N° d'ordre :02/2003-M/IN

République Algérienne Démocratique et Populaire

Université des Sciences et de la Technologie Houari Boumediene

Faculté d'Electrique et Informatique

Thèse

Présentée pour l'obtention du diplôme de Magistère en informatique

Spécialité : **Informatique**

par : **HADJ SADOK SALAH EDDINE**

.

Evolution des Modèles de Procédés Logiciels

Soutenue le 28/04/2003, devant le jury :

Mme. A.AISSANI, Maître de conférence à l'USTHB	**Président**
Mr. M. AHMED-NACER, Maître de conférence à l'USTHB	**Directeur de thèse**
Mme. Z. ALI MAZIGHI, Maître de conférence à l'USTHB	**Examinateurs**
Mme. BOUKALA. Maître de conférence à l'USTHB.	

Dédicaces

À mes très chers parents.

À mes frères et mes sœurs

Remerciements

Je remercie tout d'abord Madame A. AISSANI, Maître de conférences à l'USTHB, qui m'a fait le grand honneur en acceptant de présider le jury de cette thèse.

Je tiens à exprimer ma reconnaissance à Monsieur AHMED NACER, maître de conférences à l'USTHB, mon directeur de thèse. Sans sa bonne orientation, ses suggestions pertinentes, ses explications précieuses et sa rigueur scientifique et sa patience, ce travail n'aurait jamais pu aboutir.
Qu'il trouve ici l'expression de ma profonde gratitude.

Je remercie Madame ALI MAZIGHI et Madame M. BOUKALA, maîtres de conférences à l'USTHB, qui m'ont honoré en acceptant de juger ce travail.

Un grand merci pour mes amis et à tous ceux qui m'ont aidé.

Table des matières

INTRODUCTION

Il est acquis que la qualité des procédés logiciels conditionne la qualité du produit à réaliser. Par procédé logiciel on entend l'ensemble des activités faisant intervenir des équipes de personnes (souvent nombreuses), des outils et des techniques, dans l'objectif d'assurer le développement et la maintenance des systèmes logiciels. Les activités mises en jeu sont de natures différentes (spécification du système, la conception, le prototypage, l'implantation, la validation, les tests, le contrôle de la qualité, la maintenance, etc.). L'enjeu des compagnies de développement de logiciels est de bien organiser les développements, de mettre en oeuvre les techniques et les outils les plus appropriés et d'adopter les bonnes pratiques. La représentation sous forme d'un modèle d'un projet informatique est une première action d'amélioration, qui consiste à spécifier les étapes du procédé de développement, préciser les intervenants, leurs missions et leurs activités.

En dépit de l'évolution des technologies de l'information et de la communication, des progrès restent à faire comme le démontrent des études récentes. Sur un ensemble de projets de complexité faible, si 72% d'entre eux se terminent à peu près dans les temps, 9% présentent un retard supérieur à 6 mois, 1% un retard supérieur à 1 an et 3% sont arrêtés. Pour des projets plus complexes, seuls 53% d'entre eux respectent les délais, 24% présentent un retard supérieur à 6 mois, 10% un retard supérieur à 12 mois et 7% sont annulés [Hendrick 96].

L'intérêt porté aux procédés logiciel ne date pas d'aujourd'hui. Historiquement, ces derniers ont joué un rôle important dans le domaine du génie logiciel. C'est d'ailleurs à la suite de "la crise du logiciel" apparue vers la fin des années soixante que la discipline du génie logiciel est née [Sommerville 92]. L'objectif est d'offrir les moyens pour gérer la complexité des procédés et contribuer à l'amélioration de la qualité des produits.

Au début, les travaux autour des procédés logiciels ont été orientés vers la définition de modèles de cycle de vie. Un modèle de cycle de vie décrit les étapes globales à suivre pour développer ou

maintenir un produit logiciel. Parmi les modèles les plus connus citons le cycle de vie en *cascade* [Royce 70] et le cycle de vie en *spirale* [Boehm 86].

Grâce à l'utilisation des modèles de cycle de vie, on a pu améliorer, même si ce n'est que de manière partielle, la compréhension et la documentation du procédé. Malheureusement, on s'est vite rendu compte de l'inadéquation de ces modèles pour une prise en compte efficace des procédés logiciels.

Pour remédier aux limitations des modèles de cycle de vie plusieurs travaux visant à offrir un meilleur support au développement de logiciels, se sont succédés. Ces travaux ont été orientés vers le développement d'une nouvelle génération d'outils qui s'approche plus des activités des programmeurs. Il s'agit des *environnements intégrés de génie logiciel* [Lonchamp et al. 92]. Cette notion se réfère à des collections intégrées destinées à assister la production du logiciel.

Malgré le succès relatif de cette génération d'environnements, son apport vis à vis du procédé logiciel reste limité.

Pour remédier aux limitations des environnements intégrés de génie logiciel, les travaux postérieurs ont été orientés vers le développement d'une deuxième génération d'environnements, connue sous le vocable d'*environnements de génie logiciel centrés procédés*. Cette nouvelle génération d'environnements a fait (et continue à faire) l'objet de nombreux travaux de recherche : Marvel [Kaiser and al. 88], EPOS [Conradi et al. 94a], Prism [Madhavji and Schafer, 1991], APEL [Dami et al.97, Amiour 99], ... etc.

Un *environnement centré procédé (ECP)* est un système dans lequel la manière dont le logiciel est fabriqué (doit être fabriqué) est définie de façon explicite et avec suffisamment de détails. Cette description, appelée le *modèle de procédé*, est exprimée dans un formalisme approprié appelé *langage de modélisation de procédés.* Le langage offre des concepts pour décrire les rôles, les humains, les activités, les produits manipulés, les contraintes, etc

La nouveauté par rapport à la génération précédente est que l'assistance devient globale à l'ensemble du procédé de développement (et non seulement à la phase de programmation) et explicite, au travers de modèles interprétés par l'environnement. Alors que dans les environnements classiques, la connaissance sur les buts et les stratégies du développement n'existe que dans la tête des développeurs, elle est ici partagée entre les utilisateurs et l'environnement [Lonchamp et al. 92].

Problématique et limitations du support existant

Les procédés logiciels ont une vie très longue. Il est donc nécessaire de fournir une ou plusieurs méthodes pour assister le(s) administrateur(s) de l'environnement dans l'amélioration des descriptions de ces procédés afin de les adapter aux nouveaux besoins, de corriger les incohérences trouvées en cours d'exécution, de modifier, ajouter ou supprimer certaines contraintes [Dowson, 1993].

De plus un environnement centré procédés doit être capable de prendre en compte les modifications effectuées d'une façon dynamique [Huff and Kaiser, 1991]. C'est à dire permettre de modifier le modèle de procédé logiciel au cours de l'exécution des activités et gérer les incohérences entre les nouvelles définitions et les instances du procédé logiciel en cours d'exécution. En effet, certaines instances doivent continuer leur exécution selon les descriptions du modèle précédent, et d'autres instances ne doivent prendre en compte les nouvelles descriptions qu'à partir d'un certain moment spécifique.

Au cours de la dernière décennie, un investissement important a été réalisé dans la recherche et le développement des systèmes centrés procédés logiciels et surtout l'aspect évolution. L'objectif est d'assister le procédé logiciel de manière plus efficace. Malheureusement, la génération actuelle de systèmes se heurte à de nombreux problèmes, ce qui a limité les gains réalisés et son utilisation dans l'industrie.

Notre travail de recherche consiste à proposer une solution au problème d'évolution des modèles de procédés logiciels. Nous présentons une approche d'évolution qui permet de modifier le modèle et ses instances de façon dynamique.

Ce document est organisé comme suit :

- Le deuxième chapitre est un état de l'art. Il décrit l'évolution des environnements de génie logiciel durant ces vingt dernières années en basant sur deux aspects : *Intégrité* et *Généricité*. Nous présentons une typologie des systèmes actuels et nous étudierons les types d'environnements disponibles, dans le cadre de cette typologie, en soulignons leurs apports et leurs faiblesses.

- Le troisième chapitre est une introduction aux environnements centrés procédés. Des concepts concernant ces environnements ainsi que des exemples de systèmes existant seront donnés.

- Le quatrième chapitre est composé de deux parties. La première partie propose d'étudier l'aspect Evolution des modèles de procédés logiciels, nous citons quelques approches d'évolution dans des systèmes existants. La deuxième partie fait le point sur les travaux menés dans le domaine du versionnement. Les différents concepts de base nécessaire à la définition et à la manipulation de versions d'objets sont décrits, ensuite nous présentons quelques systèmes de gestion de versions dans différents domaines d'application.

- Le chapitre cinq porte sur notre proposition pour l'évolution des modèles de procédés logiciels. Un formalisme de modélisation ainsi qu'une approche d'évolution des modèles seront présentés. Deux types d'évolutions seront présentées, Evolution d'instances du modèle de procédé logiciel et Evolution du schéma.

- Et nous terminons par une conclusion et des perspectives.

Chapitre II
Etat de l'art

Ce chapitre propose d'étudier *les environnements intégrés de génie logiciel*. L'objectif n'est pas de présenter une étude exhaustive, il s'agit plutôt de présenter tout d'abord leurs évolution durant ces vingt dernières années en basant sur deux aspects : *Intégrité* et *Généricité*. Dans ce cadre, Nous présentons une typologie des systèmes actuels et nous étudierons les types d'environnements disponibles, dans le cadre de cette typologie, en soulignant leurs apports et leurs faiblesses.

1. Introduction

Le concept d'environnement se réfère à la collection d'outils matériels et logiciels qu'un développeur, ou une équipe de développeurs, utilise pour produire des systèmes logiciels.
Depuis 20ans, la diversité des types d'outils utilisés lors de cette production s'est accrue dans des proportions importantes. Cette prolifération d'outils s'est accompagnée d'une multiplication des incompatibilités de toutes natures (par exemple relatives à la méthode suivie ou la représentation des informations), des recouvrements et duplication des fonctionnalités, des différences de style de manipulation, etc. Le problème de l'*intégration* s'est donc imposé comme une question majeure. Des formes très divers d'intégration ont été proposés [Lonchamp et al, 92].

On appel Environnement Intégré de Production de Logiciel (*EIPL*), les collections intégrées d'outils destinées à assister la production de logiciel.

Un EIPL doit offrir à ses utilisateurs :

- une diversité de services, qui touchent aux différentes phases du cycle de vie des logiciels (de la spécification des besoins à la maintenance) et aux différents rôles assurés (concepteur, contrôleur qualité, etc.);
- une réelle homogénéité de ces services ;

- un grand nombre de propriétés, assurant un niveau élevé de qualité [LEH 87]: portabilité, extensibilité, partage, distribution, sécurité, etc.

L'émergence des *EIPL* s'inscrit dans un contexte général de crise, résultant d'un déséquilibre croissant entre la demande en logiciel et l'offre en moyen de production.

1. *l'évolution de la demande :*

   L'évolution de la demande en logiciel se caractérise par une véritable "explosion" :

   - une croissance quantitative exponentielle, accompagnée d'une faible progression de la productivité des développeurs.
   - l'élargissement des domaines d'applications (contrôle de procédés, systèmes embarqués, gros systèmes de gestion en temps réel, etc.) et l'augmentation spectaculaire de la complexité des applications traitées.
   - l'augmentation des attentes des utilisateurs : la micro informatique et les stations de travail ont popularisé des modes de travail plus riches et plus agréables, les utilisateurs attendent de toutes les applications des performances ergonomiques du même ordre, associé à une richesse fonctionnelle toujours plus grande, dans un contexte de fiabilité et de sécurité de très bon niveau.

2. *L'état de l'offre :*

   L'état de l'offre en moyens de production du logiciel laisse apparaître un grand "éparpillement", avec :

   - une multitude d'outils classiques, spécialisés dans une fonction unique ou un ensemble restreint de fonctionnalités,
   - un manque de maturité et de portée générale des "techniques de pointes" pour le développement du logiciel : spécifications formelles et synthèse de code, réutilisation, prototypage, nouveau "styles" de programmation, etc.

3. *La situation de crise :*

   Le déséquilibre patent entre les besoins et les moyens disponibles pour les satisfaire est une cause essentielle de la situation de crise :

   - 15% des projets ne produisent rien et échouent complètement,
   - des dépassements de 100 ou 200% par rapport aux prévisions sont observés,
   - des coûts de maintenance plus de deux fois supérieurs aux coûts de développement sont mesurés.

Il est admit aujourd'hui que cette "crise du logiciel" ne se résoudre pas par l'effet d'une solution miracle, mais grâce à une convergence d'efforts très divers. Citons :

- l'analyse des processus de développement, suivant les domaines d'application, pour en acquérir une meilleure maîtrise
- l'automatisation de la production autant qu'il est possible,
- la réutilisation à grande échelle de composants existants,
- l'amélioration de la communication homme/machine,
- le recours à des systèmes experts pour assister la production de logiciel.

Les *EIPL* doivent jouer un rôle clé de fédérations de ces solutions partielles, au travers des différents types d'outils que ces approches génèrent.

La percée prévisible des *EIPL* résulte d'une convergence de plusieurs facteurs :

- l'arrivée à maturité des technologies et composants de base nécessaires : stations de travail, systèmes distribués, bases de données spécialisées,…
- une meilleure compréhension et prise en compte de la diversité des tâches incluses dans la production du logiciel et de l'importance croissante des tâches de coordination et de gestion, résultant de la complexité croissante des applications. On distingue :

  - la programmation détaillée ("*programming-in-the-small*"), c'est-à-dire les tâches les plus traditionnelles, au niveau de l'unité de programme, comme l'édition, la compilation, les tests.
  - la programmation globale ("*programming-in-the-large*"), c'est-à-dire les tâches liées à la multiplicité des unités de programmes, comme la conception de haut niveau ou encore la gestion des versions et des configurations
  - la programmation coopérative ("*programming-in-the-many*"), c'est-à-dire les tâches liées au caractère collectif du développement, comme la gestion et le suivi de projet ou encore la gestion des équipes de production.

  Tous les outils associés à ces trois familles de tâches doivent coopérer ce qui contribue puissamment à imposer l'idée d'un outillage intégré et homogène.

- une meilleure compréhension et prise en compte des processus de production des logiciels, de leur diversité et de leur complexité.

2. les Environnements intégrés de production de logiciels :

Les *EIPL* ont connu une constante évolution durant ces vingt dernières années. Différents étapes ont caractérisé leur évolution. Nous présentons dans ce qui suis une classification selon la *technique d'intégration* et le *degré de généricité*. [Lonchamp et al. 92]

2.1 La technique d'intégration

Les études techniques consacrées à l'intégration dans les environnements distinguent plusieurs axes d'intégrations, touchons tout ou partie du modèle conceptuel.

Exemple, Wasserman[Was89] retient cinq :

1) *intégration au niveau de la plate-forme : les* outils de l'environnement doivent s'exécuter sur un même «système opératoire »;

2) *intégration au niveau de la présentation :* l'interface utilisateur doit être uniforme pour tous les outils de l'environnement ; cet aspect concerne aussi bien les mécanismes (systèmes de fenêtrage, gestionnaire de fenêtrage et boîte à outils) que les règles de manipulation ("look and feel guidelines"),

3) *intégration au niveau des données :* les outils doivent pouvoir se partager des objets communs et doivent pouvoir s'échanger les objets qu'ils génèrent ; cette forme d'intégration se situe au niveau des structures,

4) *intégration au niveau de contrôle :* les outils doivent pouvoir se notifier des événements et déclencher des activations, pour travailler de concert,

5) *intégration au niveau des procédés :* l'environnement joue pleinement son rôle quand il sert à imposer un procédé de fabrication du logiciel bien défini.

Dans ce qui suit, On entend par le terme intégration, l'intégration des données et des processus.

Les principales solutions d'intégration ayant marqué l'évolution du domaine et qui se basent sur l'intégration des données et des processus sont :

- *La première solution d'intégration (I1) :* Elle repose sur une simple convention de représentation des objets produits par le développement et partagés par les outils ; comme les fichier Unix ; simples suite de caractères, et les représentations plus structurées, tels les arbres de syntaxe abstraite des environnements structurels, type Mentor[Don84]

- *La deuxième solution d'intégration (I2) :* Elle repose sur une base d'objets spécialisée commune, c'est-à-dire sur le partage généralisé et systématisé des objets produits par le développement mais aussi de leurs inter-relations et d'informations descriptives sur ces objets ; comme APSE [Bux80] "Ada Programming Support Environment", dont la structuration en couche autour de la base d'objets,

- *La troisième solution d'intégration (I3) :* Elle ajoute aux précédentes une modélisation explicite du procédé de fabrication du logiciel. L'intégrité des objets produits peut dépendre également des autorisations d'utilisation des outils accordées aux divers intervenants, aux différents stades du processus de fabrication. Généralement, il s'agit de reconnaître que les outils sont utilisés et interagissent dans le contexte d'un procédé de fabrication donné ; il n'est pas suffisant de s'accorder sur les structures de données manipulées ; il faut s'accorder également sur le procédé au sein duquel les outils vont être mis en œuvre.

2.2. Le degré de généricité

A côté des environnements dédiés (D), il est indispensable de disposer de techniques économiques de construction d'environnements, c'est-à-dire de système générique (G), adaptable à tel ou tel contexte.

Deux voies s'ouvrent :

- la définition de noyaux normalisés ou « structures d'accueil », intégrant les fonctionnalités communes à une large classe d'environnements (G1) ; les environnements sont construits par couches autour de ces noyaux (Fig. II.1),

- la définition de générateurs d'outils (ou d'environnements) (G2), paramétrés par des modèles de langages, etc.

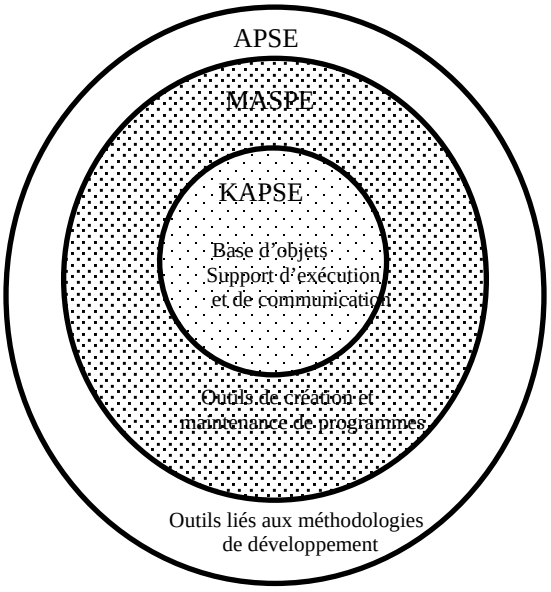


Fig. II.1 : **Structure du APSE**

Le tableau suivant représente les différentes combinaisons d'intégration et de généricité :

Tableau 1. les critères de classification et leurs modalités	
I : Intégration	I1 : convention de représentation des objets partagés ou convention d'échange d'objets
	I2 : base d'objets spécialisée
	I3 : modélisation explicite du procédé de fabrication du logiciel
G : Généricité	D : environnements dédiés
	G : environnements génériques ■ G1 : noyaux normalisés ■ G2 : générateurs

2.3. La typologie des EIPL selon les deux facteurs (intégration et généricité) :

Les *EIPL* sont répartis en six catégories principales :

Tableau 2. *La typologie des EIPL*

Catégorie	Familles	Exemples
I1/D	environnements interactifs environnements structurels environnements faiblement couplés	Interlisp, Smalltalk80 CPS, Mentor HP SoftBench
I1/G	• noyaux normalisés (I1/G1) ∗ autour d'un SGF ∗ autour d'un standard d'échange	 Unix (noyaux) Unix-PWB (environnement) ATMOSPHERE, ESF
	• générateurs d'environnement structurel (I1/G2)	Metal/Mentor, Synthesizer Generator
I2/D	environnements intégrés fermés de développement environnements intégrés fermés de gestion du développement	Toolpack, Smile PMDB, SKB
I2/G	Structure d'accueil	PCTE, CAIS(normes) Emeraude (structure d'accueil) EAST, Entreprise2 (environnements dérivés)
I3/D	environnements transformationnels	REFINE
I3/G	environnements paramétrés par des modèles de procédés logiciels	Marvel, ALF, EPOS, …etc.

