositions de modélisation de procédés et, d'autre part, à la définition d'environnements guidés par les procédés [FKN94] [DF96]. Les modèles de procédés déterminent et intègrent l'utilisation des services de l'environnement et guident les tâches des utilisateurs.

Notre travail s'inscrit dans la problématique générale de la construction d'environnements guidés par les procédés de fabrication de logiciels. La modélisation et l'exécution des modèles de procédés ne sont pas, en soi, suffisantes pour atteindre l'objectif d'amélioration de procédés. Il est nécessaire aussi de pouvoir observer, contrôler et mesurer l'exécution des procédés. Ces besoins rendent complexe la gestion de l'information de l'environnement.

Outre la grande quantité d'entités diverses et fortement interdépendantes, la gestion du produit doit prendre en compte l'aspect évolutif, les facteurs de variation du logiciel ainsi que la nature coopérative des activités de fabrication des logiciels. En effet, un logiciel doit être corrigé, amélioré et complété au cours du temps. Il peut aussi varier selon différentes plate-formes de développement ou d'utilisation, il peut être adapté aux

besoins spécifiques d'une entreprise ou en fonction des préférences des utilisateurs. Par ailleurs, les activités de fabrication des logiciels sont de longue durée et sont réalisées par des équipes nombreuses qui partagent des informations.

La gestion des procédés de fabrication recouvre la modélisation, l'exécution, l'évaluation et la supervision des procédés. Diverses informations doivent alors être pris en compte : la trace de l'exécution des procédés, les événements survenus dans l'environnement et les mesures de qualité.

Notre démarche consiste à identifier les caractéristiques requises par le gestionnaire d'objets d'un environnement guidé par les procédés. Nous proposons un modèle historique objets associations (**MOHA**) pour la représentation des informations gérées dans un tel environnement, ainsi qu'un langage de requêtes de type navigationnel et historique (**LOHA**).

Le concept de base du modèle est l'*objet historique annoté*. Il permet de représenter à la fois les entités logicielles et leur évolution. Dans ce modèle, la valeur d'un objet est fonction du temps. La notion d'annotation vient enrichir ce modèle pour permettre d'introduire des informations qui dénotent des faits (notes, mesures, observations, etc) pouvant être ponctuellement associés aux entités de l'environnement (les composants logiciels, les activités de fabrication, les utilisateurs, etc).

Associé au modèle le langage de requêtes permet d'interroger une base d'objet historiques annotés. Grâce à ce langage, l'environnement dispose d'un service puissant pour rassembler, à partir de la base, les diverses informations nécessaires à l'évaluation et au contrôle des procédés de fabrication.

Nous proposons également d'exploiter les possibilités offertes par le modèle pour définir des événements permettant d'identifier des situations liant des informations provenant aussi bien de l'état courant que des états passés de l'environnement. En effet, leur définition peut comporter des conditions exprimées dans le langage de requêtes. Par ailleurs, les annotations permettent d'enregistrer les occurrences d'un événement ainsi qu'une partie de l'état du système. Il devient, dès lors, possible de conserver un historique des événements ainsi que de leurs contextes.

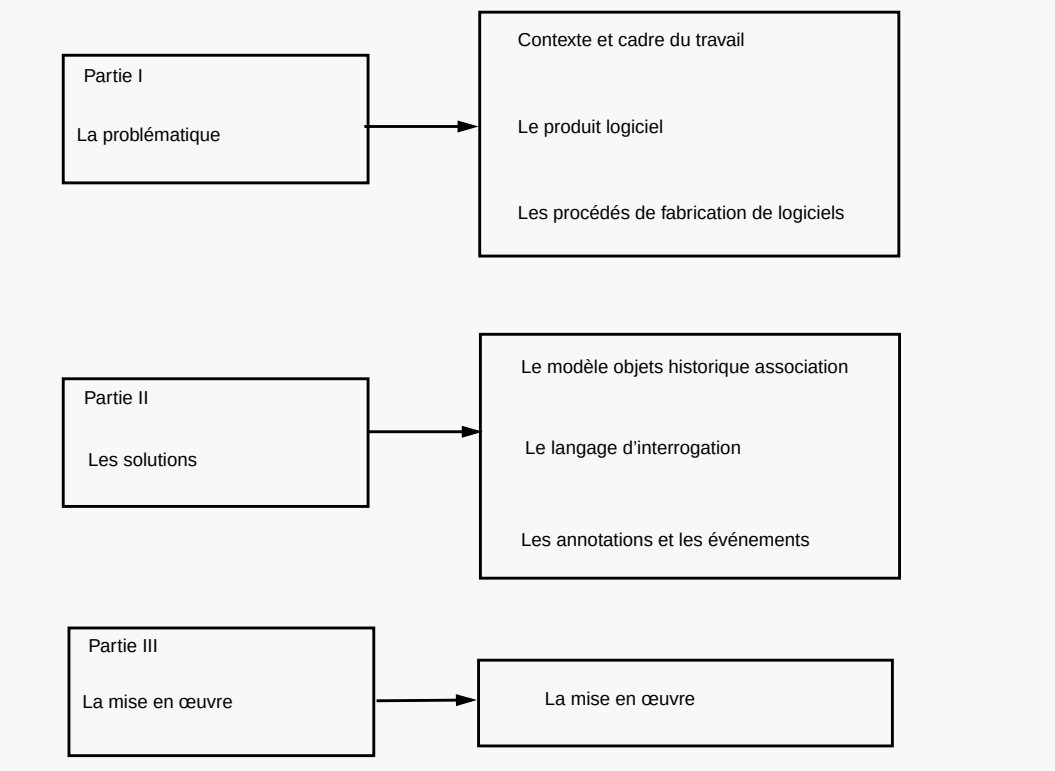
Organisation du document

Ce document comprend trois parties. La première partie, composée de trois chapitres, a pour objectif d'introduire la problématique de notre travail. Le premier chapitre fait un parcours général du contexte dans lequel s'inscrit notre travail et présente le cadre dans lequel il a été réalisé. Le deuxième chapitre est dédié à la problématique de la gestion des produits logiciels de grande taille. Le troisième chapitre introduit les procédés de fabrication de logiciels en se focalisant sur l'impact que peut avoir la gestion explicite des procédés sur le gestionnaire des objets de l'environnement.

La seconde partie, composée de trois chapitres, présente les solutions que nous proposons.

Le premier chapitre présente **MOHA**, le modèle d'objets historiques et d'associations que nous proposons pour la gestion des objets logiciels. Le second chapitre présente **LOHA**, le langage d'interrogation de notre modèle. Enfin, le troisième chapitre présente l'intégration de la notion d'annotation ainsi que l'extension de la notion d'événement dans notre modèle.

La troisième partie est une concrétisation de notre proposition dans le cadre du système Adèle.



Finalement, nous concluons ce mémoire avec un bilan des thèmes abordés, puis une évaluation de l'ensemble de nos propositions et les perspectives de notre travail.

PARTIE I

La problématique

Chapitre 1

Contexte général et cadre du travail

1 Introduction

Depuis la naissance du génie logiciel, il y a eu des progrès significatifs en ce qui concerne les méthodologies, procédures et outils permettant d'assister les procédés de fabrication des logiciels. Cependant, la maîtrise de ces procédés est encore loin d'être acquise. De nos jours, un projet de fabrication d'un logiciel est rarement réalisé dans les délais et avec l'affectation des ressources prévues. Une grande partie du problème est due à l'immaturité des organisations qui éprouvent de grandes difficultés à définir clairement leurs procédés de fabrication de logiciel et à les mettre en oeuvre de façon systématique avec les outils et le support adéquats [Hum89] [Hum94].

Le coût total des ressources humaines dans la production mondiale de logiciels a été évalué à environ $250 milliards par année [Fug93]. Ce chiffre, et l'importance des systèmes informatiques actuels, justifient largement tous les efforts visant à maîtriser les procédés de fabrication de logiciels.

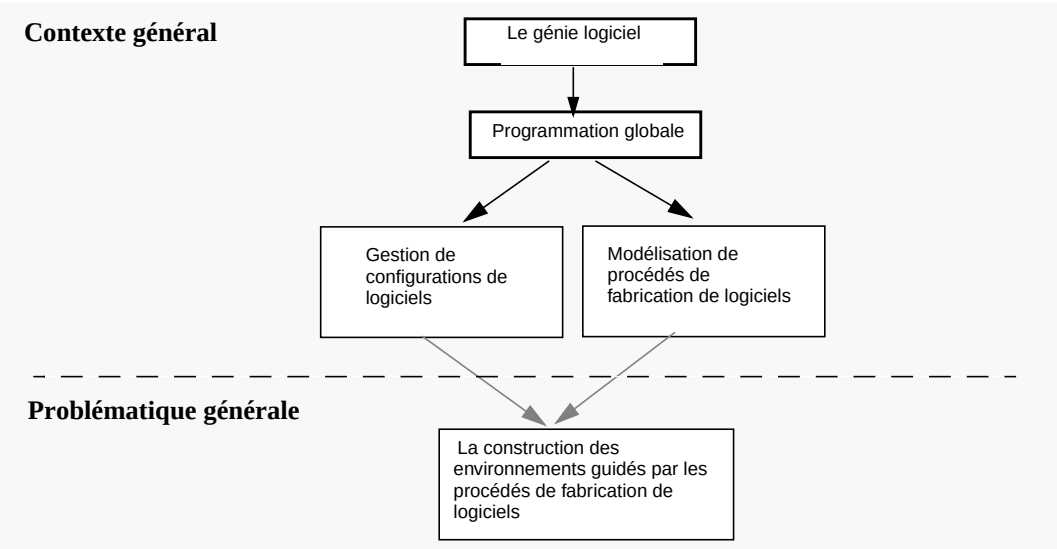


Figure 1 Le contexte et la problématique générale de notre travail.

Dans ce chapitre nous situons notre travail par rapport au domaine du génie logiciel, ainsi que le cadre dans lequel sa réalisation prend place. La figure 1 schématise le contexte et la problématique de notre travail. Dans la suite de ce chapitre, nous présentons ces deux aspects. Nous concluons en décrivant notre problématique spécifique.

2 Le contexte

Les logiciels de grande taille

> *"... many people using many tools to make many versions of many modules for use in many configurations at many sites by many users over many years..." [*ADS+*94]*

Nous nous intéressons dans cette thèse non seulement au produit final, mais au *logiciel* (ou *produit logiciel*) tout au long de sa *fabrication.* Nous utilisons le mot fabrication au lieu de développement, pour faire référence à toute la vie du logiciel : son développement proprement dit, sa maintenance et son évolution. En termes de produit, un logiciel de grande taille est composé de millions de lignes de code. Diverses informations sont manipulées dans le contexte de fabrication de tels logiciels : les informations concernant le produit lui-même et les informations liées au contexte de fabrication (les activités, les contraintes et les politiques imposées par l'organisation)[1].

Caractéristiques du produit

- *Le grand nombre et la diversité des entités* : les entités qui constituent le produit logiciel sont de natures très diverses : programmes source, spécifications, tests, scripts d'installation, documentation d'utilisation, etc. Ces entités sont fortement dépendantes les unes des autres.
- *Les facteurs de variation* : la notion de *variation* est intrinsèque au logiciel de grande taille, par exemple, un logiciel varie parce qu'il doit fonctionner pour différentes organisations, sur différentes plate-formes techniques, il doit, aussi, être adapté à certains problèmes particuliers ou à des contraintes externes spécifiques, etc.
- *L'évolution* : les entités du logiciel évoluent avec le temps, elles sont corrigées, améliorées, complétées.

Caractéristiques du contexte de fabrication

- *Le contexte extérieur :* les procédés de fabrication de logiciels sont en évolution permanente car ils doivent s'adapter au contexte extérieur, c'est-à-dire à des

1. Réification des procédés.

nouvelles contraintes ou à des changements de méthodes ou des priorités de l'entreprise fabricant du logiciel.

- *La nature "intellectuelle" des activités* : la plupart des activités menées pour construire un logiciel (par exemple, spécification d'un module ou codage d'un programme) sont de nature créative et ne sont pas, pour la plupart, automatisables.
- *La durée des activités :* les activités de fabrication de logiciels (par exemple, la compilation d'un système, l'implémentation d'un module, la validation d'une spécification) peuvent durer plusieurs heures, jours et même plusieurs mois.
- *Le travail coopératif :* pour construire un logiciel de grande taille, il faut répartir le travail sur une ou plusieurs équipes de personnes. Des dizaines, voire des centaines de personnes peuvent intervenir tout au long des activités de fabrication d'un logiciel de grande taille. Ces équipes travaillent et collaborent dans le même but. Le contexte extérieur ajoute des contraintes au travail coopératif: il est courant que ces équipes soient situes sur des sites géographiquement différents. Cette distribution peut impliquer des politiques de gestion, des méthodes et des procédures différentes.

La programmation globale

La crise du génie logiciel, constatée é la fin des années 60, a permis d'identifier un certain nombre de problèmes découlant de l'utilisation de méthodes *ad hoc* pour la fabrication de logiciels de grande taille. Les techniques de développement de petits systèmes se sont alors révélées inadéquates pour ce type de logiciels. Le développement de logiciel à cette époque était, en général, beaucoup plus cher en temps et en ressources que prévu, les systèmes ne satisfaisant pas toujours les fonctions spécifiées. De plus, la maintenance de tels logiciels est une tâche ardue et difficile.

En 1976, DeRemer et Kron [DK76] ont montré que la problématique liée au développement d'un logiciel, constitué de plusieurs composants, est différente de celle du développement de chaque composant pris individuellement. Dans ce premier travail sur la programmation globale, le problème d'architecture[2] d'un système logiciel a été identifié comme crucial. C'est ainsi que les langages d'interconnexion de modules sont apparus : NuMil [NS87], Intercol [Tic80], Gandalf [HN86], Inscape [Per87], etc. Les langages MIL permettent de décrire formellement l'architecture globale d'un système logiciel et de mettre en évidence les relations entre modules.

Parallèlement à cette démarche de structuration des logiciels de grande taille, plusieurs travaux se sont intéressés aux problèmes liés à l'évolution d'un logiciel. La gestion de configurations de logiciels est la discipline qui s'occupe du contrôle de l'évolution du logiciel. C'est à cette époque que les premiers outils pour la gestion de configurations de logiciels ont surgi, par exemple SCCS [Roc75] et RCS [Tic82] pour la gestion de versions de fichiers, et Make [Fel79] pour la manufacture[3] du produit final.

2. La décomposition d'un système en sous-systèmes et la définition de leur relations.

Plus récemment, W. Tichy [Tic92] a identifié deux autres domaines liés à la programmation globale : la modélisation de procédés de fabrication et la réutilisation de composants logiciels. Le premier domaine s'intéresse à la compréhension des procédés de fabrication de logiciels de manière à les automatiser et les contrôler efficacement et ainsi, réduire les coûts de fabrication et d'augmenter la qualité du logiciel. De même, la réutilisation cherche à réduire les coûts de fabrication en diminuant le travail de programmation, grâce à l'utilisation de composants de logiciel déjà élaborés et testés.

La gestion de configurations et la modélisation de procédés sont deux domaines directement liées à notre travail. En revanche, le problème de la réutilisation de composants logiciels sort du cadre de cette thèse, et ne sera pas pris en compte dans la suite. On trouvera dans [AF91] une excellente synthèse de ce domaine.

La gestion de configurations de logiciels

> *"...there is a need for a unified CM[4] model... this unified model should be a multi-paradigm model that supports several CM concepts cooperating in harmony. This model can become a framework for adaptation to a range of software processes." P.Feiler [Fei91].*

Une configuration de logiciel est un ensemble cohérent de composants logiciels (spécifications, programmes, documents d'installation et d'utilisation du logiciel,etc) qui constituent un produit logiciel. L'objectif de base de la gestion de configurations de logiciels est de prévenir le chaos du logiciel suite à de nombreuses corrections, adaptations, etc [Tic92]. La gestion de configurations a fait l'objet d'une grande attention ces dernières années, autant du point de vue *méthodologique* dans les entreprises fabriquant des logiciels, que dans la perspective de mettre en place un *support automatique.*

Du point de vue *méthodologique*, la définition des procédures, politiques et méthodes de gestion de configurations de logiciels a été identifié comme l'un des composants essentiels du modèle de maturité définis par le SEI-CMU[5] [Hum89]. Selon ce modèle, la gestion de configurations est l'une des fonctions indispensables pour qu'une organisation puisse sortir du stade d'immaturité, plus précisément, pour quelle puisse change de niveau dans l'échelle défini par le modèle.

En ce qui concerne la mise en place d'un *support automatique*, il existe aujourd'hui plusieurs systèmes commerciaux qui offrent des outils pour la gestion de configurations. Il a été estimé dans [IBW95] que le revenu des outils et des services pour la mise en place de procédures de gestion de configurations de logiciels en 1994 a été de 300 millions de dollars et qu'il sera de l'ordre du milliard de dollars pour l'année 1998.

3. La manufacture est la tâche (automatique) permettant de construire un système à partir de ses fichiers sources [Fav95].
4. Configuration Management
5. Software Engineering Institut, Carnegie Melon University

Cependant, la gestion de configurations continue d'être un domaine de recherche [HHW95]. Il reste encore beaucoup de problèmes à résoudre, en particulier, celui de l'intégration de la gestion de configurations avec les autres domaines du génie logiciel. Dans le chapitre 2, nous traiterons plus en détail quelques aspects concernant la gestion de configurations de logiciels.

La modélisation de procédés de fabrication de logiciel

> *"The effectiveness of a tool in general and a process description language in particular, depends on the context in which it is used, the objectives it is used for and the degree to which its features are understood and used." D. Rombach [Rom91].*

Les procédés de fabrication de logiciel comprennent l'ensemble des activités, aussi bien techniques qu'administratives, qui sont nécessaires à la fabrication d'un logiciel. Ces activités vont de l'analyse des besoins jusqu'à l'évolution ou la maintenance du logiciel, en passant par l'implémentation, la gestion de configurations, le contrôle de qualité, etc.

Wallnau et Feiler [WF91] ont défini deux niveaux de modèles de procédés de logiciel : le modèle de gros grain (le cycle de vie), et celui de grain fin, ou modèle de développement. Le premier donne un cadre général pour la fabrication du logiciel, c'est-à-dire, une macro-vue du produit en construction. Le deuxième est chargé d'organiser les activités de tous les jours, les interactions, le travail coopératif, etc. Les modèles de gestion de configurations sont des exemples de modèles de procédés de développement de grain fin [Fei91].

Même si l'idée de modélisation de procédés date d'une dizaine d'années[6], ce n'est que récemment que de nombreux chercheurs se sont intéressés à ce sujet. Cet intérêt est né du désir d'augmenter la qualité du produit logiciel en améliorant les procédés de fabrication.

Dans l'article intitulé "Les procédés de logiciels sont aussi des logiciels" [Ost87] Osterweil souligne la nécessité de pouvoir exprimer formellement la notion de procédé dans un *langage exécutable*. Il parle alors de *programmation de procédés*. Il y a deux aspects dans cette idée : d'une part, la représentation explicite de procédés et, d'autre part, le fait que cette représentation puisse être décrite avec un formalisme *exécutable* par une machine.

La représentation explicite a l'avantage de servir de cadre global de communication entre les différents agents[7] et des outils impliqués et d'aider à comprendre les procédés employés dans la vie réelle pour fabriquer des logiciels.

L'exécution d'un modèle de procédés doit guider et assister les activités réelles accomplies par les acteurs humains et les outils de l'environnement dans la production

6. Le premier colloque international sur les procédés de logiciel a eu lieu en 1984 [Pot84]
7. Le terme *agent* est utilisé ici pour désigner les différents utilisateurs d'un environnement logiciel : le chef de projet, les développeurs, les utilisateurs chargés de la validation, etc.

du logiciel. Cette exécution est censée organiser l'utilisation des outils, synchroniser le travail des différents agents, faciliter leur collaboration, assurer le partage de l'information, maintenir les relations entre les activités de fabrication et l'évolution du produit logiciel, etc.

Néanmoins, il est illusoire de croire qu'un modèle de procédés puisse être d'emblée défini pour répondre complètement et correctement à toutes les attentes d'une organisation. En effet, nombre de facteurs à prendre en compte dans un tel modèle sont par nature incertains et relèvent d'estimations ou de spéculations sur le déroulement futur de la fabrication du logiciel. Plus une organisation sera capable de prévoir le déroulement réel, moins le modèle de procédés employé aura besoin d'adaptation et d'ajustement et donc, plus le niveau de maturité de l'organisation sera élevé.

Aussi, les organisations fabricant des logiciels doivent *apprendre à estimer l'utilisation de leurs ressources* : comprendre pourquoi les projets logiciels ne satisfont pas les objectifs prévus, pouvoir estimer, par exemple, l'impact de changements sur un logiciel, etc. Pour arriver à ces fins, elles doivent pouvoir profiter au maximum de leurs expériences passées pour réutiliser au maximum la technologie de procédés logiciels et, par là-même, enrichir leur savoir-faire.

L'émergence d'environnements dits guidés par les procédés de fabrication de logiciels est le résultat de recherches visant à concilier les aspects modélisation, exécution et évaluation des procédés. De nombreuses études soulignent que pour obtenir cette conciliation, plusieurs problèmes doivent être surmontés [FKN94] [DF96]. En premier lieu, il est indispensable d'intégrer les *aspects méthodologiques* de développement de modèles de procédés logiciels tenant compte des objectifs de qualité visés par l'organisation. En second lieu, il faut mettre en place, dans l'environnement logiciel, des *mécanismes* capables de :

- saisir et stocker, tout au long de la fabrication du logiciel, des mesures sur le procédé reel, par exemple, la durée des activités, l'utilisation des ressources, le nombre d'erreurs, le nombre de composants, etc.
- observer les états d'exécution des procédés.
- évaluer/comparer la réalisation de procédés avec les modèles de qualité préconisés.
- réagir aux situations anormales détectées lors d'une comparaison entre l'exécution réelle des procédés, les modèles de procédés et les modèles de qualité.

De nombreux travaux portant sur l'intégration de mesures dans les modèles de procédés logiciels sont présentés dans [Is94].

Les environnements logiciels

Un *environnement logiciel* est un ensemble de *services* visant à assister les agents dans leurs activités de fabrication de logiciels. Ces services sont fournis par des outils, un outil pouvant éventuellement accomplir plusieurs services.

Dans les environnements logiciel, le terme *intégration* signifie :

- augmenter l'interoperabilité entre les outils, c'est-à-dire permettre la coopération entre les différents outils pour offrir les services,
- homogénéiser l'utilisation des services offerts par l'environnement,
- guider les actions des utilisateurs en respectant les politiques et les procédures imposées par l'organisation à laquelle ils appartiennent.

L'obtention d'un haut degré d'intégration constitue l'un des objectifs majeurs dans la construction d'un environnement logiciel. L'intégration peut être abordée selon deux points de vue complémentaires : *comment intégrer les outils* et, *pourquoi et quand les agents vont se servir des outils*.

Comment intégrer les outils

Dès le rapport Stoneman [BD80] en 1980, la nécessité de disposer d'une infrastructure permettant d'intégrer les outils d'un environnement logiciel a été établie. Ce rapport décrivait les besoins et l'architecture d'un environnement pour le support du développement de logiciels en langage ADA. L'environnement proposé est composé d'un ensemble d'outils interagissant à travers un gestionnaire d'objets central. Suite à cette idée, divers efforts pour normaliser les gestionnaires d'objets ont été réalisés. Les plus connus sont les normes PCTE [Tho89a], [BMT88] [PCTE93] [PCTE94] et CAIS-A [Obe88].

L'intégration entre les outils est obtenue en utilisant une représentation commune des données. L'environnement doit disposer d'un gestionnaire d'objets qui fournit le modèle permettant de décrire les données et les mécanismes pour les accéder. Même si aujourd'hui cette idée d'un gestionnaire d'objets comme élément *intégrateur* principal de l'environnement a été révisée, le gestionnaire d'objets reste un composant essentiel de l'infrastructure de l'environnement.

Les études sur l'intégration des outils [Was89] [WF91] ont abouti à la définition de trois modèles conceptuels qui décrivent différents *types* d'intégration entre les outils : l'*intégration par la présentation*, l'*intégration par le contrôle* et l'*intégration par les données*.

L'*intégration par la présentation* permet d'avoir un "*look and feel*" commun entre les outils, pour permettre une utilisation plus homogène et efficace de l'environnement.

L'intégration par le contrôle permet aux outils de l'environnement de collaborer pour atteindre un objectif commun. Les mécanismes chargés de supporter l'intégration par le contrôle doivent permettre l'exécution à distance de services offerts par les outils, c'est-à-dire, le partage et l'échange des services entre outils.

L'intégration par les données permet que toute l'information manipulée par l'environnement et plus précisément par les outils de l'environnement, soit gérée comme un tout cohérent.

Les mécanismes pour mettre en place ces modèles doivent être offerts par l'infrastructure de l'environnement.

Quand et pourquoi les agents vont se servir des outils

Les premiers environnements logiciels ont été conçus comme des “boîtes à outils” pour supporter les activités de développement. Le “workbench” de Unix [KM81] est un exemple représentatif de ce type d’environnement. Les outils de l’environnement utilisent le même modèle de données : le modèle hiérarchique de répertoires et de fichiers de Unix. Même si l’intégration entre les outils est de très bas niveau, les conventions simples et uniformes du système d’exploitation facilitent l’utilisation des entités par les différents outils.

Dans cette première génération d’environnements logiciels, l’utilisateur décide quand et pourquoi il se sert des outils. Aujourd’hui, les constructeurs d’environnements logiciels s’accordent à dire que les environnements doivent permettre de définir les conditions d’utilisation des outils (c’est-à-dire le *quand* et le *pourquoi*). Ces conditions dénotent la prise en compte, au niveau de l’environnement, d’informations relatives aux *procédés de fabrication de logiciel*[8] mis en place.

Ce constat est à l’origine d’une nouvelle génération d’environnements logiciels : les *environnements guidés par les procédés de fabrication de logiciels*. Il s’agit d’un environnement dans lequel les agents sont *guidés* dans leurs tâches de fabrication de logiciels en respectant une méthodologie établie par l’organisation pour laquelle ils travaillent. La prise en compte des procédés utilisés par une organisation permet de déterminer le choix des outils à intégrer en fonction du projet visé et d’organiser l’utilisation des services en fonction des différentes activités de fabrication.

Dans ce qui suit, nous présentons le cadre de travail de notre thèse. Les deux chapitres suivants ont pour objectif de dégager les concepts de base pour la construction d’un environnement guidé par les procédés logiciels capable d’intégrer tous les aspects soulignés jusqu’ici.

3 Cadre de travail

Notre travail a été réalisé dans le cadre du projet Adèle. Le projet Adèle a pour objectif la construction d’un environnement guidé par les procédés de fabrication de logiciels intégrant également les aspects de gestion de configurations de logiciels. Pendant la réalisation de cet travail de thèse, l’équipe du projet Adèle a participé au projet PERFECT (**P**rocess **E**nhancement for **R**eduction of software d**EFECT**s) Project-Esprit 9090. PERFECT est un projet ayant pour objectif majeur l’amélioration des logiciels par celle des procédés de fabrication. Dans ce qui suit les deux projets sont présentés brièvement.

8. On parle alors d’*intégration par les procédés*.

Le projet Adèle

L'objectif initial de ce projet était la conception d'un gestionnaire d'objets et de configurations pour les environnements de génie logiciel.

La première version du système Adèle (Adèle 1) [EGK84] [BE86] proposait un modèle de produit logiciel dédié à la gestion de configurations pour les programmes modulaires. Adèle 1 a été défini comme une base de programmes [Est88]. La gestion d'activités n'était pas explicite dans cet outil, les utilisateurs interagissaient via les commandes de gestion d'objets et de construction de configurations.

Le projet Nomade est l'un des premiers travaux dont l'objectif était de prendre en compte la gestion d'activités dans Adèle [Bel88] [BE89]. Ce travail a mis en évidence l'intérêt de l'intégration des concepts provenant du domaine des bases de données actives dans le cadre de la gestion des procédés logiciels. Ces idées ont pris place dans la version commerciale du système Adèle (Adèle 2).

Une description plus détaillée du système Adèle est présentée dans le chapitre 4. Ici, nous voulons faire remarquer deux aspects d'Adèle qui concernent directement notre travail :

- Le modèle de données fourni par Adèle est un modèle à objets étendu par la notion d'association. Il permet de décrire les objets et les associations manipulés lors de la production du logiciel.
- Les règles actives ont été intégrées au modèle de données afin de permettre l'expression dynamique et l'exécution de procédés logiciels. Les règles actives sont utilisées non seulement pour maintenir la cohérence de la base d'objets, mais également pour contrôler la mise en oeuvre des modèles de procédés logiciels. Le mécanisme qui met en oeuvre ce formalisme de règles actives est le mécanisme de déclencheurs (triggers). La sémantique d'une règle active (événement-condition-action) est la suivante : quand l'événement est signalé au système, si la condition est validée alors l'action est déclenchée.

Les expériences avec le système Adèle dans des conditions réelles d'utilisation ont permis de constater d'une part, la puissance du mécanisme de déclencheurs pour exécuter les procédés logiciels et d'autre part, la difficulté du point de vue du concepteur, de définir le modèle de procédés en utilisant le langage de règles.

Plusieurs travaux de recherche ont été menés autour du système Adèle. Les travaux directement liés aux aspects procédés de fabrication de logiciel sont : Adèle-Tempo [BEM94] [BM92] [Mel93] et PERFECT.

Adèle-Tempo est un travail qui porte sur la définition d'un formalisme de modélisation de procédés de plus haut niveau d'abstraction que les règles actives. Tempo offre les concepts de *processus* (activité), *rôle d'objet* et *connexion* pour décrire les modèles de procédés logiciels. Les modèles de procédés sont traduits dans le modèle de base d'Adèle (le modèle de données objets-associations et les déclencheurs) (figure 2).

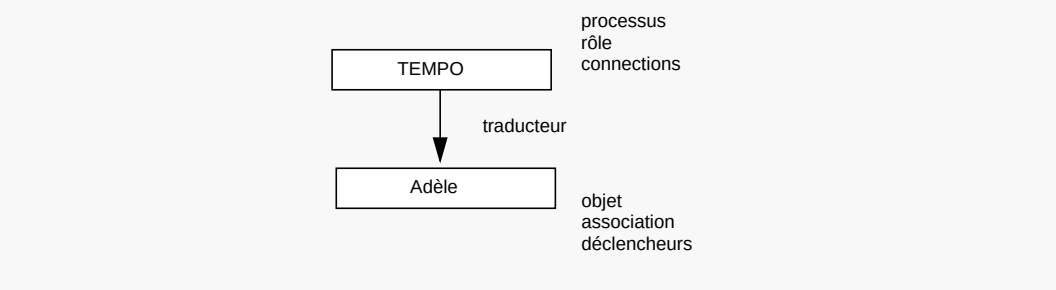


Figure 2 Couplage Adèle-Tempo

Le concepteur du modèle de procédés utilise un langage avec des concepts de haut niveau mais le modèle résultant est ensuite traduit en termes d'éléments de base du système Adèle.

Le projet PERFECT

La motivation principale de ce projet est d'augmenter la qualité des logiciels grâce à l'amélioration systématique de leurs procédés de fabrication. Le projet suit une approche basée sur la quantification d'objectifs de qualité et la réutilisation des modèles de procédés. Le projet cherche à atteindre ses objectifs par :

- la définition d'un langage commun pour représenter le produit logiciel, les procédés, et les mesures dérivées des objectifs de qualité,
- la définition et la mise en place de méthodes d'évaluation de procédés permettant de construire de bases d'expériences de projets logiciels,
- la définition de mécanismes de mesure et d'évaluation pour contrôler et superviser le développement des projets logiciels.

La plate-forme PERFECT est composée de plusieurs outils. Adèle et ProcessWeaver [Fer93] sont deux de ces outils. Adèle se charge de la gestion du produit logiciels et d'une partie de l'exécution des procédés. ProcessWeaver gère en coopération avec Adèle l'autre partie de l'exécution des procédés. Nous verrons plus en détail cette architecture dans le chapitre 3.

Le formalisme intégrant le produit, les procédés et les métriques est en cours de définition [DE95]. Il s'agit d'un langage graphique de haut niveaux d'abstraction appelé APEL (Abstract Process Engine Language). L'objectif est de traduire le modèle de procédés défini avec APEL dans les différents formalismes exécutable de la plate-forme.

4 Conclusion

Les travaux autour des procédés de fabrication de logiciels dans le cadre du projet Adèle et son utilisation au niveau industriel, ont permis d'identifier de nouveaux besoins quant à la définition et la construction d'un environnement guidé par les procédés de fabrication de logiciels.

Ces besoins peuvent être décomposés en deux catégories. La première concerne l'amélioration des capacités de modélisation à offrir aux concepteurs de modèles de procédés. La deuxième constitue l'enrichissement de l'infrastructure du système avec des mécanismes adéquats d'une part, pour gérer toute l'information produite et consommée dans l'environnement et, d'autre part, pour fournir les fonctions de base nécessaires pour observer, mesurer, évaluer et contrôler des procédés. Cette dernière catégorie de besoins est en relation directe avec le type de gestionnaire d'objets que l'environnement guidés par les procédés de fabrication doit utiliser.

Dans la définition de plate-formes pour la construction d'environnements guidés par les procédés, nous nous intéressons plus particulièrement à la définition des fonctions que doit offrir le gestionnaire d'objets de l'environnement. Nous avons cherché à identifier les fonctions de base d'un gestionnaire d'objets selon deux points de vue complémentaires de la fabrication d'un logiciel de grand taille : la gestion de configurations de logiciels et la gestion de procédés de fabrication de logiciels.

Dans le chapitre suivant nous nous intéressons aux caractéristiques du gestionnaire d'objet logiciels requises par la gestion du produit. A travers l'analyse que nous proposons, trois aspects fondamentaux de la gestion du produit sont dégagés : l'évolution, la variation et le travail coopératif. Le gestionnaire d'objets doit être capable de rendre compte de ces trois aspects lorsqu'il s'agit de gérer le produit.

Enfin, dans le chapitre 3, nous analysons l'impact de la gestion explicite des procédés de fabrication sur le gestionnaire d'objets de l'environnement. Notamment, nous soulignons la nécessité de gérer les informations relatives aux états d'exécution des procédés, aux facteurs de qualité et à la prise en compte de mesures concernant aussi bien le logiciel (produit) que les procédés.

Chapitre 2

Le Produit logiciel

Ce chapitre identifie les besoins de la gestion des objets logiciel par rapport aux caractéristiques intrinsèques des logiciels de grande taille. Le chapitre est organisé en deux sections. La première présente la gestion du produit logiciel en faisant un parallèle entre l'espace de stockage et l'espace de travail [EC94]. La seconde présente la base pour la définition d'un modèle de versionement à trois dimensions [EC95].

1 La gestion du produit logiciel

Cette section présente brièvement l'évolution historique des gestionnaires d'objets logiciels et, notamment, l'émergence d'une séparation entre espace de stockage des objets logiciels et espace de travail où les utilisateurs fabriquent les logiciels.

1.1 Espace de stockage et espace de travail

Dans la plupart des applications de CAO (Conception Assistée par Ordinateur), les entités que l'on souhaite fabriquer possèdent une matérialisation dans le monde réel. Dans le monde de l'ordinateur, ces entités ne peuvent pas être directement manipulées par les programmes et il est donc nécessaire d'en donner une représentation abstraite.

En revanche, dans le cas de la fabrication des logiciels, les entités que l'on souhaite fabriquer (les logiciels) appartiennent directement au monde de l'ordinateur. Les programmes d'aide à la fabrication (compilateurs, éditeurs, etc) travaillent directement sur les entités finales (le texte source des programmes) et non pas sur une représentation de ces entités.

La distinction entre espace de stockage et espace de travail s'inspire donc d'une métaphore générale soulignant la distinction entre l'espace dans lequel est conçu un objet (par exemple, une voiture), qui permet de mettre au point et de consulter le plan de construction de l'objet, et l'espace dans lequel est réellement construit l'objet (les différents blocs de l'usine dans lesquels sont fabriquées et assemblées les pièces d'une voiture particulière). Dans le contexte de la fabrication de logiciel, l'espace dédié au

stockage de l'information du produit logiciel est assimilable à l'espace de conception des plans de construction d'une voiture, tandis que l'espace de travail correspond à un bloc de l'usine dans lequel sont fabriquées et assemblées certaines pièces d'une voiture.

Historiquement, la fabrication du logiciel reposait sur une gestion ad hoc des fichiers contenant les programmes source. Dans cette approche, l'espace de stockage et l'espace de travail étaient confondus. Plus exactement, l'espace de stockage était considéré comme l'espace de travail.

Objet logiciel et gestion de fichiers

Les systèmes de gestion de fichiers du système d'exploitation ont constitué les premiers outils utilisés pour décrire l'architecture de logiciels. L'architecture globale du logiciel était alors simplement traduite en une structure arborescente de répertoires et de fichiers ; les répertoires correspondant aux sous-systèmes et les fichiers aux modules (cf. figure 1).

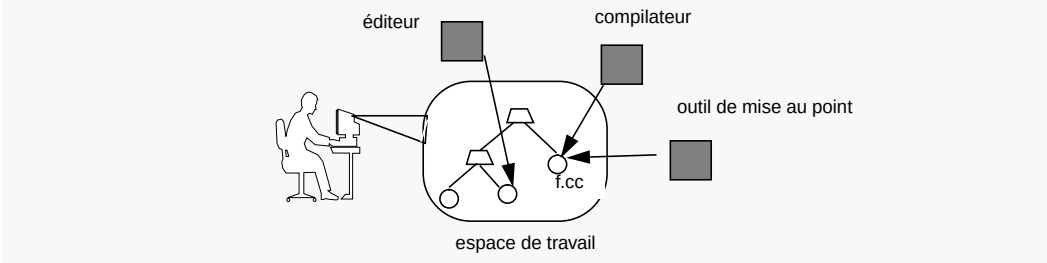


Figure 1 L'architecture d'un logiciel à travers les hiérarchies de répertoires et les fichiers du système d'exploitation.

Toutefois, la taille des logiciels augmentant, le besoin d'organisation a donné lieu à la multiplication et la diversification des informations à gérer. Différents types de fichiers ont été introduits à cet effet : modules, scripts de tests, spécifications formant le produit logiciel, etc.

Pour remédier à cette profusion de fichiers et distinguer les différentes informations, des conventions de nommage des fichiers ont dû être adoptées. Par exemple, le suffixe *doc* permet d'indiquer que les fichiers contiennent des documents de spécifications, le suffixe *h* indique les fichiers contenant les interfaces de modules, etc.

D'autres solutions, moins pragmatiques, ont cherché à augmenter la capacité de modélisation des gestionnaires de fichiers. Par exemple, les gestionnaires de fichiers *attribués* permettent d'associer des propriétés aux fichiers rendant ainsi plus souple l'identification des fichiers [Sec91].

Gestion de versions de fichiers

En plus de la diversité et de la quantité d'informations, il est apparu qu'il fallait aussi gérer les différentes évolutions des logiciels ; c'est-à-dire en gérer les différentes versions [Fav95]. Pour ce faire, des outils pour la gestion de versions de fichiers ont été conçus au dessus du modèle de fichiers du système d'exploitation (SCCS[1] [Roc75],

RCS[2] [Tic82]). A travers l'exploitation de ces outils, la distinction entre espace de travail et espace de stockage est alors devenue plus claire (cf. figure 2).

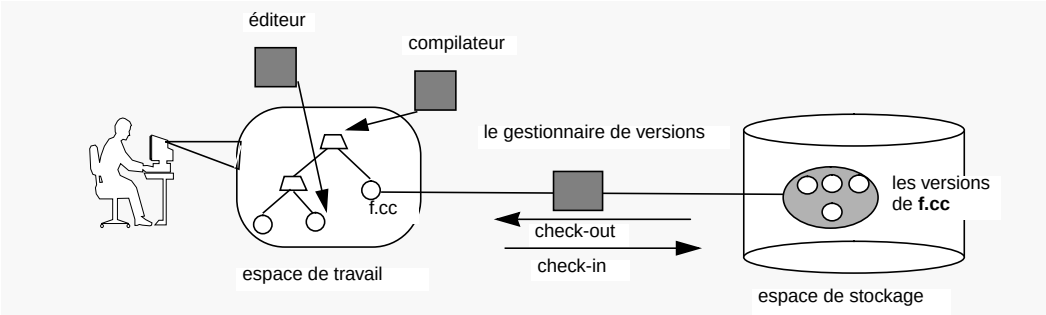


Figure 2 L'espace de travail et l'espace de stockage.

Puisque les outils ne font pas partie intégrante du système de gestion de fichiers, il a donc été nécessaire d'ajouter des commandes spécifiques pour extraire (check-out) les fichiers de l'espace de stockage et les amener vers l'espace de travail de l'utilisateur et, inversement, pour les rendre (check-in) à l'espace de stockage. Ce modèle, appelé check-in/check-out, est à la base de la plupart des modèles de gestion de configurations de logiciels [Fei91] [IBW95].

Dans certains systèmes basés sur le modèle check-in/check-out, comme DSEE [LC85], NSE [Mil89] [Cou89], ClearCase [Leb94], l'espace de stockage a une représentation similaire à l'espace de travail. Par exemple, dans ClearCase, l'utilisateur a directement accès aux objets de l'espace de stockage à travers un système de fichiers virtuel (VFS[3]). Toutes les commandes sur les fichiers sont redirigées vers le VFS. Grâce à un langage de règles de sélection, l'utilisateur définit le contenu de son espace de travail. Au lieu de faire des copies des fichiers, le VFS crée un espace de travail virtuel pour l'utilisateur (cf. figure 3).

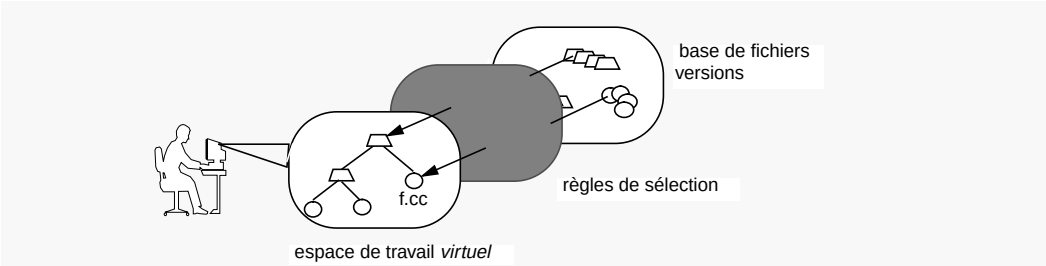


Figure 3 La relation entre l'espace de stockage et l'espace de travail dans le gestionnaire de configurations de logiciel ClearCase.

