HAL
archives-ouvertes.fr

Tempo : un environnement de développement logiciel centré procédés de fabrication

Walcelio Melo Louzada Martins

▶ **To cite this version:**

Walcelio Melo Louzada Martins. Tempo : un environnement de développement logiciel centré procédés de fabrication. Génie logiciel [cs.SE]. Université Joseph-Fourier - Grenoble I, 1993. Français. tel-00005134

HAL Id: tel-00005134

https://tel.archives-ouvertes.fr/tel-00005134

Submitted on 26 Feb 2004

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THESE

présentée par

Walcélio LOUZADA MARTINS MELO

pour obtenir le titre de

Docteur de l'Université Joseph Fourier - Grenoble I

(arrêt ministériel du 5 juillet 1984)

spécialité : **INFORMATIQUE**

TEMPO :

Un Environnement de Développement Logiciel Centré Procédés de Fabrication

date de la soutenance : 22 Octobre 1993

COMPOSITIOIN DU JURY :

Mr. Noureddine BELKHATIR
Pr. Reidar CONRADI
Pr. Pierre-Yves CUNIN
Pr. Jean-Claude DERNIAME
Mr. Jacky ESTUBLIER
Pr. Carlo MONTANGERO

Thèse préparée au sein du Laboratoire de Génie Informatique
à l'Université Joseph Fourier - Grenoble I

Je tiens à remercier :

- à DIEU pour m'avoir donné tout ce que j'ai dans ma vie

- Au *Conselho National de Desenvolvimento e Pesquisa do Brasil (CNPq)* pour m'avoir soutenu tout au long de la préparation de cette thèse.

- Mr. Noureddine Belkhatir, Maître de Conférences à l'Université Pierre Mendès France de Grenoble, pour m'avoir proposé et assuré la direction de cette thèse et pour tout le temps qu'il m'a consacré pour son aboutissement. Ses remarques critiques et constructives, ses propositions, ses relectures patientes du manuscrit, enfin, son expérience et ses encouragements m'ont permis de faire de cette thèse un tout cohérent.

- Mr. Reidar Conradi, Professeur à l'Institut Norvégien de Technologie, Trondheim, Norvège, pour sa participation au jury de cette thèse.

- Mr. Pierre-Yves Cunin, Professeur à l'Université Joseph Fourier, qui m'a fait l'honneur de présider ce jury de thèse.

- Mr. Jean-Claude Derniame, Professeur à l'Université de Nancy I, d'avoir bien voulu accepter d'être le rapporteur de cette thèse, pour sa lecture attentive du document, pour ses critiques constructives et pour l'intérêt qu'il a porté à ce travail.

- Cette étude, menée au sein de l'équipe Adèle dirigée par Mr. Jacky Estublier, chargé de Recherches au CNRS, n'aurait pu aboutir sans le concours de celui-ci qui m'a offert un environnement de travail pour la réalisation de cette thèse.

- Mr. Carlo Montangero, Professeur à l'Université de Pise, Italie, pour avoir accepté de juger ce travail, pour sa lecture soignée du document et pour ses remarques qui m'ont permis d'améliorer ce manuscrit.

Mes collegues du Laboratoire de Génie Informatique pour avoir lu les premières versions de ce manuscrit : M. Amed-Nacer, B. Benatala, C. Gadonna, J.P. Chevallet, J.M. Favre, F. Gigemili, M. Herve, M. Michkour, P. Mulhem et L. Racault.

Je ne pourrais oublier Jean-Louis Cheval, Maître de Conférences à l'Université Pierre Mendès France de Grenoble, Mr. Wladimir de Castro Alves et Mlle Selma Arboui pour leurs critiques constructives.

Enfin, je voudrais spécialement remercier Mr. Yves Chiaramela, Professeur à l'Université Joseph Fourier et Directeur du LGI, Mr. Philippe Morat, Maître de Conférences à l'Université Joseph Fourier, et Mr. Flavio Oquendo, Maître de Conférences à l'Université Pierre Mendès France pour la confiance qu'ils ont mis dans mon travail et ma compétence professionnelle.

RESUME

Un aspect important de la prochaine génération des Ateliers de Génie Logiciel (AGL) est la possibilité de gérer un modèle de processus logiciel décrit explicitement par un formalisme exécutable. Les informations décrites dans un tel modèle seront utilisées par l'AGL pour automatiser quelques étapes du processus logiciel et pour guider les utilisateurs dans le déroulement des autres étapes.

Dans cette thèse, nous analysons les AGL dirigés par un modèle de processus logiciel. Nous décrivons un certain nombre d'AGL en regard des principales caractéristiques du formalisme proposé pour modéliser les processus logiciels et des mécanismes d' interprétation.

Nous présentons ensuite notre approche : **TEMPO -** un atelier de développement dirigé par les procédés de fabrication logiciel à base d'un formalisme orienté objet et supporté par le noyau Adèle. Dans ce formalisme, les activités logiciel sont structurées en unités de bases appelées ***types de processus*** devenant des ***environnements de travail*** à l'exécution. Nous montrons comment les aspects multi-comportementaux liés à l'utilisation des objets par ces activités sont exprimés en utilisant le concept de ***rôle***.

Nous développons également notre modèle de communication pour les processus coopérants. La coopération entre les différents processus logiciels est spécifiée par le concept de ***connexions programmables et actives***. La programmation des connexions est exprimée par des ***règles temporelles événement-condition-action***. L'activation est supportée par le mécanisme de déclencheurs. Ces règles permettent de programmer des stratégies de synchronisation entre les *processus coopérants*, en propageant des effets de bord dès qu'une action est exécutée sur un des points de la connexion.

Table des Matières

CHAPITRE I **Introduction 15**

I.1 Cadre de l'étude 15

I.2 Contribution de la thèse 20

I.3 Plan de la thèse 22

CHAPITRE II **Les Ateliers de Génie Logiciel pour la Programmation Globale 27**

II.1 Introduction 27

II.2 Evolution 28

II.3 Les environnements classiques 29

II.4 L'approche base de données 33

II.5 Conclusion 39

CHAPITRE III **Les Ateliers de Génie Logiciel Centrés Processus 43**

III.1 Introduction 43

III.2 Les modèles génériques de cycle de vie 44

III.3 Nouvelle génération d'AGL : l'intégration par les processus logiciels 50

III.4 Les formalismes de description des processus logiciels 52

III.5 L'approche procédurale 52

III.6 L'approche déclarative 60

III.7 L'approche transformationnelle 69

III.8 L'évolution de la description de processus logiciel 71

III.9 Un cadre conceptuel pour supporter les transactions 72

III.10 La programmation coopérative 74

III.11 Orientation des utilisateurs 76

III.12 Conclusion 77

CHAPITRE IV TEMPO : Un Formalisme pour la Description des Processus Logiciels - Concepts de Base et Langage d'Expression 83

IV.1 Introduction 83

IV.2 Critères de Choix 83

IV.3 Terminologie utilisée 85

IV.4 Le système TEMPO 85

IV.5 La modélisation des objets logiciels et sa relation avec les processus logiciels 86

IV.6 Description et contrôle des activités 94

IV.7 Les contraintes temporelles 100

IV.8 Les rôles des objets 105

IV.9 L'évolution des processus logiciels 113

IV.10 Le graphe d'états des occurrences de processus logiciels 127

IV.11 Conclusion 133

CHAPITRE V TEMPO : Un Modèle pour la Programmation Coopérative 137

V.1 Introduction 137

V.2 Motivation 138

V.3 Le modèle "check-in/check-out" 138

V.4 Notre proposition : les environnements de travail 139

V.5 Conclusion 152

CHAPITRE VI La Réalisation de TEMPO 157

VI.1 Introduction 157

VI.2 Le gestionnaire de ressources 158

VI.3 L'historique des objets 171

VI.4 Le gestionnaire de processus 174

VI.5 Implémentation de TEMPO : état de lieu 178

VI.6 Conclusion 179

CHAPITRE VII Conclusion et Perspectives 181

VII.1 Contribution du travail 181

VII.2 Perspectives 182

CHAPITRE VIII Références 185

ANNEXE A Publications dans le cadre de la thèse 197

Liste des Figures

Figure II.1 : Le modèle cascade 45

Figure II.2 : Le modèle en Spirale 47

Figure II.3 : L'évolution des AGL : vers l'intégration par le processus logiciel. 51

Figure II.4 : Un exemple des processus de mise au point en OPM/Galois. 57

Figure III.1 : L'architecture d'Adèle/TEMPO. 86

Figure III.2 : Les différents axes d'évolution dans TEMPO 117

Figure III.3 : Ajout de rôle. 118

Figure III.4 : La méthode d'évolution de TEMPO. 120

Figure III.5 : Le graphe d'état. 127

Figure III.6 : Les Rôles et la gestion de branches. 130

Figure III.7 : La suppression d'une instance de rôle. 131

Figure IV.1 : Le modèle "check-in/check-out". 139

Figure IV.2 : Les environnements de travail. 140

Figure IV.3 : La gestion de branches. 142

Figure IV.4 : La décomposition des environnements de travail. 143

Figure IV.5 : L'échange de messages par les connexions. 147

Figure V.1 : Les principaux composants de TEMPO. 157

Figure V.2 : Exemple de spécialisation d'un type d'historique 172

Figure V.3 : Manipulations des historiques 174

Figure V.4 : L'exécution des règles ECA et les transactions courtes. 177

Figure V.5 : Parcours des historiques pour les règles TECA. 178

CHAPITRE I Introduction 13

I.6 Cadre de l’étude 13

I.6.1 Les modèles de cycle de vie 13

I.6.2 Les Ateliers de Génie Logiciel 14

I.6.3 Synthèse 17

I.7 Contribution de la thèse 18

I.7.1 La description des modèles de processus logiciels 18

I.7.2 Les rôles des objets 19

I.7.3 La programmation coopérative 20

I.7.4 La modélisation des environnement de travail 21

I.7.5 Un début à l’évolution des processus logiciels 21

I.8 Plan de la thèse 22

CHAPITRE I Introduction

I.1 Cadre de l'étude

La production de logiciels complexes et de taille importante à un coût et dans des délais raisonnables continue à être un des objectifs de base du domaine du génie logiciel. Plusieurs axes de recherche ont été identifiés et suivis dans le but de résoudre ces problèmes. Nous nous intéressons à ceux portant sur l'amélioration des procédés de développement et de maintenance des produits logiciels, ou simplement les processus logiciels (Belkhatir et al., 1991; Benali et al., 1989; Conradi et al., 1991b; Montangero and Ambriola, 1993; Peuschel et al., 1992; Warboys, 1989b).

Le processus logiciel est, en général, défini comme étant l'ensemble des activités aussi bien techniques qu'administratives mises en oeuvre pour obtenir un produit logiciel (Feiler and Humphrey, 1993; Lonchamp, 1993). Ces activités vont de l'analyse des besoins jusqu'à la maintenance des logiciels.

Les processus logiciels sont aussi importants que les produits logiciels, car ils sont devenus plus complexes et donc plus difficile à comprendre, à contrôler et à mesurer que les produits eux-mêmes (Osterweil, 1987). D'où l'importance d'expliciter les politiques dirigeant les activités de développement et de maintenance afin de les adapter ensuite (Fuggetta et al., 1991; Derniame, 1992; Thomas, 1991).

I.1.1 Les modèles de cycle de vie

Au cours de ces dernières années, l'étude sur le processus logiciel a abouti à la conception de plusieurs modèles de cycle de vie (Madhavji, 1991). Ces modèles ont permis de mieux comprendre le processus logiciel et par conséquent de l'améliorer (Royce, 1970; Boehm, 1988).

Les modèles de cycle de vie ont permis de mieux comprendre les problèmes liés à la modélisation et au support des processus logiciels. Parmi les aspects notoires de ce travail, nous pouvons citer la réduction du coût de développement et de maintenance des systèmes complexes (Boehm, 1988).

Ces modèles ignorent cependant certains points pourtant essentiels pour le succès d'un projet logiciel comme la description détaillée du flux de données entre les étapes ou l'ordonnancement des activités à un niveau fin (Curtis et al., 1992; Madhavji, 1991). Cela rend la gestion, le contrôle et l'amélioration des processus logiciel extrêmement difficile (Humphrey, 1988).

I.1.2 Les Ateliers de Génie Logiciel

Parallèlement à la recherche concernant la conception et l'amélioration des modèles de cycles de vie, de multiples Ateliers de Génie Logiciel (AGL) visant supporter et contrôler les processus logiciels ont été construits.

L'évolution des AGL porte essentiellement sur le contrôle des objets manipulés pendant le processus logiciel. En effet, les modèles de données ont intégré la technologie base de données en évoluant d'un simple système de gestion de fichiers vers des modèles sémantiques et orienté-objets offrant des capacités très riches de modélisation. Ainsi, un consensus semble se dégager dans la communauté du génie logiciel où le modèle entité-relation-attribut, étendu avec les concepts orienté-objet, est considéré comme étant le plus adéquat pour la prise en compte des informations manipulées dans un AGL.

De puissants systèmes de gestion d'objets (SGO) ont été bâtis pour supporter ces modèles, comme par exemple Adele (Belkhatir and Estublier, 1987; Belkhatir et al., 1991), EIS (Kimball, 1992), PMDB+(Penedo, 1991) et le SGO de PCTE (Boudier et al., 1988; Oquendo et al., 1991). Le système de gestion d'objets est un composant essentiel dans un AGL. Il permet d'intégrer les outils en leur faisant partager des informations descriptives et structurelles sur les objets qu'ils manipulent. Il comprend aussi bien des mécanismes de gestion de données similaires à ceux des bases de données (contrôle d'accès, intégrité des données, etc), que des services propres aux activités de programmation globale (la gestion d'évolution des objets, l'optimisation de l'espace disque par des mécanismes de delta, transactions longues, etc).

Nous nous basons ici sur le modèle conceptuel d'AGL proposé par (Wasserman, 1989) pour décrire les axes d'évolution des AGL et par la suite situer notre travail qui cadre bien avec ce modèle.

1. **intégration par l'interface**. Cet axe d'intégration concerne la manière d'utiliser un AGL. L'objectif majeur de l'intégration par l'interface est d'améliorer les performances et l'efficacité d'utilisation d'un AGL par ses utilisateurs. L'intégration par l'interface est supportée par des systèmes de gestion de fenêtre.

   On dit qu'un AGL est bien intégré par rapport à cet axe lorsqu'un utilisateur qui a déjà appris à utiliser un outil d'AGL dépense un minimum d'énergie pour apprendre à utiliser un autre outil. Les outils qui composent l'AGL doivent donc utiliser le même paradigme d'interaction, comme par exemple le paradigme objet utilisé par les logiciels pour les ordinateurs de la ligne Macintosh.

   Dans le domaine de l'intégration par l'interface, MOTIF est considéré comme le standard *de facto* pour les AGL qui utilisent le système Unix comme plateforme.

2. **intégration par les données**, qui concerne le contrôle de l'utilisation des données par les outils qui composent l'AGL. On entend par l'intégration des données le fait que toute l'information manipulée par l'environnement est gérée comme un tout cohérent. Les mécanismes qui rendent possible l'intégration des données prennent en compte la représentation, la conversion et le stockage des données.

   Par rapport à cet axe d'intégration, nous pouvons identifier les propriétés d'intégration suivantes (Thomas and Nejmeh, 1992) :

   inter-operabilité. Le but est de minimiser les conversions de données entre deux outils qui coopèrent.

   non-redondance. Le but est de stocker le minimum d'objets qui peuvent être dérivés de manière automatique ou de limiter le plus possible la duplication des données.

   cohérence. Pour expliquer ce critère, supposons que nous avons deux outils A et B manipulant respectivement les objets OA et OB liés sémantiquement. Un AGL satisfait ce critère, lorsque l'outil A informe des actions effectuées sur OA et des effets provoqués afin que l'outil B puisse appliquer correctement des contraintes sur OB.

   échange de données. On cherche à diminuer le travail nécessaire à la transformation des données échangées entre les outils, que celles ci soient persistantes ou non.

synchronisation par les données. Lorsque les outils sont capables de s'échanger des informations à propos des modifications de valeurs effectuées sur des objets partagés.

Dans le domaine de l'intégration par les données, PCTE+ est considéré comme un standard d'intégration des données au niveau européen adopté par ECMA. Des SGO à base de PCTE+ ont été construits, comme par exemple le SGO Emeraude qui est une implémentation de PCTE+ (Boudier et al., 1988; Oquendo et al., 1991).

3. **Intégration par le contrôle**. Afin de supporter des combinaisons harmonieuses de fonctions, les outils composant un AGL doivent partager et échanger des services. L'outil fournissant les services ne doit pas être directement concerné par le fait que d'autres outils utilisent ses services. Afin de rendre possible le partage de services entre les outils, un AGL doit être intégré par le contrôle, donc fournir des mécanismes pour permettre la communication entre les outils. En général, l'intégration par le contrôle est supportée par des gestionnaires de messages, qui fournissent essentiellement trois types de communication : outil – outil, outil – service et service – service.

   Dans le domaine de l'intégration par le contrôle, le système SoftBench de HP semble être bien accepté. Le système SoftBench est une version commerciale et révisée de l'AGL nommé FIELD (Reiss, 1990) considéré comme un des premiers AGL basé sur l'intégration par le contrôle. FIELD est bâti sur un serveur de communication, qui permet le partage d'informations par des échanges de messages. Dans le but d'améliorer le processus de contrôle, FOREST (Garlan and Ehsan, 1990) a perfectionné le serveur de messages de FIELD en y introduisant un mécanisme basé sur des règles de sélection. L'utilisation de telles règles rend possible la description du décodage des messages échangés entre les outils.

   Le mécanisme de déclenchement ("*trigger*") a été également utilisé pour contrôler le partage et l'échange des services des outils dans un AGL, comme par exemple les AGL : Adèle 2 (Belkhatir et al., 1991), Alf (Canals et al., 1993) et PMDB+ (Penedo, 1991). D'autres systèmes, comme par exemple Oikos (Ambriola et al., 1990), utilisent la technique dite des tableaux noirs ("*blackboard*") dérivé du domaine de l'intelligence artificielle pour contrôler l'échange de messages entre les outils.

4. **intégration par le processus**. Cet axe d'intégration concerne le rôle joué par les outils qui composent un AGL pendant le déroulement d'un processus logiciel. L'objectif majeur de l'intégration par le processus est d'assurer que les outils disponibles dans l'environnement travaillent efficacement pour supporter les processus logiciels.

Un AGL est considéré comme étant intégré par le processus si (Thomas and Nejmeh, 1992) :

- **a.** la fonction d'un outil est cohérente vis à vis de l'objectif global du processus. C'est-à-dire que lorsque un objectif X est atteint par un outil O, cela permettra, ou aidera, d'autres outils à atteindre leurs objectifs;
- **b.** il y a un consensus général entre les outils sur les mêmes hypothèses;
- **c.** les outils obéissent à un plan commun d'exécution.

Néanmoins, dans le domaine de l'intégration par le processus, les avancées obtenues par les AGL de la génération actuelle ne sont pas très significatives. En plus des services de gestion de configuration, tels que le contrôle de versions et la reconstruction automatique, la génération courante fournit également :

- **a.** des facilités permettant le travail coopératif par la synchronisation des activités, comme celles fournies par l'environnement Flecse (Dewan and Riedl, 1993);
- **b.** des services de communication entre les équipes de développement, comme par exemple l'outil Coordinator;
- **c.** des moyens pour mieux gérer des projets logiciels, comme par exemple les AGL CoNeX (Hahn et al., 1991), Daida (Rose et al., 1991), etc.

I.1.3 Synthèse

Les AGL de la génération courante proposent des mécanismes pour mieux :

1. contrôler l'utilisation des ressources mises à la disposition des utilisateurs;
2. coordonner les activités menées de manière parallèle dans l'environnement;
3. faire respecter les politiques de communication entre les utilisateurs.

Cependant, ces AGL au niveau de la pratique restent toujours trop figés, car les processus logiciels qu'ils supportent ne peuvent pas être adaptés aux différents besoins des organisations.

Aujourd'hui, un consensus semble se dégager dans la communauté du génie logiciel sur le fait que l'élément essentiel de la prochaine génération des AGL sera la possibilité de gérer un modèle de processus logiciel décrit explicitement par un formalisme exécutable (Ambriola and Montangero, 1992; Derniame, 1992; Dowson, 1991; Fuggetta et al., 1991; Osterweil, 1993; Schafer, 1993; Thomas, 1991). Les informations décrites dans un tel modèle pourront être utilisées par l'AGL, afin que celui-ci puisse automatiser

quelques étapes du processus logiciel et guider les utilisateurs dans le déroulement des autres étapes. Ce formalisme doit permettre la modélisation de toutes les étapes du processus logiciel aussi bien à un niveau global que détaillé afin de rendre possible son utilisation pour l'AGL. Ainsi nous pouvons dire que l'on assiste à un changement de génération des AGL fortement intégrés où le modèle de processus logiciel est câblé par le constructeur de l'AGL vers une nouvelle génération où les AGL doivent être dirigés par un langage de programmation orienté processus logiciels (un formalisme exécutable).

I.2 Contribution de la thèse

Dans cette étude notre contribution se situe au niveau de la modélisation des procédés de fabrication logiciel, de leur évolution et de leur exécution. Avec TEMPO, nous proposons un formalisme exécutable orienté-objet permettant de décrire des procédés de fabrication logiciels coopérants et de fournir un environnement pour exécuter ce formalisme.

I.2.1 La description des modèles de processus logiciels

L'approche orientée-objet semble être l'élément fédérateur de différents domaines de recherche (Bézivin, 1990), comme par exemple la technologie base de données (e.g. O2, Gemestone, Orion), les langages de programmation (e.g., C++ et Eiffel) et plus récemment les ateliers de génie logiciel, comme par exemple Ipse 2.5 (Bruynooghe et al., 1991), OPM/Galois (Sugiyama and Horowitz, 1991) et PMDB+ (Penedo, 1991). Dans cette cette étude, nous avons adopté l'approche orientée-objet pour la modélisation des processus logiciels pour intégrer efficacement les aspects *produits* et *procédés* menant à leur production. Le formalisme exécutable ***TEMPO*** permet de décrire les activités de génie logiciel dans un style orienté-objet.

L'utilisation de l'approche orientée-objet comme l'élément fédérateur de TEMPO nous a possibilité de structurer des modèles de procédés logiciels, d'incorporer de manière aisé les aspects dynamiques (les méthodes et les contraintes d'exécutions sur ces méthodes) et d'uniformiser la description des produits logiciel avec la modélisation des procédés logiciel.

Un modèle de processus logiciel est vu comme un ensemble d'activités coopérantes où chaque activité est représentée par un type de processus logiciel. Un modèle de processus logiciel peut donc être décrit par un ensemble de ***types de processus logiciel*** où chacun représente une ou plusieurs étapes d'un cycle de vie spécifique.

A un type de processus logiciel peuvent être attachés des attributs permettant de décrire ses caractéristiques mais aussi des méthodes ainsi que de règles contrôlant leur exécution. Les occurrences de processus logiciels sont gérés comme des objets actifs. Les types processus logiciel peuvent être également spécialisés via le mécanisme d'héritage multiple. Pour que la décomposition d'une étape complexe en plusieurs sous-étapes soit possible, un type de processus logiciel peut est une agrégation de plusieurs fragments de processus moins complexes.

I.2.2 Les rôles des objets

Chaque processus logiciel peut spécifier quels sont les d'objets qui peuvent être consommés, modifiés et créés au cours de son exécution. Pour chaque objet manipulé par une occurrence d'un processus logiciel, il est possible de changer les caractéristiques (les attributs) et le comportement (les méthodes et les contraintes d'exécution de ces méthodes) de cet objet en fonction du rôle que cet objet joue au sein de cette occurrence. Cela est possible grâce au concept de ***rôle*** proposé dans TEMPO.

Les rôles ont deux avantages par rapport aux techniques d'héritage classiques :

1. Le modèle de processus logiciel peut évoluer plus aisément. Des rôles peuvent être ajoutés ou supprimés sans qu'une restructuration de la base de données soit nécessaire.
2. Les rôles peuvent être utilisés comme un mécanisme d'héritage dynamique. Un objet jouant un rôle peut avoir son comportement et ses caractéristiques modifiés dynamiquement au cours de sa vie. Cela peut être utilisé comme une manière d'implanter l'héritage dynamique.

I.2.3 La programmation coopérative

Plusieurs occurrences de processus logiciel peuvent s'exécuter en parallèle en partageant des objets. Il faut donc assurer leur synchronisation et leur. Le concept de ***connexion active et programmable*** est proposé pour exprimer les politiques de communication entre deux processus logiciel. Les connexions permettent de synchroniser deux occurrences s'exécutant en parallèle et de spécifier les conditions de communication et d'échange de leur résultats.

I.2.4 La modélisation des environnement de travail

Les activités de longue durée sont supportées par les environnements de travail créés pour chaque occurrence de processus logiciel. Les objets partagés par plusieurs

processus peuvent donc être modifiés dans chaque environnement de travail de manière isolée. Les environnements de travail sont organisés de manière arborescente. Ainsi une longue transaction peut se décomposer en plusieurs autres transactions longues où chacune est représentée par un environnement de travail.

I.2.5 Un début à l'évolution des processus logiciels

Le style objet du formalisme TEMPO, étendu avec les rôles des objets, rend la prise en compte des évolutions de processus inhérente au modèle sans développer des concepts et des mécanismes supplémentaires. Ainsi l'évolution est intégrée agréablement au modèle. L'ajout majeur réside dans le fait que les concepts de ***rôle*** et les ***règles temporelles événement-condition-action*** permettent l'expression naturelle de l'évolution incluant le transfert des instances affectées.

La modification au niveau processus logiciel permet un développement incrémental des procédés de production pour faire évoluer le modèle et traiter des situations d'exception. Ainsi les procédés de production évoluent par ajout de rôles. Un rôle pouvant être soit une vue d'objet soit un sous-processus (agrégat de rôles). Un processus se ramenant à une définition d'un ensemble de rôles encapsulés dans une unité de cohérence : le fragment processus. Ainsi l'évolution se fait par ajout de rôles avec des transitions assurées par des règles temporelles qui contrôlent le flux d'instances de rôles. Ce flux pouvant être intra-processus (évolution d'un rôle à l'intérieur d'un processus) ou inter processus par ajout de fragment processus.

Nous pensons ainsi offrir une base raisonnable pour spécifier les évolutions dans notre environnement de développement de logiciels sans avoir l'ambition de couvrir toutes les évolutions possibles de processus logiciels. Notre souci majeur est d'éviter les situations chaotiques dues à des évolutions non contrôlées qui entraînent à long terme la destructuration des produits logiciels et des procédés qui les produisent.

I.3 Plan de la thèse

L'objectif de cette étude est de fournir un environnement dans lequel les processus logiciels peuvent être définis explicitement, instanciés et exécutés. Après la présentation du contexte général de ce travail, les principaux travaux supportant les différents aspects liés aux processus logiciels sont présentés. Le plan de la thèse se décompose comme suit :

- Chapitre 2 : Les Ateliers de génie logiciel - vers l'intégration par les données.

Nous présentons l'évolution des AGL pour supporter la programmation globale. Cette étude divise les AGL en deux générations : les AGL dits classiques et les AGL dits fortement intégrés. Les AGL classiques sont analysés par rapport aux services qu'ils fournissent pour la gestion de version, la reconstruction automatique de gros logiciels, et finalement pour supporter le stockage des objets logiciels.

Dans la second partie de ce chapitre nous consacrons plus d'importance à ce dernier aspect : la gestion des objets logiciels dans un AGL. Nous expliquons pourquoi les systèmes de gestion de fichiers couplés à une base de données relationnelle ne sont pas appropriés pour supporter toute la complexité liée à la gestion des objets dans un environnement logiciel. Nous présentons ensuite la vague qui a suivi cette approche : les AGL intégrés à un système de gestion d'objets.

- Chapitre 3 : Les ateliers de génie Logiciel centrés processus logiciels.

  Nous présentons au cours du chapitre 3 les AGL dirigés par un modèle de processus logiciel. Nous présentons le rôle joué par les modèles de cycle de vie dans ce domaine en montrant pourquoi ces modèles ont été importants dans la gestion des processus logiciels. Nous donnons comme exemple le modèle en cascade et le modèle en spirale. Nous montrons les principales caractéristiques de ces deux modèles, commentant leurs principales avantages et leurs faiblesses. Ensuite nous présentons les formalismes utilisés pour modéliser les processus logiciels.

  D'une part, nous analysons les concepts fournis par ces formalismes pour supporter la description des différents aspects liés aux processus logiciels. D'autre part, nous étudions les différents mécanismes proposés pour supporter ces formalismes. Nous classons ces formalismes en trois grandes catégories, en fonction de l'approche qu'ils utilisent : procédurale, déclarative et transformationnelle. Cette classification suit dans une certaine mesure celle proposée dans d'autres travaux (Arbaoui, 1993; Conradi et al., 1991a; Lonchamp et al., 1992).

  Dans chaque approche, nous décrivons un certain nombre d'AGL que nous croyons les plus significatifs en donnant les principales caractéristiques du formalisme proposé pour modéliser les processus logiciels ainsi que le mécanisme d'interprétation utilisé par ces AGL.

Les trois chapitres suivants présentent notre approche.

- Chapitre 4 : TEMPO - un formalisme pour la description des processus logiciels - concepts de base et langage d'expression

  Ce chapitre aborde le problème de définition d'un formalisme pour les processus logiciels présentant le formalisme qui a été retenu. On montre les concepts de base

et le langage d'expression. Après avoir introduit le problème, nous donnons les principaux points qui nous ont poussé à concevoir TEMPO. Tout d'abord, nous consacrons notre attention aux aspects liés à la description et au contrôle des activités de génie logiciel en présentant les concepts proposés pour modéliser les processus logiciels. Ensuite, nous explicitons le problème lié à la description et à la gestion de multiples points de vue en montrant comment notre approche fournit une solution à ce problème par le concept de ***rôle*** (Belkhatir and Melo, 1992a). Nous expliquons l'importance de la relation entre la modélisation des objets logiciels et les activités dans un AGL centré processus en montrant comment ces aspects sont abordés par notre approche. Ensuite nous mettons en évidence les aspects concernant l'évolution des processus logiciels en présentant la façon dont TEMPO prend en compte ces aspects. Nous décrivons les opérations permettant de mettre à jour des modèles de processus logiciel.

- Chapitre 5 : TEMPO - un modèle d'exécution pour les processus coopérants

  Nous présentons dans ce chapitre comment le travail coopératif est supporté dans TEMPO par les environnements de travail. Nous montrons le modèle de communication utilisé pour décrire les politiques de communication entre les environnements de travail (Belkhatir and Melo, 1993).

- Chapitre 6 : La réalisation de TEMPO

  Dans ce chapitre nous présentons les aspects de prototypage de TEMPO. Pour cela, nous décrivons tout d'abord le système Adele (Belkhatir et al., 1991). Ensuite nous montrons comment nous avons utilisé cette plateforme pour la réalisation de TEMPO (Belkhatir and Melo, 1992b).

CHAPITRE II Les Ateliers de Génie Logiciel pour la Programmation Globale 27

II.9 Introduction 27

II.10 Evolution 28

II.11 Les environnements classiques 29

II.11.1 Gestion de Versions 29

II.11.2 Reconstruction automatique 30

II.11.3 Les modèles de système 31

II.11.4 Discussion 32

II.12 L’approche base de données 33

II.12.1 Les bases de données vers le génie logiciel 34

II.12.2 Le génie logiciel vers les bases de données 35

II.12.3 Evolution du modèle de données 37

II.12.3.1 Expériences 37

II.12.3.2 Les classes de modification du schéma 38

II.13 Conclusion 39

CHAPITRE II Les Ateliers de Génie Logiciel pour la Programmation Globale

II.1 Introduction

Le terme d'Atelier de Génie Logiciel (AGL) a été introduit au cours des années 80 pour désigner un ensemble intégré d'outils, de techniques et de méthodes assistant le développement des produits logiciels durant tout leur cycle de vie (spécification des besoins, conception, mise au point, test et maintenance).

Alors que les outils visant à supporter les phases de mise en oeuvre et de test ont déjà une longue tradition, les outils destinés aux phases de spécification des besoins et de conception ont fait leur apparition vers la fin des années 80. Ces outils utilisent, en général, largement les facilités de manipulation graphique, grâce a une disponibilité croissante de stations de travail de plus en plus puissantes, et s'appuient sur une base de données chargée de contrôler l'échange d'informations entre les outils composant l'AGL.

Les AGL ont, dans un certaine mesure, permis d'augmenter la productivité des équipes de développement et la qualité des produits logiciels (Chikofsky, 1988). Cependant de nombreuses raisons ont fait qu'au niveau de l'état de la pratique leur synergie est en fait en deçà de ce qui était attendu. Parmi les inconvénients de ces outils, nous pouvons citer :

1. La plupart des AGL aujourd'hui disponibles ont été conçus figés, c'est à dire qu'ils ne fournissent pas de mécanismes d'intégration permettant d'étendre leurs fonctionnalités. En général, ils sont construits pour supporter un nombre pré-déterminé de techniques et de méthodes de développement et de maintenance, comme par exemple l'outil Mecano (Girod, 1991).
2. Les produits logiciels sont devenus très complexes et leur développement implique une durée de vie très longue et des équipes nombreuses. Or, la

grande majorité des AGL n'offrent pas de support pour la programmation globale. Les services les plus développées sont ceux relatifs aux contrôle de révisions des codes sources, et à la synchronisation des activités, comme ceux fournis par exemple par l'AGL Dsee (Leblang and Chase, 1985).