2.3.1 La catégorie I1/D :

IL s'agit le plus souvent d'environnements fermés, proposant un ensemble restreint de mécanismes dédiés à la programmation détaillée, pour un langage de programmation donnée et pour un utilisateur unique ;ils sont construits autour d'une convention de représentation des objets partagés impliquant des règles plus ou moins contraignantes.

Les deux familles les plus représentatives sont :

1. celle des environnements interprétatifs, comme Interlisp [Tei 81b] ou Smalltalk80 [Gol 83] : la représentation interne des programmes, partagée par tous les outils est « ad hoc » ; pratiquement aucune règle n'est imposée ; l'outil central est un interprète du langage

cible ; le langage et l'environnement, écrits dans ce langage, ne sont pas réellement séparables, ce qui offre un maximum de flexibilité à l'utilisateur.

2. celle des environnements structurels, tel que CPS [Tei81a] ou Mentor [Don84] : la représentation interne partagée par les outils est du type arbre syntaxique abstrait; cette représentation est optimisée pour un certain nombre de tâches, car elle évite d'analyser plusieurs fois la forme textuelle des programmes; mais elle peut rendre difficile l'intégration d'autres mécanismes; l'outil central est un éditeur structurel qui impose la règle de détection des erreurs "à la source". En plus des deux familles, on trouve la famille des environnements dédiés basés sur une convention d'échange d'objets au sein d'une architecture faiblement couplée, type HP Soft-Bench[ISO89].

2.3.2 La catégorie I1/G :

Il s'agit d'environnements génériques, conçus autour d'une convention de représentation commune.

a) *Les noyaux normalisés (I1/G1)*

- Les plus classiques sont constitués autour des systèmes de gestion de fichiers (SGF), des systèmes d'exploitation type Unix. Les environnements bâtis autour de ces noyaux et diffusés proposent une famille complète de mécanismes conçus pour travailler ensemble et qui partagent les mêmes conventions d'organisation des données et les mêmes conventions de travail. Le documenter Workbench[Bel84] en est un exemple typiques.

- D'autres, sont constitués autour de mécanismes standards pour l'échange d'objets ; c'est le cas par exemple des projets européens ATMOSPHERE [Gra90] et ESF [Fer89].

b) Les générateurs d'environnements structurels (I1/G2)

La description formelle du langage cible étant fournie en paramètre, ces systèmes sont indépendants de celui-ci : Ils peuvent permettre aux utilisateurs de créer leurs propres environnements structurels et travailler uniformément dans des contextes multilingues.
Exemple : Mental/Mentor [KAH 83], ou Synthesizer Generator [Rep 84]

2.3.3 La catégorie I2/D :

Il s'agit d'environnements *fermés*, construits autour d'une base de données (base d'objets) centrale. Les deux familles retenues sont :

1) Les environnements *intégrés fermés pour le développement :*

Le partage généralisé des produits du développement et de leurs caractéristiques permet d'imposer des règles de coordination entre les membres d'une équipe de développeurs ; les systèmes sont dédiés à des domaines d'application précis pour lesquels ils fournissent des ensembles de mécanismes adaptés. Par exemple Smile[Kai87] concerne le développement en "C" et offre des règles d'assistance assez évoluées mais codées dans les mécanismes et les structures, ce qui en fait un système fermé.

2) Les environnements intégrés pour la gestion du développement :

Ces systèmes sont destinés à étendre les environnements traditionnels (du type "boîte à outils") en les complétant par une gestion intégrée du développement ;

2.3.4 La catégorie I2/G :

Il s'agit de systèmes *ouverts*, servant de noyau pour la construction d'environnements et intégrant les fonctionnalités de base des environnements : gestion des objets dans une base d'objets centralisée, gestion des activités, gestion de l'interface utilisateur, de la distribution, et de la sécurité. On les appelle "*structure d'accueil* ".

2.3.5 La catégorie I3/D :

Les environnements "*transformationnels*", tels REFINE [Mar89] est un bon exemple d'environnements dédiés ou l'intégration se fait principalement autour du concept de procédé de fabrication et de son automatisation. Initialement, les programmes sont écrits dans une forme abstraite et modifiés par une séquence de transformations en une forme concrète, exécutable de manière raisonnablement efficace.

L'outil central est le "moteur de transformation", que l'on peut avoir comme un compilateur de spécifications de très haut niveau en code exécutable.

2.3.6 La catégorie I3/G :

Les environnements paramétrés par un modèle de procédé logiciel, tel Marvel[Kai88] et ALF[BEN89], mettent eux aussi l'accent sur les règles ;mais au lieu d'être codées dans les mécanismes et les structures, elles sont fournies de manière indépendante et explicite ; il s'agit donc d'une manière de construire des environnements adaptables à tel ou tel contexte : il faut fournir un modèle de procédé adéquat et les mécanismes et structures qui le concrétisent.

2.4. Les forces et faiblesses des principales approches

2.4.1 La catégorie I1/D

a) Les environnements interprétatifs

Dans ces systèmes, l'exécutif et les outils sont conçus spécifiquement pour supporter un langage donné. Exemple : SmalTalk [Gol 83], C++.

Les forces de ces environnements résident dans leur homogénéité, leur souplesse et leur convivialité.

Leurs faiblesses en termes de couverture du cycle de vie, de partage et de distribution, d'assistance et de sécurité sont :

- ces systèmes ne concernent en général que le codage ;
- ces systèmes sont mono-language ;
- beaucoup de ces systèmes sont mono-utilisateur ;
- pratiquement aucune restriction (règle) n'est imposée au développeur ; ces environnements apparaissent en l'état plus adapté à un usage individuel par des développeurs de haut niveau qu'à une production industrielle de logiciel.

b) Les environnements structurels

A l'origine, ces systèmes furent conçus pour permettre la saisie et l'édition des programmes en s'appuyant sur la structure du langage. Ceci permet, par exemple, la génération

automatique des mots clés ou encore le saut du débuts d'une structure (ex : begin, parenthèse ouvrante) à la fin de cette même structure (ex : end, parenthèse fermante).

Ces éditeurs syntaxiques sont complétés par des mécanismes d'exécution, de déboguage, d'analyse sémantique, etc., pour constituer des environnements structurels.

Les forces des environnements structurels résident :

- dans le concept même de manipulation structurelle,
- dans la possibilité d'offrir des vues multiples (textuelles et graphiques) des mêmes structures,
- dans la possibilité d'attacher des informations sémantiques aux structures et de les rendre accessibles aux utilisateurs.

Les faiblesses sont assez similaires à celles des environnements interprétatifs :

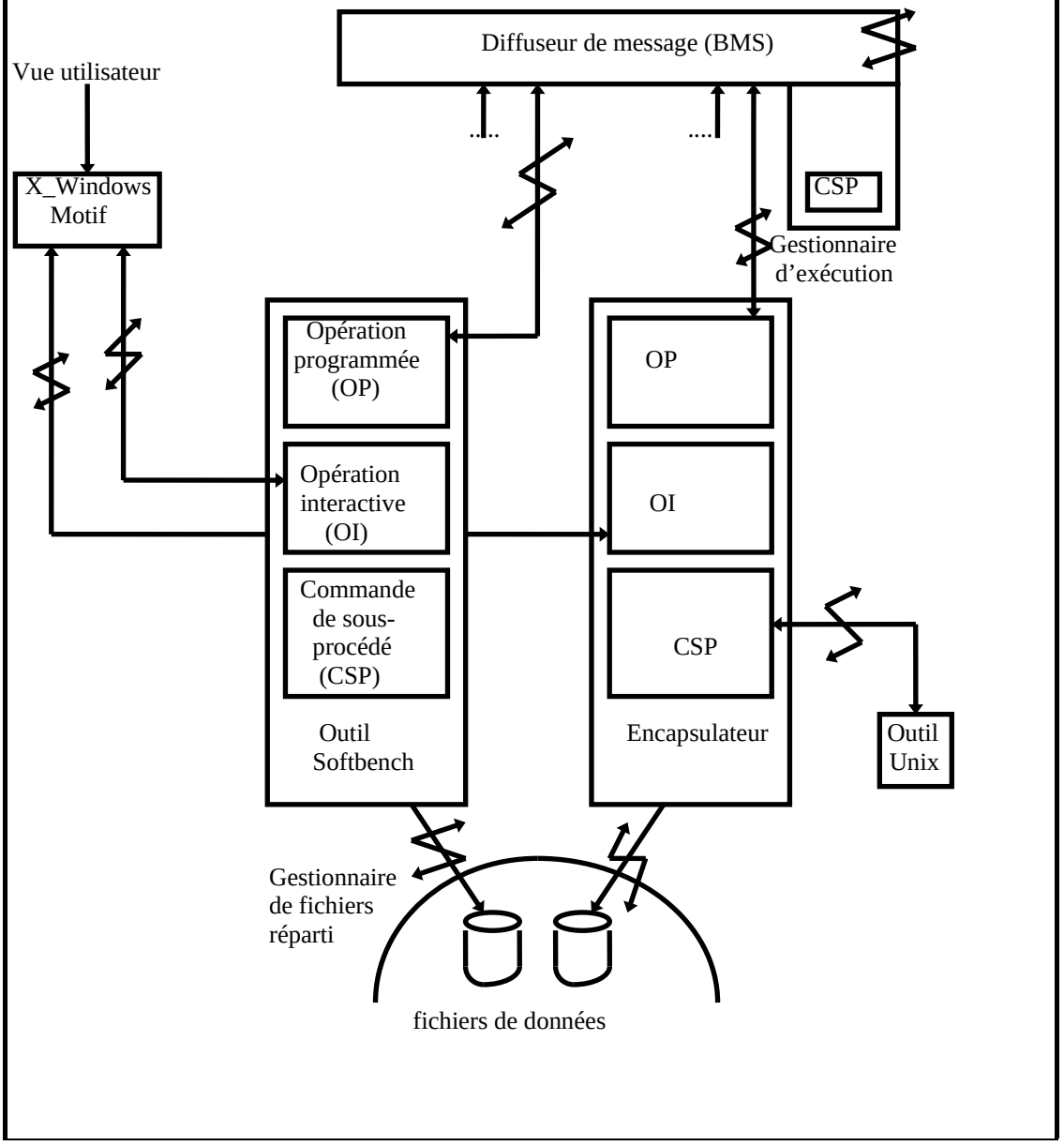
- des services offerts liés essentiellement à la phase de codage,
- une utilisation généralement mono-utilisateur, car les techniques de stockage persistant des grandes structures et la gestion des accès multiples concurrents à ces structures posent encore des problèmes mal maîtrisés.

Ces systèmes sont utilisés avant tout comme des aides à l'apprentissage de la programmation; malgré leur disponibilité commerciale, leur impact dans l'industrie reste actuellement assez faible [DAR 87].

c) Les environnements faiblement couplés

Le produit HP SoftBench est un bon exemple d'environnement faiblement couplé [Iso89]. Il intègre un ensemble d'outils, structuré en classes, répondant chacun à une fonctionnalité spécifique et qui doivent satisfaire à deux conditions pour être de « bon citoyens » de l'environnement : répondre aux requêtes faisant appel à leur fonctionnalité et notifier la fin de leurs activations.

Fig. II.2 : L'architecture faiblement couplée HPSoftBench



Les avantages de cette approche sont :

- l'environnement est ouvert,
- l'extensibilité est très facile, y compris en cours d'activité,
- les outils peuvent bénéficier de toute convention d'échange, le BMS ne cherchant pas à interpréter les données échangées.

Ces inconvénients :

- le problème général des insuffisances de l'encapsulation ;
- aucune analyse et teste des messages ne sont effectués, ce qui interdit de contrôler et d'automatiser le procédé de développement, en état actuel,
- l'absence d'une base de données homogène rend beaucoup plus difficile l'implantation de capacités d'interrogation et de navigation au sein des objets de l'environnement.

2.4.2 La catégorie I1/G1

Nous détaillerons ici qu'une partie de la première famille I1/G1, celle des noyaux. A savoir, les noyaux constitués par les SGF des systèmes d'exploitation et les boîtes à outils construites autour de ces noyaux.

L'approche boîte à outils s'appuie sur l'infrastructure des systèmes d'exploitation et en particulier leur SGF, utilisés comme support des structures partagées au sein des environnements.

Les boîtes à outils imposent relativement peu de règles, en dehors de celles codées dans les mécanismes ; l'écriture de "shell scripts" est un moyen pratique mais peu puissant pour définir des règles. Les mécanismes courants concernent la phase de codage (éditeurs, compilateurs, éditeurs de liens, débogueurs) et la gestion de versions et de configurations (ex : RCS [Tic85]).

Avec de tels environnements, les décisions essentielles (décision d'utiliser ou non les outils, manière de les utiliser, de les enchaîner, etc.) restent entièrement du ressort des utilisateurs.

Ces boîtes à outils constituent des environnements indépendants des langages de programmation, incluant des mécanismes simples pour les tâches de la programmation globale.

Les forces qui ont fait leur popularité sont leur adaptabilité et leur portabilité.

Leurs principales faiblesses, l'hétérogénéité :

- hétérogénéité des langages de commande des outils,
- hétérogénéité des structures annexes gérées par les outils, comme les Makefiles [Fel79] ou les Deltas[Tic85], d'autre part ;rien ne peut garantir, par exemple, que le contenu des Makefiles est cohérent avec la réalité des fichiers existant et de leurs dépendances.

Le niveau d'assistance procuré par ce type de système pour la production et la maintenance de gros système logiciels reste relativement limité.

2.4.3 La catégorie I2/D

a) Les environnements de gestion du développement

Ces systèmes ont pour but de donner aux développeurs et gestionnaires de projets une vision claire et juste de l'état d'avancement des projets. Ils conservent, dans une base de données, les caractéristiques des composants du logiciel (modules, documentation, etc.), de leurs multiples relations et des contraintes sémantiques qui les lient. Les objets gérés dans la base sont indépendants des objets réels qu'ils modélisent ; ceci constitue à la fois la force et la faiblesse de ces approches :

- les systèmes résultants sont simples de compréhension et d'utilisation,

- les informations gérées par ces systèmes sont censées représenter l'état du développement ; n'étant pas saisies "à la source", rien ne peut le garantir ; de plus les redondances par rapport à d'autres informations du SGF (ex : date de dernière mise à jour des composants) ou d'outil spécialisés (ex : les dépendances dans les Makefiles) peuvent se multiplier, avec tous les risques d'incohérence qui en découlent. Quand une incohérence est détectée, aucune réparation directe ne peut être assurée en raison de l'indépendance entre objets descriptifs et entités réelles du développement : seul un avertissement aux développeurs peut être généré.

b) Les environnements intégrés dédiés

Il s'agit de systèmes spécialisés, souvent fermés, construits autour d'une base centrale qui recueille tous les produits et la connaissance liée au développement.

Ces systèmes apportent des réponses intéressantes pour des types d'application précis. Les structures, mécanismes et règles sont ad hoc et pas toujours transposables efficacement à d'autres types d'applications.

2.4.4 La catégorie I3/D

Dans cette partie, on trouve les environnements transformationnels. REFINE[MAR89] de Reasoning Systems Inc, est l'un des premier environnements transformationnels commerciaux qui ont effectivement vu le jour.

L'approche transformationnelle consiste à transformer une spécification formelle initiale en un programme exécutable, par applications successives de règles de transformation préservant la cohérence des descriptions, de telle sorte que le code final satisfasse les spécifications initiales. Généralement, les systèmes transformationnels exploitent un catalogue de règles concernant :

- des connaissances sur les données (arithmétiques, logiques, les listes, les ensembles, etc),
- des connaissances sur la programmation (stratégies disponibles pour résoudre des problèmes),
- des connaissances sur l'optimisation des programmes (suppression de récursivité, etc.),
- des connaissances sur la sélection automatique des structures de données.

Ces systèmes sont souvent appelés systèmes à base de connaissances et ils restent encore éloignés de ce que doit être un environnement complet de développement.

2.4.5 La catégorie I3/G :

Cette catégorie concerne les environnements paramétrés par des modèles de procédés logiciels, et qui sera décrit en détail dans le chapitre suivant.

Conclusion :

Malgré le succès relatif des environnements intégrés, leurs apports vis à vis du procédé logiciel reste limité. D'une part, ces outils sont principalement destinés à assister les activités des développeurs de façon individuelle. D'autre part, ils se focalisent souvent sur le produit sans s'intéresser véritablement à la manière dont il est fabriqué (i.e les activités).

Chapitre III

Les Environnements centrés procédés

1. Introduction

Il s'agit d'environnements paramétrés par des modèles de procédés ou d'Environnement Intégré de Production Assistée de Logiciel (*EIPAL*).

Ces environnements se proposent de piloter et d'assister les développeurs dans l'application des procédés logiciel, en mémorisant et en exploitant les informations et règles qui les définissent.

Contrairement, aux environnements classiques ; ou la connaissance sur les buts et les stratégies du développement n'existe que dans la tête du développeur ; l'assistance pour le développement des logiciels est programmée. Elle est partagée entre les utilisateurs et l'environnement, et adaptable à tous les cas de figure (règles spécifiques à chaque entreprise et à chaque projet) et susceptible d'évolutions dynamiques, en cours de développement.

2. Terminologie du domaine

A ce jour, il n'existe pas de normes concernant la terminologie utilisée dans le domaine des procédés logiciels. Les concepts présentées ici sont inspirées de [Conradi et al. 92, Conradi et al. 94b, Amiour 99] et sont utilisés dans les différents ECPs.

*Procédé logiciel ou Processus logiciel (Software process)*est l'ensemble des activités faisant intervenir des équipes de personnes, des outils et des techniques dans l'objectif d'assurer le développement et la maintenance des systèmes logiciels.