D'autres approches tendent à rendre plus explicite la distinction entre espace de travail et espace de stockage. Ces approches proposent d'utiliser des modèles de représentation différents pour les deux types d'espaces. Ainsi, à l'instar des espaces de travail virtuels de ClearCase, l'espace de travail est basé sur le principe de structuration du modèle de fichiers. En revanche, à l'instar d'Adèle (cf. §1.2) et à l'inverse de ClearCase cette fois-ci, l'espace de stockage est structuré et organisé grâce à un modèle de données à objets.

1. Source Code Control System
2. Revision Control System
3. Virtual File System

La notion de gestionnaire d'objets logiciels est venue généraliser celle de système de gestion de fichiers. Un gestionnaire d'objets logiciels doit recouvrir la gestion des informations de l'espace de stockage, des espaces de travail et de la correspondance entre les espaces. La partie suivante est consacrée à ce sujet.

1.2 Les gestionnaires d'objets logiciels

Deux approches ont été explorées pour la définition de gestionnaires d'objets logiciels : étendre le gestionnaire d'une base de données existant, ou définir un gestionnaire dédié à un environnement logiciel. Ces deux approches résultent d'un même constat : telles quelles, les techniques de bases de données classiques ne sont pas adaptées aux besoins particuliers des environnements logiciels [Ber87].

Les gestionnaires issus des base de données

Malgré les extensions réalisées aux modèles relationnels, de nombreuses études [Ber87] [Dit89] ont montré leur inadéquation lorsqu'il s'agit de gérer des objets logiciels. En particulier, ces modèles se heurtent aux problèmes de gestion des objets complexes, des objets à grosse granularité (par exemple, un fichier peut être une valeur d'attribut), de l'aspect évolutif des objets et des schémas, du comportement dynamique des objets, etc.

Les systèmes de gestion de bases de données à objet ont fait leur apparition dans les années 80. Plusieurs caractéristiques distinguent les modèles à objets des autres types de modèles de données [ABD+89] [Kim88] et, en particulier :

- la modélisation des objets complexes interdépendants,
- l'organisation en classes au sein d'une hiérarchie d'héritage et, en conséquence, la facilité d'étendre et d'adapter un schéma de données ainsi représenté,
- l'identification des objets indépendamment de leur contenu,
- l'encapsulation à l'intérieur des classes des opérations agissant sur les objets.

Aujourd'hui, plusieurs systèmes de bases de donnés à objets commerciaux existent, par exemple, O2 [LPV89], Gemstone [BOS91], Itasca [Ita93] (la version commerciale du système Orion [Kim88]), ObjectStore [Obj94], etc.

Si le paradigme à objets procure à ces systèmes de sérieux atouts, ils ne répondent pas encore complètement aux besoins de la gestion d'objets logiciels. Certains d'entre eux ont subi des extensions et des adaptations en ce sens. Ils prennent maintenant en compte une notion de *version* et mettent en place des mécanismes pour supporter des activités de longue durée, pour leur donner un comportement réactif.

Une extension du système de gestion de bases de données à objets ObjectStore a été utilisée comme gestionnaire d'objets de l'environnement SEAMEN [TGD95]. De même, une extension du système de gestion de bases de données à objets O2 a été

utilisée comme gestionnaire d'objets de deux environnement guidés par les procédés : Merlin [PW93] [BESS94] et GoodStep [Tea94]. L'environnement Goodstep est présenté plus en détail dans le chapitre suivant.

Les gestionnaires d'objets issues du génie logiciel

Les environnements logiciels intègrent souvent leur propre gestionnaire d'objets, spécialement conçu pour répondre à leurs besoins spécifiques. Par exemple, le système Adèle, initialement conçu pour la gestion de configurations, et le système Marvel, présenté comme un système expert pour le génie logiciel.

Le gestionnaire du système Marvel [KBS90] [KBS93] propose un modèle de données à objets qui permet la représentation d'objets composites et de relations arbitraires entre objets. Les fichiers (programmes sources, les documents, etc.) sont représentés au niveau de Marvel dans des objets du modèle, mais sont en réalité stockés dans des fichiers Unix.

Le système Adèle offre un modèle objet-association (cf. chapitre 4). La composition, la dépendance, la dérivation et d'autres associations peuvent être définies pour modéliser des relations entre objets. Une caractéristique importante du gestionnaire d'objets d'Adèle est l'ensemble de services offerts pour gérer la correspondance entre les objets de l'espace de stockage et les fichiers gérés dans les espaces de travail par l'utilisateur et les outils (cf. figure 4) [EC94].

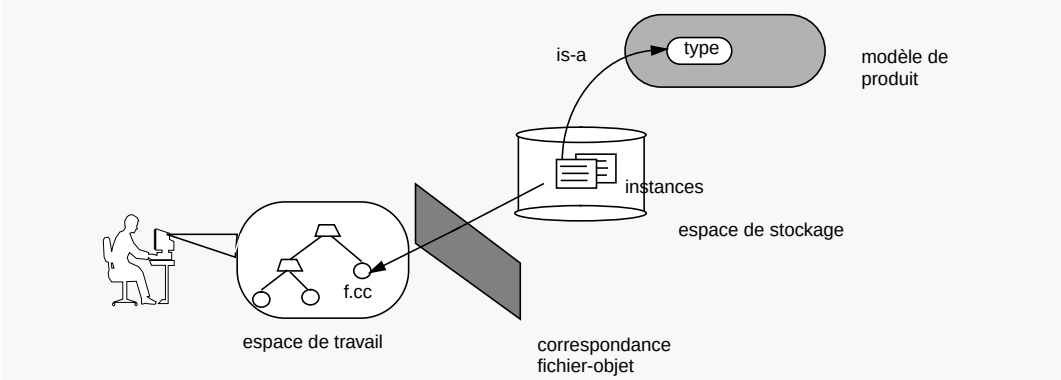


Figure 4 La relation entre l'espace de stockage et l'espace de travail dans le gestionnaire de configurations de logiciels Adèle.

Par ailleurs, d'autres travaux issus du génie logiciel ont abouti à la définition de normes pour la construction de gestionnaires d'objets. Le projet PCTE (Portable Common Tool Environment) [Tho89a], [BMT88] [PCTE93] [PCTE94] était un projet européen dont le but consistait en la définition des normes de construction des environnements logiciels. L'hypothèse de départ de ce projet était que la meilleure façon d'augmenter l'interopérabilité entre les outils était d'utiliser un modèle de données commun et un ensemble de services pour les gérer. Un des composants importants de la norme PCTE, est le gestionnaire d'objets OMS (Object Management System).

Ce gestionnaire d'objets est basé sur un modèle de données à objets. La représentation des données est fondée sur le concept d'objet. Un objet peut avoir des

attributs dont les types sont adaptés au domaine du génie logiciel. Notamment, un objet peut recevoir comme valeur un *fichier* (contenu non interprétable). Les fichiers sont gérés dans l'espace de stockage défini par l'implémentation PCTE, d'où, la nécessité d'encapsuler les outils pour manipuler les fichiers. Des liens (orientés) peuvent être établis pour représenter les relations de composition entre les objets. Les objets, les liens et les attributs sont typés.

Plusieurs environnements ont été bâtis en utilisant les normes proposées par PCTE, comme par exemple l'environnement PACT [Sim89], [Tho89b] qui fournit des services complémentaires tels que la gestion de configurations, ALF [OZG91] ou Scale [Oqu95] qui inclut d'autres modèles pour représenter les procédés logiciels.

Discussion

Aujourd'hui, il est accepté que le gestionnaire d'objets d'un environnement logiciels doit être basé sur le paradigme *à objet*. Les avantages de ces modèles on été largement énumérées et analysées dans [Ber87] [BESS94].

Cependant, dans le cadre du génie logiciel pour les logiciels de grande taille, il reste encore beaucoup de questions sans réponse comme, notamment, la gestion de versions des composants logiciels. Dans les sections suivantes nous étudions la problématique de la gestion de versions de composantes logiciels. Cette fonction est une fonction de base dans les gestionnaire d'objets des environnement logiciel.

2 Vers un modèle de versions à trois dimensions

Dans un premier temps, nous présentons l'approche usuelle de la gestion de versions de fichiers. Dans un second temps, nous proposons un modèle de versionnement basé sur une séparation claire des problèmes qu'il est censé résoudre. Puis, nous présentons des modèles de versionnement issus d'autres domaines dont les principes s'apparentent à nos propositions et qui constituent des sources d'inspiration pour nos proposition de solution au problème du versionnement.

2.1 Niveaux de versionnement

La plupart des modèles de produit logiciel proposés dans les gestionnaires de configurations de logiciels sont très proches du modèle hiérarchique de répertoires et de fichiers du système d'exploitation. Dans ce contexte, le problème du versionnement est traité par rapport à l'évolution des fichiers texte. L'unité de versionnement est le fichier

et, les versions d'un fichier sont grosso modo un ensemble de fichiers qui possèdent plusieurs lignes en commun.

2.1.1 Le mécanisme de base

La gestion de versions de fichiers proposée par les outils SCCS [Roc75] et étendue par RCS [Tic82], est basée sur la gestion d'un *graphe de versions (ou de dérivation)* représentant l'évolution d'une entité. L'évolution séquentielle de l'entité donne lieu à des *révisions*, une séquence de révisions étant appelée une *branche* (cf. figure 5).

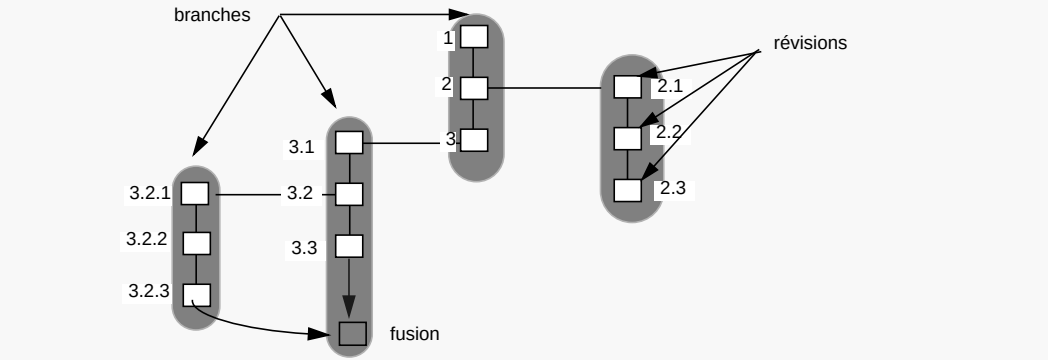


Figure 5 Le graphe de versions d'une entité.

Chaque révision représente une amélioration, une correction ou un ajout à une entité. Les révisions sont immuables ; ce qui veut dire que toute modification se traduit par la création d'une nouvelle révision. Entre deux révisions successives d'une même branche, seules les différences sont stockées. Cette technique, dite des *deltas*, permet d'économiser énormément d'espace disque.

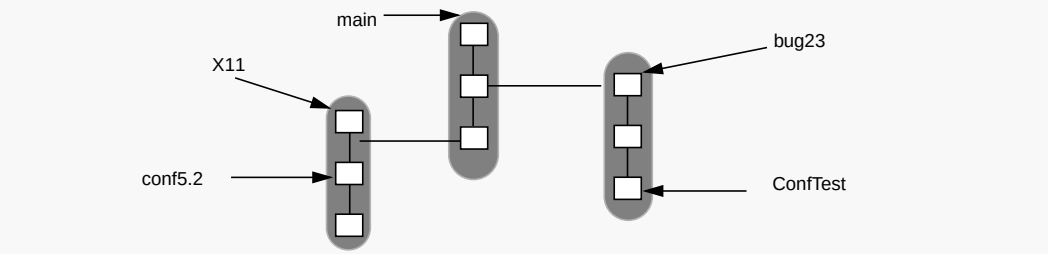


Figure 6 Identificateurs externes aux branches et aux révisions. Les noms sont donnés par les utilisateurs. Le système de nommage correspond à des conventions d'utilisation.

L'identification des révisions dans les outils de base est faite par un numéro de séquence. Quelques gestionnaires de configurations de logiciel offrent la possibilité de désigner par un identificateur externe, fixé par l'utilisateur, les révisions importantes comme, par exemple, les révisions qui font partie d'une configuration ou qui contiennent la correction d'une erreur (dans la figure 6, la branche *bug23*, la branche de la variante X11, la révision qui appartient à la configuration 5.2, etc.)

2.1.2 L'historique

Dans la plupart des gestionnaires de configuration, seul un nombre limité d'opérations réalisées dans le système peuvent bénéficier d'un historique. Généralement, cet historique est lié au remplacement d'un fichier (opération check-in) dans la base. Lorsqu'une nouvelle révision est créée, le système saisit une trace censée identifier la raison de la modification (cf. figure 7). De plus, l'information à annoter sur l'objet qu'on remplace dans la base est pré-définie et se résume à la donnée de l'auteur, la date de l'opération et, occasionnellement, un commentaire décrivant le pourquoi de l'operation.

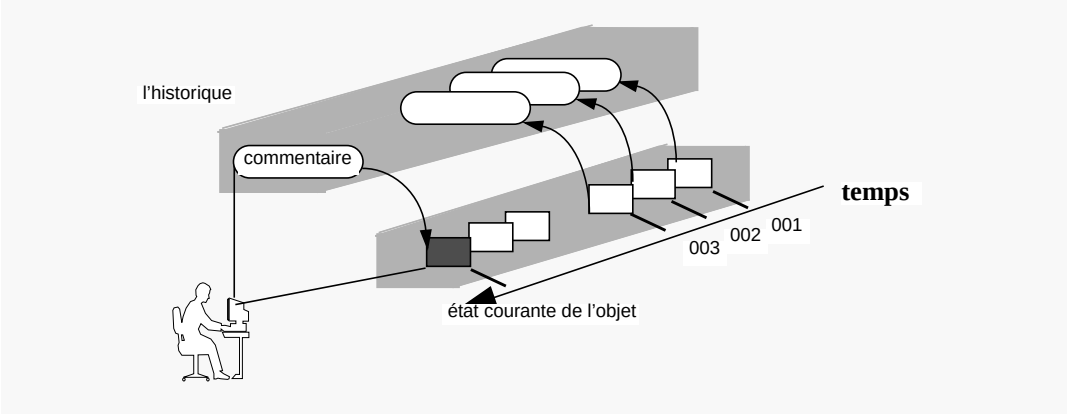


Figure 7 L'information historique dans un gestionnaire de configurations de logiciel.

2.1.3 Discussion

Le mécanisme de base pour la gestion de versions est un mécanisme de bas niveau qui permet de gérer des arbres de dérivation. Cependant, ces mécanismes n'offrent aucune sémantique de versionnement. L'opération de création d'une version est explicitement réalisée par l'utilisateur. La création d'une nouvelle *branche*, qui va évoluer indépendamment de la révision d'origine, peut obéir à différents propos :

- représentation d'une alternative ou variante de l'entité, par exemple, pour prendre en compte une implémentation différente d'un composant : technique d'optimisation différente, algorithmique différente, langue différentes (cas de la documentation), etc.
- développement expérimental correspondant à un essai de variante qui pourra être abandonné après.
- développement parallèle pour gérer le fait que deux utilisateurs puissent réaliser des changements simultanément sur le même composant. Dans ce cas, les branches existent temporairement et elles seront fusionnées une fois que les modifications seront finies (cf. figure 6).

Les interprétations ne sont que des conventions d'utilisation, généralement inconnues des outils de versionnement. L'avantage d'un tel mécanisme de gestion de versions est qu'il peut être utilisé pour traiter les problèmes à la fois d'évolution, de variation de logiciel et de travail coopératif. L'inconvénient de ce type d'approche est

que la sémantique des versions est laissée à l'utilisateur et ne peut pas être exploitée par le système.

Selon Edward Sciore [Sci91], la difficulté de rapprochement et d'uniformisation des différents travaux relatifs à la gestion de versions (génie logiciel, CAO et bases de données temporelles) provient d'une confusion entre différents niveaux de versionnement :

- Le niveau physique qui concerne la façon de stocker les versions des objets. Ce niveau s'appuie sur le mécanisme de base décrit précédemment.
- Le niveau conceptuel qui concerne la modélisation des versions d'objets, indépendamment de la façon de les stockées. A ce niveau, on s'interesse au fait qu'un objet possède des versions sans se préocuper de la structure interne gérée par le mécanisme de base.
- Le niveau logique qui concerne la sémantique de l'ensemble de versions d'un objet.

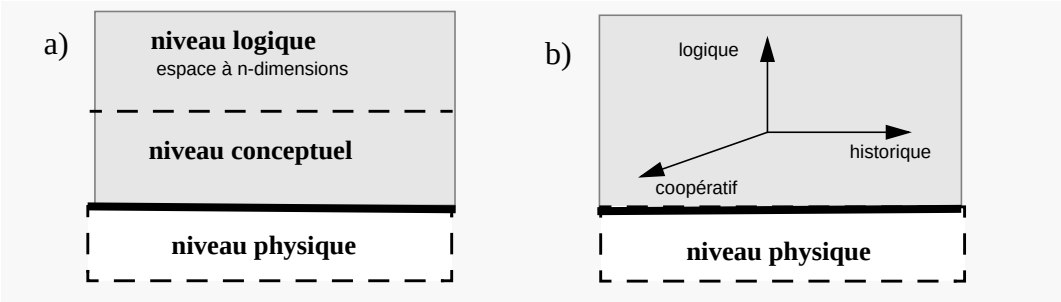


Figure 8 Niveau de versionnement et dimensions de versionnement. La partie a) schématise la proposition de Sciore [Sci91], tandis que la partie b) présente notre proposition [EC95].

Nous adhérons au principe visant à distinguer différents niveaux de versionnement. Cependant, nous proposons de séparer la problématique de l'évolution d'objets (*versions historiques*) de celle des variations du logiciels (*versions logiques)* et du travail coopératif (*versions coopératives).* Pour ce faire, nous proposons un modèle de versionnement à trois dimensions (figure 8) dont chaque dimension correspond à une problématique particulière. Chacune d'elles possède ses propres concepts et nécessite des mécanismes de gestion spécifiques et adaptés.

Dans notre modèle, un objet est référencé sans ambiguïté par rapport au moment où on veut le regarder (le temps), par rapport à la variante et par rapport à l'activité à laquelle il participe. Dans ce qui suit, nous présentons les bases du modèle de versionnement à trois dimensions.

2.2 Trois problèmes orthogonaux

2.2.1 La dimension historique

Dans le contexte du génie logiciel, il faut concilier deux besoins contradictoires. D'une part, il est nécessaire de permettre l'évolution des entités car il faut les corriger, les améliorer, les compléter. D'autre part, il est nécessaire de les rendre immuables car il faut pouvoir revenir en arrière pour reconstruire un logiciel tel qu'il était à un moment donné. Les objets logiciels doivent donc pouvoir évoluer mais cette évolution doit être contrôlée et tracée.

La sémantique du versionnement historique est définie en termes temporels, c'est-à-dire qu'elle cherche à décrire l'évolution de l'objet dans le temps linéaire. Au niveau conceptuel, le gestionnaire d'objets doit permettre de modéliser quels sont les propriétés *immuables* des objets, c'est-à-dire celles dont on veut conserver un historique.

Dans cette thèse nous nous intéressons plus particulierment à la dimension historique.

2.2.2 La dimension logique

La notion de *variation* est intrinsèque aux logiciels de grande taille. Un logiciel *varie* car les contextes d'utilisation et les besoins d'utilisation varient aussi. Ainsi, une *variante* du logiciel fait référence à l'utilisation d'un logiciel dans un contexte spécifique.

Les facteurs de variations [Fav95] peuvent faire référence aux préférences des utilisateurs du logiciel, aux objectifs de l'organisation dans laquelle le logiciel va être installé (fonctions spécifiques, adaptations, etc.), aux objectifs ou à des activités spécifiques des fabricants (logiciels de démonstration, des essais de nouvelles techniques, logiciel de validation, en cours de développement, etc.). Ils peuvent aussi être liés aux aspects techniques comme les plate-formes logicielle et matérielle sur lesquelles le logiciel est développé et sur lesquelles le logiciel va être exécuté chez les clients (le système d'exploitation, les langages, les systèmes de fenêtrage, etc).

La gestion de *variantes*[4] du logiciels est assez souvent un problème traité au niveau de la gestion de versions. Pour la plupart des outils de gestion de versions de logiciels une *variante* correspond à un développement *en parallèle*, c'est-à-dire à une branche dans l'arbre de versions.

Dans notre modèle de versionnement, la dimension logique fait référence à la représentation des variantes logicielles. Le gestionnaire d'objets doit permettre de représenter les facteurs de variations des composants logiciels indépendamment de la

4. "Variant management is one of the more obscure issues in software configuration management. (...) It is often misconstrued as simply another version control problem". A. Mahler [Mah94]

représentation physique choisie pour gérer les variantes de logiciels. Par ailleurs, le gestionnaire d'objets doit offrir des fonctions telles que la sélection d'une variante particulière, la construction de configurations, l'élimination d'une variante, la propagation d'une modification à travers les variantes, etc.

2.2.3 La dimension coopérative

Pour augmenter la productivité des équipes intervennant dans la fabrication de logiciel de grande taille, il faut permettre et assister le travail coopératif.

Le travail coopératif fait partie de la problématique de domaines tels que la Conception Assisté par Ordinateur (CAO) [Elm93], la CSCW (Computer Supported Cooperative Work) [Ell92], l'Intelligence Artificielle Distribué (IAD)[DHB93]. La plupart des solutions proposées dans ces domaines sont basées sur des modèles transactionnels sophistiqués.

Une caractéristique commune à ces domaines et à celui de la gestion de configurations réside dans le fait que les activités peuvent durer plusieurs jours, ou semaines et que plusieurs agents peuvent avoir besoin des mêmes ressources pour réaliser leurs tâches. Dans ce contexte, le gestionnaire d'objets doit offrir des services de support aux *transactions longues*. La visibilité des *résultats intermédiaires* est l'un des principes fondamentaux de la gestion de transactions longues [BKK85]. En effet, pour garantir la collaboration des agents, il faut que les activités puissent s'échanger des résultats intermédiaires.

Dans un tel contexte, les propriétés que l'on associe habituellement aux transactions courtes (Atomicité, Cohérence, Isolation et Durabilité) ne sont pas forcément souhaitables [ABAK94]. Par exemple, le critère d'atomicité ne convient pas toujours puisqu'il peut y avoir des opérations qui ne pourront pas être défaites même si la transaction a échouée. Dans un environnement logiciel, il y a une grande variété d'outils que le système de transactions ne peut pas contrôler comme, par exemple, l'envoi d'un e-mail au cours d'une transaction. Dans ce type de transaction on ne cherche pas à défaire mais à offrir des actions de compensation lorsque cela est possible.

La plupart des solutions offertes par les gestionnaires de configurations de logiciels sont réalisées en s'appuyant sur l'utilisation du mécanisme de versions. Pour éviter des conflit entre plusieurs utilisateurs qui partagent les mêmes données, le système crée des versions (copies) de l'entité à partager.

Dans le modèle de base check-in/check-out, quand un utilisateur extrait pour modification un fichier de l'espace de stockage, celui-ci est verrouillé dans la base et ne pourra pas être accédé par un autre utilisateur tant que le verrou est présent.

Avec le système RCS, plusieurs utilisateurs peuvent extraire le même objet. Chacun d'eux possédera sa propre version de l'objet. Au moment de retourner les versions à la base, il faut réaliser leur fusion si on veut garder les différentes modifications apportées par chaque utilisateur. Evidemment, ce mécanisme de fusion ne peut pas toujours

décider sans ambiguïté quels morceaux des versions prendre pour reconstituer l'objet. L'utilisateur doit donc intervenir pour résoudre les éventuels conflits.

Les versions coopératives font référence à des copies d'objets. Leur gestion constitue le mécanisme de base des protocoles de contrôle de concurrence. La création de ce type de *version* doit être complètement transparente pour l'utilisateur. Au niveau conceptuel, on doit définir les protocoles de partage d'information et de synchronisation du travail en termes d'*espace de travail* et non en termes de versions.

2.3 Autres solutions à la gestion de versions

Dans cette section, nous introduisons des solutions au problème de gestion de versions. En premier lieu, nous présentons les versions dans le modèle à objet. La modélisation des versions d'instance dans les systèmes de bases de données à objet n'est pas normalisée. Les différences ne portent pas seulement sur la forme (le langage de description) mais aussi, et surtout, sur la sémantique qu'on accorde aux versions. En second lieu, nous présentons l'approche des bases de données temporelles.

2.3.1 Les versions dans le modèle à objets dans le domaine de la CAO

Dans le domaine de la CAO, le problème du versionnement a fait l'objet de nombreux travaux et essais de normalisation [Kat90]. La plupart de ces travaux ont été réalisés sur des gestionnaires, issus des bases de données, qui reposent sur des modèles de données entité-association ou des modèles à objets.

Sans perte de généralité, il est possible de résumer les caractéristiques générales du versionnement dans les modèle à objets, de la façon suivante :

- La propriété de pouvoir être *versionné* est une propriété de la classe à laquelle appartient l'objet.
- Un objet versionnable possède une partie *invariable*, instanciable une seule fois, et une partie *variable*, qui peut être instanciée plusieurs fois.
  La partie invariable est appelée *objet générique* et la partie variable, correspondant aux *versions de l'objet*, est appelée *groupe ou famille* de versions.
- L'identificateur de l'objet générique est une *référence générique*, tandis que la référence à une version particulière est une *référence spécifique*.

Les différences entre les divers systèmes sont liées principalement à la sémantique donnée aux versions, l'identification des versions, et la composition d'objets. Ces points sont traités par la suite.

La sémantique

Dans plusieurs modèles utilisés dans des systèmes d'aide à la conception, les attributs sont classés selon le rôle qu'ils jouent dans l'objet de conception. La création d'une nouvelle version est liée à la mise à jour de certains attributs. Ces attributs sont appelés significatifs [AN91], ou sensitifs [TOC93].

Il y a d'autres systèmes dont la sémantique du versionnement est définie par rapport au travail coopératif. C'est le cas des systèmes Orion [KGW91] et IRIS [BM88] dans lesquels le versionnement est lié au fait qu'on puisse créer une copie de l'objet pour travailler avec lui dans une base privée. Dans ces systèmes, l'utilisateur décide quand et à partir de quelle version il veut en dériver une nouvelle. Il dispose d'opérateurs pour changer le statut des versions. Une version d'un objet peut avoir le statut de:

- version provisoire (transient version) : elle représente une version temporaire qui peut être mise à jour ou effacée à n'importe quel moment.
- version de travail (working version) : elle représente une version dans un état stable et elle peut être partagée. Cette notion de stabilité est un point de vue de l'utilisateur, c'est lui qui décide de transformer la version provisoire en stable car il considère qu'elle a un degré de robustesse suffisant. Les versions de travail ne peuvent pas être modifiées mais elles peuvent être effacées.
- version définitive (released version) : il s'agit d'une versionde travail qui ne peut pas être effacée.

L'identification d'une version

L'identification d'une version se fait de manière générale par le couple <référence générique, référence specifique>. Comme dans les gestionnaires de versions du génie logiciel *(Cf. §2.1.2),* l'identificateur de la version est une fonction de sa position dans l'arbre de dérivation.

Dans [Sci91] [Sci93], Edward Sciore propose un langage de manipulation permettant d'accéder aux versions d'un objet vérifiant certains prédicats de sélection. La référence générique représentant l'ensemble des versions, les prédicats sont appliqués sur cet ensemble :

$$O(a_1 \text{ op } v_1 \text{ and } a_2 \text{ op } v_2 \text{ ..... })$$

Le résultat d'une telle expression est constitué par l'ensemble des versions (les références spécifiques) qui vérifient le prédicat. Si cette expression est affecté à un objet, le contenu de l'objet est évalué dynamiquement à chaque fois que l'ensemble des versions de O est modifié.

Les objets composites et les versions

Un objet composite possède des attributs ayant pour valeur des références à d'autres objets. Le versionnement des objets composites pose le problème de la définition de la relation entre les versions de l'objet composite et les versions de ses composants.

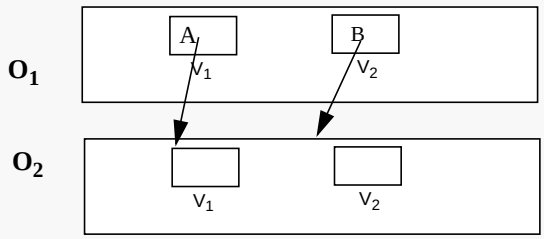


Figure 9 Relations entre les versions de l'objet composite O_1 et les versions d'un de ces composants O_2.

Si la valeur de l'attribut composant est une référence spécifique, c'est-à-dire une référence à une version particulière, les deux objets (composite et composant) sont liés statiquement (*statically bound)*. Si, au contraire, il s'agit d'une référence *générique*, ils sont liés dynamiquement (*dinamically bound)*. Dans la figure 9, pour la version V_1 de l'objet O_1, la valeur de l'attribut A est une référence *statique* à la version V_1 de l'objet O_2. L'attribut B de la version V_2 de l'objet O_1, la valeur de l'attribut est une référence *générique* à l'objet O_2.

La propagation des modifications est l'une des préoccupations majeures des systèmes proposant des mécanismes de versionnement des objets composites. Les versions d'objets composites peuvent être créées de façon explicite ou implicite [TOC93]. Dans le premier cas, les versions de composites sont créées par l'utilisateur. Dans le second cas, la création automatique d'une version d'un objet composite est une conséquence de la modification d'un de ses composants.

2.3.2 Les versions dans les modèles temporel

Les bases de données conventionnelles représentent l'état des données à un instant du temps. Quand les données sont modifiées, les changements sont perçus comme des modifications de l'état de la base, les nouvelles valeurs remplacent les anciennes.

Au cours des 15 dernières années, la gestion de l'évolution temporelle des informations est devenue un thème de recherche important [TCG+93] [FS95] [BJS95]. Cet intérêt pour la représentation et la manipulation explicite de la notion de *temps* dans les systèmes de base de données s'explique par les besoins d'applications telles que le contrôle d'inventaires, les informations médicales, l'information des entreprise, etc.

La plupart de ces propositions sont basées sur des modèles de données relationnels[5] [BJS95] [OS95]. Ces modèles n'étant pas adaptés au génie logiciel, cette partie se propose de présenter les concepts de base de la représentation temporelle dans les modèles à objets uniquement. Un état de l'art plus complet peut être trouvé dans [Sno95]. Dans ce qui suit, nous faisons une brève description des aspects à prendre en compte lors de la définition d'un modèle temporel.

5. Dans l'étude réalisé par Ozsoyoglu et Snodgrass [OS95] plus de douze modèles temporels relationnels avec un ou plusieurs langages de requêtes associés ont été étudiés.

Représentation du temps

Une *séquence de temps* est une série de points de temps t_1, t_2, ... , t_{n-1}, t_n, tels que $t_1 < t_2 < ... < t_{n-1} < t_n$. La distance entre deux points adjacents est constante et est appelée l'*unité de temps*. L'unité de temps dépend de la granularité qu'on souhaite modéliser, et peut correspondre à une seconde, une heure, une semaine, etc.

Un *instant* est un point particulier de la droite du temps. Un *intervalle* de temps est un segment de la droite du temps, délimité par deux instants précis. La *durée* d'un intervalle est le nombre d'unités de temps qu'il comporte.

Lorsque le temps est modélisé de façon discrète, il est isomorphe aux entiers. Un entier correspond à une unité de temps non décomposable et d'une durée de temps arbitraire. Cette unité est appelée un *chronon* et correspond à la plus petite unité de temps qui peut être représentée dans le modèle [JSS93].

Les dimensions du temps

Beaucoup de systèmes de bases de données temporelles prennent en compte deux dimension de temps : *le temps de validité* (*valid time*) et *le temps de transaction* (*transaction time)* (Voir par example, [JSS93], [CI94]).

Dans le premier cas, le temps considéré se réfère aux instants réels auxquels surviennent des événement du monde extérieur. Cette dimension de temps peut être aussi utilisée pour dénoter le temps auquel un événement sera vrai dans le futur [Sno95]. Dans le second cas, le temps considéré dénote les instants auxquels les événements du monde extérieur sont enregistrés dans le système.

Ces deux dimensions sont orthogonales et peuvent être supportées séparément ou de façon intégrée. Les bases de données intégrant le temps de validité sont appelées les bases de données *historiques*[6], tandis que celles intégrant le temps de transaction sont appelées les bases de données de *reprise*[7]. Les bases intégrant les deux dimensions sont, quant à elles appelées les bases de données *bi-temporelles*.

L'intérêt d'une telle distinction réside dans la possibilité de pouvoir modéliser des traitements de type rétroactif . Par exemple, dans une base de données dédiée à la gestion du personnel, l'enregistrement de l'augmentation de salaire d'un employé deux mois après sa date d'effet demande un traitement rétroactif nécessitant les connaissances à la fois de la date d'enregistrement et de la date d'effet de l'augmentation. Une étude détaillée de la relation entre ces deux dimensions peut être trouvée dans [JS92].

Les types de données temporelles

Un *instant* est un point particulier de la droite du temps. Un *intervalle* de temps est un segment de la droite du temps, délimité par deux instants précis. Un *élément temporel* est une union finie d'intervalles de temps [GV85] [Gad88]. Cette notion permet d'introduire des discontinuités dans la validité des faits temporels.

6. Historical database, valid-time database.
7. rollback database, transaction-time database

Le choix d'un type de données temporelles dépend du type de l'application. Dans [GN93] on peut trouver une comparaison entre les intervalles et les éléments temporels. Une des conclusions de cette comparaison est qu'au niveau conceptuel, l'utilisation des éléments temporels rend plus facile l'utilisation d'un système temporel (notamment, en ce qui concerne l'expression de requêtes temporelles) mais, qu'au niveau de l'implantation, leur gestion est difficile à mettre en place.

Dans [WD93], un modèle à objets (OODAPLEX) permet la définition de types temporels en fonction des besoins des applications visées. Ces types sont définis en tant que sous-classes de la classe pré-définie *point*. Un objet de cette classe est uniquement muni des opérateur = et < définissant la relation d'ordre. La spécialisation de la classe *point* permet d'ajouter des opérations spécifiques (fonction de distance, période de vie, etc).

Objets ou attributs ?

Dans une base de données temporelles, les valeurs sont fonction du temps. Dans les modèles à objets, le changement de valeur par rapport au temps peut être enregistrée au niveau des attributs des objets, attribut par attribut, ou au niveau de l'objet comme un tout[8].

La taxonomie présentée dans [EKF93] présente sur l'axe horizontal les principales techniques de représentation du temps dans le modèle : versions des attributs (*attribut versionning*) ou versions des objets (*object versionning*). Quant à l'axe vertical, il présente la façon dans le modèle de représenter les relations entre objets : par des relations explicites (les relations sont des objets de première classe) ou par des liens référence.

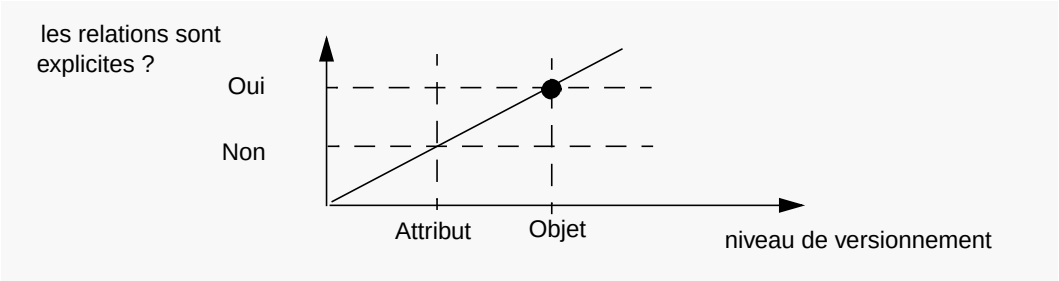


Figure 10 La grille définie dans [EKF93].

Dans le cas du versionnement des attributs [GV85] [EW90], chaque attribut a pour valeur une liste de paires <*valeur, temps*> ou *temps*[9] indique le temps de validité de cette *valeur* pour l'attribut. Le principe d'*homogénéité* temporelle impose que tous les attributs d'un objet aient la même période de vie. Cette propriété garantit l'absence de valeurs *nulles*, ou indéfinies. Par exemple, si l'attribut A_1 à l'instant *t* possède une valeur, alors tous les autres attributs de l'objet doivent posséder une valeur au même instant.

Dans le cas du versionnement globale des objets (tuples dans le cas du modèle relationel [Sno93], ou objet dans le modèle à objets [AN91] [EKF93]), chaque objet

8. Dans les modèles relationnels, le même principe est appliqué au niveau des champs (ou attributs) et des n-uplets.
9. *temps* peut être défini en termes d'instants, intervalles, éléments temporels ou autres (défini par l'utilisateur comme c'est le cas dans OODAPLEX [WD93]).

possède des attributs supplémentaires correspondant aux temps (par exemple, deux instant de temps t_1 et t_2 définissant l'intervalle)

3 Conclusion

Ce chapitre a introduit la problématique de la gestion du produit logiciels de grande taille. Notre démarche a consisté à étudier les problèmes d'évolution et de variation des logiciels, ainsi que les problèmes liés au travail coopératif. Même si ces trois problèmes sont orthogonaux, fréquemment ils sont résolus en utilisant les mêmes mécanismes de base.Il y a une confusion entre les niveaux physique et logique du versionnement, ce qui rend difficile l'utilisation d'autres techniques issues de domaines différents au génie logiciel. De plus, les gestionnaires d'objets existants n'offrent pas aux utilisateurs les concepts adéquats à chacune des problématiques.

Nous avons défini trois dimensions du versionnement : historique, logique et coopératif. Ceci signifie qu'on attend du gestionnaire d'objets des fonctions spécifiques à la gestion de chaque dimension du versionnement.

Dans cette thèse nous nous intéressons plus particulierment à la dimension historique. Notre proposition pour résoudre le problème de gestion de l'évolution des objets logiciel est inspirée des modèles issues des bases de données temporelles. Ces modèles incluent explicitement la notion de temps. La valeur d'une entité est une fonction du temps. Un *état* d'un objet est sa valeur à un moment donné dans le temps. La séquence des états d'un objet représente l'évolution de l'objet dans le temps et nous l'appelons un *objet historique.*

Le chapitre suivant va reprendre certains aspects traités ici mais du point de vue de la modélisation et l'automatisation des procédés de fabrication du logiciel.

Chapitre 3

Les procédés de fabrication de logiciels

Ce chapitre n'a pas pour prétention de fournir un état de l'art complet sur les procédés de fabrication de logiciels mais de montrer la complexité et la profusion de concepts qui découlent de ce domaine. Plus précisément, nous analysons l'impact de la représentation et de la gestion explicite des procédés de fabrication de logiciels sur le gestionnaire d'objets d'un environnement logiciel. Notre démarche consiste à dégager les concepts et les éléments de base des procédés de fabrication de logiciels. Nous introduisons d'abord la modélisation de procédés de fabrication de logiciels en montrant les différents aspects qui doivent être considérés dans la définition d'un modèle de procédés de fabrication de logiciel. Ensuite, la problématique de l'exécution et de l'évaluation des procédés logiciels est abordée. Des exemples d'environnements guidés par les procédés illustrent nos propos.

1 La modélisation de procédés

De nombreux travaux récents [FH93] [CFFS92] [DNR91] [Lon93] se sont intéressés à la définition des concepts et des objectifs de la modélisation de procédés logiciels. Un modèle des procédés de fabrication de logiciel est une abstraction de l'ensemble des activités de fabrication d'un logiciel. Malgré l'absence d'un consensus global, notamment en ce qui concerne la relation entre les objectifs de la modélisation et les propriétés des formalismes, les différents travaux s'accordent sur plusieurs objectifs :

- *faciliter la compréhension des procédés* : leur représentation explicite doit aider à mieux comprendre les procédés employés dans la vie réelle pour fabriquer des logiciels.
- *servir de cadre global de communication entre les différents agents impliqués* : la représentation explicite des procédés doit aider à améliorer la communication, l'estimation de ressources et la planification du projet en général.
- *guider les procédés* : l'exécution d'un modèle de procédés doit guider la réalisation[1] du projet, c'est-à-dire, les activités réelles accomplies par les acteurs humains et les outils de l'environnement.

1. Les termes utilisés pour *exécution* et *réalisation* dans la littérature anglophone sont respectivement *enaction* et *performance* [FH93] [DNR91] [Lon93].

- *évaluer les procédés* : Le modèle de procédés doit décrire formellement les interactions entre l'exécution du modèle et la réalisation réelle du projet pour d'une part, pouvoir réagir à de possibles divergences et d'autre part, recueillir des informations quantitatives et qualitatives sur le déroulement des procédés.

Dans ce qui suit, nous présentons d'abord les modèles de cycle de vie, car ce sont les premiers travaux à s'être intéressés aux procédés de fabrication de logiciel. Ensuite, nous précisons les différents éléments qui doivent pouvoir être représentés dans un modèle de procédés de fabrication de logiciel.

1.1 Des modèles de cycle de vie de logiciel aux formalismes de modélisation de procédés

Les modèles de cycle de vie ont comme objectif primordial de structurer et de guider les procédés de fabrication de logiciel. De nombreux schémas de cycle de vie de logiciel, ainsi que diverses méthodes ont été développés pour répondre aux problèmes identifiés lors de la crise du logiciel des années 70.

Parmi les nombreux modèles proposés, le modèle en cascade est le plus connu. Il a été présenté dans [Roy70] et a fait l'objet de nombreux raffinements et variations par la suite. L'idée de base est de structurer les activités de développement en plusieurs étapes (Figure 1) ainsi que de décrire le flot de données entre chaque étape. La notion de *retour en arrière* a été introduite pour prendre en compte le problème d'itération entre les étapes.

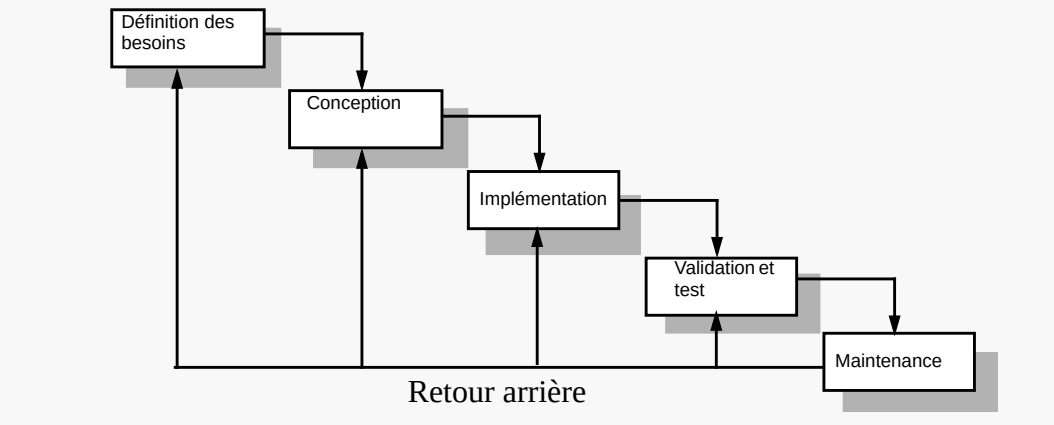


Figure 1 Le cycle de vie en cascade.