Les différentes étapes de l'évolution des AGL sont décrites par rapport aux différentes approches proposées pour résoudre les problèmes concernant l'intégration des données et des activités.
En ce qui concerne l'intégration des données, nous nous intéressons aux outils qui composent un AGL utilisant un modèle de données partagé.
Pour l'intégration des activités, notre première préoccupation concerne la capacité d'un AGL à incorporer un modèle de processus logiciel et à utiliser ce modèle pour coordonner l'activation et l'utilisation des outils qui composent l'AGL.

II.2 Evolution

Pour supporter d'une manière adéquate les activités de programmation globale (DeRemer and Kron, 1976; Romamoorthy, 1986), un AGL a besoin de traiter principalement deux types d'informations : des informations statiques et des informations dynamiques.
Les premières modélisent les différents composants logiciels tels que les documents de conception, la documentation, les jeux de tests, les codes sources, les objets dérivés, etc, leurs relations et leurs évolutions en versions (Tichy, 1985).
Les informations dynamiques, c'est-à-dire le modèle de comportement et les activités de contrôle, décrivent la façon de développer et de maintenir un produit logiciel, de partager et de synchroniser les activités entre les différents utilisateurs, d'intégrer de nouveaux outils afin de contrôler l'exécution des outils composant un AGL. Le modèle de comportement est aussi utilisé pour décrire les méthodes, les procédés logiciels et les conventions gouvernant les processus logiciels. Ce modèle permet à l'AGL de garantir le suivi effectif de ces politiques.

Cette distinction nous mène à considérer trois étapes majeures dans l'évolution des AGL : les environnements classiques, les environnements intégrés bases de données et finalement les environnements centrés processus.

Dans les deux sections suivantes, nous analysons les environnements classiques et les environnements intégrés bases de données. Les environnements centrés processus sont étudiés dans le chapitre suivant.

II.3 Les environnements classiques

Les environnements classiques ont été les premiers AGL à prendre en compte les problèmes relatifs à la programmation globale. Ils ont généralement été bâtis sur d'un système de gestion de fichiers (SGF), par exemple le SGF d'Unix. Ils se caractérisent par une approche ensembliste:

> *les outils qui composent ces premiers environnements sont mis ensemble sans vraiment être intégrés.*

L'environnement "*Unix programmer workbench*" est représentatif de cette génération. Il se compose d'un ensemble d'outils d'aide aux activités de programmation (yacc, lex, compilateurs, etc.). L'intégration entre ces différents outils est accomplie par le système de chaînage d'entrée-sortie propre au système Unix ("pipe"). La génération de ces AGL est très simple. Pour ajouter une nouvelle fonction il suffit de mettre à la disposition des utilisateurs un nouvel outil dans l'environnement. Malgré cet avantage, aucune contrainte sur l'utilisation des ces outils ne peut être décrite ni imposée de manière automatique par l'environnement. Les processus logiciels n'étant pas pris en compte par l'AGL, les activités menées par les utilisateurs d'environnement se déroulent sans contrôle, sans coordination et sans synchronisation.

Bien que les informations dynamiques soient faiblement contrôlées par les AGL de cette génération, les informations statiques sont de plus de en plus prises en compte essentiellement grâce aux avances réalisées dans le domaine des bases de données. Par rapport à l'environnement *"Unix programmer workbench*" qui ne supporte que de simples fichiers Unix, les AGL ont fait de considérables progrès dans la gestion des versions, les activités de reconstruction automatique de gros logiciels et la structuration des logiciels. De cela, nous pouvons dégager trois étapes de l'évolution des environnements classiques des AGL : au début la gestion de versions, ensuite la reconstruction automatique et enfin les modèles de systèmes.

II.3.1 Gestion de Versions

Par gestion de versions, nous entendons le contrôle d'évolution d'un produit logiciel. Ceci inclut, entre autres services, le contrôle du code source, la gestion de la mise en disponibilité d'un nouveau produit, le suivi des modifications, le compte rendu des problèmes de maintenance, etc.

Tous les systèmes de gestion de contrôle de versions maintiennent une base d'archivage des données où les différents types de composants logiciels sont stockés. Dans les premiers systèmes, cette base n'était qu'une simple structure de fichiers construite et

contrôlée par le système d'exploitation subalterne, comme par exemple SCCS. D'autres gestionnaires de versions ont utilisé par la suite un système de gestion de bases de données, comme par exemple la base de données Damokles (Dittrich, 1989). Les plus récents sont bâtis autour d'un système de gestion d'objets, comme par exemple les AGL Adèle (Belkhatir et al., 1991) et PACT (Thomas, 1989b). Plusieurs outils visant à supporter la gestion de versions ont été proposés et construits, comme par exemple les outils SCCS (Rochkind, 1974) et RCS (Tichy, 1985).

SCCS a été un des premiers outils à mettre en oeuvre le concept de version d'objets, donnant une solution au problème d'économie d'espace disque, via le mécanisme de *delta* (Rochkind, 1974). Le problème de la gestion d'historiques et le problème du partage des objets est résolu par le mécanisme de réservation et de blocage d'objets. Un objet est vu par SCCS comme composé de plusieurs versions, chaque version n'étant manipulé que par des commandes de SCCS. Ces commandes rendent possible l'extraction, à partir du système de stockage de SCCS, d'une version spécifique d'un objet (*"check-out"*) et la création d'une nouvelle version à chaque fois qu'un objet est mis-à-jour (*"check-in"*). Une version d'un objet extraite du système de stockage de SCCS ne peut être modifié pour les autres utilisateurs.

Bien que les mécanismes de gestion de versions proposé par SCCS soient très largement utilisés, ils possèdent néanmoins deux contraintes importantes :

1. l'outil SCCS ne permet pas la modification parallèle d'un même objet par différents utilisateurs.
2. l'outil SCCS ne fournit pas de services de réutilisation d'objets dérivés.

Pour résoudre le problème de partage d'objets, l'outil RCS (Tichy, 1982) a introduit le concept de variante ("*branche*"). Chaque objet est géré comme un arbre de versions, dont chacune des branches est une variante. A chaque fois qu'un utilisateur a besoin de manipuler une version d'un objet simultanément avec un autre utilisateur, une nouvelle branche est créée, réservée et mise à sa disposition.

L'AGL Epos est bâti sur un gestionnaire de versions propre au projet Epos (Conradi and Holager, 1990) permettant l'utilisation parallèle d'une même version d'objet par plusieurs utilisateurs (Conradi et al., 1989).

II.3.2 Reconstruction automatique

La reconstruction automatique est généralement définie comme étant l'activité de génération d'une version exécutable d'un produit logiciel. Lorsque un produit logiciel

est construit ou modifié, il faut s'assurer que les éléments logiciels sélectionnés une fois assemblés forment un produit dont les caractéristiques et le comportement lors de l'exécution sont celles souhaitées. Un historique doit être généré et maintenu pour contenir les composants sélectionnés, leurs inter-relations, leurs caractéristiques, les procédures de construction utilisées avec leurs contenus, etc. Vu la complexité des gros systèmes logiciels, la nature éphémère des objets et des structures contrôlées ainsi que le nombre important de changements des composants logiciels, l'automatisation des activités de reconstruction joue un rôle fondamental dans la programmation globale. Pour cette raison, les AGL tentent d'incorporer des outils de reconstruction automatique pour supporter ce genre d'activités.

Make (Feldman, 1979) est l'outil le plus populaire utilisé pour la reconstruction automatique. L'outil Make représente, par un fichier "*Makefile*", la correspondance entre le code source d'un produit logiciel avec ses objets dérivés et les procédés utilisés pour générer ceux-ci. Il utilise le système de gestion de fichiers d'Unix pour stocker et gérer les objets manipulés. Il ne contrôle que des activités de reconstruction automatique dans un contexte mono utilisateur et mono version. L'outil Make ne peut pas assurer que les informations décrites dans "*Makefile*" sont celles du monde réel.

L'outil Odin (Clemm and Osterweil, 1990) étend les fonctionnalités offertes par Make en typant les objets primitifs, les objets dérivés et les objets composés. Pour reconstruire de façon automatique un produit logiciel, il faut décrire les règles de transformation de types qui, une fois mises ensemble, forment le graphe de transformation des types. L'outil Odin possède son propre système de stockage d'objet où le code source, les objets dérivés et les descripteurs d'outils sont stockés.

L'outil Build (Waters, 1989) a élargi les concepts d'Odin en réalisant la séparation entre le modèle de données et le modèle d'activités de reconstruction. Dans l'outil Build, les objets comme les relations sont typés. Il permet la spécification des règles de transformation de types, comme dans l'outil Odin, avec en plus des règles de transformations applicables aux relations entre les composants logiciel. Au contraire de l'outil Make, qui est un outil d'usage professionnel, les outils Odin et Build sont des prototypes universitaires.

II.3.3 Les modèles de système

La dernière étape d'évolution des AGL classiques concerne donc les AGL basés sur des modèles de système. Un modèle de système présente la description d'un produit logiciel, ou une partie de celle-ci. Ce modèle rend donc possible la définition des propriétés statiques et structurales d'un produit logiciel. C'est-à-dire que les relations qui existent

entre les différents composants d'un produit logiciel peuvent être décrites. Il est également utilisé pour décrire les codes sources, les objets dérivés avec les outils qui ont été utilisé pour leur dérivation, les règles de reconstruction automatique, l'identification des bibliothèques, les chemins d'accès des objets logiciels, et enfin, toutes les informations pertinentes pour la génération des configurations (Perry, 1987). Les outils qui composent l'AGL utilisent les informations disponibles dans le modèle de système afin de s'exécuter de manière compatible avec les politiques de génération des nouvelles versions, de compilation, de construction des nouvelles configurations, etc, définies par ce modèle.

Les AGL Dsee (Leblang and Chase, 1985) et Jasmine (Marzullo and Wiebe, 1986) sont des exemples de systèmes qui ont largement utilisé les concepts et les mécanismes proposés par l'approche "modèle système". Dsee fournit d'autres services pour la programmation globale, comme par exemple, le contrôle de versions pour l'évolution des objets, l'outil de reconstruction distribué afin d'augmenter les performances des activités de reconstruction (Leblang and Chase, 1988), l'historique des activités qui permet de suivre le processus logiciel, et un service primitif qui permet de notifier aux utilisateurs certains événements. Par rapport à l'AGL Dsee, le principal service fourni par Jasmine, est que les opérateurs fonctionnels permettent aux utilisateurs la construction de nouvelles relations en utilisant des relations primitives fournies par Jasmine.

II.3.4 Discussion

Les modèles système ont apporté une nouvelle description des composants et de leur relation, mais la gestion des composants reste classique. Bien que les composants et leurs inter-connexions soient décrits dans un modèle de système, les composants sont stockés dans un système de gestion de fichiers classique (SGF), par exemple Unix. Comme le niveau des concepts fourni par les SGF est trop bas, la modélisation et le contrôle des informations statiques manipulés par l'AGL sont des activités coûteuses, difficiles et souvent sujettes à des erreurs dûes à l'intervention humaine. Une autre déficience grave des SGFs concerne l'intégration des outils. Les SGFs ne fournissent pas un schéma de données commun par lequel les outils composant l'AGL peuvent partager les données et leur définition. Cela oblige à faire des transformations coûteuses des données entre les outils ou à connaître dans le détail les formats des données utilisés par les outils qui veulent échanger ces données.

Un autre défaut de l'approche "modèle système" est directement issu de sa simplicité. Malgré son efficacité et les facilités de modélisation, les mécanismes de contrôle et de stockage des données restent rudimentaires, figés et limités à des domaines très

spécifiques d’application. Par exemple, le système Cedar ne supporte qu’un seul langage, le système Gandalf (Habermann and Notkins, 1986) est dépendant d’une seule méthode. Cela est dû au fait que les *modèles de systèmes* ont été développés avec une vision particulière d’un environnement. Ils ne peuvent pas être généralisés à d’autres domaines d’application. Toute information manipulée est codée de manière statique et la gestion du processus de développement est de bas niveau.

II.4 L’approche base de données

La communauté de Génie Logiciel a pensé que les concepts développés par les systèmes étudiés dans les sections précédentes, tels que le modèle de système, pouvaient être supportés par les base de données relationnelles. Il a donc parut logique d’utiliser ces bases comme plateforme d’AGL. La fonction de gestion des composants logiciel manipulées par un AGL, ainsi que leurs relations, est alors déléguée à un Système de Gestion de Base de Données (SGBD). Cependant ces bases de données doivent également fournir des facilités pour la définition, la classification et l’évolution de ces composants.

Deux approches ont été suivies. La première a consisté à construire un AGL sur une base de données relationnelle couplée à un système de gestion de fichier. Ce dernier sert à stocker les objets de grande taille, c’est-à-dire les codes sources, des codes binaires, etc. La base de données sert à gérer la définition des composants logiciels, ses attributs et leurs relations. Les AGL PMDB (Penedo, 1986) et Sodos (Horowitz and Williamson, 1986) sont des exemples de systèmes construits suivant cette approche.

La deuxième approche consiste à construire une base de données supportant un modèle de données sémantique (Peckhan and Maryanski, 1988). Dans cette approche, l’AGL est bâti sur un système de gestion d’objets capable de supporter l’évolution des données en plusieurs versions, et dirigé par un modèle de données voisin du modèle entité-relation élargi avec le concept de versions. Damokles (Dittrich, 1989) et Cactis (Hudson and King, 1988) utilisent cette approche.

Ces deux approches sont traitées dans les sections suivantes. Nous montrerons que la première approche peut être considérée comme une évolution du domaine des bases de données vers le génie logiciel, tandis que la deuxième concerne l’évolution inverse, c’est-à-dire du génie logiciel vers les bases de données.

II.4.1 Les bases de données vers le génie logiciel

Plusieurs projets ont considéré les bases de données relationnelles comme une technologie viable pour répondre aux besoins du génie logiciel. A partir d'une base de données nous pouvons aisément gérer une grande partie des informations manipulées durant le processus logiciel. Aux années 80, les bases de données relationnelles, étaient jugées satisfaisantes. Le modèle relationnel permet de décrire explicitement les types d'objets et les contraintes associées sous forme de définition de relations et d'attributs, contrairement aux modèles de systèmes où ces informations sont figées. Suivant la base de données utilisée, des contraintes relationnelles peuvent également être assurées par un système de déclenchement intégré à celle-ci.

Le système PMDB (Penedo, 1986) est un exemple typique qui a marqué cette époque. PMDB a eu pour but de gérer la presque totalité des informations manipulées dans un projet logiciel autour d'un système d'information central permettant la consultation et la mise à jour de ces informations. Ce système a été dirigé par un modèle voisin du modèle entité-relation permettant la caractérisation des objets aussi bien que des activités d'un projet logiciel, y compris ses attributs et ses associations avec d'autres objets et d'autres activités.
Comme PMDB a été bâti sur une base de données relationnelle, cette base a été couplée avec un système permettant d'un côté de traduire les définitions faites dans le modèle PMDB vers la base de données. D'un autre côté, ce système a été aussi chargé de transformer les requêtes et les mises à jour d'informations du PMDB vers la base de données. Il est difficile de gérer la cohérence entre les informations de haut niveau décrites par le modèle de données propre à PMDB et les informations gérées par la base de données relationnelle sur laquelle PMDB a été bâti. Récemment PMDB a évolué vers un système orienté objet, nommé PMDB+ (Penedo, 1991).

• Synthèse

Malheureusement, la plupart des travaux dans cette direction ont échoué. Ceci est dû à la différence existante entre la technologie des bases de données disponibles au début des années 80, c'est à dire la technologie relationnelle, et les besoins du génie logiciel. L'expérimentation menée, en particulier par PMDB, avec les bases de données relationnelles a montré que de telles bases ne permettent pas une expression sémantique suffisamment élevée pour supporter les AGL. Les bases de données relationnelles ont été conçues essentiellement pour répondre aux besoins des applications commerciales. Le conclusion est que les AGL conçus avec cette architecture ne sont pas satisfaisants et ont soulevé les problèmes suivants :

1. les objets complexes sont très difficiles à modéliser. Le modèle relationnel ne fournit aucune facilité pour structurer les objets (une des prémisses élémentaires du modèle relationnel est que les objets doivent être mis à plat par les formes normales pour une meilleure gestion);
2. absence d'encapsulation. Les opérations applicables sur un objet ne sont pas exprimées dans le modèle, mais laissées à la charge des applications;
3. en général, le contrôle des activités, c'est à dire les aspects dynamiques d'un projet logiciel, n'est pas pris en compte, car ces systèmes ne fournissent pas de mécanismes suffisamment puissant pour décrire des contraintes et des règles de gestion d'objets.

II.4.2 Le génie logiciel vers les bases de données

La technologie des bases de données au début des années 80 ne répondait pas aux besoins exprimés dans le domaine du génie logiciel. La communauté du génie logiciel a été donc amenée à concevoir elle-même des bases de données pour les applications non-conventionnelles. Ainsi, au lieu de se préoccuper de la construction de bases de données capables de manipuler rapidement des milliers d'informations, les bases de données pour le génie logiciel ont été plutôt conçues pour palier aux problèmes non résolus par les bases de données relationnelles, c'est-à-dire : supporter des modèles de données sémantiques, gérer des objets complexes et fournir des mécanismes pour mieux contrôler les activités de génie logiciel.

Des bases de données spécifiques pour le génie logiciel, comme par exemple Adele 1, Damokles et Marvel et des plateformes, telles que PCTE (Boudier et al., 1988) et CAIS (Oberndorf, 1988), ont été proposés pour prendre en compte ce genre de problèmes.

Damokles (Dittrich, 1989) a étendu le modèle entité-association en rendant possible la modélisation des objets complexes. Damokles supporte également les versions d'objets et les transactions longues. Du point de vue dynamique, Damokles a introduit la notion de déclencheur (*trigger*). Les déclencheurs permettent de contrôler des aspects liés à la gestion dynamique des informations, c'est à dire le processus logiciel. Ils fournissent aux utilisateurs la possibilité de définir des contraintes d'intégrité via des règles permettant ainsi d'automatiser certaines activités.

Une règle Damokles est un couple <événement, action>, où l'événement décrit une situation spécifique de modification de la base de données, et une action représente la réaction à cette situation. Les événements sont toujours exprimés par rapport aux opérations de modification de l'état des données, et les actions sont des contraintes

définies par l'administrateur du système qui peuvent déclencher d'autres opérations de modifications ou de notification.

L'outil Adele 1 (Belkhatir and Estublier, 1987) a proposé un modèle de produits logiciels spécifiquement orienté vers la gestion de configuration pour les programmes modulaires. Ce modèle est basé sur la structuration hiérarchique des objets, où un objet complexe peut être composé d'objets complexes ou d'objets primitifs. La principale et la plus importante facilité fournie par Adele 1 est sa capacité à gérer des configurations de logiciel en adhérant à un modèle de produit câble dans le système (Estublier et al., 1984). Cependant le processus logiciel n'est pas explicité par un modèle d'activités, comme c'est le cas dans la base de données Damokles. Le seul service fournit par l'outil Adele 1 pour prendre en compte les informations dynamiques est le mécanisme de gestion de configurations qui est câblé dans le système. Le projet Nomade (Belkhatir, 1988; Estublier and Belkhatir, 1988; Estublier and Belkhatir, 1989) a été justement développé dans le but de fournir un cadre où ces aspects peuvent être traités. Ce projet a abouti à la réalisation d'un produit nommé Adèle 2 (Belkhatir et al., 1991). Ce produit intègre aussi bien les aspects de modélisation des objets complexes que des contraintes d'intégrité sur l'utilisation de ces objets via un mécanisme de déclenchement ("*trigger*").

Dans l'AGL Marvel, toutes les informations statiques manipulées sont traitées de manière abstraite, comme des objets stockés dans le SGO (Kaiser et al., 1988a). Dans Marvel, les objets sont structurés par un modèle de données orienté objet où les composants logiciels sont modélisés en termes de classes. Associé a chaque classe, nous pouvons définir des attributs typés qui peuvent être hérités de multiples super classes. Un type d'attribut peut être : un type de base (entier, réel, chaîne de caractères, etc), un type texte pour prendre en compte les informations propres au génie logiciel (les codes sources, les binaires, etc.), ou une référence à une classe (attributs de référence). Les attributs de type référence permettent donc de décrire des objets complexes. Toutes les opérations de création, modification et suppression des objets sont guidées par le modèle de données

PCTE et CAIS ont beaucoup de points en commun. Les différences se trouvent plus au niveau des détails qu'au niveau conceptuel. Les deux systèmes s'appuient sur le modèle entité-association et représentent des interfaces publiques définissant des normes pour la gestion, l'accès, la modélisation des objets et la communication entre les outils. Bien que ces deux systèmes aient été proposés presque au même moment, PCTE par la communauté Européenne et CAIS par le Département de Défense Américain, le modèle proposé par PCTE a été plus largement accepté que celui de CAIS. Plusieurs AGL's ont

été bâtis en utilisant les normes proposées par PCTE, comme par exemple PACT (Thomas, 1989b) qui fournit des facilités supplémentaires, comme la gestion de configuration. Ni PCTE ni CAIS n’offrent des facilités pour rendre explicite le processus logiciel. CAIS reste un projet de recherche tandis que PCTE est devenu une norme Européenne et des AGL intégrant cette norme sont disponibles industriellement.

II.4.3 Evolution du modèle de données

Dans cette section nous décrivons les stratégies utilisés par les AGL intégrés à une base d’objets pour la prise en compte de l’évolution du schéma de données. Rappelons-nous que ces AGL utilisent un schéma de données où sont décrites les informations (objet, outils, etc) manipulées au cours de l’exécution des processus logiciels.

Comme le processus logiciel a une vie très longue, il est nécessaire de pouvoir faire évoluer la définition des informations statiques (descriptions des outils, des données, etc) manipulées par les activités menées dans l’AGL. Cela revient à mettre-à-jour le schéma de données. Il est préférable que cette évolution soit dynamique; c’est-à-dire qu’un processus en exécution ne doit pas être suspendu afin que la définition des objets qu’il manipule soit modifiée par le changement du schéma.

Dans la littérature, rares sont les AGL qui supportent une évolution du schéma de données. En général les AGL utilisent les services d’évolution déjà disponibles dans la base de données au-dessus de laquelle ils ont été bâti. D’autres AGL fournissent eux-même une méthode d’évolution. Dans le premier cas, nous trouvons Merlin (Peuschel et al., 1992) qui utilise les services de la base de données Gemstone (Butterworth et al., 1991a). Dans le deuxième cas, nous trouvons Marvel (Kaiser and Ben-Shaul, 1993) qui a implémenté lui-même son propre mécanisme d’évolution en utilisant les propositions introduites par le système Orion (Kim, 1988). Alf (Derniame et al., 1992) utilise les services fournis par PCTE.

II.4.3.1 Expériences

Les systèmes Gemstone (Penny and Stein, 1987) et Orion (Kim, 1988) convertissent les objets de la base pour les rendre conformes à la description de leur classe modifiée; toute incohérence de la base est ainsi éliminée. Cependant, l’inconvénient majeur est que la base est inaccessible pendant le temps (assez long) de réorganisation des objets et que l’ancien schéma ne peut plus être utilisé; les anciennes applications deviennent donc obsolètes.

La deuxième stratégie, appelée “*émulation*”, est basée, à l’opposé de la conversion, sur un versionnement des types (classes) (Skarra and Zdonik, 1986); elle permet d’avoir

plusieurs vues des instances du même type et les anciennes applications continuent d'utiliser l'ancien schéma.

Les systèmes adoptant l'approche de mise à jour immédiate, tels Gemstone (Penny and Stein, 1987) et O2 (Adiba and Collet, 1993; Bancilhom et al., 1992), ne font aucun suivi d'évolution. Les tranformations sont directement appliquées sur le schéma qui est considéré comme global et unique; aucun état précédent du schéma n'est conservé.

Orion (Kim, 1988) par contre, utilise le versionnement des schémas et des types. L'entité sur laquelle s'appliquent les transformations est le schéma. Le versionnement s'avère très utile lors des traitements simultanés de la base. La difficulté réside, cependant, dans la gestion du nombre important de versions. Toute modification de schéma, dans Orion, implique une version du schéma complet de la base.

Seul Alf/PCTE+ (Oquendo et al., 1991) utilise la notion de vue logique de la base par le concept de SDS (*"Schema Definition Sets"*). Le schéma de travail ("*working-schema*"), établi par la composition d'un ensemble de SDS, est modifié en rajoutant au schéma de travail de nouveaux SDS ou en mettant à jour les SDS existantes. La modification d'un schéma est alors considérée comme la modification d'une vue de la base.

II.4.3.2 Les classes de modification du schéma

On identifie généralement trois grandes classes de modifications (Banerjee et al., 1987) :

1. les modifications de types,
2. les modifications de liens d'héritage,
3. les modifications de définitions de types.

Cependant, le problème de l'évolution est traité différemment d'un système à un autre. Les solutions proposées ne sont généralement pas complètes et chaque système possède sa propre sémantique de modification.

Ainsi, dans le système Orion, la suppression d'un type implique la suppression de toutes ses instances et récursivement si ce type fait partie d'un type d'objet composite. L'opération de suppression n'est pas possible dans les systèmes Gemstone et O2 si le type possède des instances, ou si le type est référencé comme un paramètre dans la signature d'une méthode pour O2.

Les règles d'évolution sont non seulement largement arbitraires mais de plus elles sont imposées par le système. Par exemple Gemstone interdit de supprimer un type qui possède des instances dans la base, et dans Orion le domaine d'un attribut ne peut que se

généraliser. Cette approche simple, adoptée par la quasi totalité des systèmes, constitue un inconvénient car elle ne donne pas la possibilité aux utilisateurs d'adapter les règles d'évolution à leur type d'application.

II.5 Conclusion

Dans ce chapitre nous avons montré les différentes voies d'évolution que les AGL ont suivi pour supporter la programmation globale. Nous avons décrit :

1. l'évolution des modèles de données pour supporter la modélisation des informations statiques manipulées pendant le processus logiciel,
2. et l'évolution des modèles d'activités pour exprimer les procédés logiciels.

Les travaux de recherche concernant les modèles de données ont aboutis à des normes déjà valorisées, comme par exemple PCTE+. Par contre, le deuxième aspect concernant les modèles d'activités, ont à peine commencé à être traités dans les AGL. Le chapitre suivant est consacré à ce deuxième point. Nous allons montrer les tendances actuelles visant à régler ces problèmes et présenter les AGL expérimentaux les plus représentatifs.

CHAPITRE III Les Ateliers de Génie Logiciel Centrés Processus 43

III.1 Introduction 43

III.2 Les modèles génériques de cycle de vie 44

III.2.1 Le modèle en cascade 44

III.2.2 Le modèle en spirale 46

III.2.3 Evaluation 47

III.3 Nouvelle génération d'AGL : l'intégration par les processus logiciels 50

III.3.1 L' approche boite à outils 50

III.3.2 L'approche câblée 50

III.3.3 L'approche processus logiciels 51

III.4 Les formalismes de description des processus logiciels 52

III.5 L'approche procédurale 52

III.5.1 Arcadia/Appl-A 53

III.5.2 Triad/CML 55

III.5.3 OPM/Galois 56

III.5.4 Ipse 2.5/PML 58

III.5.5 **Synthèse 59**

III.6 L'approche déclarative 60

III.6.1 Les systèmes basés sur le mécanisme de chaînage 61

III.6.1.1 Marvel 61

III.6.1.2 Merlin 63

III.6.1.3 Synthèse 64

III.6.2 Les AGL basés sur les techniques de planning 64

III.6.2.1 Grapple 65

III.6.2.2 Epos 66

III.6.2.3 Synthèse 67

III.6.3 Les AGL basés sur le mécanisme de déclencheurs 67

III.6.3.1 Elen 68

III.6.3.2 AP5 68

III.6.3.3 Darwin 68

III.6.3.4 Synthèse 69

III.7 L'approche transformationnelle 69

III.8 L'évolution de la description de processus logiciel 71

III.9 Un cadre conceptuel pour supporter les transactions 72

III.10 La programmation coopérative 74

III.10.1 Communication sur la réservation des ressources 74

III.10.2 Communication sur les modifications d'objets 75

III.10.3 Communication sur un conflit 76

III.10.4 Négociation 76

III.11 Orientation des utilisateurs 76

III.12 Conclusion 77

CHAPITRE III Les Ateliers de Génie Logiciel Centrés Processus

III.1 Introduction

Ces dernières années, différentes études ont explicité le besoin d'un meilleur contrôle du processus de développement et de maintenance de gros logiciels (plus simplement appelé "*processus logiciel*") pour augmenter la productivité et accroître la qualité des produits logiciels (Dowson et al., 1991; Thomas, 1989a).

Le niveau de maturité des processus logiciels est à ses premiers balbutiements (Humphrey and Kellner, 1989). Nous manquons de méthodes adaptables à chaque entreprise et à chaque processus de production. Pour que ces méthodes produisent une vraie amélioration de processus, elles doivent être supportées par des outils permettant de développer de gros logiciels. Autrement dit, nous avons besoin de fournir d'une part un cadre conceptuel où les processus logiciels peuvent être représentés et d'autre part des mécanismes pour les automatiser.

Le *"processus logiciel"* est défini comme un ensemble intégré d'activités requises pour transformer les besoins de l'utilisateur en un logiciel fonctionnel y compris les activités nécessaires à la maintenance de ce logiciel (Feiler and Humphrey, 1993; Lonchamp, 1993). On parle aussi des procédés par analogie avec des procédés de fabrication.

Les processus logiciels sont structurés en phases distinctes. Une spécification particulière de la granularité des différentes phases, l'ordonnancement et le degré de concurrence entre les phases ainsi que leurs composants constituent un modèle de cycle de vie d'un logiciel.

Plusieurs modèles de cycle de vie ont été proposés pour fournir des traits généraux (Royce, 1970; Boehm, 1988). Ces modèles ont permis d'identifier les activités d'un

projet, et de décrire leurs interactions. Comme exemples de modèles génériques de cycle de vie, nous pouvons citer :

- le modèle en cascade, le modèle en spirale;
- les standards du Département de la Défense Américaine (DoD-STD-2167a);
- et les cycles de vie standards des logiciels (*"configuration management procedures"*, etc).

Ces modèles sont de haut niveau d'abstraction (vue macroscopique). La granularité du processus décrit est très large, les descriptions des cycles de vie sont, en général, centrées de façon abstraite sur les produits génie logiciel et ne precisent pas les différentes stratégies qui doivent être suivies pour gérer et coordonner des projets logiciels.

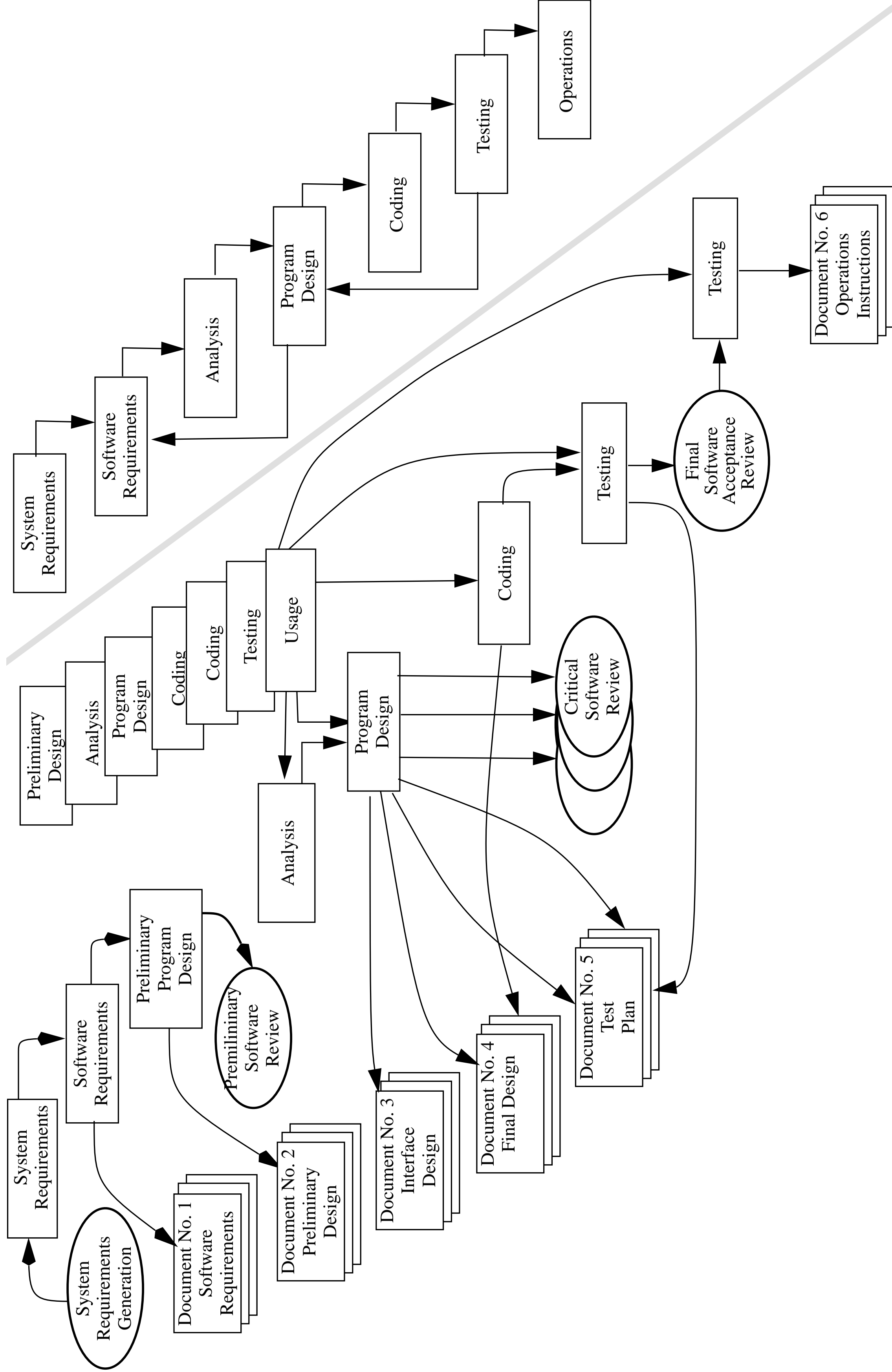
Dans la section suivante, nous présentons comme exemples de modèles de cycle de vie le modèle en cascade et le modèle en spirale. Nous montrons pourquoi ces modèles ne sont pas suffisants pour offrir un cadre d'automatisation des processus de développement. Nous avons choisi ces deux modèles parce qu'ils sont très représentatifs du génie logiciel. Le modèle en cascade est le plus utilisé par les entreprises concernées par la production de logiciels. Le modèle en spirale est plus récent. Il a servi de point de départ méthodologique pour des projets importants, dont Arcadia (Taylor and et al, 1988).