Activité : une opération atomique ou composite ou une étape dans un procédé, qui contribue à la réalisation d'un objectif. Elle vise généralement à générer ou modifier un ensemble de produits.

Exemple : Spécification des besoins du système, Conception, Compilation.

Produit : est une donnée manipulée par ou résultante d'une activité.

Exemple : Code source, Document de spécification, Plan de test.

Ressource : un bien de la compagnie nécessaire à la conduite des activités.

Exemple : les agents humains, les outils logiciels (compilateurs, éditeurs, etc.).

Un *modèle de procédé logiciel* est une description générique du procédé, spécifiant des types d'activités, des types de produits, des prescriptions et contraintes méthodologiques. Le modèle de procédé est exprimé dans une notation de modélisation appropriée appelé *langage de modélisation de procédés ou formalisme de modélisation de procédés.*

Formalisme de modélisation (Process Modeling Langage) : Langage textuel ou graphique qui permet de décrire le modèle de procédé.

Instance du modèle (Process Instance) : est une adaptation de ce modèle à un projet spécifique en vue de gérer ses données et ses différentes étapes de réalisation.

Exécution du modèle (Enactment) : l'acte d'interpréter/exécuter un modèle de procédé exécutable. L'exécution du modèle peut être effectuée par des machines ou par des agents humains.

Un *environnement centré procédé (ECP)* est un système dans lequel la manière dont le logiciel est fabriqué (doit être fabriqué) est définie par un modèle de procédé logiciel. [Amiour 99].
Le modèle du procédé étant au cœur de l'environnement, ce dernier se charge de son interprétation afin de guider les utilisateurs, les assister et automatiser des tâches (figure III.1).

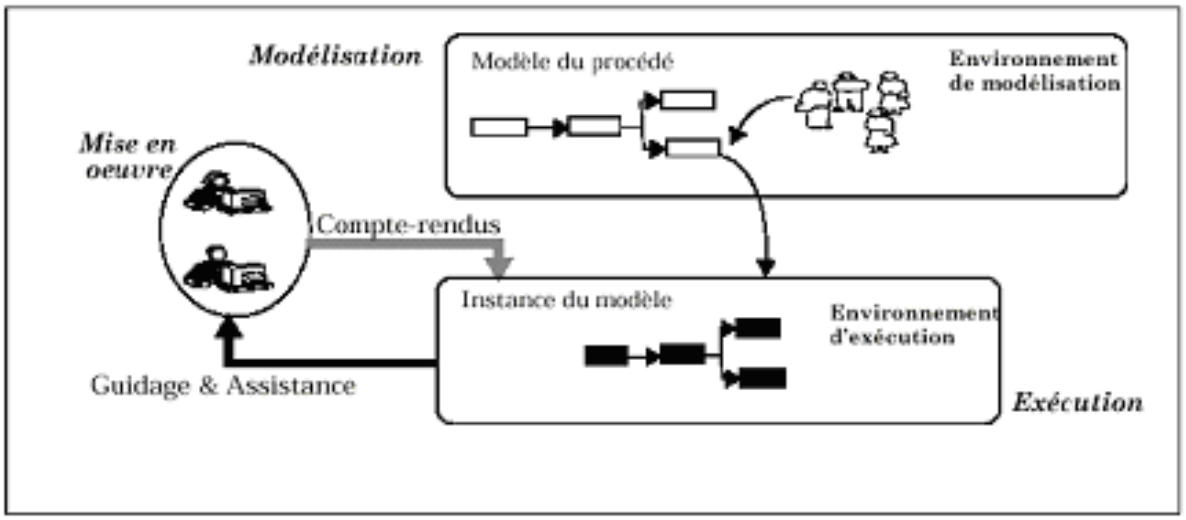


Figure III.1 : Principe des environnements centrés procédés logiciels

Les environnements de procédés logiciels sont des environnements de développement de logiciels qui fournissent un *langage de modélisation de procédé* (*PML*) dans lequel un processus logiciel spécifique à un projet donné et spécifié par l'administrateur du processus et exécuté par un *moteur d'exécution* correspondant [Ben Shaul 95].

3. Objectifs et Fonctionnalités des Environnements centrés procédés :

Les objectifs principaux des environnements centrés procédés sont ambitieux. Ils peuvent être résumés dans les points suivants [Curtis et al. 92, Karen 96] :

La compréhension et la communication des modèles. Il s'agit de représenter le procédé de développement sous une forme compréhensible, de façon à permettre aux participants de bien communiquer [Huff 96], de comprendre le travail à accomplir, leur rôles et responsabilités, leur interactions, etc.

L'analyse des procédés. L'existence d'une représentation explicite et rigoureuse du procédé logiciel permet d'estimer le coût et les effets des décisions qui peuvent être prises, et de comparer entre différentes alternatives pour la conduite du projet. De cette façon, on peut identifier les problèmes et les points faibles pour effectuer ensuite les améliorations nécessaires. Les approches d'amélioration peuvent par exemple être dirigées par les objectifs et les priorités internes de l'organisation de développement [Basili et al. 92].

La réutilisation des modèles. Les modèles de procédés sont le fruit d'efforts et d'investissements importants. Un procédé bien défini, peut être réutilisé dans plusieurs situations similaires et permet de capitaliser l'expérience acquise durant la conduite des différents projets.

Le guidage et l'assistance des participants. L'interprétation du modèle par l'environnement offre une assistance aux utilisateurs dans l'exécution de leur travail.

L'automatisation des tâches. Il s'agit d'automatiser l'exécution de certaines tâches répétitives qui ne nécessitent pas une intervention humaine (activités de compilation, invocation d'outils, etc.).

4. Un scénario d'utilisation

Ce scénario décrit les interactions entre un utilisateur et un ECP :

EIPAL : Login
Utilisateur : Arthur
EIPAL : Rôle
Utilisateur : Programmeur
EIPAL : Projet
Utilisateur : X 24
EIPAL : Mot de passe
Utilisateur : |||||
EIPAL : J'ai compilé avec succès tous les modules liés au module MOD-TR-2 que vous avez édité au cours de votre dernière session.
Vous êtes en retard pour la tâche 1.2.3. Vous souvenez-vous de son objet ?
Utilisateur : Non
EIPAL : Voici la spécification du paquetage que vous devez coder.
Voulez-vous le coder maintenant ?
Utilisateur : Oui
(L'EIPAL ouvre une fenêtre d 'édition pour le paquetage ;Arthur code le paquetage et referme la fenêtre)
EIPAL : Avez-vous fini la codification ?
Utilisateur : Oui
EIPAL : Attention **!**le ratio (nombre de commentaires/nombre d'instructions) est inférieur au standard défini par le contrôleur qualité du projet.
Voulez-vous documenter votre code maintenant ?

Utilisateur : Non
EIPAL : Vous pouvez soit :
arrêter le travail avec ce paquetage,
le compiler et remettre à plus tard la gestion de configuration,
le compiler et exécuter la gestion de configuration.
A cause d'une surcharge temporaire du réseau, je ne conseille pas la solution 3.
Quel est votre choix ?
Utilisateur : 2
EIPAL : Fin de compilation, pas d'erreurs
.....

Ce court scénario illustre plusieurs fonctionnalités importantes : le contrôle, le pilotage, la prise d'initiative par le système, le conseil, l'exploitation.

5. Les formalismes de modélisation des procédés

Les ECPs possèdent une description explicite du procédé qui est le modèle de procédé. Ce modèle est exprimé dans un ou plusieurs langages de modélisation de procédés (*PML*) ou formalisme de procédé [Conradi et Liu 99]. Ces langages permettent de décrire les différents éléments et les différentes étapes de procédé.

Un PML peut être formel, semi-formel ou informel [Conradi et Jaccheri 98]. Un PML formel est donné avec une syntaxe et une sémantique formelle. La syntaxe des langages semi-formels est formelle mais leur sémantique ne l'est pas ce qui veut dire qu'ils ne sont pas exécutables. Les langages informels n'ont ni sémantique ni syntaxe formelle. Les langages naturels sont des exemples de PML informels.

Il existe plusieurs approches pour les formalismes de modélisation de procédés. Beaucoup de ces approches sont basées sur des approches existantes telles que les graphes, les langages de programmation ou les réseaux de Pétri.

Les principales approches utilisées dans la modélisation de procédés sont :

1. **L'approche par les règles**
   Dans cette approche, les informations concernant les activités et les tâches d'un procédé de développement de logiciels sont modélisées par des règles. Les techniques

d'Intelligence Artificielle et des bases de données actives sont utilisées (planification, déclencheurs). Les outils sont intégrés dans l'environnement par l'utilisation de PRE et de POST conditions.

Les règles peuvent être différentes suivant l'implémentation choisie par le système (raisonnement par chaînage avant/arrière, règles d'événement-condition-action). Cette approche est utilisée dans Marvel [Barghouti, 1992], Epos [Conradi et al., 1991] et Adèle [Belkhatir et al., 1992].

2. **L'approche procédurale :**

L'approche procédurale, proposée par Osterweil [Osterweil, 1987], est telle que le modèle complet du procédé logiciel est défini sous la forme d'un programme. Ce modèle est décrit par un langage écrit par l'administrateur de l'environnement avant l'activation du procédé logiciel. Il spécifie de manière détaillée comment le procédé de développement doit être conduit.

3. **L'approche par les graphes :**

Cette approche utilise les graphes pour décrire les procédés logiciels. Elle se base sur le fait que ces procédés sont similaires aux systèmes temps réel. Des techniques pour la spécification des systèmes en temps réel tels que les réseaux de Pétri ont donc été adaptées et étendues pour modéliser ces procédés. Cette approche est utilisée dans Prism [H.Madhavji and Shafer, 1991]

4. **l'approche multi-paradigme :**

La modélisation multi-paradigme se base sur la constatation que le procédé logiciel est particulièrement complexe à décrire et que l'utilisation d'un langage basé sur un seul paradigme ne répond pas aux besoins. Dans cette approche, on essaie de combiner plusieurs paradigmes, et on ne peut donc parler de paradigme de base pour ce type de modélisation.

Par exemple, le système JIL [Sutton et al. 97] combine l'utilisation de l'approche procédurale et l'approche par les règles. La première est utilisée pour décrire l'exécution des activités du procédé. Il est basé sur le langage Ada avec des extensions spécifiques aux domaines des procédés logiciels (introduction du concept d'activité, description du parallélisme entre activités)

L'approche par les règles est utilisée pour décrire la réaction des activités à des événements qui se produisent durant l'exécution du procédé (changement d'état des produits et des activités, les exceptions)

6. Quelques environnements centrés procédés existants :

Cette section est une brève description de quelques environnements centrés procédés existants.

6.1 Adèle/Tempo [Belkhatir et al., 93, Melo, 1993]

TEMPO est un prototype de recherche réalisé dans le cadre du projet Adèle [Estublier, 1991]. Il permet de décrire et d'exécuter des procédés logiciels. Un modèle de procédé logiciel peut donc être décrit par un ensemble de types de procédé logiciel. Un type de procédé logiciel a une définition récursive. C'est une agrégation de plusieurs types de procédé logiciel. Les concepts de spécialisation/généralisation et de composition/décomposition, définis dans le domaine de la modélisation des données, sont également utilisés pour modéliser les procédés logiciels.
Par exemple, une activité de mise au point d'un document de conception d'un module peut se décomposer en deux sous-procédés

1) un sous-procédé modélisant l'activité de modification qui permet d'apporter des modifications au document de conception ;

2) un sous-procédé modélisant l'activité de révision de la modification qui permet d'approuver les modifications faites sur le document de conception.

```
Monitordesign  ISA PROCESS ;
        CONTROL md ;
                sub = ModifyDesign ;
                card = 1 ;
        CONTROL rd ;
                sub = ReviewDesign ;
                card = 1 ;
END_OF  MonotorDesign ;
```

```
ModifyDesign ISA PROCESS
        ATTRIBUTES
 begin_date = DATE :=now
        end_date    = DATE ;
        deadline      = DATE ;
        METHODS    ...
        RULES         ...
END_OF  ModifyDesign ;

ReviewDesign  ISA PROCESS ;   ...
```

L'exemple ci-dessus décrit le type de procédé logiciel *MonitorDesign* qui représente l'activité qui coordonne la modification d'un document de conception d'un module. Ce type de procédé est composé, dans le formalisme TEMPO, de deux sous-procédés :

ModifyDesign et *ReviewDesign.*

ModifyDesign est le type décrivant le procédé de modification du document de conception, et *ReviewDesign* pour la révision de cette modification.
Il est possible de définir, pour chaque type de procédé logiciel, des attributs, des méthodes et des contraintes temporelles en utilisant le formalisme des règles d'événement-condition-action.

Concept de rôle

Un concept de *rôle* est introduit de manière à intégrer plusieurs comportements et plusieurs propriétés, venant de différents types d'objets, dans une unique perspective : Un type de *rôle* peut faire référence à différents types d'objets.
En utilisant ce concept, TEMPO permet d'unifier le traitement d'un ensemble hétérogène d'objets. L'avantage de cette approche est qu'un ensemble de types d'objets, avec des caractéristiques statiques et dynamiques différentes, peut être vu durant l'exécution d'une étape spécifique du procédé logiciel d'une manière cohérente et homogène.

Un type de procédé logiciel peut avoir plusieurs types de rôles; un procédé logiciel devient une liste de rôles, où chaque type d'objet peut jouer différents types de rôles. Comme conséquence, deux objets d'un même type peuvent être gérés différemment dans un même procédé logiciel. Parallèlement, un même objet peut jouer différents rôles dans différents procédés logiciels.

6.2. APEL [Dami et al., 97, Amiour 99]

APEL est le fruit des travaux de l'équipe de M. Jacky Estublier de l'université de Grenoble en France. Son langage a deux représentations, une *représentation graphique* et une *représentation textuelle*. La première permet une plus grande compréhension pour les utilisateurs et pour une description de haut niveau du processus, la deuxième permet son exécution [Dami et al. 97].

L'environnement inclut les éléments suivants :

- Un éditeur graphique pour la modélisation du procédé.
- Un convertisseur qui transforme la représentation graphique en représentation textuelle.
- Un compilateur qui transforme la notation textuelle en un formalisme exécutable (le formalisme exécutable d'Adel est utilisé)
- Un serveur d'état commun qui maintient l'état courant du processus en exécution.
- Un ensemble d'autres services tels que le gestionnaire de dialogues ou le gestionnaire de contrôle.

Les concepts de base de APEL :

Deux concepts sont représentés dans le formalisme de APEL, les aspects **statiques** et les aspects **dynamiques**. Les activités, les produits et les agents sont des exemples d'aspects statiques. Les aspects dynamiques sont le flux de contrôle, le flux de données et les diagrammes d'état [Dami et al. 97].

- Le **flux de contrôle** représente le séquencement et la concurrence des activités dans l'environnement.
- Le **flux de données** illustre la transition du produit entre les différentes activités. Ce produit étant la sortie d'une activité et l'entrée d'une autre activité.
- Les **diagrammes d'états** représentent l'évolution d'un élément de l'environnement (produit, activité ou agent). L'évolution d'un élément d'un état à l'autre est générée par un *événement* et est appelée *transition*.

Pour représenter les diagrammes d'états, une notation a été adoptée, inspirée de Harel's statecharts[D.Harel, 96], avec quelques adaptations. Statecharts constituent une manière simple et expressive pour décrire graphiquement le comportement des différentes entités de l'environnement.

La représentation graphique du langage de APEL est faite avec UML, les diagrammes de flux de données et les statecharts.

Les diagrammes de flux de données permettent de représenter l'échange de données entre les différentes activités. Cependant, elles ont été étendues dans le contexte de APEL pour tenir compte de la coopération entre les activités.

La notation textuelle de langage de APEL permet à la fois de faciliter l'exécution de ce langage et de représenter des entités qui ne peuvent pas être représentées de manière graphique. Elle permet de décrire des types de données dans un paradigme objet où chaque objet possède des attributs et des méthodes et peut éventuellement hériter d'un autre objet.

L'exemple qui suit est la déclaration dans APEL de l'objet *Entity* qui est la racine (héritage) de tous les objets de l'environnement.

```
Type Entity
        Attributes
            String  entityUid ;
            Static   String Description ;
            String  ProcessName ;
            String  ProcessInstanceId ;
        Methods
            ChangeState (State Es,  State Ed);
End;
```

Exemple de définition d'un type dans le langage de APEL.

6.3. EPOS [Conradi and Letizia, 1993], [Westby et al. 94]

EPOS (*expert system for program and system development*) est un travail de l'équipe de M. Conradi de l'université de Trondheim en Norvège. EPOS est un environnement support procédé qui offre :

- Un langage de modélisation de procédés logiciels orientée objet appelé : SPELL (*Software process evolutionary language*). SPELL est basé sur les règles de production et conçu au-dessus de PROLOG. Il décrit les différentes entités du procédé par des types et des instances.
- Un schéma du modèle de procédé logiciel initial (un ensemble de types).
- Un ensemble d'outils de procédé.
- Une base de données client/serveur EPOS-DB

Un schéma de modèle de procédé logiciel fournit une description d'un groupe d'éléments de procédé, exemple - des activités, des produits, outils, rôles humains, projet, avec interconnexion. Un schéma de modèle de procédés dans EPOS est un ensemble de classes et de relations.

Le modèle de EPOS est décomposé en deux sous-modèles, le *modèle d'activité* et le *modèle de produit* [Jaccheri et al. 92]

Toutes les entités de l'environnement sont décrites par des objets et forment une hiérarchie d'objets. La racine de cette hiérarchie est la classe *Entity*. Deux autres classes descendent de la classe *Entity*, il s'agit des classes *TaskEntity* et *DataEntity* qui sont respectivement les racines de la hiérarchie des tâches et la hiérarchie des produits.

Chaque projet est une tâche spéciale qui s'exécute dans un contexte d'une transaction. Chaque changement effectué au sein d'un projet est gardé localement jusqu'à la validation de la transaction [Jaccheri et al. 92].

Les outils du modèle de procédé sont les outils qui supportent la gestion et l'exécution des modèles de procédés. Le gestionnaire de modèle de procédés, le moteur d'exécution sont des exemples de ces outils.

La base EPOS-DB est une base orientée objet servant à stocker les modèles de procédés dans un contexte transactionnel et versionné.

Les modèles de procédés exécutables accèdent à cette base via DML ou DDL (Data Manipulation Language/ Data Definition Language).