Ce modèle a été largement critiqué dans [Gla82] et [MJ82]. La critique principale réside dans le fait que les activités identifiées ne correspondent pas aux activités réelles de la construction du logiciel. En effet, dans la réalité, les différentes étapes se superposent et les itérations entre elles ne sont pas clairement définies. Pour remédier à cet inconvénient, des travaux postérieurs ont introduit : l'idée de prototypage pour mieux évaluer l'état d'avancement des étapes et aussi, l'idée d'identifier une situation de retour en arrière lorsque des problèmes sont détectés.

Le modèle en spirale [Boe88] a été introduit pour résoudre le problème des itérations entre les étapes du cycle de vie. Le processus global de construction est également représenté par une séquence d'étapes mais les itérations sont conditionnées par une analyse des risques. Les activités sont organisées comme une spirale avec plusieurs cycles. La dimension radiale de la spirale représente le coût du système et la dimension angulaire le progrès du projet. A chaque étape du développement, le modèle exige qu'une évaluation de risques soit réalisée. Cette évaluation doit être exprimée en termes des problèmes rencontrés et des stratégies permettant de les résoudre.

Les modèles de cycle de vie ont permis de mieux comprendre l'importance de la description des procédés de fabrication de logiciel, en identifiant les étapes et l'ordre global d'exécution. Cette compréhension devait conduire à une amélioration de la qualité des produits, ainsi qu'à l'application effective de méthodes et à l'utilisation d'outils dans les différentes activités.

Cependant, dans la pratique, les résultats n'ont pas été à la hauteur des attentes, pour différentes raisons :

- *le niveau de description :* la plupart de ces modèles restent à un niveau de description trop général et trop vague, ne tenant pas compte :
    - des conditions d'enchaînement des activités : quand une activité se termine-t-elle et sous quelles conditions doit être commencée l'activité suivante ?
    - des rapports entre l'état du produit et les différentes activités (comment l'information doit-elle être transmise ?).
    - des données qui sont exploitées et produites par l'activité
- *le caractère non déterministe des activités :* les modèles proposent un ordre prédéfini pour l'enchaînement des activités qui ne correspond pas toujours à la réalité.
- *l'absence du rôle des personnes :* le rôle joué par les personnes dans les activités n'est pas représenté, la problématique liée au travail coopératif est alors complètement occultée.
- *la non intégration de méthodes et l'utilisation d'outils disparates dans chaque activité :* les modèles ne considèrent pas l'intégration des différentes méthodes et outils utilisés.

Pour remédier aux insuffisances des modèles de cycles de vie de logiciels, des formalismes de modélisation de procédés ont vu le jour. Dans ce qui suit, nous présentons brièvement les éléments à représenter dans un modèle de procédés de fabrication de logiciels.

1.2 Les modèles des procédés

"... the idea of one, standard, all-encompassing Process Modelling Language is utopia, because of theoretical problems and interoperability requirements". R. Conradi.
[Cea95]

La figure 2 montre, de manière très générale, les différents éléments qui doivent être pris en compte dans la modélisation de procédés.

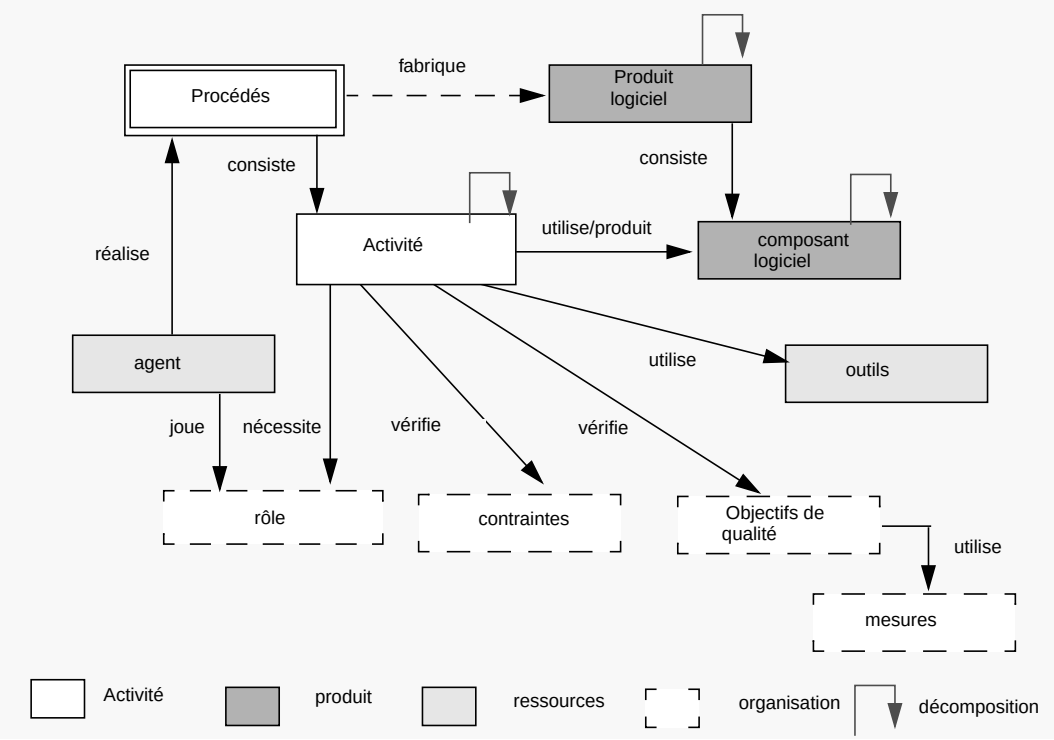


Figure 2 Les éléments de base de la modélisation de procédés. Inspiré de [DF96].

La figure 2 peut être lue de la façon suivante. Les *procédés* fabriquent le *produit logiciel*. Un *procédés* corresponde à l'ensemble d'activités aussi bien technique qu'administratives pour produire un logiciel. Un *procédé* est réalisé par des *agents* et consiste en un ensemble d'*activités*. Pour être exécutée, une *activité* nécessite des *agents* jouant des *rôles* précis. Une *activité* utilise et produit des *composants logiciels*, ainsi que des services offerts par les *outils*. Une *activité* doit vérifier des *contraintes* et des *objectifs de qualité* établis par l'organisation. Les *objectifs de qualité* doivent servir à dériver les *mesures* qui permettront de vérifier si les objectif ont été atteints.

A ces éléments, il faut ajouter l'aspect dynamique des activités, c'est-à-dire l'*enchaînement des activités*, ainsi que l'aspect *coopératif ;* notamment, la problématique de la gestion du produit logiciel introduite dans le chapitre précédent.

1.3 Les formalismes de modélisation de procédés

Le modèle de procédés doit inclure différents sous-modèles ou fragments de procédés pour prendre en compte les divers aspects. Etant donnée la complexité de la description d'un procédé de fabrication de logiciel, il n'est donc pas surprenant que les travaux cherchent à utiliser différents paradigmes pour décrire l'exécution des activités et proposent aussi différents langages de modélisation (à des niveaux d'abstraction différents) avant d'obtenir le modèle de procédés exécutable.

Un formalisme de modélisation de procédés doit permettre de décrire l'ensemble des phases des procédés (le modèle global ou modèle horizontal), ainsi que tous les niveaux de décomposition (les différents modèles de développement de grain plus fin, ou modèles verticaux) [WF91].

Un formalisme de modélisation *exécutable* se caractérise par le paradigme de programmation utilisé pour décrire et exécuter ces activités : procédural (APPL/A [Tit et al88], CML [SV91]), logique (Peace [Arb93]), règles de production (Marvel [KBS90], [KBS93] et Merlin [JPSW94]), règles actives (Adèle [Cas94], Naos [CCS94]), réseaux de petri (ProcessWeaver [Fer93], Spade [BBFL93]), etc.

Les formalismes à base de règles, soit de production (condition-action ; CA), soit actives (événement-condition-action ; ECA), permettent de résoudre les aspects de "non déterminisme" mais rendent, en revanche, difficile l'obtention d'une vue globale des procédés. En effet, la description des activités et du contrôle est éparpillée à travers diverses règles.

Les formalismes procéduraux permettent de décrire aisément le schéma normal d'exécution des activités. Leur caractère impératif constitue, en revanche, leur principal inconvénient car ces formalismes ne tiennent pas compte de l'aspect dynamique et évolutif des procédés.

Les travaux récents préconisent une approche multi-paradigme pour la représentation du modèle du produit et le modèle de procédés mais aussi, à l'intérieur du modèle de procédés, pour le contrôle d'enchaînement des activités, le déclenchement des actions correctives, etc [HKD92] [Arb93] [CLJ91]. Pour ce qui est de l'enchaînement des activités, il est en effet souhaitable de disposer, d'une part, d'un formalisme (par example, procédural) pour modéliser globalement la décomposition des activités et, d'autre part, d'un formalisme à base de règles pour détailler le contrôle du déroulement des procédés en termes de réactions (par example, a des situations d'anomalie).

Kellner, dans [Kel91], considère que c'est une contradiction d'exiger qu'un formalisme soit à la fois *compréhensible* et *exécutable* (c'est-à-dire ayant une syntaxe et une sémantique claire et précise pouvant être exécutée). Les travaux convergent vers l'idée que le langage externe, qui est offert au concepteur du modèle, ne doit pas nécessairement être celui qui sera exécuté.

Ainsi, par exemple, le formalisme de modélisation de procédés du système Adèle est basé sur des règles actives. Les projets Tempo et APEL (cf. chapitre 1 §2) cherchent à offrir au concepteur du modèle de procédés un langage de modélisation de plus haut niveau d'abstraction, qui est ensuite *traduit*[2] en termes des règles actives Adèle. Le

concepteur travaille donc avec un langage possédant des concepts plus proches de la réalité (activité, rôle, outils, etc). Puis, les modèles décrits avec ce langage sont transformés en concepts de base du formalisme exécutable (cf. chapitre 1 §3).

Dans ce qui suit, nous présentons brièvement les environnements guidés par les procédés de fabrication de logiciel.

2 Les environnements guidés par les procédés

De nos jours, il n'y a de consensus ni sur l'architecture ni sur les composants qui doivent constituer une plate-forme pour la construction d'environnements guidés par les procédés[3]. Cette section essaie de dégager les éléments principaux des EGPFL ainsi que, les problèmes principaux auxquels se heurtent les constructeurs de plates-formes d'EGPFL.

En premier lieu, nous présentons trois exemples de plate-formes qui sont à l'état de l'art de cette technologie. Ensuite, nous faisons une synthèse dans le but de dégager les composants principaux d'une plate-forme d'EGPFL. Finalement, nous analysons l'impact de la gestion explicite des procédés de fabrication de logiciels sur le gestionnaire d'objets de l'environnement.

2.1 Présentation des exemples

Nous avons choisi de présenter les plate-formes *GoodStep*, *Provence*, et *Weadèle* car ils sont représentatifs des recherches actuelles sur les procédés de fabrication de logiciels. Ces trois exemples partagent le même but : offrir une infrastructure sur laquelle on puisse bâtir des environnements de fabrication de logiciel guidés par les procédés. Il faut remarquer que des plate-formes telles que Arcadia, Epos, Peace, Oikos, Alf, Scale, Peace, Merlin, entre autres, mériteraient tout autant d'être présentées comme exemple. Une étude approfondie de ces plate-formes peut être trouvée dans [FKN94] [DF96] [Oqu95].

GoodStep

GoodStep (Esprit-III No. 6115) est un projet de recherche de la communauté européenne, dont le but est de développer un système de base de données orienté-objet

2. Cette traduction n'est pas automatique.
3. Un environnement proprement dit correspond à une instanciation de la plate-forme. C'est-à-dire que la plate-forme est configurée pour répondre à un modèle de procédés spécifique.

(SGBDOO) sur lequel on puisse bâtir des environnement logiciels et construire des outils.

Le système SPADE[4] [BBFL93] [BFGL94] est un résultat du projet GoodStep. SPADE est une application bâtie au dessus du système de gestion de bases de données à objets O2 [LPV89]. SPADE fournit des services pour la définition, l'analyse, l'exécution et l'évolution des procédés de fabrication de logiciel.

Dans SPADE, le langage de modélisation de procédés (appelé SLANG) est un langage basé sur le formalisme des réseaux de Pétri. Les transitions représentent les événements qui peuvent survenir dans l'environnement et le déclenchement d'une transition correspond à l'occurrence d'un événement. Les conditions qu'une occurrence d'événement doit vérifier pour déclencher une transition sont définies dans les places du réseau de Pétri. La topologie du réseau définit l'ordre de précédence entre les événements, ainsi que le parallélisme.

Le SGBDOO O2 gère toute l'information de l'environnement, le produit et les procédés. Le produit logiciel est représenté en utilisant le modèle à objet fourni par O2. L'information du produit est accédée via les outils préalablement intégrés à l'environnement. Le modèle de procédés (i.e., les réseaux de Pétri) et l'état d'exécution des procédés (i.e., les placements des jetons dans le réseaux) sont aussi conservés et gérés par O2.

Provence

Provence est une plate-forme pour construire des environnements guidés par les procédés de logiciels [BK95] [Bar94]. Le principe le plus important de l'architecture du système *Provence* est la séparation logique et physique entre l'exécution et la réalisation des procédés. Cette séparation apparaît principalement au niveau de la gestion de données de l'environnement.

La plate-forme est constituée de trois composants principaux : le système d'exécution des procédés, le système de supervision et de contrôle de la réalisation des procédés et le système *traducteur* qui fait la liaison entre les deux premiers (cf. figure 3).

4. SPADE (Software Process Analysis and Enactment).

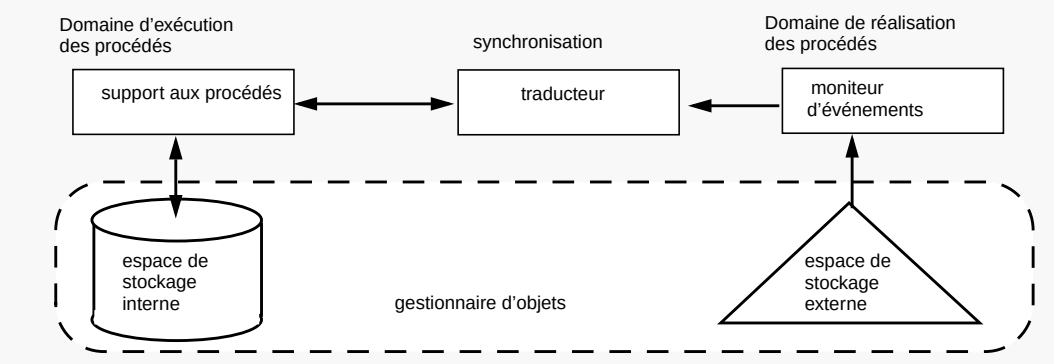


Figure 3 Architecture de *Provence.*

Provence a été construit en utilisant le système Marvel [BK90] [BSK94] comme support des procédés et de la gestion d'objets. Le système Yeast (event-action engine) [RK95] est le système chargé de superviser (monitoring) la réalisation des activités entreprises dans les espaces de travail des utilisateurs.

Dans *Provence* les modèles de procédés sont exprimés sous la forme de règles de production du système Marvel. Marvel exécute ces règles grâce à un moteur d'inférences (chaînage arrière et chaînage avant). Cependant, la partie des procédés correspondant à la supervision (monitoring) des activités en cours de réalisation est définie en utilisant le langage de spécification de "événement-actions" du système Yeast.

Les actions réalisées dans les espaces de travail sont annoncées à Yeast, soit manuellement (la terminaison d'une réunion, l'approbation d'une spécification...), soit via N-DFS [FKR94] (invocation d'un outil, modification d'un fichier...). Le système N-DFS est un moniteur du système d'exploitation. Après avoir interprété les événements, Yeast les notifie au traducteur.

Le gestionnaire d'objets de *Provence* est dit *ouvert* car il sépare l'espace de stockage en plusieurs espaces. Notamment, il distingue l'espace de stockage consacré à l'*exécution* du modèle de procédés (appelé espace de stockage interne) de celui dans lequel a effectivement lieu la *réalisation* des procédés (appelé espace de stockage externe). Le premier stocke l'information décrivant le modèle et son état d'exécution (produit, procédés...), tandis que le deuxième se charge du stockage des données, notamment celles du produit exploitées par les activités en cours ou qui ont déjà eu lieu. Une partie de la gestion d'objets est réalisée dans le système de support d'exécution des procédés, tandis que l'autre l'est en dehors de cette architecture, dans des espaces de stockage externes.

Weadèle

Weadèle est un résultat du projet Perfect [Dam95] (cf. chapitre 1 §2). Il s'agit d'une plate-forme pour la modélisation et l'automatisation des procédés. Cette plate-forme est constituée de deux outils principaux : ProcessWeaver [Fer93] et Adèle [EC94]. ProcessWeaver joue le rôle de machine d'exécution de procédés de haut niveau (le modèle global) et Adèle est à la fois le gestionnaire d'objets et une machine d'exécution de procédés de grain plus fin.

Dans *Weadèle*, le modèle de procédés est exprimé en utilisant deux formalismes : les réseaux de Pétri fournis par ProcessWeaver et les règles actives d'Adèle. A travers le formalisme fourni par ProcessWeaver, le modèle global de procédés peut être exprimé en termes de décomposition des activités, d'enchaînement des activités et de définition de chaque activité [Fer93]. L'enchaînement des activités est réalisé en utilisant des réseaux de transitions basés sur des réseaux de Pétri. La définition d'une activité est très incomplete car le système ne gère pas directement le produit logiciel. Ainsi, les ressources d'une activité correspondent aux noms de documents. Quant aux règles actives d'Adèle, elles sont utilisées pour modéliser la partie des procédés directement liées à la gestion du produit et des espaces de travail des utilisateurs.

Dans *Weadèle*, le modèle de produit est défini en utilisant le modèle de données fourni par Adèle. La gestion du produit est réalisée complètement par Adèle. ProcessWeaver ne possède aucune connaissance sur le produit.

Adèle et ProcessWeaver collaborent pour fournir l'ensemble des fonctions de l'environnement. Nous soulignons le mot collaboration pour faire remarquer qu'il ne s'agit pas d'une relation maître-esclave ou procédés-produit. Dans cette architecture, Adèle n'est pas utilisé comme l'espace de stockage passif mais comme une partie de la machine d'exécution de procédés.

2.2 Synthèse

Même si les plate-formes que nous avons brièvement décrites ont le même but, leurs architectures sont très différentes. Récemment, il y a eu un colloque pour comparer différentes architectures et pour essayer d'arriver a un consensus sur les services de base que doit offrir l'infrastructure d'un environnement [Ban95]. Ces services sont : l'exécution du modèle de procédés, les mécanismes d'intégration d'outils, la gestion d'espaces de travail, et la gestion d'objets. La figure 4 montre schématiquement comment ces services prennent place dans l'environnement.

Le modèle de procédés est exécuté à l'intérieur de l'environnement, grâce au service d'exécution des procédés[5] fourni par l'infrastructure. Cette exécution est censée guider la réalisation des tâches des utilisateurs dans leurs espaces de travail. Dans le cas de *GoodStep*, l'exécuteur est le programme qui interprète le langage SLANG. Dans *Provence*, l'exécuteur est un des composants du système Marvel (le moteur d'inférence qui exécute les règles). Dans *Weadèle*, l'exécuteur est d'une part l'interpréteur des réseaux de Petri de ProcessWeaver et d'autre part, l'exécuteur de règles dans Adèle.

5. *process engine (PE)* dans la littérature anglophone.

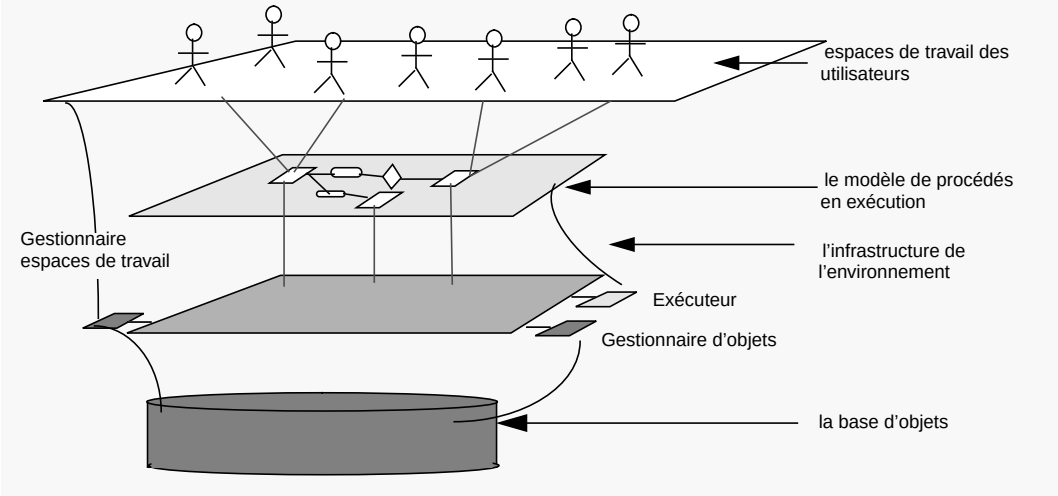


Figure 4 Les services minimal de l'infrastructure d'un environnement guidé par les procédés.

Quant au gestionnaire d'objets il a la charge de gérer toute l'information de l'environnement. Dans *GoodStep,* le SGBDOO O2 gère toute l'information de l'environnement. Dans *Provence*, une partie est réalisée par le système Marvel (les modèles, l'états d'exécution de procédés) et une autre partie est gérée par gestionnaires d'objets externes comme le système de gestion de fichiers N-DFS. Dans *Weadèle*, la gestion de l'information est aussi répartie entre les outils mais seul Adèle gére le produit.

2.3 L'impact des procédés sur le gestionnaire d'objets

L'impact de la gestion explicite des procédés de fabrication sur le gestionnaire d'objets a été un des sujets de discussion du récent workshop sur l'intersection entre les bases de données et le génie logiciel [Kin94]. La conclusion de cette discussion a été que le gestionnaire d'objets devait offrir la possibilité de :

- détecter des *événements* pour y réagir ou, plus généralement, pour permettre d'évaluer l'état du système et,
- gérer les états d'exécution des activités.

Bien qu'elle n'émane pas directement de cette conclusion, on peut aussi ajouter à cette liste la possibilité pour le gestionnaire d'objets de prendre des mesures sur le produit en construction ainsi que sur l'exécution des activités.

Pour justifier ces besoins, il faut rappeler ici l'objectif primordial des organisations fabricant des logiciels qui a été présenté dans l'introduction de cette thèse. Leur objectif est d'arriver à produire de façon systématique et prévisible des logiciels de bonne qualité. Or, il est désormais admis que la qualité des logiciels peut être améliorée si les procédés de fabrication sont eux-mêmes améliorés. C'est pourquoi les fabricants du logiciel commencent aussi à admettre que l'évaluation, et donc la *mesure*, est une tâche indispensable pour obtenir une amélioration systématique des procédés de fabrication [LHR95].

A cause du caractère imprévisible des activités logicielles (longue durée, contexte coopératif, non automatisables), la réalisation de procédés par les agents humains peut sensiblement, voire radicalement, s'écarter de celle prévue par le modèle des procédés. Dans ce cas, la réalisation des procédés et l'exécution du modèle de procédés divergent.

Il est clair que le modèle de procédés peut devenir complètement obsolète si :

- l'environnement est incapable de fournir l'information requise pour évaluer l'état de réalisation des procédés vis-à-vis du modèle et, si
- le modèle ne peut être réadapté à la nouvelle situation.

Il est donc nécessaire que l'environnement puisse contrôler, superviser, évaluer l'exécution d'un modèle pour, le cas échéant, faire appel à des mécanismes capables de réadapter le modèle. Dans le cadre de notre travail, nous ne nous préoccupons pas de la description et de la mise en œuvre des mécanismes d'évolution dynamique, ou de réadaptation, du modèle de procédés. De récents travaux sur ce sujet peuvent être consultés dans [CFF94] [ARCW95]. En revanche, nous nous intéressons à la problématique liée aux tâches d'*observation*, d'*évaluation* et de *supervision* de l'exécution du modèle qui sont à l'origine de toute opération de réadaptation du modèle.

Pour pouvoir mettre en place ces tâches il faut que le gestionnaire d'objets offre les moyens pour représenter et gérer toute l'information historique liée d'une part, à l'évolution des entités logiciels et d'autre part, aux traces de l'exécution des activités. Dans la figure 5, l'activité *A* a démarrée à l'instant t_0 et elle s'est terminée à l'instant t_1. Le temps de réalisation de l'activité a pu durer plusieurs heures ou jours.

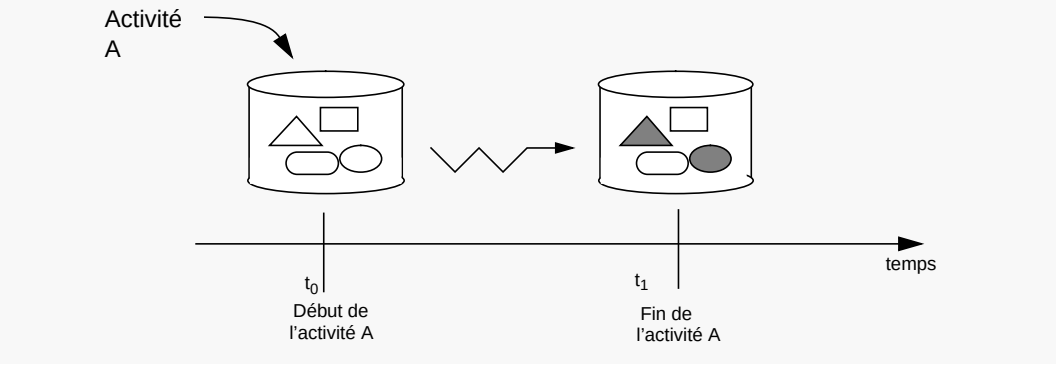


Figure 5 La trace de l'exécution des activités

A la fin de l'exécution de l'activité A, nous voulons disposer de : la trace des modifications des objets (c'est-à-dire l'évolution des objets), la trace de l'activité et, les traces des situations intermédiaires importantes pour l'évaluation de l'activité. Du point de vue de l'exécution des procédés, il est essentiel d'identifier les situations intermediaires car elles peuvent avoir déclenché d'autres activités. Du point de vue de la qualité et du contrôle, il faut pouvoir tracer certaines informations dont le comportement temporel fournit des informations sur un procédé en cours d'exécution : son bon déroulement, la vérification des contraintes de qualité, de temps, de synchronisation, etc.

3 Conclusion

La modélisation de procédés de fabrication de logiciels est une discipline en plein essor. La problématique autour de ce sujet est vaste et complexe. Dans le cadre de cette thèse, nous nous sommes intéressés uniquement à un sous-ensemble de cette problématique. En effet, notre but est d'identifier l'impact sur le gestionnaire d'objet que peut avoir la représentation explicite de procédés dans l'environnement logiciel.

Un EGPFL doit mettre en place des tâches d'observation, d'évaluation et de contrôle de procédés. L'idée de base est d'offrir dans le gestionnaire d'objets la possibilité de représenter et gérer l'historique de l'évolution des objets mais aussi les traces de l'exécution des activités.

Concrètement nous proposons un modèle à objets historiques et associations (MOHA) (cf. chapitre 4). Le modèle est proposé pour gérer l'évolution des objets dans le temps. Pour exploiter ce modèle, nous proposons un langage navigationnel adapté aux besoins d'accès d'un EGPFL (cf. chapitre 5).

Le modèle est étendu avec la notion d'*annotation* (cf. chapitre 6 §1) qui répond au besoin d'introduire des informations complémentaires, dénotant l'exploitation des entités pour des intérêts différents de ceux de la fabrication proprement dite du logiciel. La notion d'événement est inclue dans le modèle pour l'identification des situation complexes impliquant des information passées et présentes de l'environnement (cf. chapitre 7 §2).

PARTIE II

Les solutions

Chapitre 4

Le modèle MOHA

Ce chapitre présente le modèle à objet historique et association (**MOHA**), que nous proposons pour la représentation et la gestion des objets logiciels. Dans la section §1 nous présentons les concepts de base. La section §2 présente la notion d'objet historique. La section §3 présente les associations non-historiques et les associations historiques. Dans la section §4, nous concluons avec une comparaison avec d'autres travaux.

1 Concepts de base

Cette section décrit les concepts sur lesquels nous nous sommes appuyés pour définir le modèle **MOHA**. Ces concepts sont, à quelques détails près, ceux du modèle de données du gestionnaire d'objets du système Adèle [EC94].

Objets et identité

Les objets représentent des entités concrètes ou abstraites du monde réel (un module, un document, une activité, un utilisateur, etc.). Un objet possède une identité unique (oid physique donné par le système). L'identité de l'objet permet de référencer l'objet et elle est indépendante de sa valeur. La valeur d'un objet correspond à l'ensemble des valeurs de ses attributs.

Types d'objet

Un objet est une *instance* d'un type. La structure et le comportement des objets sont définis par leur type. La structure correspond à l'ensemble des attributs qui modélisent les propriétés d'un objet. Le comportement est décrit grâce à des méthodes et à des règles actives (cf. chapitre 6 §1).

Dans la figure 1, nous montrons un objet dont l'identité physique est oid_1. Cet objet est instance du type *Programme*. Ce type est défini par en ensemble d'attributs (langage, corps), un ensemble de méthodes (m_1, m_2) et un ensemble de règles actives (R_1, R_2 ...).

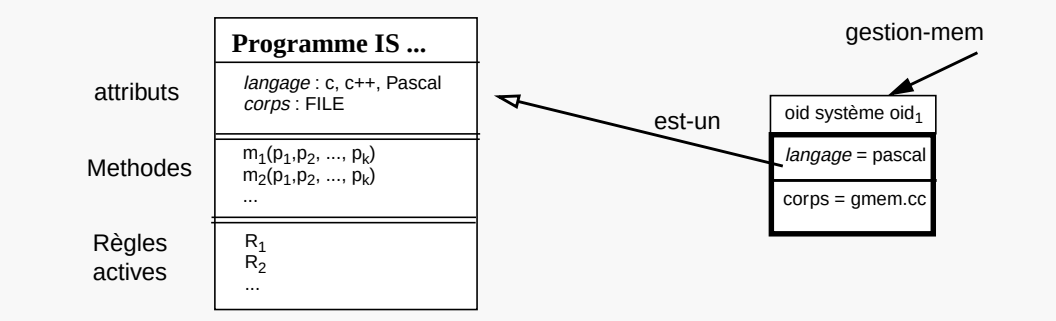


Figure 1 Le type d'objet *Programme* et l'instance *gestion-mem.*

Attributs

Un attribut est défini par un nom (identification), sa cardinalité (mono-valeur ou multi-valeur) et son domaine. Le domaine d'un attribut peut être atomique (entier, date, booléen, chaîne, énuméré ou fichier) ou un type d'objet défini par l'utilisateur. Ce dernier permet de représenter des associations que nous détaillerons par la suite.

Héritage

Un type peut avoir un ou plusieurs super-types directs. Par exemple, dans la figure 2, les types d'objet *Utilisateur, Document et ACtivité* sont sous-types de la racine appelée *Objet.*

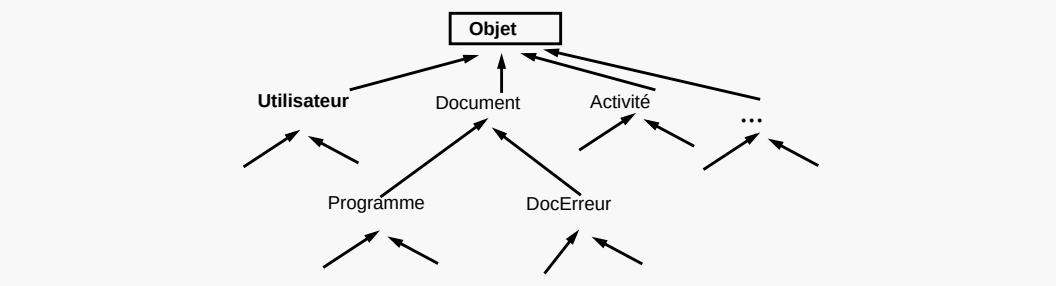


Figure 2 Exemple d'une hiérarchie de types d'objets.

La relation de sous-typage possède une sémantique de spécialisation et d'inclusion. C'est-à-dire que si un type T_1 est sous-type d'un type T_0, alors :

- T_1 hérites les propriété définies dans T_0. T_1 peut raffiner les propriétés définies dans T_0 ou ajouter d'autres propriétés.
- l'extension de T_1 est incluse dans l'extension de T_0 (L'extension d'un type est l'ensembl des instances de ce type).

Méthodes

Les méthodes sont utilisées pour décrire le comportement des instances d'un type. Une méthode est définie par une signature (ou l'interface de la méthode) et un corps (ou implémentation). Une méthode est héritée à travers la hiérarchie de types et son implémentation peut être redéfinie.

Le corps d'une méthode est écrit dans le langage de programmation associé au modèle. Il s'agit d'un langage de programmation impératif pour la manipulation des données.

Associations

Les associations servent à modéliser des relations entre deux objets. Les associations sont dirigées. Une association est établie entre un objet origine et un objet destination. L'existence de l'association est liée à l'existence de son origine et de sa destination.

Dans l'exemple de la figure 3, l'objet *casallas* est une instance du type d'objet *Utilisateur* et l'objet *codage* est une instance du type *Activité*. L'association représente que *casallas* est le *responsable* de l'activité *Codage*.

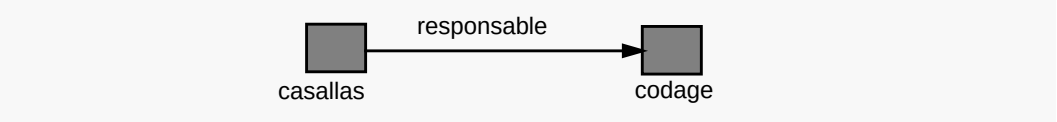


Figure 3 Exemple d'association.

Dans notre modèle, une association est considérée comme un objet. Une association est une instance d'un type qui définit sa structure et son comportement.

Type d'association

Comme pour un objet, la structure et le comportement d'une association sont donnés par son type. Dans un type d'association, on trouve, comme pour un type d'objet, des définitions d'attributs, de méthodes et de règles actives.

On trouve également le domaine de l'association qui détermine les types d'objets composant l'association. L'attribut *origine* indique le type des objet à partir desquels l'association peut être établie, et l'attribut *destination* indique le type des objet qui peuvent être destination de l'association (cf. figure 4).

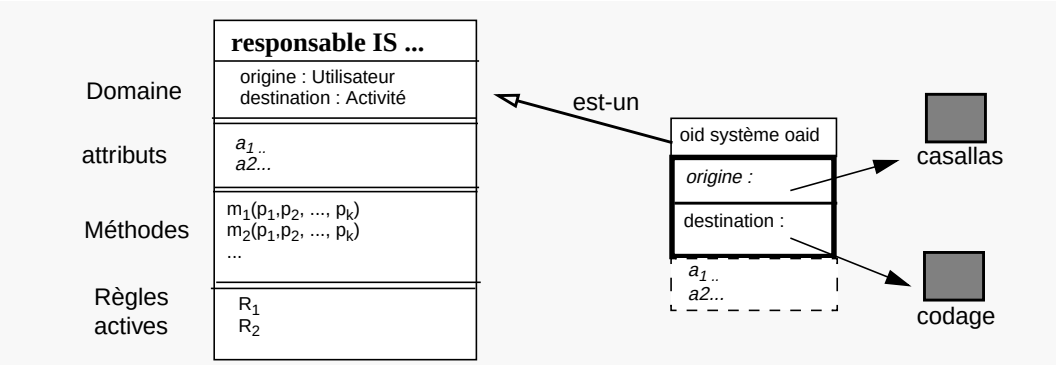


Figure 4 Le type d'association *responsable* et une de ses instances, qui lie les objets *casallas* (du type *Utilisateur*) et c*odage* (du type *Activité*).

Dans l'exemple de la figure 4, le type d'association *responsable* a pour domaine :

Utilisateur → Activité

L'origine d'une association, instance de *responsable*, est une instance du type (ou sous-type) *Utilisateur*, tandis que la destination de cette association est une instance du type (ou sous-type) *Activité (cf.* figure 4).

Sémantique des associations

Les associations entre objets peuvent correspondre à diverses relations telles que la dépendance, la dérivation, la composition, etc. La racine de la hiérarchie des types d'associations, appelée *Association*, ne possède pas une sémantique particulière. Le modèle offre toutefois un type d'association pré-défini, appelée *composition forte*, sous-type d'*Association* dont la sémantique est donnée par les deux propriétés suivantes :

a. exclusivité : un objet ne peut pas être destination (en même temps) de deux associations de type composition forte.

b. existencialité : si l'objet origine de l'association de composition forte est détruit, la destination est aussi détruite.

Objets composites et associations

Lorsqu'un type d'objets *T* possède un ou plusieurs attributs dont le domaine est un type d'objets, ce ou ces attributs sont appelés des attributs *composant*s. Une instance de *T* est alors un objet composite.

Dans notre modèle, les attributs composants sont traduits automatiquement (et gérés par le système) en des associations. Les associations de composition crées par défaut sont du type *composition forte*. Néanmoins, l'utilisateur peut redéfinir le type d'association. Nous verrons dans le chapitre 6 comment l'utilisateur peut définir la sémantique d'un type d'association en utilisant les règles actives (cf. chapitre 6 §1).

Dans l'exemple de la figure 5, *Programme* et *DocErreur* sont des sous-types du type d'objet *Document*. Dans le type *Programme*, l'attribut *rapport-erreurs* représente un composant dont le type est *DocErreur*.

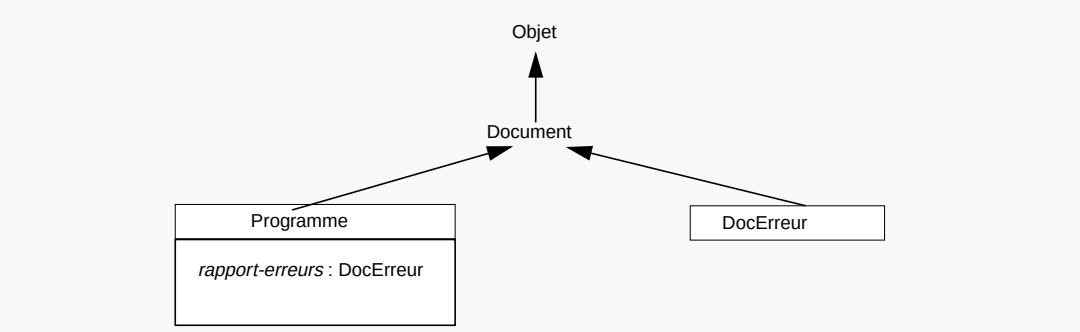


Figure 5 *Programme* est un type d'objets composite.

Le système crée le type d'association *rapport-erreur* sous-type de l'association pré-définie *Composition Forte*. Si l'utilisateur veut utiliser un autre type d'association de composition, il doit, en premier lieu, le définir (par exemple, *CompositionPartagée*) et,

en second lieu, déclarer le type de l'association de l'attribut (dans l'exemple, l'association rapport-erreurs) comme sous-type de ce nouveau type (cf. figure 6).

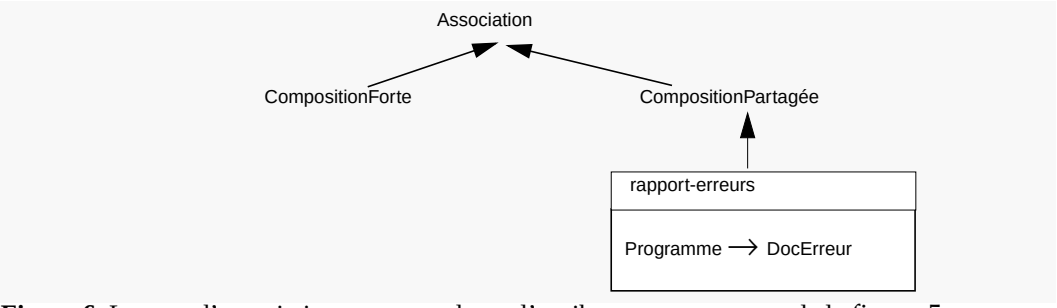


Figure 6 Le type d'association correspondant a l'attribut *rapport-erreurs* de la figure 5.

Lorsqu'un attribut composant est multi-valué, il y a autant d'instances d'association que d'éléments dans l'ensemble constituant sa valeur.

2 Objets Historiques

2.1 Introduction

Dans le chapitre 2, nous avons montré le manque de sémantique de la gestion de versions d'objets logiciels. Généralement abordée en termes de mécanisme plutôt qu'en termes de sémantique, cette gestion ne s'appuie sur aucun modèle expliquant les raisons qui sous-tendent le versionnement des objets. En l'absence d'un tel modèle, la création des versions est une opération dont les utilisateurs ont nécessairement la charge.

Pour remédier à ces insuffisances, nous avons défini *un modèle de versionnement à trois dimensions* (cf. chapitre 2 §2). Ce modèle permet de dégager et de distinguer trois problèmes orthogonaux qui se cachent derrière la gestion de versions. L'objectif de ce modèle consiste à profiter de cette distinction pour apporter à chaque type de problème des solutions spécifiques et adaptées.

Ces problèmes sont liés à l'évolution, à la variation de logiciels et au travail coopératif. Dans le modèle de versionnement, trois dimensions sont introduites pour prendre en compte chacun de ces problèmes : les dimensions historique, logique et coopérative.

L'objectif de **MOHA** est de proposer une solution aux problèmes liés à la dimension historique du versionnement. Dans un gestionnaire d'objets n'intégrant pas la notion de *temps*, les valeurs des attributs d'un objet décrivent son état courant. Quand les attributs de l'objet subissent des mises à jour, leur ancienne valeur est effacée et remplacée par une nouvelle. En revanche, dans le modèle **MOHA**, la mise à jour des attributs d'un objet n'efface pas nécessairement leur ancienne valeur. En effet, une mise à jour peut

provoquer la création d'un nouvel état de l'objet dans lequel prennent place les nouvelles valeurs des attributs modifiés. L'ancien état est, quant à lui, conservé.

La notion d'*objet historique* est introduite dans le modèle pour la prise en compte de l'évolution temporelle d'un objet. La valeur d'un *objet historique* est une séquence d'états. Un état constitue la valeur de l'objet historique à un instant donné.