III.2 Les modèles génériques de cycle de vie

III.2.1 Le modèle en cascade

Le modèle en cascade a été proposé au début des années 70 (Royce, 1970). Ce modèle structure les activités de développement de gros logiciels en plusieurs étapes et décrit de manière explicite le flux de données entre ces différentes étapes, comme montre la figure III.1. Chaque fois qu'une étape se termine, des documents attestant le résultat de l'étape sont créés. Lorsqu'un problème est détecté à la fin d'une étape, il est possible de retourner à l'étape précédente.

Figure III.1 Le modèle cascade (Royce, 1970).



Les principaux défauts de ce modèle sont les suivants :

1. des projets logiciel de vraie grandeur suivent rarement le séquencement d'activités proposé par le modèle.
2. ce modèle impose à l'utilisateur de définir dès le début du processus logiciel ses besoins. Or, il est difficile de décrire complètement à l'avance chaque activité de processus logiciel lorsqu'on est dans un cadre de programmation globale.
3. le délai entre les spécifications et la production de la première version exécutable du logiciel est très long. En utilisant le modèle en cascade, l'utilisateur ne pourra vérifier qu'à la fin du processus si le logiciel produit correspond à ce qu'il a spécifié, et surtout à ce qu'il veut réellement.

En conclusion, lorsque le processus logiciel est géré comme une évolution d'activités qui se succèdent les unes après les autres, le processus résultant est trop rigide pour s'adapter aux changements. Un autre inconvénient de ce modèle est le manque de support pour retourner aux étapes précédentes. Toutes les décisions concernant le retour d'une étape à une autre et les changements qui doivent être effectués sont laissés à la responsabilité de l'utilisateur.

Malgré ces inconvénients, ce modèle est toujours considéré comme une référence importante, car il fournit un cadre où les procédures et les méthodes de développement et de maintenance peuvent être intégrées.

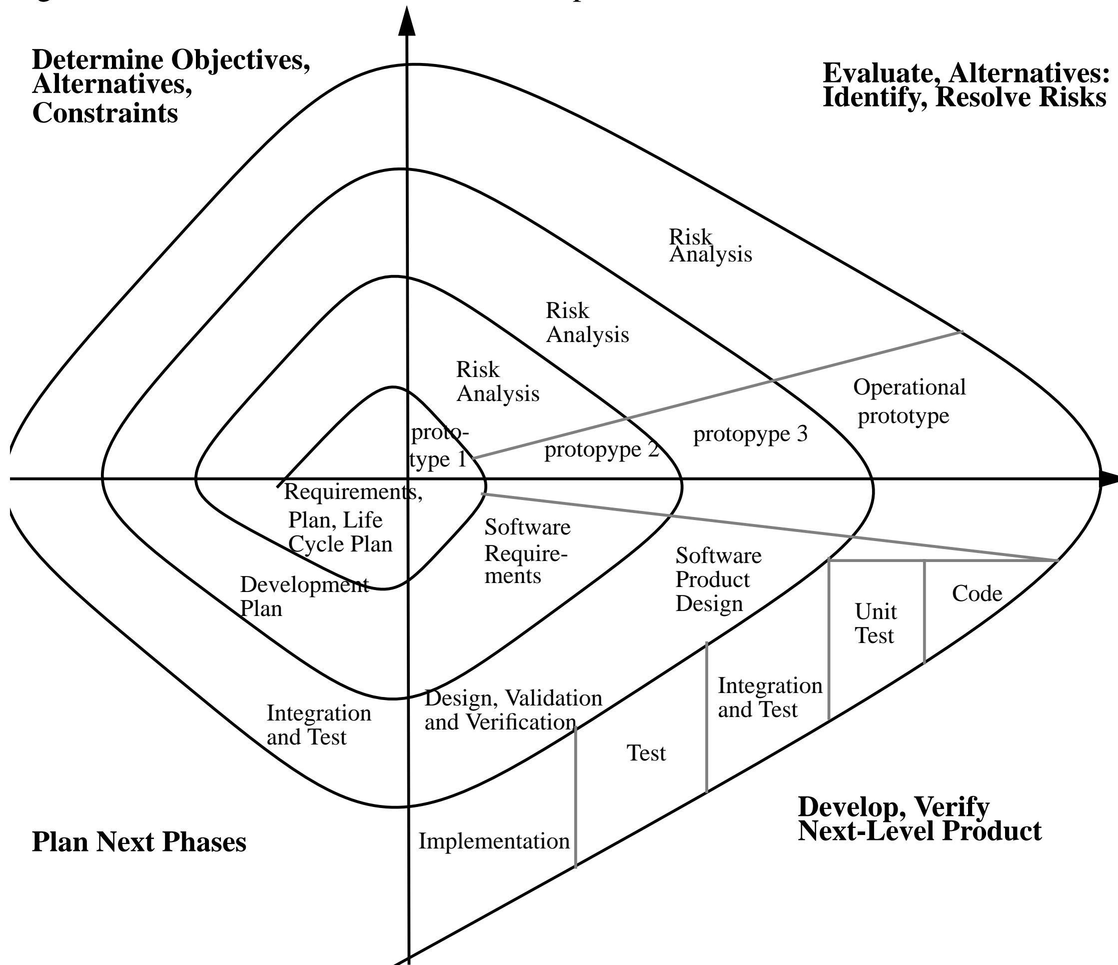
III.2.2 Le modèle en spirale

Le modèle en spirale a été développé pour remédier aux inconvénients du modèle en cascade (Boehm, 1988). L'idée principale de ce modèle est de fournir un cadre conceptuel permettant de se déplacer d'un modèle dirigé par le flux de documents, caractéristique du modèle cascade, vers un modèle dirigé par des *risques* (décisions critiques sur l'avenir du projet).

Dans le modèle en spirale, le processus de production d'un logiciel est modélisé par une séquence itérative d'étapes. A chaque itération, les risques sont analysés et résolus pour une partie spécifique du produit logiciel. L'identification des risques, l'élaboration de solutions et l'exposition des résultats deviennent les activités les plus critiques. Ce sont donc ces activités qui demandent le plus d'attention de la part des équipes de développement. L'évaluation des risques étant la partie la plus critique de ce modèle, la qualité du processus est le résultat de la capacité des responsables de développement à

identifier et à gérer les sources de risques dans un projet logiciel. La figure III.2 montre une exemple de ce modèle.

Figure III.2 Le modèle en Spirale. Boehm, 1988



Nous pouvons condidérer le modèle en spirale comme le meilleur cadre conceptuel pour le développement de logiciels complexes et de grande taille. Il permet en effet au développeur et au client de comprendre et de réagir à temps aux risques à chaque niveau d'évolution. Le modèle en spirale met également en valeur le prototypage comme un mécanisme de réduction des risques. A l'inverse d'autres approches, il permet aux responsables de développement d'appliquer l'approche de prototypage à n'importe quel point de l'évolution d'un produit. Le modèle en spirale reprend l'approche systématique d'évolution pas-à-pas, propre au modèle en cascade, et l'intégre dans un cadre interactif qui reflète de manière plus réaliste le monde réel.

III.2.3 Evaluation

Les modèles de cycle de vie, comme les modèles en spirale et en cascade, ont montre l'importance d'une définition explicite des processus logiciels. Les travaux de recherche

qui s'y réfèrent ont permis de mieux comprendre ce processus, et de mieux déterminer l'ordre des activités impliquées dans la production des gros logiciels, bien qu'elles soient décrites à un niveau d'abstraction assez élevé. Ces modèles ont permis d'une part d'améliorer la qualité des produits logiciels, d'augmenter l'efficacité des méthodes et des outils de développement, et d'autre part de réduire le coût de développement et de maintenance (Boehm, 1988).

Malgré ces avantages, les modèles en question présentent une forte limitation inhérente à leur généralité : ils ne précisent pas les détails du processus logiciel, qui sont pourtant d'une importance fondamentale pour une bonne conduite des projets logiciels. Ces modèles sont trop généraux et incomplets pour faire ressortir des informations importantes permettant de contrôler et d'automatiser les processus logiciels (Madhavji, 1991; Penedo, 1991; Sarkar, 1989) :

- les activités ou les étapes dans le processus de développement qui doivent être exécutées pour prendre en compte les problèmes mis en valeur par les clients;
- les conditions de terminaison et de déclenchement d'une activité, les contraintes appliquées à chaque activité, les états des composants logiciels avant, pendant et après l'exécution d'une activité;
- les outils utilisés pour supporter l'opération, la méthodologie à chaque étape du processus aussi bien que le choix des étapes séquentielles et parallèles dans un processus en exécution;
- les objets d'entrée et de sortie de chaque activité, ainsi que leur source et leur destination dans le processus logiciel aussi bien que la façon dont le flux de données se propage dans une activité, et d'une activité à une autre;
- les différents rôles joués par les différents utilisateurs aussi bien que le moyen par lequel la communication entre ces utilisateurs est supportée.

Pour prendre en compte toutes ces informations, lorsqu'une organisation est chargée de développer ou de maintenir un projet logiciel complexe, trois solutions sont en général adoptées:

1. les modèles de cycle de vie génériques sont redéfinis et/ou complétés de manière *ad hoc* pour prendre en compte les besoins de l'organisation;
2. les standards proposés par des agences gouvernementales sont choisis;
3. l'organisation crée et maintient son propre manuel de procédés avec des descriptions informelles, ou semi-formelles, qui spécifient le processus

logiciel qui doit être suivi par les équipes chargées du développement et de la maintenance.

Malheureusement, ces types de descriptions manquent de précision et souvent ne sont pas suivis! Dans la plupart des cas, ces descriptions ne correspondent pas aux processus logiciels réellement exécutés dans l'environnement. Cette déviation entre l'environnement courant et le processus pré-établi a différentes origines (Curtis et al., 1992) :

1. le haut niveau d'abstraction du processus. Le manuel des procédés de développement décrit le processus logiciel de manière prescriptible, c'est-à-dire qu'il fournit les activités qui doivent être exécutées pour produire un logiciel. S'il n'existe pas de mécanisme automatique pour imposer ces prescriptions, les activités menées au sein de l'environnement peuvent diverger de cette *recette*.
2. le manque de précision du manuel de procédés. En général les descriptions du manuel des procédés sont imprécises, ambiguës, et même parfois incompréhensibles, et par conséquent elles sont difficiles à utiliser.
3. le processus change plus vite que sa description. A cause de la rapidité de changement du processus logiciel, le manuel de procédés devient vite incohérent.

Au vu de ces problèmes, il n'est pas étonnant que le processus logiciel n'ait pas atteint sa maturité, même en utilisant ce genre de modèle. Il n'est pas étonnant non plus que les activités liées à la gestion, au contrôle et à l'amélioration des processus logiciels soient extrêmement coûteuses (Humphrey and Kellner, 1989).

Nous pouvons alors conclure qu'il est important, pour une entreprise qui développe ou maintient de projets logiciel de grande taille, d'avoir un modèle de processus logiciel le plus détaillé possible et d'adhérer aux politiques décrites par ce modèle. Par politique, nous englobons, entre autre, les règles qui gouvernent le processus et les procédés logiciels. Des moyens doivent être mis à la disposition de l'entreprise pour faire suivre ces politiques aux équipes de développement. Conformément à un modèle détaillé de processus logiciel, l'entreprise doit être capable de contrôler les activités qui se déroulent de manière parallèle dans l'environnement et de faciliter la communication et le flux de données entre ces activités.

Les langages de programmation orientés processus ont été justement proposés pour permettre à une enterprise de modéliser les procédés de production de logiciel. Les

activités logiciels, les règles gouvernant ces activités ainsi que la communication et la synchronization entre ces activites peuvent alors être spécifiées dans un modèle de processus logiciel de manière précise et non ambiguë. Un AGL centré processus permet d'automatizer certaines activités et d'aider les utilisateurs dans la réalization des activités manuelles.

III.3 Nouvelle génération d'AGL : l'intégration par les processus logiciels

Parallèlement à la recherche concernant la conception et l'amélioration des modèles de cycles de vie, de multiples Ateliers de Génie Logiciel (AGL) regroupant des outils nécessaires au développement du logiciel ont été conçus. En général, les AGL disponibles sur le marché sont dédiés à quelques phases spécifiques du cycle de vie. On peut citer entre autres l'outil Field (Reiss, 1990) pour la phase de réalisation ou l'outil Mecano (Girod, 1991) pour les phases de spécification et de conception.

Il est difficile de réaliser un AGL prenant en compte toutes les phases, car d'une part le coût de sa réalisation est très élevé, et d'autre part cela impose une méthode figée de développement. Des AGL qui ont été construit avec cette optique ont vu leur utilisation limitée à des entreprises où le procédés logiciels sont déjà bien définis. Par exemple, l'AGL HERMES a été construit essentiellement pour supporter les activités de développement pour les logiciel de la Navette Spatiale Européenne Ariane.

On peut distinguer fondamentalement trois approches correspondant à l'évolution historique des AGL prenant en compte explicitement les processus logiciels (Belkhatir et al., 1992).

III.3.1 L' approche boite à outils

Dans cette approche les processus logiciels ne sont pas supportés. Les AGL sont bâtis en regroupant plusieurs outils. Les résultats produits par chaque groupe d'outils sont alors transmis de manière *ad hoc* vers d'autres outils (Brown and MacDermid, 1992). *Unix programmer workbench* est le système représentatif de cette approche.

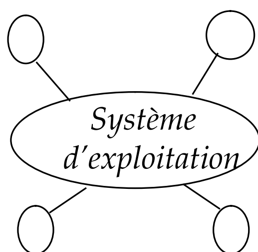
III.3.2 L'approche câblée

Dans l'approche câblée les outils et les processus sont indiscernables. L'AGL est bâti en privilégiant un ensemble d'étapes, par exemple la spécification et la réalisation, et en permettant que des nouveaux outils soient couplés dans l'AGL de manière *ad hoc*. Les étapes choisies par l'AGL sont assistées et tous les aspects spécifiques à ces étapes sont

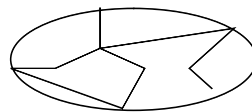
pris en compte par l’AGL. Cela est dû au fait que les outils sont fortement intégrés permettant ainsi que le transfert de résultats d’un outil vers un autre soit complètement contrôlé par l’AGL.

Le principal point faible de cette approche est justement le fait que le modèle de processus logiciel (les activités supportées, l’ordre d’exécution de ces activités, le flux d’informations entre ces outils, etc) est pré-défini par le constructeur de l’AGL. Ces AGL sont très difficilement adaptables à d’autres formes d’utilisation. Les utilisateurs doivent donc modifier leur manière de travailler pour s’adapter aux politiques pré-définies par l’AGL. L’AGL PACT (Thomas, 1989b; Simmonds, 1989) est représentatif de cette approche.

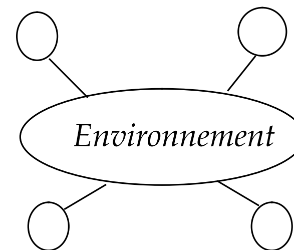
Figure III.3 L’évolution des AGL : vers l’intégration par le processus logiciel.



BOITE A OUTILS
Pas de support
de processus



CABLE
outils et processus
indiscernables



PROCESSUS EXPLICITE
Processus séparé des
outils

III.3.3 L’approche processus logiciels

Dans cette approche les politiques de développement et de maintenance des logiciels sont décrites en dehors des outils composant l’AGL. Les politiques sont décrites par un modèle. Les descriptions fournies via ce modèle permettent à l’AGL d’automatiser certaines activités, de guider les utilisateurs dans l’exécution d’autres activités, de contrôler l’utilisation des ressources de l’environnement, de supporter la communication entre les équipes, etc.

Un tel modèle doit donc être suffisamment flexible pour permettre la modélisation des processus logiciels à différents niveaux de granularité. Il doit permettre la décription et l’ordonnancement des étapes, ainsi que le flux de données entre les étapes. C’est le cas comme par exemple des AGL Softman (Mi and Scacchi, 1992) et Process Weaver (Fernstrom, 1993).

En outre, un tel modèle doit aussi permettre la spécification des contraintes sur la manipulation des objets et le flux d'information. Il doit également autoriser la description des attributs, des opérations et des outils utilisés dans les activités.

III.4 Les formalismes de description des processus logiciels

L'idée de représenter la connaissance sur le processus par un modèle détaillé a été originellement proposé dans (Lehman and Belady, 1985) et (Osterweil, 1987). Selon eux, le processus logiciel doit être décrit rigoureusement dans un formalisme exécutable (un langage de programmation orienté processus logiciel). Lorsque le processus logiciel est exprimé par un tel formalisme, il devient possible de l'imposer, de le mesurer (Basili and Rombach, 1988; Basili, 1992; Rombach and Verlage, 1993), de l'exécuter (Sutton et al., 1990), de le simuler (Mi and Scacchi, 1990) et de le ré-utiliser (Oivo and Basili, 1992), et de contrôler son évolution (Conradi et al., 1993).

Plusieurs projets de recherche ont été lancés récemment pour mieux étudier les aspects liés à la modélisation des processus logiciels. De nombreuses approches proposées tentent supporter les processus logiciels aussi automatiquement que possible. Nous exposons quelques approches et les AGL les supportant. Nous n'avons pas la prétention de couvrir ici toutes les techniques de modélisation de processus logiciel. Nous présentons seulement les concepts les plus importants issus des approches qui semblent les plus prometteuses.

III.5 L'approche procédurale

L'approche *procédurale* a été proposée dans (Osterweil, 1987). L'idée fondamentale est la spécification d'un modèle de processus logiciel sous la forme d'un programme. Ce programme décrit de manière détaillée comment le processus logiciel doit être réalisé. Par l'interprétation de ce langage le processus logiciel pourra être automatiquement exécuté, permettant ainsi de fournir une assistance aux différents usagers durant le déroulement de leurs activités. Si on arrive à décrire un processus logiciel par un programme, il devient possible d'appliquer les techniques et les outils standards de programmation aux processus permettant ainsi de le supporter et de le contrôler. Par exemple un programme processus pourra, par exemple, évoluer en versions et être géré par un gestionnaire de configurations, etc.

Deux catégories de systèmes peuvent être distinguées

1. les AGL qui ont adopté le langage Ada, par exemple les AGL Arcadia/Appl-A (Sutton et al., 1990) et Triad/CML (Sarkar and Venugopal, 1991a).
2. les AGL qui ont adopté un langage de programmation orienté-objet, par exemple les AGL OPM/Galois (Sugiyama and Horowitz, 1991) et Ipse/PML (Warboys, 1989a; Snowdon, 1989)

III.5.1 Arcadia/Appl-A

Le but d’Arcadia est de fournir une plateforme sur laquelle des AGL centré processus peuvent être bâtis (Taylor and et al, 1988). Le processus logiciel des AGL gérés doit être décrit par un programme et contrôlé par l’exécution de ce programme. Appl/A est le langage de programmation choisi par Arcadia pour programmer le processus. Appl/A est une extension du langage Ada étendu par :

1. des relations programmables et persistantes. La structuration d’un produit logiciel et les associations entre ces composants sont faites par ces relations. Exemple :

   Les relations suivantes décrivent respectivement la dérivation d’un module source vers un module objet et le comptage de mots dans un texte. La compilation est faite par l’opérateur `compile` décrit dans la relation `Source_to_Object`. Le comptage de mots est réalisé par l’opérateur `wc` dans la relation `Word_Count` :

```
with compile;with code_defs; use code_defs;
Relation Source_to_Object is
     type src_to_obj_tuple is tuple
          src : in source_code; % paramétre d'entree
          obj : out object_code; % paramétre de sortie
     end tuple;
entries
     entry insert(src: source_code);
dependencies
     determine obj by compile(src, obj);
end Source_to_Object

with WC; % outil qui permet de compter les mots
with text_def; use text_def;

Relation Word_Count is
     type wc_tuple is tuple
          text: in text_type;
          lines, words, characters: out integer;
     end tuple;
entries
     entry insert(text: in text_name);
```

```
dependencies
     determine lines, words, characters
          by wc(text, lines, words, characters);
end Word_Count
```

2. des déclencheurs (*“triggers”*) associés aux relations. Ces déclencheurs sont activés sur les modifications d’états des relations par des opération prédéfinies : opérations de création, de suppression et de modification.
   Exemple d’un déclencheur qui permet de mettre-à-jour le numero de lignes d’un code source toutes les fois que ce module est compilé.

```
with Source_to_Object, Word_Count;
with Code_Types; use Code_types;

trigger body Maintain_Source_WD is
begin
     loop
          upon Source_to_Object.insert
               (src: source_code)
          completion do
               Word_Count.insert(src);
          end upon;
     end loop
end Maintain_Source_WC;
```

3. des contraintes. Les contraintes sont spécifiées par des prédicats logiques associés aux relations et vérifiées chaque fois qu’une opération de modification est effectuée sur un élément d’une relation.
4. le concept d’atomicité, qui permet la spécification d’un groupe d’opérations sur une relation en termes d’une transaction courte.

• Expérimentation et résultats

Appl/A a été utilisé pour décrire l’étape d’analyses des besoins (Sutton et al., 1991). Cette expérience a démontré qu’Appl/A peut être utilisé pour automatiser quelques activités concernant les aspects de contrôle de communication des résultats entre les équipes de développement. Appl/A (Sutton et al., 1991) a surtout aidé les analystes à gérer les activités de modélisation des structures de données d’une organisation spécifique, lorsque ces activités sont menées de manière parallèle.

Cette expérience a démontré que des langages comme Appl/A sont trop rigides pour modéliser le processus logiciel. Par son aspect procédural, Appl/A exige que toutes les caractéristiques liées au processus soient connus à l’avance par l’administrateur du processus logiciel pour que celui-ci puisse l’exprimer en Appl/A. Or, cela est très

difficile et des mécanismes permettant de faire évoluer le modèle de processus logiciel sont indispensables. Ces mécanismes, nécessaires pour la prise en compte de ces évolutions, sont aujourd'hui absents aussi bien dans Appl/A que dans Arcadia (Sutton, 1993).

III.5.2 Triad/CML

Dans Triad (Ramanathan and Sarkar, 1988), le processus logiciel est décrit par un langage impératif nommé CML (*"Conceptual Modeling Language"*) (Sarkar, 1989).

Tout comme Appl/A, CML a également étendu le langage Ada pour prendre en compte le processus logiciel. Le langage CML fournit quatre sous-modèles, chacun étant utilisé pour décrire un aspect spécifique du processus logiciel. Les quatre sous-modèles sont :

1. Un sous-modèle de données orienté-objets, où les produits logiciels sont structurés par une hiérarchie de classes. Les associations entre les classes sont spécifiées par des attributs référence. Des règles événement-condition-action peuvent être définies pour prendre en compte les modifications effectuées sur les attributs référence et sur les attributs valeur.
   Exemple :

```
object module
     attributes name : string(20); %attribut valeur
          coded_by : person; % attribut référence
          LOC : integer, derived, init(0);
     subobject specs : module_specs % objet composite
     rules
          if (item == LOC and action = get) then
               pre: (LOC := m_body.LOC + specs.LOC);
end module;
```

2. Un sous-modèle spécialisé pour la description des outils qui composent l'environnement de développement;
3. Un sous-modèle pour la définition des rôles qui peuvent être joués par les informaticiens. Chaque classe de rôles définit les droits des utilisateurs de cette classe, c'est-à-dire quel genres d'opérations les usagers peuvent effectuer pendant le déroulement de leurs activités;

```
type design_leader is role
     can create_data {system_design, % type de l'activité
                 subsystem_design}% type de l'activité
     view of system_design is ...
          % informations qui peuvent être vues par
          % l'utilisateur
     can_invoke {...}
```

```
          % outils dont l'utilsateur peut se servir
     end design_leader;
```

4. Le sous modèle d'activités. Chaque activité passible d'automatisation est décrite par le sous-modèle d'activité comme étant une tâche. Une tâche dans le langage CML ressemble beaucoup à une tâche Ada. La principale différence porte sur le mécanisme d'exécution. Une tâche Ada est compilée vers un code binaire et exécutée par un système standard d'exécution. Une tâche CML est exécutée par Triad comme étant une transaction longue.

```
     activity design;
          state_variables sys_des_approved : boolean;
     subactivities
          sys_design : system_design;
          sys_review : system_design_review;
     precondition req_approved or redo_design;
     postcondition des_approved;
     action
          start_subactivities;
          checkin inputs
          ...
end design;
```

Triad permet l'exécution parallèle des tâches. Chaque tâche se déroule dans un espace privé qui lui est propre. Triad fournit également des primitives de communication basées sur l'échange de messages permettant la coordination et la synchronisation entre les tâches parallèles.

III.5.3 OPM/Galois

OPM (Sugiyama and Horowitz, 1991) permet de décrire un modèle spécifique de processus logiciel dans un style orienté-objet étendu avec des concepts propres à la modélisation de processus logiciel. Le formalisme utilisé pour décrire les processus logiciels est nommé Galois (Sugiyama and Horowitz, 1990). Un modèle de processus logiciel est décrit en Galois par un ensemble de classes. Les occurrences du processus sont gérées comme instances de ces classes (voir figure III.4).

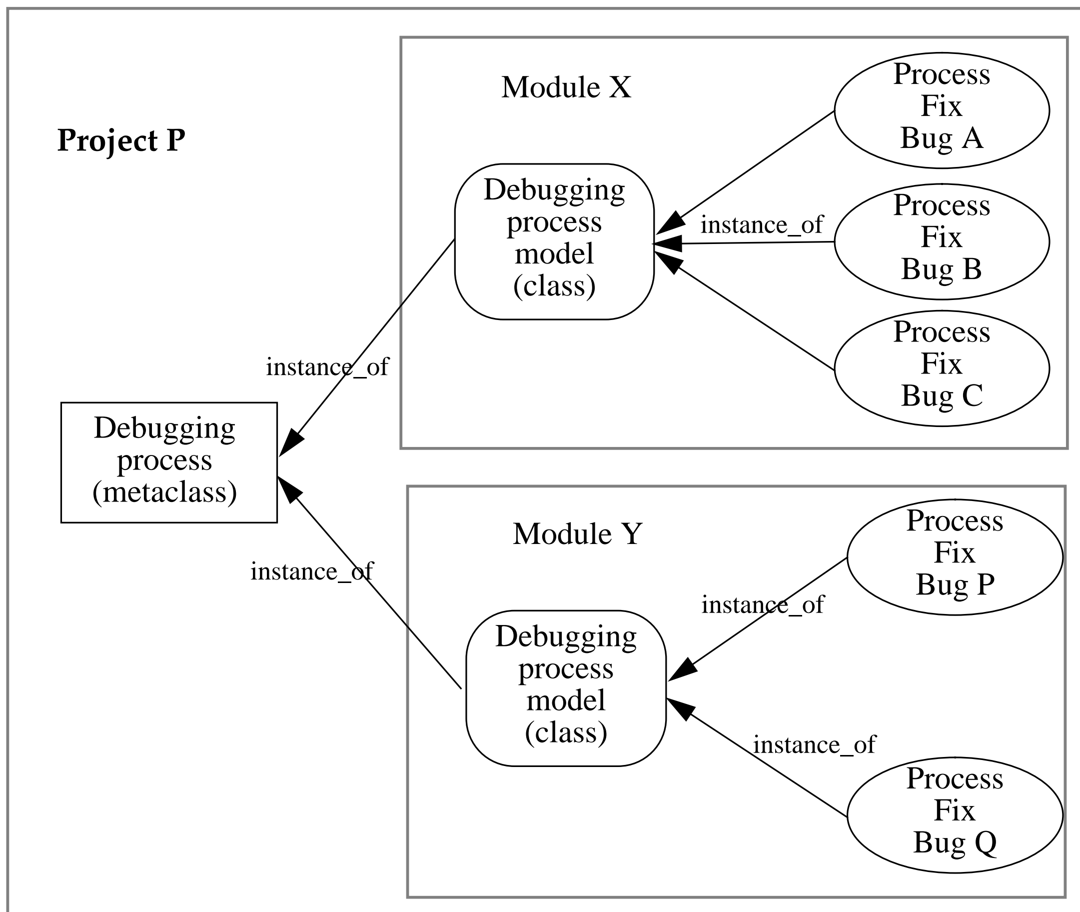
Galois a adopté le langage C++ et l'a étendu avec :

1. des meta-classes (classes manipulées comme des objets). Les méta-classes sont ajoutées pour permettre la coordination et la supervision des occurrences de processus logiciels.

2. des règles de production. Les règles ont été ajoutées afin d’encapsuler l’exécution des méthodes d’une classe par ajout de pré et de post-conditions aux méthodes de la classe.

3. le concept d’atomicité des opérations. Par ce concept, l’exécution d’un ensemble d’opérations est rendu atomique. Cela a été fait pour permettre la modélisation des politiques de partage de ressources entre les différentes occurrences de processus logiciel.

Figure III.4 Un exemple des processus de mise au point en OPM/Galois.



Chaque instance de processus logiciel est exécutée par OPM dans un environnement de travail propre à un utilisateur. Un utilisateur donné peut avoir plusieurs environnements de travail actifs. Pour chaque environnement de travail OPM fournit par une interface conviviale, les actions que l’utilisateur peut exécuter. Les contraintes et les conditions sur l’exécution des ces actions sont décrites dans un programme écrit en Galois par le concept de processus. Un modèle de processus logiciel dans Galois peut avoir plusieurs environnements de travail (instances de modèle) dans OPM. Comme décrit dans la figure III.4 le modèle de mise au point (“*debugging process model”*) a cinq instances. Trois de ces instances ont été créées pour la mise au point du module X et les deux

autres pour le module Y. Chaque instance est contrôlée par OPM comme étant un environnement de travail. OPM peut contrôler de manière globale tous les instances par le concept de *méta-classe*. Les politiques décrites dans les méta-classes sont générales pour les instances de processus et les règles décrites dans les méta-classes peuvent être utilisées pour contrôler le déroulement des activités exécutées pour tous les environnements de travail.

III.5.4 Ipse 2.5/PML

Dans Ipse (Warboys, 1989a; Snowdon, 1989) le processus de développement logiciel est décrit comme un ensemble d'activités concurrentes et distinctes correspondant à plusieurs *rôles*. Le langage de description de processus d'Ipse 2.5 est nommé PML.

Le langage PML permet de décrire les rôles et leurs interactions en utilisant l'approche orienté objet. Un modèle spécifique de processus logiciels est défini comme étant des sous-classes de cinq classes primitives de PML. Ces cinq classes forment le noyau de PML où chaque classe définit une facette du processus. Les cinq classes sont les suivantes :

1. La classe des *rôles*. Cette classe est utilisée pour modéliser un "fragment de processus logiciel" correspondant à un ensemble d'activités fortement correlées. Les rôles sont décrits en utilisant les concepts de classification/ instantiation et spécialisation/agrégation. Cette classe définit :
   a. les types d'entités qui peuvent être manipulés par un rôle, à l'aide d'attributs références,
   b. les rôles avec lesquels un rôle peut communiquer,
   c. les actions et les interactions qu'il possède ainsi que les pré et les post-conditions qui encapsulent ces actions et ces interactions.
2. La classe des *actions*. Dans cette classe nous décrivons les opérations manuelles ou automatiques qui peuvent être exécutées dans un rôle.
3. La classe des *interactions*. Dans cette classe nous décrivons les stratégies de communication entre les rôles. Le langage PML ne permet que la description d'interactions bi-directionelles et asynchrones. Cependant il n'est possible ni de décrire de politiques de partage d'objets comme dans Galois, ni de contrôler la coopération entre les rôles manipulant de manière parallèle des objets communs.
4. La classe des *entités* qui représente les ressources manipulées par les rôles.

5. La classe des *assertions*. Les assertions sont un ensemble d'expressions booléennes avec des paramètres. Elles peuvent être utilisées pour définir les conditions de déclenchement et d'arrêt des actions, l'ordonnancement des actions et interactions entre les rôles. Ces assertions peuvent déclencher des activités par l'activation de nouvelles instances de rôles.