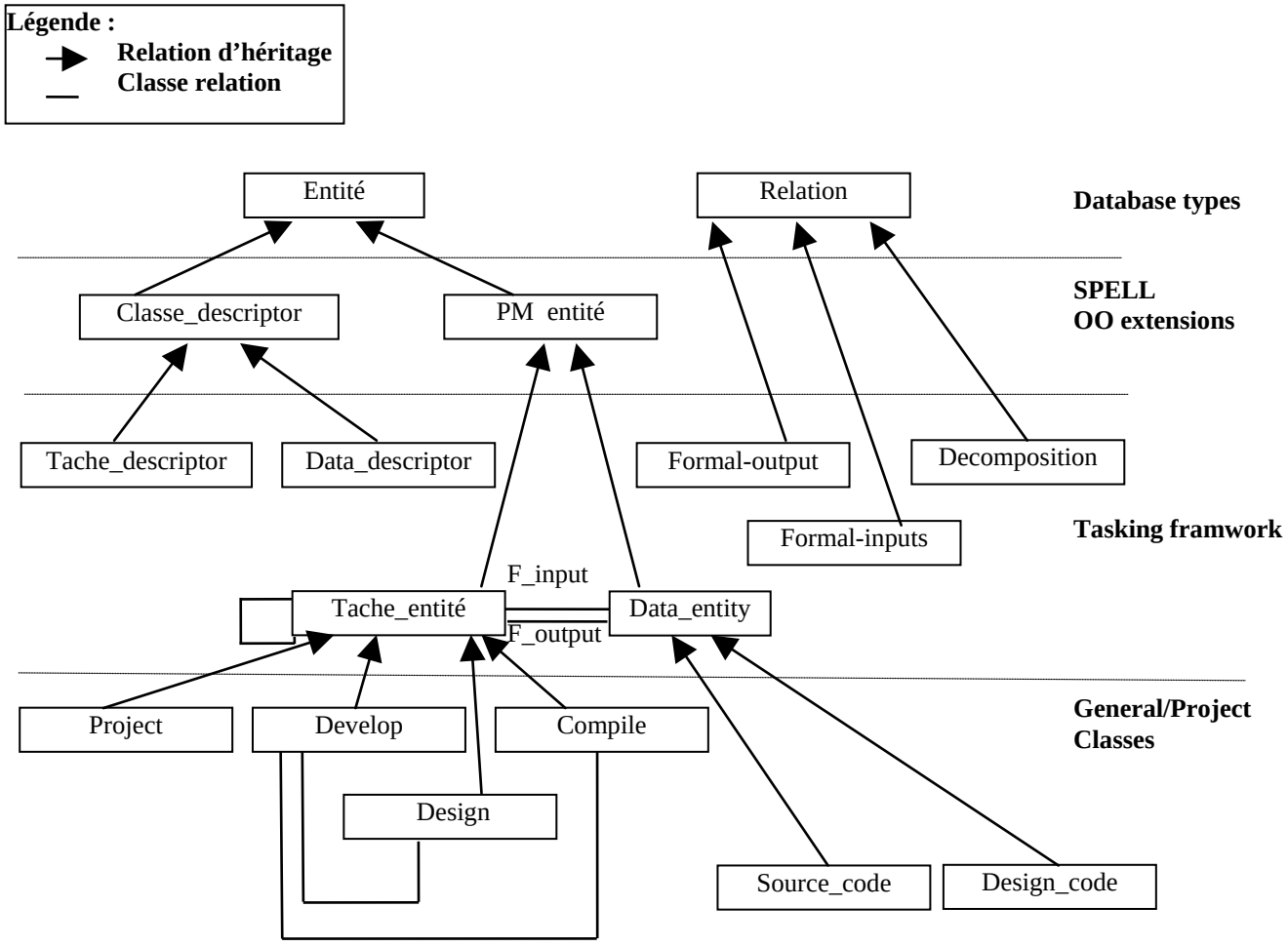


Figure : **Les classes d'EPOS**

6.4. Le système Prism [Madhavji and Schafer, 1991]

Prism est un environnement de génie logiciel dirigé par les procédés logiciels. Il supporte le développement, l'instanciation et l'exécution de modèles de procédés logiciels. Son architecture est conçue de façon à supporter les différentes étapes du processus de production de modèles de

procédés logiciels ainsi que leurs exécutions. L'environnement Prism est formé de trois composants essentiels : Le composant de *simulation* qui permet de simuler un modèle adapté aux besoins du programmeur et du projet, un composant d'*initialisation* qui permet d'instancier le modèle de façon incrémentale et un composant *opération* qui contient les modèles de procédés logiciels décrits en *Funsoft* ainsi que l'interpréteur de ces modèles.

6.5. Le système Marvel

Marvel est un systèm américain de l'université de Columbia. Commencé au début des années 80, Marvel [Barghouti and Kaiser, 1990] est l'un des premiers systèmes dont le modèle de procédé logiciel est basé sur des règles de production qui permettent de modéliser et de contrôler l'exécution des procédés logiciels. Une règle est composée d'une précondition pour spécifier les conditions d'exécution de la règle, d'une action qui est un appel à un outil ou à une opération propre à Marvel et d'une postcondition qui peut éventuellement changer l'état de la base d'objets.

L'exécution d'un procédé logiciel en Marvel se fait par un mécanisme de chaînage avant et arrière sur la base de règles. Chaque fois que l'utilisateur active une règle, la précondition de la règle est analysée. Si la précondition n'est pas vérifiée, un mécanisme de chaînage arrière essaye de satisfaire la précondition de la règle en enchaînant toutes les règles qui peuvent satisfaire une partie de cette précondition. Chacune de ces règles dont la précondition n'est pas vérifiée conduit à d'autres enchaînements. Le chaînage arrière s'arrête lorsque la précondition de la règle demandée par l'utilisateur est satisfaite.

7. Conclusion :

Les environnements centrés procédés sont des environnements dans lesquels la manière de produire un logiciel est définie de manière explicite. Cette partie a introduit quelques concepts généraux concernant les procédés logiciels, les modèles de procédés logiciels et les environnements centrés procédés.

Quelques exemples d'environnements centrés procédés ont été décrits dans cette partie (Tempo/Adel, APEL, EPOS, Prism et Marvel), mais il existe une multitude d'autres travaux sur ces environnements. Ce chapitre ou même ce document ne pourraient faire une étude exhaustive de tous ses environnements.

Chapitre IV :

EVOLUTION DES MODELES DE PROCEDES LOGICIELS

Ce chapitre se compose de deux parties. La première partie propose d'étudier l'aspect Evolution des modèles de procédés logiciels, nous citons quelques approches d'évolution dans des systèmes existants. La deuxième partie fait le point sur les travaux menés dans le domaine du versionnement. Les différents concepts de base nécessaire à la définition et à la manipulation de versions d'objets sont décrits, ensuite nous présentons quelques systèmes de gestion de versions dans différents domaines d'application.

Partie A : Evolution des modèles de procédés logiciels

1. Introduction :

Les procédés logiciels ont une vie très longue. Il est donc nécessaire de fournir une ou plusieurs méthodes pour assister le(s) administrateur(s) de l'environnement dans l'amélioration des descriptions de ces procédés afin de les adapter aux nouveaux besoins, de corriger les incohérences trouvées en cours d'exécution, modifier, ajouter ou supprimer certaines contraintes (Dowson, 1993).

De plus, un AGL doit être capable de prendre en compte les modifications effectuées d'une façon dynamique. C'est à dire permettre de modifier le modèle de procédé logiciel au cours de l'exécution des activités et gérer les incohérences entre les nouvelles définitions et les instances du procédé logiciel en cours d'exécution. En effet, certaines instances doivent continuer leur exécution selon les descriptions du modèle précédent, et d'autres instances ne doivent prendre en compte les nouvelles descriptions qu'à partir d'un certain moment spécifique etc.

Résoudre le problème de l'évolution des modèles de procédés logiciels nécessite souvent une réponse aux questions suivantes : quand, comment et quelle partie du modèle de procédés doit être modifié ? et comment peut on analyser et guider ces modifications ?.

La figure IV-1 répond sur la question « Quelles sont les parties du modèle de procédés à changer », elle représente les différents Niveaux impliqués dans la modélisation et l'évolution du modèle de procédés logiciel.

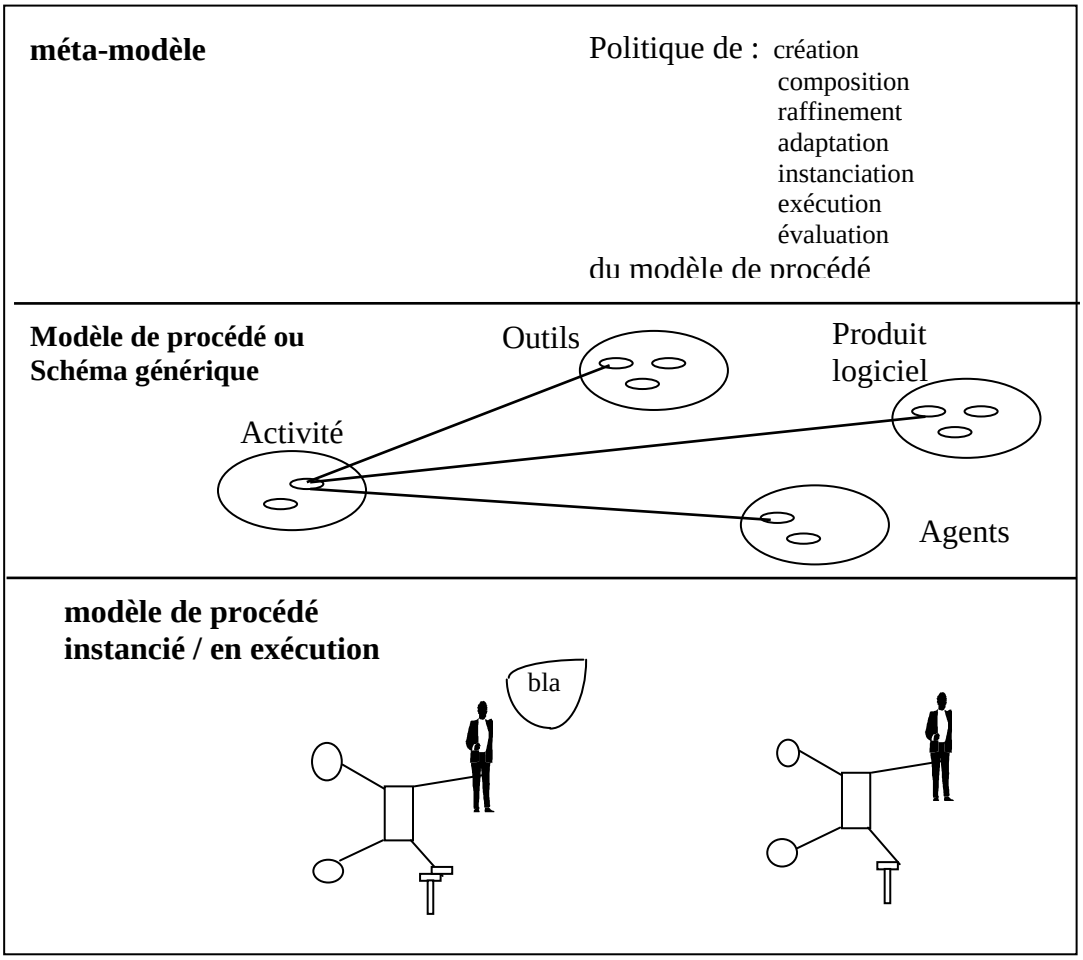


Figure IV-1 : Les fragments du modèle du procédé logiciel : candidats pour des changements

Niveau 1 représente les fragments du modèle instancié / en exécution {les lignes dénotent les flux de données entres les produits (cercles) et les activités (rectangles)}.

Niveau 2 représente le schéma générique, consiste en sous-schéma avec des relations et des contraintes.

Niveau 3 représente le meta-modèle c'est à dire : les règles et les procédures pour la définition et la manipulation des modèles de procédés logiciels.

Chaque niveau du modèle dans la figure1 pourrait s'avérer insuffisant et aurait besoin d'être changé. Cependant, rares sont les cas où le méta-modèle serait modifié car source beaucoup d'incohérences. Donc l'évolution du modèle de procédés logiciel se ramène alors à la modification du niveau modèle de procédés et de ses instances pendant l'exécution des activités.

1. Le modèle des procédés instanciés / en exécution :

- Les fragments du modèle de produit devraient toujours être modifiables, puisque l'évolution des produits est l'objet principal de l'environnement de production logiciels,
- L'ajout ou le changement des activités sont plus difficiles.
- La modification (changement) des outils ou des affectations des rôles aux humains doivent aussi être considérés ;

Tels changements aboutissent à :

(1) des changements dans le schéma du modèle de procédé ou (2) des changements temporaires dans des modèles instanciés/ en exécution

2**. Le schéma générique :** des changements dans des schémas des modèles de procédés génériques ou spécifiques consistent en : des changement dans des descriptions d'éléments simples ou des contraintes sur leurs interactions. Des éléments à ce niveau décrivent des éléments au niveau du modèle de procédé instancié / en exécution, un changement d'un de ces éléments peut avoir un impact non seulement sur des éléments de même niveau mes aussi sur des éléments du niveau inférieur.

3. Le meta-modèle : le meta-modèle peut être trouvé insuffisant dû aux rétroactions à partir des niveaux inférieurs. Ces changements sont très délicats car ils ont un impact sur la façon auxquels des éléments sont manipulés aux niveaux inférieurs et au niveau meta-modèle lui-même exemple : Comment changer une procédure qui fixe les changements du meta-modèle lui même ? La modification du méta modèle est très rare.

La recherche dans le domaine des procédés logiciels est récent, d'où le peu d'expérience acquise dans le domaine de l'évolution des modèles de procédés logiciels. Parmi les travaux citons Adel/Tempo, Prism, Marvel.

2. Les différentes approches d'évolution

2.1. Mécanisme d'évolution dans Tempo/Adel

L'évolution dans Tempo [Belkhatir and Ahmed-nacer, 1995] intervient à deux niveaux de granularité complémentaires ;

1) L'évolution du modèle de procédé logiciel correspond à la création /modification des définitions des types procédé logiciel et des types rôle. Cette évolution se fait par ajout de rôles de manière dynamique. Comme Tempo possède une approche dirigée par les objets, la sémantique d'une instance de procédé logiciel est définie par le comportement des objets dans les différents rôles qu'ils sont amenés à jouer dans cette instance. Par conséquent, toute modification du comportement des objets, par la mise à jours des rôles correspondants, entraîne une modification de la sémantique de cette instance.

2) Au niveau hiérarchie de procédés logiciels : A cause de la longue durée de vie des activités de développement de logiciels, il est nécessaire d'ajuster le modèle de procédé logiciel pour prendre en compte de nouvelles stratégies de développement, connaissances. Ce qui se traduit par la décomposition d'un procédé en plusieurs sous-procédés ou le changement dynamique de types de procédés.

2.2. Mécanisme d'évolution dans Prism

Prism permet de gérer l'exécution des procédés logiciels et d'assister les activités concernées par l'évolution des modèles des procédés. Ce modèle est composé de trois phases : la phase de simulation, la phase d'initialisation et la phase d'exécution. Ces phases sont enchaînées de manière similaire au modèle en spirale. Ainsi lorsqu'un problème est détecté ou une mise à jour est demandée durant l'exécution du procédé logiciel, Prism permet de retourner vers la phase de simulation. Durant cette phase, le modèle de procédé est mis à jour puis validé par des simulations avant d'être remis en exécution ; Ces changements de phases sont contrôlés par une méthodologie propre à Prism [Madhavji, 1992]. La communication des comptes rendus d'une

phase vers une autre (feedbacks) permet de faire évoluer les modèles de procédés logiciels au fur et à mesure de leur d'exécution.

2.3. Mécanisme d'évolution dans Marvel

Marvel [Kaiser and Ben-Shaul, 1993] permet l'évolution de la description du procédé logiciel par le changement des pré et des postconditions qui encapsulent les règles. Ce système assure la cohérence des règles par rapport à son mécanisme d'exécution ainsi que par rapport aux descriptions faites dans son schéma de données [Barghouti and Kaiser, 1992]. L'évolution de la description du procédé logiciel est faite de manière statique ; les procédés logiciels doivent être arrêtés pour permettre la modification de leurs descriptions. Une fois modifiée, la nouvelle description est compilée et validée. Après ces étapes, les procédés peuvent continuer leur exécution avec le nouveau contexte intégrant les évolutions.

L'approche prise par Marvel est d'accepter tout nouveau modèle de procédé logiciel, et de mettre automatiquement à jour la base d'objets pour la rendre conforme au nouveau procédé logiciel. Cette approche a été implémentée à travers un outil appelé ''Evolver'' et qui utilise le mécanisme suivant :

- Pour un modèle de procédé logiciel : ''Evolver'' procède à une comparaison de l'ancien et du nouveau modèle de procédé logiciel pour déterminer les incohérences à travers les règles affectées par la modification.

L'idée clé dans l'approche de Marvel est qu'il n'est pas nécessaire d'analyser le contenu de la base d'objets pour déterminer les incohérences qui peuvent résulter d'un changement dans le modèle de procédé logiciel. Une liste de toutes les instanciations possibles de ces règles est générée .

L'évolution du modèle de procédé logiciel porte sur les prédicats définis dans les règles (ajout, suppression des conditions), sur l'ajout de nouvelles règles et la suppression des règles existantes. Lors de la comparaison du nouveau et de l'ancien modèle de procédé logiciel, un script de commandes Marvel est généré et exécuté pour rendre la base d'objets conforme au nouveau modèle de procédé logiciel. Les paramètres des commandes du script sont déterminés en créant, pour chaque paramètre de chaque règle affectée par la modification, une liste des objets de la classe de ce paramètre. Ces commandes permettent le déclenchement des règles pour rendre cohérent la base avec le nouveau modèle de procédés logiciels.

3. Conclusion :

Plusieurs systèmes de modélisation des procédés logiciels ont été réalisés pour supporter les aspects dynamiques. L'efficacité de ces systèmes diffère sur plusieurs points suivant le formalisme de description utilisé.

L'évolution des modèles de procédés logiciels est l'un des exemples de problèmes non encore résolus dans les ateliers de génie logiciel. Il est nécessaire dans les environnements dynamiques de connaître précisément l'impact des modifications sur l'environnement, comme par exemple, l'impact d'une modification d'une politique de gestion sur le développement d'un produit ; Les solutions dégagées par les systèmes actuels pour traiter l'évolution des procédés logiciels ne sont pas encore satisfaisants.

Des formalismes d'expression simple de modèles de procédés logiciels et d'évolution aisée sur tous les aspects de modification de procédés s'avèrent nécessaires.

Partie B. Versionnement

Cette partie a pour vocation d'expliciter le concept de versions d'objets. Pour ce faire les différents concepts de base nécessaires à la définition et à la manipulation de versions d'objets sont décrits, ainsi que les différents problèmes induits par le versionnement. Ensuite quelques systèmes de gestion de version existant sont présentés.

1. Introduction

La gestion des versions d'objets est une fonctionnalité indispensable pour la plupart des applications d'ingénierie telles que la conception assistée par ordinateur, l'ingénierie logicielle, l'ingénierie de production, le multimédia, etc. Dans ces applications il est souvent utile voire nécessaire de faire coexister au sein d'une même base de données de multiples copies de description d'objets. Les bases de données doivent être capables de supporter les processus de conception qui consistent à développer différentes versions de ces objets.