Les états sont ordonnés selon l'instant de leur création. Par exemple, l'état $\texttt{état}_0$ est l'état initial de l'objet créé à l'instant t_0 ; l'état $\texttt{état}_n$ est l'état créé après l'état $\texttt{état}_{n-1}$. Si t_k est l'instant de création de l'état $\texttt{état}_{n-1}$, alors l'instant de création de l'état $\texttt{état}_n$ est $t_k+\delta$, avec $\delta > 0$. La durée entre deux états successifs n'est pas forcément la même.

Le symbole *now* représente l'instant courant par rapport à l'horloge du système.

La suite de cette section décrit la représentation d'un objet historique dans **MOHA** (cf. §2.2), les propriétés temporelles des attributs (cf. §2.3) et finalement, la création des états d'un objet historique (cf. §2.4).

2.2 Représentation d'un objet historique

Dans **MOHA**, tous les objets créés par l'utilisateur sont historiques. Le type d'un objet historique est un type construit (appelé *historique-of)*. Ainsi, pour tout type *T* défini par un utilisateur, le système crée un type d'objet historique correspondant (*historique-of T*).

Les objets historiques et les états sont représentés par des objets. Ainsi, la valeur d'un *objet historique OH* est une séquence d'objets d'un type *T* donné.

$$OH = <oid, \lceil état_0, état_1, ..., état_n \rceil>$$

oid est l'identificateur de l'objet historique et $\lceil état_0, état_1, ..., état_n \rceil$ est sa valeur qui représente une séquence d'états. Chaque état de l'objet historique est un objet.

Un *instant* est un point particulier de la droite du temps. Un *intervalle* de temps est un segment de la droite du temps, délimité par deux instants précis. La *durée* d'un intervalle est le nombre d'unités de temps qu'il comporte.

Nous modélisons le temps de façon discrète (il est isomorphe aux entiers). Un entier correspond à une unité de temps non décomposable et d'une durée de temps (*chronon)* dans notre modèle le chronon correspond à la seconde.

Les états et les intervalles

La figure 7 montre l'objet historique O_1. Supposons que t_0 soit l'instant de création de l'objet O_1. Les instants nommés t_1, t_2 et t_3 correspondent aux instants auxquels un nouvel état de l'objet O_1 a été créé.

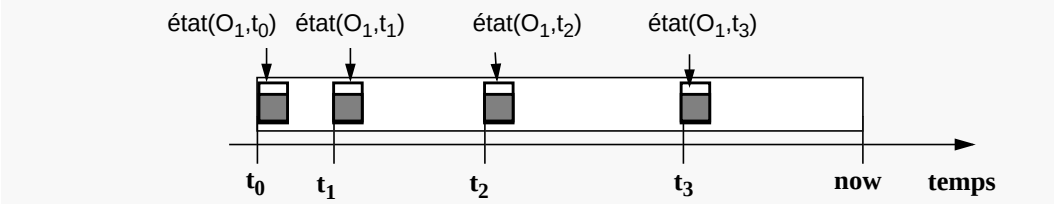


Figure 7 Les état de l'objet historique O_1.

Bien que les états d'un objet historique soient enregistrés de manière discrète, la valeur de cet objet est interprétée de façon continue dans le temps. En effet, la valeur d'un état reste valide pour l'objet historique tant qu'elle n'est pas changée[1].

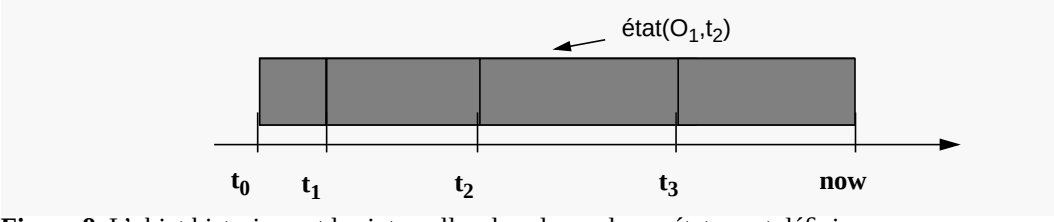


Figure 8 L'objet historique et les intervalles dans lesquels ses états sont définis.

La figure 8 montre l'interprétation continue des valeurs des états de l'objet historique. La valeur de l'objet O_1 dans l'intervalle $[t_0, t_1[$ correspond à la valeur de l'état état(O_1,t_0), sa valeur dans l'intervalle $[t_1, t_2[$ correspond à la valeur de l'état état(O_1,t_1), etc.

De façon générale, la valeur d'un objet historique *O* à un instant t est la valeur de l'état de *O* à cet instant. Ainsi, si t_a et t_b correspondent à deux instants auxquels deux états de *O* ont été créés, et s'il n'y a pas eu d'autre création d'état pendant l'intervalle $]t_a, t_b[$, alors, pour tout t, tel que

$$t_a \leq t < t_b,$$

$$\text{valeur (OH,t)} = \text{valeur (état(OH, } t_a\text{))}$$

La période de vie d'un objet historique

L'opération consistant à *détruire* un objet historique est réalisée explicitement par l'utilisateur. Un objet historique peut être détruit uniquement s'il ne participe à aucune association (ni aucun de ses états). Si ce n'est pas le cas, nous dirons que l'objet ne peut plus évoluer ou qu'il est *mort*. Tant que l'objet peut évoluer, nous dirons que l'objet est *vivant*.

Un objet historique, en tant qu'objet, possède des informations qui sont gérées par le système et qui sont accessibles à l'utilisateur via des fonctions spéciales prédéfinies. En particulier, *lifespan* est la fonction qui rend la *période de vie* de l'objet historique. La période de vie d'un objet historique correspond à l'intervalle de temps pendant lequel l'objet a été *vivant* dans la base. Supposons qu'un objet historique ait été créé à l'instant t_0 et qu'il ait été détruit à l'instant t_{mort}, dans ce cas, la période de vie de l'objet

1. *step-wise constant* dans la littérature anglophone.

correspond à l'intervalle $[t_0, t_{mort}]$. Tant que l'objet n'est pas mort, sa période de vie correspond à l'intervalle $[t_0, now]$.

Si *valeur (OH)* = $\lceil$ état$_0$, état$_1$, ... , état$_i$, ... , état$_n$ $\rceil$ alors les propriétés suivantes sont vérifiées :

a. lifespan(état$_k$) = [t_k, t_{k+1}[ou t_k, t_{k+1} sont les instants de création respectivement de l'état état$_k$ et de son état successeur état$_{k+1}$, k = 0, ..., n-1

b. $\forall$ i,j i≠j lifespan(état$_i$) $\cap$ lifespan(état$_j$) = $\varnothing$

2.3 Propriétés des attributs

Tous les attributs d'un objet historique ne possèdent pas nécessairement le même comportement vis-à-vis du temps. Avant de présenter les trois catégories d'attributs proposées par **MOHA**, un exemple simple permet d'en souligner les différences fondamentales.

Considérons un type d'objet *Programme*, représentant les composants logiciels, défini par :

- un attribut *langage* dont la valeur indique le langage dans lequel est codé le programme,
- un attribut *corps* contenant le code source du programme,
- un attribut *rapport-erreurs* contenant l'ensemble des erreurs trouvées dans le *corps* du programme.

Soit un objet historique *gestion-mem* de type *historique-of Programme.* Cet objet représente un composant logiciel rassemblant les fonctions de gestion de la mémoire d'un système en cours de développement. Son évolution se caractérise par l'ensemble des états dans lesquels l'attribut *corps* a subi des améliorations ou des corrections.

Si l'attribut *corps* peut être modifié d'un état à l'autre, il n'en va pas de même pour l'attribut *langage*. En effet, le langage de programmation utilisé reste le même pour tous les états. La valeur de l'attribut *langage* est donc partagée par tous les états de l'objet historique *gestion-mem.*

La valeur de l'attribut *rapport-erreurs* répresente les erreurs trouvées dans un état spécifique de l'objet. Or, la découverte de toutes les erreurs d'un programme est un processus progressif qui peut même intervenir lorsqu'il est déjà en service chez le client. L'attribut *rapport-erreurs* peut donc être *modifié,* à n'importe quel instant, sur n'importe quel état de l'objet historique *gestion-mem*, simplement pour notifier une nouvelle erreur rencontrée sur cette version du programme. Cette modification de l'attribut *rapport-erreurs* ne justifie pas la création d'un nouvel état de *gestion-mem*.

Les trois attributs de l'objet historique *gestion-mem* possèdent des comportements temporels différents. **MOHA** permet de distinguer ces comportements en offrant la possibilité de déclarer trois sortes d'attributs :

Attributs partagés

Les attributs partagés sont communs à tous les états de l'objet. Par exemple, la valeur de l'attribut *langage* du type *Programme* est commune à tous les états de l'objet historique *gestion-mem*.

On peut changer la valeur d'un attribut partagé à n'importe quel instant et cette modification est visible par tous les états de l'objet historique.

Attributs modifiables

Ces attributs représentent des caractéristiques spécifiques à un état qui peuvent être modifiées (complétées) au cours du temps. Même si en théorie on ne peut pas changer le passé, dans le domaine du génie logiciel, la valeur de certains attributs des états peut changer sans que ses anciennes valeurs soient conservées. L'attribut *rapport-erreurs* de l'exemple constitue un exemple de tels attributs.

Les attributs modifiables d'un état peuvent donc être mis à jour à n'importe quel instant après la création de l'état.

Attributs immuables

Ce sont les attributs dont la valeur, une fois enregistrée dans un état de l'objet, ne peut pas être mise à jour dans cet état. Dans l'exemple, l'attribut *corps* est un attribut immuable dont la valeur, une fois enregistrée dans un état, ne peut plus changer.

2.4 Création d'états

La création d'un objet d'un type *T* implique la création par le système d'un objet historique (du type *historique-of T*) et de son premier état (du type *T*).

La modification de l'état d'un objet de la base se fait durant une transaction, ce qui permet de modifier simultanément plusieurs attributs immuables. Si des attributs immuables sont modifiés, le système crée, à la fin de la transaction, un nouvel état de l'objet historique. Ce nouvel état devient l'état courant de l'objet historique. Entre deux états successifs, il y a donc au moins un attribut immuable qui a changé. Autrement dit :

- si, état(O, t_i) est le dernier état de l'objet historique O, alors, le nouvel état créé, état(O,t_i+δ) (δ>0), devient le dernier état (et également l'état courant état(O,now)) et les valeurs de ses attributs immuables ne peuvent plus être changées.

3 Associations et évolution d'objets

Dans une base de données historique, il est non seulement important de pouvoir représenter et gérer l'évolution des entités dans le temps mais aussi l'historique des relations entre ces entités. Par exemple, dans une base historique contenant l'histoire des employés d'une entreprise, il est primordial de pouvoir maintenir des informations telles que : "pendant l'été 1995, l'employé *Dupont* a travaillé au rayon *chaussures"* ou, "l'employé *Dupont* a été *le chef* du rayon *vêtements* pendant les cinq dernières années". Dans ces deux cas, il s'agit de représenter des associations (*travaille-pour* et *être-chef* dans les exemples) dont la validité dure un certain temps dans la réalité.

Dans le contexte des environnements guidés par les procédés de fabrication de logiciel, s'il est nécessaire de pouvoir gérer l'historique de certaines associations, il est également nécessaire de pouvoir établir des associations entre entités provenant d'époques différentes. Cette dernière caractéristique, absente des bases de données historiques, est présente dans au contexte du génie logiciel.

Par exemple, il faut pouvoir décrire des faits comme "l'utilisateur *casallas* a été responsable de l'activité *codage* durant le mois dernier" qui implique que la relation *responsable-de* possède un historique. Mais, il faut aussi permettre la création d'une configuration de logiciels composée d'objets (états) provenant du passé. C'est-à-dire que cette configuration fait référence à des états d'objets historiques qui ne sont pas les plus récents (ils ne peuvent plus évoluer).

De même qu'un état du présent peut faire référence à des états du passé, un état du passé peut faire référence à un objet créé à un instant postérieur. En effet, dans l'exemple de la section précédente, l'attribut *rapport-erreurs* est un attribut composant dont la valeur peut être une référence à un objet qui a été créé après la mort du composite. En effet, le programme contenu dans un des attributs immuables, peut être utilisé même après que son état ait été figé, et des erreurs le concernant peuvent encore être trouvées.

On parlera d'*association non historique* (cf. §3.1) lorsqu'il s'agit de créer des associations entre des objets qui n'appartiennent pas aux mêmes époques (ou périodes de temps). On parlera *d'association historique* (cf. §3.2) lorsqu'il s'agit de représenter et gérer la validité temporelle d'une association entre deux objets. Pour répondre aux besoins des environnements guidés par les procédés, notre objectif consiste à offrir un modèle capable de représenter et gérer des associations historiques et non historiques.

3.1 Les associations non-historiques

Une association *non historique* est un cas courant dans le domaine du génie logiciel. Bien souvent, les objets passés (ceux qui n'évoluent plus) continuent a être utilisés dans le présent (par exemple, un logiciel qui a été livré chez un client). Dans l'exemple de la figure 9 nous avons un objet appelé *NewConf* qui maintient des relations avec des états d'objets historiques provenant de différentes périodes de temps.

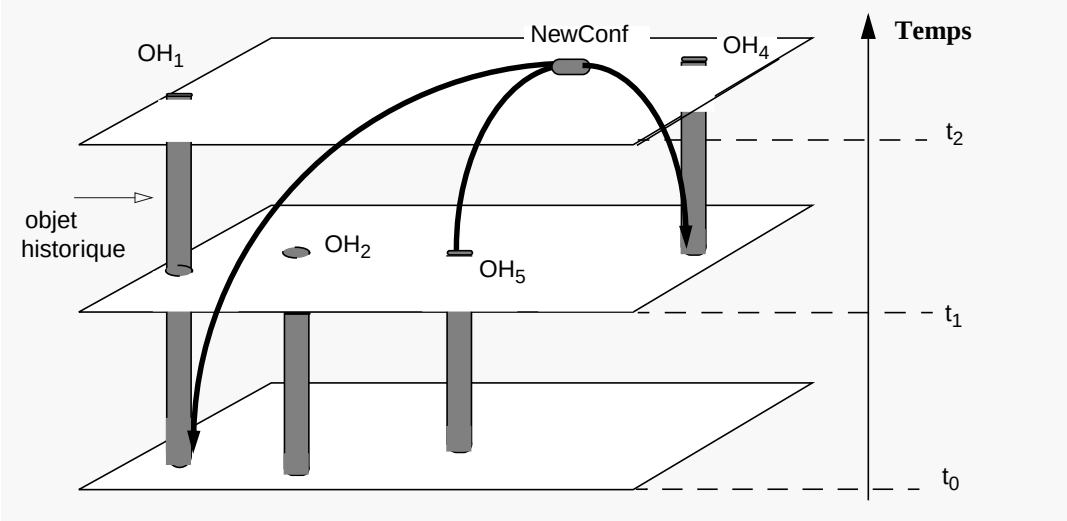


Figure 9 Les objets historiques et les associations non historiques.

La figure 10 montre les diverses associations qui peuvent être établies entre des objets historiques ou des états.

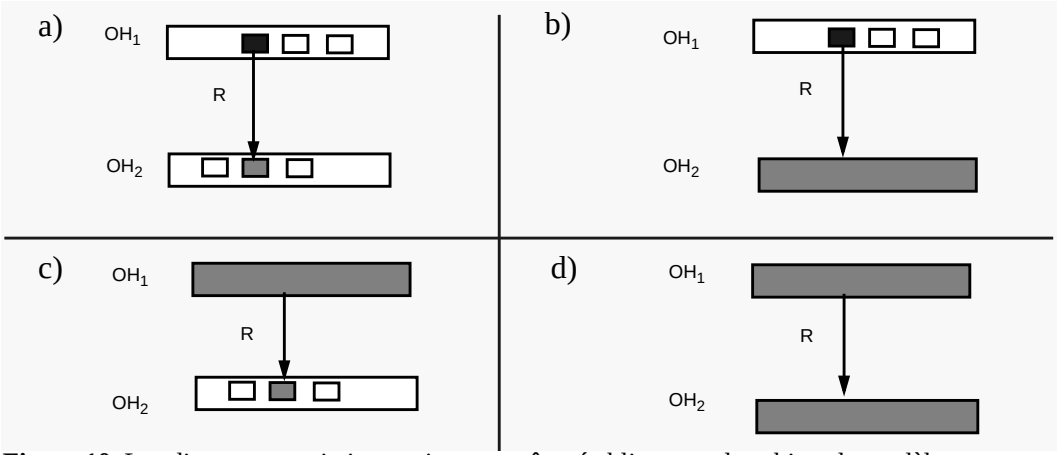


Figure 10 Les diverses associations qui peuvent être établies entre les objets du modèle.

Dans **MOHA**, les associations non historiques ont les mêmes caracteristiques que les associations du modèle de référence (cf. §1), c'est-à-dire qu'une instance d'association est un objet avec un attribut *origine* qui contiendra une référence à l'objet origine et un attribut *destination* qui contiendra une référence à l'objet destination de l'association.

Les objet historiques dans les associations non historiques

Une association impliquant un objet historique signifie que tous les états constituant l'objet historique sont eux aussi impliqués dans l'association. C'est-à-dire que l'association est *partagée*, de la même façon que les valeurs des attributs *partagés*, par tous les états de l'objet historique. Si, par exemple, un objet historique est destination d'une association (figure 11-a, l'objet O_1 est destination de l'association A_j) alors chacun de ses états est aussi destination de l'association (figure 11-b, en particulier, l'objet état(O_1,t_k) est destination de l'association A_j).

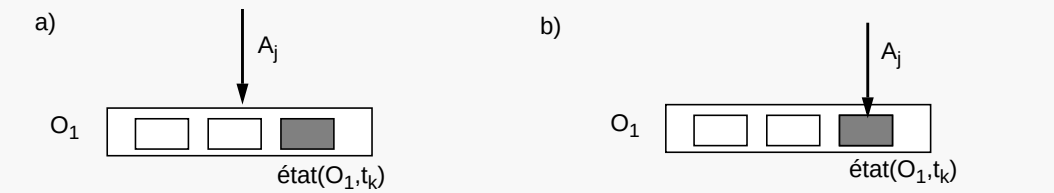


Figure 11 Exemple de partage d'une association dans laquelle la destination est un objet historique.

Nous verrons dans le chapitre 5 comment cette propriété est exploitée par le langage d'interrogation.

Les états dans les associations non historiques

En tant qu'objet, un état peut être l'origine ou la destination d'une association. Dans ce cas, la valeur de l'attribut origine, ou destination, de l'association est une référence à l'état impliqué. Dans la figure 10, le cadre a) montre qu'un état de OH_1 et un état de OH_2 sont respectivement origine et destination d'une association *R*. Les cadres b) et c) illustrent les cas où un état et un objet historique peuvent être mis en relation à travers une association *R*.

Dans le cas des attribut composants *immuables* ou *modifiables*, l'association correspondante (créée par le système) a toujours pour origine un état. Si l'attribut est *partagé*, l'association correspondante a pour origine l'objet historique. En revanche, la destination peut être un objet historique ou un état selon les besoins de l'utilisateur. C'est à lui de définir le type de référence qu'il souhaite donner comme valeur à l'attribut.

Dans l'exemple de la figure 12, l'attribut A_i (supposons immuable ou modifiable) de l'objet état(O_1,t_m) a pour valeur une référence à un état de l'objet O_2 (état(O_2,t_k)). Puisqu'il s'agit d'une référence à un état, l'évolution de l'objet O_2, n'est pas visible par l'attribut A_i de l'objet état(O_1,t_m). Seules les modifications sur les attributs *partagés* de O_2 et sur les attributs *modifiables* de état(O_2,t_k) sont visibles depuis A_i.

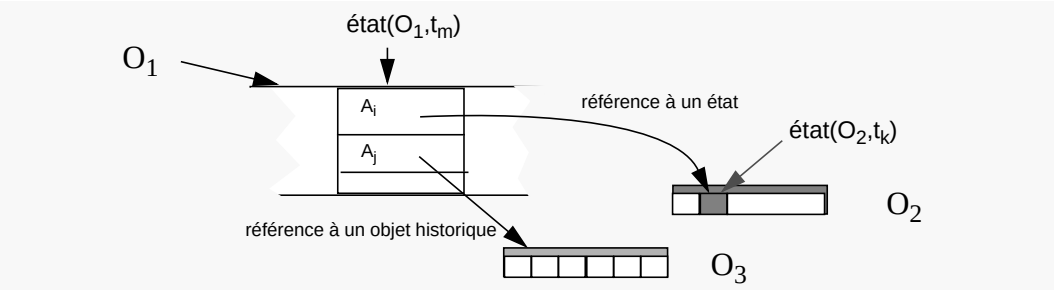


Figure 12 L'objet historique O_1. Les états de cet objet sont des objets composites.

L'attribut A_j est une référence à l'objet historique O_3. L'évolution de cet objet sera visible par cet attribut. L'intérêt est, par exemple, d'avoir accès aux dernières modifications de l'objet.

Dans ce qui suit, nous présentons un exemple de gestion de configurations dans lequel des associations non historiques sont utilisées.

Exemple

Une des fonctions de base que doit offrir un système de gestion de configurations est la reconstruction d'un logiciel dont les composants appartiennent au passé. Supposons que nous représentons une configuration de logiciel par un objet. L'attribut nommé *comps* a pour valeur l'ensemble des modules composant la configuration (cf. figure 13). Les modules sont de type *Programme*. Ces programmes sont des objets historiques. Or, pour être bien instancié, une configuration doit référencer un état unique pour chaque programme. De cette façon, la configuration (l'état de la configuration) n'est pas altérée par l'évolution (les modifications) des modules.

Dans la figure 13, *conf-test* est un objet historique dont les états sont de type *Configuration.* Chaque état de *conf-test* correspond à une configuration de test particulière créée à un instant donné. Par exemple, l'attribut *comps* de l'état état(conf-test,t_k) a comme valeur l'ensemble de références : état(M_1, t_r), état(M_2, t_i), état(M_3, t_j). Notons que l'association entre état(conf-test, t_k) et état(M_3, t_j) est une association non-historique car l'état de l'objet M_3 considéré n'est plus l'état courant.

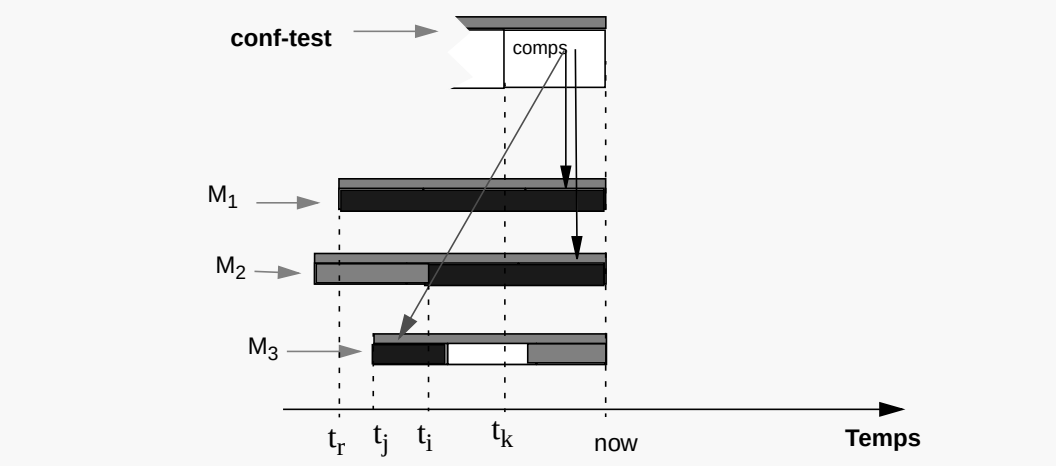


Figure 13 L'attribut *comps* de l'objet état(conf-test, t_k).

Si l'attribut *comps* est immuable, et si t_k est l'instant de création de l'état contenant la configuration, alors cette composition est figée dans l'état état(conf-test, t_k) de la configuration. On pourra toujours revenir en arrière pour reconstruire un système logiciel à partir d'une configuration donnée.

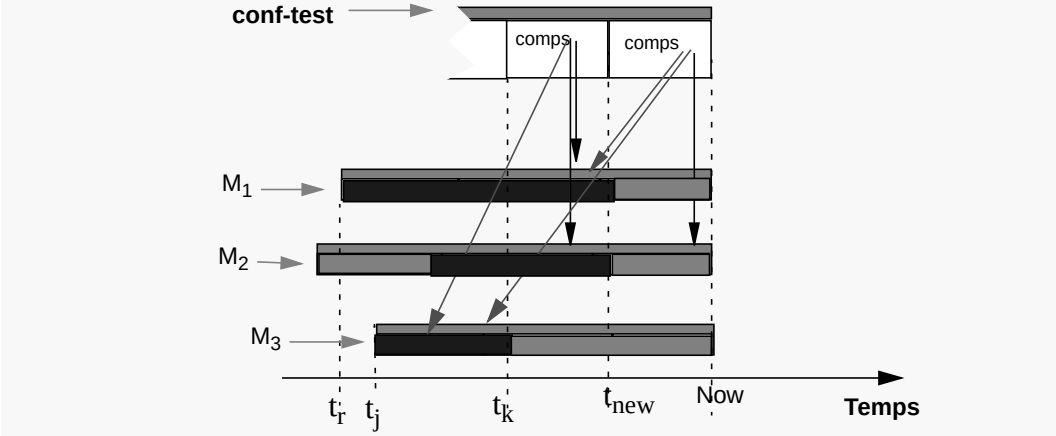


Figure 14 Evolution de l'objet configuration *conf-test*.

Nous voulons construire une nouvelle configuration en prenant en compte uniquement les nouvelles modifications faites sur le module M_2 (l'état état(M_2, t_{new}) de la figure 14). On modifie la composition de la configuration *conf-test* de telle façon que l'attribut *comps* contienne, d'une part, les mêmes références aux états du module M_1 et M_3 et, d'autre part, la nouvelle référence à M_2.

Le système crée un nouvel état de l'objet historique *conf-test*, car l'attribut *comps* est un attribut immuable. Notons que la configuration précédente est conservée (figure 14). L'attribut *comps* du nouvel état de *conf-test* a pour valeur :

{état(M_1, t_r), état(M_2, t_{new}), état(M_3, t_j) }

3.2 Associations historiques

Les associations historiques sont établies entre des objets historiques. Elles ont une *période de validité* qui correspond à l'intervalle de temps entre l'instant de création et l'instant de mort de l'association. La notion de référence historique sur un objet est introduite (cf. §3.2.1). Cette notion permet de restreindre la visibilité d'un objet historique à certains intervalles de temps inclus dans sa période de vie. C'est à travers des références historiques qu'est maintenue la contrainte d'existence temporelle imposée par une association historique (cf. §3.2.2).

3.2.1 Références Historiques

Les objets historiques sont manipulés à travers des identificateurs qui nous permettent de tenir compte de la période de vie de l'objet, ces identificateurs sont appelées *références historiques.*

Une référence historique contient les informations suivantes : l'identité de l'objet historique et un ensemble d'intervalles de temps qui définit la *période de visibilité* de l'objet historique. Cette notion de référence historique permet d'avoir une vue partielle

de l'objet ; plusieurs références historiques peuvent être définies pour le même objet, différentes vues partielles de l'objet peuvent être obtenues simultanément.

oid	$\{I_1 = [t_{11} , t_{12}[, I_2 = [t_{21}, t_{22}[, ... I_n = [t_{n1},t_{n2}[\}$

Un ensemble d'intervalles est appelé un *élément temporel* [Gad88]. Les intervalles doivent être disjoints et doivent être inclus dans la période de vie de l'objet :

a. $I_1 < I_2 < ... < I_n$, où $I_j < I_k$ signifie que si $I_j = [t_{j1}, t_{j2}[$ et $I_k = [t_{k1}, t_{k2}[$ alors, $t_{j2} \leq t_{k1}$

b. $t_{création} \leq t_{11}$ et, $t_{n2} \leq t_{mort}$ ou $t_{n2} \leq$ now, $t_{création}$ est l'instant de création de l'objet.

Si pour une référence historique on ne précise pas la période de visibilité, l'intervalle pris par défaut correspond à la période de vie de l'objet référencé. Ainsi, dans l'exemple de la figure 15, nous avons la référence historique appelée *gestion-mem* qui a pour valeur l'identificateur de l'objet historique qu'elle référence et la période de vie de cet objet.

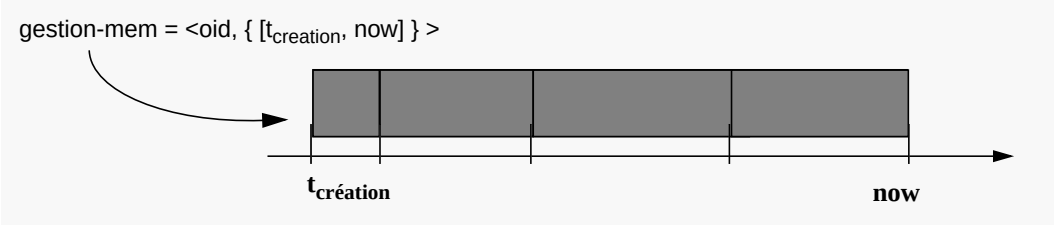


Figure 15 Les références historiques.

Si, on s'intéresse à un objet uniquement dans l'intervalle $[t_i, t_j[$ de sa vie, il suffit de définir une référence historique sur cet objet dont la période de visibilité est $[t_i, t_j[$ (figure 16).

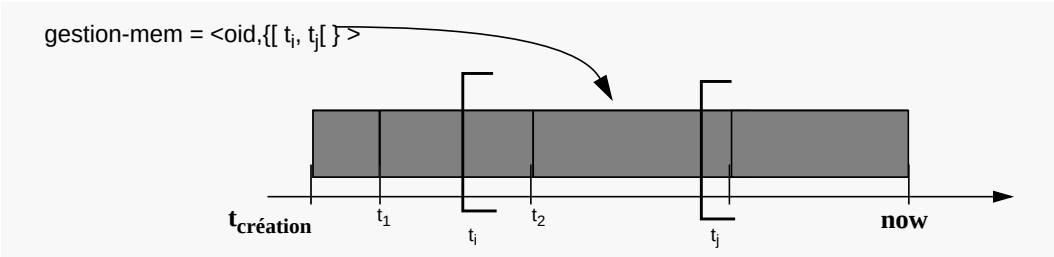


Figure 16 Référence historique avec une période de visibilité plus courte que la période de vie de l'objet historique.

Les valeurs de l'objet historique aux instants t_i et t_j sont :

valeur (gestion-mem, t_i) = état(gestion-mem, t_1)

valeur (gestion-mem, t_j) = état(gestion-mem, t_2)

3.2.2 La contrainte d'existence temporelle

Une association historique est une association dont les attributs *origine* et *destination* ont pour valeur des références historiques qui vérifient la *contrainte d'existence temporelle.* Cette contrainte impose que les références historiques de l'origine et de la destination de l'association possèdent les mêmes périodes de visibilité. La contrainte d'existence temporelle est une propriété que l'utilisateur peut donner aux types des associations.

Dans le cas général, la création d'une association historique est indépendante de la création des états des objets qu'elle lie. Par exemple, supposons que l'utilisateur *casallas* devienne, à l'instant 92-09-01, *responsable-de* l'activité *codage* (cf. figure 17*)*. A l'instant 95-04-01, l'utilisateur *casallas* n'est plus responsable de cette activité. Durant la période de validité de l'association, l'utilisateur *casallas* et l'activité *codage* ont suivi leur propre évolution. L'état de l'utilisateur est resté le même, tandis que celui de l'activité a changé deux fois.

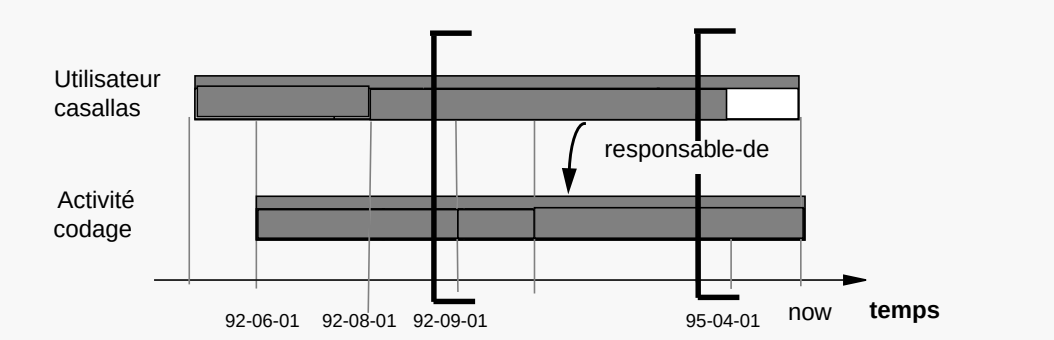


Figure 17 Intervalle de validité de l'association *responsable-de* entre l'utilisateur *casallas* et l'activité *codage*.

La figure 18 montre la valeur de l'association précédente (figure 17).

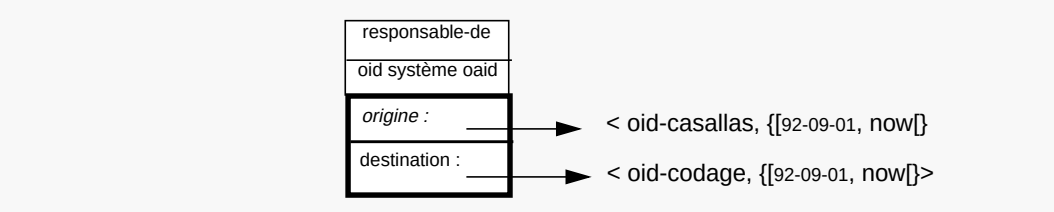


Figure 18 Objet *responsable-de* à l'instant de la création de l'association.

Tant que l'association est vivante, sa période de validité est donnée par l'intervalle [92-09-01, now[. A l'instant de la destruction de l'association, le système modifie la période de visibilité des références origine et destination de l'association en précisant l'instant de la destruction de l'association, dans l'exemple 95-04-01.

4 Conclusion

4.1 Synthèse

Dans ce chapitre nous avons présenté **MOHA**, le modèle pour le gestionnaire d'objets d'un environnement guidé par les procédés. Le modèle de référence sur lequel nous avons défini la notion d'objet historique est celui qui a été présenté dans la section §1 et qui correspond, à quelques détails près, au modèle de données du système Adèle.

La notion d'objet historique prend place dans ce modèle pour représenter l'évolution des objets dans le temps. Cette évolution correspond à la dimension historique du versionnement que nous avons définie dans le chapitre 2 de ce document. L'évolution d'objets est stockée dans le système en tenant compte des modifications faites sur les propriétés définies comme *immuables* par l'utilisateur. La création des états qui représentent les *versions historiques* des objets est gérée de façon automatique par le système.

Nous avons aussi défini les associations historiques et les associations non-historiques. Les associations historiques sont établies entre des objets historiques. Elles ont une *période de validité* qui correspond à l'intervalle de temps entre l'instant de création et l'instant de mort de l'association. L'association meurt soit lorsque l'utilisateur l'annonce explicitement, soit parce qu'un des objets impliqués meurt. Après sa mort, l'association n'est plus valide entre les deux objets, elle devient une information historique.

Les associations non historiques n'imposent pas de contraintes temporelles entre les objets qu'elles lient. C'est-à-dire qu'en utilisant les associations non-historiques, on peut établir des associations entre des objets qui appartiennent à des intervalles de temps différents.

4.2 Liens avec les autres travaux

Cette section a pour objectif de comparer MOHA et les travaux introduits dans le chapitre 2. Tout d'abord, nous situons notre modèle par rapport aux travaux portant sur les gestionnaires de versions, et sur ceux des bases de données temporelles à objets.

MOHA et la gestion de versions

Alors que les gestionnaires de versions du type SCCS [Roc75] ou ClearCase [Leb94] (cf. chapitre 2) se limitent au versionnement des fichiers, **MOHA** est un modèle qui permet de prendre en compte de façon générale l'évolution des objets.

En ce qui concerne la gestion du graphe de dérivation, **MOHA** s'appuie sur une sémantique précise de versionnement. En effet, basé sur une sémantique de

versionnement historique, **MOHA** permet de modéliser quelles sont pour chaque objet les propriétés dont on veut conserver un historique, c'est-à-dire les propriétés *immuables* de l'objet.

MOHA et les modèles à objets temporels

Contrairement aux modèles de données temporelles qui introduisent deux dimensions du temps, à savoir le temps de validité et le temps de transaction (cf. chapitre 2 §2.3.2), nous considérons une seule dimension du temps. Deux arguments peuvent être avancés pour justifier ce choix.

Le premier argument provient de la nature même des événements qui sont considérés dans le domaine du génie logiciel. La plupart d'entre eux sont effectivement des événements relatifs à des entités qui n'ont pas d'existence dans le monde réel mais seulement dans la machine (construction d'une configuration, compilation d'un système, etc.). Dans ces conditions, c'est donc l'enregistrement d'un événement dans la machine qui détermine son occurrence.

Le second argument reprend la critique générale, formulée dans [Sar93], à l'encontre du principe de gestion d'un décalage entre l'occurrence d'un événement et son enregistrement dans le système. En effet, pour les utilisateurs, l'intérêt d'un système dans lequel de tels décalages apparaissent fréquemment devient limité, voire inadéquat. Cette constatation est d'autant plus vraie dans le contexte de procédés de fabrication de logiciel que le système a pour fonction principale de contrôler la concordance entre l'exécution dans la machine du modèle de procédés et sa réalisation dans le monde réel.

Représentation de l'évolution des objets

Parmi les modèles de données temporels, on trouve deux types de représentations possibles pour décrire l'évolution des objets. L'une se base sur la description de l'évolution temporelle de chaque attribut d'un objet, tandis que la seconde s'appuie sur la notion d'état de l'objet qu'elle organise en séquence.

L'avantage évident de la première solution réside dans le fait qu'en ramenant l'évolution d'un objet à l'ensemble des évolutions de ses attributs, on évite de dupliquer les informations qui ne varient pas d'un état à l'autre. En revanche, l'inconvénient de cette solution provient de la perte de la structuration logique de l'évolution d'un objet en états. En effet, dans ce type de modèle, un état particulier est éparpillé à travers l'ensemble d'évolutions des attributs de l'objet. La notion d'état ne constituant pas un concept de ces modèles, elle ne peut donc pas être exploitée facilment.

La seconde solution, adoptée dans **MOHA**, donne à la notion d'état un rôle central. Son avantage réside dans la possibilité de considérer les états comme des objets à part entière et donc de pouvoir les référencer individuellement, les impliquer dans des associations, etc. Par ailleurs, le problème de duplication d'information entre deux états successifs peut aussi être évité avec cette solution. Il suffit de mettre en place un mécanisme de *delta* (cf. chapitre 2 §2.1.1), transparent pour les utilisateurs, permettant de ne considérer au niveau d'un état que les informations qui le distinguent de l'état qui le précède.

Propriétés temporelles des objets

En général, les modèles de données temporels ne permettent pas de préciser au cas par cas le comportement temporel des entités modélisées. Ainsi, l'évolution d'un objet est fixée par l'évolution de chacun de ses attributs. D'un attribut à un autre, l'évolution se définit sans distinction aucune : elle décrit l'histoire de la valeur de l'attribut.

En revanche, dans **MOHA**, il y a une catégorie d'attributs qui ne peut *jamais* être modifiée et une autre catégorie qui peut être corrigée, complétée, etc. Même si les états des objets sont figés (c'est-à-dire qu'on ne peut plus les modifier), ces états continuent à participer ou en quelque sorte, à être actifs même si leur période de vie par rapport au système est déjà finie. Ceci explique d'une part, le besoin de disposer des attribut *modifiables* dont leur valeur peuvent être mises à jour et, d'autre part, le besoin de pouvoir établir des associations non-historiques. Ce point est traité par la suite.

Les association entre objets et le temps

Selon la taxonomie présenté par [EKF93](cf. chapitre 2 §2.3.2) notre modèle est à l'intersection entre les modèles qui représentent explicitement les relations entre objets par des objets de première classe, et ceux qui proposent l'évolution des objets par objet comme un tout et non pas attribut par attribut. Comme dans le modèle présenté dans [EKF93] ou le modèle OSAM*/T [SC91], **MOHA** permet de représenter des associations historiques et de garantir les contraintes d'existence temporelle. Les associations historiques correspondent à des relations possédant une période de validité dans la réalité.

De plus, pour les besoins du domaine, **MOHA** permet aussi de représenter et gérer des associations *non-historiques* qui établissent des liens entre objets sans se préoccuper de satisfaire la contrainte d'existence temporelle. C'est-à-dire que nous pouvons établir des associations à un instant donné entre des objets du passé. Ces objets sont encore *utilisables* même s'ils ne peuvent plus évoluer.

Chapitre 5

Le langage d'interrogation de MOHA

Ce chapitre présente **LOHA** un langage permettant d'interroger une base **MOHA**. Ce langage d'interrogation est un complément au langage de programmation du gestionnaire d'objets. Ce dernier, qui ne sera pas présenté dans ce document, est un langage impératif servant à la manipulation des données. Le langage **LOHA**, quant à lui, ne possède aucune primitive de mise-à-jour et est de nature déclarative. Les expressions **LOHA** peuvent être utilisées dans du code impératif (par exemple, dans une méthode).

Ce chapitre est organisé de la façon suivante. Les deux premières sections introduisent de façon intuitive le langage en présentant brièvement ses concepts principaux puis en les illustrant à travers un exemple. Les sections suivantes détaillent les différents éléments du langage.

1 Introduction

LOHA est un langage déclaratif dont le but est d'offrir un moyen simple et puissant d'accéder aux informations contenues dans une base **MOHA**. Les types d'informations accessibles correspondent aux différents niveaux de structuration qu'offre le modèle à objets **MOHA** : objets et attributs. La prise en compte de la dimension historique permet de raffiner le type de résultat selon que l'on accède à un objet historique complet, à une vue partielle de cet objet ou encore à l'un de ses états.

Etant données les associations entre objets, une base **MOHA** se présente comme un ensemble de graphes orientés. Dans ce type de graphes, les objets constituent les sommets tandis que les associations correspondent aux arêtes. La figure 1 donne un exemple d'une base **MOHA**.