L'exécution d'un processus est accomplie dans Ipse2.5 par l'interprétation des instances de ces classes. L'utilisateur interagit avec Ipse2.5 à travers l'autorisation d'actions manuelles. A l'aide d'assertions, Ipse2.5 décrit les conditions où ces interactions peuvent avoir lieu. Les actions automatiques sont déclenchées au fur et à mesure que ces pré-conditions sont satisfaites. Le réseau d'instances des rôles et leur interactions composent un modèle de processus logiciel spécifique.

Ipse 2.5 a évolué vers un produit commercial nommé PSS (Bruynooghe et al., 1991).

III.5.5 Synthèse

PML est un langage créé pour supporter la description des processus logiciels.
Galois au contraire a adopté le langage C++ et l'a étendu avec des concepts propres au processus logiciel.
Les concepts fournis par PML pour la description des activités sont de plus haut niveau que ceux fournis par Galois. Les relations entre les activités et les ressources manipulées par celle-ci ainsi que les politiques de communications entre activités peuvent être mieux décrites .

Galois fournit des pré et des post-conditions sur l'activation des méthodes. PML propose en plus des assertions qui peuvent être utilisées pour déclencher des activités et pour les arrêter automatiquement.

Triad/CML fournit un modèle permettant la description des utilisateurs, modèle absent en Appl/A.

Appl/A fournit un concept de déclenchement plus élaboré que celui-ci de Triad/CML. Les règles de déclenchement Triad/CML ne peuvent être décrites que sur les opération de mises à jour des attributs, tandis que dans Appl/A deux types de déclencheurs sont disponibles : les intra-relations et les inter-relations.

Les déclencheurs intra-relations sont utilisés pour décrire les actions à effectuer après la mise-à-jour d'une relation paticulière, comme par exemple l'opérateur **`compile`** défini dans la relation `Source_to_Object`

Les déclencheurs inter-relations, quand à eux, sont défini par un type de relation appelé TRIGGER. Les règles se rapportant à plusieurs relations peuvent être regroupées au sein d'un même TRIGGER. Cela permet le déclenchement d'une opération de modification d'une relation suite à la mise-à-jour d'une autre relation.

Par rapport à Appl/A et CML, Galois et PML présentent l'avantage de permettre la description des processus logiciels dans un style orienté objet. Cela rend plus aisé l'évolution des processus car nous pouvons utiliser les concepts de spécialisation via l'héritage multiple, concept propre au paradigme objet. Cependant, les politiques de communications entre activités peuvent être aisément décrites en Appl/A et CML car ces langages utilisent le concept de *rendez-vous* disponible dans le langage Ada.

CML, Appl/A et PML fournissent des règles (ou des assertions) permettant de prendre en compte automatiquement les mise-à-jour des objets ou de l'environnement via un mécanisme de déclenchement (*"trigger"*). Cela mène à penser, que même pour une approche *procédurale*, certains aspects des processus logiciels doivent être exprimés d'une façon déclarative. Dans la section suivante nous présentons les systèmes où la prise en compte des aspects déclaratifs des processus logiciels prédomine sur celle des aspects procéduraux.

III.6 L'approche déclarative

L'approche déclarative utilise des déclarations logiques pour décrire le processus logiciel. Ceux-ci sont décrits en termes de résultats attendus par l'utilisateur sans détailler la manière dont ces résultats sont obtenus de façon algorithmique (Williams, 1988).

Dans cette approche, les activités qui composent un modèle de processus logiciel sont décrites par des règles de production. Les outils supportant les activités de développement et de maintenance sont incorporés dans l'environnement en utilisant le concept *d'encapsulation.* Selon chaque type d'utilisation, un outil est enveloppé par une règle. Les pré-conditions de cette règle fournissent les conditions de déclenchement de l'outil, tandis que les post-conditions fournissent les effets provoqués par son exécution. Exemple :

pré-condition : si le module X a déjà été compilé
action : tester le module X
post-condition : si les tests se sont déroulés convenablement, alors modifier l'état du module X pour prendre en compte cette nouvelle information.

Selon le mécanisme d’exécution utilisé pour interpréter les règles, nous pouvons diviser les AGL représentatifs de cette approche en trois catégories :

1. les AGL qui utilisent la chaînage avant et/ou chaînage arrière, par exemple Marvel, Merlin et Alf ((Derniame et al., 1992));
2. les AGL qui utilisent un système de planning, par exemple Grapple et Epos.
3. les AGL qui utilisent un système de déclenchement, par exemple AP5, ELEN et Adèle.

III.6.1 Les systèmes basés sur le mécanisme de chaînage

III.6.1.1 Marvel

Parmi les AGL centrés processus et basés sur des règles de production, Marvel est sans doute le plus connu (Kaiser et al., 1988b). Une grande partie des idées proposées par ce système ont fortement influencé plusieurs autres AGL, comme ALF (Canals et al., 1993) et Peace (Arbaoui, 1993). Il est un des premiers AGL centrés-processus à aboutir à un prototype démontrable et diffusé. Pour ces raisons, nous allons étudier ce système d’une manière plus détaillée.

Le processus logiciel est décrit par des règles de production. Une règle en Marvel est composée d’une précondition, d’une action, et de deux types d’effet de bord.

- La précondition spécifie les conditions sur lesquelles la règle peut être exécutée. Elle est définie en termes d’une formule logique sur l’état de la base d’objets.
- Une action est un appel à un outil ou à une opération propre à Marvel qui met en oeuvre une activité dans le processus logiciel, par exemple, l’appel à un compilateur ou l’appel à une opération de suppression d’un objet stocké dans la base Marvel.
- Les effets de bord mettent à jour de nouveaux faits dans la base d’objets. Deux types d’effets de bord peuvent être spécifiés :
  - **a.** les modifications de l’état des objets qui doivent être faites si l’action est accomplie.
  - **b.** les modifications de l’état des objets qui doivent être faites si l’action échoue.

Exemple de définition :

```
compile [?f.Cfile]:
    #pré-conditions
```

```
        # obtenir les fichiers interfaces du fichier C
        # sous compilation
        (bind(?h to_all HFILE suchthat
             (linkto [?f.includes ?h])))
        :
        # verifier si le fichier C a été compilé aprés
        # sa dernière modification
        no_backward (?f.status != Compiled)
        # cette pré-conditon ne déclenche pas le mécanisme de chainage.

        # Activer l'outil charge de la compilation
        {Compile ?f.contents ?h.contents "-g"
             output: ?f.objet_code ?f.error_msg }

        #post-conditons
        (and (?f.status = Compiled)
             (?f.object_timestamp = CurrentTime__;
        (?f.status = Error);

    build [?p:Program]:
        # pré-conditions
        (and(forall Project ?proj suchthat (member
                                             [?proj.programs ?p]))
             (forall LIb ?I suchthat(member[?proj.libraries ?I]))
             (forall C_File ?c suchthat (member[?p.c_files ?c])))
        :
        (and (?c.analyze_status = Analyzed)
             (?c.compile_status = Compiled)
             (?I.archive_status = Archived))

        #Voila l'activité de cette règle
        {Build build_program ?p}

        #post-conditions
        (?p.build_status = Built);
             #effet déclenché en cas de succès
        (?p.build_status = Not_built);
             #effet déclenché en cas d'échec
```

Cette dernière règle exprime que :

1. si toutes les bibliothèques sont stockées correctement,
2. et si tous les fichiers C composant le programme sont compilés et analysés
3. alors le programme **p** peut être généré par l'application de l'outil "`Build`".
4. En fonction du résultat obtenu par l'application de cet outil, l'état du programme `p` est modifié par "`Build`" ou "`Not_built`"

L'exécution d'un processus logiciel en Marvel se fait via le mécanisme de chaînage avant et arrière sur la base de règles (Kaiser et al., 1988b). 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. La chaînage arrière s'arrête lorsque la précondition de la règle demandée par l'utilisateur est satisfaite. Après l'exécution de chaque règle, les post-conditions peuvent éventuellement changer l'état de la base d'objets. A l'issue de l'exécution de la règle qui a été demandée par l'utilisateur, le mécanisme de chaînage avant est déclenché. Ce mécanisme correspond au mécanisme standard de chaînage utilisé par les systèmes de règles de production, par exemple OPS5 (Brownston et al., 1985).

III.6.1.2 Merlin

Merlin (Peuschel et al., 1992) est un système expérimental fortement inspiré de Marvel. Merlin a éclaté les règles de production de Marvel en deux groupes pour mieux décrire le processus logiciel (Emmerich et al., 1991). Un groupe de règles est utilisé pour modéliser la partie déclarative du processus. Ces règles sont exécutées par le mécanisme de chaînage arrière. Le deuxième groupe est utilisé pour modéliser la connaissance procédurale du processus. Ces règles décrivent l'ordonnancement de l'activation des outils et les assertions des postconditions. Les règles de ce dernier groupe sont exécutées par le mécanisme de chaînage avant.

Exemple des règles utilisées dans le processus de chaînage arrière.

```
desing(Name) :-   display_menu(design, Item),
                  do_item(Name, Item).
test(Name) :-     display_menu(test,Item),
                  do_item(Name,Item).
do_item(Name, print) :-   is_tester(Proj,Module,Name),
                          do_print(Proj, Module),
                          test(Name).
```

Exemple des règles utilisées dans le processus de chaînage avant.

```
IF exist_test_environment(Proj,Module,incomplete)
THEN     CALL(editor,Proj,Module,Status),
         INSERT(exist_test_environment(Proj,Module,Status)).

IF exist_test_data(Proj,Module,incomplete)
AND exist_test_environment(Proj,Module,complete)
THEN     CALL(editor,Proj,Module,Status),
         INSERT (exist_test_data(Proj,Module,Status)).

IF exist_test_data(Proj,Module,complete)
THEN     CALL(test_tool,Proj,Module,Status),
         INSERT(tested(Proj,Module,Status)).
```

Ces trois règles expriment que pour tester un module les étapes suivantes doivent être suivies :

1. créer l'environnement de test,
2. créer les jeux des tests,
3. et enfin, tester le module.

III.6.1.3 Synthèse

La raison principale qui a poussé les concepteurs de Merlin à éclater les règles en deux groupes est d'assurer un meilleur contrôle du mécanisme de chaînage. Les auteurs de Marvel ont constaté expérimentalement que l'application automatique du mécanisme de chaînage avant chaque fois qu'une règle est exécutée peut éventuellement rendre la base d'objets incohérente. En éclatant les règles en deux groupes, et en appliquant un mécanisme d'exécution différent pour chaque groupe, les auteurs de Merlin ont amélioré le contrôle du mécanisme d'exécution. Les résultats obtenus pour Merlin ont poussé les auteurs de Marvel à ajouter dans les règles une clause spéciale dans la précondition et dans la post-condition, indiquant sur quelles conditions la règle peut être prise en compte par le mécanisme de chaînage (Barghouti and Kaiser, 1991b).

III.6.2 Les AGL basés sur les techniques de planning

L'idée directrice de cette catégorie d'AGL est qu'un modèle de processus logiciel de vraie grandeur est difficile à contrôler car toutes les éventualités liées à l'exécution du modèle doivent être prises en compte par l'administrateur du processus logiciel. Ce problème peut être traité en utilisant une technique dite de *planning* issue du domaine de l'intelligence artificielle.

Dans un système de planning, l'exécution du processus logiciel est basée sur deux étapes cycliques :

1. La sélection des opérateurs sur les objets est dirigée par le but. Le réseau résultant du processus de sélection est nommé le *plan d'exécution.*
2. Après qu'un plan d'exécution est généré, il est interprété par le système.

Donc, l'exécution du processus logiciel se fait récursivement de la manière suivante : la génération du plan d'exécution puis l'interprétation de celui-ci. Si l'état des objets manipulés est changé, l'utilisateur peut demander de refaire le plan pour prendre en compte cette mise-à-jour. Comme il est difficile de planifier toutes les étapes d'un processus logiciel, le plan d'exécution est fait de manière hierarchique. C'est-à-dire que

lorsqu'une activité de haut niveau, dans le plan d'exécution, doit être exécutée le planner est ré-active pour construire un plan d'exécution détaile pour cette activité. Pour cette raison nous disons que le processus de plannification et d'exécution est récursive.

Grapple (Huff and Lesser, 1988; Huff, 1989) et Epos (Conradi et al., 1990; Conradi et al., 1991b) sont deux systèmes caractéristiques de cette approche.

III.6.2.1 Grapple

La modélisation d'un processus logiciel en Grapple est réalisée par un ensemble de buts, sous-buts, préconditions, contraintes et conséquences. Ces éléments sont agrégés par deux composants fondamentaux :

1. l'ensemble des *fragments de processus*;
2. un ensemble de contraintes qui décrivent comment ces fragments peuvent être sélectionnés, ordonnés et appliqués.

Le concept d'*opérateur* est fourni pour représenter les activités passibles d'être exécutées durant le processus logiciel. Un opérateur est similaire à une règle de production. Un opérateur peut pontentiellement être sélectionné pour s'exécuter si sa pré-condition est vérifiée, et comme dans Marvel et Merlin, son exécution met à jour de nouveaux faits décrits par ses postconditions.
L'exemple suivante montre comment ces éléments sont intégrés via le concept d'opérateur en Grapple.

```
operator:         release,
goal:             current-release (System),
preconditions:    built(System),
                  regression-tested(System),
                  performance-tested(System),
constraints:      current-release(C),
                  not-equal(C, System),
effects:          DELETE current-release(C),
                  ADD current-release(System),
                  ADD customer-release(System),
                  ADD prior-release(System, C).
```

L'opérateur nommé `release` a pour but de construire une version livrable d'un logiciel nommé `System`. Pour cela, le logiciel `System` doit être généré, et passé par les tests de régression et de performance avant l'application de cet opérateur. Ces sous-étapes sont exprimées par les préconditions. Cet opérateur impose par les contraintes que la nouvelle version de `System` soit différente de la version actuelle.

Les préconditions doivent être les buts d'autres opérateurs ou des états sur les objets manipulés par Grapple. En se basant sur ces préconditions, Grapple génère le plan d'exécution, qui est interprété durant l'exécution du processus logiciel.

III.6.2.2 Epos

Epos est un noyau expérimental d'AGL dirigé par le processus logiciel. Il fournit des services intégrés de gestion de configurations et de modélisation de processus logiciels. L'élément central d'Epos est une base de données où les informations statiques (objets) et dynamiques (les instances de processus) sont stockées. Cette base est dirigée par un modèle de données voisin du modèle entité-association étendu avec les concepts d'héritage multiple et de versions. Cela permet la structuration des produits logiciels par types d'objets et de relations. Le modèle de processus logiciel est également décrit par ce modèle par des types spéciaux : les types tâche.

Les tâches d'Epos ressemblent aux règles de production de Marvel ou aux opérateurs de Grapple. Chaque tâche est composée d'un ensemble de pré et de postconditions, un ensemble de sous-buts (décomposition), et d'une partie action. L'action a la même fonction qu'une activité dans Marvel, c'est-à-dire une enveloppe d'appel à un outil. De même que, Epos fournit un mécanisme d'interprétation des tâches, nommé *Planner* (Liu, 1990).

Le *Planner* génère un plan d'exécution basé sur la description donnée par les tâches et par l'état des objets de l'environnement. Le plan d'exécution est interprété par un gestionnaire d'activités. Au fur et à mesure que le plan d'exécution est interprété, le *Planner* peut être ré-activé pour construire un plan d'exécution détaillé pour les tâches de haut niveau. Une tâche est considérée comme étant de haut niveau lorsqu'elle est composée d'un ensemble de sous tâches. Ce processus cyclique s'exécute jusqu'à ce que le but de la tâche racine soit accompli.

Example :

```
ENTITY Deriver IS TaskEntity {
    ATTRIBUTES
         DeriverSwitches : CONST String = '-g -w';

    PRECONDITION
         % Verification des dates des mises-à-jour des
         %parametres d'entre et de sorti
         MAX(SET(i:GenInputs(SELF) ! i.DateTime)) >
         MIN(SET(o:GenOutputs(SELF) ! o.DateTime))
    FORMALS
         [[input = cprograms], [output=binaray]]
    CODE
```

```
            CALL C-Compiler;
        POSTCONDITION
            'Success' OR 'Failure'
    };
```

III.6.2.3 Synthèse

La principale innovation d'Epos par rapport à Grapple se trouve au niveau des transactions. En Grapple, le processus de génération du plan d'exécution et l'exécution elle-même se fait en mode mono-utilisateur et sans le support d'une base de données — toutes les informations manipulées sont mises à disposition de Grapple par une base de faits Prolog.

En Epos, ce processus se fait en mode multi-utilisateurs. Un plan d'exécution est associé à chaque utilisateur. Ce plan est interprété sur les états des objets de la base de données locale propre à cet utilisateur. Epos fournit une base de données publique, pour tous les utilisateurs, et une base de données privée pour chaque utilisateur.

Le gestionnaire d'activités d'Epos est le mécanisme responsable de gérer la cohérence entre les différentes bases locales en propageant les effets de l'exécution des tâches entre les processus actifs via l'envoi des messages..

III.6.3 Les AGL basés sur le mécanisme de déclencheurs

Dans cette catégorie les processus logiciels sont modélisés par la description de contraintes sur la manipulation des objets. Les contraintes sont décrites par des règles événement-condition-action (ECA) qui se déclenchent lorsque les objets stockés dans la base de données sont modifiés, tel que :

1. un événement est déclenché lorsque l'état de la base de données est soumis à un changement.
2. la condition agit comme un filtre sur les événements. Une condition est exprimée en utilisant la logique du premier ordre étendu avec des constructeurs de la logique temporelle.
3. l'action décrit les opérations qui doivent être exécutées si la condition est satisfaite. Une action peut modifier ou refuser l'opération qui a déclenché la règle.

AP5, Elen et Darwin sont répresentatifs de cette approche.

III.6.3.1 Elen

Elen (Jarwah, 1992) est un prototype de recherche développé au Laboratoire de Génie Logiciel de Grenoble au sein de l'équipe Aristote. Le but de système Elen est d'aider les activités de développement et de maintenance de logiciels.

Les activités de génie logiciels sont décrites par des méthodes associées aux types d'objets. Les règles ECA sont utilisés pour contrôler l'exécution de ces méthodes.

III.6.3.2 AP5

AP5 (Goldman and Narayanaswamy, 1992) est un langage conçu pour aider les activités de prototypage plutôt que pour la modélisation de processus logiciel (Narayanaswamy, 1993).

AP5 utilise le modèle entité-relation pour décrire les informations statiques dans l'environnement. Les entités manipulées durant l'exécution du processus logiciel sont modélisées par des types d'objets. Les relations sémantiques entre les objets sont décrites par des types de relations. AP5 a adopté le langage LISP en l'étendant avec le concept de relation persistante. Il permet aux programmes utiliser une base de données dans laquelle des structures de données complexes sont stockées et manipulées de manière transparente.

Les informations dynamiques sont décrites par des contraintes sur les objets et sur les relations. De cette façon AP5 permet que des contraintes sur la manipulation de données propre à chaque organisation puissent être spécifiées. AP5 permet aussi d'automatiser certaines activités par l'utilisation des règles ECA. Les règles ECA sont toujours liées au mécanisme de transactions. Les événements sont traités après que la transaction soit terminée mais avant qu'elle soit validée.

Les fragments de processus logiciel sont également décrit selon style Lisp. Par exemple, l'activité de compilation peut être décrite en AP5 de la manière suivante :

```
(defsetp Compile (:automated nil)
     (let ((result (arbitrary compile-result)))
           (atomic (setf (compile-result-for self) result)
           (unless (eql result :error)
                (setf (object-code-produced self)
     (create object-code))))))
```

III.6.3.3 Darwin

Pour modéliser un processus logiciel, Darwin introduit le concept de loi (*law*) (Minsky, 1991). Une loi est un ensemble de règles Prolog (Bratko, 1990). Ces règles décrivent :

1. les opérations disponibles dans l'environnement,
2. les possibilités d'évolution de celui-ci,
3. les opérations nécessaires pour prendre en compte l'évolution des règles.

L'exécution de processus est contrôlé par l'activation et la gestion des lois, donc des règles Prolog.

L'originalité de Darwin est dans le contrôle du mécanisme d'envoi des messages dans l'environnement. Les objets logiciels interagissent par l'échange de messages où les lois sont chargées de contrôler ces échanges. Chaque fois qu'un message est envoyé, Darwin intervient dans le processus de communication et exécute les lois appropriées. En utilisant les descriptions disponibles dans les lois, Darwin peut :

1. changer les messages;
2. modifier la destination des messages;
3. supprimer des messages.

III.6.3.4 Synthèse

AP5 et Elen sont très similaires dans le principe : utilisation intensive des règles ECA pour modéliser les processus logiciels. L'originalité d'Elen réside dans le fait de considérer les méthodes comme partie intégrant des règles ECA. Pour chaque méthode nous pouvons définir des pre- et des post-conditions pour encapsuler son exécution. Les méthodes peuvent être automatiquement déclenchées si les pré-conditions sont valables.

Contrairement à Elen, Darwin maintient la cohérence de la base de règles lors de modification de cette. Les opérations de mise à jour de la base de règles permettent la création de nouvelles lois ou la suppression de lois déjà existantes, mais la modification des lois déjà existantes n'est pas supporté. Cette facilité a été également introduite dans AP5 (Narayanaswamy, 1993).

Darwin utilise les concepts de méthodes et d'envois de messages dérivés de la technologie orienté-objet. Contrairement Elen et à AP5, les concepts d'héritage et délégation, ne sont pas présent explicitement dans Darwin.

III.7 L'approche transformationnelle

Le processus de génie logiciel se caractérise par une transformation constante des objets. La spécification des besoins est transformée en une conception de haut niveau, qui elle

même est transformée en une conception détaillée, etc, jusqu'à créer un produit logiciel exécutable. Pour prendre en compte les transformations successives d'objets, l'approche transformationnelle a été proposée (Katayama, 1989). Dans cette approche, un modèle de processus logiciel est décrit comme étant une collection d'activités hiérarchiquement organisée. Chaque activité est caractérisée par ses paramètres d'entrée et de sortie. Ces paramètres sont des objets de génie logiciel, c'est-à-dire des fichiers, des programmes, des objets dérivés, etc. Chaque activité est chargée de transformer les paramètres d'entrée en paramètres de sortie, par l'application d'une fonction.

Par exemple, l'activité de compilation est modélisée, par cette approche, comme étant une fonction qui reçoit en entrée le code source et produit en sortie le code binaire correspondant. Le système HFSP (Katayama, 1989) est à la base de cette approche.

- **HFSP**

Dans HFSP (Katayama, 1989), le processus logiciel est décrit par une structure arborescente, permettant de décomposer les activités complexes en activités plus simples, où :

1. un noeud correspond à une activité. Les objets logiciels consommés et produits par les activités sont représentés par des attributs associés à ces activités
2. les arcs représentent la structure hiérarchique des activités.
3. les feuilles correspondent aux appels d'outils, comme par exemple, la compilation d'un module.

L'exécution du processus correspond au contrôle de la croissance de l'arbre d'activités. Au fur et à mesure que les activités de haut niveau donnent naissance à des activités de niveau inférieur, l'arbre d'activités grandit pour prendre en compte ces nouvelles activités avec leurs paramètres. .

Pour contrôler la création, la terminaison, la suspension temporaire et la communication entre les activités, HFSP fournit une ensemble d'opérations qui peuvent être appelées par l'utilisateur de HFSP ou directement par une activité (Suzuki and Katayama, 1991). Ces opérations peuvent aussi être utilisées pour changer dynamiquement le modèle de processus logiciel (Suzuki et al., 1993).

Exemple :

```
type
    spec = sequence_of word
    formalSpec = sequence_of spectoken
```

```
analyzedSpec = sequence_of spectoken
prog = sequence_of token
oprs = sequence_of operation
keywords = sequence_of token
nodeType = {'seq, 'select, 'iter }
dataTree, progTree = ['componentName : string,
                      'componentType : nodeType,
                      'subStructure : sequence_of progTree ]
     . . .


activity
    SJP( spec | prog.out ) -> MakeProgTree( spec | progTree)
                              EnumerateOprs(spec | oprs)
                              MakeProg(oprs, progTree | prog.mid)
                              DoProgInversion(prog.mid | prog.out)
    MakeProgTree(spec | progTree) -> AnalyzeSpec(spec|formalSpec)
                  MakeInputTree(formalSpec|dataTree.in)
                  ComposeTree(dataTree.in, dataTree.out|progTree)
    ExtractKeywords(spec|keywordsd) -> %appelle outil Lex
```

• **Synthèse**

Cette approche demeure limitée au système HFSP. Elle fournit les avantages suivants :

1. de combiner l’approche procédurale avec l’approche déclarative. Les activités sont définies dans un style déclaratif où l’entête de chaque activité exprime la transformation d’un ensemble d’objets vers un autre ensemble par une fonction. Le contrôle d’exécution des activités est fait par des opérateurs procéduraux (*while-do, repeat*, etc) permettant le déroulement concurrent des activités.
2. d’exprimer la relation entre le modèle de produit et le modèle d’activités. Les objets logiciels sont décrits par des types qui sont utilisés comme paramètres d’entrée et de sortie des activités
3. d’avoir expérimentée la mise-à-jour dynamique des modèle d’activités.

III.8 L’évolution de la description de processus logiciel

La recherche dans le domaine des processus logiciels est récente, d’où le peu d’expérience acquise dans le domaine d’évolution des modèles de processus (Conradi et al., 1993). Lors de récents congrès (6ème et 7ème Congrès Internationaux sur les Processus Logiciels) plusieurs sessions ont été dédiées à ce thème. Parmi les travaux récents, nous pouvons citer les systèmes Prism, AP5 et Marvel.

Prism (Madhavji and Schafer, 1991) permet de gérer l'exécution des processus logiciels et d'assister les activités concernées par l'évolution des modèles de processus. 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 : lorsqu'un problème est détecté ou une mise à jour est demandée durant l'exécution du processus logiciel, Prism permet à ce dernier de retourner vers la phase de simulation. Durant cette phase, le modèle de processus est mis à jour, et ensuite validé par des simulations avant d'être remis en exécution. Ces changements de phases sont contrôlés par une méthodologie propre à Prism (Madhavji, 1992).

Dans AP5 (Narayanaswamy, 1993) l'évolution du modèle de processus logiciel est faite par des commandes d'ajout, de suppression ou modification des règles de déclenchement. Comme AP5 a été bâti sur le langage LISP, l'administrateur de l'environnement peut changer le modèle de processus sans avoir besoin d'arrêter l'exécution des processus logiciels. La gestion de cohérence entre les modifications effectuées et les processus en cours d'exécution est reléguée aux personnes responsables de ces modifications.

Marvel (Kaiser and Ben-Shaul, 1993) permet l'évolution de la description du processus par le changement des pré- et des post-conditions encapsulant les règles. Marvel 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). Au contraire d'AP5, l'évolution de la description du processus est faite de manière statique; les processus doivent être arrêtés pour permettre la modification de leurs descriptions. Une fois modifiée, la nouvelle description est compilée et validée. Ce n'est qu'après ces étapes que les processus peuvent continuer leur exécution.

III.9 Un cadre conceptuel pour supporter les transactions

Le processus de développement de logiciels exige la coopération de plusieurs utilisateurs travaillant ensemble. Dans ce contexte chaque utilisateur doit coopérer pour que les résultats intermédiaires de ses activités puissent collaborer à l'achèvement d'activités ménées par d'autres utilisateurs. Comme chaque activité correspond à une transaction, les transactions concurrentes doivent être capables de propager leurs résultats au fur et à mesure que ces résultas sont produits.

Dans le cadre du génie logiciel, les transactions ont une longue durée, par conséquent les mécanismes de fermeture doivent permettre un grand degré de concurrence (Barghouti

and Kaiser, 1991a). De cette manière, les transactions ne doivent pas attendre systèmatiquement la fin d'autres transactions.

Les transactions sont créées pour atteindre un but, et un but complexe peut se décomposer en un arbre de sous-but, où chaque sous-but correspond à une transaction (Perry and Kaiser, 1987; Kaiser, 1990). Dans ce cas, nous nous trouvons dans un cadre où une transaction est composée de plusieurs autres transactions imbriquées (Barghouti and Kaiser, 1991a). Donc, un AGL centré-processus doit fournir un cadre de travail qui supporte les transactions courtes, les transactions longues ainsi que les transactions imbriquées.

• **Expériences**

Marvel (Barghouti and Kaiser, 1990) fournit un cadre de travail où les transactions longues sont supportées. Basé sur une architecture client/serveur, à chaque fois qu'un client demande une mise à jour des objets, une transaction est ouverte et le contrôle est passé au serveur qui garanti l'achèvement de la transaction. Ce contrôle est accompli de manière intégrée avec le mécanisme d'interprétation des règles de production. Marvel a implémenté une politique d'ordonnancement de transactions permettant l'exécution concurrente de plusieurs règles (Barghouti, 1992).

Triad (Sarkar and Venugopal, 1991b) supporte les transactions longues par le concept d'espace de travail. Les objets ainsi que les outils sont importés de la base centrale dans l'espace de travail où les activités sont exécutées. A la fin de la transaction, les modifications faites dans l'espace de travail sont exportées vers la base centrale.

Epos (Malm and Conradi, 1991) utilise un schéma similaire à celui de Triad pour supporter les transactions longues. Epos-DB supporte également les transactions courtes en utilisant des politiques similaires à celles introduites par les bases de données classiques. Les transactions longues sont supportées par des *check-in/check-out*, c'est-à-dire qu'une transaction longue débute lorsque les objets sont importés de la base centrale vers l'espace de travail, et finit lorsque ces objets sont retournés ver la base. OPM s'appuie aussi sur le modèle *check-in/check-out* pour supporter les transactions longues.

(Godart, 1993b) a adopté le modèle *check-in/check-out* et l'étendu avec trois niveaux :

1. le premier niveau est représenté par la base de données publiques où les objets globalement stables sont stockés.

2. le deuxième niveau est représenté par un base de données semi-publique où les résultats intermédiaires produits par les espaces de travail privés sont stockés;
3. le troisième niveau est représenté par les espaces de travail privés où les objets sont mis-à-jour.

III.10 La programmation coopérative

La programmation coopérative est l'ensemble des activités liées à la multiplicité des agents impliqués dans un projet logiciel. Un des plus importants besoins dans un AGL centré-processus est le cadre de travail qu'il doit fournir pour supporter la programmation coopérative. Un AGL centré-processus doit gérer la communication entre les utilisateurs afin que ces utilisateurs soient avertis correctement de l'état d'avancement du processus résultant des activités menées en parallèle dans l'environnement. Un manque de coordination entraîne une réduction du parallélisme, et par la suite une mauvaise utilisation des ressources et du temps.

Pour le faire, un AGL centré-processus doit fournir aux utilisateurs un protocole de communication pour stimuler l'échange d'informations, améliorer la synchronisation, intensifier la concurrence, permettre la notification à la suite des modifications effectuées sur les données et sur l'état des processus. Plusieurs types de protocoles de communication ont été proposés, et spécialement dans les AGL étudiés dans ce chapitre nous pouvons distinguer les modes de communications suivants : communication sur le réservation des ressources, communication sur les modifications de ressources, communication sur un conflit et négociation. Les systèmes qui fournissent des mécanismes pour supporter les transactions longues, comme ceux qui nous avons analysés dans la section III.9, peuvent également supporter la programmation coopérative.

III.10.1 Communication sur la réservation des ressources

Dans ce mode de communication, le système est chargé d'informer un utilisateur lorsqu'une ressource nécessaire pour l'achèvement de sa tâche est bloquée par un autre utilisateur. Cela lui permet de mieux repartir ses activités en se consacrant à celles ne dépendant pas de cette ressource.

Parmi les AGL étudiés, ce protocole de communication est le plus utilisé. Ceci est dû à sa simplicité et à la facilité de bâtir un mécanisme le supportant. Marvel , OPM et Alf ont montré, ou explicitement mentionné avoir implémenté cette politique.

III.10.2 Communication sur les modifications d'objets

Dans ce mode de communication l'AGL doit avertir les utilisateurs qui partagent ou sont en voie de partager un objet sur les mises-à-jour effectuées sur cet objet.

Pour supporter ce protocole de communication l'AGL doit disposer d'une part, de mécanismes permettant de surveiller les événements provoqués à la suite de changements d'objets, et d'autre part d'avertir les utilisateurs sur ces événements. Deux types de mécanismes ont été expérimentés par les AGL, à savoir :

1. Le déclencheur. Ce mécanisme, en général construit sur une base d'objets, permet d'associer des actions aux changements d'état des objets stockés dans cette base (McCarthy and Dayal, 1989). Adèle, AP5, Arcadia (Sutton et al., 1990), Alf (Oquendo et al., 1990), Yeast (Rosenblum and Krishnamurthy, 1991) sont des exemples de systèmes utilisant ce mécanisme pour supporter ce type de protocole de communication. En utilisant ce mécanisme, les actions associées au déclenchement peuvent altérer la modification requise ou l'invalider.
2. Le serveur de messages. Ce mécanisme permet l'échange de messages entre les outils de l'AGL (Cagan, 1990; Reiss, 1990). Certains systèmes, e.g. Triad, Process Weaver (Fernstrom and Ohlsson, 1991), Epos (Malm and Conradi, 1991) et OPM (Sugiyama, 1993), implémentent un protocole basé sur ce mécanisme de communication. Pour cela, chaque fois que l'état de l'environnement est mis-à-jour par l'application d'un outil, un message est envoyé à tous les outils pour les informer de cette modification. Ceux-ci peuvent alors prendre en compte ce message et exécuter les actions convenables, comme par exemple avertir les utilisateurs, déclencher de opérations permettant de modifier les objets, etc.