Il convient de rappeler que la taille et la complexité d'un grand nombre d'applications d'ingénierie ont rendu inadéquats les systèmes de bases de données traditionnelles. Les SGBD conçus pour les applications commerciales souffrent du fait qu'ils ne prennent en compte qu'une seule vue valide du monde réel, à savoir la valeur courante des données stockées dans la base de données. Au contraire, dans les applications d'ingénierie, un utilisateur ou un groupe d'utilisateurs peuvent êtres intéressés par le suivi de l'évolution de leur modèle en essayant plusieurs voies simultanément et décider enfin que l'une des alternatives ou bien certaine fusion d'alternatives correspond à leur modèle final. C'est pourquoi plusieurs représentations valides d'un objet doivent exister dans la base de données et ce à n'importe quel moment. Les problèmes évoqués ci-dessus ont soulevés un grand intérêt dans le développement de nouveaux SGBD, en particulier les systèmes orientés objet (SOO) offrent un grand potentiel pour répondre aux requêtes précédentes. Nous nous concentrerons essentiellement sur les modèles de version dans les SOO.

La notion de version peut être interprétée de différentes façons et servir à plusieurs desseins. De manière générale, le maintien des versions d'une entité signifie le maintien de différentes représentations des mêmes aspects de l'entité, toutes ces représentations étant logiquement indépendantes les unes des autres.

En pratique, les versions ont été utilisées pour différents buts [*Dittrich et al., 1988*]

- Pour être le support du processus de conception en ingénierie [*Katz et al., 1987a*] : dans les applications de conception, les données représentent initialement quelques idées relatives à un artefact. Par suite, des objets hypothétiques sont mis ensemble pas à pas et un long processus interactif peut conduire éventuellement à une description assez complète pour servir de base à toute production.
- Pour le contrôle de la concurrence [*Papadimitriou et al., 1984*] : les versions peuvent être utilisées pour supporter, gérer et coordonner des développements parallèles, où un groupe d'utilisateurs travaillent en même temps sur une version d'un ensemble d'objets. Dans de telles situations, les utilisateurs veulent souvent disposer d'un espace de travail privé qu'ils pourront modifier sans perturber d'autres utilisateurs. Dans ce cas, un petit nombre de versions est maintenu pour un temps limité et ce dans le but d'éviter à un utilisateur de subir des changements opérés par d'autre utilisateurs.
- Pour la restauration des données [*Bayer et al., 1980*] : les versions sont utilisées pour représentées des états de bases de données consistants auxquelles le système peut retourner suite à un échec de transaction ou un effondrement dudit système.
- Pour les mises à jour des bases de données [*Dadam et al., 1984*] : dans ce cas, les mises à jour des données ne sont pas assurées par des remplacements d'anciennes valeurs mais par génération incrémentales d'une nouvelle représentation de la donnée affectée. Ceci conduit à un séquencement de versions et peut par exemple être utiliser pour vérifier des objectifs ou pour des requêtes concernant l'historique de ces données.
- Pour l'amélioration des performances dans les systèmes distribués [*Adiba et al., 1980*] : plusieurs copies de la même données peuvent exister dans différents sites.
- Enfin, pour l'enregistrement des historiques de développement de logiciels [*Katz et al., 1984*] : les versions sont utilisées pour enregistrer les évolutions de données. En développement de logiciels, il est important de garder la trace des modifications qui sont faites à un ensemble de programmes. Les modifications introduisent souvent de nouveaux bogues, et de tels cas il est très important de connaître exactement ce qui a changé.

2. Exemples de versions d'objets :

La figure IV-02 étend le diagramme de hiérarchie d'objets de la figure IV-01 en des propriétés sémantiques telles que la dérivation de version, les hiérarchies de composition et de configuration.

Par exemple, la Golf TDI-V3 est dérivée de la Golf GL-V2. Par ailleurs, pour identifier les parties composantes d'une instance d'un véhicule touristique, on doit définir la configuration logique entre les différents niveaux de la hiérarchie de composition et de la hiérarchie de version. A titre d'exemple, nous donnons les deux hiérarchies de configuration consistantes de la figure VI-02: la Golf-V1 est composée du moteur 1.6 *l*, et la Golf TDI-V3 est composée du moteur 1.9*l*.

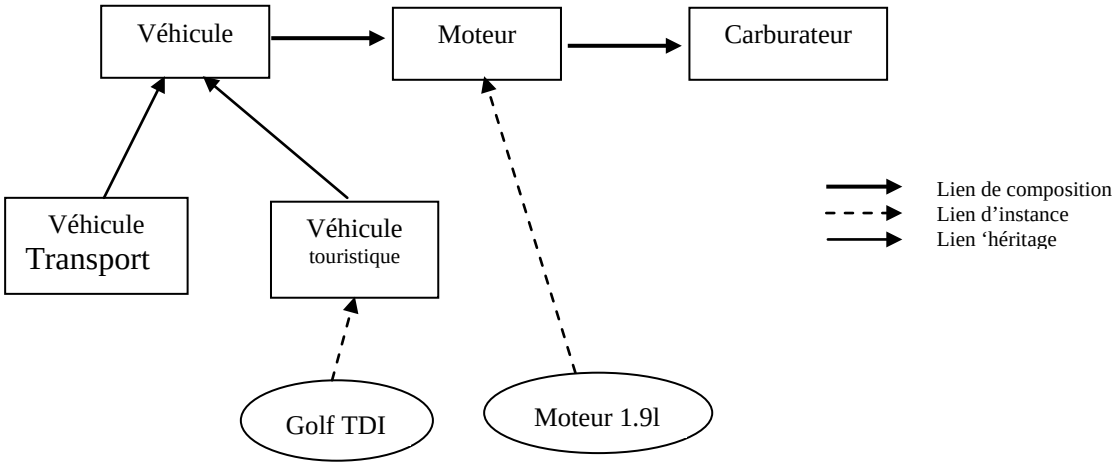


FIGURE VI-01 : *Modélisation d'un véhicule via des hiérarchies d'héritage et de composition*

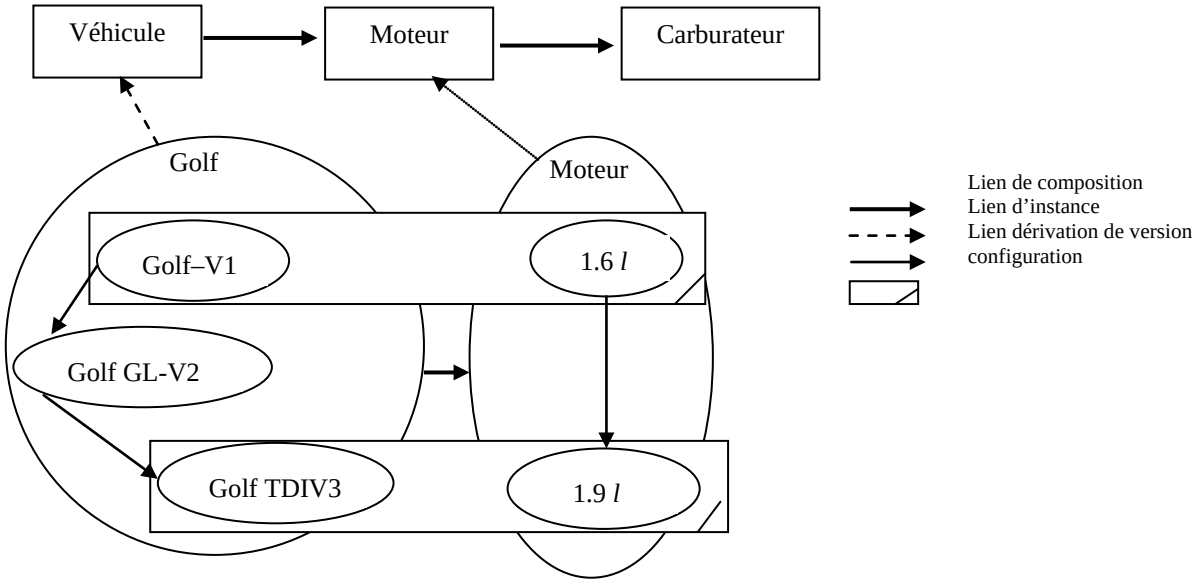


FIGURE IV- 02 : *Composition et configuration*

3. Définitions et sémantiques de base de versions d'objets.

Nous décrivons dans cette partie les principaux concepts de base nécessaires à la définition de versions d'objet [*Katz, 1990*] et [*Talens et al., 1996*]

3.1 Version d'objet

Une version d'objet peut être considérée comme une description instantanée d'un objet se trouvant dans un état particulier. L'état d'un objet est caractérisé par la valeur de ses différentes propriétés à un instant ***t*** donné.

3.1.1 Etats des versions :

Les états d'une version permettent de connaître la stabilité des données contenues dans cette dernière. En général, deux états sont associés aux versions :

- Etat permanent : une version dans un tel état possède des données stables, c'est-à-dire qu'elle ne peut être sujette à aucune modification,
- Etat temporaire : dans ce cas, la mise à jour d'une version est possible.

Toute version permanente peut être utilisée ou référencée par opposition aux versions temporaires qui sont en cours de développement et sont donc visible seulement par le créateur.
Les différents états d'une version constituent son cycle de vie. A chaque état est associé des contraintes de mise à jour, de validation et de destruction. Certaines opérations sont applicables uniquement sur certains états d'une version. En générale, l'état d'une version est un critère de stabilité ou de non-stabilité d'une version. Le cycle de vie d'une version constitue son évolution, de sa création à sa destruction, ou son archivage.

3.2 types de versions

Nous distinguons deux types de versions :

- Les versions de classe/schema
- Les versions d'instances.

3.3 relations de dérivation :

A partir d'une classe, d'une instance ou d'une version, la création d'une version est possible. Une infinité de versions peut donc être créée à partir de ces dernières.

Un ordonnancement doit être établi entre ces versions afin que tout utilisateur puisse facilement retrouver la version dont il a besoin.

L'ordonnancement des différentes versions d'une classe (ou d'une instance) est géré par la relation de dérivation. Chaque version est donc liée à sa version mère (version à partir de laquelle elle a été dérivée). Cette relation permet d'établir un ordonnancement prédécesseur/successeur entre les différentes versions.

Une sémantique est généralement associée à la relation de dérivation afin de connaître les variations entre une version successeur et une version prédécesseur. Ceci permet entre autres aux différents utilisateurs de mieux suivre l'évolution d'une classe ou d'une instance et d'éviter la redondance d'informations contenues dans les versions.

3.4 Évolution des versions :

L'évolution d'un objet peut se faire :

- Soit de façon incrémentale (donc linéaire), c'est à dire par modification successive.
- Soit en parallèle (non linéaire) : des choix sont effectués et testés simultanément. Ce type d'évolution permet de trouver une solution de manière plus facile et surtout beaucoup plus conviviale que si les différents choix avaient été faits de façon successive. De plus, chacun de ces choix peut être fait par des utilisateurs différents travaillant initialement, en parallèle, sur la même version.

Lors d'une évolution linéaire, une version a un seul prédécesseur et un seul successeur, sauf la première qui n'a pas de prédécesseur et la dernière qui n'a pas de successeur. Avec ce type d'évolution, la dérivation de versions ne peut se faire qu'à partir de la dernière version créée.

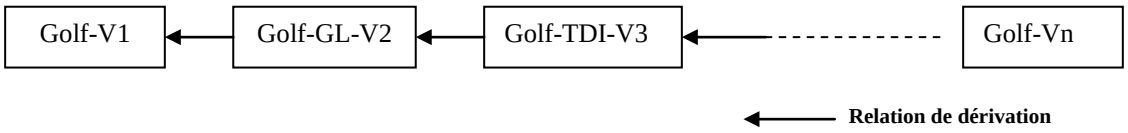


Figure IV- 3 : ***Historique linéaire***

L'évolution non linéaire peut être représentée soit sous forme :

- D'arbre : à partir d'une ou plusieurs versions, plusieurs branches de dérivation peuvent être créées. Une VERSION peut dons avoir plusieurs successeurs. Les successeurs directs de cette version sont appelés alternatives [*Katz et al., 1984*] ou versions parallèles. Chaque alternative constitue le départ d'une nouvelle branche de l'arbre de dérivation. Dans l'exemple de la figure IV-04, Golf-TD-Vj et Golf-TDI-Vk sont les successeurs de Golf-GL-Vi ; Golf-TD-Vj et Golf-TDI-Vk sont donc des alternatives de Golf-GL-Vi. La racine de l'arbre est la classe ou l'instance.

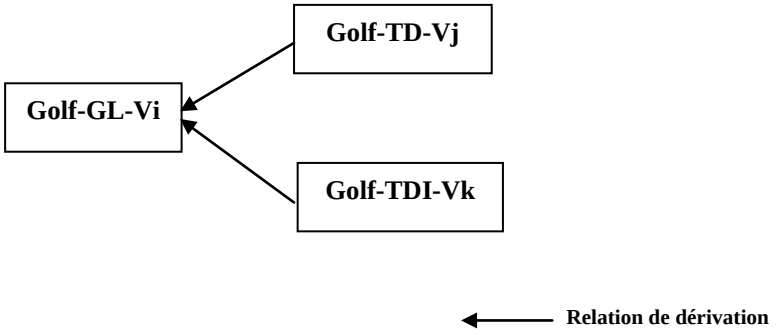


Figure IV- 04 : ***Alternative***

- De graphe : des branches de dérivations provenant d'une même version mère ou non peuvent être fusionnées [*Loomis, 1992*]. Une version peut donc être le résultat de la fusion de 2 ou n versions prédécesseurs. Par exemple, Golf-GT-Vj et Golf-TD-Vk sont les prédécesseurs de Golf-BelAir-Vi, Golf-BelAir-Vi est donc le résultat de la fusion de Golf-GT-Vj, et Golf-TD-Vk (Figure IV- 05).

Quand une version peut avoir plus d'un successeur ou plus d'un prédécesseur, la hiérarchie de dérivation forme une relation d'ordre partiel. Elle est donc représentée par un arbre ou un graphe orienté.

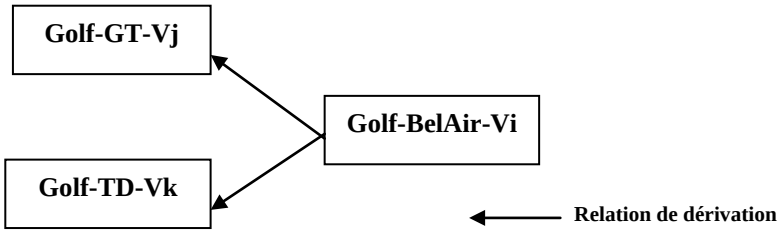


Figure IV- 05 : ***Fusion***

3.5. Gestion d'un ensemble de version :

Les versions d'une classe ou d'une instance sont donc hiérarchisées et reliées entre elles par la relation de dérivation. Ces différentes versions forment l'ensemble des versions dérivées directement ou indirectement à partir d'une classe ou d'une instance (Figure 06). La question à poser est : peut-on créer des versions à partir d'une classe ou d'une instance quelconque ?

une classe ayant des versions de classe ou d'instance est appelée une classe versionnable. Donc, seules les classes versionnables ont des versions. Par classe versionnable, nous entendons une classe dérivée à partir d'une classe système spécifique. Soit c'est le concepteur qui décide lors de la conception de la classe si elle est versionnable et le choix est irrévocable. Soit toute classe hérite des propriétés de versionnabilité et tout utilisateur peut par la suite créer ou non des versions de classes et/ou d'instances.

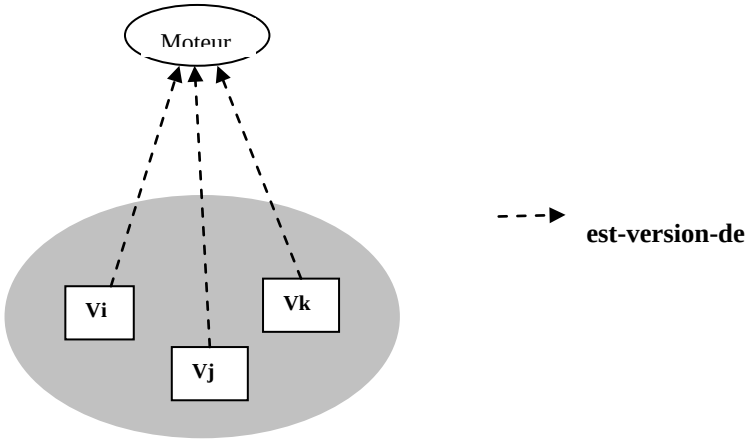


Figure IV- 06 : ***Ensemble de versions d'une classe versionnable***

La création des versions est faite ensuite :

- Soit par l'utilisateur : il demande la dérivation de versions à partir d'une classe versionnable, d'une instance versionnable ou d'une version de classe ou d'instance,
- Soit par le système: l'ordre de dérivation vient du système, cet ordre est généralement la conséquence de manipulations faites par les utilisateurs :
  - tentative de modification de version dont l'état est permanent,
  - mise à jour des attributs sensibles (attributs immuables [*Estublier et al., 1995*])

Cet ensemble de versions est généralement gérée par la classe versionnable elle-même ou par une classe système spécifique. Ces dernières permettent la gestion de la hiérarchie de dérivation, mais également la gestion de la version courante [*Kim, 1990*]. La version courante est la version qui sera proposée à la demande de toute utilisateur. Elle est désignée soit par le système (dernière version validée donc stable (de type permanent)), soit par le concepteur (une des versions permanentes).

4. Opérations de base

Un certain nombre d'opérations permettent de créer, modifier, manipuler et détruire des versions.
Tout d'abord avant de pouvoir créer une version à partir d'un objet, il faut que ce dernier soit versionnable. Selon le système cette opération existe ou n'existe pas.
Ensuite, tout utilisateur peut dériver une version à partir d'une autre en utilisant l'opération de dérivation.
Dans certains systèmes, une version peut être dérivée à partir de plusieurs versions du même objet, très souvent une opération spécifique permet la réalisation de cette fusion.
La modification d'une version ne peut s'effectuer qu'en fonction de l'état de la version. Seules les versions ayant un état non stable sont modifiables.
Les versions stables sont utilisables par tout utilisateur et référençables par tout autre objet ou version. Différentes fonctionnalités permettent de sélectionner une version parmi l'ensemble des versions d'un objet :

- Opérations de recherches de la versions courante (dernière version créée et validée ou version désignée comme telle),
- Opérations de comparaison de versions qui permet à l'utilisateurs de connaître les différences entre les versions et donc qui permet de l'aider à en choisir une,
- Opérations permettant d'exprimer des clauses de sélection.