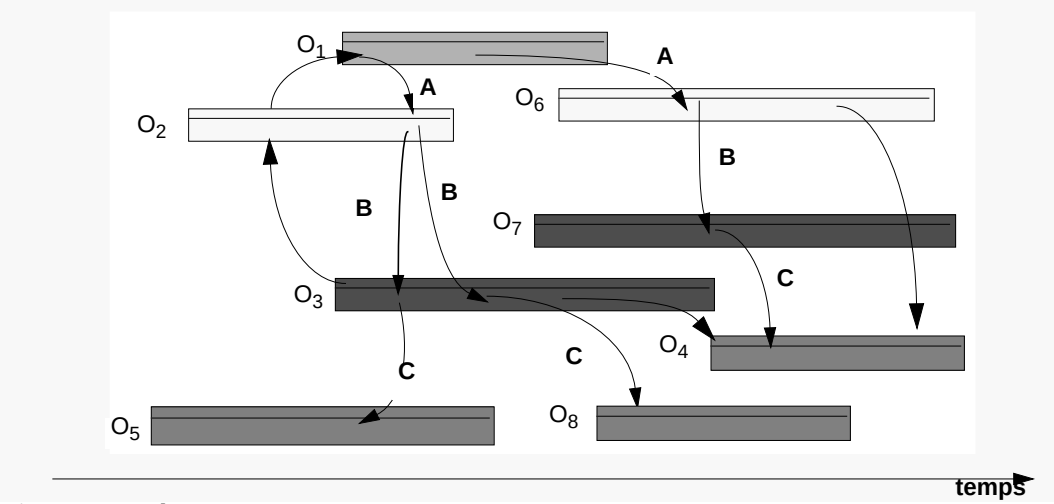


Figure 1 Une base MOHA.

La puissance du langage **LOHA** provient des possibilités de navigation à travers les graphes d'une base **MOHA**. Ce langage repose sur deux notions principales : les *expressions de chemins* et les *filtres*.

Une expression de chemins permet de sélectionner des sous-graphes d'une base **MOHA** à partir d'une sélection initiale d'objets. Les sous-graphes sélectionnés sont des chemins de la base **MOHA** dont la racine appartient à la sélection initiale d'objets et dont la structure s'unifie avec l'expression donnée. Cette unification consiste à vérifier que le chemin est constitué de la succession d'associations décrite dans l'expression.

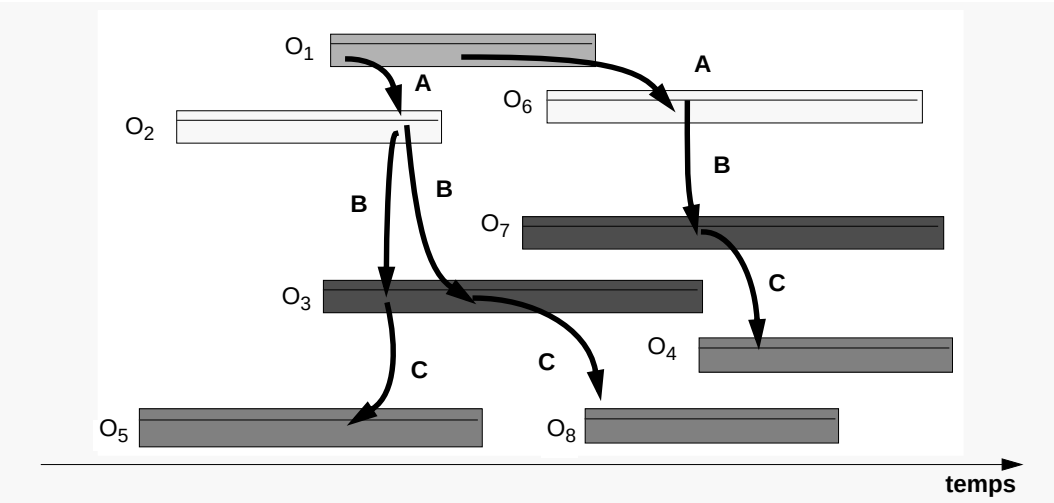


Figure 2 Exemples de chemins

Dans l'exemple de la figure 2, nous avons trois chemins qui satisfont une expression de chemins pouvant être décrite informellement de la façon suivante : "en partant de l'objet O_1 et en traversant l'association A, puis l'association B et, finalement l'association C".

La notion de filtre, appliquée à un objet, permet d'établir si l'objet vérifie les propriétés exprimées dans le filtre. Cette notion, combinée à celle d'expression de chemins, permet de raffiner la navigation de **LOHA**.

2 Un exemple

Nous avons choisi de présenter un exemple simple qui va nous permettre d'illustrer les différents types de requêtes que nous pouvons exprimer en utilisant **LOHA**. L'exemple décrit une base de projets de fabrication de logiciels dont les entités gérées sont : les projets, les agents qui y participent et les activités dont ils sont responsables.

Pour simplifier, seules les activités de codage sont prises en compte. Une activité de codage a comme propriétés une configuration (c'est-à-dire un ensemble de programmes) d'entrée et une configuration de sortie. Nous supposons que chaque activité de codage est un cycle qui alterne une phase de développement et une phase de test. Le cycle se termine quand la configuration de sortie possède un statut *délivré*.

La figure 3 présente la définition des types de notre exemple. La partie *Association* décrit, d'une part, les types d'associations de composition créés par le système et, d'autre part, le type d'association *Configuration* défini par l'utilisateur et dont le but est de regrouper les composants (les programmes) d'une configuration. Nous utilisons aussi le type d'objet *Programme* qui a été défini dans le chapitre précédent (cf. chapitre 4 §1).

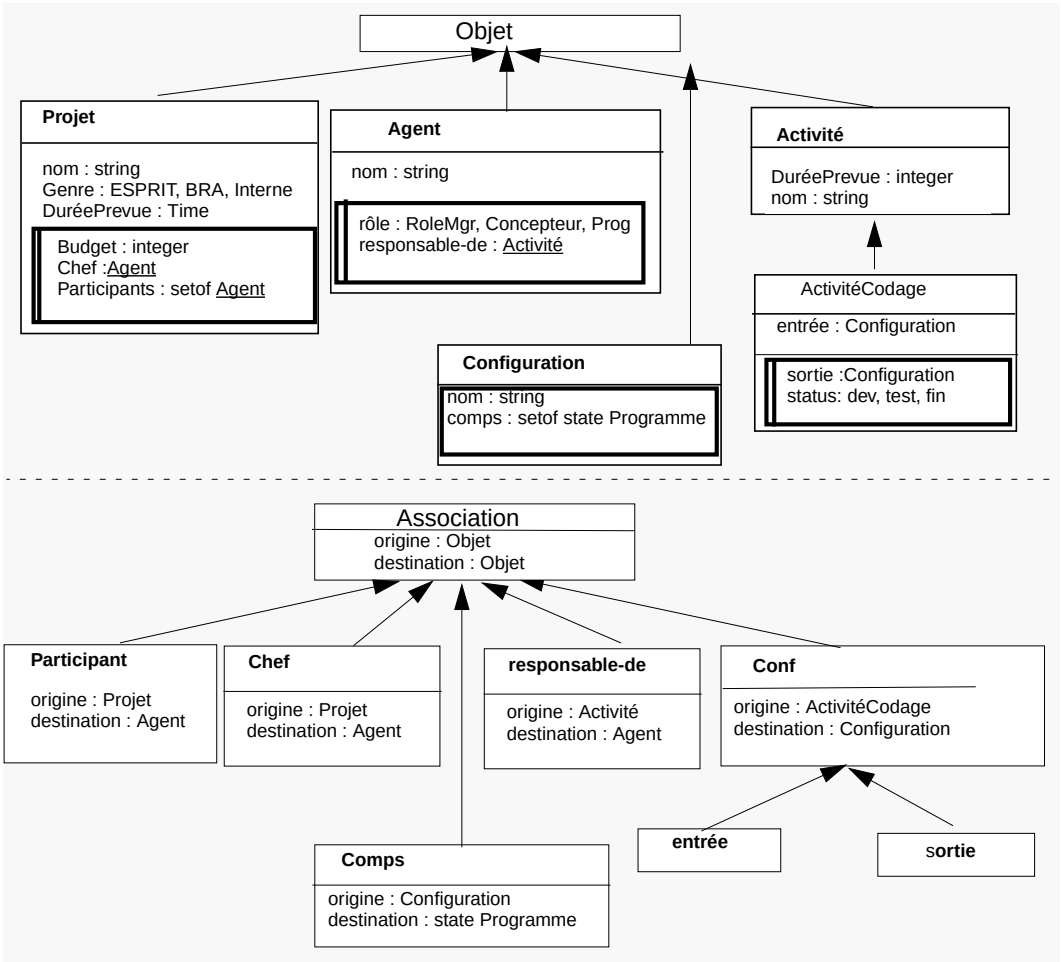


Figure 3 Le schéma de données de la base *Projets de fabrication de logiciel*.

A un instant donné, un agent participe à un seul projet (il peut être le chef), joue un seul rôle et peut être responsable d'une activité. Nous supposons que la base de *Projets* a été gérée depuis 1990.

Nous donnons ici quelques données de la base que nous compléterons au fur et à mesure de la présentation des exemples de requêtes.

Le projet appelé PBDT (Projet de développement d'une base de donnée temporelle) existe depuis 1990 (figure 4). Entre 1990-1992, le budget assigné était de 10Kf et est passé à 20Kf en 1993.

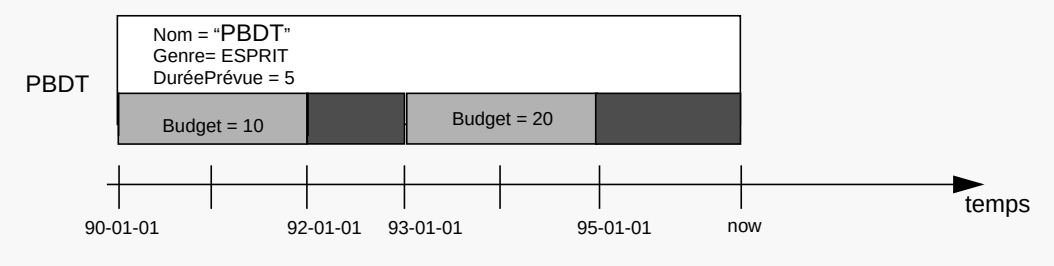


Figure 4 L'objet historique du projet PBDT de l'exemple.

L'association Chef permet d'indiquer, pour un projet, la personne qui en est le responsable. Ainsi, le chef du projet PBDT entre 1990-1993 et depuis 1995 est l'agent appelé Boss (figure 5). Entre 1993 et 1995, il a été chef d'un autre projet appelé GCL (gestion de configurations de logiciels).

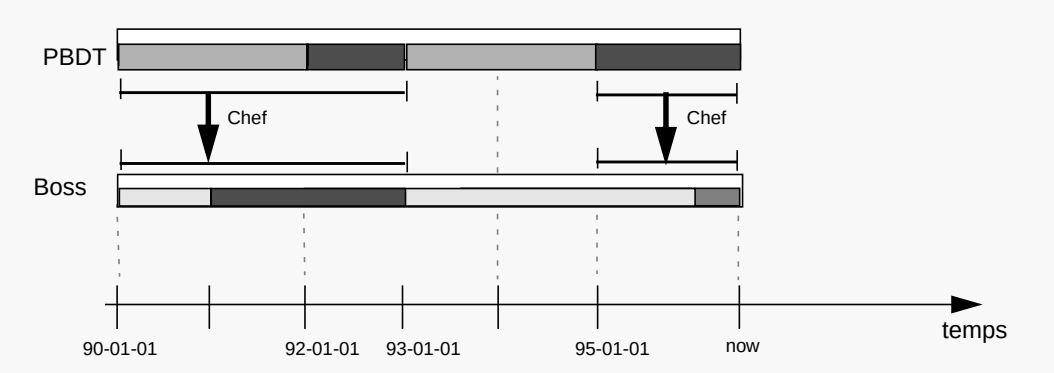


Figure 5 L'association *Chef* entre les objets historiques PBDT et Boss.

L'association Chef est de type historique. Cela permet d'indiquer, par exemple, que Boss a toujours été le chef du projet PBDT, excepté durant la période allant de 1993 à 1995. C'est dans ce cas l'utilisateur Boss2 qui le remplace (l'association Chef entre Boss2 et PBDT n'est pas montrée dans la figure 5).

3 L'accès aux objets

Les trois éléments principaux de **MOHA** (cf. chapitre 4) sont les états, les objet historiques et les associations. Cette section présente les opérateurs de base de **LOHA**

permettant l'accès aux informations liées aux deux niveaux de structuration : objet historique (cf. §3.1) ou état (cf. §3.2). L'utilisation d'associations est présentée dans la section §5.

3.1 Les objets historiques

La manipulation d'un objet historique se fait toujours à travers des références historiques (cf. chapitre 4 §3.2.1). On rappelle qu'une référence historique est définie par :

refHist = < oid-OH, PV >

où *oid-OH* est l'identificateur de l'objet historique et *PV* est la période de visibilité sur cet objet historique. Cette période de visibilité est définie par un élément temporel. Un élément temporel est un ensemble d'intervalles [Gad88]. Les opérations d'*union*, d'*intersection* et de *différence* sur les éléments temporels sont disponibles dans notre implantation.

La figure 6 montre l'exemple de la référence historique Boss-PBDT, définie ainsi :

Boss-PBDT = < Boss, {[90-01-01, 93-01-01[, [95-01-01, now[} >

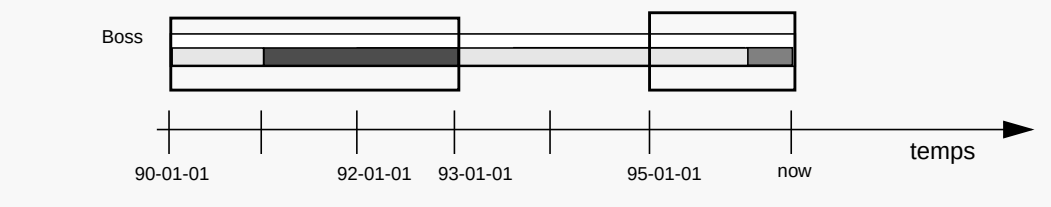


Figure 6 Une référence historique sur l'objet historique BOSS

Les opérations pouvant être faites au niveau d'une référence historique peuvent être la fonction d'accès à la période de visibilité ou des fonctions d'accès aux différents états que la référence recouvre.

Période de visibilité d'une référence historique

<référence Historique>.VisibilitySpan : < élément temporel>

a pour valeur l'élément temporel qui définit la période de visibilité de la référence historique. Par exemple,

Boss-PBDT.VisibilitySpan est égal à :

{[90-01-01, 93-01-01[, [95-01-01, now[}

Accès aux états

L'accès à un état particulier d'un objet historique se fait à travers l'opérateur unique "@". En outre, pour prendre en compte le statut particulier de certains états d'un objet historique, tels que le premier état et le dernier état, l'opérateur peut être exploité avec des symboles prédéfinis pour permettre ces accès particuliers.

L'opérateur "@"

L'opérateur **"@"** permet de désigner directement un état de l'objet. Les opérandes de cet opérateur sont la référence historique et un instant de temps.

<référence Historique>@t

Si la valeur de *t* est en dehors de la valeur de la période de visibilité de la référence historique alors l'expression produit la valeur Nil.

L'expression t peut correspondre à une date explicite, par exemple, PBDT@90-01-01, à une expression qui rend comme valeur une date, ou à un des symboles prédéfinis présentés ci-dessous.

Avec la référence historique suivante :

Boss-PBDT = < Boss, {[90-01-01, 93-01-01[, [95-01-01, now[} >

on peut par exemple avoir les accès suivants

Boss-PBDT@90-01-01 qui donne accès au premier état de Boss-PBDT, et

Boss-PBDT@94-01-01 qui retourne Nil.

Accès aux états particuliers

Les états particuliers se caractérisent par rapport à la période de visibilité imposée par la référence historique à l'objet historique.

- *le premier état visible de l'objet historique référence (symbole First) :*

  Boss-PBDT.First

  rend une référence au premier état visible de la référence historique Boss-PBDT. Ce qui signifie qu'elle rend le premier état de Boss visible à partir du premier intervalle de la période de visibilité de la référence historique Boss-PBDT.

- *le dernier état visible de l'objet historique référence (symbole Last) :*

  Boss-PBDT.Last

  rend une référence au dernier état visible de la référence historique Boss-PBDT. Ce qui signifie qu'elle rend le dernier état de Boss visible à partir du dernier intervalle de la période de visibilité de la référence historique Boss-PBDT.

L'ensemble d'états

Grâce à la méthode *AllStates*, nous pouvons transformer les états d'un objet historique visibles par la référence historique en un ensemble d'états. Par exemple,

PBDT.AllStates est égal à l'ensemble d'états :

{PBDT@90-01-01, PBDT@92-01-01, PBDT@93-01-01, PBDT@95-01-01}

3.2 Les états

Caractéristiques temporelles d'un état

TimeStart et TimeEnd sont deux fonctions qui, pour un état donné, rendent comme valeur respectivement l'instant de création de l'état et l'instant de création de l'état suivant. C'est-à-dire que l'intervalle [TimeStart, TimeEnd[correspond à la durée de l'état.

Par exemple, si *initPBDT* est un identificateur logique de l'état initial de l'objet historique appelé PBDT (cf. figure 4), nous avons :

initPBDT.TimeStart est égal à 90-01-01

initPBDT.TimeEnd est égal à 92-01-01

L'expression suivante montre l'intérêt de ce type de fonction lorsqu'elle est combinée avec l'opérateur "@" :

Boss@(PBDT.First.TimeStart)

qui exprime l'accès à l'état de l'objet historique Boss correspondant à l'instant où le projet PBDT a démarré.

L'accès aux attributs

Pour accéder aux attributs des états, nous utilisons l'opérateur "." (point). Si réf_état désigne un état (objet du type T) d'un objet historique, alors, l'expression :

réf_état.attr

rend comme valeur, la valeur de l'attribut *attr* de l'objet référencé par *réf_état.* Par exemple,

initPBDT.DuréePrévue est égal à 5 et,

initPBDT.Budget est égal à 10

Prédécesseur et successeur d'état

Appliquées à un état, les fonctions PreState et SuccState rendent respectivement son état prédécesseur et son état successeur. Si l'état demandé n'existe pas, la fonction rend la valeur Nil. Par exemple,

initPBDT.PreState est égal à Nil et,

initPBDT.SuccState est égal à l'état crée à l'instant 92-01-01

L'objet historique de l'état

Appliquée à un état, la fonction ObjHist rend une référence historique à l'objet historique auquel l'état appartient et la période de vie de l'objet pour période de visibilité de cette référence. Par exemple,

initPBDT.ObjHist est égal à la référence historique PBDT

3.3 Conversions de types

Les fonctions présentées précédemment se divisent en deux catégories selon qu'elles s'appliquent au niveau d'une référence historiques ou au niveau d'un état. Toutefois, pour faciliter l'usage de **MOHA**, un système de conversions de types permet de remplacer une référence historique par un état ou vice-versa. De même, on peut vouloir convertir une référence historique en un ensemble d'états.

Conversion état vers référence historique

Lorsqu'une valeur de type *état* apparaît dans une expression à l'endroit où une valeur de type *référence historique* est attendue, cet état est automatiquement converti en une référence historique. L'objet historique référencé est celui dont provient l'état et la période de visibilité de la référence correspond à la période de vie de l'état.

Conversion référence historique vers état

De même, si nécessaire, une référence historique sera convertie en un état correspondant au dernier état visible de la référence.

Avec cette règle de conversion, l'expression :

OH.attr

correspond à :

OH.attr = OH.Last.attr

3.4 Exemples

R1. Quel est le genre du projet PBDT ?

PBDT.genre retourne ESPRIT

R2. Quel est le budget du projet PBDT ?

PBDT.Budget retourne 20

R3. Quel est le budget courant du projet PBDT **?**

PBDT@Now.budget retourne 20

R4. Quel a été le premier rôle de l'utilisateur Boss ?

Boss@First.rôle retourne RôleMgr

R5. Quel était le rôle de l'utilisateur Boss quand le projet PBDT **a commencé ?**

Boss@(PBDT@First.TimeStart).rôle retourne RôleMgr

4 Filtres

Dans cette section nous présentons quelques opérations de base sur les références historiques et sur les ensembles. Intuitivement, la notion de filtre fait référence à une idée de sélection. Cependant, la sémantique précise de cette opération n'est pas la même s'il s'agit de filtrer une référence historique ou de filtrer un ensemble.

4.1 Filtres temporels

Un filtre temporel est appliqué sur une référence historique pour restreindre la période de visibilité sur l'objet historique aux intervalles qui vérifient un prédicat donnée. Par exemple, si PBDT est la référence historique suivante :

<PBDT, { [90-01-01, now[} >

l'expression,

PBDT[budget >10]

est évaluée de la façon suivante. Le prédicat budget > 10 est testé sur chaque état visible à travers la référence. Le résultat de l'expression est une nouvelle référence historique dont la période de visibilité est :

<PBDT, { [93-01-01, now[} >

Le calcul de la nouvelle période de visibilité est réalisé par l'intersection des intervalles de départ avec les intervalles correspondant aux états vérifiant le filtre. Dans l'exemple de la figure 7, les états e_1 et e_2 sont visibles dans l'intervalle $I_{initial}^1$ et les états e_3, e_4 et e_5 sont visibles dans l'intervalle $I_{initial}^2$. Supposons que les états e_1, e_3 et e_5 vérifient le filtre, alors, les intervalles résultat de la période de visibilité de l'objet *O* sont :

$$I_{final}^{1} = (I_{initial}^{1} \cap e_1.lifespan) \cup (I_{initial}^{2} \cap e_1.lifespan)$$

$$I_{final}^{2} = (I_{initial}^{1} \cap e_3.lifespan) \cup (I_{initial}^{2} \cap e_3.lifespan)$$

$$I_{final}^{3} = (I_{initial}^{1} \cap e_5.lifespan) \cup (I_{initial}^{2} \cap e_5.lifespan)$$

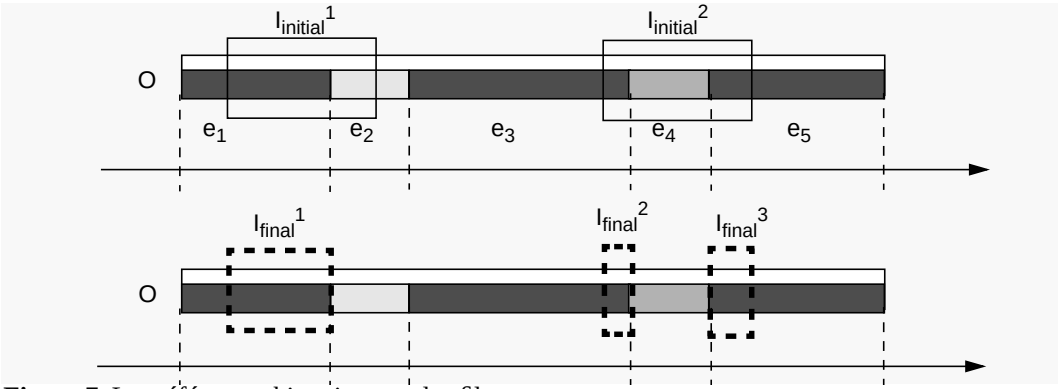


Figure 7 Les références historiques et les filtres.

Formellement, l'application du filtre défini par le prédicat *p* sur la référence historique *rh*, noté *rh[p]*, rend une référence historique *rh'* dont la période de visibilité est calculée de la façon suivante :

$$rh'.VisibilitySpan = rh.VisibilitySpan \cap \left(\bigcup_{e_i \in p\text{-}states} \{e_i.VisibilitySpan\} \right)$$

où, l'ensemble *p-states* correspond à l'ensemble d'états de *rh* vérifiant le prédicat *p* :

$$p\text{-}states = \{ e \mid e \in rh.ObjHist.Allstates \text{ et } p(e) \}$$

4.2 Filtre et image sur les ensembles

Sur les ensembles nous supposons avoir deux opérateurs de base : filtre et image. Après avoir présenté la sémantique de ces opérateurs, nous montrons comment ils sont utilisés à travers des expression **LOHA**.

Opérateurs de base

Filtre

Nous notons S | P le fait de filtrer l'ensemble S avec le prédicat P. La sémantique de cette opération est la suivante :

$$S \mid P = \{ x \mid x \in S \text{ et } P(x) \}$$

signifiant que le filtrage rend l'ensemble des éléments de S qui vérifient le prédicat P.

Par exemple si l'on veut sélectionner les éléments inférieurs à 4 dans l'ensemble d'entiers {2,3,4}, il suffit d'exprimer le filtre suivant :

{2,3,4}| (this<4) *où this représente un élément de l'ensemble*

dont le résultat est l'ensemble {2,3}.

Image

Nous notons S map F le fait de calculer l'image de l'ensemble S par la fonction F. La sémantique de cette opération est la suivante :

$S\ map\ F = \{x' / x'=F(x) \wedge x \in S\}$

Par exemple si nous voulons ajouter 2 à chaque élément de l'ensemble d'entiers {2,3,4}, il suffit d'appliquer le calcul d'image suivant :

{2,3,4} map (this+2) *où this représente un élément de l'ensemble*

qui rend pour résultat l'ensemble {4, 5, 6}.

Expression LOHA faisant appel aux opérateurs de base

Dans ce qui suit, nous présentons comment des expressions LOHA impliquant des ensembles d'état ou de références historiques sont interprétées en termes des opérateurs filtre et image précédemment présentés.

Expression de filtre sur un ensemble

Dans **LOHA**, l'interprétation de l'expression de filtre

S [Filtre]

dépend du type de l'ensemble S : ensemble d'états ou ensemble de références historiques.

- *ensemble d'états*

Si nous avons un ensemble d'*états*, par exemple, celui déterminé par PBDT.AllStates, l'expression dans **LOHA**:

PBDT.AllStates [budget <10]

correspond à l'application de l'opérateur filtre | suivante :

PBDT.AllStates | this.budget <10

et donne donc comme résultat, l'ensemble d'états qui vérifient le prédicat budget <10.

- *ensemble de références historiques*

Si nous avons un ensemble de références historiques S_{rh}, l'expression LOHA :

S_{rh} [P] où P est un filtre temporel,

s'interprète de la façon suivante :

S_{rh} *map P*

c'est-à-dire, le calcul de l'image du filtre temporel P sur l'ensemble de références historiques S_{rh} : $\{ x' \mid x' = x\,[P] \wedge x \in S_{rh} \}$.

Par exemple, avec *LesProjets*, l'ensemble des références historiques de type *historiqueof Projet*, l'expression **LOHA** suivante :

```
LesProjets [DuréePrévue > 2 and genre = ESPRIT]
```

rend l'ensemble de projets ESPRIT pour lesquels la valeur de l'attribut *DuréePrévue* est supérieure à 2. C'est-à-dire :

$\{ p' \mid p' = p[p.DuréePrévue > 2 \wedge p.genre = ESPRIT]$ *pour chaque* $p \in LesProjets \}$

Opérateurs LOHA appliqués à un ensemble

L'opérateur *point* et les opérateur de navigation (cf. §5.2), appliqués sur des ensembles (d'états ou de références historiques), sont interprétés de la façon suivante.

Si *S* est un ensemble (d'états ou de références historiques) et *attr* est un attribut défini dans son type, alors l'expression LOHA :

```
S.attr
```

rend pour résultat :

S *map .attr* $= \{v \mid v$ *est la valeur de* $s.attr$ *et* $s \in S\}$

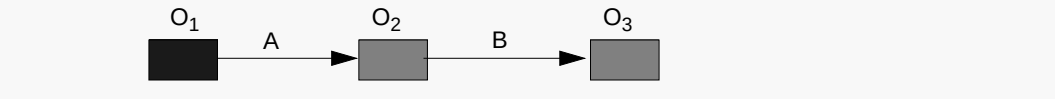
5 La navigation

5.1 Les expressions de chemins

Une expression de chemins décrit un parcours dans un graphe et rend pour valeur les objets terminaux de ce parcours. Les graphes constitués par des objets et associations **MOHA** sont hétérogènes et la sémantique du parcours dépend des catégories d'associations à traverser (historique ou non-historique).

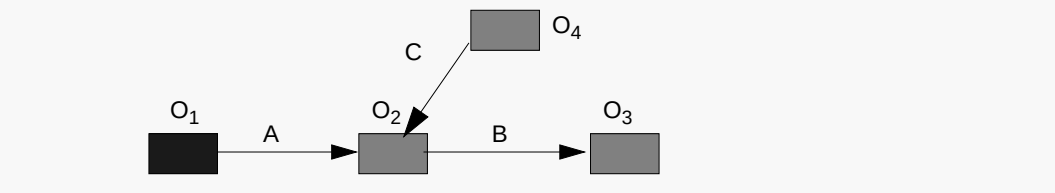
Dans cette section nous présentons une définition d'expression de chemins de façon générale sans se préoccuper de la catégorie des associations. Les spécificités de chaque catégorie d'associations, historique et non historique, sont respectivement présentées dans la section §5.2. Dans les exemples suivants, les objets considérés peuvent être des états ou des objets historiques.

Par exemple, dans la figure ci-dessous, nous avons une association de type A entre l'objet O_1 et l'objet O_2 et puis, une association de type B entre O_2 et O_3. L'expression de chemins: $O_1 \rightarrow A \rightarrow B$

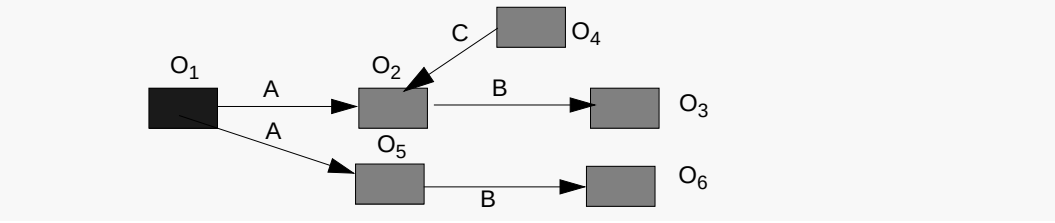


rend comme résultat l'objet O_3. L'objet O_1 est appelé la *racine* de l'expression, tandis que A et B sont des noms de type association. L'ensemble d'objets $\{O_1, O_2, O_3\}$ constitue un *chemin* qui satisfait l'expression $O_1 \rightarrow A \rightarrow B$.

Sur le même exemple, l'expression de chemins : $O_3 \leftarrow B \leftarrow A$ rend comme résultat l'objet O_1. Dans ce cas, nous naviguons dans le sens inverse des associations. Dans une même expression de chemins, les deux sens de navigation peuvent être combinés. Par exemple (figure suivante), l'expression $O_1 \rightarrow A \leftarrow C$ rend comme résultat l'objet O_4.



Dans **MOHA**, un objet peut être origine et/ou destination de plusieurs associations du même type. Par exemple, dans la figure suivante, c'est le cas de l'objet.



L'objet O_1 est origine de deux associations du type A. La première a comme destination l'objet O_2 et la deuxième l'objet O_5. Dans ce cas, les chemins qui satisfont l'expression de chemins $O_1 \rightarrow A \rightarrow B$ sont : $\{O_1, O_2, O_3\}$ et $\{O_1, O_5, O_6\}$. Le résultat de l'expression sont les objets terminaux de chaque chemin, c'est-à-dire l'ensemble d'objets $\{O_3, O_6\}$.Si T_{B_dest} est le type des objets qui peuvent être destinations des associations du type B, alors le résultat de l'expression de chemins $O_1 \rightarrow A \rightarrow B$ est un ensemble d'objets du type T_{B_dest}.

De façon générale, la définition suivante peut être donné.

Si Racine représente un ensemble de références aux objets et l'association *A* a pour domaine :

$$T_{Aorigine} \rightarrow T_{Adest}$$

alors l'*expression de chemins E* définie par :

$E = Racine \rightarrow A$ (Resp. $E = Racine \leftarrow A$)

rend comme résultat un *ensemble* d'objets du type T_{Adest} (Resp. $T_{Aorigine}$).

5.2 Navigation à travers des associations

Dans cette section les opérations de navigation sont présentées pour chaque type d'association : les associations non historiques et les associations historiques.

5.2.1 Associations non historiques

La navigation via des associations non historiques portant sur des états ne posent pas de problèmes, elle se fait de la façon expliquée dans la partie précédente (cf. §5.1). Par contre, une association non historique impliquant un objet historique signifie que tous les états constituant l'objet historique sont eux aussi impliqués dans l'association, c'est-à-dire que l'association est *partagée* par les états (cf. chapitre 4 §3.1). Cette propriété permet d'exploiter une association partagée pour naviguer à partir un état de l'objet historique.

Par exemple, dans la figure 8, les associations A et B sont des associations impliquant l'objet historique O_1. Elles sont partagées par tous les états de O_1.

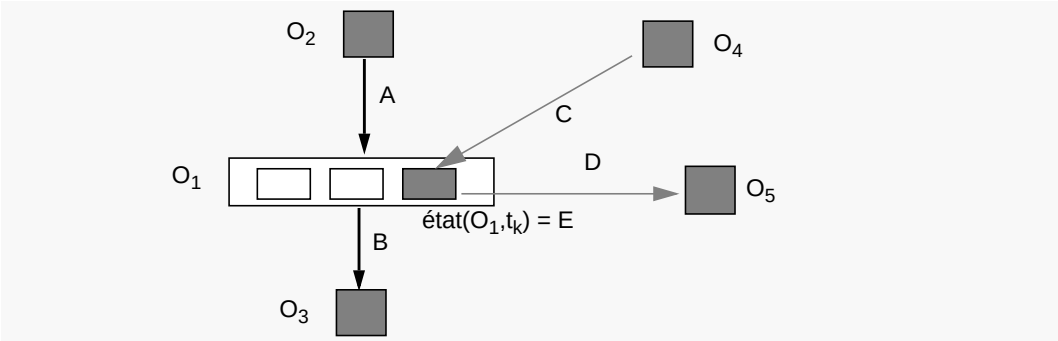


Figure 8 Le partage des associations non-historiques.

En partant de l'état appelé *E*, l'expression suivante :

$E \leftarrow A$

rend comme résultat l'objet O_2. De même, l'expression :

$E \rightarrow B$

rend comme résultat l'objet O_3.

De même, l'expression $O_4 \rightarrow C \rightarrow B$ rend comme résultat l'objet O_3.

5.2.2 Associations historiques

Considérons maintenant le cas de la navigation via des associations historiques. De telles associations sont établies entre deux objets historiques (cf. chapitre 4 §3.2). Comme la période de validité d'une association ne coïncide pas forcément avec la totalité de la période de vie des objets liés, la notion de référence historique a été introduite pour permettre de considérer des vues partielles d'objet historique. La valeur de l'attribut

origine et la valeur de l'attribut *destination* d'une instance d'association historique sont des références historiques. La période de visibilité des références historiques détermine la période de validité de l'association. Par définition, la période de visibilité de la référence historique correspondant à l'origine est la même que celle de la destination.

Dans ce qui suit, nous présentons quelques exemples de requêtes impliquant des associations historiques. A la fin de cette partie, nous présentons, de façon générale, l'interprétation de la navigation à travers les associations historiques.

Exemples

Tous les exemples de requêtes sont basés sur l'exemple présenté dans la section §2 et sur les données de la figure 9. L'objet du type *Agent*, appelé Boss, a été responsable de deux activités (pendant sa période de vie) : l'activité appelée GMEM (développement du module de gestion de mémoire) entre 1990 et 1992 et l'activité IHM (développement de l'interface homme-machine) par la suite.

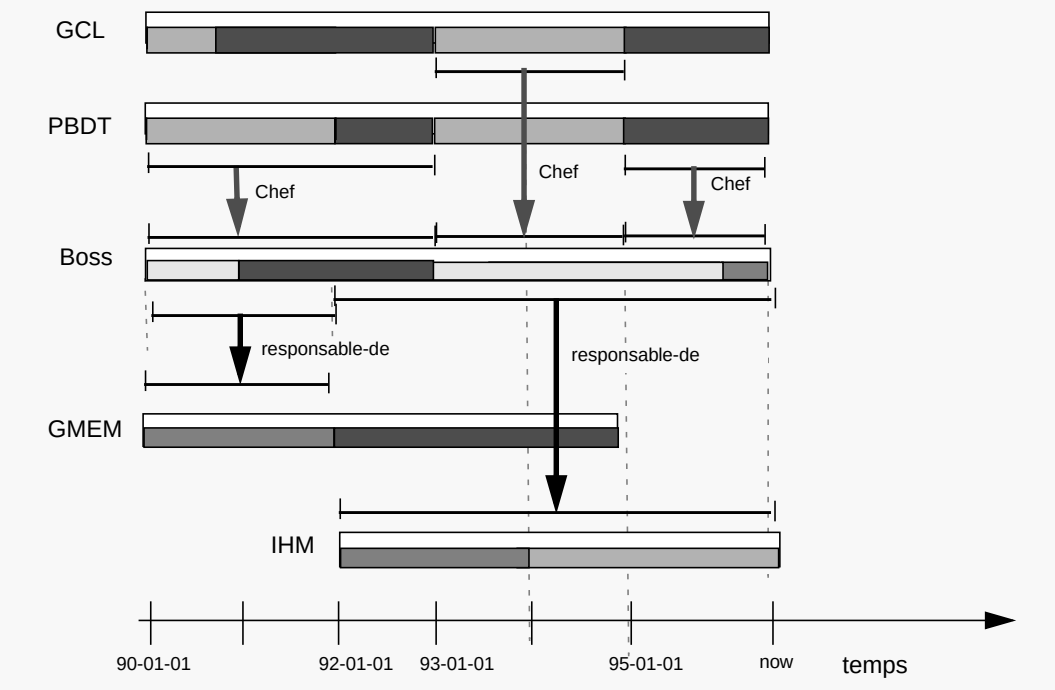


Figure 9 Les activités dont l'agent Boss à été responsable pendant sa période de vie.

R6. Quel est le dernier budget des projets pour lequel l'utilisateur Boss a été chef ?

Boss←Chef.budget

Boss←Chef rend pour valeur un ensemble de références historiques :

$\{PBDT_{Boss}, GCL_{Boss}\}$ *où*

$PBDT_{Boss}$ = <*PBDT*, {[*90-01-01,93-01-01*[, [*95-01-01,now*[}> *et*

GCL_{Boss} = <*GCL*, { [*93-01-01, 95-01-01*[} >

L'opérateur point s'appliquant à chaque référence historique (cf. §4.2),

$\{PBDT_{Boss}.budget, GCL_{Boss}.budget\}$

et suivant les règles de conversion de types (cf. §3.3), la requête est automatiquement convertie en :

{PBDT.Last.budget, GCL.Last.budget}

R7. De quelles activités était responsable l'agent Boss **lorsqu'il était chef du projet** PBDT **?**

PBDT→chef[nom = "Boss"]→responsable-de.nom

Une fois la partie de l'expression de chemins PBDT→chef[nom = "Boss"] évaluée, le résultat est le singleton contenant la référence historique :

{ <Boss, { I_1, I_2 } > } *avec* I_1 = [90-01-01, 93-01-01[et I_2 = [95-01-01, now [

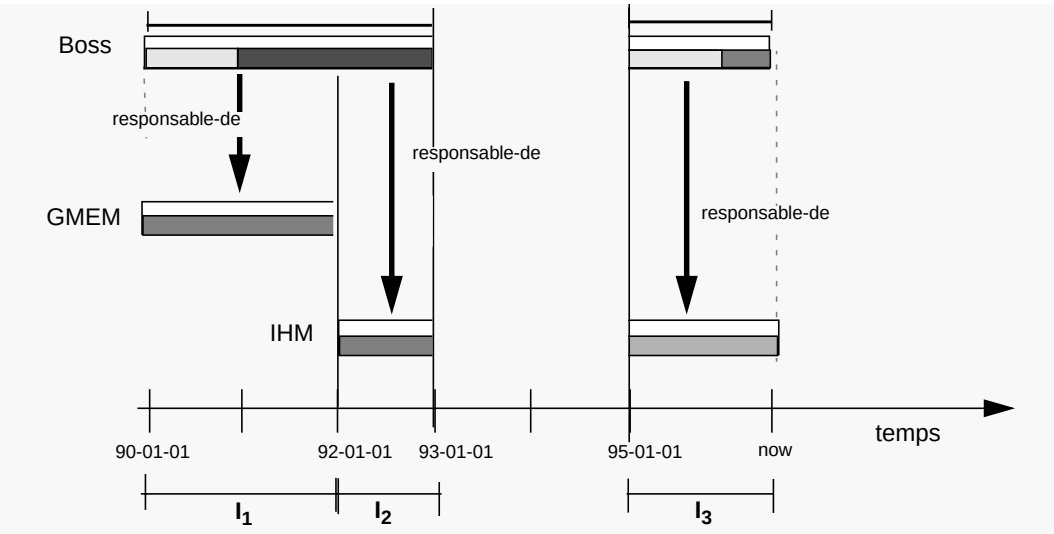


Figure 10 Résultat de la requête PBDT→chef[nom = "Boss"]→responsable-de

Si à partir de cette référence historique on traverse l'association responsable-de on obtient pour résultat l'ensemble de références historiques sur des objets du type *Activité* (cf. figure 10):

{ <oid-GEM, {I_1}>, <oid-IHM, {I_2, I_3} >} avec I_1 = [90-01-01, 92-01-01[et I_2 = [92-01-01, 93-01-01[

et, I_3 = [95-01-01, now [

Le résultat final de la requête est donc,

{"GMEM", "IHM"}

Généralisation

Supposons que nous ayons l'expression de chemins E_n où A est une association historique.

$E_n = E_{n-1} \leftrightarrow A$

Le résultat de l'expression E_n est un ensemble de références historiques sur des objets du type soit $T_{Aorigine}$, soit T_{Adest}. Le type dépend du sens choisi pour naviguer à travers l'association : soit de l'origine vers la destination (→), soit de la destination vers l'origine (←).

La spécificité de la navigation à travers des associations historiques provient du calcul des nouvelles périodes de visibilité pour les références historiques. Chaque fois que l'on traverse une association historique, on réduit la période de visibilité. Ce qui a pour effet de réduire et/ou éclater les intervalles qui décrivent cette période.