A notre avis, ces deux mécanismes sont complementaire un par rapport à l'autre. Ils peuvent être intégrés dans AGL centré-processus de la façon suivante :

1. le mécanisme de déclencheur est consacré à la gestion des événements provoqués dans la base.
2. le serveur de messages pourra être averti par le déclencheur, qui ensuite se chargera d'avertir les outils qui composent l'environnement.

III.10.3 Communication sur un conflit

Lorsqu'un conflit est provoqué par les utilisateurs intervenant dans un processus de coopération, ces utilisateurs doivent être notifiés par l'AGL. Par exemple, dans un conflit de validation de deux transactions concurrentes, les utilisateurs doivent être avertis de l'avortement de la transaction.

Marvel (Barghouti, 1992) propose un mécanisme basé sur le contrôle de la validation de transactions pour supporter ce protocole de communication. Dans cette approche, les règles qui changent l'état de la base sont encapsulées dans une transaction par des commandes d'initialisation, de terminaison et d'annulation.

Dans Triad (Sarkar and Venugopal, 1991b) ce protocole est implicite dans le mécanisme qui supporte les espaces de travail. Les conflits sont détectés et résolus lorsque les espaces de travail essayent de rendre persistantes les modifications dans la base centrale.

Epos permet que les espaces de travail soient divisés en plusieurs sous-espaces. Lorsqu'un sous-espace de travail essaye de rendre persistantes les modifications, Epos par son mécanisme d'interprétation de règles, décide de la manière de résoudre le conflit en se basant sur les connaissances stockées dans la base de règles. Ce mécanisme n'a pas encore été validé dans un cadre multi-utilisateurs.

III.10.4 Négociation

Dans ce mode, le protocole de communication se fait par la négociation entre les utilisateurs. Les utilisateurs peuvent d'un commun accord déterminer une solution aux conflits apparaissant au cours de l'exécution des processus coopérants.

Ce protocole de communication est le plus ambitieux de tous, car il permet que le travail coopératif soit supporté et contrôlé par l'ADL. Dans le domaine du génie logiciel, nous ne trouvons pas encore de système supportant ce mode de communication d'une manière explicite. Les expériences menées jusqu'à maintenant ont introduit ce protocole de communication de manière câblée dans l'AGL (Dewan and Riedl, 1993). Ces systèmes sont difficiles d'être adaptés aux particularités de chaque enterprise, puisque le modèle de processus logiciel utilisé est cablé dans l'AGL.

III.11 Orientation des utilisateurs

Un AGL centré-processus doit être capable d'informer l'utilisateur de l'état de ses processus : les tâches déjà accomplies, les états des tâches en cours d'exécution, les

tâches à exécuter ainsi que leur ordre d'exécution. Il peut aussi intervenir pour réparer, contrôler et suggérer des plans d'exécutions (Benali et al., 1989).

Alf (Canals et al., 1993), Epos et Grapple sont des systèmes bien avancés dans ce domaine. Par le réseau de tâches, un utilisateur peut à tout moment connaître l'état d'exécution de son processus et des processus actifs dans l'environnement.

Ipse 2.5 (Bruynooghe et al., 1993), Merlin (Peuschel et al., 1992) et OPM (Sugiyama, 1993) fournissent chacun une interface graphique permettant d'informer l'utilisateur, en fonction de son rôle, des activités actives et inactives de son espace de travail.

Appl/A, Triad et HFSP ne permettent pas directement d'orienter ni d'informer l'utilisateur sur l'état des processus logiciels dans l'environnement. Dans ces systèmes l'ensemble des processus actifs peut être explicité. Ce n'est pas le cas des activités déjà exécutées et à exécuter.

Dans Genesis (Ramamoorthy et al., 1988), AP5 et Marvel l'utilisateur peut s'orienter en se basant sur l'analyse des pré-conditions encapsulant les règles de production. A l'aide de cette analyse l'utilisateur peut connaître quelles sont les règles qu'il peut activer à un moment donné.

ProcessWall (Heimbigner, 1992) propose un aspect innovateur dans le domaine des AGL centré-processus. La définition et le contrôle des processus coopérant se font par un serveur des processus. Ce serveur fournit essentiellement deux fonctions : la mise-à-jour du graphe d'état des processus logiciel et l'ordonnancement de leur exécution. Les AGL clients de ProcessWall sont responsables de contrôle de l'exécution des activités correspondant aux noeuds du graphe.

III.12 Conclusion

Nous avons décrit dans ce chapitre l'évolution des AGL centré-processus ainsi que les principales directions de recherche suivies pour décrire de manière explicite les processus logiciels. Nous pouvons conclure qu'aucune approche ne semble s'imposer comme un standard. Notons aussi qu'une unification entre les approches procédurale et déclarative se manifeste nécessaire.

A notre avis, les AGL de l'approche déclarative utilisant le mécanisme de déclencheur sont des exemples de cette unification. Les processus logiciels sont décrits à la fois de

manière déclarative par des règles événement-condition-action et à la fois de manière procédurale par des programmes attachés à ces règles (la partie action).

L'idée lancée par ProcessWall (Heimbigner, 1992), i.e. la construction d'une plateforme au-dessus de laquelle de différents AGL centrés-processus peuvent être aisément intégrés, nous semble très prometteuse. La plateforme fournit les services de base pour supporter la description et l'exécution de processus logiciels, sans pour autant imposer ou privilégier un formalisme particulier.

CHAPITRE IV TEMPO : Un Formalisme pour la Description des Processus Logiciels - Concepts de Base et Langage d'Expression 83

IV.1 Introduction 83

IV.2 Critères de Choix 83

IV.3 Terminologie utilisée 85

IV.4 Le système TEMPO 85

IV.5 La modélisation des objets logiciels et sa relation avec les processus logiciels 86

IV.5.1 Le problème 86

IV.5.2 Le modèle de données d'Adèle 87

IV.5.3 Les valeurs 88

IV.5.4 Les objets 89

IV.5.5 Les attributs 89

IV.5.6 Les relations 90

IV.5.7 Les types d'objets 90

IV.5.8 Les types de relations 92

IV.6 Description et contrôle des activités 94

IV.6.1 Présentation informelle 94

IV.6.2 Le type processus logiciel 95

IV.6.3 La définition des fragments de processus logiciel 98

IV.6.3.1 Aspect syntaxique 99

IV.7 Les contraintes temporelles 100

IV.7.1 Les règles temporelles événement-condition-action 100

IV.7.2 Le modèle d'exécution des règles TECA 101

IV.7.3 Les méthodes 103

IV.7.3.1 L'état d'une méthode dans une occurrence de processus logiciel 103

IV.7.4 Les règles TECA et les transactions courtes 104

IV.7.5 Synthèse 105

IV.8 Les rôles des objets 105

IV.8.1 Motivation 105

IV.8.2 Synthèse 107

IV.8.3 La définition de types de rôles 108

IV.8.3.1 Dérivation de types de rôles 111

IV.8.3.2 Aspects syntaxiques 111

IV.9 L'évolution des processus logiciels 113

IV.9.1 Motivation 113

IV.9.2 Nature de l'évolution 113

IV.9.2.1 L' évolution du modèle de produit 113

IV.9.2.2 L'évolution du modèle de processus 114

IV.9.3 Deux axes pour supporter l'évolution des processus 114

IV.9.4 Notre proposition 115

IV.9.4.1 Au niveau du modèle de cycle de vie d'un objet : ajout de rôles 117

IV.9.4.2 Au niveau processus logiciel : ajout de sous-processus 120

IV.9.5 Les opérations sur le modèle : le support à l'évolution 120

IV.9.5.1 Ajout d'un type de processus logiciel 120

IV.9.5.2 Ajout d'un type de rôle 121

IV.9.5.3 Ajout d'un fragment dans un type de processus logiciel 122

IV.9.5.4 Ajout d'un type d'attribut 123

IV.9.5.5 Ajout d'une méthode 124

IV.9.5.6 Ajout d'un événement 124

IV.9.5.7 Ajout d'une règle temporelle événement-condition-action 125

IV.9.5.8 Suppression d'un type de processus logiciel 126

IV.9.5.9 Suppression d'un type de rôle 126

IV.9.5.10 Suppression d'une méthode 126

IV.9.5.11 Suppression d’une règle 127

IV.9.5.12 Suppression d’une définition d’attribut 127

IV.10 Le graphe d’états des occurrences de processus logiciels 127

IV.10.1 L’état “stopped″ *128*

IV.10.2 L’état “killed” 129

IV.10.2.1 La suppression de rôles et la gestion de branches 130

IV.10.3 L’etat “broken” 131

IV.10.4 L’état “sastified” 131

IV.10.5 L’état “created” 132

IV.10.5.1 La création de processus fils 132

IV.11 Conclusion 133

CHAPITRE IV TEMPO : Un Formalisme pour la Description des Processus Logiciels - Concepts de Base et Langage d'Expression

IV.1 Introduction

L'objectif majeur de la recherche dans le domaine du génie logiciel a toujours été d'augmenter la productivité des équipes chargées de développer et de maintenir des logiciels et d'améliorer la qualité des produits logiciels. Un grand nombre de méthodes, de langages, d'outils d'aide à la conception ont alors été développés.

Plusieurs AGL ont été bâtis pour supporter certaines phases du cycle de vie des logiciels. Cependant, les AGL qui supportent toutes les phases des processus logiciels sont une tendance récente dans le domaine génie logiciel.

Nous avons montré précédemment les efforts entrepris dans le domaine de l'intégration via des modèles explicites de processus logiciel (intégration par les processus logiciels). Dans cette approche, les politiques qui gouvernent l'utilisation des ressources, la communication entre les équipes de développement, la coordination entre ces équipes et la synchronisation des activités de génie logiciel peuvent être décrites et suivies dans l'environnement via un AGL dirigé par un tel modèle.

Le système TEMPO, qui représente l'aboutissement de nos recherches dans le cadre de cette thèse, entre dans cette catégorie. Dans ce chapitre, nous présentons le système TEMPO et son formalisme qui permet la descriptions des modèles de processus logiciels.

IV.2 Critères de Choix

Les projets de génie logiciel ont certaines caractéristiques qui les rendent particulièrement difficiles à gérer. Nous pouvons citer :

1. Un nombre important d'activités sont menées de manière parallèle. Ces activités ont une longue durée et partagent des objets logiciels complexes. Un AGL doit donc fournir un cadre de travail où les activités de génie logiciel peuvent être décrites et contrôlées dans un environnement multi-utilisateur.
2. Un nombre important d'utilisateurs travaille ensemble en exécutant leurs tâches respectives. Chaque utilisateur possède une connaissance incomplète sur les processus logiciels. Ces utilisateurs produisent, modifient et partagent un ensemble important d'informations interdépendantes. Dans un cadre de programmation globale, il est indispensable que chaque utilisateur ne possède qu'un point de vue partiel des informations manipulées par l'ensemble des utilisateurs. De plus il est préférable, du point de vue de la gestion, que chacun puisse ignorer, dans une certaine mesure et durant un certain temps, les activités menées par ses collègues. Dans ce contexte, il est important qu'un AGL puisse gérer ces vues multiples.
3. Un nombre important d'informations est produit, modifié et consommé au cours de l'exécution des processus logiciels. D'un côté, il est nécessaire d'administrer la distribution des activités parmi les membres des équipes de développement. D'un autre côté, il est nécessaire de gérer l'évolution des objets partagés entre ces activités ainsi que de contrôler la synchronisation des processus logiciels menés de manière parallèle. L'AGL doit également assurer le respect des protocoles d'échange d'informations au sein de l'environnement de travail.

Pour prendre en compte toutes les caractéristiques qui existent dans le cadre de la programmation globale (DeRemer and Kron, 1976; Romamoorthy, 1986), nous devons intégrer, via un modèle de processus logiciels, le contrôle de l'exécution des processus logiciels et la gestion des contraintes temporelles sur le déroulement de ces processus. Tout cela doit être fait dans l'optique de fournir un environnement où les activités de longue durée peuvent être suivies, coordonnées et synchronisées.

Nous avons conçu TEMPO de façon à ce qu'il puisse être dirigé par un formalisme exécutable orienté processus logiciels. Le formalisme TEMPO permet de décrire différents modèles de processus logiciels, de supporter la notion de ***rôle*** des objets, d'expliciter des politiques de coopération et de communication entre les activités.

IV.3 Terminologie utilisée

Des travaux de recherche récents ont porté sur la définition des termes et des concepts utilisés dans le domaine du processus logiciel (Arbaoui, 1993; Dowson et al., 1991; Feiler and Humphrey, 1993; Lonchamp, 1993). Le concepts de modèle, de type et de fragment de processus logiciel sont définis informellement de la manière suivante :

Un modèle de processus logiciel est une description formelle d'un processus logiciel spécifique. Dans TEMPO un modèle de processus logiciel est décrit par un ensemble de *types de processus logiciel.*

Un type de processus logiciel modélise une étape du processus dans un modèle de processus logiciel. Un type de processus logiciel peut agréger plusieurs *fragments de processus logiciel.*

Un fragment de processus logiciel modélise une sous-étape du processus dans un type de processus logiciel

Un formalisme pour le processus logiciel est une notation ou un langage permettant de décrire des modèles de processus logiciel, comme par exemple le formalisme TEMPO, le formalisme MASP en Alf (Canals et al., 1993), le "*backboard*" en Oikos (Montangero and Ambriola, 1993), les types de tâches en Epos (Conradi et al., 1991b), etc.

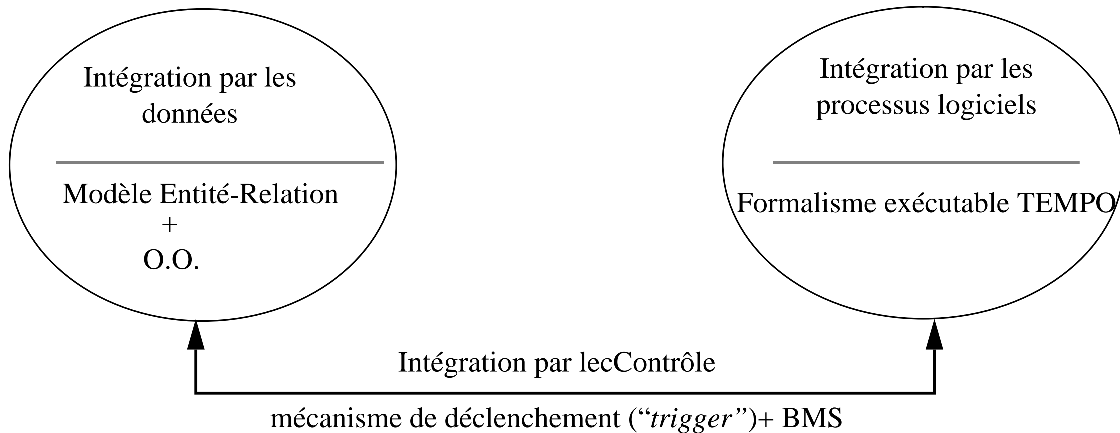
IV.4 Le système TEMPO

TEMPO est bâti sur Adèle (Belkhatir et al., 1991; Belkhatir et al., 1993) selon les axes d'intégration suivants (Thomas and Nejmeh, 1992; Wasserman, 1989) :

1. l'intégration par les données contrôlée par un modèle de produit définissant un cadre commun de description du produit logiciel : ses composants et les relations entre eux.
2. l'intégration par le contrôle via un mécanisme de déclenchement ("*trigger*");
3. l'intégration par le processus logiciel via un modèle de processus logiciel.

La figure IV.1 montre comment TEMPO s'intègre au système Adèle.

Figure IV.1 L'architecture d'Adèle/TEMPO.



BMS ::= *Broadcast Message Server*

IV.5 La modélisation des objets logiciels et sa relation avec les processus logiciels

IV.5.1 Le problème

Un objet logiciel est caractérisé par un ensemble d'attributs et par son contenu. Par exemple, un code source pascal (`PROG1.pas`) a des attributs (nombre de lignes, date de création, etc) et un contenu qui est le code source proprement dit.

Un objet peut être lié à d'autres objets par différents types de relations. Par exemple, cet objet peut faire partie d'une configuration, être utilisé dans différents produits logiciels, être en cours de validation dans un processus de test, etc.

Un objet logiciel complexe peut être composé de plusieurs objets primitifs. Par exemple un module Pascal peut être composé d'une interface et d'une réalisation. Dans ce cas, l'objet module Pascal peut être représenté comme étant un objet lié à deux autres objets par deux types de relations :

possède-interface et *possède-réalisation*.

Un objet peut avoir de multiples révisions et être en différentes versions. En prenant notre exemple d'un module Pascal, la réalisation de ce module peut avoir de multiples révisions afin de capturer l'évolution de cette réalisation dans le temps. Cette même réalisation peut avoir de multiples versions, par exemple une version pour les machines `Sun`, une autre pour les machines `Bull`, etc.

Un AGL doit donc fournir un cadre de représentation de ces types d'informations dans l'environnement.

Au cours de l'exécution des processus logiciels, plusieurs objets logiciels sont produits, consommés et modifiés. L'AGL doit donc permettre de modéliser les relations entre les différentes occurrences de processus logiciels et les objets ainsi que de contrôler l'utilisation de ces objets par ces processus.

IV.5.2 Le modèle de données d'Adèle

TEMPO utilise la base de données Adèle comme gestionnaire de ressources (objets, outils et activités). Les objets logiciels sont représentées à l'aide du modèle de données d'Adèle. Comme PCTE (Boudier et al., 1988), ce modèle est dérivé du modèle entité-relation (Chen, 1976). Ceci permet de modéliser les objets logiciels ainsi que les relations entre ces objets.

Les différents types d'objets logiciels, par exemple programmes Pascal, programmes C, code binaire, textes écrits en Latex ou avec un autre éditeur de texte, etc, peuvent être modélisés par un ensemble de types d'objets. Pour chaque type d'objet, nous pouvons définir un ensemble d'attributs permettant de caractériser les objets. Un attribut spécial est également fourni pour permettre le stockage du contenu de l'objet logiciel, par exemple le code source d'un programme. Tous les objets sont susceptibles d'être améliorés dans le temps par la génération des révisions; les objets abstraits sont donc également considérés comme étant des objets possédant plusieurs versions.
Exemple :

```
TYPEOBJECT C_interface ISA interface;
    DEFATTRIBUTE
         no_of_lines MONO = INTEGER;
END C_interface;

TYPEOBJECT C_body ISA body;
    METHOD
         compilation_with_debug;
         link;
END C_body;
```

De manière similaire à PCTE, le modèle de données d'Adèle supporte les relations entre les objets logiciels. Les liaisons sémantiques entre les objets logiciels peuvent être modélisées par des relations. Les relations sont typées et peuvent avoir des attributs pour mieux caractériser les liens sémantiques entre les objets logiciels. Adèle ne supporte que les relations binaires. Une relation lie toujours un objet origine avec un objet destination et son existence est liée à l'existence des objets qu'elle connecte.

Exemple :

```
TYPERELATION dep;
    DOMAIN
         [ type = body ]        -> [ type = interface ] OR
         [ type = interface ]   -> [ type = interface ]
    DEFATTRIBUTE
         no_of_functions MONO = INTEGER;
END dep;
```

Ceci exprime que la relation dep est définie entre un corps d’un module et une vue d’interface ou entre deux vues d’interface.

Dans Adèle, on décrit aussi bien les types de relations que les types d’objets comme la spécialisation d’un ou de plusieurs types. Le modèle de données d’Adèle a été étendu avec des concepts Orienté-Objet comme la spécialisation qui est appliquée uniformément aux objets et aux relations.

Nous donnons ensuite la définition du modèle de données.

IV.5.3 Les valeurs

Les valeurs représentent des caractéristiques liées aux objets et aux relations, elles ne correspondent pas à des données ayant une existence propre dans le monde réel. Pour commencer, nous désignons un nombre limité de notations nommées valeurs atomiques : **integer, string, real, boolean** et **date**, où le domaine de ces types de valeurs est évident. Nous considérons le symbole **nil** comme étant une valeur indéfinie. Maintenant nous considérons les ensembles suivants :

- Un ensemble fini de domaines $\mathcal{D}_1, \ldots, \mathcal{D}_n$, $n >= 1$. Nous désignons par $\mathcal{D}$ l’union de tous les domaines $\mathcal{D}_1, \ldots, \mathcal{D}_n$, $n >= 1$. Chaque domaine $\mathcal{D}_i$ désigne un ensemble de valeurs, par exemple l’ensemble de tous les entiers est un domaine.
- un ensemble infini $\mathcal{A}_n$ de symboles où les éléments sont appelés *attributs*.

Nous définissons la notion de *valeur* ainsi :

1. le symbole spécial **nil** est une valeur, nommée *valeur de base*.
2. tout élément d de $\mathcal{D}$ est une valeur nommée *valeur de base*.
3. si $v_1, \ldots, v_n$ sont des valeurs, alors $[a_1{:}v_1, \ldots, a_i{:}v_n]$ est une valeur construite. [] est une valeur construite vide. Nous considérons que les attributs a_i sont distincts et que l’ordre des attributs n’est pas important.

4. si $v_1, \ldots, v_n$ sont des valeurs distinctes, alors $\{v_1, \ldots, v_n\}$ est un ensemble de valeurs. { } est un ensemble de valeur vide.
5. si $v_1, \ldots, v_n$ sont des valeurs distinctes, alors $<v_1, \ldots, v_n>$ est une liste de valeurs. <> est une liste de valeur vide.

Nous désignons $\mathcal{V}$ l'ensemble de toutes les valeurs. Nous considérons l'opérateur **setof** pour former des listes de valeurs de longueur variable, et l'opérateur **listof** pour former des listes ordonnées de taille variable. Ainsi l'ensemble des listes ordonnées de l'ensemble $\mathcal{V}$ est définie par **listof(**$\mathcal{V}$**)**, et dans la suite l'ensemble des listes non nécessairement ordonnées de l'ensemble $\mathcal{V}$ par **setof(**$\mathcal{V}$**)**.

IV.5.4 Les objets

Contrairement aux valeurs, chaque objet a un identificateur unique. Grâce à cet identificateur l'objet peut toujours être retrouvé. Pour définir le concept d'objet nous considérons deux ensembles :

1. l'ensemble fini $\mathcal{O}_t$ composé de tous les objets.
2. l'ensemble infini $\mathcal{O}_n$ de symboles dont les éléments sont utilisés pour identifier les objets.

Un objet O $\in$ $\mathcal{O}_t$ est défini par un couple O=(o, $[a_1{:}v_1, \ldots, a_i{:}v_n]$), où o $\in \mathcal{O}_n$ est l'identificateur de l'objet O. $[a_1{:}v_1, \ldots, a_n{:}v_n]$ est un n-uplet d'attributs où $a_i \in \mathcal{A}_n$ est un attribut de l'objet O et $v_i \in \mathcal{V}^*$ est la valeur de cet attribut.

IV.5.5 Les attributs

Comme nous l'avons montré dans la section précédente les attributs permettent de définir des valeurs construites et de modéliser les propriétés des objets.

Nous avons utilisé la notion classique d'attribut valeur, c'est-à-dire qu'un attribut a est défini comme étant le couple (a,v), tel que $a \in \mathcal{A}_n$ est le nom de l'attribut et $v \in \mathcal{V}^*$ est sa valeur.

Nous considérons l'opérateur **AValeur(**O,A**)**$=v^*$ qui permet d'obtenir la valeur v^* d'un attribut A pour un objet O.

Nous considérons l'opérateur **Atype(**O,A**)**$=t$ qui permet d'obtenir le type t de l'attribut A pour un objet O. L'opérateur **ACons(**O,A**)**$=c$ permet d'obtenir le type de constructeur de l'attribut A pour un objet O, où $c \in$ {elementof, listof, setof}.

IV.5.6 Les relations

Une relation rend possible la liaison entre deux objets sémantiquement associés dans le monde réel. Pour définir le concept de relation nous considérons les ensembles suivants :

1. un ensemble fini $\mathcal{L}_t$ composé de toutes les relations.
2. l'ensemble infini $\mathcal{L}_n$ de symboles dont les éléments sont utilisés pour nommer les relations.

Une relation binaire $L \in \mathcal{L}_t$ est définie par un n-uplet $L=(l,s,d,[a_1{:}v_1, \ldots, a_i{:}v_n])$, où $l \in \mathcal{L}_n$ est le nom de la relation. $s \in \mathcal{O}_n$ est l'identificateur de l'objet source de la relation L et $d \in \mathcal{O}_n$ est l'identificateur de l'objet destination de la relation. $[a_1{:}v_1, \ldots, a_i{:}v_n]$ est un n-uplet d'attributs valeurs de la relation L, où $a_i \in \mathcal{A}$ est un attribut de la relation et $v_i \in \mathcal{V}^*$ est sa valeur.

Le tripet (l,s,d) forme l'identificateur unique de la relation L.

Nous désignons l'opérateur **AValeur**$(l,s,d,a)=v$ qui permet d'obtenir la valeur $v \in \mathcal{V}^*$ de l'attribut a de la relation l qui lie l'objet s avec l'objet d.

De manière similaire aux objets, nous définissons l'opérateur **AType**$(l,s,d,a)=t$ qui permet d'obtenir le type t de l'attribut a appartenant à la relation l qui lie l'objet s à l'objet d.

Nous désignons l'opérateur **ACons**$(l,s,d,a)=c$ qui permet d'obtenir le type de constructeur $c \in \{\text{setof}, \text{listof}, \text{elementof}\}$ de l'attribut a.

IV.5.7 Les types d'objets

Un type d'objet spécifie une structure commune à plusieurs objets. Par la structure, nous désignons les attributs ainsi que les méthodes et les règles de cohérence des objets de ce type. Pour donner une définition aux types d'objets nous considérons les ensembles suivants :

1. $\mathcal{T}_u$ un ensemble dont les élément sont appelés types d'objets et $\mathcal{T}_n$ l'ensemble infini de symboles désignant les noms des types d'objets.
2. $\mathcal{M}_u$ un ensemble dont les éléments sont appelés méthodes.
3. $\mathcal{E}_u$, un ensemble dont les éléments sont appelés règles temporelles événements-condition-action (ou simplement TECA).

Considérons l'ensemble de tous les types d'objets noté par $\mathcal{T}_t$, chaque type d'objet *T* de $\mathcal{T}_t$ est un n-uplet (*t, S, M, E*) tel que :

- $t \in \mathcal{T}_n$ est le nom du type d'objet,
- *S* est la partie structurelle du type d'objet *T*,
- *M* est un sous-ensemble de $\mathcal{M}_u$ désignant les méthodes du type d'objet *T*,
- et enfin *E* est un sous-ensemble de $\mathcal{E}_u$ désignant les règles temporelles événements-condition-action du type d'objet *T*.

Nous désignons l'opérateur **Attribute** qui appliqué à un type *T*=(*t,A,M,E*) fournit ses attributs, tel que :

> $Attribute(T)=(a_1{:}t_1, \ldots, a_i{:}t_n)$ tel que $a_i \in \mathcal{A}_n$ est le nom de l'attribut et *t* son type et si $a_i = a_j$, alors $i = j$.

Nous considérons l'opérateur **Methode** qui permet d'obtenir les méthodes d'un type d'objet *T* de $\mathcal{T}_t$, tel que :

> $Methode(P)=(m_1, \ldots, m_n)$ tel que $\forall\ i \leq n,\ m_i \in \mathcal{M}_t$.

Nous considérons l'opérateur **Rules**. Cet opérateur appliqué à un type d'objet *T* de $\mathcal{T}_t$ donne ses règles, tel que :

> $Rules(T)=(r_1, \ldots, r_n)$ tel que $\forall\ i \leq n,\ r_i \in \mathcal{E}_t$.

Nous appelons extension d'un type d'objet l'ensemble des objets ayant une structure commune et un comportement commun décrits par ce type. Nous utilisons l'opérateur **Extension(*T*)** pour désigner cet ensemble, et l'opérateur **Type(*O*)** pour retrouver le type d'un objet *O* :

> $Type(O)=T$ si et seulement si $O \in Extension(T)$.

Chaque type d'objet peut être spécialisé pour préciser sa structure. Spécialiser la structure d'un type revient à :

1. rajouter des attributs;
2. préciser la définition de certains attributs en spécialisant leurs domaines,
3. ajouter des nouvelles méthodes et règles,
4. modifier la définition des méthodes déjà existantes.

Pour permettre la spécialisation de types d'objets la relation d'ordre partiel **ISAo** est fournie. Cette relation peut classer les types d'objets suivant un treillis de types, où chaque type d'objet peut être considéré comme étant un sous-type d'un ou de plusieurs autres types d'objets. Pour définir la relation d'ordre partiel **ISAo**, nous supposons l'existence d'un symbole *OBJECT* $\in \mathcal{T}_n$ qui constitue le nom du type d'objet racine du treillis des types, tel que :

1. pour chaque type d'objet *T*, on a *T* **ISAo** *OBJECT.*
2. soient deux types d'objets *T1=(t1, A1, M1, E1)* et *T2=(t1, A2, M2, E2)*, on dit que *T1* **ISAo** T2 si et seulement si *A1* **ISAa** *A2*, *M1* **ISAm** *M2* et *E1* **ISAe** *E2* tels que :

   ISAa est une relation d'ordre partiel définie sur les ensembles d'attributs valeurs : soient les deux ensembles d'attributs propriétés A_1 et A_2, on dit que A_1 **ISAa** A_2 si et seulement si pour chaque attribut a_{2i} de type t_{2i} appartenant à l'ensemble A_2, a_{2i} appartient à l'ensemble de A_1 avec un type t_{1i} tel que t_{1i} **ISAo** t_{2i}

   ISAm est une relation d'ordre partiel sur les ensembles de méthodes définie de la manière suivante : soient deux ensembles de méthodes $M_1=\{m_{11}, m_{12},\ldots, m_{1n}\}$ et $M_2=\{m_{21}, m_{22},\ldots, m_{2n}\}$, on dit que M_1 **ISAm** *M1* si et seulement si pour chaque méthode $m_{2i} \in M_2$ implique que $m_{2i} \in M_1$. Autrement dit, M_1 **ISAm** M_2 si et seulement si l'ensemble des méthodes de M_2 est inclus dans l'ensemble des méthodes de M_1.

   ISAe est une relation d'ordre partiel définie sur les ensembles de règles : soient les deux règles E_1 et E_2, on dit que E_1 **ISAe** E_2 si et seulement si E_2 est inclus dans E_1.
3. si *T1* **ISAo** *T2* alors l'extension de *T1* est incluse dans l'extension de T2, donc *Extention*(T1) $\subseteq$ *Extention(T2).*

Nous désignons l'opérateur $SuperTypes(T) = \{T_1, ..., T_n\}$ qui permet d'obtenir la liste des super types de *T*, où pour chaque i <=n, on a T_i **ISAo** *OBJECT* et *T* **ISAo** T_i

IV.5.8 Les types de relations

De même que pour les types d'objets, un type de relation spécifie une structure commune à plusieurs relations. Pour donner une définition aux types de relations nous considérons les ensembles suivants :

- $\mathcal{L}_u$ un ensemble dont les éléments sont appelés types de relations,

- $\mathcal{L}_n$ l'ensemble infini de symboles désignant les noms des types de relations.

Nous imposons que l'intersection entre les ensembles $\mathcal{L}_n$ et $\mathcal{T}_n$ est un ensemble vide, donc $\mathcal{L}_n \cap \mathcal{T}_n = \{\}$.

Considérons l'ensemble de tous les types de relations noté par $\mathcal{L}_t$, chaque type de relation L de $\mathcal{L}_t$ est un n-uplet *(l, S, D , A, M, E)* tel que :

- $l \in \mathcal{L}_n$ est le nom du type de relation,
- A fournit les attributs de L,
- M est un sous-ensemble de $\mathcal{M}_u$ désignant les méthodes de L,
- E est un sous-ensemble de $\mathcal{E}_u$ désignant les règles temporelles événements-condition-action de L.
- S et D sont chacun un sous-ensemble de $\mathcal{T}_t$ désignant les types d'objets qui peuvent être connectés par la relation L. Comme les relations sont binaires, S fournit les types d'objets qui peuvent être utilisés comme source de la relation L tandis que D fournit les types d'objets qui peuvent être utilisés comme destination.

De même que pour les types d'objets, nous désignons les opérateurs **Attribute**(L), **Methode**(L) et **Rules**(L) qui permettent respectivement de retrouver la définition des attributs valeurs, les méthodes et les règles appartenant à un type de relation L. Nous fournissons également l'opérateur **Type**(L) qui permet de retrouver le type de relation L et l'opérateur **Extention**(L) qui fournit les relations qui appartiennent au type de relation L.

Chaque type de relation peut également être spécialisé pour préciser sa structure et son comportement. Pour permettre la spécialisation de types de relations, nous fournissons la relation d'ordre partiel **ISAl**. Cette relation peut classer les types de relations suivant un treillis de types, où chaque type de relation peut être considéré comme étant un sous-type d'un ou de plusieurs autres types de relations.

Pour définir la relation d'ordre partiel **ISAl**, nous supposons l'existence d'un symbole *RELATION* $\in \mathcal{L}_n$ qui constitue le nom du type de relation racine du treillis des types de relations, tel que :

1. pour chaque type de relation L, on a L **ISAl** *RELATION*.

2. soient les deux types de relation $L_1=(l_1, A_1, M_1, E_1, A_1, D_1)$ et $L_2=(t_2, A_2, M_2, E_2, A_2, D_2)$, on dit que L_1 **ISAl** L_2 si et seulement si A_1 **ISAa** A_2, M_1 **ISAm** M_2, E_1 **ISAe** E_2, S_1 **ISAo** S_2 et D_1 **ISAo** D_2.
3. si L_1 **ISAl** L_2 , alors l'extension de L_1 est incluse dans l'extension de L_2.