La destruction de versions est aussi différente selon les systèmes et n'a donc pas toujours les mêmes conséquences. En fait, la question qui revient souvent chez les concepteurs de systèmes concerne l'opération de destruction : Que fait-on des objets référencés ou référençant (relation de dérivation ou de composition ou autres) la version à détruire ? ce problème est lié à celui de l'intégrité référentielle dans les bases de données. Il se répercute au niveau des versions. Plusieurs réponses ont été avancées :

- Suppression interdite.
- Suppression logique et non physique (les utilisateurs ne la voient pas mais elle est toujours présente afin de conserver la cohérence de la hiérarchie de dérivation). Les objets ou versions

la référençant par d'autres relations que celle de dérivation sont mis à jour ou détruits en fonction des systèmes de la sémantique des relations (pour garder la cohérence du référencement entre objets).

- Suppression physique entraînant la mise à jour de la hiérarchie de dérivation de cette version

5. Quelques systèmes de gestions de versions d'objets

Dans cette partie, nous présentons quelques systèmes de gestion de versions d'objets existants. Pour chacun d'eux, nous montrons comment sont gérés et utilisés les différents concepts de versions vus précédemment.

5.1. Orion [Chou et al.,1988], [Kim, 1990]

Ce système gère des versions de schémas. L'évolution et la gestion de ces versions sont prises en compte par un ensemble de règles. Des versions d'instances sont créées à partir de versions de schémas. Le contrôle d'accès des versions de schéma sur les versions d'instances ainsi que les opérations applicables sur ces dernières sont également gérées par des règles spécifiques.
Un objet versionné est une instance d'une classe versionnable. Des versions d'instances peuvent être associées seulement à une instance de classe versionnable. Cette instance versionnable est nommée « objet générique ». Lors de la création d'une telle instance, le système crée l'objet générique qui gère la hiérarchie de dérivation des versions et la première version d'instance. La hiérarchie de dérivation forme un arbre, par conséquent la fusion est interdite.
Les versions ont des états qui leurs sont associés. Ces états sont : transitoire, de travail et stabilisé. Des règles permettent la promotion d'une version d'un état dans un autre et inversement le rétablissement dans l'état précédent. La notion d'espace de travail public et privé est intégrée au concept d'état. Les versions dans un état stable ne sont ni modifiables, ni destructibles. Elles sont par conséquent dans une base de données publique, par opposition aux versions dans une base de données privée qui sont :

- Soit non modifiables mais destructibles (de travail),
- Soit modifiables et destructibles (transitoire)

Orion est un système de gestion de bases de données orientées objets (SGBDOO) qui prend en compte les application de CAO, d'intelligence artificielle et de systèmes d'informations ouverts (OIS). Il existe en deux versions : une implémentée en Lisp et l'autre en C.

5.2 Encore [Zdonik , 1986], [Skarra, 1987]

Dans Encore, un objet spécifique est également créé pour gérer l'ensemble des versions d'un objet. Par exemple, quand une opération de création d'un objet X de type T est invoquée, l'instance de type T est créée mais il est aussi créé une instance V de type « Version Set » pour contenir X et ces futures versions.

L'ordonnancement peut être linéaire ou non linéaire sans aucune restriction. «Version Set » accumule toutes les versions d'un objet qui évolue à travers le temps, c'est la date d'une version qui gère la stabilité ou l'instabilité d'une version.

Encore permet également la maintenance de plusieurs versions de type, de ce fait les « anciens » objets et leurs « anciennes » définitions de types coexistent avec les « nouveaux » objets et leurs « nouvelles » définitions. Afin de pouvoir utiliser les anciennes versions d'instances avec les nouvelles versions de type, deux solution sont possibles :

- conversion,
- émulation.

Chapitre V :

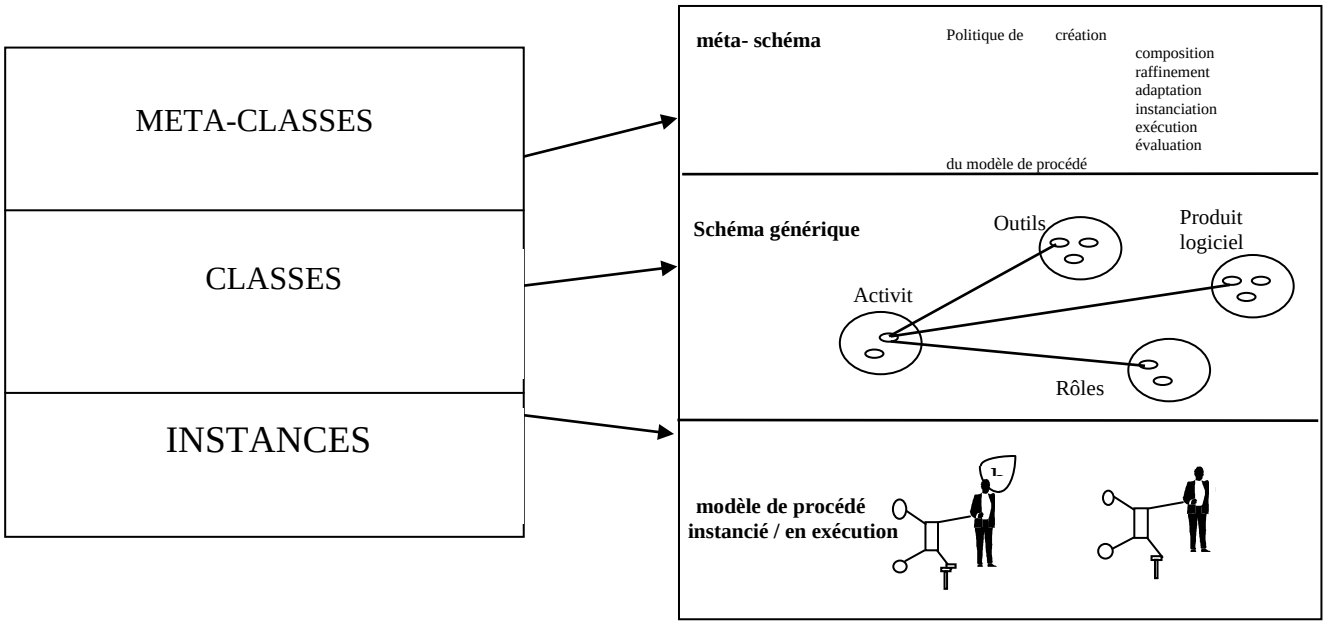
Notre approche d'évolution des modèles de procédés logiciels

Dans ce chapitre nous décrivons notre approche d'évolution des modèles de procédés logiciels. Deux approches de modifications (d'évolutions) sont considérées. (1) la modification du modèle de procédé logiciel et (2) la modification de ses instances (activités, produits,…, relations) pendant leurs exécution. Nous présentons la nature et la sémantique des différentes opérations de modification. Ces opérations donnent à l'utilisateur la possibilité:
(1). d'améliorer le modèle de procédé en modifiant des types d'activités, des types produit, … ou en ajoutant de nouveaux types.
(2) d'améliorer le procédé logiciel et de définir de nouveaux procédés logiciels en utilisant la création dynamique des différent entités (activité, produit, flux de contrôle,…).

1. Concepts de modélisation :

L'approche objet pour les bases de donnés constitue une technologie prometteuse pour le développement des applications avancées [Kim90]. Une des caractéristiques fondamentales de ces applications est la nature dynamique des données modélisées et de leurs schémas. Le modèle objet se prête mieux à la conception incrémentale [Booch 94].

Le méta modèle que nous retenons pour notre étude est le suivant :



- Le méta-schéma est un ensemble prédéfini de méta-classes (voir fig. V-02). Ces méta-classes décrivent des propriétés qui caractérisent toutes les activités, les relations, tous les objets et tous les attributs.

- Le schéma est un ensemble de classes qui sont des instances des méta-classes. Ces classes décrivant la structure et le comportement des objets (instances)

Nous définissons le méta-schéma suivant :

Le méta-schéma :

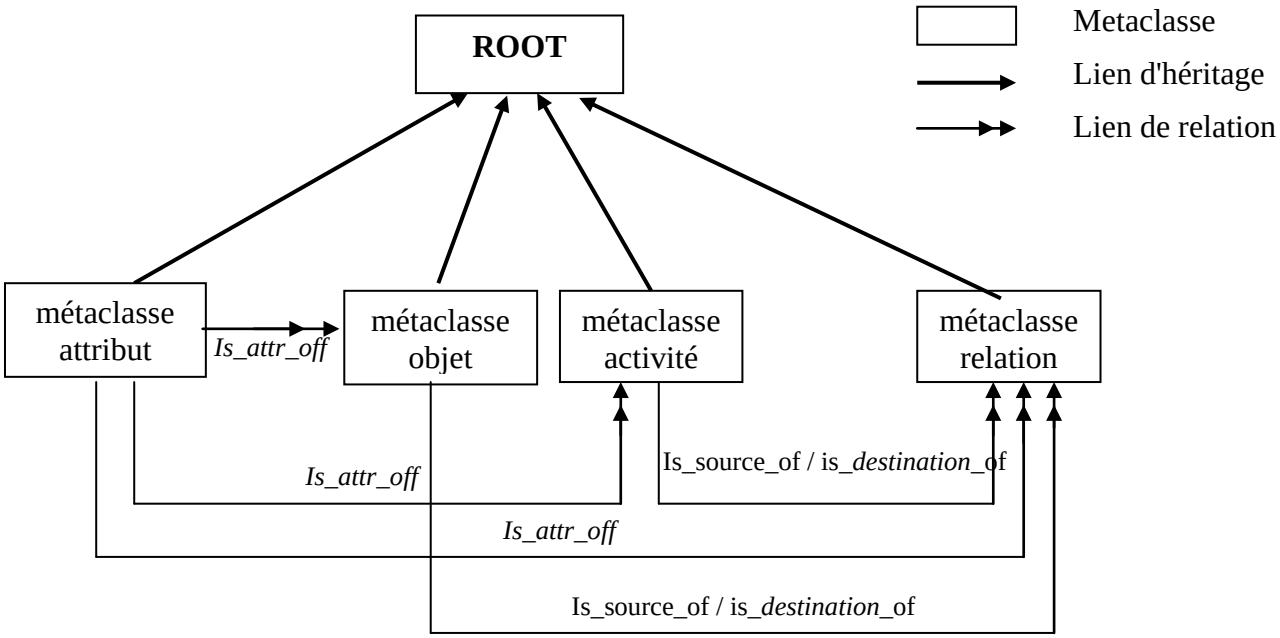


Figure V.2 : Le Méta-schéma

Les métaclasses:

Cinq métaclasses sont prédéfinies dans le métaschéma:

1. la métaclasse *ROOT* représente la racine du métaschéma. Elle définit les propriétés communes à toutes les autres métaclasses et donc communes à toutes les classes qui seront définies dans la métabase.

2. La métaclasse *metaclasse_Objet* définit les caractéristiques des objets de la métabase qui correspondent à des définitions de classes objets. Les classes objet décrivent des types produits, des types d'agents et des types d'outils.

3. La métaclasse *metaclasse_Activité* définit les caractéristiques des objets de la base qui correspondent à des définitions de classes d'activités.

4. La métaclasse *metaclasse_Relation* définit les caractéristiques des objets de la métabase qui correspondent à des définitions de classes de relations. Les classes de relations décrivent les différentes connexions de flux de contrôle et de flux de données.

5. La métaclasse *metaclasse_Attribut* définit les caractéristiques des objets de la métabase qui correspondent à des définition de classes d'attributs.

Les types de relations prédéfinies:

Nous définissons les types de relations suivantes:

1. la classe *is_source_of* définit la relation entre une classe d'objet T et une classe de relation R pour exprimer le fait que tout objet de la classe T peut être un objet source de toute relation de la classe R.

2. la classe *is_destination_of* définit la relation entre une classe d'objet T et une classe de relation R pour exprimer le fait que tout objet de la classe T peut être un objet de destination de toute relation de la classe R.

3. la classe *is_attr_of* définit la relation entre une classe d'attribut et une classe d'objet ou d'activité ou de relation pour lequel il s'applique.

La métabase :

les objets de la métabase représentent alors les différentes définitions de classes d'objets, de classes relations, de classes d'attributs et de classes d'activités. elles sont obtenues par instantiation des métaclasses définies dans le métaschéma.

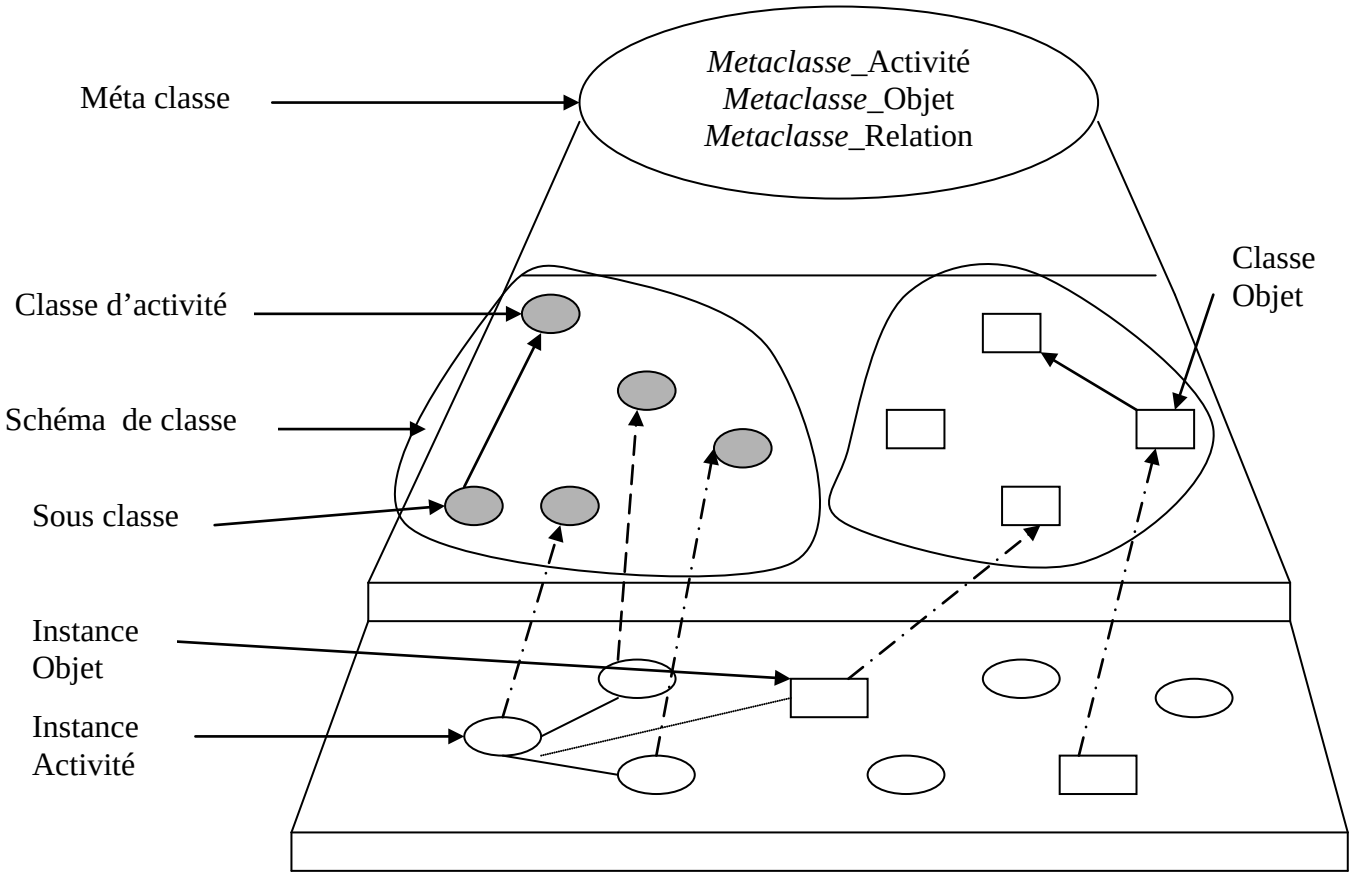


Figure V.3 : Exemple de Méta-base

Les instances :

Les différentes instances manipulées dans notre modèle sont des instances des classes activités, classes objets et classe relations et qui sont respectivement les différents types d'activités, produits, agents, outils, relation de composition entre activités, flux de données et flux de contrôle.

- Le **flux de contrôle** représente le séquencement et la concurrence des activités dans l'environnement.
- Le **flux de données** illustre la transition du produit entre les différentes activités. Ce produit étant la sortie d'une activité et l'entrée d'une autre activité.

La nature et la sémantique des opérations de mise à jour (opérations et sémantique d'évolution):

Notre modèle permet deux niveaux d'évolution des modèles de processus :

L'évolution du modèle de processus de logiciel consiste à modifier les deux niveaux (niveau instances et schéma : modèle de processus) pendant l'exécution des activités.

1. **Les opérations de mise à jour des instances : l'évolution des instances**

   Ces opérations permettent la modifications des instances de processus (activité, produit, agent, outils et relations (flux de contrôle,…..)) pendant leur exécution. Cette approche de modification directe possède l'avantage de contrôler l'exécution du processus. Elle permet:

   - De modifier l'exécution par :
     - l'ajout, la suppression,…, et la substitution des activités.
     - La modification du flux de données.
     - La modification du flux de contrôle (ajouter un état et sauter un autre).

   - De définir des processus par la création dynamique des différentes entités (activités, produits,……).

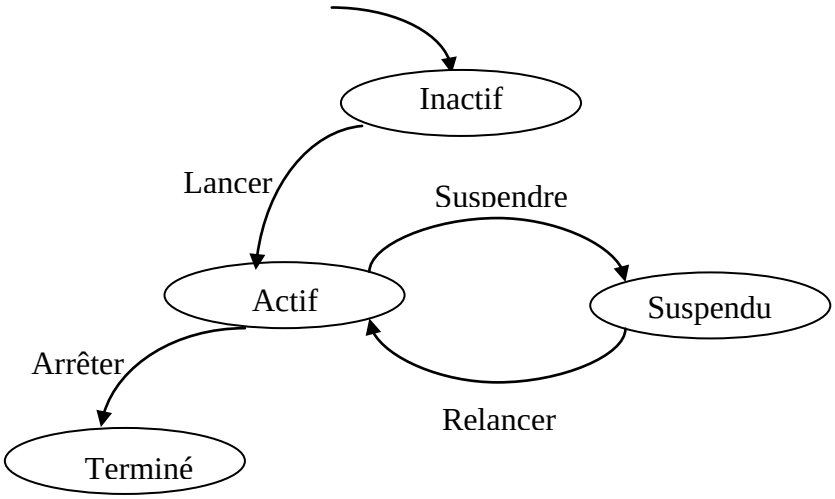
La taxinomie des différentes opérations est la suivante:

1. Lancer, suspendre, relancer et arrêter une activité.
2. créer, supprimer, renommer une activité.
3. Insérer, déplacer et substituer une activité.
4. Créer, supprimer, renommer, affecter, désaffecter, substituer, transférer, partager et copier un produit.
5. Création, suppression et renommage d'un flux de contrôle.
6. Création, suppression et renommage d'un flux de données.
7. Ajouter, supprimer, substituer, affecter et désaffecter un agent.
8. Ajouter, supprimer, substituer, affecter et désaffecter un outil.