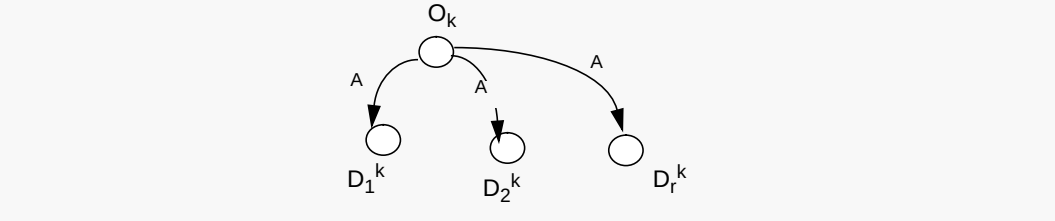
De façon générale, supposons que les références historiques contenues dans l'ensemble E_{n-1} est le suivant :

$$E_{n-1} = \{ \text{rh-}O_1 , \text{rh-}O_2 , \ldots , \text{rh-}O_m \}$$

Pour chaque $\text{rh-}O_k$ de E_{n-1}, on considère :

$$\text{rh-}O_k \rightarrow A$$

supposons que $ObjHist(\text{rh-}O_k) = O_k$ est un des objets de l'ensemble de départ et qu'il y a plusieurs instances d'associations A qui ont pour origine cet objet (figure ci-dessous).



tous les objets destinations D^k font partie du résultat E_n. La période de visibilité de ces objets est définie par :

$$\text{rh-}O_k = < O_k , PVisibilité_{Ok} >$$

nous dénotons un des objets destination de l'association A à partir de l'objet O_k par :

$$D_j^k$$

et, la période de validité de l'association A entre O^k et D_j^k par :

$$PValidité (A (O_k, D_j^k))$$

alors, la nouvelle période de visibilité de l'objet D_{i-j} est défini par :

$$PVisibilité_{Dj} = PVisibilité_{Ok} \cap PValidité (A (O_k, D_j^k))$$

Il faut prendre en compte une dernière chose. Un objet D_j^k peut être, plusieurs fois, destination de l'association *A* mais avec des origines différentes contenues dans l'ensemble de départ. Alors, il faut faire l'union des éléments temporels de chaque D_j^k.

Par exemple, dans l'expression PBDT→chef[nom = "Boss"], la référence historique de départ est l'objet historique PBDT et sa période de vie :

$$PVisibilité_{PBDT} = \{[90\text{-}01\text{-}01, now[\}$$

La période de validité de l'association *chef* entre l'objet historique PBDT et l'objet Boss est :

$$PValidité (chef (PBDT, Boss)) = \{ [90\text{-}01\text{-}01, 93\text{-}01\text{-}01[, [95\text{-}01\text{-}01, now [\}$$

La nouvelle référence historique créée sur l'objet Boss est calculée en intersectant les éléments temporels :

newref-Boss = < *oid-Boss*, $PVisibilité_{Boss}$> telle que :

$PVisibilité_{Boss}$ = $PVisibilité_{PBDT} \cap PValidité(chef\ (PBDT, Boss))$

= { [90-01-01, 93-01-01[, [95-01-01, now [}

5.2.3 Combinaison entre les deux catégories d'associations

Pour compléter les exemples de navigation, il faut prendre en compte les expressions de chemins combinant des associations historiques et non historiques. Par exemple, supposons que nous ayons l'expression de chemins suivante :

Boss→responsable-de[nom = GMEM]→sortie

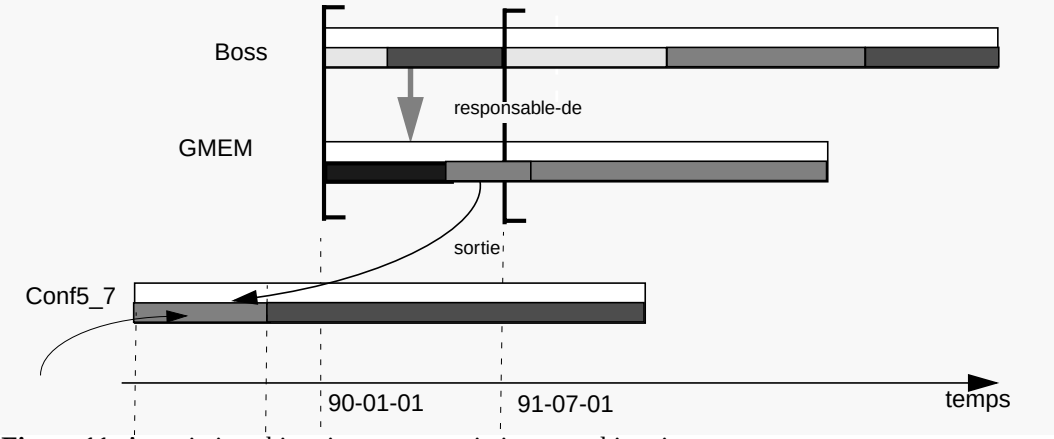


Figure 11 Associations historiques et associations non-historiques.

La figure 11 montre les différentes associations entre les objets. Sur cet exemple, la première association (*responsable-de)* est historique, et donc le résultat de son parcours est une référence historique. Le problème est de savoir comment va être interprétée l'association suivante (*sortie)* qui n'est pas une association historique.

Par définition, une association non historique est établie entre objets historiques et/ou états. Or, une référence historique peut contenir plusieurs morceaux d'états. En utilisant la règle de conversion de types, le dernier état impliqué dans la référence historique sera pris en compte pour traverser l'association.

6 Conclusion

Nous avons présenté dans ce chapitre, le langage d'interrogation déclaratif associé au modèle **MOHA**. Ce langage repose sur deux notions fondamentales : les expression de chemins et les filtres. Une expression **LOHA** définit un parcours (d'un graphe) de la base à travers les associations historiques ou non historiques.

LOHA s'est inspiré de divers travaux des domaines des bases de données à objets, des bases de données temporelles et aussi de la gestion de versions. **LOHA** est un complément du langage de programmation du gestionnaire d'objets. Les expressions **LOHA** peuvent être utilisées dans le code impératif (par exemple, dans une méthode).

Dans le cadre du génie logiciel, l'idée de base de la navigation était déjà présente dans l'utilisation du gestionnaire de fichiers. En effet, le symbole "/" permet d'exprimer un parcours à travers l'arborescence de répertoires pour accéder aux fichiers.

Récemment, dans le domaine de bases de données orientées-objets, plusieurs chercheurs se sont intéressés à la définition de langages d'interrogation basés sur la notion d'*expressions de chemins*[1]. Le langage XSQL [KKS92] est l'un des travaux de référence en ce qui concerne les expressions de chemins. Ce langage permet d'exprimer des requêtes sophistiquées d'une façon très concise. De plus, il est adapté aux structures imbriquées que procure le modèle à objets et évite une consultation systématique des schémas de données à chaque interrogation de la base.

Mise à part sa simplicité, il y a deux raisons primordiales pour lesquelles nous avons défini un langage navigationnel. La première provient du fait que, dans **MOHA**, les associations jouent un rôle fondamental pour structurer les objets et pour définir des agrégats d'objets. La deuxième provient du fait que, dans les environnements logiciels, il est courant que les objets aient un nom externe et qu'ils soient accéder directement par ce nom. Les accès à la base se font donc en général à travers un objet particulier.

Nous avons repris cette idée d'*expression de chemins* de telle façon que l'on puisse naviguer à travers n'importe quel type d'association. Notre langage ne possède pas pour l'instant de variables, nous nous sommes restreints à rendre pour valeur les objets (ou les valeurs des objets) terminaux des chemins.

Dans les travaux de E. Sciore [Sci91] [Sci93] sur le versionnement logique des objets (cf. chapitre 2), on retrouve l'idée de filtre pour sélectionner un sous-ensemble de versions d'un objet parmi un ensemble de versions. Nous avons inclus cette fonctionnalité dans notre langage en permettant d'appliquer de filtres aux objets historiques. Les filtres sur les références historiques permettent de restreindre la période de visibilité sur les objets historiques. De plus, en permettant d'examiner le contenu des objets à travers desquels on navigue, les filtres enrichissent les possibilités d'expressions de chemins.

Les aspects historiques sont inclus dans **LOHA** de façon uniforme et simple. L'utilisation des associations historiques ou non historiques est traitée de façon uniforme. Ce qui simplifie l'usage du langage.

1. *path expressions* dans la littérature anglophone.

Chapitre 6

Les annotations et les événements

Ce chapitre présente l'intégration de la notion d'*annotation* dans **MOHA**. Dans le contexte des environnement guidés par les procédés, les annotations vont permettre de représenter et de gérer l'information historique liée à l'exécution des activités de façon complémentaire à celle de l'évolution des objets. En particulier, nous montrons comment les annotations peuvent être utilisées pour représenter l'historique d'événements. Nous proposons une extension de la notion d'événement dans **MOHA**.

Dans la section §1 nous présentons le formalisme de règles actives de notre modèle de référence. Plusieurs des nos propositions supposent que les règles actives sont disponibles et les utilisent. Aussi, nous ne prétendons pas faire une présentation exhaustive de ce formalisme mais seulement en donner les concepts principaux. Dans la section §2 nous présentons l'intégration de la notion d'annotation dans **MOHA**. Dans la section §3, nous présentons comment les annotations et le langage **LOHA** peuvent être exploités pour étendre la notion d'événement dans le modèle.

1 Les règles actives

Dans le modèle de référence, inspiré du système Adèle, les règles actives jouent un rôle fondamental pour la programmation de procédés de fabrication de logiciel. Elles sont aussi utilisées pour définir et vérifier des contraintes de cohérence de données.

Pour plus de détail sur le modèle des règles actives dans le système Adèle, on pourra consulter [BE87] [Bel88] [Cas94]. Nous présentons d'abord les éléments de définition des règles actives et ensuite le modèle d'exécution.

1.1 Définition

Les règles actives peuvent être définies dans les types d'objets et dans les types d'associations. Dans le premier cas, elles réagissent à des opérations sur les objets et, dans le deuxième cas, aux opérations aussi bien sur l'association que sur ses objets origine et destination.

Une règle active est de la forme événement-condition-action (<E,C,A>) et la sémantique est la suivante : si l'événement est détecté et la condition vérifiée, alors l'action est exécutée. Par la suite nous présentons brièvement chaque composant.

Les événements

Les événements sont générés par des appels de méthodes sur des objets. L'exécution d'une méthode donne lieu à deux événements correspondant à deux instants particuliers de l'exécution de la méthode.

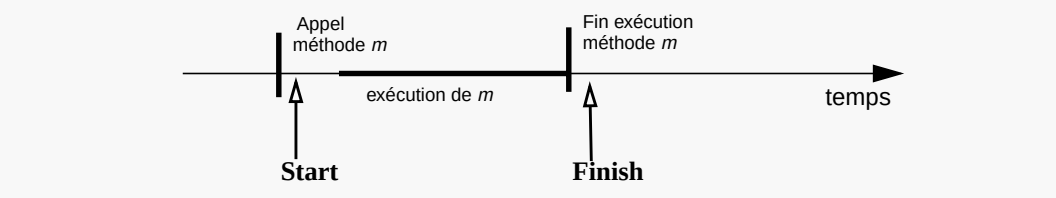


Figure 1 Définition des événements par rapport à l'exécution d'une méthode.

Les clauses *Start* et *Finish* permettent de spécifier ces deux instants. *Start* détermine l'instant après l'appel de la méthode précédant le début d'exécution et *Finish* l'instant juste après la fin de l'exécution de la méthode (figure 1).

Un type d'événements est toujours défini dans un type d'objet ou dans un type d'association. La définition d'un type d'événement comporte aussi une priorité qui est utilisée lors de l'activation de règles (cf. §1.2). Plus précisément un type d'événement, identifié par *idevt*, est défini de la façon suivante :

idevt : **Start**|**Finish** <méthode> [**Priority** <nn>]

nn possède une valeur entière nn = 1, .. ,99. Plus la valeur est grande, plus la priorité est importante.

Par exemple, le type d'événement id_evt_1, défini par :

id_evt_1 : **Start** delete, **Priority** 99

sera notifié avant l'exécution de la méthode *delete*.

Dans le cas des associations, la définition du type de l'événement doit spécifier si l'on s'intéresse à une méthode exécutée sur l'objet origine, l'objet destination ou sur l'association elle-même.

idevt : **Start**|**Finish** <méthode> [IN **Origin**|**Destination**], **Priority** nn

Par exemple, le type d'événement défini dans un type d'association *A*:

Deletecomposite : **Start** del_objet IN **Origine**, **Priority** 99

sera signalé, avant l'exécution de la méthode del_objet sur un objet origine d'une instance d'association *A*.

La condition

La condition est une expression booléenne ne comportant aucune opération de mise à jour sur les données de la base. Cette expression peut concerner l'objet sur lequel la méthode a été appelée mais aussi d'autres objets et associations de la base. Les expression booléennes peuvent être definies sous la forme de requêtes sur la base. Ces requêtes se font à l'aide d'un langage de meta-substitution d'expressions que nous ne détaillerons pas ici [Ade93].

L'action

L'action représente le traitement à effectuer lorsque l'événement a été détecté et la condition a été vérifiée. Ce traitement peut inclure des appels de méthodes et ainsi, provoquer l'activation d'autres règles actives (activation en cascade). L'action est écrite dans le langage de programmation impératif.

L'héritage

Les règles actives sont héritées à travers la hiérarchie de spécialisation et elles ne peuvent pas être redéfinies dans les sous-types d'objets.

1.2 Modèle d'exécution

L'activation des règles est guidée par les opérations réalisées sur les objets et les associations de la base. Nous verrons, par la suite, quelles sont les règles à activer quand un événement est détecté, où elles seront activées (par rapport à la transaction en exécution) et dans quel ordre si l'on a plusieurs règles à activer en même temps.

Les règles à activer

L'instance sur laquelle une méthode a été invoquée est appelée *l'instance courante*. Cette instance joue un rôle particulier car elle sert à déterminer les règles qui seront activées. Les règles recherchées sont définies dans le type de *l'instance courante*.

Lorsque *l'instance courante* est un objet qui est origine ou destination d'une ou plusieurs associations, la recherche sera aussi effectuée dans la définition des types de ces associations. Par exemple, dans la figure 2, l'objet *O* fait partie des instances d'associations A_1, A_2 et A_3. Si l'objet *O* est l'instance courante alors, les règles à activer seront recherchées dans le type de *O*, mais aussi dans le type des associations A_1, A_2 et A_3.

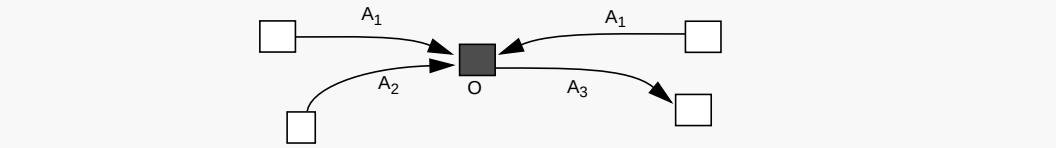


Figure 2 Les règles actives et les associations.

Le contexte de l'événement

Le contexte de l'occurrence d'un événement est constitué de l'ensemble des informations présentes au moment de l'apparition de l'événement. Ce contexte correspond à :

- la méthode courante et ses paramètres,
- l'instance courante (l'objet ou l'association)

Ces informations sont transmises aux règles que l'événement va activer. C'est-à-dire que la condition et l'action de la règle activée peuvent accéder aux valeurs de la méthode courante et de l'instance courante. En particulier, dans le cas ou l'instance courante est une association, l'origine et à la destination de l'association peuvent être accédées (cf. exemple).

Les modes d'activation

Les modes d'activation des règles actives permettent de spécifier les points d'activation des règles par rapport à l'occurrence de l'événement déclencheur et à l'exécution des transactions du système.

Une transaction est une unité de travail composée d'une ou plusieurs opérations (méthodes) sur des données de la base. Elle garantit l'*atomicité* de l'exécution, préserve la *cohérence* des données, assure l'*isolation* de l'exécution et la *persistance* des modifications faites [EN89]. Une transaction possède un point de démarrage et un point de validation.

Il y a deux modes d'activation : le mode *immédiat* et le mode *différé*. Les points d'activation des règles *immédiates* coïncident avec l'occurrence des événements déclencheurs. Dans ce cas, une fois qu'un événement est détecté, les règles associées seront activées *immédiatement* dans la transaction courante. Le mode d'activation *différé* spécifie que le point d'activation des règles est à la fin de la transaction (c'est-à-dire après la validation ou l'annulation de la transaction) dans laquelle la méthode, correspondant à l'événement, a été exécutée. Les règles correspondantes sont exécutées à la fin de la transaction courante, dans des transactions séparées et indépendantes. Les points d'activation des règles sont montrés dans la figure 3.

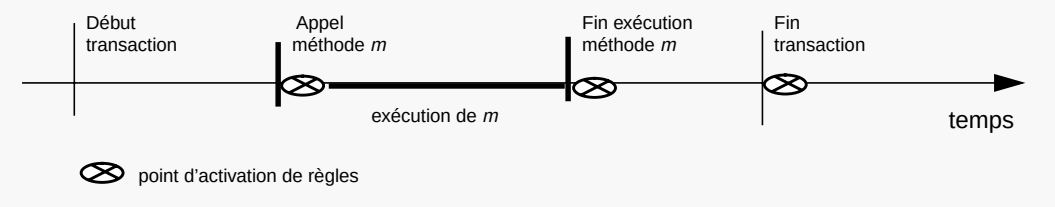


Figure 3 Les points d'activation des règles par rapport à la transaction dans laquelle la méthode, correspondant à l'événement, a été exécutée.

L'ordre d'activation

A chaque point d'activation, le système cherche l'ensemble des règles à activer. Ces règles sont ordonnées en fonction des niveaux de priorité specifiés dans les types des évènements qui les activent. Elles sont exécutées séquentiellement. L'exécution des actions déclenchées peut générer des nouveaux événements qui activent d'autres règles.

Un exemple

Grâce aux règles actives, l'utilisateur peut définir des types de composition avec des sémantiques particulières. Par exemple, la *composition partagée* peut être définie de la façon suivante :

a. Les composantes ne sont pas exclusives, c'est-à-dire qu'un objet peut être destination de plusieurs associations du type composition partagée.

b. Quand un composite est effacé ses composant le sont aussi s'ils ne sont pas destination d'une autre association du type composition partagée.

Dans l'exemple de la figure 4, l'objet O_1 a deux composant O_3 et O_4, tandis que l'objet O_2 a un composant O_4 qu'il partage avec l'objet O_1. D'après la propriété b), la destruction du composite O_1 doit entraîner la destruction de l'objet O_3 mais pas celle de l'objet O_4 car il est composant d'un autre objet.

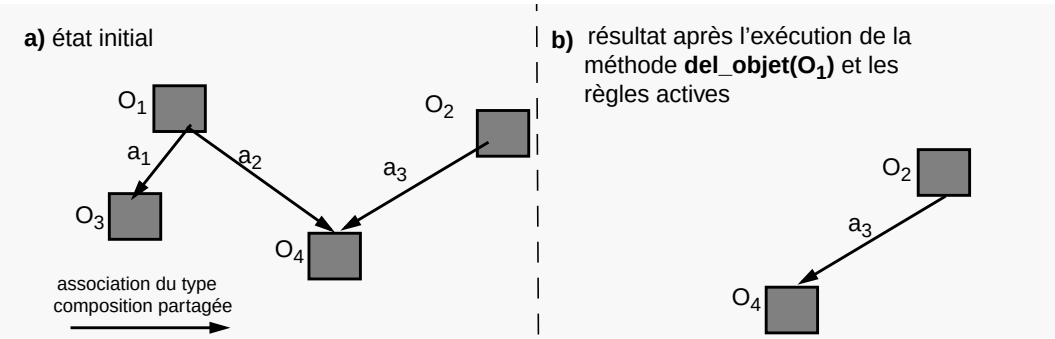


Figure 4 Exemple des objets composites avec une relation de composition du type *composition partagée.*

Nous avons défini la méthode *del_objet* dans le type *Objet* ainsi que deux règles actives dans le type de l'association *Composition Partagée.* L'une de ces règles permet de propager la méthode *del_objet* tandis que l'autre permet d'annuler la méthode

del_objet si elle s'applique sur un composant appartenant à un autre objet composite. Les règles utilisent les événements suivants :

a. *Deletecomposite* : **Start** del_objet IN **Origine**

"Avant l'exécution de la méthode *del_objet* sur l'objet origine de l'association"

b. *Deletecomposant :* **Start** del_objet IN Destination

"Avant l'exécution de la méthode *del_objet* sur l'objet destination de l'association"

Nous définissons la règle différée R_1 et la règle immédiate R_2 :

R_1 *:* ON *Deletecomposite* DO **Deffered** [1] del_objet (!destination)[2]

R_2 *:* ON *Deletecomposant* DO abort();

Dans la figure 4, la partie b) montre le résultat de l'appel de la méthode *del_objet(O_1)* à partir de l'état initial décrit dans la partie a). La méthode *del_objet(O_1)* efface l'objet O_1 (et les associations a_1 et a_2).

Après la validation de la transaction incluant *del_objet(O_1)*, deux règles R_1 sont activées, l'une à partir de l'objet O_3 et l'autre à partir de l'objet O_4. Dans le cas de l'objet O_4, la règle R_2 sera activée du fait de la présence de l'association a_3. En effet, l'événement *Deletecomposant* surviendra et l'objet O_4 ne sera pas effacé.

1.3 Conclusion

Cette partie a décrit brièvement la notion de règles actives proposée dans le modèle de référence. Ces règles prennent place dans la définition des objets et décrivent les réactions que peuvent avoir ces objets face aux événements qui les concernent. Ces événements, dits *primitifs* ou *basiques* (cf. section §3.1), correspondent à des instants particuliers de l'exécution des méthodes d'objets.

Si ce modèle de règles actives est classique dans son principe, la notion d'association y introduit une particularité importante. En effet, les règles définies dans les associations peuvent concerner aussi bien l'association que les objets qui en sont origine et destination. De ce fait, lorsqu'un événement a lieu sur un objet, l'ensemble des règles à considérer est composé des règles à la fois propres à la définition de l'objet et à celles des associations liant cet objet à d'autres. Dès lors, le comportement d'un objet peut être étendu et/ou modifié de façon dynamique en fonction des associations auxquels ils participent. Ce principe d'extension de la recherche des règles à activer permet de modéliser le *comportement contextuel* des objets [Cas94].

1. Le mot clef Deffered est utilisé pour définir que la règle sera activée en mode différé, sinon, la règle est activée en mode immédiat.
2. *!destination* dénote la valeur de l'objet destination de l'association courante lors de la génération de l'événement *Deletecomposite*. Cette valeur fait partie du contexte de l'événement.

Par la suite nous supposons que le modèle de règles actives fait partie intégrante du modèle de référence à partir duquel nous avons bâti **MOHA** (cf. chapitre 4 §1). La partie §2, qui introduit la notion d'annotation dans **MOHA**, trouve dans le mécanisme des règles actives un moyen adéquat d'automatiser le recueillement d'annotations. Enfin, à travers une gestion de l'historique des événements, la partie §3 fournit un moyen d'étendre les types d'événements et donc, les possibilités d'activation des règles du modèle de référence.

2 Les annotations

Dans la partie §2.1, nous présentons la notion d'annotation telle qu'elle est utilisée dans d'autres domaines. La partie §2.2 présente les objectifs de l'intégration des annotations dans **MOHA**. La section §2.3 présente la façon dont les annotations sont représentées dans notre modèle. La section §2.4 présente la relation entre les annotations et les objets. La section §2.5 présente l'exploitation des annotations à travers le langage d'interrogation.

2.1 Introduction

La notion d'annotation a été utilisée dans plusieurs domaines tels que les éditeurs structurés (par exemple, le système *Mentor* [DKLM84], Field [Rei90]), les éditeurs syntaxiques (par exemple le système *MicroScope* [AO88]), les gestionnaires de configurations de logiciels [Leb94] [CW93], etc.

Une caractéristique commune à ces différents domaines réside dans le fait que les annotations sont considérées comme des informations complémentaires aux données principales du système. Cette complémentarité peut correspondre à l'enregistrement d'une note, une mesure, une observation, une remarque ou un commentaire que l'on souhaite associer à une entité.

Par exemple, dans le cas des éditeurs structurés comme *Mentor*, les composants d'un document sont décorés avec une variété d'annotations (notes de pied de pages, référence, exemples, commentaires, assertions de programmes, etc). Les annotations donnent des informations additionnelles sur les composants des documents mais elles n'interfèrent pas avec l'organisation du document lui-même.

MicroScope est un système d'analyse de programmes écrits en *Common lisp* et *Common Object.* Ce système aide les programmeurs à comprendre et à modifier les programmes. L'information associée aux programmes est stockée dans une base de connaissances. Les annotations de *Microscope* sont associées à des *items*, qui sont les éléments de base du modèle de représentation, tels que les noeuds dans un graphe, les procédures, les symboles, etc. Les annotations peuvent contenir du code source, des

déclarations de variables et constantes, des commentaires de conception, des rapports d'erreurs connues, etc.

L'originalité de cette notion d'annotation est donc de permettre d'introduire différentes informations dans le système, accessibles à l'utilisateur, tout en préservant l'organisation et l'unité des données "principales" sur lesquelles repose le fonctionnement du système.

La notion d'*annotation* a été introduite dans **MOHA** pour représenter des *faits* ponctuels liés à l'exécution des activités. Les annotations sont toujours associées à un objet de la base. Elles *annotent* l'objet avec des informations externes provenant des situations particulières dans lesquelles l'objet est impliqué.

2.2 Objectifs de l'intégration des annotations dans MOHA

Nous avons vu qu'un environnement guidé par les procédés doit permettre une gestion de faits liés à l'exécution des activités (cf. chapitre 3 §xx). Pour ce faire, nous proposons l'intégration de la notion d'annotation dans le modèle. Cette intégration a pour objectifs principaux :

- *Permettre la représentation de différents types d'informations* : Une annotation peut contenir des mesures sur un programme (le nombre d'erreurs trouvées lors d'une validation), sur la réalisation d'une activité (la durée d'une activité, le nombre de modules modifiés par l'activité), la trace d'une erreur, etc.

  Le contenu de l'annotation est donc arbitraire et doit pouvoir être défini par l'utilisateur selon ses besoins.

- *Permettre d'annoter tout objet* : De même que le contenu de l'annotation est arbitraire, l'objet qu'elle annote peut correspondre à n'importe quel objet de l'environnement représenté dans le modèle (une activité, un utilisateur, un programme, un espace de travail, etc).

  Une annotation ne doit pas être considérée comme une caractéristique de l'objet mais comme une caractéristique de son contexte d'utilisation. L'objet doit être indépendant des annotations qui lui seront associées.

- *Rendre accessibles les annotations aux utilisateurs sans les lui imposer* : Lors de la manipulation d'objets (par exemple, création, mise à jour, destruction, accès, etc), l'utilisateur ne doit pas être perturbé par le fait que les objets sont ou ne sont pas annotés. Les objets et les annotations ne doivent pas être mélangés. L'accès aux annotations doit être le résultat d'une demande explicite de l'utilisateur.

  Le système doit fournir des moyens d'exploiter l'information contenue dans les annotations.

- *Permettre le recueillement automatique des annotations* : Les annotations doivent pouvoir être automatiquement recueillies lors de situations précises

survenant dans le système. Ces situations peuvent être considérées comme des événements significatifs, par exemple, à chaque fois qu'une activité est initialisée ou terminée, qu'une opération est effectuée (le remplacement d'un fichier dans la base), que le temps estimé pour une activité a été dépassé, que le nombre de modules modifiés est supérieur à la moyenne, etc.

L'utilisateur doit pouvoir définir dans le modèle quand, comment et sur quels objets les annotations vont être créées.

Dans ce qui suit, nous décrivons les choix effectués pour satisfaire ces objectifs.

2.3 Représentation d'une annotation

Les annotations sont représentées dans notre modèle par des objets. C'est-à-dire qu'une annotation est une instance d'un type d'objet défini par l'utilisateur. Les types d'annotations sont organisés dans une hiérarchie de types dont la racine est un type prédéfini appelé *Annotation*.

Les types d'annotations se définissent de la même façon que les types d'objets, c'est-à-dire, avec des attributs, méthodes, règles actives, etc.

Les instances des types d'annotations se différencient des autres objets (instances de types de la hiérarchie *Objet*) par la façon dont elles peuvent être créées et référencées, ainsi que par leur comportement *historique*. Nous verrons dans les sections §2.4 et §2.5 ces différences.

La définition d'un type d'annotations est indépendante du type d'objets qu'elles vont annoter. D'ailleurs, les annotations d'un même type peuvent annoter des objets de types différents. Dans ce qui suit, nous présentons un exemple d'une hiérarchie de types d'annotations.

Pour garder une trace des opérations de remplacement d'un fichier dans la base, l'utilisateur a défini un type d'annotation *Check-in* contenant les attributs suivants :

- responsable : la valeur de cet attribut indique quel est l'utilisateur qui a réalisé l'opération
- type de modification : indique le type de mise à jour, par exemple, correction d'une erreur, ajout d'une fonction, etc.

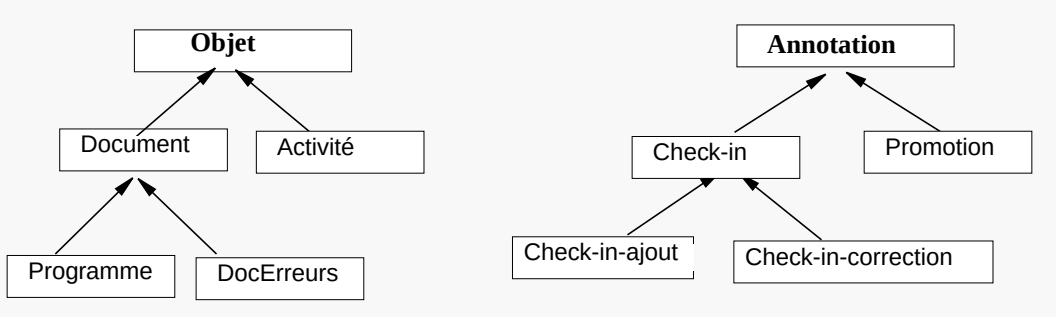


Figure 5 Exemple d'une hiérarchie de types d'objets et d'une hiérarchie de types d'annotations.

Des sous-types de l'annotation *Check-in* sont aussi définis pour garder des informations relatives au type d'opération à l'origine de la modification (figure 5). Par exemple, l'annotation *Check-in-ajout*, sous-type de *Check-in*, possède un attribut supplémentaire appelé *modification* qui est une référence au document expliquant la modification.

2.4 Les annotations et les objets

Une des différences entre les objets et les annotations est qu'une annotation n'a pas d'existence par elle-même. Une annotation existe si elle annote un objet. L'objet annoté peut aussi bien être un objet historique qu'un état spécifique.

Création d'une annotation

Les paramètres du constructeur de création d'une annotation doivent inclure l'objet à annoter et le type de l'annotation à créer. L'opération peut être réalisée soit explicitement par l'utilisateur, soit automatiquement par le système à l'aide des règles actives (cf. §1).

Supposons que nous voulons associer à l'opération de remplacement d'un programme (opération check-in) une annotation pour y stocker des informations expliquant les raisons de la modification. L'utilisateur peut alors faire la création explicite de l'annotation en appelant :

```
create-annotation (programme, check-in)
```

programme correspond à l'objet à annoter et check-in au type de l'annotation à créer.

Pour remédier à d'éventuels problèmes d'incomplétude (par exemple, on n'a pas toujours créé une annotation au moment du remplacement d'un programme), l'utilisateur peut se décharger de cette opération en l'associant à une action d'une règle active. Cette dernière sera déclenchée, de manière systématique, dès que l'événement "*l'opération check-in a été terminée*" est signalé (cf. §1.2). Pour ce faire, le type d'événement *end-check-in* est défini dans le type d'objet *Programme*,

```
end-check-in : FINISH check-in
```

ainsi que la règle active suivante :

```
ON end-check-in DO create-annotation (!objet, check-in)
```

la création de l'annotation du type *check-in* sera ainsi réalisée dès que le remplacement du fichier sera effectué.

Initialisation des attributs

Le constructeur de l'annotation (la fonction create-annotation) doit aussi initialiser tous les attributs de l'annotation. Pour définir les valeurs des attributs l'utilisateur peut utiliser deux mécanismes complémentaires.

Le premier mécanisme consiste à passer explicitement comme paramètre du constructeur une liste d'initialisations (attribut := expression), où expression est une expression du langage **LOHA** qui rend la valeur avec laquelle l'attribut attribut doit être initialisé. Cette liste constitue le troisième paramètre de la fonction create-annotation comme l'illustre l'exemple suivant :

```
create-annotation(!objet, check-in,
                  { responsable:= _usercurrent; comments:=remplir_formulaire () })
```

Dans cet exemple, la variable _usercurrent est une variable de l'environnement qui possède pour valeur la référence à l'utilisateur courant. La méthode remplir_formulaire va permettre à l'utilisateur d'initialiser les commentaires de l'annotation de façon interactive.

Le second mécanisme consiste à définir des fonctions d'initialisation appelées par défaut. Une *fonction d'initialisation* intervient uniquement lors de la création de chaque instance d'annotation : si aucune valeur n'est spécifiée explicitement dans la liste de paramètres du constructeur, le champ correspondant à l'attribut sera mis à jour avec la valeur retournée par l'évaluation de cette fonction. Par abus de langage, on parlera de *valeur d'initialisation*.

Par exemple, dans le type d'annotation check-in, les initialisations des attributs responsable et comments peuvent être définies de la façon suivante :

```
// Déclaration et initialisation des attributs du type check-in
    responsable : User := _usercurrent ;
    comments : Text := remplir_formulaire ();
```

Lors de la création d'un objet de type donné, les différentes initialisations se font dans l'ordre de déclaration des attributs. Les expressions peuvent faire référence à d'autre objets. Il est par exemple possible lors de la création d'une nouvelle annotation d'accéder aux attributs de l'annotation antérieur.

2.5 Annotations Historiques

Les instances d'annotation du même type, associées à un objet donné, sont enregistrées dans une structure appelée *annotation historique*. Une *annotation historique* est une liste d'instances du même type d'annotation. L'ordre de cette liste est celui de la

création des instances. Par exemple, dans la figure 6, l'objet O_1 a été annoté par des annotations de type T_A.

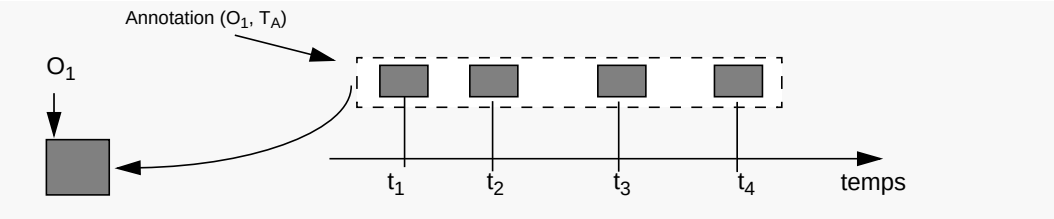


Figure 6 L'*annotation historique* de type T_A de l'objet O_1 référencée par *Annotation(O_1,T_A)*.

Une *annotation historique* est désignée par la référence résultant de l'application de la fonction *Annotation(objet, type d'annotation)*. Ainsi, *Annotation(O_1,T_A)* dénote la référence à l'*annotation historique* de type T_A de l'objet O_1.

Contrairement à un objet historique, une annotation historique ne représente pas *l'évolution d'un objet*. Chaque instance d'annotation de la liste représente un *fait* ponctuel. Il n'y a pas d'interprétation continue du temps dans une annotation historique. Dans l'exemple de la figure 6, cela implique que l'annotation historique est définie uniquement pour les instants t_1, t_2, t_3 et t_4. Par exemple, dans l'intervalle $]t_1, t_2[$, il n'y a pas d'annotation du type T_A pour l'objet O_1.

Un objet peut avoir une ou plusieurs *annotations historiques* différentes qui lui sont associées. Elles se distinguent alors par le type des annotations qu'elles contiennent.

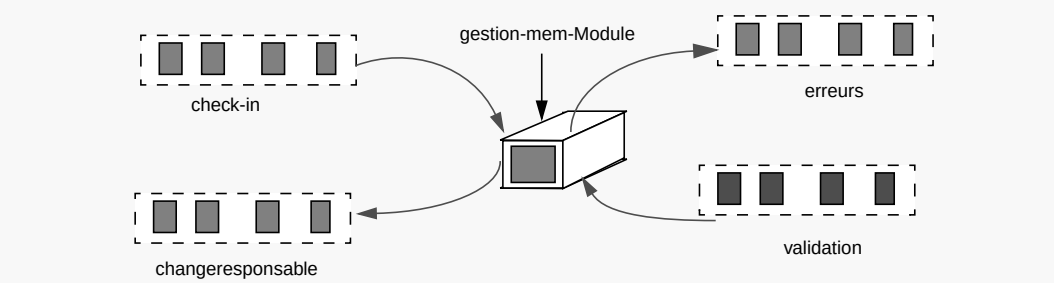


Figure 7 L'objet *gestion-mem-Module* et ses annotations.

Dans l'exemple de la figure 7 nous avons un objet historique (appelé *gestion-mem*) qui possède différentes annotations. Les annotations du type *check-in* contiennent les commentaires et justifications de la création des états de l'objet historique. Les annotations *changeresponsable* stockent des informations sur les utilisateurs qui ont été responsables du module. Les annotations *erreurs* gardent des informations qui permettront de tracer les erreurs rencontrées dans le module.

2.6 Manipulation des Annotations

Le langage d'interrogation **LOHA** a été étendu pour prendre en compte les annotations. Le principe de ces extensions consistent à offrir les moyens de construire des références sur les annotations.

Accès aux annotations historiques d'un objet

L'utilisateur a accès à une annotation historique associée à un objet en appliquant une fonction qui prend comme paramètres une référence à l'objet et le type d'annotation. Dans **LOHA**, nous la notons $O_1\#T_A$, c'est-à-dire :

$O_1\#T_A$ = Annotation(O_1,T_A)

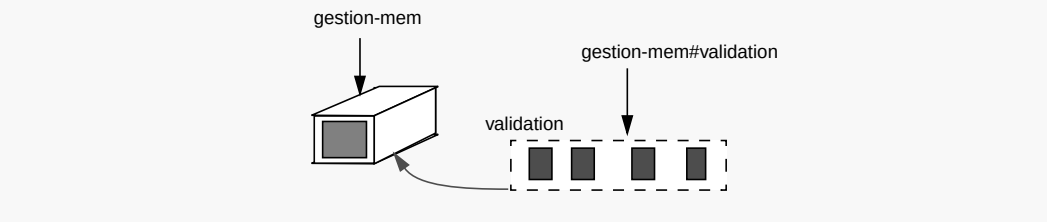


Figure 8 Accès à l'annotation hitorique validation de l'objet gestion-mem.

Ainsi, par exemple (figure 8), gestion-mem#validation correspond à une référence sur l'association historique de type validation qui est associée à l'objet référencé par gestion-mem.

Accès aux informations portées par les annotations historiques d'un objet

Une fois que l'on sait référencer une annotation historique associée à un objet donné, cette référence peut être exploitée dans les expression du langage **LOHA**. Notamment, on peut accéder à chaque annotation qui composent l'annotation historique, ainsi qu'à leurs différents attributs.

Par exemple, si le type de l'annotation historique gestion-mem#validation introduit l'attribut origine, la requête **LOHA** suivante (cf. R1) permet l'accès à la valeur de cet attribut dans la dernière annotation de type validation qui ait été enregistrée sur l'objet gestion-mem :

R1 Quelle est l'origine de la dernière validation qui a eu lieu sur le programme de gestion de mémoire ?

gestion-mem#Validation@LAST.origine

A travers **LOHA**, l'exploitation des annotations historiques est ainsi similaire à celle des objets historiques. Toutefois, puisque les références historiques supposent que les objets référencés décrivent l'évolution temporelle et continue d'une entité, les références historiques ne s'appliquent pas dans le cas des annotations historiques. Autrement dit, les annotations ne peuvent pas être partie des associations historiques.

2.7 Exemple

Description du problème

L'exemple proposé concerne une activité de codage définie pour implanter un ensemble de fonctions (des extensions) sur un logiciel existant. Cette activité est une activité assez complexe et de longue durée, qui requiert le concours de plusieurs utilisateurs. Pour ce faire, l'activité doit se décomposer en plusieurs sous-activités. Chaque sous-activité a comme objectif d'implanter un sous-ensemble de fonctions, possède un temps d'exécution prévu et est donnée à un utilisateur responsable.

Nous voulons effectuer des mesures pour évaluer la performance des sous-activités. A la fin de chaque sous-activité, nous voulons recueillir les informations suivantes : le responsable, la durée, et le nombre de modules modifiés par la sous-activité.

Principe de résolution

La solution que nous proposons consiste à associer à l'activité principale une annotation historique contenant des instances du type *FinSousActivité* (figure 9). Lorsqu'une sous-activité termine, une nouvelle instance d'annotation est créée pour recueillir les informations requises.

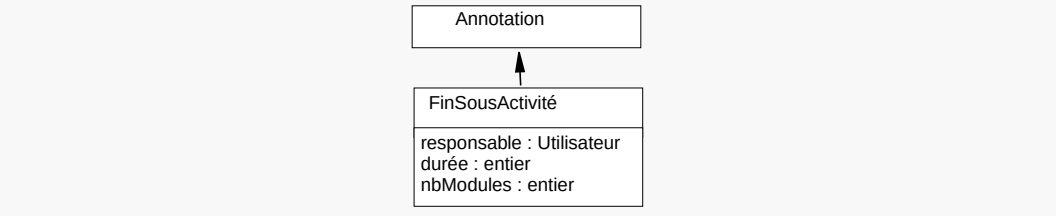


Figure 9 Le type d'annotation *FinSousActivité*.

On suppose que toutes les activités (l'activité principale et ses sous-activités) sont du type *Activité* ou de l'un de ses sous-types. Dans ce type, la relation entre une activité et ses sous-activités est exprimée à travers l'attribut composite *sous-activité* (c'est-à-dire l'association *activité&sous-activité*). De plus, pour qu'une activité puisse signaler sa terminaison, une méthode, appelée *SignalFinActivité()*, est définie dans le type *Activité*. Lorsqu'une instance d'activité fait appel à cette méthode, elle signale sa terminaison.

Pour automatiser le recueillement des annotations de type *FinSousActivité*, il faut réagir au signal de terminaison intervenant sur une sous-activité pour créer une annotation sur l'activité principale. Pour ce faire, il suffit d'introduire une règle dans la définition de l'association *sous-activité* liant une activité à une sous-activité :

```
FinSousActivité : FINISH SignalFinActivité() in DEST

ON FinSousActivité DOcreate-annotation (  !origine, FinSousActivité,
                                          {  (responsable := !destination.responsable);
                                             (durée := !destination.LifeSpan);
                                             (nbmodules := !destination.nbModules)
                                          } )
```

Les valeurs des attributs de chaque annotation sont initialisées par rapport aux valeurs de la sous-activité. Par exemple, la *durée* correspond à la période de vie de la sous-activité, le *responsable* à la valeur de l'attribut responsable de la sous-activité, etc.