Nous désignons l'opérateur $SuperTypes(L) = \{L_1, ..., L_n\}$ qui permet d'obtenir la liste des super types de L, où pour chaque i <=n, on a L_i **ISAl** *RELATION* et L **ISAl** L_i

IV.6 Description et contrôle des activités

TEMPO fournit un formalisme permettant de décrire des modèle de processus logiciels. Un modèle de processus logiciels de taille importante peut être décrit par un ensemble d'activités (Belkhatir et al., 1991). Chaque activité est représentée par un type de processus logiciel.

Les types de processus logiciels modélisent des entités dynamiques, créées et détruites selon un ensemble de règles. Ils permettent de représenter les interactions entre une collection de vues d'objets pour exécuter une tâche ou atteindre un but.

Par exemple, la tâche de compilation d'un module peut être représenter par un type de processus logiciel. Les objets manipulés par cette tâche sont vue et ont un comportement propre à cette tâche : les options de compilation utilisées dans une activité de mise au point peuvent être différentes de celles utilisées dans une tâche de test.

Chaque type de processus logiciel peut être décrit comme une spécialisation d'un type de plus haut niveau d'abstraction. Un type de processus logiciel peut également être composé d'un ou plusieurs fragments de processus logiciels. TEMPO fournit donc les concepts de spécialisation/généralisation et composition/décomposition, amplement utilisés dans le domaine de modélisation de données, pour modéliser les processus logiciels.

IV.6.1 Présentation informelle

Comme exemple, prenons une activité de mise au point d'un document de conception d'un module. Cette activité est composée de deux sous-activités :

1. l'activité de modification qui permet d'apporter des modifications au document de conception;
2. l'activité de révision de la modification qui permet d'approuver les modifications faites sur le document de conception.

Ces activités sont décrites de la manière suivante avec le formalisme TEMPO :

```
MonitorDesign ISA PROCESS;
    CONTROL modify;
         fragment := ModifyDesign;
         cardinality := 1;
    CONTROL review;
         fragment = ReviewDesign;
         cardinality :=1;
END_OF MonitorDesign;

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 indique que trois types d'activités sont définis par les types de processus logiciels `MonitorDesign, ModifyDesign` et `ReviewDesign`. Le type de processus logiciel `MonitorDesign` est composé de deux fragments de processus logiciels :

1. **`Modify`** est le fragment de processus logiciel responsable de la modification du document de conception. Les caractéristiques de ce fragment sont décrites dans le type de processus logiciel **`ModifyDesign`**. (Un fragment de processus logiciel est donc un attribut référence vers un autre type de processus logiciel).
2. **`Review`** est le fragment responsable de la révision de cette modification. Les caractéristiques associées à ce fragment sont décrites dans le type de processus logiciel **`ReviewDesign`**.

Pour chaque type de processus, nous pouvons définir des attributs, des méthodes et des contraintes temporelles. Ces dernières sont exprimées par des règles de type déclencheur ou événement-condition-action (McCarthy and Dayal, 1989).

Nous fournissons ensuite la définition des types de processus logiciels

IV.6.2 Le type processus logiciel

Pour donner une définition aux types de processus logiciels nous considérons les ensembles suivants :

$\wp_n$ un ensemble infini de symboles désignant les noms des *types de processus logiciel*. Nous imposons que l'intersection entre les ensembles $\wp_n$, $\mathcal{T}_n$ et $\mathcal{L}_n$ soit vide. Donc, $\wp_n \cap \mathcal{T}_n \cap \mathcal{L}_n = \emptyset$.

$\mathcal{F}_u$ un ensemble dont les éléments sont appelés *fragments de processus logiciel*.

$\mathfrak{R}_u$ un ensemble dont les éléments sont appelés *rôles*.

$\mathcal{A}_u$ un ensemble dont les éléments sont appelés attributs.

$\mathcal{M}_u$ un ensemble dont les éléments sont appelés méthodes.

$\mathcal{E}_u$ un ensemble dont les éléments sont appelés règles temporelles événements-condition-action (ou simplement TECA).

Considérons l'ensemble de tous les types de processus logiciel noté par $\wp_t$, chaque type processus P de $\wp_t$ est un n-uplet *(p, F, R, A, M, E)* tel que :

- $p \in \wp_n$ est le nom du type de processus.
- F est un sous-ensemble de $\mathcal{F}_u$ désignant les fragments de processus logiciel d'un type de processus logiciel p.
- R est un sous-ensemble de $\mathfrak{R}_u$ qui désigne les *rôles* du type P. Les rôles sont définis dans la section IV.8.
- A est un sous-ensemble de $\mathcal{A}_u$ désignant les attributs du type de processus p.
- M est un sous-ensemble de $\mathcal{M}_u$ désignant les méthodes du type de processus p.
- E est la partie réactive du type rôle r désignant les règles temporelles événements-condition-action, E est un sous-ensemble de $\mathcal{E}_u$.

Nous appelons le **contexte** d'une instance d'un processus logiciel l'ensemble des objets manipulés par cette instance. Nous utilisons l'opérateur **Context(***P***)** pour désigner cet ensemble, et inversement, pour chaque objet O nous pouvons retrouver l'ensemble des instances des processus logiciels manipulant l'objet O en utilisant l'opérateur **InContext(***O***)** :

$O \in Context(P)$ si et seulement si $P \in InContext(O)$ où $O \in O_t$ et P $\in \wp_t$.

Pour permettre la spécialisation des types de processus logiciel, nous définissons une relation d'ordre partiel nommée **ISAp**. Cette relation peut classer les types suivant un treillis de types de processus logiciel. Chaque type de processus logiciel peut être considéré ainsi comme étant un sous-type d'un ou de plusieurs autres types de processus logiciel. Spécialiser un type de processus logiciel revient à affiner sa définition de la façon suivante :

On dit qu'un type de processus logiciel P_2 est une spécialisation de P_1 si et seulement si les rôles, les fragments de processus et les attributs définis dans P_2 **contiennent** les définitions des rôles, des fragments et des attributs de P_1, l'ensemble des méthodes de P_2 **inclut** l'ensemble des méthodes de P_1, et l'ensemble des règles *TECA* de P_2 **étend** l'ensemble des règles *TECA* de P_1, et enfin les instances de processus de P_2 sont aussi des instances de P_1

Pour définir la relation d'ordre partiel **ISAp**, nous supposons l'existence d'un symbole nommé *PROCESS* $\in \wp_n$, le type *PROCESS* constitue la racine du treillis des types de processus logiciel, la relation **ISAp** est définie de la façon suivante:

1. *P*our chaque type de processus logiciel, $P_i \in \wp_t$ implique que P_i **ISAp** *PROCESS*.
2. si P_2 **ISAp** $P_1 \Rightarrow$ Extension(P_2) $\subseteq$ Extension(P_1).
3. soient les deux types processus $P_1=(p_1, F_1, R_1, A_1, M_1, E_1)$ et $P_2=(p_2, F_2, R_2, A_2, M_2, E_2)$, nous disons que P_2 **ISAp** P_1 si et seulement si R_2 **ISAr** R_1, F_2 **ISAf** F_1, A_2 **ISAa** A_1, M_2 **ISAm** M_1, tel que :

ISAf est une relation d'ordre partiel définie sur les fragments de processus logiciels. Soit deux ensembles de fragments F_2 et F_1, on dit que F_2 **ISAf** F_1 si et seulement si F_1 est inclus dans F_2.

ISAr est une relation d'ordre partiel définie sur les rôles. Soit deux ensembles de rôles R_2 et R_1, on dit que R_2 **ISAr** R_1 si et seulement si R_1 est inclus dans R_2.

Cela signifie que lorsqu'un type processus P_2 est défini comme une spécialisation de P_1, P_2 hérite de la structure de rôles de P_1, c'est-à-dire que l'ensemble des rôles et des fragments de processus de P_1 est importé par P_2, de même l'ensemble des attributs et des méthodes de P_2 est hérité de P_1.

Nous désignons l'opérateur *SuperTypes(P)* = $\{P_1, ..., P_n\}$ qui permet d'obtenir la liste des super types de *P*, où pour chaque i <=n, on a P_i **ISAo** *PROCESS* et *P* **ISAo** P_i

La syntaxe de définition d'un type processus est la suivante :

```
process-name ISA list-of-super-process-types ;
        [ ATTRIBUTE
                [ list-of-attributes ] ]
        [ METHOD
                [ list-of-methods ] ]
        [ RULES
```

[*list-of-rules*]]

[*list-of-process-fragments-*]
[*list-of-roles*]
END_OF *process-name ;*

Tels que :

Process-name est un symbole $\wp_n$ désignant le nom du type processus logiciel;

List-of-super-process-types est une liste de noms de types processus de $\wp_n$ désignant les noms des super types de processus logiciel;

List-of-roles est un ensemble de définitions de rôles étant inclus dans l'ensemble $\mathfrak{R}_u$.

List-of-attributs est un ensemble de définitions d'attributs du processus logiciel *process-name;*

List-of-methods est un ensemble de méthodes appartenant au processus logiciel *process-name*;

List-of-rules est un ensemble de règles temporelles événements-condition-action faisant partie du processus logiciel *process-name;*

List-of-process-fragments fournit un ensemble de fragments de processus logiciel composant le type de processus logiciel *process-name*.

IV.6.3 La définition des fragments de processus logiciel

La décomposition est une approche largement utilisée dans la communauté du génie logiciel pour réduire la complexité et structurer les activités. Un modèle de processus logiciel doit intégrer un constructeur de décomposition.

Dans notre cas, nous pouvons définir un type de processus logiciel comme une agregation d'un ou de plusieurs fragments de processus logiciels. (Nous avons préféré utiliser le nom fragment au lieu de sous-processus.) Ainsi une activité génie logiciel complexe peut être modélisée par un arbre de processus logiciels et contrôlée comme tel.

Pour donner une définition aux fragments de processus logiciels nous considérons l'ensemble $\mathcal{F}_n$ comme un ensemble infini de symboles désignant les noms de fragments. Nous imposons que l'intersection entre les ensembles $\wp_n$, $\mathcal{T}_n$, $\mathfrak{R}_n$, $\mathcal{F}_n$ et $\mathcal{L}_n$ soit vide. Donc, $\wp_n \cap \mathcal{F}_n \cap \mathcal{T}_n \cap \mathcal{L}_n \cap \mathfrak{R}_n = \varnothing$

Considérons l'ensemble de tous les fragments de processus noté $\mathcal{F}_t$, chaque fragment *F* de $\mathcal{F}_t$ est un n-uplet *(f, P, ProcessTypes, Card)* tel que

- $f \in \mathcal{F}_n$ est le nom du fragment.

- *P* est le type de processus logiciel dans lequel *F* est défini.
- *Card* fournit le nombre maximal d'occurrences que le fragment *F* peut avoir.
- *ProcessTypes* est la liste des types de processus logiciels à laquelle la spécification du fragment *F* fait référence.

Nous désignons les opérateurs suivants :

1. *Card(P, F)=Card* qui appliqué à un fragment *F* appartenant au type de processus logiciel *P* fournit sa cardinalité.
2. *SType(P, F)* = $\{P_1, ..., P_n\}$ qui appliqué à un fragment *F* appartenant au type de processus logiciel *P* retourne la liste des types de processus logiciels d'où le fragment *F* est dérivé. Pour chaque P_i de $\{P_1, ..., P_n\}$ P_i est un type de processus logiciel, donc P_i **ISAp** *PROCESS*. Le fragment *F* doit être compatible avec chaque P_i de $\{P_1, ..., P_n\}$.

Nous appelons la liste de fragments d'un type processus logiciel, l'ensemble des type de fragments inclus dans un type de processus logiciel. Nous utilisons l'opérateur **Fragmentlist(***P***)** pour désigner cet ensemble. Cet opérateur appliqué à un type de processus *P=(p,FL)*, donne sa structure de fragments.

> $Fragmentlist(P)=\{f_1{:}D_1, f_2{:}D_2, \ldots, f_n{:}D_n\}$, $n >= 1$
> tel que $f_i \in \mathcal{F}_n$ et $f_i \neq f_j$ pour $i \neq j$, $D_i=\{Card_i, ProcessTypes_i\}$.

De manière similaire aux types de rôles, nous appelons les instances d'un fragment l'ensemble des processus logiciels sous le contrôle d'une occurrence de processus logiciel donnée. Nous définissons l'opérateur **IProcess(***P, F***)** pour retrouver cet ensemble, et l'opérateur **PlayProcess(***P, Z***)** pour retrouver le type de fragment de processus qu'une occurrence de processus logiciel *Z* joue dans un processus logiciel *P*. Ces deux opérateurs sont définis ainsi :

> *PlayProcess(P, Z) = F* si et seulement si $Z \in$ *IProcess(P, F),* où *P* et *Z* sont deux occurrences de processus logiciels et *F* est un type de fragment de processus logiciel.

IV.6.3.1 Aspect syntaxique

Nous donnons ci-dessous la syntaxe des fragments de processus logiciels.

```
list-of-sub-process ::=
 {  CONTROL fragment-name;
          Fragment          = process-types;
          card              = CardMax; }*
```

Tel que :

`fragment-name` est un symbole de $\mathcal{F}_n$ désignant le nom du fragment de processus logiciel.

`process-types` fournit la liste des types de processus logiciels que le fragment `fragment-name` peut référencer.

`CardMax` fournit la cardinalité maximale d'un type de sous-processus.

IV.7 Les contraintes temporelles

Un autre aspect qui a suscité notre attention est le contrôle de l'exécution des activités dans le temps. Le processus de modification est inhérent aux activités de génie logiciel. Nous devons donc assurer que les modifications se déroulent conformément à certaines propriétés temporelles liées au processus logiciel. Pour cela, il est nécessaire de fournir des concepts permettant de décrire les contraintes temporelles ainsi que des mécanismes qui rendent possible la vérification de ces contraintes dans l'AGL au cours de l'exécution des processus logiciels.

IV.7.1 Les règles temporelles événement-condition-action

Les contraintes temporelles sont exprimées sous forme de règles temporelles événement-condition-action (TECA) (McCarthy and Dayal, 1989).

En reprenant l'exemple de l'activité de mise à jour du document de conception d'un module, il est possible de définir des règles sur l'ordonnancement des opérations. Par exemple :

```
ModifyDesign ISA PROCESS;
    ATTRIBUTES
         begin_date = DATE := now();
         end_date = DATE;
         deadline = DATE;
    METHODS
         continue_run;
         . . .
    RULES
(1)      AFTER WHEN deadline_arrived
             DO stop_run;
(2)      PRE WHEN continue_run and
             PAST not deadline_changed
               FROM last(deadline_arrived) UNTIL now()
             DO ABORT;
END_OF ModifyDesign;
```

- La règle décrite dans la ligne 1 spécifie que l'activité de modification du document de conception doit être arrêtée lorsque la date prévue pour son achèvement est atteinte
- la règle de la ligne 2 exprime que pour reprendre l'exécution de cette activité (car non terminée), il est alors nécessaire que la date prévue pour sa fin soit préalablement changée.

IV.7.2 Le modèle d'exécution des règles TECA

Les règles TECA sont définies à la fois dans le modèle de données et dans le modèle d'activités. Elle sont héritées dans la hiérarchie des types.
Dans le modèle de données, les règles TECA décrivent des contraintes d'intégrité indépendantes du contexte d'utilisation des objets.
Dans le modèle d'activités ces règles sont utilisées pour exprimer les politiques de développement de logiciels : l'ordre d'exécution des activités, leur synchronisation, les contraintes d'utilisation des ressources logicielles.

Une règle TECA prend la forme suivante :

"**WHEN** ***event*** **DO** ***Method***"

où

- ***event*** est un prédicat qui exprime un événement sur l'état présent où passé du système ou sur la base d'objets,
- ***method*** est une méthode.

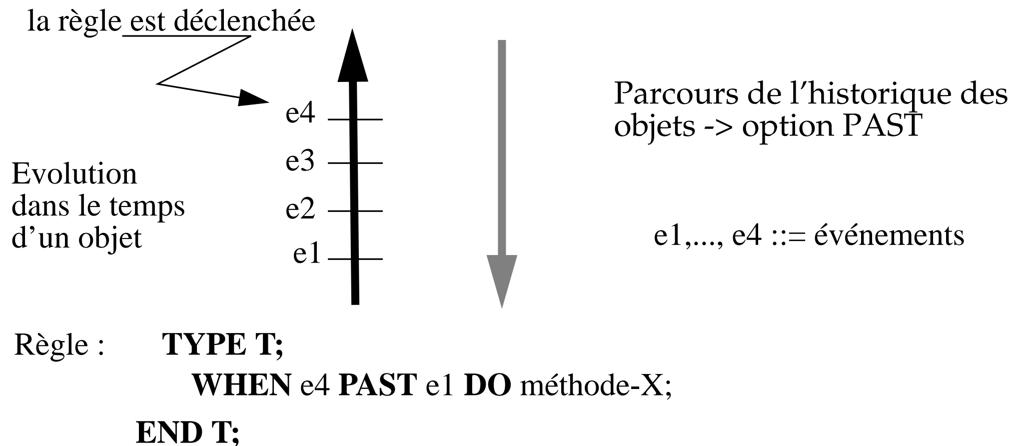
Exemple:

```
DEFEVENT delete_obj = [ !cmd = rmobj] ;
```

L'événement *delete_obj* est défini dans cet exemple comme étant l'événement qui est déclenché lorsque la commande courante (!*cmd*) est une commande de suppression d'objet (*rmobj*).

Nous avons ajouté l'opérateur < **PAST** > dans l'expression définissant l'événement afin de permettre d'exprimer des conditions sur le passé. Cet opérateur est interprété par rapport à l'historique d'évolution des objets. Les mises à jour effectuées sur un objet sont stockées dans cet historique (le changement des attributs et des événements). Les contraintes temporelles sont vérifiées par un parcours inverse de l'historique des objets depuis le déclenchement de l'événement jusqu'à satisfaction de la contrainte temporelle.

Les contraintes temporelles sont exprimées par rapport aux propriétés de l’objet (les attributs et les événements stockés dans l’historique des objets). Si les contraintes temporelles ne sont à aucun moment vérifiées, alors aucune opération n’est exécutée.
Exemple :

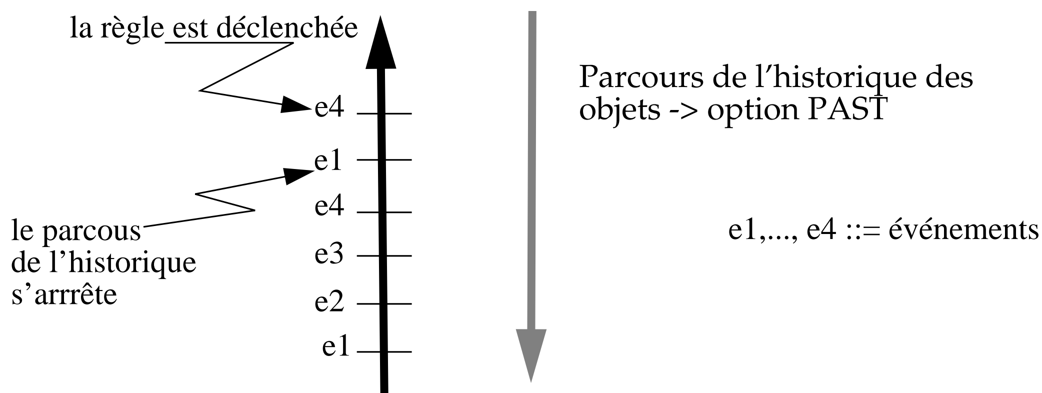


Dans l’exemple cité ci-dessus, lorsque l’événement **e4** est levé la règle

WHEN e4 **PAST** e1 **DO** méthode-X;

est déclenchée. L’historique de l’objet sur lequel l’événement **e4** est survenu est parcouru pour vérifier si l’événement **e1** a survenu précédemment (la clause PAST). La **méthode-X** est exécutée si l’événement **e1** a été déjà enregistré dans l’historique de l’objet.

Le processus de parcours de l’historique s’arrête lorsque l’expression fournie dans la clause **PAST** est satisfaite, même si l’historique contient d’autres informations qui peuvent changer le contexte d’exécution de la règle (*“late-binding” des informations).* Par exemple, avec la règle définie précédemment, le clause **PAST** sera satisfaite lorsque le processus de parcours de l’historique rencontre le dernier événement **e1**.



IV.7.3 Les méthodes

Une méthode est un programme écrit dans un langage impératif simple similaire au “*shell*” d’Unix.

```
METHOD delete ;
    IF [%state == stable] THEN ABORT
    ELSE "rmobj %name ";
END delete;
```

Cette méthode permet de supprimer des objets dont l’état n’est pas stable.

Le mécanisme de liaison-retardée (*“late-binding”*) est utilisée lors de l’exécution des méthodes. Dans l’exemple ci-dessus, lorsque que la méthode **delete** est exécutée, la variable **%name** prend comme valeur l’identificateur de l’objet sous suppression.

Des méthodes peuvent être associées à un type d’objet, à un type de relation ainsi qu’à un type de processus et de rôle. Considérons un ensemble $\mathcal{M}_n$ infini de symboles servant à nommer les méthodes et l’ensemble M_t de toutes les méthodes, une méthode $M \in \mathcal{M}_t$ est définie ainsi :

> *M={m,Signature, Body}*, où $m \in \mathcal{M}_n$ est le nom de la méthode *M*, *Signature* est sa signature, et *Body* est le corps de la méthode.

IV.7.3.1 L’état d’une méthode dans une occurrence de processus logiciel

Nous considérons qu’une méthode peut avoir deux états : **armée** ou **désarmée**. Lorsqu’une méthode **M** est **armée** elle est habilitée à envoyer des messages, et inversement, si elle est **désarmée** elle n’est pas habilitée à le faire. L’état d’une méthode est fonction de l’état d’avancement d’un processus logiciel. Par exemple, dans un processus d’édition de texte, normalement, la méthode ***“coller”*** peut être **armée** ou **désarmée** suivant l’état du processus d’édition.

Dans TEMPO, lorsqu’une méthode dans un état **désarmé** reçoit un message, un code d’erreur est retourné à l’objet qui a envoyé le message et les procédures d’exception sont alors exécutées.

Les opérateurs suivants permettent de mettre à jour et de consulter l’état d’une méthode :

unmask(P,R,M). Cet opérateur permet de rendre active la méthode $\mathbf{M} \in \mathcal{M}$ associée au rôle $\mathbf{R} \in \Re$ qui appartient à une occurrence de processus logiciel $\mathbf{P} \in \wp$. Nous supposons l’existence d’un symbole spécial nommé **allmethods** $\in \mathcal{M}_n$. Lorsque l’opérateur *unmask(P,R,allmethods)* est utilisé,

toutes les méthodes du rôle **R** de l'occurrence de processus **P** seront démasquées.

mask(P,R,M). Cet opérateur produit l'effet inverse de l'opérateur précédent. Il permet de rendre inactive la méthode **M** $\in \mathcal{M}$ associée au rôle **R** de l'occurrence de processus logiciel **P**. Lorsque l'opérateur *mask(P,R,allmethods)* est utilisé, toutes les méthodes du rôle **R** du **P** seront donc inactives

lsmt(P,R,M)=state. Cet opérateur fournit l'état (*state* $\in$ {*armée, désarmée*}) de la méthode **M** associée au rôle **R** du processus **P**. Si l'opérateur *lsmt(P,R,allmethods)* est utilisé, les états de toutes les méthodes du rôle **R** appartenant au processus logiciel **P** seront positionnés.

IV.7.4 Les règles TECA et les transactions courtes

Une règle TECA, définie dans un type, est exécutée par le mécanisme de déclencheurs d'Adèle (Belkhatir et al., 1991) lorsque l'événement associé est vrai pour une instance de ce type.

L'exécution des règles TECA est liée au mécanisme de transaction d'Adèle. Il existe quatre modes d'exécution des règles TECA par rapport à la validation des transactions :

```
PRE         {liste de règles}
POST        {liste de règles}
AFTER       {liste de règles}
EXCEPTION   {liste de règles}
```

Certains règles peuvent être définies comme des *pré-conditions* (avant l'action principale), d'autres comme des *post-conditions* (après l'action principale). Toute incohérence détectée lors de l'exécution des règles entraîne le rejet de l'opération effectuée sur la base de données. Ainsi, pour toute opération, le bloc suivant est exécuté :

```
PRE {liste de triggers}
    méthode
POST {liste de triggers}
```

L'ensemble du bloc est considéré comme une transaction unique même si les règles associées ou l'action correspondante déclenchent d'autres opérations. Une primitive **"EXCEPTION"**, rencontrée dans ce bloc, permet d'annuler toutes les opérations effectuées dans ce bloc, donc de faire avorter la transaction.

Si la transaction est validée, les règles associées au bloc "**AFTER**" sont exécutées; sinon une fois la transaction défaite, les règles associées au bloc **"EXCEPTION"** sont exécutées.

IV.7.5 Synthèse

D'autres AGL utilisent également les règles ECA, comme par exemple Elen, AP5, Alf, Triad et Appl/A. Tandis qu'Alf, Elen et TEMPO fournissent quatre modes d'exécution des règles TECA (PRE, POST, AFTER et EXCEPTION), AP5, Triad et Appl/A ne supportent qu'un mode d'exécution, le mode POST. Cependant ces systèmes ne fournissent pas de concept pour prendre en compte les événements temporels comme TEMPO.

Les règles événement-condition-action d'ALF sont très similaires à celles proposés par Adèle/TEMPO. Cependant Alf ne fournit pas le concept de méthode. Tous les actions doivent être définies par les opérateurs (règles de production selon le formalisme MASP). L'exécution d'un opérateur peut déclencher un processus de chaînage avant dans l'espace privé de l'utilisateur (un ASP selon la terminologie ALF). Les règles ECA sont définies ailleurs et sont exécutées par un autre mécanisme ("*trigger*"). Dans TEMPO nous n'avons adopté qu'un concept pour définir les contraintes sur l'exécution de méthodes ainsi que les contraintes sur l'utilisation des objets, i.e., les règles ECA.

D'autres systèmes comme par exemple AP5 (Narayanaswamy, 1993), ODE (Gehani et al., 1992) et SAMOS (Gatziu et al., 1991) fournissent aussi en quelque sorte des règles ECA sur le temps. Cependant dans ces systèmes il n'est pas permis de spécifier des conditions sur les activités qui ont eu lieu dans le passé. Ils se contentent de décrire que certaines activités (méthodes dans la plus part des cas) doivent exécuter dans un temps absolu dans le futur, comme par exemple tous les jours à 19:00 ou le lendemain à 18:00, etc.

IV.8 Les rôles des objets

IV.8.1 Motivation

Le problème de multiples perspectives ou multiples points de vue est souvent posé dans la vie d'un produit logiciel. Dans ce contexte, plusieurs utilisateurs manipulent simultanément des objets, utilisant des vues différentes de ces objets, et ayant des actions limitées et dirigées par des contraintes propres à leurs activités. Ces utilisateurs, gérés par de multiples stratégies de développement, manipulent différents modèles d'un même produit. Un AGL doit donc fournir un cadre de travail permettant de décrire et de contrôler ces aspects dans l'environnement.

TEMPO fournit des concepts permettant la description et la structuration de multiples points de vue. En s'appuyant sur le concept de *rôle*, chaque occurrence de processus

logiciel peut avoir, pour chaque objet manipulé, des contraintes (règles TECA) et des opérations locales (méthodes) ainsi que des propriétés locales (attributs). Par exemple, un module appartenant au type d'objet Pascal, M1, a des propriétés et des contraintes hérités de ce type. Par le concept du rôle, le module M1 peut avoir des nouvelles propriétés, de nouvelles méthodes et de nouvelles contraintes temporelles en fonction du rôle qu'il joue dans une activité.

Par exemple :

```
TYPEOBJECT C_body ISA body;
    METHOD
         compilation; # avec option de "debug"
         link;
END C_body;

test ISA PROCESS;
    ROLE under_test;
         derived_from := C_body;
         METHOD
              compilation; # sans option de "debug"
END_OF test;

integration ISA PROCESS;
         ROLE under_integration;
              derived_from := C_body;
              METHOD
                   compilation; # sans option de "debug"
                                # avec option d'optimization
END_OF integration;
```

L'exemple ci-dessus montre que les objets de type **C_body** peuvent être manipulés différemment suivant le rôle qu'ils jouent. Les objets de type **C_body** jouant le rôle **under_integration** dans un processus d'intégration seront compilés de manière différente de celle décrite dans le type **C_body**. De même, lorsque ces objets jouent le rôle **under_test** dans un processus de test, ils sont compilés de façon différente.

Les rôles sont typés. Un type de rôle peut faire référence à différents types d'objets. Cela permet d'intégrer plusieurs comportements et plusieurs propriétés, venant de différents types d'objets, dans une unique vue. 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, par le concept de rôle, être vu durant l'exécution d'une étape spécifique du processus de logiciel d'une manière homogène. Cette homogénéité est maintenue en utilisant les règles d'héritage multiples utilisées dans les modèles orientés objet.

Par exemple :