Nous présentons dans ce qui suit ces différentes opérations d'évolution en précisant pour chacune d'elle:

- les conditions d'applications
- les sémantiques associées.

1.1. L'évolution des instances activités :



Le diagramme d'état par défaut d'une activité

1. **Créer une activité :**

Entrées: Nom_ Activité, Nom_Classe, Activité_Mère, Nom_Agent, Rôle ;

Sortie: instance d'activité.

Condition :

- L'instance d'une activité (Nom_Activité) est toujours ajouter par rapport à son activité mère (Activité_Mère), elle appartient à l'espèce de travail de son activité mère.
- Nom_ Activité $\notin$ l'ensemble des instances de la classe Nom_Classe.
- Nom_classe $\in$ au schéma.

Sémantique :

- Cette opération permet d'ajouter une nouvelle instance de la classe Nom_Classe à la base des instances. Cette instance est ajoutée dans l'état inactif, elle peut être exécuter indirectement à travers un flux de contrôle ou directement (commande lancer Nom_Activité)
- Nom_Agent est ajouter pour l'activité (si $\notin$ la base des agents). Un rôle est lui assigné pour cette activité.

2. Supprimer une activité :

Entrée: Nom_ Activité, Acitivté_Mère;

Condition :

- Nom_ Activité ∈ à la base des instances activités.
- Activité_Mère ∈ à la base des instances activités.

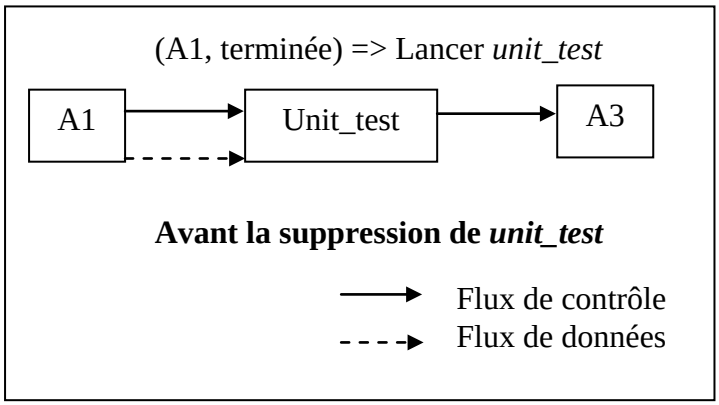
Sémantique :

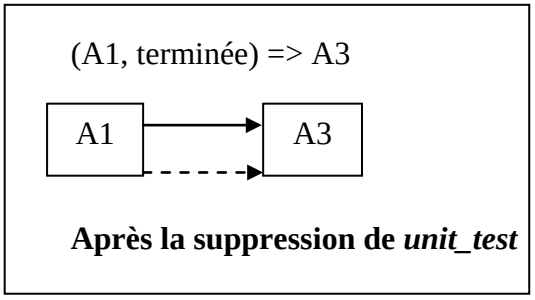
- cette opération permet à l'utilisateur de supprimer une Activité (Nom_ Activité),
- la suppression d'une activité en exécution est interdite,
- la suppression d'une activité implique la suppression de tous ses sous-activités et les différents flux de contrôle et de données.
- la suppression d'une activité affecte toutes les activités liées à celle-ci.

Exemple:

Supprimer-Activiter (unit_test, global_test);

Les activités qui sont reliées avec l'activité unit_test par des flux de contrôle sont automatiquement connectées entre eux par un nouveau flux de contrôle. (C'est la même chose pour le flux de données)





3. Suspendre l'exécution d'une activité :

Entrée: Nom_ Activité, ActivitéMère;

Condition :

- Nom_Activité doit appartenir à la base des instances activités.
- ActivitéMère ∈ à la base des activités
- Etat(instance) = exécution,

Sémantique :

- cette opération permet de suspendre l'exécution d'une instance,
- la suspension d'une instance activité affecte toutes les instances liées à celle-ci par des flux de contrôle ou flux de données.

4. Lancer une activité :

Entrée: Nom_ Activité, ActivitéMère ;

Condition :

- Nom_ Activité doit appartenir à la base des instances activités.
- ActivitéMère ∈ à la base des activités
- Etat(Activité) = Arrêt ou Suspendue,

Sémantique :

- cette opération permet de démarrer l'exécution d'une instance activité,

5. Relancer une activité :

Entrée: Nom_ Activité, ActivitéMère ;

Condition :

- Nom_ Activité doit appartenir à la base des instances activités.
- ActivitéMère ∈ à la base des activités
- Etat(Activité) = Suspendu ;

Sémantique :

- cette opération permet de relancer l'exécution d'une instance activité suspendue.

6. Arrêter une Activité :

Entrée: Nom_ Activité, ActivitéMère ;

Condition :

- Nom_ Activité doit appartenir à la base des instances activités.
- ActivitéMère ∈ à la base des activités
- Etat(Activité) = Exécution ;

Sémantique :

- cette opération permet de terminer l'exécution d'une instance activité.

7. Insérer une Activité :

Inserer_Activité (A1, A, A2);

Cette opération permet à l'utilisateur d'ajouter une activité A entre les activités A1 et A2. L'activité est ajoutée dans le but de réaliser de nouvelles taches entre les activités existantes (nouveaux besoins, changement de stratégie,…...etc.) donc il est très utile d'avoir la possibilité d'insérer directement une activité entre des activités existantes.
Deux relations de flux de contrôle sont automatiquement ajoutées entre la nouvelle activité et les activités adjacentes. Des relations de flux de données sont dynamiquement ajoutées par l'utilisateur selon la tache voulue.

8. **Substituer une activité:**

Substituer_Activité (Nom_Activité, ActivitéMère, NomActivité_1);

Cette opération permet à l'utilisateur de remplacer l'activité Nom_Activité par l'activité Nom_Activité_1. Les relations de flux de contrôle et flux de données sont conservées après la substitution.

9. Déplacement d'une activité:

Dépalcer_Activité (A1, A, A2);

Cette opération permet à l'utilisateur de déplacer l'activité A entre les activités A1 et A2. les relations de flux de contrôle et flux de données qui relie l'activité A aux activités précédentes sont supprimées, l'activité A est insérer ensuite entre A1 et A2.

1.2. L'évolution des instances Outils, Agents et **Produits:**

Notre modèle d'évolution s'applique également aux outils, agents et produits.

Outils :

1. Affecter un Outil à une activité :

Affecter_Outil(Nom_Outil, Nom_Activite) ;

Condition :

- Nom_Outil existe dans la base des outils.
- Nom_Activite existe dans la base des activités.

Sémantique :

- cette opération permet d'affecter un outil à une activité.
- Un même outil peut être affecter à plusieurs activités simultanément.

2. Désaffecter un outil :

Désaffecter_Outil(Nom_Outil, Nom_Activite) ;

Condition :

- Nom_Outil doit être déjà affecté à Nom_Activite,
- Nom_Activité doit exister dans la base des activités.

Sémantique :

- Cette opération permet de supprimer un outil associé à une activité, cependant l'outil existe toujours dans la base des outils.
- La désaffectation d'un outil en exécution dans une activité n'est pas autorisée.

3. Substituer un outil :

Substituer_Outil (Nom_Outil, Nouveau_Nom_Outil, Nom_Activite) ;

Condition :

- Nom_Outil doit être déjà affecté à Nom_Activite,
- Nouveau_Nom_Outil doit exister dans la base des outils,
- Nom_Activite doit exister dans la base des activités.

Sémantique :

- cette opération permet de substituer un outil " Nom_Outil" par un autre outil "Nouveau_Nom_Outil" à une activité.
- La substitution d'un outil en cours d'exécution dans une activité n'est pas autorisée.

4. Ajouter un Outil à la base des outils :

Créer_Outil(Nom_Outil, Nom_Classe) ;

Condition :

- Nom_Outil est unique dans la base des outils.
- Nom_Classe $\in$ schema

Sémantique :

- cette opération permet d'ajouter un outil à la base des outils.

5. Supprimer un Outil de la base :

Supprimer_Outil (Nom_Outil) ;

Condition :

- Nom_Outil doit exister dans la base des outils.

Sémantique :

- cette opération permet de supprimer un outil de la base.
- la suppression de la base d'un outil affecté à une instance activité n'est pas autorisée.

Agents :

1. Affecter un Agent à une activité :

Affecter_Agent (Nom_Agent, Nom_Activite, rôle) ;

Condition :

- Nom_Agent doit exister dans la base des agents.
- Nom_Activite doit exister dans la base des activités.

Sémantique :

- Cette opération permet d'affecter un agent à une activité en lui assignant un rôle.
- Un agent peut es voir affecter plusieurs rôles dans une même activité ou dans des activités différentes.

2. Désaffecter un agent :

Désaffecter_Agent(Nom_Agent, Nom_Activite) ;

Condition :

- Nom_Agent doit être déjà affecté à l'activité Nom_Activite,
- Nom_Activité doit exister dans la base des activités.

Sémantique :

- cette opération permet de supprimer un agent affecté à une instance activité, l'agent existe toujours dans la base des agents.
- La désaffectation d'un agent est interdite si état(instance_activité)=exécution ;

3. Substituer un Agent :

Substituer_Agent (Nom_Agent, Nouveau_Nom_Agent, Nom_Activite) ;

Condition :

- Nom_Agent doit être déjà affecté à une activité,
- Nouveau_Nom_ Agent doit exister dans la base des agents,
- Nom_Activite doit exister dans la base des activités.

Sémantique :

- cette opération permet de substituer un agent "Nom_Agent" par un autre agent "Nouveau_Nom_Agent" pour une activité.

4. Ajouter un Agent :

Ajouter_Agent(Nom_Agent, Nom_Classe) ;

Condition :

- Nom_Classe ∈ au schéma
- Nom_Agent ∉ base des agents.

Sémantique :

- Elle permet d'ajouter un agent à la base des agents.

5. Supprimer un Agent :

Supprimer_Agent(Nom_Agent) ;

Condition :

- Nom_Agent existe dans la base des agents.

Sémantique :

- Cette opération permet de supprimer un agent de la base des agents.
- La suppression d'un agent affecté à une activité en cours d'exécution n'est pas autorisée.

Produit :

1. **Affecter un produit à une activité :**

Affecter_Produit(Nom_Produit, Nom_Activité) ;

Condition :

- Nom_Produit doit exister dans la base des produits,
- Nom_Activité doit exister dans la base des activités.

Sémantique :

- cette opération permet d'affecter un produit à une instance activité, ce produit est nécessaire pour l'exécution de l'activité Activité.

2. **Désaffecter un produit :**

Désaffecter_Produit (Nom_Produit, Nom_Activite) ;

Condition :

- Nom_Produit doit être déjà affecté à l'activité Nom_Activite,
- Nom_Activité doit exister dans la base des activités.

Sémantique :

- cette opération permet de supprimer un Produit associé à une activité, Cependant le produit existe toujours dans la base des produits.
- La désaffectation est interdite si le produit est nécessaire pour l'exécution de l'activité.

3. **Substituer une instance Produit :**

Substituer_Produit (Nom_Produit, Nouveau_Nom_Produit, Nom_Activite) ;

Condition :

- Nom_Produit doit être déjà affecté à l'activité Nom_Activité,
- Nouveau_Nom_ Produit doit exister dans la base des Produits,
- Nom_Activite doit exister dans la base des activités.

Sémantique :

- cette opération permet de substituer un Produit " Nom_Produit " par un autre Produit "Nouveau_Nom_ Produit " dans une activité.

4. **Créer une Instance produit :**

Créer_Produit (Nom_Produit, Nom_Classe) ;

Condition :

- Nom_Classe $\in$ au schéma,
- Nom_Produit est unique dans la base des produits.

Sémantique :

- cette opération permet d'ajouter un produit dans la base des produits. Le nom doit être unique.

5. **Supprimer une Instance produit :**

Supprimer_Produit(Nom_Produit) ;

Condition :

- Nom_Produit doit exister dans la base des produits.

Sémantique :

- cette opération permet de supprimer un produit de la base des produits.
- La suppression d'un produit affecté à une activité en exécution n'est pas autorisée. (produit nécessaire pour l'exécution de l'activité)

6. **le transfert, le partage et la copie d'un produit:**

Tous les produits peuvent être transférer dynamiquement, copier ou partager entre plusieurs activités.

a). Share (Outprod, Activité_1, Inprod, Activité_2, Mode);

Cette opération permet à l'utilisateur de partager le produit Outprod qui appartient à l'espace de travail de l'activité Activité_1 avec l'espace de travail de l'activité Activité_2. Le mode d'accès au produit dans l'activité Activité_2 est donné par le paramètre Mode.

b). Transfert (Outprod, Activité_1,Inprod, Activité_2);

Cette opération transfère le produit Outprod de l'activité Activité_1 vers Activité_2. Le nouveau nom local est Inprod. Le transfert est effectif : le produit appartient seulement à l'espace de travail de Activité_2.

c). Copy (Outprod, Activité_1, Inprod, Activité_2, Mode);

Cette opération crée une copie du produit Outprod qui appartient à l'espace de travail d'Activité_1 pour l'inclure dans l'espace de travail de l'activité Activité_2. Le paramètre Mode indique le mode d'accès à cette copie dans l'activité_2.

Flux de données et flux de contrôle:

1. **Création d'un flux de données:**

Nouveau_FluxDonnées (DF_type, Activité_1, Outprod, Activité_2, Inprod);

Cette procédure permet de créer une relation flux de données entre le produit Outprod de l'Activité_1 et l'Activité_2. Cette opération exprime un transfert, une copie ou un partage d'un produit selon le paramètre DF_type. Le nouveau nom local du produit dans l'activité Activité_2 est désigné par Inprod.

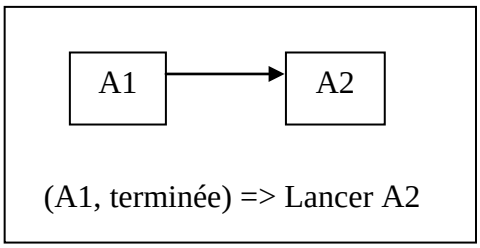
2. **Création d'un flux de contrôle:**

Nouveau_FluxControle (Activité_1, Activité_2);

Cette opération permet de créer une relation de flux de contrôle entre une activité Activité_1 (comme origine) et une activité Activité_2 (comme destination). Cette relation exprime la séquence d'exécution des activités Activité_1 et Activité_2. La condition de cette exécution dépend de la nature de ces activités:

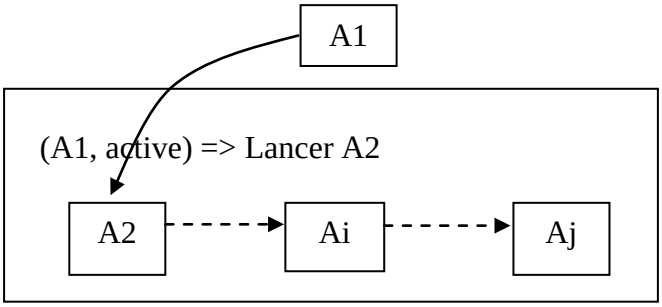
1. si aucune activité n'est l'activité mère de l'autre (même niveau), la sémantique du flux de contrôle est:

   " Commencer l'exécution de l'Activité A2 dés que l'Activité A1 se termine"



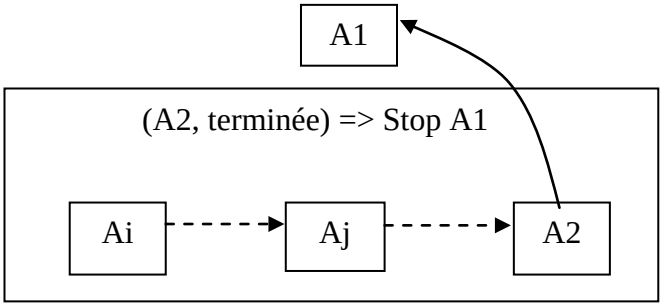
2. si l'activité A1 est l'activité mère de l'activité A2, et l'activité A2 est l'activité initiale des sous activités de l'activité A1 alors:

   " Dés que A1 devient active (début de l'exécution), alors commencer l'exécution de sa première sous activité l'activité A2 "



3. Si A1 est l'activité mère de l'activité A2, et A2 est l'activité finale des sous activités de A1 alors:

   " Dés que la sous activité finale A2 de l'activité mère A1 termine son exécution, alors stoper (termine) l'exécution de A1"



3. Suppression d'un flux de données ou 'un flux de contrôle:

```
Supprimer ( Type_Flux, Activité_1, Activité_2);
```

Cette fonction permet de supprimer le flux de contrôle ou le flux de données entre Activité_1 et Activité_2.

L'évolution du schéma (les opérations de modification du schéma) :

Les opérations d'évolution du schéma permettent d'apporter toute forme de modification sur le schéma. Les différentes opérations de modification sont :

1. *Les opérations de modification des classes :*
   - Elles portent sur les opérations de modification du graphe d'héritage

2. *Les opérations de modification dans la définition des classes :*
   - Ce sont des opérations de mise à jours des propriétés et des méthodes des classes

1. Les opérations de modification des classes :

Les opérations sont :
- créer, supprimer ou renommer une classe dans un schéma.

1.1- Ajouter une classe dans un schéma :

Créer_classe (Nom_classe, super_classe) ;

Conditions :

Nom-classe $\notin$ au schéma
Super_classe $\in$ au schéma

Sémantique :

- La classe Nom_Classe est ajoutée comme classe feuille dans le schéma.
- La définition de la nouvelle classe n'affecte pas la définition des classes existantes.
- La création d'une nouvelle classe n'affecte pas les instances en exécution.

1.2 **Supprimer une classe dans un schéma :**

Supprimer_Classe(Nom_Classe) ;

Condition :

Nom-classe $\in$ schéma

Sémantique :

- La suppression d'une classe dans un schéma n'est autorisé que si elle est sans instance.
- Le graphe d'héritage doit rester connexe après la suppression. Si le graphe d'héritage n'est plus connexe après la suppression, alors connecter les sous classes de la classe supprimée à sa super classe immédiate.

1.3 **. Renommer une classe du schéma :**

Renommer_Classe (Nom_Classe, Nouveau_Nom_Classe)

Conditions :

- Nom_Classe doit $\in$ au schéma.
- Nouveau_Nom_Classe $\notin$ au schéma (Nouveau_Nom_Classe doit être unique).

Sémantique :

- La classe Nom_Classe est renommée dans le schéma par Nouveau_Nom_Classe.
- Les propriétés après le renommage de Nom_Classe = Propriétés avant le renommage de Nom_Classe.
- Les sous classes de la classe Nouveau_Nom_Classe= Les sous classes de Nom_Classe
- Les instances de la classe nom_Classe doivent être convertir à la nouvelle définition.