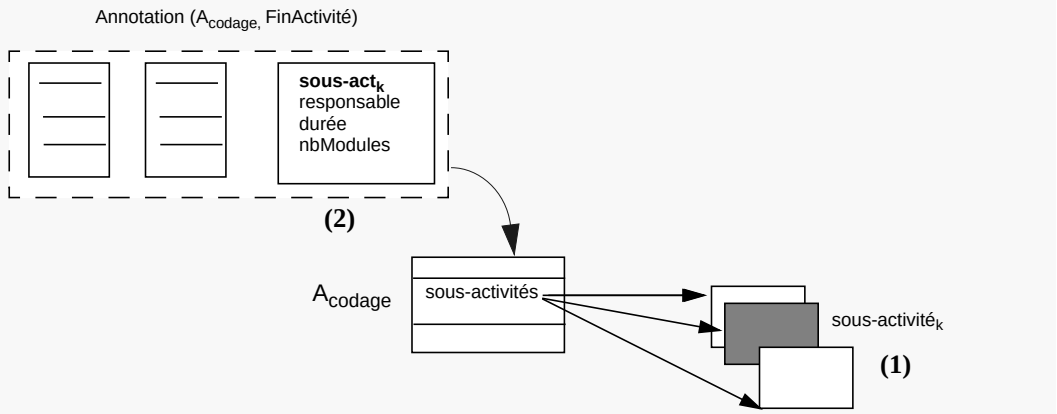


Figure 10 (1) l'activité appelée sous-activité$_k$ est terminée. (2) une annotation est créée dans l'annotation historique du type *FinActivité* associée à l'objet A_{codage}.

Exploitation de l'information de l'annotation historique

Dans ce qui suit, nous présentons quelques exemples de l'exploitation des annotations historiques en utilisant **LOHA**.

R2 **Quelle a été la sous-activité qui a modifié le plus de modules ?**

A_{codage}#FinSousActivité(max (nbModules))

R3 **Quelle a été la durée moyenne des sous-activités ?**

avg (A_{codage}#FinSousActivité.AllAnnotations.durée)

A_{codage}#FinSousActivité.AllAnnotations est l'ensemble d'annotations enregistrées dans A_{codage}, #FinSousActivité. avg est une fonction qui calcule la moyenne d'un ensemble de valeurs.

R4 **Qui était le responsable de la sous-activité qui a duré le plus ?**

A_{codage}#FinSousActivité(max(durée)).responsable

R5 **Quelle est la sous-activité qui a duré le plus et dans laquelle l'agent U_1 a été responsable ?**

A_{codage}#FinSousActivité(responsable = "U_1" and max(durée)).sous-activité

3 Les événements

D'une façon générale, toutes les applications qui s'appuient sur la relation cause-effet nécessitent la spécification et la détection d'événements complexes [CM91]. Dans

les environnements guidés par les procédés de fabrication, ce type de fonction est vital pour contrôler et superviser l'exécution des activités.

Le propos de cette partie est de montrer d'une part que l'aspect temporel inhérent à la notion d'événement en fait un concept qui s'intègre parfaitement dans notre modèle et, d'autre part, que l'on peut tirer parti des possibilités offertes par le langage **LOHA** et les annotations pour étendre les possibilités de spécifications d'événements.

L'originalité de notre approche est d'aborder la notion d'événement en dehors des mécanismes de déclenchement de règles qui vont l'exploiter. L'intérêt de cette démarche est d'étendre les possibilités d'exploitation des événements comme, notamment, la possibilité de conserver l'historique des événements pour mettre en place des mécanismes d'analyse, de statistique basé sur des métriques, etc. En effet, les événements étant porteurs d'informations relatives à des entités abstraites de nature instantanée et fugitive, comme les actions, les activités, etc., la gestion de leur historique permet de disposer d'une représentation effective et exploitable de l'ensemble du déroulement des procédés.

Dans la section §3.1, nous présentons brièvement la notion d'événement telle qu'elle est utilisée dans d'autres domaines et, plus particulièrement, nous introduisons la notion d'événement dit complexe ou encore composite. La section §3.2 présente comment le langage de requête **LOHA** et les annotations peuvent être utilisés pour étendre la notion d'événement dans **MOHA**.

3.1 Introduction

Dans les bases de données actives, les événements jouent un rôle essentiel. En effet, un gestionnaire d'objets dit *actif* se base sur la détection d'*événement* afin de *réagir* sans avoir recours à l'intervention d'un agent extérieur (par exemple, un utilisateur ou un outil). Dans ces systèmes, l'introduction de la notion d'événement résulte directement de celle de règle active qui décrit quelle action doit être entreprise lors de l'apparition d'un événement particulier. Les possibilités de réaction de ces systèmes sont donc complètement déterminées par les types d'*événements* qui leur sont possibles de détecter.

De façon générale, un événement dénote un instant du temps dans lequel il s'est passé quelque chose de remarquable et digne d'intérêt. Un type d'événements décrit un ensemble d'événements qui partagent des propriétés communes, tandis qu'une occurence d'événement décrit l'apparition à un instant donné d'un événement d'un type particulier.

Evénements complexes ou composites

La plupart des gestionnaires d'objets actifs, par exemple, Peplom^{A} [Ron94], Ariel [Han89], Postgres [SJGP90], Starbust [LLPS91] et Adèle (cf. chapitre 1 §2) (cf. §1) ne traitent que des événements dits basiques ou primitifs. Ces types d'événements recouvrent des événements dénotant des instants particuliers relatifs à l'exécution d'une

opération sur la base, au temps écoulé et mesuré par une horloge, ou à une intervention externe sur le système.

Certains systèmes permettent la définition et la détection d'événement complexes. C'est, par exemple, le cas dans les systèmes *Samos* [SG94], *Ode* [GJ91], *Hipac* [MD89], *Naos* [CCS94] et *Sentinel* [CM91]. Le principe de cette extension est de rendre possible la définition d'un type particulier d'événements en fonction d'autres et donc, d'être capable de caractériser un événement en fonction de ce qui s'est déjà passé dans le système.

Pour définir un type d'événement complexe, chaque système propose un langage basé sur un ensemble prédéfini d'événements primitifs et sur différents opérateurs de composition. Typiquement, les opérateurs correspondent à la conjonction, la disjonction, la séquence, la négation, etc. Ils permettant de définir de nouveaux types d'événements en fonction de ceux qui existent déjà.

L'exploitation des événements complexes dans un gestionnaires actif impose d'étendre le mécanisme de détection d'événements afin de prendre en compte l'aspect historique qu'ils introduisent. En effet, un événement complexe consiste en général en une composition d'événements qui ont eu lieu à différents moments du temps. Cela signifie que pour pouvoir détecter un événement composite, il faut que le système soit capable d'enregistrer, d'une façon ou d'une autre, l'histoire des événements déjà apparus.

Par exemple, pour détecter les événements complexes, le système *Samos* utilise des réseaux de petri, *Ode* des automates et *Sentinel* des arbres. Quelque soit le formalisme utilisé par le système de détection, l'objectif reste le même pour chacun de ces gestionnaires d'objets actifs : conserver une trace des événements qui sont susceptibles de participer à la détection d'un événement complexe.

Contexte d'un événement

Un événement est porteur d'information. Cette information, appelée le contexte de l'événement, est définie par le type de l'évenement. Elle constitue une sélection de données de l'état du système lors de l'instant d'apparition de l'événement permettant d'identifier l'événement en précisant les conditions de son occurrence, des informations de la base qu'il concerne, etc. Certains systèmes, comme *Ode* [GJS93], offrent la possibilité d'associer à un événement des attributs pour définir son contexte.

Par la suite, nous proposons d'offrir un mécanisme de gestion de l'historique des événements et de leur contexte. Cette proposition repose sur une une définition de la notion d'événement qui exploite la notion d'annotation comme support de représentation de l'historique des événements.

Puisqu'il est ainsi possible d'enregistrer la trace des événements dans la base historique, il est aussi possible d'exploiter le langage **LHOA** pour exprimer des requêtes complexes impliquant les événements enregistrés. Suivant ce principe, nous fournissons les moyens de définir des événements complexes dans le modèle.

3.2 Définition d'événement

Un événement est une situation d'une importance particulière. L'occurrence d'un événement détermine un point précis dans le temps, c'est-à-dire un instant. Quand un événement est signalé, l'état courant du système est appelé le *contexte général* de l'événement. Pour des raisons évidentes, l'état global du système ne peut pas être conservé pour chaque occurrence d'événement apparaissant dans le système.

Les types d'événements sont définis dans les types d'objets et les types d'associations. Un type d'événement est constitué de trois composants :

$$<E_{primitive}, C, OA>$$

$E_{primitive}$ correspond à l'événement primitif qui détermine le déclenchement de l'événement, *C* est la condition de l'événement et *OA* est l'objet à annoter.

Par la suite nous présentons plus en détail chacun de composants de la définition d'un type d'événement.

3.2.1 Les événements primitifs

Les événements primitifs dans notre modèle sont les événements liés à l'exécution de méthodes. Ces événements correspondent aux événements que nous avons défini dans la partie §1 de ce chapitre. L'exécution d'une méthode donne lieu à deux événements identifiés par des instants précis relatifs à l'exécution de la méthode : *START* et *FINISH*.

3.2.2 La condition

La *condition* est une expression booléenne décrivant les propriétés que doit vérifier l'état du système à l'instant de l'apparition de l'événement mais aussi les états du système constituant l'historique de la base.

L'expression booléenne définissant la condition permet d'utiliser des requêtes exprimées dans le langage d'interrogation **LOHA**. Ces requêtes permettent d'impliquer n'importe quelle donnée historique relative aux objets mais aussi aux occurrences d'événements qui ont déjà été stockées dans la base.

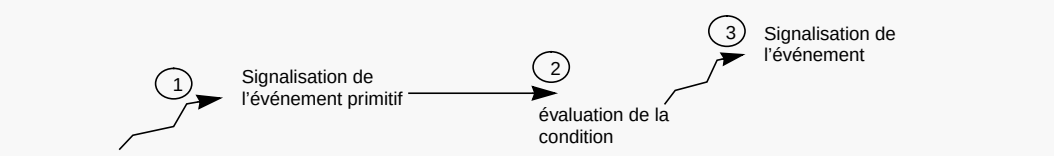


Figure 11 Détection des événements.

La figure 11 montre comment la détection d'un événement s'articule autour de la détection de son événement primitif et la vérification de sa condition. Il convient de noter qu'on ne fournit pas ici un mécanisme capable de détecter tout type d'événements

complexes comme le proposent certains systèmes (cf. §3.1). En effet, il n'est pas possible de détecter les événements complexes dont on ne connaît pas avec certitude le type de l'événement primitif final.

3.2.3 L'objet à annoter et la persistence des événements

Si la définition de l'événement contient un objet à annoter, alors, chaque fois qu'une occurrence est signalée au système, la création de l'annotation contenant son contexte est déclenchée. Le type de cette annotation est identifié par le même nom que l'événement. La définition de ce type décrit le contexte de l'événement que l'on souhaite enregistrer.

L'annotation historique correspondant à un événement peut être interprétée comme sa *trace* ou son historique.

Dans l'exemple précedent, (cf. §2.7) l'annotation FinSousActivité constitue le contexte de l'événement FinSousActivité. Ainsi, nous pouvons réecrire la définition d'événement de la façon suivante :

```
FinSousActivité : FINISH SignalFinActivité() in DEST
                  AnnotatedIn !origine WITH {
                                    responsable := !destination.responsable,
                                    durée := !destination.LifeSpan,
                                    nbmodules := !destination.nbModules
                                    }
```

4 Conclusion

Comme nous l'avons souligné dans l'introduction du chapitre, la notion d'annotation a été déjà utilisée dans d'autres domaines. La caractéristique commune est de considérer les annotations comme des information complémentaires aux informations "principales" du système. Notre but a été d'utiliser cette notion d'annotation et de proposer une représentation dans **MOHA** pour faciliter la modélisation et la gestion des informations historiques liées à l'exécution des activités de fabrication de logiciel (cf. chapitre 3).

Les historiques dans les gestionnaires de configurations de logiciels

Notre idée de départ a été d'étendre la gestion d'historiques des gestionnaires de configurations de logiciel. Cette gestion consistée à saisir des informations (prédéfinies) liées exclusivement à l'opération de remplacement d'un fichier dans l'espace de stockage. Il y a donc, dans cette approche, une limitation par rapport au type d'information à récupérer et par rapport au type d'opération à annoter. Cette approche a, de plus, l'inconvénient que ces informations ne peuvent pas être exploitées de la même façon que les autres données du système (cf. chapitre 2 §2.1). Elles sont stockées "ailleurs" puis, pour les consulter il faut utiliser des commandes spécialement conçues à ce propos.

Par rapport à ce type de gestion de l'information historique, notre proposition est beaucoup plus ambitieuse car elle permet de définir la situation (ou l'opération à annoter) et le type d'information à garder. De plus, les annotations étant modélisées de la même façon que les autres données du système, nous n'avons pas besoins des commandes spécifiques pour accéder aux annotations. Ceci peut être réalisé en utilisant **LOHA**.

Les annotations et les mesures

La notion d'annotation a été reprise dans la spécification du langage Apel[3] [DE95] du projet Perfect pour représente et gérer les *mesures* (cf. chapitre 1 §3). Rappelons que l'objectif majeur de ce projet est la définition d'un langage commun pour représenter le produit logiciel, les procédés, et les mesures dérivées des objectifs de qualité.

APEL doit permettre aux utilisateurs de :

- définir diverses types de *mesures*.
- définir une façon de regrouper les *mesures* sur lesquels peuvent être établies ou des mesures plus complexes (métriques).
- associer des *mesures* aux activités et aux produits.

Les événements complexes

Concilier les notions d'annotation et d'événement répond à deux objectifs complémentaires. D'une part, à travers la notion des règles actives, les événements s'avèrent un moyen expressif et puissant d'indiquer quand des annotations doivent être automatiquement recueillies. D'autre part, les annotations fournissent un support de représentation adéquat pour matérialiser les occurrences d'événements et aussi pour leur associer diverses informations. Un véritable historique des événements peut ainsi être constitué, puis être exploité à travers **LOHA** pour définir d'autres types d'événements.

De façon générale, une *situation* spécifique de l'environnement est un ensemble de circonstances définies en termes d'événements et de conditions sur l'état courant et, éventuellement, passé du système.

La reconnaissance des *situations* est très complexe. En effet, elle implique, d'une part, de stocker les historiques des données et des événements survenus dans le système et, d'autre part, de disposer d'un mécanisme capable d'y accéder afin d'identifier les situations.

Pour détecter les événements composites, chaque système offre un mécanisme de détection qui permet aussi de garder de façon plus ou moins explicite l'histoire des événements. Nous proposons de profiter de la base historique, spécifiquement de la notion d'annotation historique, pour conserve l'historique des événements et de leur contexte d'apparition. Nous croyons que un mécanisme général de détection d'événements complexe peut être mis en place en profitant de la base historique.

3. Abstract Process Engine Language.

PARTIE III

La mise en œuvre

Chapitre 7

La mise en œuvre

Nous présentons dans ce chapitre la mise en œuvre de l'ensemble des propositions présentées dans la partie II de ce document. La première section donne une vue globale de cette mise en œuvre en précisant les objectifs et les contraintes imposées par le cadre de notre travail. La section deux présente le système initial à partir duquel nous sommes partis pour effectuer l'implantation et la section trois décrit l'ensemble des modifications que nous y avons apporté.

1 Présentation générale

1.1 Objectifs et contraintes

Le cadre de notre travail est le système Adèle (cf. chapitre 1 §2). L'objectif initial du projet Adèle était la conception d'un gestionnaire d'objets et de configurations pour les environnements de génie logiciel. A présent, l'objectif de ce projet Adèle est plus ambitieux. Il s'agit de la définition et de la mise en place d'une plate-forme pour la construction des environnements logiciel guidés par les procédés (cf. chapitre 3 §2.2). Notre travail est inscrit dans cette démarche.

Le système Adèle est un système d'utilisation industrielle. Jusqu'à présent, il y a eu deux versions commerciales. Nous avons participé directement au développement de ces deux versions.

La prochaine version d'Adèle inclura la plupart des propositions présentées dans cette thèse. Dans cette perspective, notre mise en œuvre avait deux objectif primordiaux :

- *Réutilisation* : la réutilisation des composants du noyau Adèle a constitué un des objectifs majeurs de notre mise en œuvre. C'est pourquoi la plupart des décisions de conception d'Adèle sont maintenues et que la plupart des modules conservent leur sémantique.
- *Performance* : un de nos objectifs était d'introduire les nouvelles fonctions sans pour autant nuire aux performances des services déjà offerts par le système. Par exemple, les annotations ont été intégrées sans surcharger la gestion des objets.

Ceci veut dire que lorsque on accède un objet, l'ensemble de ses annotations ne sont pas impliquées. Elles ne sont accédées que si on le demande explicitement.

1.2 Architecture globale

Le système Adèle est organisé en une architecture client/serveur (cf. figure 1). Le serveur est essentiellement chargé de la gestion de verrous et de la reprise après panne. La base de données peut être répartie sur différents sites (via NFS). Chaque client dispose de tous les services fonctionnelles du système.

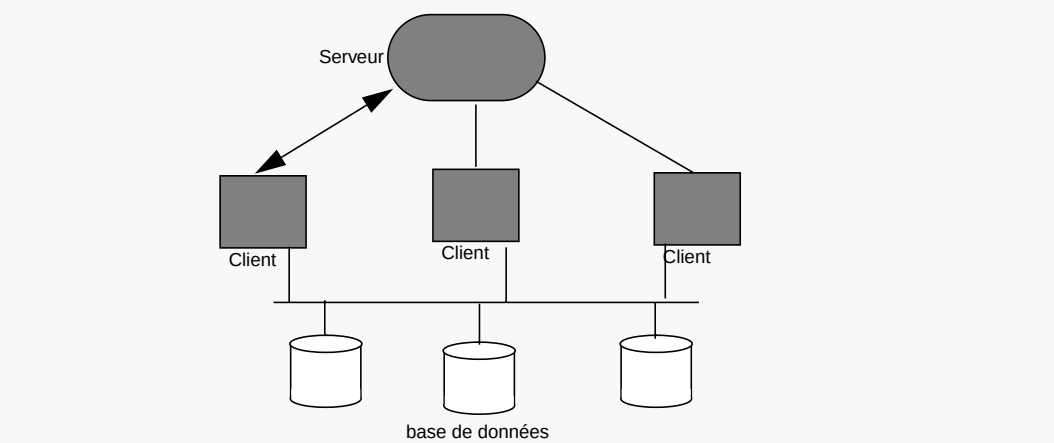


Figure 1 L'architecture client/serveur du système.

La base de données

L'information correspondant aux schémas de données et aux instances est stockée dans la base de données d'Adèle. Cette base est répartie sur différents sites et partagée par tous les clients.

La base de données repose sur le gestionnaire de fichiers du système d'exploitation. Chaque objet composite est stocké dans un fichier séparé.

Le serveur

Le serveur Adèle prend en charge deux fonctions du système : la gestion de verrous et la reprise après panne (cf. figure 2).

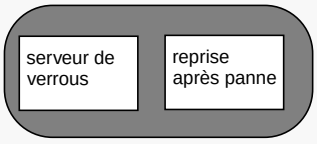


Figure 2 Le serveur du système Adele.

Le système de verrouillage d'Adèle permet de gérer les droits d'accès des client aux objets de la base. L'unité de verrouillage est établie en fonction du modèle interne de

représentation des données. Elle permet une gestion des droits d'accès à un niveau de granularité plus fin que l'unité de stockage (c'est-à-dire le fichier).

Le système de reprise, quant à lui, assure la restauration de la base en cas de panne du système. Il met en œuvre un mécanisme de journalisation des transactions qui permet de récupérer l'état de la base tel qu'il était avant le lancement des transactions interrompues par la panne.

Le client

Les services fonctionnels d'Adèle sont implantés dans le client. Les couches basses du client sont chargées du transfert de données entre l'espace de travail du client et la base de données partagée (cf. figure 3). Le chargement des objets du disque vers la mémoire n'est réalisé par le client qu'une fois que le serveur lui a délivré les verrous nécessaires.

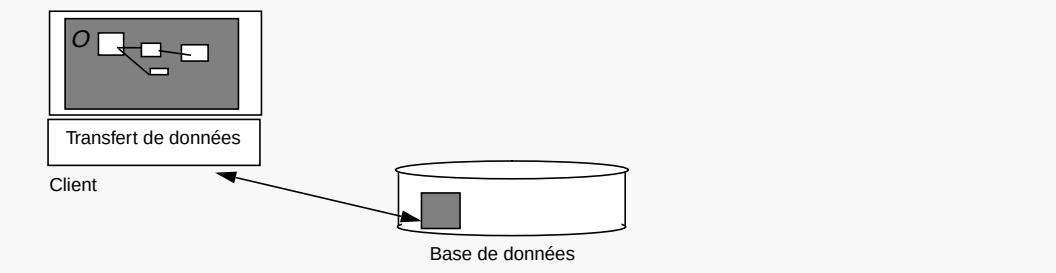


Figure 3 Transfert de données entre le client et la base de données.

Dans la figure 4, nous présentons les composants qui ont une relation directe avec notre mise en œuvre. La section suivante décrit brièvement le rôle et le fonctionnement de chacun d'eux.

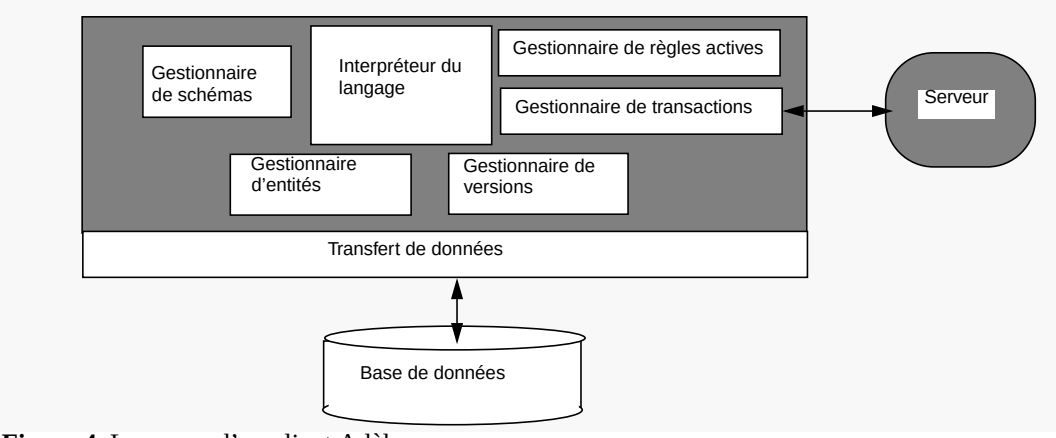


Figure 4 Le *noyau* d'un client Adèle.

2 Le système initial

Dans cette section, nous donnons un aperçu du fonctionnement du système initial à travers la description des composants du client (cf. figure 4). Les différentes modifications, que nous avons apporté à ce système, sont présentées dans la section suivante (cf. §3).

2.1 Le gestionnaire de schémas

Le gestionnaire de schémas de données permet de gérer les définitions de données d'une base Adèle. Ces définitions sont exprimées dans un modèle correspondant au modèle de référence présenté dans la section §1 du chapitre 4.

Le compilateur du langage Adèle traduit les schémas de données fournis par l'utilisateur en structures internes qui sont gérées par le gestionnaire de schémas. Une fois compilés, les schémas peuvent être manipulés et exploités à travers l'ensemble de fonctions du gestionnaire (son interface).

2.2 Le gestionnaire d'entités

Le gestionnaire d'entités inclut toutes les fonctions de manipulation d'instances aussi bien d'objets que d'associations. Ces fonctions comprennent la création et la destruction des objets et/ou des associations, la consultation et la modification d'attributs, etc.

2.3 Le gestionnaire de versions

Le gestionnaire de versions recouvre l'ensemble des mécanismes spécifiques à la gestion de graphes de dérivations.

Le mécanisme de gestion de versions du système Adèle repose sur un modèle interne basé sur les concepts de *branche* et *révision*. Une branche contient l'évolution d'un fichier. Une révision contient la valeur du fichier à un instant donné.

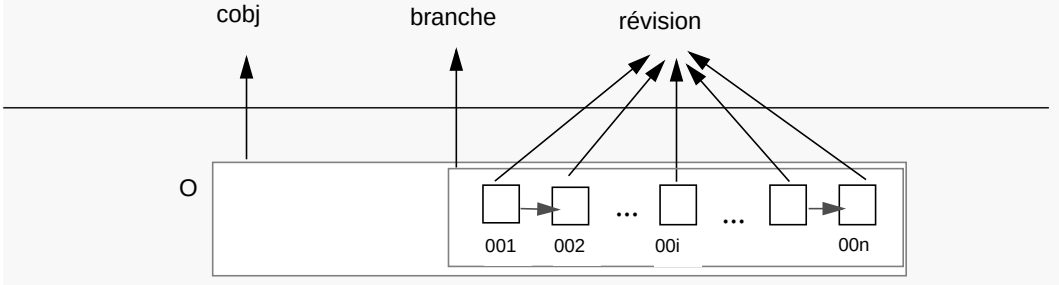


Figure 5 Objet, branche et révision dans Adèle

Dans ce modèle interne, quatre types d'objets de base sont fournis : les objets appelés *cobj* (composite objets), les objets *branche*, les objets *révision* et les objets *élément (cf.* figure 5).

Les objets *cobj* sont composés d'objets *branches*. Les objet *branches* contiennent des révisions, qui contiennent eux même des *éléments*. Un *élément* représente un fichier.

Le gestionnaire de branches

Les révisions d'une branche correspondent aux révisions du fichier utilisateur qu'il contient. Lorsqu'un utilisateur désire travailler avec un fichier stocké dans la base, il demande une copie de ce fichier.

Le support des révisions repose sur un mécanisme des delta, appliqué aux révisions consécutives, permet de ne pas encombrer inutilement le disque sur lequel se trouve la base. En effet, Adèle ne conserve pas tous les fichiers correspondants aux révisions d'un document. Seule la dernière révision correspond à un fichier qui est identique a celui présent dans un espace de travail. Les révisions antérieures sont, quant à elles, rassemblées dans un même fichier contenant la liste des différences existant entre deux révisions successives.

Lorsqu'un fichier est replacé dans la base, Adele appelle le mécanisme permettant d'obtenir les différences entre la dernière révision présente dans la base et la nouvelle qui va être créée. Le résultat est ajouté au fichier des deltas et le fichier correspondant à la dernière révision est remplacé par le nouveau venu dans la branche.

Le mécanisme de stockage par deltas permet un gain de place au niveau de la base, toutefois il ne s'applique que sur des fichiers texte (sur lesquels un *diff* est possible).

2.4 Le gestionnaire de règles actives

Le gestionnaire de règles actives constitue le noyau d'exécution des règles. La compilations de règles actives définies dans un schéma de données est assurée par le gestionnaire de schémas (cf. §2.1).

Nous avons choisi de traiter la détection des événements en incluant directement des instructions de notifications d'événements au début et à la fin des méthodes. Ainsi, les

fonctions du gestionnaire de règles actives sont uniquement la sélection de règles à déclencher, l'ordonnancement selon la priorité et la gestion des informations associées.

2.5 Le gestionnaire de transactions

Le gestionnaire de transactions fournit les fonctions de démarrage, validation et annulation d'une transaction. Ce composant gère la partie cliente de la communication entre le client et le serveur de verrous (cf. figure 4).

2.6 L'interpréteur du langage

Ce composant a en charge l'interprétation du langage de programmation d'Adèle. Ceci inclut l'exécution des méthodes de base et de celles définis par l'utilisateur. On peut accéder ce composant à travers une des interfaces du système (API, l'interface de commendes ou l'interface graphique).

3 L'implantation réalisée

Dans la figure 6 nous présentons, par rapport au système initial (cf. figure 4), les composants que nous avons modifié. Il y a eu deux modifications majeures. D'abord la redéfinition du gestionnaire d'entités et du gestionnaire de versions (cf. figure 7). Ensuite, l'ajout des nouveaux types de données de base qui vont aussi permettre d'implanter le langage d'interrogation **LOHA**.

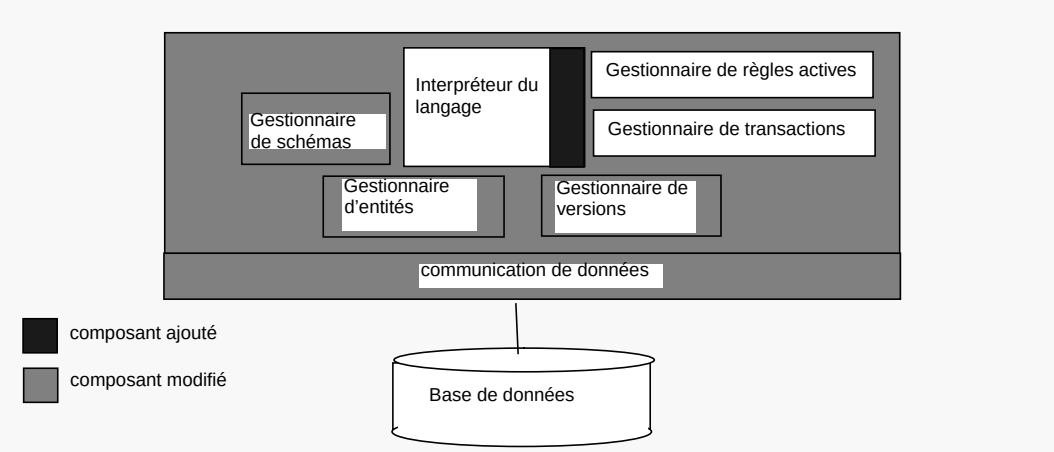


Figure 6 Le nouveau système client.

3.1 Gestionnaire d'entités et de versions

Le gestionnaire d'entités du système Adèle a été transformé afin d'inclure la gestion d'objets historiques et d'annotations (cf. figure 7). En généralisant la gestion de versions, nous avons pu uniformiser la gestion d'annotation avec celle d'objet historique.

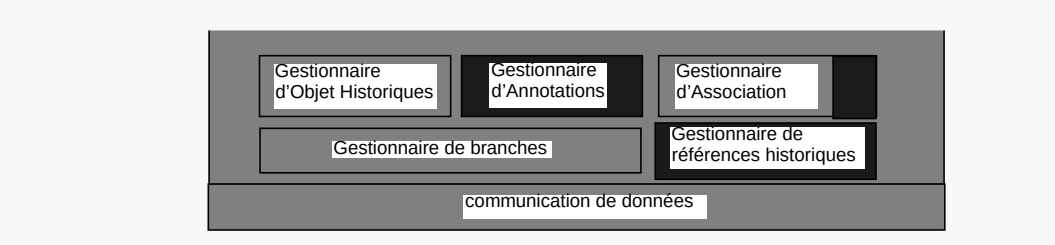


Figure 7 Redéfinition des composants : gestionnaire d'entités et gestionnaire de versions.

Comme nous avons vu dans le chapitre 6, plusieurs annotations historiques peuvent être associées à un objet. Dans notre implantation, nous considérons l'objet historique comme un cas particulier d'annotation, au moins en ce qui concerne le stockage et la gestion en mémoire.

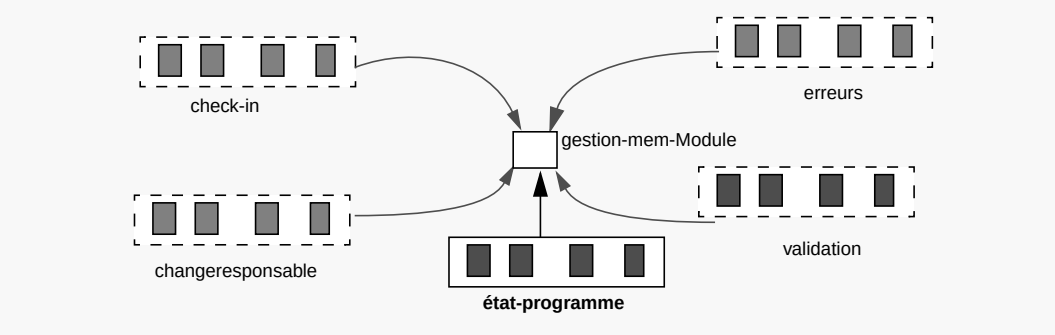


Figure 8 L'objet historique vu comme une annotation historique de plus.

La figure 8 donne la structure *interne* de l'exemple présenté dans la figure 6 du chapitre 6 (pag. 112). Dans ce cas, la liste d'états de l'objet historique est représentée comme une annotation historique (que nous avons appelée *état-programme* ; *programme* étant le type des états de l'objet historique appelé *gestion-mem-Module*).

3.2 Les gestionnaire de schémas

Définition d'objets historique

Pour un type d'objet *historiqueof T* défini dans un schéma, le compilateur génère trois types d'objet internes. Ces types sont : *T_cobj*, *T_br et T_rev.* Le type *T_cobj* sous-type de *cobj* dont le but est de regrouper toutes les informations concernant les versions coopératives de l'objet et toutes ses annotations historiques. Les attributs partagés sont définis dans ce type.

Type cobj :

components : setof branche

Type T_cobj IS cobj :

<attribut partagés définis en T>
traces : setof branche

Le type *branch* sert à gérer un objet historique particulier. Ce type est défini par une liste de révisions. Les attributs partagés sont aussi définis dans ce type

Type branche

rev : listof révision

Type T_br IS branche

<attribut partagés définis en T>

Le type *révision* sert à gérer les états de l'objet historique. Les attributs immuables et modifiables sont définis dans ce type.

Type révision ...

Type T_rev IS révision {
<attribut immuables définis en T>
<attribut modifiables définis en T>
}

La figure 9 montre de façon simplifiée la compilation d'un type **MOHA** ver le modèle interne d'Adèle.

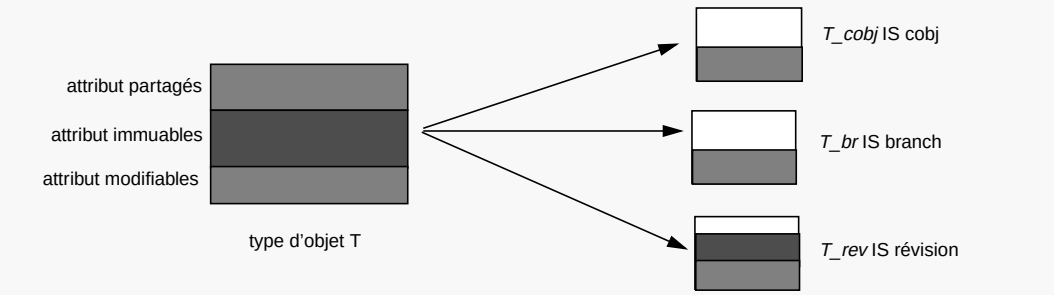


Figure 9 Transformation d'un type *T* défini dans **MOHA** vers le modèle interne d'Adèle.

Les événements

Pour chaque définition d'événements contenant un objet à annoter, nous générons deux choses. Dans le type de l'annotation correspondante, la fonction d'initialisation est définie en utilisant l'information donnée dans la définition de l'événement. Une règle active de priorité maximale qui indiquera la création de l'annotation. A cette fonction de création nous passons en paramètre l'objet spécifié dans l'événement.

3.3 Le gestionnaire de branches

Les mécanismes de gestion de versions déjà existant dans le système forment le niveau physique. Nous avons toutefois été amenés à étendre certains des composants de ce niveau physique car ils n'étaient pas suffisamment généraux pour être réutilisés tels quels.

Nous avons dû étendre les mécanismes de gestion de *branches* offerts par Adèle pour pouvoir, d'une part, avoir des *révisions* de n'importe quel type et, d'autre part, avoir des révisions sans fichier.

3.4 Transfert et stockage de données

Dans le cas des annotations, le client va charger une annotation historique associée à un objet uniquement sur demande explicite. Même si en mémoire, l'annotation est rattachée à l'objet qu'elle annote, les deux opérations sont indépendantes.

Dans le modèle interne, nous avons défini un type d'objet dont le contenu est stocké dans un fichier. Les annotations historiques sont représentées par des objets à la fois sous-type de branches et aussi de ce type d'objet dont le contenu est stocké dans un fichier et l'unité de verrou correspond aussi au même fichier.

Nous avons aussi modifié le mécanisme de gestion de verrous pour prendre en compte les annotations. Pour atteindre cet objectif, nous avons dû modifier la gestion de verrous et des fichiers. Dans ce qui suit, nous présentons les diverses modifications que nous avons réalisées.

Chaque annotation historique d'un objet donné est stockée dans un fichier indépendant. Dans l'exemple de la figure 10, on montre que l'objet *O* est stocké dans un fichier de la base Adèle (appelé contenu#). Chaque annotation historique de cet objet est stockée dans un fichier indépendant du stockage de *O*.

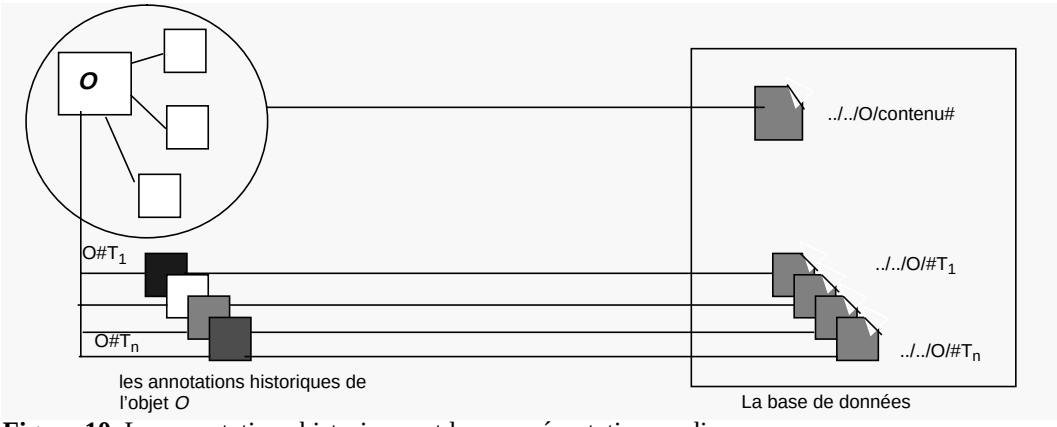


Figure 10 Les annotations historiques et leur représentation en disque

3.5 L'interpréteur du langage

La figure 6 montre le composant *interpréteur du langage* avec une extension. Cette extension correspond à l'ensemble de fonctions nécessaires pour évaluer les expression **LOHA**. La mise en place de **LHOA** repose sur la manipulation de nouveaux types de données : les intervalles, les éléments temporelles et les références historiques (cf. figure 11).

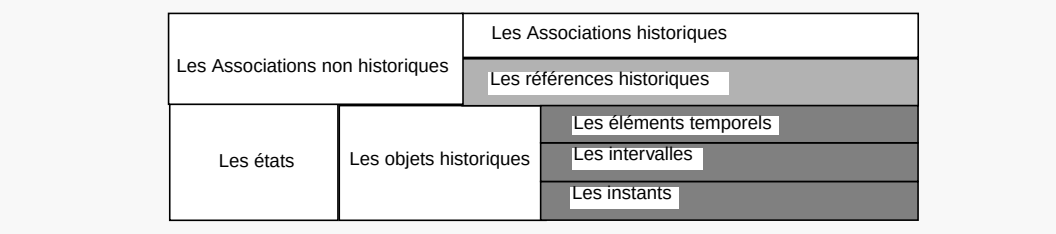


Figure 11 Les nouveaux types de données.

Dans ce qui suit, nous présentons brièvement les fonctions de base de ces nouveaux types de données.

Les éléments temporels

Un élément temporel est un ensemble fini d'intervalles, il sera représenté comme une liste de couples de valeurs, chacune représentant un intervalle. Les fonctions de base de la manipulation d'éléments temporelles sont l'intersection, union et différence. Nous avons aussi des fonctions permettant de retrouver un intervalle parmi cette liste.

Les références historiques

Une référence historique (cf. chapitre 4, chapitre 5) est représentée dans notre implantation par un type contenant l'identificateur de l'objet historique concerné et l'élément temporel déterminant la période de visibilité.

Pour ce type d'objet référence historique nous avons défini l'ensemble de fonctions permettant l'accès à la manipulation de la période de visibilité (l'élément temporel) et l'accès à l'objet historique. De même, les fonctions de base qui permettent de réaliser la navigation à travers les associations historiques sont implantées dans ce composant.

4 Conclusion

Dans ce chapitre nous avons présenté les grandes lignes de la mise en œuvre de nos propositions. Cette mise en œuvre a été réalisée en partant de l'implantation du système Adèle (Adèle V3). Cette mise en œuvre a été effectuée en langage c++. Le noyau du système est composé par 150000 lignes de code.

Notre travail intervient à plusieurs niveaux. Pour certains composants, l'extension s'est traduite par une généralisation qui favorise leur réutilisation. Pour d'autres, l'extension a nécessité un changement radical d'implantation (le cas des informations historiques). Et enfin, de nouveaux composants ont été introduits pour assurer le support du nouveau modèle au dessus du modèle existant.

Conclusion et perspectives

1 Rappel de la problématique

Les fabricants des logiciels doivent pouvoir estimer l'utilisation de leurs ressources, c'est-à-dire : comprendre pourquoi les projets logiciels ne satisfont pas les objectifs prévus, pouvoir estimer l'impact de changements sur un logiciel, etc. Pour arriver à ces fins, ils doivent pouvoir profiter au maximum de leurs expériences passées pour réutiliser leurs *procédés de fabrication de logiciels* et ainsi, enrichir leur *savoir-faire*.

Les procédés de fabrication de logiciel comprennent l'ensemble des activités, aussi bien techniques qu'administratives, qui sont nécessaires à la fabrication d'un logiciel. Ces activités vont de l'analyse des besoins jusqu'à la gestion de l'évolution ou la maintenance du logiciel, en passant par l'implémentation, la gestion de configurations, le contrôle de qualité, etc.

L'émergence d'environnements dits guidés par les procédés de fabrication de logiciels (EGPFL) est le résultat de recherches visant à concilier les aspects modélisation, exécution et évaluation des procédés. La problématique de la définition des plate-formes pour la construction des EGPFL est vaste et complexe. Dans le cadre de cette thèse, nous nous sommes intéressés uniquement à un sous-ensemble de cette problématique. En effet, nous avons cherché à identifier l'impact sur le gestionnaire d'objet que peut avoir la représentation explicite de procédés dans l'environnement logiciel.

Notre but était de définir un gestionnaire d'objets logiciels intégrant les concepts et les mécanismes adéquats pour :

- gérer l'information produite et consommée dans l'EGPFL et,
- fournir les *mécanismes* de base nécessaires à la mise en place des tâches de mesure, d'évaluation et contrôle des procédés.

2 Démarche suivie

Le dénominateur commun à l'exécution, au contrôle, à l'évaluation et à la supervision des procédés réside dans la nécessité de gérer l'aspect *historique* des informations. En effet, il s'agit selon le cas de gérer soit l'évolution du produit, soit une trace de l'exécution des activités. Etant donné notre objectif, notre démarche a donc consisté à centrer notre étude autour de la gestion de l'information historique d'un EGPFL.