```
test ISA PROCESS;
    ROLE under_test;
        derived_from := C_body;
        METHOD
            compilation; # sans option de "debug"
    ROLE interfaces;
        derived_from := C_interface, CPP_interface;
            METHOD
                list;
END_OF test;
```

Les types `C_interface` (interfaces de programmes C) et `CPP_interface` (interfaces de programmes C++) sont spécialisés dans le processus test par le rôle `interfaces`. Les objets de type `C_interface` ou `CPP_interface` jouant ce rôle seront manipulés par les méthodes décrites dans ce rôle. La méthode **`list`** peut donc être utilisée sur les interfaces C ainsi que sur les interfaces C++.

Plusieurs rôles peuvent être définies pour un processus logiciel;
un processus 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 processus logiciel. Parallèlement, un même objet peut jouer différents rôles dans différents processus logiciels.

IV.8.2 Synthèse

D'autres AGL ont introduit récemment des concepts similaires au concept de rôle proposé dans TEMPO, comme par exemple Merlin (Peuschel et al., 1992), ES-TAME (Oivo and Basili, 1992) et PCTE par le concept de "*Schéma de Travail*".

Merlin permet de restreinte ou d'élargir la vision d'un utilisateur en fonction de son rôle (manager, programmer, ingénieur, etc). Il permet également de déterminer les actions qui peuvent d'être appliquées sur les objets formant la vision de l'utilisateur. Ainsi les méthodes, peuvent être masquées en fonction de rôle de l'utilisateur.

ES-TAME permet qu'un objet change son type dynamiquement au cours de l'exécution des activités logicielles. Par conséquent, les attributs et les méthodes de l'objet peuvent changer en fonction de l'activité où l'objet est manipulé. Cette flexibilité de modification de type peut être utilisée comme un mécanisme d'implémentation du concept de ***rôle*** proposé par TEMPO.

Dans PCTE (Boudier et al., 1988) un même objet peut avoir des propriétés différentes (des attributs et de relations) en fonction du "*schéma de travail*" où il est manipulé. Cependant PCTE ne fournit ni de méthodes ni de règles TECA comme TEMPO.

Dans le domaine des bases de données, le problème lié à la modélisation des rôles des objets tent à être utilisé dans le but de mieux décrire les étapes d'évolution des objets dans le temps ou les facettes d'utilisation de ces objets (Bachman and Daya, 1977). Plusieurs travaux ont été entamés et continuent à être menés pour résoudre ce problème. En général, la démarche utilisée consiste à adopter un langage persistant orienté-objet et à l'étendre avec le concept de vue, comme par exemple les langages Aspects (Richardson and Schwartz, 1991), Fibonacci (Albano et al., 1993) et Views (Shilling and Sweeney, 1989). La manipulation d'un objet se fait donc via ces vues. Par rapport à notre approche, ces langages présentent les limitations suivantes :

- Ils ne fournissent pas de concepts pour modéliser les activités où les objets doivent être manipulés. Par conséquent, la manière d'apercevoir et de manipuler un objet est décrite sans prendre en considération le contexte où cet objet va être utilisé.
- Le choix de la vue à utiliser est délégué au programmer de l'application. Cette décision est faite en décrivant dans le programme la vue qui doit répondre aux messages envoyés à l'objet.
- En général, ces langages ne fournissent pas de concepts pour l'agrégation de vues.

IV.8.3 La définition de types de rôles

Les rôles sont utilisés pour modéliser les relations entre une occurrence d'un processus logiciel et les objets logiciel qui sont manipulés par celui-ci. Un rôle désigne donc un ou plusieurs objets identifiés. Un type de rôle spécifie une structure commune et un comportement commun à plusieurs objets de type différents manipulés de manière similaire dans une occurrence de processus logiciel. Par la structure commune, nous désignons la partie statique du rôle, c'est-à-dire les attributs valeur du type de rôle. Par le comportement, nous désignons la partie dynamique du rôle, c'est-à-dire les méthodes et les règles TECA du type de rôle.

Pour donner une définition aux types de rôles nous considérons les ensembles suivants :

1. $\Re_n$ un ensemble infini de symboles désignant les noms de types de rôles. Nous imposons que l'intersection entre les ensembles $\wp_n$, $\mathcal{T}_n$, $\Re_n$, $\mathcal{F}_n$ et $\mathcal{L}_n$ soit vide. Donc, $\wp_n \cap \mathcal{T}_n \cap \mathcal{L}_n \cap \mathcal{F}_n \cap \Re_n = \varnothing$
2. O_t l'ensemble formé de tous les objets manipulés durant tout le déroulement des processus logiciels.

Considérons l'ensemble de tous les types de rôles noté par $\Re_t$, chaque type de rôle R de $\Re_t$ est un n-uplet *(r,P,Cons,Mode,I,T,S,M,E)* tel que :

- $r \in \Re_n$ est le nom du type de rôle.
- $P \in \wp_n$ est le nom du type de processus logiciel dans lequel le rôle *r* est défini.
- *Cons* est un **constructeur** qui spécifie si le rôle se réfère à un objet, un ensemble d'objets ou une liste ordonnée d'objets :

  Cons $\in$ *{ElementOf, SetOf, ListOf}*.

  Dans le premier cas, le constructeur est **ElementOf**, dans le deuxième cas le constructeur est **SetOf**, et dans le troisième cas le constructeur est **ListOf**. **ElementOf** est le valeur par défaut.

- *Mode* désigne la manière par laquelle les objets référencés par ce rôle peuvent être manipulés, où

  Mode $\in$ *{modifiable, visualizable, new}*.

  a. Le mode *modifiable* associé à un des rôles du type de processus *P* signifie que les objets référencés par ces rôles peuvent être modifiés au cours de l'exécution du processus.

  b. Le mode *visualizable* signifie que les objets référencés par ces rôles ne pourront être que consultés au cours de l'exécution du processus.

  c. Le mode *new* implique que les objets référencés par ce rôle seront créés durant l'exécution du processus.

- *I* est une *expression logique* qui stipule des contraintes d'instantiation sur les propriétés des objets référencés par le rôle.
- *T* est un sous-ensemble de $\mathcal{T}_u$. *T* fournit la liste des types d'objets d'où le rôle *R* est dérivé.
- *S* est la partie structurelle du type de rôle *R* qui appartient à un type de processus logiciel *P*.
- *M* est un sous-ensemble de $\mathcal{M}_u$ désignant les méthodes du type de rôle *R*.
- *E* est la partie ré-active du type rôle *R* désignant les règles TECA, où *E* est un sous-ensemble de $\mathcal{E}_u$.

Nous appelons la liste des types de rôles d'un type processus logiciel, l'ensemble de types de rôles inclus dans un type de processus logiciel. Nous utilisons l'opérateur **Rolelist**(*P*) pour désigner cet ensemble. Cet opérateur appliqué à un type de processus *P=(p,RS)*, donne sa structure de rôles.

$Rolelist(P)=\{r_1:D_1, r_2:D_2, \ldots, r_n:D_n\}, n >= 1$
tels que $r_i \in \Re_n$ et $r_i \neq r_j$ pour $i \neq j$, $D_i=\{Mode_i, Cons_i, I_i, T_i, M_i, E_i\}$.

Nous considérons les opérateurs suivants :

1. $Mode(P,R_j) = Mode_j$ qui appliqué à un type de rôle R_j fournit son mode où $Mode_j \in \{modifiable, visualizable, new\}$.
2. $Cons(P,R_j) = Cons_j$ qui appliqué à un type de rôle R_j fournit son constructeur, où $Cons_j \in \{ElementOf, SetOf, ListOf\}$.
3. $RType(P,R_j) = T_j$ qui appliqué à un type de rôle R_j retourne la liste de types d'objets d'où le rôle R_j est dérivé.

Nous appelons les instances d'un type de rôle l'ensemble des objets référencés par un rôle dans une occurrence de processus logiciel. Nous utilisons l'opérateur **IRole**(P,R) pour désigner cet ensemble, et l'opérateur **Playrole**(P,O) pour retrouver le type de rôle qu'un objet O joue dans un processus logiciel. Les opérateurs **Playrole** et **IRole** sont définis de la manière suivante :

$Playrole(P,O)=R$ si et seulement si $O \in IRole(P,R)$, où $P \in \wp$, $R \in \Re$ et $O \in O$.

Un objet ne peut jouer qu'un rôle dans un fragment de processus logiciel :

1. *étant donné que* $O_i \in Context(P_i)$ *et* $O_j \in Context(P_j)$.
2. *que* $R_i \in Rolelist(Ptype(Pi))$ *et* $R_j \in Rolelist(Ptype(P_j))$.
3. *et que* $Playrole(Pi,Oi)=Ri$ *et* $Playrole(Pj, Oj)=R_j$.
4. *alors si* $O_i=O_j$ *et* $P_i=P_j$ *implique que* $R_i=R_j$.

Par contre, le même objet peut jouer plusieurs rôles dans différents processus logiciels :

1. *étant donné que* $O_i \in Context(P_i)$ *et* $O_j \in Context(P_j)$.
2. *que* $R_i \in Rolelist(Ptype(P_i))$ *et* $R_j \in Rolelist(Ptype(P_j))$.
3. *et que* $Playrole(P_i,O_i)=R_i$ *et* $Playrole(P_j, O_j)=R_j$.
4. *alors si* $O_i=O_j$ et $R_i \neq R_j$ on a $P_i \neq P_j$.

De même que pour les types d'objets, nous désignons les opérateurs **Structure**(P,R), **Methode**(P,R) et **Rules**(P,R) qui permettent respectivement de retrouver la définition des attributs valeurs, les méthodes et les règles d'un type de rôle R appartenant à un type de processus logiciel P.

IV.8.3.1 Dérivation de types de rôles

Un type de rôle est toujours formé par la spécialisation d'un ou de plusieurs types d'objets. Donc un type de rôle doit être compatible avec les types d'objets qui composent celui-ci. Rappelons-nous qu'un type de rôle *R* est défini comme étant un n-uplet *R :=(r, Cons, Mode, I, T, S, M, E)* tel que :

- *r* est le nom du type de rôle
- *T* est un sous-ensemble de $\mathcal{T}_u$ désignant les types d'objets d'où le type de rôle *R* est dérivé.

Ainsi, par compatible, nous voulons dire que la règle suivante doit être respectée dans la formation d'un type de rôle:

> Pour chaque type d'objet $T_i \in T$, on dit que *R* est dérivé de T_i si et seulement si la structure de *R* **contient** la structure de T_i, l'ensemble des méthodes de *R* **inclus** l'ensemble des méthodes de T_i, et l'ensemble des règles *TECA* de *R* **étend** l'ensemble des règles *TECA* de T_i.

La relation d'ordre partiel **IDO** (*"is derived of"*) est définie ainsi :

> pour chaque type d'objet $T_i=\{t_i, S_i, M_i, E_i\}$, où $T_i \in T$, on dit que R **IDO** T_i si et seulement si S **ISAs** S_i et M **ISAm** M_i et E **ISAe** E_i.

IV.8.3.2 Aspects syntaxiques

```
List-of-roles ::=
 {  ROLE RoleName ;
        cons := { elementof | listof | setof };
        mode := { modifiable | visualizable | new };
        card := Minimun , Maximum;
        derived = ListOfObjectTypes;
        [ ATTRIBUTE
            [ list-of-attributes ]]
        [ METHOD
            [ list-of-methods ]]
        [ RULES
            [ list-of-rules ]] }*
```

Tel que :

- `RoleName` est un symbole de l'ensemble $\Re_n$ désignant le nom du type de rôle.
- **`cons`** est le constructeur du rôle.

a. Le constructeur `elementof` indique que rôle *RoleName* ne peut référencer qu'un objet,

b. `listof` indique que rôle peut faire référence a une liste d'objets. Cette liste est de taille variable et ordonnée.

c. Le constructeur `setof` indique que rôle peut faire référence à un ensemble d'objets.

- Le constructeur **`mode`** fournit le mode d'utilisation des objets référencés par le rôle.

  a. Le mode **`modifiable`** signifie que les objets référencés par le rôle peuvent être soumis à des modifications,

  b. **`visualizable`** indique que ces objets ne seront disponibles qu'en lecture,

  c. **`new`** signifie que les objets référencés par le rôle seront créés durant l'exécution du processus logiciel.

- Le constructeur **`card`** fournit la cardinalité minimale et maximale d'un type de rôle. Si **`mode`** est égal à **`elementof`**, la cardinalité doit être égale à `{0,1}` ou `{1,1}`.
- **`derived`** fournit la liste des types d'objets et la liste des types de rôles d'où le rôle *`RoleName`* est dérivé, tel que :

  ListOfObjectTypes ::= TypOrRole [, TypOrRole] *
  TypeOrRole ::= { T | R | T (expression) | R (expression) }

  où $T \in \mathcal{T}_u$ est un type d'objet et $R \in \mathfrak{R}_u$ est un type de rôle. `expression` est une expression logique. Cette expression a pour fonction d'exprimer la contrainte d'instantiation sur les objets. C'est-à-dire qu'un objet, pour être référencé par un rôle, doit satisfaire cette expression.

Le rôle *RoleName* doit être compatible avec la définition de chaque élément de la liste *ListOfObjectTypes*. Par compatible, nous entendons que la structure, les méthodes et les règles définies dans le rôle *RoleName* doivent respecter la définition de chaque élément de la liste *ListOfObjectTypes*. Autrement dit, le rôle *RoleName* est un sous-type de l'ensemble de types qui composent la liste **`derived`**.

IV.9 L'évolution des processus logiciels

IV.9.1 Motivation

Les premières expériences concernant les langages de programmation orientés processus ont été consacrées surtout à la capacité de ces langages à modéliser les différents aspects du cycle de vie des processus logiciels. La préoccupation a été de définir un langage capable de représenter les utilisateurs, les activités de génie logiciel, les objets logiciels, les outils, et finalement les relations entre ces différentes entités. L'expérience acquise dans le domaine de la description des processus logiciels et dans le contrôle de l'exécution de ces descriptions a soulevée les problèmes concernant l'évolution des descriptions des processus logiciels. Cette évolution est une conséquence de la longue durée de vie des projets logiciels. Elle se manifeste d'une part par l'ajout de nouvelles ressources comme par exemple de nouveaux types d'objets logiciels, d'outils, d'agents et d'autre part par la déviation des stratégies de développement pour prendre en compte de nouvelles situations.

IV.9.2 Nature de l'évolution

Nous pouvons classifier les évolutions en deux grandes catégories :

1. les évolutions liées à la mise à jour du modèle décrivant les objets manipulés durant l'exécution des processus logiciels,
2. les évolutions liées à la mise à jour du modèle décrivant les processus.

Ces deux types d'évolution seront appelés par la suite évolution du modèle de produit et évolution du modèle de processus.

IV.9.2.1 L' évolution du modèle de produit

Cette évolution porte aussi bien sur le contenu de l'objet que sur sa description. L'évolution du contenu d'un objet pris en compte dans les AGL, est traitée par des mécanismes de versionnement (Tichy, 1985). Cependant, peu de travaux relatifs au domaine du génie logiciel ont porté leur attention sur l'évolution des descriptions des objets (évolution de schémas de objets), ceci en comparaison des nombreux travaux menés dans les bases de données classiques (Roddick, 1992) et plus récemment dans les bases orientées-objet (R.Zicari, 1989 Banerjee et al., 1987; Butterworth et al., 1991b). Cette évolution impacte fortement la gestion des processus logiciels car toute modification dans la description d'un objet (i.e. modification du schéma) a des retombées sur la manière dont les objets sont perçus et manipulés durant les activités de génie logiciel.

Une première approche pour faire évoluer le schéma est la notion de spécialisation/ généralisation d'un type permettant à ce dernier d'évoluer à l'intérieur d'une hiérarchie de types (Pedersen, 1989). Une approche plus générale est de permettre à un type d'évoluer "*arbitrairement*" (B.Zdonik, 1990).

Les différents mécanismes utilisés pour gérer l'évolution de schémas différents sont dépendants du modèle de données, la nature du domaine d'applications ainsi que des priorités retenues. Certains critères se retrouvent dans la plupart des systèmes, d'autres restent plus spécifiques au modèle de données et au domaine d'application. Cependant, l'un des problèmes majeurs dans la gestion de l'évolution de schéma réside dans la mise en correspondance entre les données persistantes de la base et les modifications apportées à leurs types; d'où le problème crucial de la ré-organisation de la base de données (Lerner and Habermann, 1990).

Dans un travail récent nous avons décrit comment ce type d'évolution est pris en compte dans Adèle (Estublier et al., 1993). Le modèle de données d'Adèle est utilisé par TEMPO, donc nous ne le décrivons pas ici.

IV.9.2.2 L'évolution du modèle de processus

Les processus 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 des processus logiciels 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 AGL 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 processus logiciels au cours de l'exécution des activités et gérer les incohérences entre les nouvelles définitions et les instances de processus logiciels 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...

IV.9.3 Deux axes pour supporter l'évolution des processus

Partant de l'architecture d'environnement TEMPO montrée dans la figure IV.1, nous avons identifié deux types d'évolution l'une complémentaire par rapport à l'autre :

1. une évolution, que nous appellerons structurelle, correspondant à l'évolution des schémas supportant le modèle de produit et le modèle de processus. Ce

type d'évolution correspond à la modification des informations liées à la définition des types.

La gestion de cette évolution, devenue classique, a fait l'objet de nombreux travaux principalement dans la communauté base de données avec l'apparition de systèmes intégrant cet aspect comme par exemple Adèle (Estublier et al., 1993), Gemstone (Butterworth et al., 1991b), Orion (Kim, 1988) et O2 (Adiba and Collet, 1993; Bancilhom et al., 1992).

De manière générale, il s'agit d'appliquer les techniques de versionnement aux types et de définir des règles plus ou moins draconiennes pour contrôler cette évolution. Ce genre d'évolution est dite (Conradi et al., 1992) d'évolution dure (*"hard"*).

2. une évolution des procédés de production que (Conradi et al., 1992) qualifie de légère (*"soft")*. Cette évolution est dynamique puisqu'on fait évoluer le processus en cours d'exécution. La principale différence par rapport à l'évolution dure est que la base de données n'a pas besoin d'être re-structurer pour prendre en compte les nouvelles descriptions. En effet dans ce type d'évolution le schéma de données et le modèle de processus logiciels existant n'est mise à jour que par l'ajout de nouvelles informations.

   Ce type d'évolution fait l'objet d'une attention particulière dans les travaux sur les AGL dits *dirigés par les processus logiciels*. Les travaux dans ce type d'évolution sont à leur premiers balbutiements (Conradi et al., 1993).

   Les premiers travaux sont apparus dans les systèmes de processus logiciels à base de règles, par exemple Marvel (Barghouti and Kaiser, 1992), Grapple (Huff and Lesser, 1988), Alf (Canals et al., 1993), Peace (Arbaoui, 1993) et Spell (Conradi et al., 1992). Ces systèmes essayent d'intégrer les résultats obtenus par la communauté d'Intelligence Artificielle en rapprochant ce type d'évolution des systèmes à règles. Il s'agit dans ce cas d'étudier la complétude du système à règles et la cohérence entre le modèle de données et les règles.

IV.9.4 Notre proposition

Dans TEMPO, les aspects évolution font partie intégrante du modèle de processus. En effet, nous pensons qu'un modèle orienté-objet pour les processus logiciels comme TEMPO, supportant les ***rôles*** des objets, intègre l'aspect évolution d'une part grâce aux

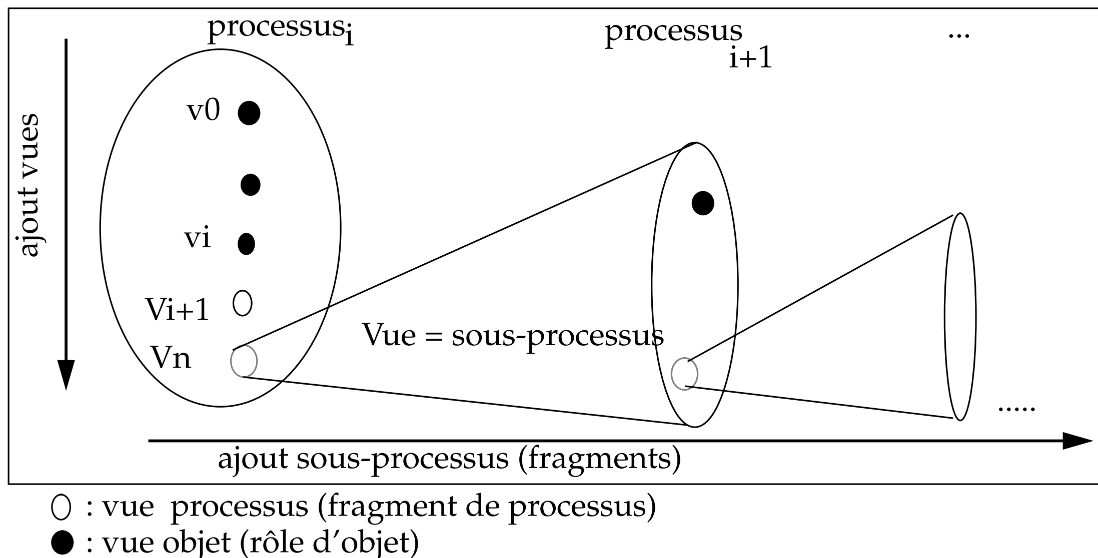
concepts du paradigme O.O. comme l'héritage multiple, la liaison retardée *("late-binding"),* le polymorphisme, et d'autre part parce que les effets de modification sont très localisés (Atkinson et al., 1989).

Les évolutions prises en compte concernent l' évolution des rôles et des sous-processus. En effet, un processus peut se voir ajouter, supprimer ou modifier des rôles de manière dynamique. De plus, la longue durée de vie des activités de développement de logiciels nécessite fréquemment d'ajuster le modèle de processus pour prendre en compte de nouvelles politiques de développement, par exemple la décomposition d'un processus en plusieurs sous-processus, l'unification d'un processus à partir de plusieurs sous-processus ou le changement dynamique de types de processus.

L'évolution dans TEMPO intervient à deux niveaux de granularité l'un étant complémentaire par rapport à l'autre (figure IV.2):

1. Au niveau du modèle de cycle de vie d'un objet. Cette évolution se fait par l'ajout incrémental des rôles de manière dynamique. Comme TEMPO a une approche dirigée par les objets, la sémantique d'une instance de processus 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 jour des rôles correspondants, entraîne une modification de la sémantique de cette instance.
2. Au niveau hiérarchie de processus. C'est l'évolution par le développement de la hiérarchie de sous-processus. A cause de la longue durée de vie des activités de développement de logiciels, il est nécessaire d'ajuster le modèle de processus pour prendre en compte de nouvelles stratégies de développement, de nouvelles connaissances, etc. Ce qui se traduit par la décomposition d'un processus en plusieurs sous-processus, l'unification d'un processus à partir de plusieurs sous-processus ou le changement dynamique de types de processus, etc.

Figure IV.2 Les différents axes d’évolution dans TEMPO



Nous analysons dans ce qui suit chacune de ces évolutions.

IV.9.4.1 Au niveau du modèle de cycle de vie d’un objet : ajout de rôles

Des travaux récents dans le domaine de base de données, comme par exemple Fibonnaci (Albano et al., 1993), ont utilisés également le concept de rôle pour modéliser le cycle de vie d’un objet logiciel. Dans TEMPO nous utilisons la même approche.
Par exemple :

> un objet logiciel représentant une configuration évolue conformément au cycle de vie : *spécification*, *construction*, *test* et *qualification*. Pour chacune des ces étapes, nous pouvons définir un rôle correspondant :
>
> **a.** dans le rôle *spécification*, sont définies les propriétés de la configuration;
>
> **b.** dans le rôle *construction*, on décrit comment la configuration doit être générée;
>
> **c.** dans le rôle *test*, on définit les procédés de mise au point de la configuration (compilation, “édition de liens” et test);
>
> **d.** et finalement dans le rôle *qualification*, on approuve la configuration pour sa mise à disposition aux utilisateurs.

Ceci pose le problème de l’intégration de nouveaux rôles correspondant à des situations non prévues à l’avance (situations d’exceptions). On peut également utiliser cette approche pour prendre en compte le développement incrémental du cycle de vie d’un

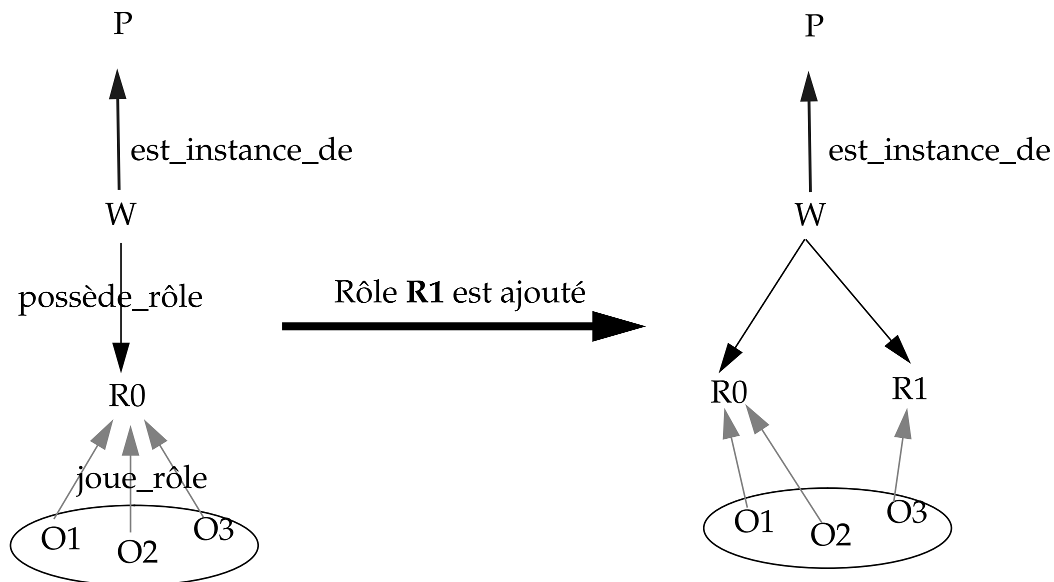
objet par la mise au point progressive des rôles, car un objet participant à une activité a une structure et un comportement conformes au rôle qu'il y joue. Ces rôles peuvent retracer les différentes étapes de l'évolution de l'objet.

Utilisant cette approche, les séquences de rôles d'un objet représentent les étapes de son cycle de vie. Les transitions d'une étape à l'autre sont gouvernées par des règles TECA définies explicitement par l'administrateur de l'environnement (par exemple, le rôle de qualification d'un objet représentant une configuration n'est ajouté que lorsque la configuration a été testée et approuvée).

Soit W une instance du type de processus logiciel **P** en cours d'exécution (figure IV.3). Les objets manipulés dans cette instance sont vus et possèdent un comportement conformes aux rôles qu'ils jouent dans **W**.

L'ajout d'un rôle **R1** dans **P** déclenche une modification de l'organisation des objets dans **W**, car certains objets qui jouaient auparavant le rôle **R0** dans **W** vont jouer le nouveau rôle **R1**.

Figure IV.3 Ajout de rôle.



Ce changement de rôle a deux effets sur l'évolution des processus :

1. un objet déjà existant (O3) et jouant un rôle donné (RO) dans le processus W aura un nouveau point de vue lorsqu'il aura changé de rôle (R1),

2. le comportement de l'objet (O3) dans le nouveau rôle (R1) devient différent, car de nouvelles méthodes et règles sont décrites dans R1.
3. Le passage d'un objet d'un rôle à un autre est dynamique; ce passage fait évoluer le processus logiciel car, d'une part, un nouveau type de rôle est dynamiquement ajouté à l'instance du processus logiciel et, d'autre part, l'objet qui change de rôle modifie son comportement. Pour contrôler ces évolutions, on utilise les règles TECA d'un type de processus logiciels.

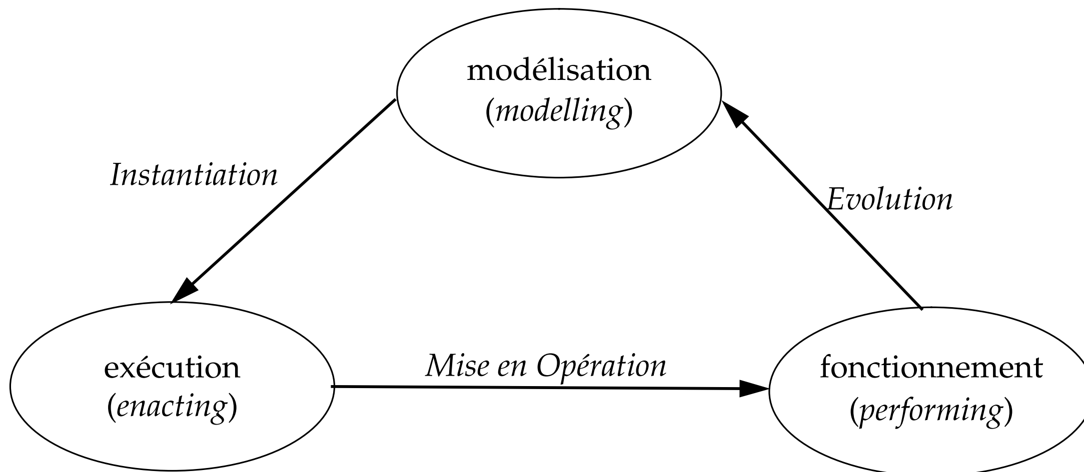
   Par exemple :

   Lorsque la configuration, jouant le rôle de spécification, se trouve dans l'état "*specified*" alors on exécute les opérations suivantes pour faire évoluer le modèle de processus:

   a. créer le type de rôle *construction* (modélisation)
   b. instancier le type de rôle dans le processus, en cours d'exécution, d'installation de la configuration (instantiation)
   c. transférer la configuration vers le nouveau rôle *construction* (exécution et mise en opération)
   d. continuer l'exécution du processus d'installation de la configuration (*fonctionnement*)

Nous utilisons ce schéma (modélisation exécution et fonctionnement des processus logiciels), initialement proposé en (Dowson, 1993) (mais sans support d'exécution), pour modifier la description des processus logiciels et les exécuter en prenant en compte leur évolution (figure IV.4). Les politiques de développement de logiciels et leur manière d'évoluer sont définies, d'une façon uniforme, par des opérations et des règles exprimées dans les types de rôles et dans les types de processus du même modèle de description. Ceci rejoint les récents travaux dans le domaine de la gestion de processus qui tentent d'offrir un cadre commun permettant de décrire les processus logiciels ainsi que leur évolution par un formalisme unique (Balzer, 1993; Narayanaswamy, 1993).

Figure IV.4 La méthode d'évolution de TEMPO.



IV.9.4.2 Au niveau processus logiciel : ajout de sous-processus

De manière similaire aux rôles, les instances de processus sont des unités dynamiques créées et détruites suivant un ensemble de règles. Dans notre approche, un objet logiciel et une instance de processus ont des caractéristiques en commun (attributs, méthodes et règles ECA). Cependant, contrairement aux objets, l'instanciation d'un processus se fait par une collection de rôles contrôlés par des contraintes associées à cette instance. Ainsi l'évolution d'une instance de processus logiciel se ramène dans notre cas à une évolution d'un objet agrégat dans une collection de rôles. On peut appliquer ainsi de manière récurrente le processus d'évolution pour chaque rôle de l'agrégat.

IV.9.5 Les opérations sur le modèle : le support à l'évolution

Dans les section suivantes nous présentons les opérateurs de manipulation des types de processus, des fragments processus et des types de rôles. Ces opérateurs permettent de mettre à jour un modèle spécifique de processus logiciels, et donc de le faire évoluer par ajout de nouvelles définitions.

Nous considérons qu'un modèle de processus logiciel est immuable. Pour le faire évoluer de nouvelles informations doivent être ajoutées. Nous ne pouvons ni supprimer ni altérer des informations, sauf dans le cas où les types supprimés n'ont pas d'instances.

IV.9.5.1 Ajout d'un type de processus logiciel

Pour permettre la création d'un type de processus logiciel nous définissons l'opérateur **mkprty(***ProcessName***,** *ListOfSuperProcess***)**. Cet opérateur permet la création d'un nouveau type de processus logiciel nommé *ProcessName* $\in \wp_n$. *ListOfSuperProcess* est

un ensemble où chacun de ses éléments, P_i, est un type de processus logiciel. Autrement dit *ListOfSuperProcess* doit respecter les règles d'héritage définies pour les types de processus logiciel.

Exemple :

1. **mkprty**(MonitorDesign, PROCESS);
2. **mkprty** (ModifyDesign, PROCESS);
3. **mkprty**(ReviewDesign, PROCESS);

Résultat :

1. MonitorDesign **ISA PROCESS; END_OF** MonitorDesign;
2. ModifyDesign **ISA PROCESS**; **END_OF** ModifyDesign;
3. ReviewDesign **ISA PROCESS**; **END_OF** ReviewDesign;

IV.9.5.2 Ajout d'un type de rôle

Pour permettre l'addition d'un type de rôle à un type de processus logiciel nous définissons l'opérateur **mkrty(***RName***,** *PName***,** *cons***,** *mode***,** *der***)**, tel que :

- *RName* $\in \Re_n$ est le nom du type de rôle.
- *PName* $\in \wp_n$ fournit le nom du type de processus logiciel dans lequel le rôle *RName* sera ajouté.
- *cons* est le constructeur du type de rôle, tel que :
  cons $\in$ *{ elementof, listof, setof}*
- *mode* fournit le mode d'utilisation pour les objets qui vont jouer ce rôle, tel que :
  mode $\in$ *{modifiable, visualizable, new }*
- *der* fournit la liste des types d'objets et des types de rôles qui composent le rôle *RName*. Ce paramètre est facultatif. Le type d'objet `Rset` est pris par défaut.
  `ObjTyList::={ObjectType | ObjectType «Expression»}*`

Exemples :

1. **mkrt**y(ccb, MonitorDesign, elementof, visualizable, [1, 1], [[spec,""]])
2. **mkrty** (under_design, ModifyDesign, setof, modifiable, [0, N],
   [[spec, "%role.ower=!userid"]])
3. **mkrty** (old_design, ModifyDesign, elementof,
   [1, N, [under_design, "%role.changes > 50"]])

Résultat :

```
MonitorDesign ISA PROCESS;
(1)      ROLE ccb;
             derived := spec
             mode := visualizable;
             cons := elmentof;
             card := 1, 1;
END_OF MonitorDesign;

ModifyDesign ISA PROCESS;
(2)      ROLE UnderDesign;
             derived := spec "%role.ower=!userid";
             mode := modifiable;
             cons := setof;
             card := 0, N;
(3)      ROLE OldDesign;
             derived := under_design "%role.changes >
50";
             mode := modifiable;
             cons := elementof;
             card := 1, 1;
END_OF ModifyDesign;
```

IV.9.5.3 Ajout d'un fragment dans un type de processus logiciel

Soit l'opérateur **mkprty**(*Fragment*, *ProcessType, ListOfProcessTypes, Card*). Cet opérateur permet d'ajouter un fragment, nommé *SousProcess* $\in \mathcal{F}_n$, dans un type de processus logiciel. *ProcessType* fournit le nom du type de processus logiciel dans lequel le type de sous-processus sera ajouté. *ListOfSuperProcess* fournit la liste des types de processus logiciels qui composent le type de sous-processus logiciel *SousProcess*. *Card* fournit la cardinalité maximale du type de sous-processus logiciel *SousProcess*.

Exemples :

1. **mkfty(** `Modify, MonitorDesign, ModifyDesign, 1`**)**
2. **mkfty(** `Review, MonitorDesign , ReviewDesign, N`**)**

Résulat :

```
ModifyDesign ISA design;...
ReviewDesign ISA design;...