2. Les opérations de modification dans la définition des classes :

Elles sont appliquées sur les propriétés et les méthodes définies dans les classes.
Les opérations sont :

- Rajout, suppression et renommage des attributs.
- Modification du domaine des attributs.
- Ajout, suppression des méthodes.

2.1**. Ajouter un attribut :**

Ajouter_attribut (Nom_Classe, Nom_Attribut) ;

Condition :

- Nom_Classe ∈ au schéma.
- Nom_Attribut ∉ à l'ensemble des attributs de la classe Nom_Classe.

Sémantique :

- l'attribut Nom_Attribut est ajouté à la classe nom_Classe.
- Les instances de la classe Nom_Classe doivent être converties à la nouvelle définition de la classe.

2.2 Supprimer un attribut :

Supprimer_attribut (Nom_Classe, Nom_Attribut) ;

Condition :

- Nom_Classe ∈ au schéma.
- Nom_Attribut ∈ à l'ensemble des attributs de la classe Nom_Classe.

Sémantique :

- La suppression d'un attribut n'est pas autorisée si une instance est en train d'évaluer la valeur de cet attribut.
- La suppression d'un attribut est propagé aux sous types qui héritent cet attribut.
- Cette opération entraîne le masquage de cet attribut pour toutes les instances de la classe Nom_Classe c.a.d la conversion des instances à la nouvelle définition.

2.3 **Renommer un attribut :**

Renommer_Attribut (Nom_Classe, Nom_Arttribut, Nouveau_Nom_Attribut) ;

Condition :

- Nom_Classe ∈ au schéma.
- Nom_Attribut ∈ à l'ensemble des attributs de la classe Nom_Classe.
- Nouveau_Nom_Attribut ∉ à l'ensemble des attributs de la classe Nom_Classe.

Sémantique :

- Le renommage est interdit si il y a une instance qui est en train d'évaluer la valeur de cet attribut.
- Cette opération entraîne la conversion des instances à la nouvelle définition.

2.4. Modifier le domaine (type) d'un attribut :

Modifier_Domaine_Attribut (Nom_Classe, Nom_Attribut,
Type_Attribut, Nouveau_Type_Attribut) ;

Nouveau_Type_Attribut ∈ {string / integer / float /date / boolean }

Condition :

- Nom_Classe ∈ au schéma.
- Nom_Attribut ∈ à l'ensemble des attributs de la classe Nom_Classe.
- Nouveau_Type_Attribut ∈ {string / integer / float / date / boolean }

Sémantique :

- Le changement du domaine d'un attribut est propagé aux sous classes
- Type_Attribut ⊆ Nouveau_Type_Attribut. Une généralisation du type est autorisée, sa spécialisation ne l'est pas.
- Cette opération entraîne la conversion des instances à la nouvelle définition.

2.5. Ajouter une méthode :

Ajouter_methode (Nom_Classe, Nom_Methode, Signature_Methode) ;

Condition :

- Nom_Classe ∈ au schéma.
- Nom_Methode ∉ à l'ensemble des méthodes de la classe Nom_Classe.

2.6. Supprimer une méthode :

Supprimer_methode (Nom_Classe, Nom_Methode) ;

Condition :

- Nom_Classe ∈ au schéma.
- Nom_Methode ∈ à l'ensemble des methodes de la classe Nom_Classe.

Sémantique :

- La suppression d'une méthode est interdite si il y a une instance qui est en train d'exécuter cette méthode.

- La suppression d'une méthode est propager au sous classes qui héritent cette méthode.

Conclusion :

Nous avons présenté dans ce chapitre une proposition pour la résolution du problème d'évolution dans les modèles de procédés logiciels.
L'évolution des modèles de procédés logiciels est l'acte de modifier le modèle pendant l'exécution des activités et de contrôler cette modification. L'approche d'évolution proposée permet l'évolution selon deux niveaux : niveau modèle et niveau instances. L'évolution au niveau modèle permet de gérer toutes les modifications inhérentes à la conception du procédé logiciel. L'évolution au niveau des instances tente de contrôler l'exécution du procédé par un jeu de simulation/modification.

Nous avons décrit pour chaque opération de modification une syntaxe et une sémantique appropriée.

Chapitre V :

CONCLUSION ET PERSPECTIVES :

Notre travail de recherche a eu pour but principal le développement de concepts de modélisation et d'évolution représentant des caractéristiques fondamentales tant au niveau du concept propre de procédés logiciels qu'au niveau de leurs supports bases de données. L'objectif étant de tendre vers un modèle général de description des modèles de procédés logiciels ainsi que leurs évolutions basées sur une intégration des différents types complémentaires d'évolution (des modèles de procédés logiciels et des modèles de produit ou de schéma de bases de données). Ceci, par une étude globale cernant tous les aspects ayant trait à l'évolution.

Notre travail a commencé par une classification des environnements de génie logiciel, selon la technique d'intégration et le niveau de généricité. Une typologie des systèmes a été donné en soulignant les points faibles et les point forts de chaque catégorie. Nous avons ensuite mis en exergue l'importance des environnements centrés procédés et leurs impacts sur le développement des systèmes logiciels. Quelques exemples des ECPs existants ont été présentés.

La deuxième partie a été consacrée à l'évolution des modèles de procédés logiciels, une définition ainsi que plusieurs approches d'évolution ont été présentées. Les solutions dégagées par les systèmes actuels pour traiter l'évolution des modèles de procédés logiciels ne sont pas encore satisfaisantes, comme en témoignent des nombreux travaux. Pour cela nous avons proposé une approche d'évolution qui permet cette évolution.

Enfin, le modèle que nous avons proposé traite l'évolution selon deux niveaux: Le niveau modèle et le niveau instance. L'évolution au niveau modèle de procédé permet de gérer toutes les modifications inhérentes à la conception du procédé logiciel. L'évolution au niveau des

instances tente de contrôler l'exécution du procédé logiciel par un jeu de simulation/modification.

La gestion des versions de modèles de procédés (versions de schémas) constitue l'une des principales perspectives à ce travail, ajoutée à l'intégration de l'IA pour le contrôle de la coopération dans un modèle de procédé.

D’autres perspectives concernant la prise en compte de toutes les opérations de modification dynamique est de rendre possible l’utilisation de toutes les fonctions d’interaction avec le système à l’aide d’une interface programmable.

Il semble intéressant d’étudier la façon dont pourrait évoluer le méta-modèle lui-même par la modification des méta-classes et d’analyser les retombées que pourrait apporter une telle évolution, ainsi que la cohérence du système.

BIBLIOGRAPHIE :

[Alloui 96] I. Alloui. “Peace+ : un formalisme et un système pour la coopération dans les environnements de génie logiciel centrés processus”. Thèse de Doctorat, Université Pierre Mendès France, Grenoble, France, Septembre 1996.

[Amiour 99] Vers une fédération de composants interopérables pour les environnements centrés procédés logiciels.
Mahfoud Amiour. Thèse pour l’obtention de diplôme de docteur de l’université de Joseph Fourier-Grenoble 1. Juin 1999

[Bandinelli et al. 94] S. Bandinelli, M. Barga, A. Fuggetta, C. Ghezzi, and L. Lavazza. “SPADE An Environment for Software Process Analysis, Design and Enactment”.Software Process Modeling and Technology (eds. A. Finkelstein, J. Krammer, B. Nuseibeh), Research Studies Press, Taunton, 1994.

[Barghouti and Kaiser, 1990] Modeling concurrency in rule-based development environments. IEEE Expert, 5(6): 15-27.

[Barghouti and Kaiser, 1992] Scaling-up rule-based development environments. Int’l Journal of Software Engineering and knowledge Engineering, 2(1): 57-78.

[Basili et al. 92] V. Basili et al. “The Software Engineering Laboratory- An Operational Software Experience Factory”. In Proc. 14th. Int’l. Conf. on Software Engineering, IEEE CS Press, 1992.

[Bel, 84] Unix System V Documenter’s Workbench introduction and Reference Manual, AT&T Bell Labs., avril 1984.

[Belkhatir et al., 1992] Belkhatir, N., Estublier, J., Melo, W.L., and Ahmed Nacer, M. "Supporting software maintenance evolution processes in the Adele system."In Pancake, C.M. and Reeves, D.S., editors, Proc. of the 30th Annual ACM Southeast Conf., p. 165-172, Raleigh, NC.

[Belkhatir et al, 1993] Belkhatir, N. Estublier, J., Melo, W.L, (1993). "Supporting Software Maintenance Processes in Tempo". In Proc. of the conference on Software Maintenance, Montreal, Canada. IEEE Press.

[Belkhatir et Ahmed Nacer, 95] Supporting Evolution in software environments based on object roles software process models. Technology of Object-Oriented Databases and Software Systems.

[Ben-Shaul, 95] A paradigm for decentralized process modelling and its realization in the Oz Environment. Submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in the Graduate School of Arts and Science, Columbia.
Israel Z.Ben-Shaul. Columbia University 1995.

[Ben-Shaul and Kaiser 98] I. Z. Ben-Shaul and G. E. Kaiser. “Federating Process-Centred Environments: the Oz Experience”. ASE journal, Vol. 5, Issue 1, January 1998, Kluwer Academic Publishers.

[Boehm 86] B. W. Boehm. “A Spiral Model of Software Development and Enactment”. ACM SIGSOFT Software Engineering Notes. Vol. 11, N° 4, August, 1986.

[Booch 94] Grady Booch. « Conception Orientée Objet et applications ».

[Conradi et al. 92] R. Conradi, C. Fernstrom, A. Fuggetta and R. Snowdon. ”Towards a Reference Framework for Process Concepts”. Second European Workshop on Software Process Technology, EWSPT’92, Lecture Notes in Computer Science 635, J. C. Derniame Ed. 1992.

[Conradi et al. 94a] R. Conradi, M. Hagaseth, J. O. Larsen, M. N. Nguyen, B. P. Munch, P. H. Westby, W. Zhu, M. L. Jaccheri and C. Liu. “EPOS: Object-Oriented Cooperative Process Modelling”. In [Finkelstein et al. 94].

[Conradi et al. 94b] R. Conradi, C. Fernstrom and A. Fuggetta. “Concepts for Evolving Software Processes”. In [Finkelstein et al. 94].

[Conradi et al. 98] R. Conradi, A. Fuggetta and M. L. Jaccheri. “Six Theses on Software Process Research”. EWSPT’98, Weybridge, UK, Seprember, 1998.

[Conradi et Jaccheri 98] Process Modeling Languages. Chapter 3.
Reidar Conradi and M. Letizia Jaccheri. 1998.

[Conradi et Liu 99] Process Modeling Languages: One or Many?
Reidar Conradi, Chunnian Liu. Sep. 1999. September 19, 1999 – Submitted for EWSPT’95, April 1995, Leiden.

[Courington et al. 88] W. Courington, J. Feiber and M. Honda. “NSE Highlights”. Sun Technology, Winter 1988.

[Curtis et al., 92] B. Curtis, M. Kellner and J. Over. “Process Modelling”. Communications of the ACM, Vol. 35, No. 9, September 1992.

[Dami et al., 97] APEL : A Graphical yet executable formalism for process modeling. S. Dami, J. Estublier, M. Amiour. Technical Report, Adele Research Group, LSR-IMAG Laboratory, Mars 1997.

[DAR 87] DART A., ELLISON R.J., FEILER P.H., HABERMANN A.N., "Software Development Environments", Computer, novembre 1987, p.18-28.

[Don84] DOWSEAU-GOUGE V., KAHN G., HUET G., LANG B., LEVY J.J., "Programming Environments Based on structure Editors: the Mentor Experience", in "Interactive Programming Environments", Barstow D.R. et al. Ed., McGraw-Hill,1984, p.128-140.

[Dowson, 1993] Process variable and process change. In Schafer, W., editor, Proc. of the 8th Int'l Software Process Workshop, Germany. IEEE Computer Society.

[Estublier, 91] Configuration and process management. In Fuggeta, A., Ghezzi, C., and Conradi, R., editors, Proc. of the 1st European WorkShop on Software process modeling, Milan, Italy. AICA Press.

[Feldman 79] ou [FEL 79] S. I. Feldman. "Make- A program for maintaining computer programs". Software Practice and Experience, 9. 1979.

[Fer89] FERNSTROM C., OHLSSON L., "The ESF Vision of a Software Factory", Proc. of Int. Conf. on Software Development Environments & Factories, Berlin, Mai 1989.

[Gol, 83] GOLDBERG A., ROBSON R., Smalltalk 80.The language and its implementation, Addison-Wesley, 1983.

[Goldberg 84] A. Goldberg. "Smalltalk-80: The Interactive Programming Environment". Reading MA: Addison Wesley Publishing Company, 1984.

[Gra90] GRANGE J.L., OBBINK H., " An overview of the ATMOSPHERE Project", Proc. ESF Seminar, Berlin, novembre 1990.

[Grundy and Hosking 98] J. C. Grundy and J. G. Hosking. "Serendipity: integrated environment support for process modelling, enactment and work coordination. Automated Software Engineering, Vol. 5, No. 1, January 1998, Kluwer AcademicPublishers.

[Hendrick 96] A. "Hendrick La Qualité du Logiciel". http://www.spiral.lu/lil/qualite/wort_ampli.htm.

[Huff and Kaiser, 1991] Huff, K.E and Kaiser, G.E (1991). Change in the software process. In Thomas, I., eeditor, *Proc. of the 7th Int'l Software Process Workshop*, pages 10-13, San Francisco, CA. IEEE Computer Society Press.

[Huff 96] K. E. Huff. “Software Process Modelling”. Trends in Software Process, Pages 1- 24, A. Fuggetta and A. Wolf ed., Wiley and Sons Ltd, 1996.

[ISO89] ISON R. " An experimental Ada Programming Support Environment in the HP CASEEdge Integration Framework", LNCS 467, Springer-Verlag, 1989, p. 179-193.

[Jaccheri et al. 92] Software Process Modeling and Evolution in EPOS
Letizia Jaccheri, Jens Otto-Larsen, Reidar Conradi.
Technical Repport, Div. of Computer Systems and Telematics (DCST) Norwegian Institute of technology (NTH), Trondheim, Norway 1992.

[Jaccheri et al. 98] M. L. Jaccheri, G. P. Picco and P. Lago. “Eliciting Software Process Models with the E3 Language.”.TOSEM 7(4): 368-410, 1998.

[KAH 83] KAHN G., LANG B., MELESE B., " METAL: a Formalism to Specify Formalisms", Science of Computer programming, 3, 1983, p.151-188.

[Kai87] KAISER G.E., FEILER P.H., "Intelligent Assistance without Artificial Intelligence", Proc. 32th IEEE Computer Society Int. Conf., San Francisco, Février 1987, p.236-241.

[Kai88] G. E. Kaiser, P. H. Feiller and S. Popovich, “Intelligent Assistance For Software Development and Maintenance”. IEEE Software, 1988, p.40-49.

[Kaiser and al. 88] G. E. Kaiser, P. H. Feiller and S. Popovich, “Intelligent assistance for software development and maintenance”. IEEE Software, 5(3), 1988.

[Kim90] W.KIM «Research Directions in Object-Oriented Database Systems »

[LEH, 87] LEHMAN M.M., TURSKI W.M., " Essential properties of IPSEs", ACM SIGSOFT Software Engineering Notes, vol.12, n° :1, Janvier 1987, p.52-57.

[Lonchamp et al. 92] J. Lonchamp, C. Godart and j.C. Derniame.”Les environnements integrés de production de logiciel”. Technique et science informatique. Vol. 11-n°1,1992.

[Madhavji, 91] N. Madhavji. “The Process Cycle”. Software Engineering Journal, September 1991.

[Madhavji, 92] Madhavji, N. 1992. Environment evolution: The Prism Model of changes. Transactions on Software Engineering, 18(5):380-392.

[Madhavji and Schafer, 1991] Madhavji, N. and Schafer, W. (1991). Prism – Methodology and Process-Oriented Pnvironment. IEEE Transactions on Software Engineering, 17(12): 1270-1283.

[Mar 89] MARKOSIAN L. et al., "Knowledge-Based Software Engineering Using REFINE", in "Modern Software Engineering", Ng P.A., Yeh R.T. Ed., Van Nostrand Reinhold, 1989, p.448-478.

[Melo, 1993] Melo, W.L.(1993). TEMPO: Un Environnement de Développement de logiciels Centré Procédés de fabrication. Thèse de doctorat. Université Joseph Fourier. Grenoble.

[Montangero and Ambriola 94] C. Montangero and V. Ambriola. “OIKOS: Constructing Process-Centred SDEs”. In [Finkelstein et al. 94], 1994.

[Osterweil, 87] Software processes are software too. In Proc, of the 9th Int’l Conf. on Software Engineering, Monterey, CA.

[Paulk et al. 93] M. Paulk, B. Curtis, M. B. Chrissis and C. Weber. “Capability Maturity Model version 1.1”. IEEE Software 10:4 July 1993.

[Promoter 98] Promoter Group. “Software Process: Principles, Methodology, Technology”. Promoter Second Book Deliverable Book2, J. Derniame, B. Warboys and A. B. Kaba editors, 1998.

[Rep 84] REPS T., MAECEAU C., "Remote Attribute Updating for language-Based Editors", Proc. 13th ACM Symp. 0n Principles of Programming Languages, 1986. p.1-13.

[Rout 95] T. Rout. “Spice: A Framework for Software Process Assessment”. Software

[Royce 70] W.W. Royce. “Managing the development of large software systems”. In Proc. WESTCON, San Francisco, 1970.

[Sommerville 92] I. Sommerville. “Software Engineering”. Fourth edition, Addison-Wesly, 1992.

[Sutton and Osterweil 97] S. M. Sutton Jr. and L. J. Osterweil. “The Design of a Next-Generation Process Language”. Proceedings of the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering, Zurich, Switzerland,September 1997.

[Taylor et al. 88] R. N. Taylor, R. W. Selby, M. Young and F. C. Belz. “Foundation of the Arcadia Environment Architecture”. Proc. 3rd ACM Symposium on Practical Software Development Environments. Boston, November, 1988.

[Tei81a] TEITELBAUM T., REPS T., "The Cornell Program Synthesizer : a Syntax- Directed Programming Environment", CACM, 24, September 1981, p.563-573.

[Tei 81b] TEITELMAN W., MASINTER L., "The interlisp Programinig Environment" Computer, 14, avril 1981, p.25-34.

[Teitleman and Masinter 84] W. Teitleman and L. Masinter. “The Interlisp Programming Environment”. In Interactive Programming Environments. D.R. Barstow, H. E. Shrobe and E. Sandewall ed.. New York: McGraw-Hill, 1984, p. 637-654.

[Tic 85] TICHY W.F., "RCS- A System For Version Control", Software Practice and Experience, vol. 15, n°7, juillet 1985.

Was, 89 WASS ERMAN A.I., "Tool Integration in Software Engineering Environments" LNCS 467, Spring-Verlag, septembre 1989, p.137-149.