En ce qui concerne l'évolution du produit logiciel, nous nous sommes intéressés à la gestion de *versions*. Notre expérience, en tant qu'à la fois utilisateur et développeur du gestionnaire de configurations Adèle, nous a permis de mettre en évidence la confusion régnant entre les concepts de haut niveau de la gestion de versions (par exemple, l'évolution d'un logiciel, le développement en parallèle, etc) et les mécanismes de bas niveau qui sont fournis (par exemple, graphe de dérivation, mécanisme de delta, etc).

Rejoignant les travaux réalisés par E. Sciore [Sci91] [Sci93], nous avons étendu nos recherches à d'autres domaines comme par exemple, la conception assistée par ordinateur, les éditeurs structurés, et les bases de données temporelles. Cette étude nous a permis de mettre en évidence l'existence de différentes façons d'aborder le problème de la gestion de versions.

Sur la base de cette étude et de l'expérience acquise avec le système Adèle, nous avons défini un modèle de versionnement à trois dimensions. Son intérêt provient du fait qu'il sépare clairement trois problématiques orthogonales et généralement confondues. Une version peut être *historique* (problématique de l'évolution d'objets), *logique* (problématique de variations du logiciels) ou *coopérative* (problématique du travail coopératif). Chaque dimension (historique, logique et coopérative) possède ses propres concepts et nécessite des mécanismes de gestion spécifiques et adaptés.

Ayant ainsi isolé la problématique de l'évolution des objets, la définition d'un modèle correspondant à la dimension historique a constitué la suite logique de notre travail. Pour ce faire, le domaine des bases de données temporelles s'est avéré notre source d'inspiration principale. Toutefois, s'il constitue un cadre conceptuel déjà bien défini, l'essentiel des résultats concerne le modèle de données relationnel. Aussi, nous a-t-il fallu adapter les solutions proposées à notre cadre particulier, c'est-à-dire le modèle à objets, et aussi prendre en compte de nouveaux problèmes liés au spécificités du génie logiciel.

Après l'évolution des objets, nous nous sommes intéressés au problème des informations historiques liées aux traces d'activités d'un procédé de fabrication logiciel. Contrairement aux informations constituant le modèle du produit et le modèle de procédés, les informations historiques liées aux traces peuvent être introduites pour des objectifs différents comme le contrôle de qualité, la supervision des procédés, etc. Pour offrir cette souplesse, nous avons repris la notion d'annotation utilisée dans d'autres domaines. En effet, dans notre cadre, cette notion permet de représenter des *faits* ponctuels liés à l'exécution des activités, tout en préservant l'organisation et l'unité des données "principales" sur lesquelles repose le fonctionnement du système.

3 Synthèse du travail réalisé

Concrètement nous avons proposé un modèle à objets historiques et associations (**MOHA**). Ce modèle permet de gérer l'évolution des objets dans le temps. Pour l'exploiter, nous proposons un langage navigationnel adapté aux besoins d'accès d'un EGPFL. Pour permettre l'introduction d'informations complémentaires, le modèle est étendu avec la notion d'*annotation*. La notion d'*événement* est incluse dans le modèle pour l'identification des situation complexes impliquant des informations passées et présentes de l'environnement.

Le modèle MOHA

MOHA est un modèle qui permet de prendre en compte de façon générale l'évolution des objets logiciels. Le concept de base de **MOHA** est l'*objet historique*. Un objet historique est une séquence d'états où chaque état représente la valeur de l'objet pendant un intervalle de temps donné. **MOHA** offre une sémantique de versionnement historique qui permet de modéliser quelles sont pour chaque objet les propriétés dont on veut conserver un historique.

MOHA permet de définir des associations historiques et des associations non historiques. Les associations historiques sont établies entre des objets historiques. Elles ont une *période de validité* qui correspond à l'intervalle de temps entre l'instant de création et l'instant de mort de l'association. L'association meurt soit lorsque l'utilisateur l'annonce explicitement, soit parce qu'un des objets impliqués meurt. Après sa mort, l'association n'est plus valide entre les deux objets, elle devient une information historique.

Les associations non historiques n'imposent pas de contraintes temporelles entre les objets qu'elles lient. C'est-à-dire qu'en utilisant les associations non-historiques, on peut établir des associations entre des objets qui appartiennent à des intervalles de temps différents.

LOHA

LOHA est un langage déclaratif dont le but est d'offrir un moyen simple et puissant d'accéder aux informations contenues dans une base **MOHA**. Ce langage d'interrogation est un complément du langage de programmation du gestionnaire d'objets. La puissance du langage **LOHA** provient des possibilités de navigation à travers les graphes d'une base **MOHA**. Ce langage repose sur deux notions principales : les *expressions de chemins* et les *filtres*.

Une expression de chemins décrit un type de parcours à travers un graphe et rend pour valeur les objets terminaux de ce parcours. Les graphes constitués par des objets et des associations **MOHA** sont hétérogènes et la sémantique du parcours dépend des catégories d'associations à traverser (historique ou non-historique).

Les annotations

La notion d'*annotation* a été introduite dans **MOHA** pour représenter des *faits* ponctuels liés à l'exécution des activités. Les annotations sont toujours associées à un objet de la base. Elles *annotent* l'objet avec des informations externes provenant des situations particulières dans lesquelles l'objet a été impliqué. Les annotations permettent de représenter et de gérer des traces de l'exécution des activités ainsi que des mesures.

Les événements

La notion d'événement est incluse dans le modèle pour l'identification des situations complexes, impliquant des information passées et présentes de l'environnement. Nous avons étendu les événements du système de base en utilisant le langage d'interrogation **LOHA**. A travers ce langage nous pouvons exprimer des conditions complexes impliquant d'autres événements. De plus, la notion d'annotation permet de matérialiser dans la base historique la trace des occurrences d'événements et d'une partie de l'état du système à l'instant ou l'événement est signalé.

La mise en œuvre

L'ensemble des propositions a été implanté et est en cours d'expérimentation. Notre mise en œuvre rentre dans le cadre du développement d'une nouvelle version du système Adèle. Notre travail intervient à plusieurs niveaux. Pour certains composants, l'extension s'est traduite par une généralisation qui favorise leur réutilisation. Pour d'autres, l'extension a nécessité un changement radical d'implantation. Et enfin, de nouveaux composants ont été introduits pour assurer la traduction entre le nouveau modèle et celui existant. Nous présentons une évaluation de notre mise en œuvre dans la section suivante.

4 Evaluation

Afin de permettre de mieux apprécier le travail réalisé, nous décrivons ci-après le points forts de notre travail, mettant ainsi en évidence ses principaux apports.

Les trois dimensions du versionnement

Nous considérons que le modèle de versionnement à trois dimensions que nous avons proposé est un apport important à l'étude de la problématique de la gestion de *versions*. La séparation que nous avons faite, permet d'isoler trois problèmes orthogonaux. Ce modèle a été non seulement utilisé par nous-même pour étudier la problématique de la dimension historique dans le cadre du système Adèle, mais il sert aussi de cadre de référence à d'autres travaux [GCCMS96].

Même si dans cette thèse nos travaux abordent essentiellement la dimension historique, nous avons aussi participé, dans le cadre de recherches mené par l'équipe

Adèle, à l'élaboration de propositions visant les deux autres dimensions [EC94] [EsCa95] [CEF96]. Dans les perspectives, nous présentons un brève aperçu de nos travaux en cours.

MHOA et LHOA

Le modèle **MOHA** dépasse largement le cadre de la gestion de l'évolution des fichiers qui est souvent le seul aspect pris en compte par les gestionnaires de configurations de logiciels.

La différence fondamentale entre notre modèle et les modèles temporels issus du domaine des base de données [Sno95] est la façon dont nous avons intégré les associations. **MOHA** permet de représenter et gérer des associations *historique* et *non historiques*. Contrairement aux associations historiques, les associations non historiques permettent d'établir des liens entre objets sans se préoccuper de satisfaire la contrainte d'existence temporelle. C'est-à-dire que nous pouvons établir des associations entre des objets appartenant à des époques différentes. Cette caractéristique étend la problématique généralement considérée par les bases de données temporelles.

Le langage **LOHA**, quant à lui, permet d'exploiter les différents concepts du modèle **MOHA** et notamment ses aspects historiques. En reposant seulement sur deux notions, les chemins et les filtres, **LOHA** est simple, mais il permet toutefois de formuler de manière concise des requêtes sophistiquées.

Annotation et événements

La notion d'*annotation* a été introduite dans **MOHA** pour représenter des *faits* ponctuels liés à l'exécution des activités. Les objectifs fixés (cf. chapitre 6 §2.2) pour l'intégration des annotations ont été atteints.

En ce qui concerne l'extension que nous proposons pour les événements, l'originalité de notre approche est d'aborder la notion d'événement en dehors des mécanismes de déclenchement de règles qui vont l'exploiter. L'intérêt de cette démarche est d'étendre les possibilités d'exploitation des événements comme, notamment, la possibilité de conserver l'historique des événements pour mettre en place des mécanismes d'analyse, de statistique basé sur des métriques, etc.

Par ailleurs, les notions d'annotations et d'événements ont été reprises dans la définition du langage APEL du projet PERFECT.

La mise en œuvre

Le prototype que nous avons réalisé est un produit pré-industriel. En tant que tel, il a été conçu pour respecter de fortes contraintes de performance.

Notre participation active au développement des précédentes versions commerciales du système Adèle, nous a permit de proposer une mise en œuvre complète et robuste.

Notre travail de réalisation contribue à améliorer le système Adèle sur deux niveaux. Au niveau conceptuel, nos travaux ont permis de mieux formaliser les fonctions du système. Au niveau de l'implantation, nos propositions permettent d'augmenter les possibilités de réutilisation des composants grâce à une réorganisation du système et à la généralisation de certains composants.

Nous sommes en train de spécifier la prochaine version commerciale d'Adèle. Cette spécification intègre les idées présentées dans cette thèse et définit les deux autres dimensions du modèle de versionnement [CEF96].

Les aspects liées à la manipulation des associations historiques ne serons pas intégrés immédiatement dans la version commerciale d'Adèle. Nous devons d'abord réaliser une phase d'expérimentation pour mieux évaluer les sur coûts des fonctionnalités de ce type dans le cadre des procédés logiciels réels. Cette phase doit être entreprise à court terme.

Par ailleurs, l'extension de la dimension historique passe par la mise en place d'un mécanisme de détection d'événements complexes offrant, au niveau conceptuel, les mêmes fonctions que les systèmes à événements existant et basé, au niveau physique, sur notre principe de gestion d'historiques des événements.

5 Perspectives

Mises à part les améliorations sur la mise en œuvre énoncées dans la section précédente, le travail que nous avons réalisé ouvre des perspectives sur plusieurs plans. Nous présentons d'abord les différentes perspectives liées à l'extension des événements et l'extension du modèle de versionnement puis celles liées à l'intégration de nos propositions avec les aspects méthodologiques du projet *PERFECT* (cf. chapitre 1 §3).

Les événements complexes

Nous proposons de profiter de la base historique, spécifiquement de la notion d'annotation historique, pour conserve l'historique des événements et de leur contexte d'apparition. Un mécanisme général de détection d'événements complexe doit être mis en place en profitant de la base historique.

La dimension coopérative

Les activités d'un procédé de logiciels peuvent s'exécuter en parallèle et partager des ressources afin d'aboutir à un objectif commun. Un *espace de travail* est l'endroit dans lequel vont s'exécuter ces activités. La dimension coopérative du versionnement a pour but de permettre la définition de politiques de collaboration entre les *espaces de travail* et d'assurer leur contrôle. Le travail envisagé pour la mise en œuvre du gestionnaire d'espaces de travail, consiste à garantir deux propriétés :

- *Isolation* : imposant que les modifications réalisées dans un espace de travail

soient uniquement visibles à partir de cet espace et, réciproquement, que les modifications réalisées en dehors de cet espace de travail ne soient pas visibles.

- *Transparence* : signifiant que chaque activité croit qu'elle travaille sur les objets réels de la base.

Ces propriétés peuvent être garanties grâce à la gestion de versions coopératives. Cette gestion est basée sur la notion d'objets historiques *partiels* (c'est-à-dire avec une période de visibilité restreint à un intervalle).

La dimension logique

La dimension logique du versionnement fait référence aux variantes ou versions compétitives. Ces versions existent simultanément et évoluent en parallèle. C'est-à-dire que chacune des versions logiques a sa propre évolution historique.

Le gestionnaire d'objets doit permettre de représenter *les facteurs de variations* des composants logiciels indépendamment de la représentation physique choisie pour gérer les variantes de logiciels. Par ailleurs, le gestionnaire d'objets doit offrir des fonctions telles que la sélection d'une variante particulière, la construction de configurations, l'élimination d'une variante, la propagation d'une modification à travers les variantes, etc.

Par rapport a cette dimension du versionnement, la perspective la plus directe est d'étendre le langage d'interrogation pour que l'on puisse exprimer des requêtes permettant d'inclure à la fois le temps et des facteurs de variation.

Les aspects méthodologiques

Bien que les annotations soient utilisées pour représenter et gérer les mesures d'un produit ou d'une activité, nous ne proposons pas directement une méthodologie d'utilisation du modèle proposé. A ce titre, le projet PERFECT permet de faire le lien entre notre travail et la mise en place d'un modèle de qualité. Parmi les objectifs du projet PERFECT il faut rappeler :

- la définition et la mise en place de méthodes d'évaluation de procédés permettant de construire de bases d'expériences de projets logiciels,
- la définition de mécanismes de mesure et d'évaluation pour contrôler et superviser le développement des projets logiciels.

Ce projet constitue le prolongement logique de notre travail. Deux aspects particuliers du projet PERFECT nous intéressent. D'une part l'aspect méthodologique, c'est-à-dire, comment savoir ce qu'il faut mesurer, et d'autre part, les aspect techniques qui permettent la représentation et gestion de ces mesures. A travers APEL, on permet de définir des métriques sur le produit et le procédé. Ces métriques ont été dérivées grâce à la méthode GQM (Goal, Questions and Metrics) [BR88]. Dans cette méthode, les métriques correspondent à la quantification des interrogations (*questions)* qui vont permettre de vérifier si l'*objectif* a été atteint. GQM inclut aussi une méthode qui permet de *dériver* les métriques en fonction des objectifs de qualité.

L’objectif à plus long terme consiste à s’appuyer sur notre travail et son prolongement dans le projet PERFECT pour concevoir un environnement permettant d’évaluer un procédé logiciel, de proposer des améliorations, et de mesurer les gains attendus et effectivement réalisés par une modification du procédé. Les gains en questions pouvant porter aussi bien sur les aspects coût, délai qualité des logiciels produits, que sur des aspects tels que la prédictibilité (des coûts, délai et qualité), la qualité du contrôle, la souplesse des réactions aux événements imprévus.

La réalisation d’un tableau de bord et de contrôle, permettant de consulter en temps réel l’état courant du projet, constitue un premier sous-objectif de ce vaste projet.

Références

[ABAK94] D. Agrawal, J.L. Bruno, El Abbadi et V. Krishnaswamy. Relative serializability: An approach for relaxing the atomicity of transaction. In *Proceedings of the ACM SIGACT/SIGMOD Symposium on Principles of DatabaseSystem*, pages 139–149, May 1994.

[ABD+89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier et S. Zdonik. The object-oriented database system Manifesto. In *Deductive and Object Oriented Databases*, Kyoto, Japan, 1989.

[AC93] M. Adiba et C. Collet. *Objets and bases de données. Le SGBD O2*. Editions Hermes, 1993.

[Ade93] Adèle. *Adèle. Configuration Management.* Verilog SA, 1993.

[ADS+94] R. Adomeit, W. Deiters, F. Shulke, H. Weber, B. Holtkamp et R. Rockwell. Software engineering environments. In J.J. Marchiniak, editor, *Encyclopedia of Software Engineering*. John Wiley and sons, 1994.

[AF91] R.S. Arnold et W. B. Frakes. Software reuse and reengineering. In *CASE trends*. february 1991.

[AN91] R. Ahmed et S.B. Navathe. Version management of composite objects in CAD databases. In Cliford and R. King, editors, *Proceeding of the ACM SIGMOD conference on managemnet of data*, pages 218–227, Colorado, USA, 1991.

[AO88] J. Ambras et V. ODay. MicroScope : A knowledge-based Programming Environment. *IEEE software*, 1988.

[Arb93] Selma Arbaoui. *PEACE : Un formalisme fondé sur la logique modale pour la modélisation et la mise en ouvre des processus logiciels évolutifs et non-monotones*. PhD thesis, Université Pierre Mendes France Grenoble II, 1993.

[ARCW95] D. Avrilionis, I. Robertson, PY. Cunin et B. Warboys. Meta Process. In Derniame et Fugetta DF96.

[Ban95] S. Bandinelli. Report on Workshop on software process architecture. Technical report, Milano, Italy, march 1995.

[Bar94] N. S. Barghouti. Separating process model enactment and execution in provence. In *9th International Software Process workshop*, Airlie, Virginia, October 1994. IEEE Computer Society Press.

[BBFL93] S. Bandinelli, L. Baresi, A. Fuggetta et L. Lavazza. Requirements and early experiences in the implementation of the SPADE repository. In Shafer Sha93.

[BD80] J. Buxton et L. Druffel. Requierements for an ADA Programming Support Environment: Rationale for Stoneman. In *Proc. of the IEEE Conf. on Computer Software and Aplications*, Chicago, Illinois, October 1980.

[BE86] N. Belkhatir et J. Estublier. Experience with a database of programs. In *Proc. of the ACM SIGPLAN/SIGSOFT Software Engineering Symposium on Software Practical Development Environments*, Palo Alto, CA, December 1986. In it SIGPLAN Notices/, 22(1):84–91, January, 1987.

[BE87] N. Belkhatir et J. Estublier. Software management constraints and action triggering in Adèle program database. In *1st European Software Engineering Conf.*, pages 7–57, Strasbourg, France, Sept. 1987.

[BE89] N. Belkhatir et J. Estublier. Un gestionnaire d'activités de programmation globale pour NOMADE. In *2rd International Workshop on Software Engineering and its Applications*, pages 135–155, Toulouse, France, December 5–9 1989.

[Bel88] Noureddine Belkhatir. *NOMADE : Un noyau d'environnement pour la programmation globale*. Thèse, LGI, Institut national polytechnique de Grenoble, Décembre 1988.

[BEM94] N. Belkhatir, J. Estublier et W. L. Melo. Cooperative work in large–scale software systems. *Journal of Software Maintenance: Research and Practice*, 6(6):319-335, 1994.

[Ber87] Ph. A. Bernstein. Database system support for software engineering: an extended abstract. In *Proc. of the 9th Int'l Conf. on Software Engineering*, Monterey, CA, March 30-April 2 1987.

[BESS94] N. S. Barghouti, W. Emmerich, W. Schafer et A. Skarra. Information management in process-centered software engineering environments. Technical report. GoodStep project. 1994.

[BFGL94] S. Bandinelli, A. Fuggetta, C. Ghezzi et L.Lavazza. chapter 9, pages 223–248. In Finkelstein et al. FKN94, 1994.

[BJS95] M. H. Bohlen, C. S. Jensen et R. T. Snodgrass. Evaluating the completeness

of TSQL2. In S. Clifford and A. Tuzhilin, editors, *Recent Advances in Temporal Databases*, pages 153–174, Zurich, Switzerland, September 1995. Proceedings of the International Workshop on Temporal Databases, Springer Verlag.

[BK90] N.S. Barghouti et G. E. Kaiser. Modelling concurrency in rule-based development environments. *IEEE Expert*, December 1990.

[BK95] N. S. Barghouti et B. Krishnamurthy. Monitoring, modelling, and enacting processes. In *Practical Reusable UNIX Software, edited by B. Krishnamurthy of AT&T Bell Laboratories*, pages 275–298. John Wiley & Sons, Inc, 1995.

[BKK85] Francois Bancilhon, Won Kim et Henry F. Korth. A model of CAD transactions. In *Proceedings of the 11th International Conference on Very Large Data Bases*, pages 25–33, 1985.

[BM88] D. Beech et B. Mahbod. Generalized version control in an object-oriented database. *IEEE conference on data engineering*, 1988.

[BM92] N. Belkhatir et W. L. Melo. TEMPO: a software process model based on object context behavior. In *Proc. of the 5th Int'l Conf. on Software Engineering & its Applications*, pages 733–742, Toulouse, France, December 7–11 1992.

[BMT88] G. Boudier, R. Minot et I. M. Thomas. An overview of PCTE and PCTE+. In *Proc. of the 3rd ACM Symposium on Software Development Environments*, Boston, Massachusetts, November 28–30 1988. In it ACM SIGPLAN Notices, 24(2):248–257, February 1989.

[Boe88] B.W. Boehm. A spiral model for software development and enhancement. *IEEE Computer*, pages 61–72, May 1988.

[BOS91] P. Butterworth, A. Otis et J. Stein. The Gemstone object database. *Communications of the ACM*, 34(10):64–77, 1991.

[BR88] V. R. Basili et H. D. Rombach. The TAME project: towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, 14(5):758–773, 1988.

[BSK94] I.Z. Ben-Shaul et G. E. Kaiser. A paradigm for decentralized process modelling and its realization in the OZ environment. In *Proc. of the 16th Int'l Conf. on Software Engineering*, Sorrento, Italy, May 1994.

[Cas94] R. Casallas. Using triggers in a software configuration manager. In *CLEI 94*, Mexico, Septembre 1994.

[CCS94] C. Collet, T. Coupaye et T. Stevens. NAOS Efficient and modular reactive capabilities in an object oriented database system. In *20th VLDB*, Santiago,

Chile, May 1994.

[Cea95] R. Conradi et al. Process modelling languages. In Derniame et Fugetta DF96.

[CEF96] R. Casallas, J. Estublier et J.M. Favre. Internal specification of Adèle version 4. Technical report, LSR, fevrier 1996.

[CFF94] Reidar Conradi, Christer Fernstrom et Alfonso Fuggetta. *Concepts for Evolving Software Processes*. In Finkelstein et al. FKN94, 1994.

[CFFS92] R. Conradi, Christer Fernstrom, Alfonso Fugetta et Bob Snowdon. Towards a reference framework for fundamental software process concepts. In *Proc. of the 2nd European Workshop on Software Process Technology*, pages 3–17, 1992.

[CI94] J. Clifford et T. Isakowitz. On the semantics of (bi) temporal variable databases. In *4th International conference on Extending Database technology*- LNCS 779, pages 215-230. Springer Verlag, 1994.

[CHCA94] C. Collet, P. Habraken, T. Coupaye et M. Adiba. Active rules for the software engineering platform GOODSTEP. In *Workshop on the intersection between databases and software engineering*, Sorrento, Italy, May 1994.

[CLJ91] R. Conradi, C. Liu et M.L. Jaccheri. Process modeling paradigms: an evaluation. In *Proc. of the 7th Int'l Software Process Workshop*, San Francisco, CA, October 16–18 1991.

[CM91] S. Chakravarthy et Deepak Mishra. Snoop: An expressive event specification language for active databases. Technical Report UF-CIS-TR-91-007, University of Florida, 1991.

[Cou89] W. Courington. *The Network Software Environment*. Sun Microsystems, Inc, 1989.

[CW93] M. Cagan et A. Wright. Untangling configuration management: Mechanism and methodology in CM systems. In *Proc, 4th International workshop on Software Configuration Management*, Baltimore, May 1993.

[Dam95] S. Dami. Bringing the gap between Process Weaver and Adèle concepts. Esprit working paper, project perfect, LGI, February 1995.

[DE95] S. Dami et J. Estublier. Internal specification of apel v3. Proposal, projet perfect, LGI, Dec 1995.

[DF96] JC. Derniame et A. Fugetta, editors. *Software Process: Principles, Methodology, Technology. In preparation*. J. Wiley and Sons, 1996.

[DHB93] H. Dai, J.G. Hughes et D.A. Bell. A distributed real-time knowledge based

system and its implementation using o.o. techniques. In *Int. Conf on Intellignet and Cooperative Information Systems. ICISIS93*, Rotterdam, Nehterland, May 1993.

[Dit89] K.R. Dittrich. The Damokles database system for design applications: its past, its present, and its future. In K. H. Bennett, editor, *Software Engineering Environments: Research and Practice*, pages 151–171. Ellis Horwood Books, Durhan, UK, 1989.

[DK76] F. DeRemer et H. Kron. Programming-in-the-large verus programming in the small. *IEEE Transactions on Software Engineering*, 2:80–86, June 1976.

[DKLM84] V. DonzeauGouye, G. Kahn, B. Lang et B. Melese. Document structure and modularity in Mentor. *ACM Software Engineering Note*, 9(3), May 1984.

[DNR91] M. Dowson, B. Nejmeh et W.. Riddle. Fundamental software process concepts. In A. Fuggeta, C. Ghezzi et R. Conradi, editors, *Proc. of the 1st European Workshop on Software Process Modeling*, pages 16–37, Milan, Italy, 30–31 May 1991. AICA Press.

[EC94] J. Estublier et R. Casallas. *The Adèle Software Configuration Manager*, chapter 4, pages 99–139. In Tichy Tic94, 1994.

[EC95] J. Estublier et R. Casallas. Three dimentional versionning. In Estublier Est95.

[EGK84] J. Estublier, S. Ghoul et S. Krakowiak. Premilinary experience with a configuration control system for modular programs. In *Proc. of the ACM SIGPLAN/SIGSOFT Software Engineering Symposium on Software Practical Development Environments*, Pittsburgh, April 23–25 1984. In it Software Enginnering Notes/, 9(3):149–156, May 1984.

[EKF93] R. Elmasri, V. Kouramajian et S. Fernando. Temporal database modelling: An object-oriented approach. In ACM, editor, *Proceedings of the Conference on Information and Knowledge Management*, 1993.

[Ell92] C. Ellis. A model and algorithm for concurrent access within groupware. Technical Report CU-CS-593-92, University of Boulder at Colorado,, Dep. of Comp. Science, 1992 1992.

[Elm93] A. Elmagarmid, editor. *Database Transactions models for advanced applications*. Series in Database mangement. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1993.

[EN89] R. Elmasri et S. Navathe. *Fundamentals of Database Systems*. the Benjamin Cummings Publishing Company, 1989.

[Est88] J. Estublier. Configuration management: the notation and the tools. In *International Workshop on Software Version and Configuration Control*,

Grassau, FRG, January 27–29 1988.

[Est95] J. Estublier, editor. *Proc. of 5th Int'l Workshop on Software Configuration Management*, Seatle, Washington, USA, May 1995. ACM, Software Engineering Notes.

[EW90] R. Elmasri et G. Wuu. A temporal model and query language for ER databases. In *IEEE Data engineering conference*, 1990.

[Fav95] J.M. Favre. *Une approche pour la maintenance and re-ingenierie globale des logiciels*. These, Institut National Polytechnique de Grenoble, 1995.

[Fei91] P.H. Feiler. Configuration management models in commercial environments. Technical Report CMU/SEI-91-TR-7, Carnegie-Mellon University, Software Enginnering Institure, March 1991.

[Fel79] S. I. Feldman. Make: a program for maintaining computer programs. *Software–Practice and Experience*, 9(4):255–265, April 1979.

[Fer93] C. Fernstrom. Process Weaver: adding process support to Unix. In Osterweil Ost93, pages 12–26.

[FH93] P. H. Feiler et W. S. Humphrey. Software process development and enactment: Concepts and definitions. In Osterweil Ost93, pages 28–40.

[FKN94] A. Finkelstein, J. Kramer et B. Nuseibeh, editors. *Software Process Modelling and Technology*. John Willey and Son inc, Research Study Press, Tauton Somerset, England, 1994.

[FKR94] G. Fowler, D. Korn et H. Rao. n-dfs: The multiple dimentional file system. In Tichy Tic94, pages 135–155.

[FS95] M.-C Fauvet et P.-C. Scholl. Temps and bases de donnees : Concepts temporels pour l'evolution des donnees. Technical Report RR945 I, Institut IMAG Grenoble I, Mars 1995.

[Fug93] A. Fuggetta. A Classification of CASE Technology. *IEEE Computer*, pages 25–38, December 1993.

[Gad88] S. K. Gadia. A homogeneous relational model and quary query language for temporal database. In *ACM transactions on Database Systems*, volume 13, pages 13(4):418–448. 448, december 1988.

[GCCMS96] C. Godart, G. Canals, F. Charoy, P. Molly et H. Skaf. Designing and implementing COO: Design Process, Architecture Style, Lessons Learned. *18th International Conference on Software Engineering*, march 25-29. Berlin-Germany.1996

[GJ91] N. H. Gehani et H. V. Jagadish. Ode as an active database: Constraints and

triggers. In *Proceedings of the 17th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Barcelona*, September 1991.

[GJS93] N. H. Gehani, H. V. Jagadish et Oded Shmueli. Temporal queries for active database support. *Proceedings of the International Workshop on an Infrastructure for Temporal Databases.* R.~T. Snodgrass editor. Arlington, TX. June 1993.

[Gla82] G.R. Gladden. Stop the life cycle - I want to get off. *ACM software Engeneering Notes*, 7(2):35–39, 1982.

[GN93] S. Gadia et S. Nair. *Temporal Databases: A Prelude to Parametric Data*, chapter 2, pages 28–66. In Tanzel et al. TCG^{+}93, 1993.

[GV85] S. K. Gadia et J.H. Vaishnav. A query language for a homogeneous temporal database. In *Proc. of the ACM symposium on Principles of Database Systems*, pages 51–56, March 1985.

[Han89] E.N. Hanson. An initial report on the design of Ariel: a DBMS with an integrated production rule system. *ACM SIGMOD Record*, 18(3):12–19, September 1989.

[HHW95] A. Van Der Hoek, D. Heimbigner et A. Wolf. Does Configuration Management Research have a futur? In Estublier Est95.

[HKD92] C. Hoffmann, B. Kramer et B. Dinler. Multiparadigm description of system development processes. In *Proc. of the 2nd European Workshop on Software Process Technology*, pages 123–137, 1992.

[HN86] A.N. Habermann et D. Notkins. Gandalf: Software development environment. *IEEE Transaction on Software Engineering*, SE-12(12):1117–1127, December 1986.

[Hum89] W.S. Humphrey. *Managing the Software Process*. Addison-Wesley, Reading, Mass., 1989.

[Hum94] Watts S. Humphrey. Process Maturity Model. In John C. Marciniak, editor, *Encycopedia of Software Engineering*, volume 2, pages 851–860. John Wiley and Sons, 1994.

[IBW95] Pat Ingram, Clive Burrows et Ian Wesley. *Configuration Management Tools : a detailed evaluation*. Ovum limited, 1995.

[Is94] IEEE-software. *Measurement-based process Improvement*. Special issue of IEEE software. IEEE, july 1994.

[Ita93] Itasca. *Itasca Distributed Object Database Management System*. Itasca systems, inc, 7850 Metro Parkway, Minneapolis, Minnesota 55425, 1993.

[JPSW94] G. Junkerman, B. Peuchel, W. Schaefer et S. Wolf. *MERLIN: Supporting Cooperation in Software Developemnt Through a Knowledge-Based Environment*. In Finkelstein et al. FKN94, 1994.

[JS92] C. Jensel et R. Snodgrass. Temporal specialization. In *Proceeding of the International Conference of data engineering*. IEEE, 1992.

[JSS93] C. Jensel, M.D. Soo et R. Snodgrass. Unification of temporal data models. Technical Report TR 93-31, Dept. Computer Science. University of Arizona, 1993.

[Kat90] R. H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Computing Surveys*, 22(4):375–408, [12] 1990.

[KBS90] G. E. Kaiser, N. S. Barghouti et M. H. Sokolsky. Preliminary experience with process modeling in the Marvel software development environment kernel. In *Proc. of the 23th Annual Hawaii Int'l Conf. on System Sciences*, pages 131–140, Kona, HI, January 1990.

[KBS93] G. E. Kaiser et I. Z. Ben-Shaul. Process evolution in the Marvel environment. In Shafer Sha93.

[Kel91] M.I. Kellner. Multiple-paradigms:approach for software process modelling. In *Proc. of the 7th Int'l Software Process Workshop*, San Francisco, CA, October 16–18 1991.

[KGW91] W. Kim, N. Ballou J.F. Garza et D. Woelk. A distributed object-oriented database system supporting shared and private databases. *ACM Transactions on Information Systems*, 9(1):31–51, January 1991.

[Kim88] W. Kim. Features of the Orion object-oriented DBMS. In W. Kim and E.H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*. Addison-Wesley, 1988.

[Kin94] R. King, editor. *Workshop on the intersection between databases and software engineering*, Sorrento Italy, may 1994.

[KKS92] M. Kifer, W. Kim et Y. Sagiv. Querying object-oriented databases. In M. Stonebraker, editor, *sigmod*, volume 21, pages 393–402, San Diego, California, June 1992. acm, Acm Press.

[KM81] Kernighan et Mashey. The Unix Programming Enviroment. *IEEE Computer*, April 1981.

[LC85] D. Leblang et R. P. Chase. Configuration management for large scale software development efforts. In *Workshop on Software Engeneering Environment for Programming in Large*, Harwichport, Massachussets, June 1985.

[Leb94] D. Leblang. *The CM Challenge: Configuration Management thats work*, chapter 1, pages 1–37. In Tichy Tic94, 1994.

[LHR95] Christopher M. Lott, Barbara Hoisl et H. Dieter Rombach. The use of roles and measurement to enact project plans in MVP-S. In W. Schäfer, editor, *Proceedings of the Fourth European Workshop on Software Process Technology*, pages 30–48, Noordwijkerhout, The Netherlands, April 1995. Lecture Notes in Computer Science Nr. 913, Springer–Verlag.

[LLPS91] G.M. Lohman, B. Lindsay, H. Pirahesh et K.B. Schiefer. Extensions to Starburst: objects, types, functions, and rules. *Communications of the ACM*, 34(10):94–109, October 1991.

[Lon89] F. Long, editor. *Proc. of Int'l Workshop on Software Engineering Environments*, volume 467 of *LNCS*, Chinon, France, September 18–20 1989. Springer-Verlag, Berlin, 1990.

[Lon93] J. Lonchamp. A structured conceptual and terminological framework for software process engineering. In Osterweil Ost93, pages 41–53.

[LPV89] C. Lecluse, P.Richard et F. Velez. O2, an Object-Oriented data model. *Proc. of ACM SIGMOD 89 Int. Conf. on the management of data*, pages 424–433, 1989.

[Mah94] Alex Mahler. Variants: Keeping things together and telling them apart. In Tichy Tic94, pages 73–97.

[MD89] D. R. McCarthy et U. Dayal. The architecture of an active database management system. In *Proc. of ACM SIGMOD 89*, pages 215–224, Portland, OR, May 1989.

[Mel93] M. Melo. *TEMPO: Un environnement de developpement Logiciel Centre Procede de Fabrication*. PhD thesis, Universite Joseph Fourrier, October 1993.

[Mil89] T. Miller. Configuration management with the NSE. In Long Lon89, pages 99–106.

[MJ82] D.D. MacCracken et M.A. Jackson. Life cycles concepts consideres harmful. *ACM software Engeneering Notes*, 7(2):28–32, 1982.

[NS87] K. Narayanaswamy et Walt Scacchi. Maintaining configurations of evolving software systems. *IEEE Transactions on Software Engineering*, SE-13(3):324–334, March 1987.

[Obe88] P. A. Oberndorf. The commom Ada programming support environment (APSE) interface set (CAIS). *IEEE Transactions on Software Engineering*, 14(6), June 1988.

[Obj94] ObjectStore. *ObjectStore Technical Overview*. Object Design Inc, March 1994.

[Oqu95] “SCALE: Process Modelling Formalism and Environment Framework for Goal-directed Cooperative Processes”, *Proceedings of the 7th International Conference on Software Engineering Environments*, Noordwijkerhout, Pays-Bas, Avril 1995. IEEE Computer Society Press.

[OS95] G. Ozsoyoglu et R. Snodgrass. Temporal and real time databases: A survey. In *IEEE transactions on Knowledge and Data Engineering*, volume 7, August 1995.

[Ost87] L. J. Osterweil. Software processes are software too. In *Proc. of the 9th Int’l Conf. on Software Engineering*, Monterey, CA, March 30-April 2 1987.

[Ost93] L. Osterweil, editor. *Proc. of the 2nd Int’l Con.f on the Software Process*, Berlin, Germany, 25 – 26 February 1993. IEEE Computer Society Press.

[OZG91] F. Oquendo, J.D. Zucker et P. Griffiths. The Masp approach to software process description, instantiation and enaction. In *First European Workshop on Software Process Modeling*, pages 147–155, Milan, Italy, May 30–31 1991.

[PCTE93] ECMA Standard 149, Portable Common Tool Environment (PCTE): Abstract Specification, *2nd Edition, European Computer Manufacturers Association (ECMA)*, Juin 1993.

[PCTE94] ISO Standard 13719, Portable Common Tool Environment (PCTE): Abstract Specification, *International Standards Organization*, 1994.

[Per87] D. E. Perry. Software interconnection models. In *Proc. of the 9th Int’l Conf. on Software Engineering*, pages 61–69, Montery, CA, March 30–April 2 1987.

[Pot84] C. Potts, editor. *Proc. of the 1st Int’l Software Process Workshop*, Egham, UK, february 1984. ispw1.

[PW93] B. Peuschel et S. Wolf. Archictural support for distributed process centered software development environments. In Shafer Sha93.

[Rei90] S. P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, 7(4):57–66, July 1990.

[RK95] D. Rosenblum et B. Krishnamurthy. Generalized event-action handling. In *Practical Reusable UNIX Software, edited by B. Krishnamurthy of AT&T Bell Laboratories*, pages 247–273. John Wiley & Sons, Inc, 1995.

[Roc75] M. Rockhind. The source code control system. *IEEE Trans on Soft. Eng.*, SE-1(4):364–370, Dec 1975.

[Rom91] H. D. Rombach. MVP-L: a language for process modeling in-the-large. Technical Report CS-TR-2709, Dept. of CS, Univ. of Maryland, College Park, 1991.

[Ron94] C. Roncancio. *Regles actives et regles deductives dans les bases de donnees a objets*. PhD thesis, UJF Grenoble I, 1994.

[Roy70] W.W. Royce. Managing the development of large software systems. In *Proc. IEEE Western Computer Conf.*, Los Alamitos, CA, August 1970. IEEE Computer Society Press.

[Sar93] N. L. Sarda. *HSQL : Historical Query Language*, chapter 5, pages 110–140. In Tanzel et al. TCG^{+}93, 1993.

[SC91] Stanley.Y.W. Su et Hsin-Hsin. M. Chen. A temporal knowledge representation model OSAM*/T et its query language OQL/T. In *Proceedings of the VLDB 17*, Barcelona Spain, 1991.

[Sci91] E. Sciore. Multidimensional versioning for object-oriented databases. *Proc. Second International Conf. on Deductive and Object-Oriented Databases*, December 1991.

[Sci93] E. Sciore. Versioning and configuration management in an object-oriented data model. *Journal of Very large Database*, 1993.

[Sec91] Stuart Sechrest. Attributed-based naming of files. Technical Report CSE-TR-78-91, University of Michingan, 1991.

[SG94] K. Dittrich S. Gatziu, A. Geppert. The samos active dbms prototype. Technical Report TR-94-16, Zurich University, 1994.

[Sha93] W. Shafer, editor. *Proc. of the 8th Int'l Software Process Workshop*, Germany, 1993. IEEE Computer Society Press.

[Sim89] I. Simmonds. Configuration management in the PACT software enginnering environment. *ACM Software Engineering Notes*, 14(7):118–121, November 1989.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh et S. Potamianos. On rules, procedures, caching and views in database systems. In *Proc. of ACM SIGMOD 90*, pages 281–290, Atlantic City, NJ, May 1990.

[Sno93] R. T. Snodgrass. *An Overview of TQuel*, chapter 6, pages 141–182. In Tanzel et al. TCG^{+}93, 1993.

[Sno95] R. Snodgrass. Temporal object-oriented databases: A critical comparison. In Won Kim Editor, editor, *Modern Data Base systems*. Addison Wesley, 1995.

[SV91] S. Sarkar et V. Venugopal. Transaction mechanisms for software

environment databases. In *Proc. of the 24th Annual Hawaii Int'l Conf. on System Sciences*, pages 511–518, Kona, HI, 1991. IEEE Computer Society, Software Track, v. II.

[TCG+93] A. Tanzel, J. Clifford, S. Gadia, S. Jajodia, A. Segev et R. Snodgrass, editors. *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings, 1993.

[Tea94] GoodStep Team. The GoodStep project: General object-oriented databases for software engineering processes. In K. Ohmaki, editor, *Proc. of the Asia-Pacific software engineering conference*, pages 410–420, Tokyo, Japan, 1994. IEEE Computer Society Press.

[Tit et al88] R. N. Taylor et it et al. Foundations for the Arcadia environment architecture. In *Proc. of the 3rd ACM Symposium on Software Development Environments*, Boston, Massachusetts, November 28–30 1988.

[TGD95] D. Tombros, Andreas Geppert et Klaus Dittrich. Seamen: implementing process-centered software development environments on top of an active database management system. Technical Report TR 95-03, Zurich University, 1995.

[Tho89a] I. Thomas. PCTE interfaces: supporting tools in software engineering environments. *IEEE Software*, 6(6):15–23, November 1989.

[Tho89b] I. Thomas. Version and configuration management on a software enginnering database. *ACM Software Engineering Notes*, 14(7):23–25, November 1989.

[Tic80] Walter F. Tichy. *Software Development Control Based on System Structure Description*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1980.

[Tic82] W.F. Tichy. Design, implementation, and evaluation of a revision control system. In *Proc. of the 6th Int'l Conf. on Software Engineering*, Tokyo, Japan, September 1982. IEEE Computer Society.

[Tic92] W.F. Tichy. Programming-in-the-large: Past, Present, and Futur. *icse14*, pages 362–366, 1992.

[Tic94] W. Tichy, editor. *Configuration Management*. Trends in Software. J. Wiley and Sons, Baffins Lane, Chichester West Sussex, PO19 1UD, England, 1994.

[TOC93] G. Talens, C. Oussalah et M. Colinas. Version of simple and composite objects. In *Proc. of the Very Large Database Systems*, pages 62–72, Dublin-Irlande, 1993.

[Was89] A. I. Wasserman. Tool integration. In Long Lon89.

[WD93] Gene T.J. Wuu et Umeshwar Dayal. *A Uniform Model for Temporal and Versioned Object-oriented Databases*, chapter 10, pages 230–247. In Tanzel et al. TCG^{+}93, 1993.

[WF91] Kurt C. Wallnau et Peter H. Feiler. Tool integration and environment architecture. Technical Report CMU/SEI-91-TR-11, Carnegie-Mellon University, Software Enginnering Institure, May 1991.