Design ISA PROCESS;
    ...
(1)      CONTROL Modify;
             Fragment := ModifyDesign;
             card := 1;
(2)      CONTROL Review;
             Fragment := ReviewDesign;
```

```
                card := N;
    END_OF Design;
```

IV.9.5.4 Ajout d'un type d'attribut

L'opérateur **mkaty** permet de créer de nouveaux attributs aussi bien des attributs de rôles que des processus logiciels. Cet opérateur est défini ainsi :

```
mkaty(Aname, Pname, Rname, BasicType,
        Constructor, InitialValue)
```

tel que :

- `Aname` est le nom de l'attribut.
- `Pname` est le nom du type de processus dans lequel l'attribut `Aname` sera ajouté.
- `Rname` fournit le type de rôle appartenant au type de processus `Pname`.
- `BasicType` est le type de base pour l'attribut `Aname`, tel que : BasicTypeOrEnumerate ::= **integer** | **real** | **boolean** | **string** | **date**
- `Constructor` est le constructeur de type de l'attribut `Aname`, tel que : `Constructor` ::= **elementof** | **setof** | **listof**
- `InitialValue` fournit la valeur initiale de l'attribut `Aname`.

Exemples :

1. **mkaty(** `progress`**,**`MonitorDesign,elementof, [ok not_ok moreOrLess], ok`**).**
2. **mkaty(** `deadline, Modify, MonitorDesign, listof,date, NOW()+ 1 week)`
3. **mkaty(** `changes, ModifyDesign, UnderDesign, elementof,integer, 0)`

Résultat :

```
MonitorDesign ISA PROCESS;
    ATTRIBUTE
(1)       progress = ok not_ok moreOrLess := ok;
    CONTROL Modify;
          . . .
          ATTRIBUTE
(2)           deadline = listof date := now() + 1 week;
END_OF MonitorDesign;

ModifyDesign ISA PROCESS
    ROLE under_design;
          . . .
```

```
(3)      ATTRIBUTE
              changes = elementof integer := 0
END_OF ModifyDesign
```

IV.9.5.5 Ajout d'une méthode

L'opérateur **mkmet** permet d'ajouter un méthode dans le modèle. Cet opérateur est définie de la façon suivante :

mkmet (`Mname, Pname, Rname, FileName`)

tel que :

- **`Mname`** est le nom de la méthode.
- **`Pname`** fournit le type de processus dans lequel la méthode **`Mname`** sera ajoutée.
- **`Rname`** fournit le type de rôle appartenant au processus **`Pname`** dans lequel la méthode sera ajoutée. Si le rôle **`SELF`** est utilisé la méthode sera ajoutée dans le type de processus logiciel.
- **`FileName`** spécifie le nom du fichier contenant le corps de la méthode.

Exemple :

1. **mkmet(** `alert, ModifyDesign, self, alert.met`**)**;
2. **mkmet (**`countchanges, ModifyDesign, UnderDesign, changes.met`**).**

Résultat :

```
ModifyDesign ISA PROCESS;
     METHOD
(1)      alert;
     ROLE UnderDesign;
          . . .
          METHOD
(2)            countchanges;
END_OF MOdifyDesign;
```

IV.9.5.6 Ajout d'un événement

L'opérateur **mkenv**(*TriggerEventName*, *EventExpression*) permet d'ajouter un nouvel événement passible d'être utilisé par un règle. *TriggerEventName* fournit le nom d'événement tandis que *EventExpression* fournit une expression logique de première ordre.

Exemples :

1. **mkenv**(rpe, “!methode=replace_object”);
2. **mkenv**(rm, “!methode=remove_object”);
3. **mktemp**(was_rpe, “rpe FROM now()-2 weeks UNTIL now()”);
4. mktemp(was_removed, “rm FROM rpe UNTIL copied”).

Résultats :

```
EVENT rpe = [“!methode=replace_object”];
EVENT rm = [“!methode=remove_object”];
EVENT was_rpe = [“rpe FROM now()-2 weeks UNTIL now()”];
EVENT was_removed = [“rm FROM rpe UNTIL copied”];
```

IV.9.5.7 Ajout d’une règle temporelle événement-condition-action

L’opérateur **mktrigger** permet d’ajouter de nouvelles règles temporelles d’événement-condition-action aux types de processus logiciels et aux types de rôles. Cet opérateur est défini de la manière suivante :

mktrigger (`RuleName, ProcessName, Rolename, Mode, TriggerEvent, Method)`

tel que:

- `RuleName` fournit le nom de la règle.
- `ProcessName` est le nom du type de processus logiciel dans lequel la nouvelle règle sera ajoutée.
- `Rolename` est le nom de type de rôle appartenant au processus `ProcessName` dans lequel la nouvelle règle sera ajoutée. Si `Rolename` est égal à SELF, la règle sera directement ajoutée dans `ProcessName`.
- `Mode` fournit le mode de couplage de la nouvelle règle, tel que :
  `Mode ::= PRE | POST | AFTER | ERROR`
- `TriggerEvent` fournit le nom d’événement utilisé pour déclencher la règle.

Exemple :

1. **mkrg (**`alert,ModifyDesign, UnderDesign, pre, rpe bell)`
2. **mkrg (**`promote, ModifyDesign, UnderDesign, post, promoteevt, body.post)`

IV.9.5.8 Suppression d'un type de processus logiciel

rmpty (`Pname`)

où `Pname` est le nom du type de processus.

Cette opération supprime le type de processus `Pname` et tous les types de rôles lui étant associés.

Contrainte :

> *Extention(Pname) = {}*. C'est-à-dire que pour supprimer un type de processus nous imposons que ce type n'ait pas d'occurrence.

rmpty (`FragmentName, ProcessName`)

Cette opération permet de supprimer le fragment `FragmentName` du type de processus `ProcessName`.

Contrainte :

> $\forall P \in$ *Extention(Pname)* $\Rightarrow$ *IProcess(P, sub-process)* = { }. C'est-à-dire que nous ne pouvons pas supprimer un fragment s'il y a des occurrences des processus logiciels jouant le rôle fourni par ce fragment.

IV.9.5.9 Suppression d'un type de rôle

L'opérateur **rmrty (**`Rname,Pname`**)** permet de supprimer le type de rôle `Rname` du type de processus `Pname`.

Contrainte :

> $\forall P \in$ *Extention(Pname)* $\Rightarrow$ *IRole(P, Rname)* = { }. Autrement dit, un type de rôle ne peut être supprimé que s'il n'a pas d'objets qui jouent ce rôle.

IV.9.5.10 Suppression d'une méthode

L'opérateur **rmmethod**`(Mname, Pname, Rname` **)** permet de supprimer la méthode `Mname` du type de processus `Pname`. Lorsque `Rname` est différent de SELF, la méthode `Mname` du type de rôle `Rname` appartenant au type de processus logiciel `Pname` sera supprimée.

IV.9.5.11 Suppression d’une règle

Pour supprimer une règle temporelle événement-condition-action nous considérons l’opérateur **rmtrigger** (`RuleName, Pname, Rname`). Cette opérateur supprime la règle `Rname` du rôle `Rname` qui appartient au type de processus logiciel `Pname`. Si `Rname` est égal à `SELF`, cela implique que la règle `RuleName`, appartenant au type de processus logiciel, sera supprimée.

IV.9.5.12 Suppression d’une définition d’attribut

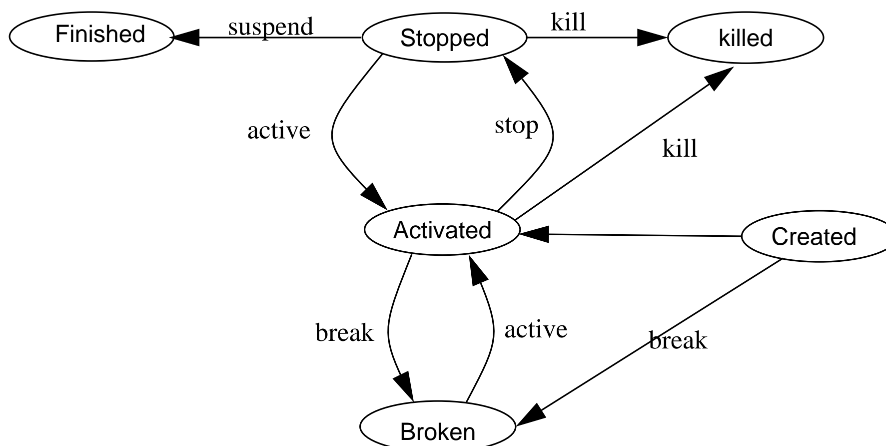
L’opérateur **rmaty**(`Aname, Pname, Rname`) permet de supprimer du type de processus logiciel `Pname` la définition de l’attribut `Aname`. Si `Rname` est différent de SELF, l’attribut `Aname` attaché au rôle `Rname` sera supprimé.

IV.10 Le graphe d’états des occurrences de processus logiciels

TEMPO gère un graphe d’état pour les occurrences de processus logiciel pour contrôler l’exécution des processus logiciels, comme par exemple Process-Wall (Heimbigner, 1992) et Softman (Mi and Scacchi, 1992).

Nous spécifions un ensemble d’états pour chaque rôle d’une occurrence de processus logiciel. Cela nous permet d’avoir une vision globale de l’état d’avancement des processus logiciels. Nous définissons ensuite la sémantique de chaque état.

Figure IV.5 Le graphe d’état.



IV.10.1 L'état "stopped"

L'état *"stopped*" indique que les activités d'une occurrence de processus logiciel ont été arrêtées volontairement. L'utilisateur responsable de l'exécution de l'occurrence de processus logiciel a décidé de l'arrêter pour une raison qui ne peut pas être maîtrisée ou qui n'offre pas d'intérêt à être modélisée par **TEMPO**.

Exemple :

> Dans un processus de modification de code, l'administrateur de ce processus peut demander au responsable de cette activité d'arrêter la modification en attendant une confirmation quelconque du client. Cette confirmation est de nature informelle et donc ne peut pas être prise en compte de manière automatique par le système. Pour permettre que ce genre d'intervention externe puisse être prise en compte, nous admettons qu'un processus puisse être arrêté sans une justification formelle, décrite dans le système.

Lorsque un rôle **R** appartenant à une occurrence de processus logiciel **P** est dans l'état "***stopped***", tous les objets jouant ce rôle ne seront pas habilités à recevoir des messages. Les messages sont rendus inactifs.Dans TEMPO, lorsqu'un rôle est dans un état ***"stopped"*** et qui il reçoit un message, un code d'erreur est retourné à l'objet qui a envoyé le message. En fonction de ce code erreur les procédures d'exécution sont propagées.

Les opérateurs suivants permettent de rendre actif ou inactif un ou plusieurs rôles appartenant à une occurrence de processus logiciel :

active(P,R). Cet opérateur permet de rendre actif le rôle **R** appartenant au processus **P**. Nous considérons un symbole spécial nommé **allroles** $\in \Re$. Si opérateur *active(P,allroles)* est utilisé, tous les rôles de **P** seront activés.

stop(P,R). Cet opérateur permet de rendre inactif le rôle **R** appartenant au processus **P**. L'opérateur *stop(P,allroles)* permet de rendre inactifs tous les rôles appartenant à l'occurrence de processus logiciel **P**.

active(P). Cet opérateur permet de rendre active l'occurrence de processus logiciel **P**.

stop(P). Inversement à l'opérateur précédent, cet opérateur permet de rendre inactif **P**.

IV.10.2 L'état *"killed"*

L'état *"killed"* indique que l'occurrence de processus logiciel ou les rôles sont été supprimés. Des opérateurs de suppression des occurrences de processus logiciels, des rôles et des objets jouant des rôles sont fournis. L'exécution de ces opérateurs est dépendante du contexte, c'est-à-dire que pour supprimer une occurrence de processus logiciel, une instance rôle ou un objet jouant un rôle, il faut d'abord se positionner dans une occurrence de processus logiciel :

kill(P, R, O). Cet opérateur permet de détruire l'objet **O** qui joue le rôle **R** dans une occurrence de processus logiciel **P**. Nous supposons l'existence d'un symbole **allobjets** $\in O_n$. Lorsque l'opérateur **kill(P,R,allobjets)** est utilisé, tous les objets jouant le rôle **R** dans **P** seront supprimés. Cependant l'instance de rôle **R** gardera son état courant.

kill(P, R). Cet opérateur permet de détruire le rôle **R** appartenant à l'occurrence de processus logiciel **P**. Lorsque l'opérateur **kill(P,allroles)** est utilisé toutes les instances de rôles de **P** seront détruites. La destruction d'une instance de rôle **R** implique la destruction de tous les objets qui jouent ce rôle-ci dans l'occurrence de processus logiciel **P**.

kill(P). Cet opérateur permet de supprimer l'occurrence de processus logiciel, par conséquent toutes les instances de rôles appartenant au processus **P** seront détruites. Certaines contraintes limitent la suppression d'occurrences des processus logiciels et des rôles.

a. une occurrence de processus logiciel **P** chargée de contrôler d'autres sous-processus logiciels ne peut pas être supprimée, sauf si tous ces sous-processus ont été déjà détruits ou achevés.

b. supposons qu'une instance de rôle **R** appartenant à l'occurrence de processus logiciel **P** fait référence à une autre occurrence de processus logiciel, c'est-à-dire que le rôle **R** représente un ou plusieurs sous-processus de **P**. Ce rôle ne pourra être détruit que si et seulement si tous les occurrences de processus référencées par **R** ont été détruites ou achevées auparavant.

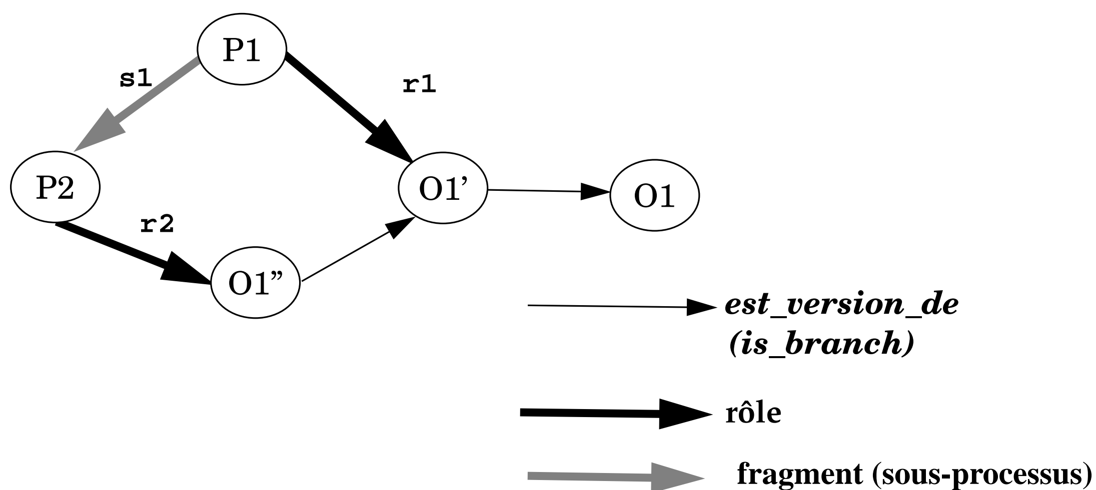
Ces contraintes ont été imposées pour aider à la gestion de l'exécution de processus logiciels. Les processus logiciels peuvent avoir une vie très longue. La destruction recursive de l'arbre des sous-processus logiciels est donc extrêmement dangereuse. Pour ce faire, l'utilisateur doit parcourir cet arbre, et détruire une par une les occurrences de processus logiciel.

IV.10.2.1 La suppression de rôles et la gestion de branches

La suppression des occurrences de processus logiciels et des instances de rôles a un impact direct sur la gestion de branches d’objets logiciels.
Supposons le scénario suivant :

> Une instance de processus logiciel **P1** composé d’un rôle, **r1** , et d’une instance de fragment **s1**. Le fragment **s1** représente un sous-processus logiciel de **P1** qui est occupé par le processus **P2**. Le rôle **r1** est occupé par l’objet **O1’** qui est à son tour une version (une branche) de l’objet **O1**. C’est-à-dire que l’objet **O1** joue le rôle **r1** dans le processus **P1**. En revanche le processus **P2** a un rôle, le rôle **r2**. Ce rôle est occupé par l’objet **O1”**. L’objet **O1”** est une version (une branche) de l’objet **O1’**, c’est-à-dire que l’objet **O1”** joue le rôle **r2** dans le processus **P2**. Ce scénario est illustré par la figure IV.6.

Figure IV.6 Les Rôles et la gestion de branches.

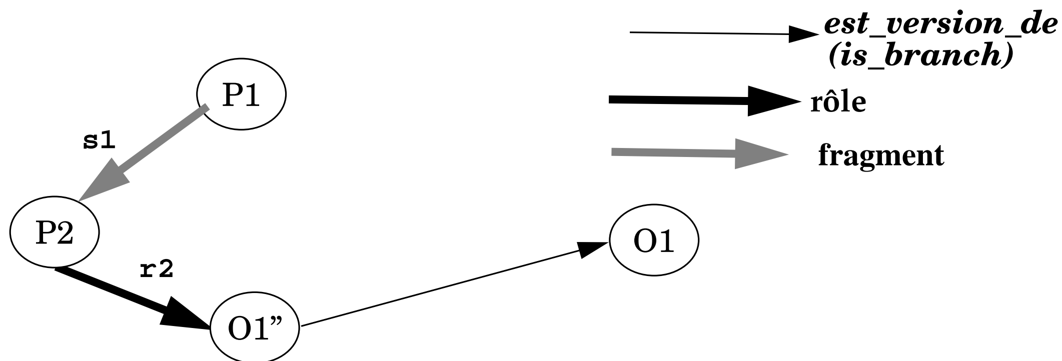


La destruction d’un rôle a pour effet de supprimer de manière successive tous ses composants et de reconstruire l’arbre de versions (branches) de tous les composants.

```
si kill(P,R) alors
    ∀ O ∈ IRole(P,R) ⇒ si origin(O) ≠ {} alors
        ∀ O’∈branch(O) ⇒ origin(O’) = origin(O) et remove(O).
```

Dans notre exemple la destruction du rôle **r1** a pour effet de supprimer l’objet **O1’** et de changer l’origine de l’objet **O1”**. La figure IV.7 montre le scénario après la destruction du rôle **r1**.

Figure IV.7 La suppression d’une instance de rôle.



IV.10.3 L’etat *“broken”*

L’état de défaillance (*“broken*”) indique que les activités d’une occurrence de processus logiciel ou d’une instance de rôle ont été arrêtées à cause d’un erreur d’exécution.

Les opérateurs suivants permettent d’arrêter et de remettre dans un état actif un ou plusieurs rôles dans un processus :

broken(P,R). Cet opérateur bloque l’exécution du rôle **R** appartenant à l’occurrence de processus logiciel **P**. Si opérateur **broken(P,allroles)** est utilisé toutes les instances de rôles de **P** seront bloquées.

repair(P,R). Cet opérateur débloque l’exécution d’un rôle **R** appartenant au processus **P**. Si opérateur **broken(P,allroles)** est utilisé toutes les instances de rôles de **P** seront débloquées.

IV.10.4 L’état *“sastified”*

L’état *“sastified”* indique que les activités d’une occurrence de processus logiciel ou d’une instance de rôle ont atteint leur fin. Dans ce cas tous les objets logiciels alloués par celui-ci sont libérés.

Les opérateurs suivants permettent d’arrêter l’exécution d’une occurrence de processus ou d’une instance de rôle.

suspend(P, R). Cet opérateur permet de suspendre l’exécution du rôle **R** du processus **P**. Si **supend(P, allroles)** est utilisé, toutes les instances de rôles appartenant à l’occurrence **P** seront suspendues.

supend(P). Cet opérateur permet de suspendre l’exécution de **P**. Il a le même effet que l’opérateur **supend(P, allroles)**.

Certaines contraintes limitent la suspension de l'exécution d'une occurrence de processus logiciel et de ses instances de rôles.

1. nous ne pouvons pas suspendre l'exécution d'un rôle qui représente un fragment de processus logiciel.
2. nous ne pouvons pas suspendre l'exécution d'un processus qui a des fragments de processus actifs.

IV.10.5 L'état *"created"*

Une occurrence de processus logiciel est dans l'état *"created"* lorsqu'elle a été créée. L'opérateur **create(P,T)** permet de créer une occurrence de processus logiciel **P** de type **T**.

L'opérateur **allocate(P,R,O)** permet d'allouer un ensemble d'objets **O** pour un rôle **R** appartenant à une occurrence de processus logiciel **P**. Lorsque l'opérateur **allocate(P,allroles,O)** est utilisé la liste d'objets **O** sera allouée à ***P*** de la manière suivante:

```
Pour chaque objet O′ ∈O
    pour chaque rôle R′  ∈ RoleList(P) alors
         pour chaque type T′ ∈ RType(P,R′) alors
              si Otype(O′) = T′ alors
                    allocate(P,R′,O′)
                    fin_de_bouclage
         fin_de_bouclage
```

IV.10.5.1 La création de processus fils

Une activité complexe ou de taille importante peut être décomposée en un arbre d'occurrences de processus logiciels. Pour permettre la création d'une occurrence de sous-processus logiciel, nous définissons l'opérateur **create(*P,T,S*,F)**. Cet opérateur crée l'occurrence de processus logiciel ***P*** de type ***T.*** Cette occurrence va être attachée au processus logiciel ***S*** comme un fragment de processus ***F***. Autrement dit, ***P*** a son exécution contrôlée par l'occurrence de processus logiciel ***S.***

Un processus fils ne peut manipuler qu'un sous-ensemble des objets manipulés par son père. Dans le chapitre suivant nous montrons avec plus de détails comment les occurrences de processus logiciels peuvent être imbriquées par une hiérarchie des environnements de travail.

IV.11 Conclusion

Nous avons consacré ce chapitre aux problèmes ayant motivé la conception de TEMPO ainsi que les solutions que nous avons adoptées. Les problèmes abordés sont :

1. une description uniforme des activités de génie logiciel;
2. les rôles des objets dans le cycle de vie d'un produit logiciel;
3. la gestion de contraintes temporelles;

TEMPO (Belkhatir and Melo, 1992b) fournit un formalisme exécutable pour la description et l'exécution des modèles de processus de production de logiciel. C'est un modèle fondé sur l'approche orientée-objet étendue par l'aspect multi-comportamentaux des objets. Cet aspect se rencontre dans le développement de systèmes complexes et de taille importante caractérisés par la présence de plusieurs agents travaillant sur des objets partagés utilisant des représentations et de multiples stratégies de développement.

Pour prendre en compte la description et le contrôle des activités de longue durée nous avons fourni les règles temporelles événement-condition-action. Notre principal apport par rapport à d'autre systèmes est le fait que nous pouvons travailler avec l'historique des objets. Nous utilisons également les règles TECA pour résoudre le problème d'ordonnancement d'activités dans un processus logiciel. D'autre systèmes comme par exemple ALF (Derniame et al., 1992) fournissent le concept de contraintes d'ordonnancement. Ces contraintes permettent d'exprimer de contraintes sur l'ordre d'exécution des activités dans une instance de processus logiciel. Cependant, ALF ne prend pas en compte les contraintes sur les activités qui ont été exécutées dans le passé. Cela est du au fait que ces systèmes ne possèdent pas d'une base d'objets permettant de tracer l'évolution des objets dans le temps.

CHAPITRE V TEMPO : Un Modèle pour la Programmation Coopérative 137

V.1 Introduction 137

V.2 Motivation 138

V.3 Le modèle “check-in/check-out” 138

V.4 Notre proposition : les environnements de travail 139

V.4.1 Exemple de partage 140

V.4.1.1 La gestion des branches 141

V.4.1.2 La décomposition d’un environnement de travail 142

V.4.1.3 L’unification des environnements de travail 144

V.4.2 Le protocole de communication 145

V.4.2.1 La coopération 146

V.4.2.2 La synchronisation 146

V.4.3 Notre proposition pour supporter la communication 147

V.4.3.1 Les contraintes de mise en connexion 148

V.4.3.2 Les contraintes de rupture 149

V.4.3.3 Les règles de contrôle de collaboration 150

V.4.3.4 Définition 151

V.5 Conclusion 152

CHAPITRE V TEMPO : Un Modèle pour la Programmation Coopérative

V.1 Introduction

Les problèmes dans le développement de logiciels de taille importante sont actuellement biens identifiés. On peut les classer en programmation détaillée (DeRemer and Kron, 1976), globale (Romamoorthy, 1986) et coopérative (Floyd et al., 1989). Nous désignons par programmation détaillée, les activités de développement liées à une personne développant seule un module ou programme, par programmation globale, les activités de développement liées à la multiplicité des composants et la programmation coopérative les activités de développement logiciel impliquant plusieurs agents. Les travaux de recherche se sont focalisés depuis longtemps sur le premier aspect avec le développement d'environnements de programmation puis le second aspect avec la gestion de versions et configurations. Plus récemment avec le développement de travaux sur les environnements de développement centrés processus, la programmation coopérative a été identifiée comme un domaine majeur devant être pourvue en concepts, mécanismes et outils.

Après le développement d'Adèle (Belkhatir et al., 1991) comme environnement pour supporter la programmation globale avec une plateforme d'intégration de composants multiples et versionables, nous avons initié le projet TEMPO (Belkhatir and Melo, 1992b) pour prendre en compte les stratégies de production et d'évolution de ces composants mettant en oeuvre une équipe importante. Dans le chapitre précédant nous avons montré comment TEMPO permet de décrire différents modèles de processus logiciels et de supporter les rôles des objets (Belkhatir et al., 1993). Dans ce qui suit nous présenterons les aspects de TEMPO liés à la coopération mettant en exergue les deux dimensions suivantes (Belkhatir and Melo, 1993):

1. Coordination des ressources. Ce sont les problèmes de répartition de ressources partagées par une même équipe. Nous allons montrer comment

TEMPO supporte les activités de longue durée par les environnements de travail. Un environnement de travail (ET) est une instance d'un type de processus logiciel. L'ET est donc une unité de regroupement de rôles fortement liés par un degré de communication.

2. Coopération des agents partageant un modèle de processus logiciel commun. Nous introduisons et développons le concept de *connexion active et programmable* comme support à l'expression des politiques de coopération et de synchronisation.

V.2 Motivation

Contrairement aux applications des base de données relationnelles, où la cohérence des objets doit être toujours assurée par le système, dans le cadre du génie logiciel où les activités ont une longue durée, il est difficile d'imposer que la cohérence soit toujours vérifiée au cours de l'exécution des processus logiciels (Balzer, 1991). Cette incohérence est dûe au fait que différentes activités peuvent partager un même objet durant une longue durée. Un AGL doit gérer l'incohérence afin de permettre le travail coopératif et parallèle dans toutes les étapes du cycle de vie des logiciels (Godart, 1993a; Schwanke and Kaiser, 1988).

Pour gérer la cohérence (ou incohérence!) des objets partagés, il est nécessaire de fournir des mécanismes permettant de coordonner les utilisateurs qui partagent ces objets. Généralement, la cohérence est assurée par le concept d'atomicité des transactions. La coordination est également prise en compte par la mise en série de ces transactions. Dans le domaine du génie logiciel, bien que ce genre de mécanisme soit également nécessaire pour assurer la cohérence de la base d'objets, il ne fournit pas une solution suffisante, car nous nous trouvons dans un contexte où de nombreuses activités concurrentes partagent des objets durant une longue durée.

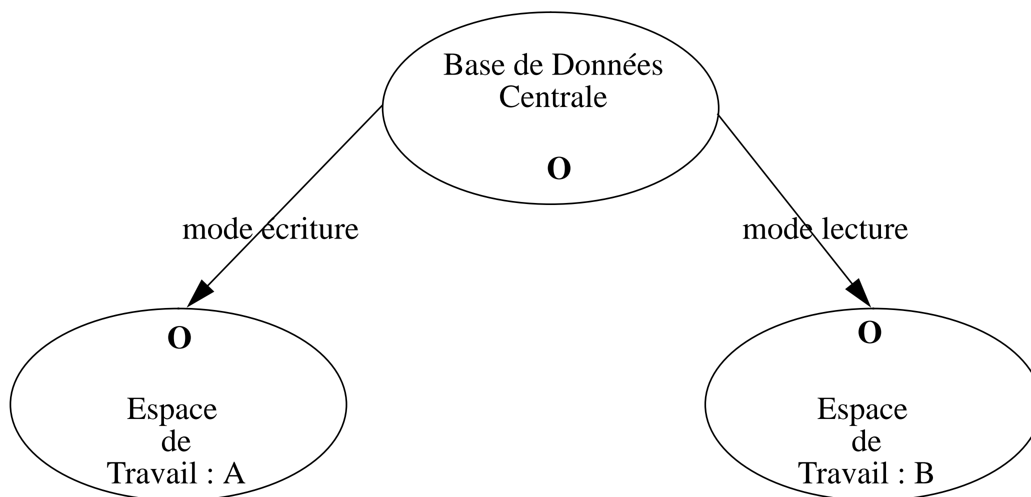
V.3 Le modèle "*check-in/check-out*"

Plusieurs travaux ont été faits dans le domaine des AGL, afin de fournir un cadre de travail pour supporter les transactions longues (Barghouti, 1992; Courington, 1989; Kim et al., 1991; Sarkar and Venugopal, 1991b). Ces travaux ont proposé des modèles pour les transactions longues voisins du modèle *"check-in/check-out"* (Dittrich et al., 1987; Feiler, 1991a; Godart, 1993b; Kaiser, 1990). Dans ce modèle, les objets partagés sont extraits de la base de données centrale et sont mis à la disposition des utilisateurs dans leurs espaces de travail respectifs. Dans son espace de travail, l'utilisateur peut modifier

l'objet partagé sans entrer en conflit avec les autres utilisateurs de l'environnement qui peuvent continuer à travailler en consultant la version disponible dans la base de données centrale.

La figure V.1 montre deux espaces de travail (A et B) partagant un même objet (Objet O) dans le modèle *"check-in/check-out"*. Seulement l'espace de travail A a le droit d'écriture sur l'objet O. Tous les autres espaces de travail ne peuvent que consulter la copie de l'objet O qui est dans la base de données centrale ou publique. Lorsque l'espace de travail A finit ses activités, l'objet O est remis dans la base de données et la protection sur cet objet est retiré permettant ainsi qu'un autre espace de travail puisse mettre à jour cet objet.

Figure V.1 Le modèle *"check-in/check-out"*.



V.4 Notre proposition : les environnements de travail

Pour chaque occurrence de processus logiciel, TEMPO fournit un environnement de travail dans lequel les activités sont exécutées, et les objets modifiés par l'application des outils automatiques (par exemple, des compilateurs) ou interactifs (par exemple, des éditeurs de texte), etc. (Belkhatir and Melo, 1993).

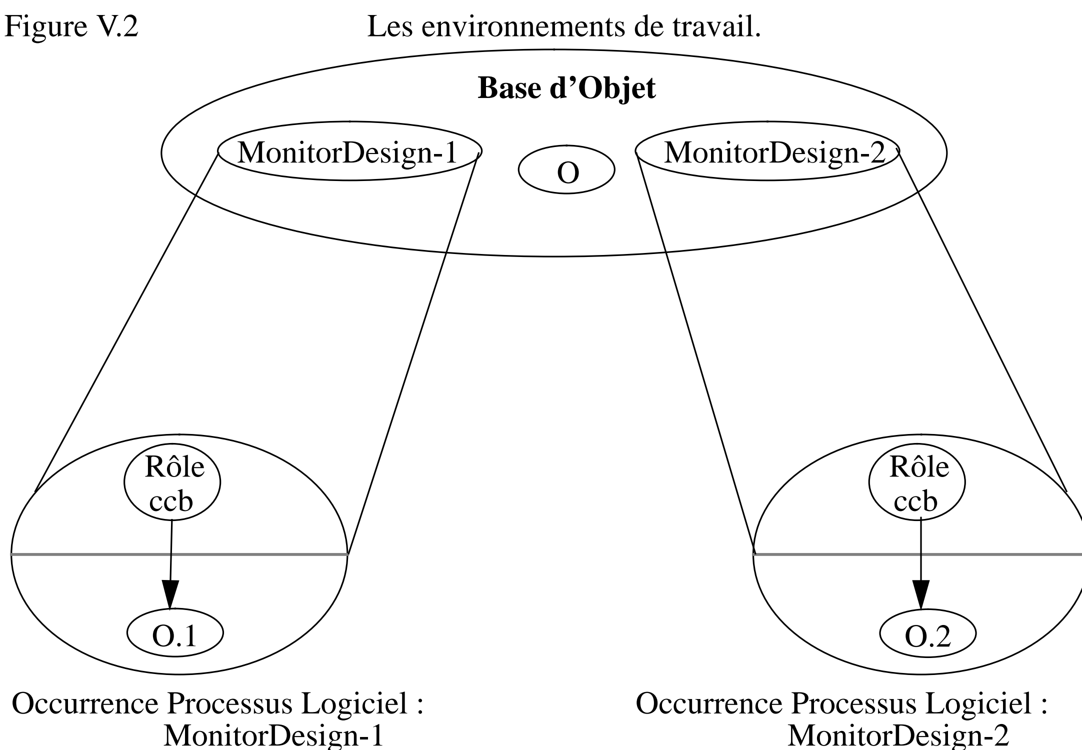
Un objet partagé par de multiples environnements de travail peut être modifié dans chaque environnement de travail où cet objet est utilisé. Nous gérons ces copies multiples en utilisant les versions d'un même objet logiciel. Lorsqu'un objet partagé est la cible d'une modification, une nouvelle version de cet objet est créée et mise à la disposition de l'utilisateur dans l'environnement de travail où la modification a été demandée. La modification est accomplie sur la nouvelle version de l'objet et non sur l'objet source. Cette nouvelle version a une durée de vie limitée à la durée de vie de

l'environnement de travail auquel elle appartient.

Nous imposons que ces transactions soient accomplies de manière hiérarchique, comme cela est décrit en (Feiler, 1991a; Godart, 1993b; Kaiser, 1990). Donc, lorsque deux environnements de travail veulent partager un même objet afin de le modifier parallèlement, il faut que ces deux environnements utilisent le même objet racine.

V.4.1 Exemple de partage

Figure V.2 Les environnements de travail.



La figure V.2 montre un exemple de partage d'un objet logiciel. L'objet **O** est partagé entre les occurrences de processus logiciels **MonitorDesign-1** et **MonitorDesign-2**, où il joue le rôle **ccb**. Dans les environnement de travail de ces deux occurrences, l'objet **O** en fait les versions **O.1** et **O.2** peuvent être soumises à des mises à jour. Ces mises à jour ne sont pas propagées. C'est-à-dire que l'objet **O** de l'occurrence **MonitorDesign-1** peut être modifié sans que ces modifications viennent affecter les activités menées dans l'occurrence **MonitorDesign-2** et vise-versa. Pour que cela soit possible, une version de l'objet **O** est automatiquement créée et mise à disponibilité pour chaque occurrence lorsqu'une mise à jour est demandée sur cet objet. L'alternative créée est réservée à l'espace de travail appartenant à l'occurrence de processus logiciel. Dans la figure V.2, les alternatives **O.1** et **O.2** d'objet **O** sont respectivement privées aux occurrences **MonitorDesign-1** et **MonitorDesign-2**.

Lorsqu'une version est créée et mise à disposition d'un environnement de travail elle hérite toutes les caractéristiques de l'objet source. Les attributs et le contenu de la version sont donc les mêmes que ceux de l'objet source. Une fois dans l'environnement de travail, la version peut être modifiée, créant des révisions. Les attributs de la version peuvent également être mis à jour localement.

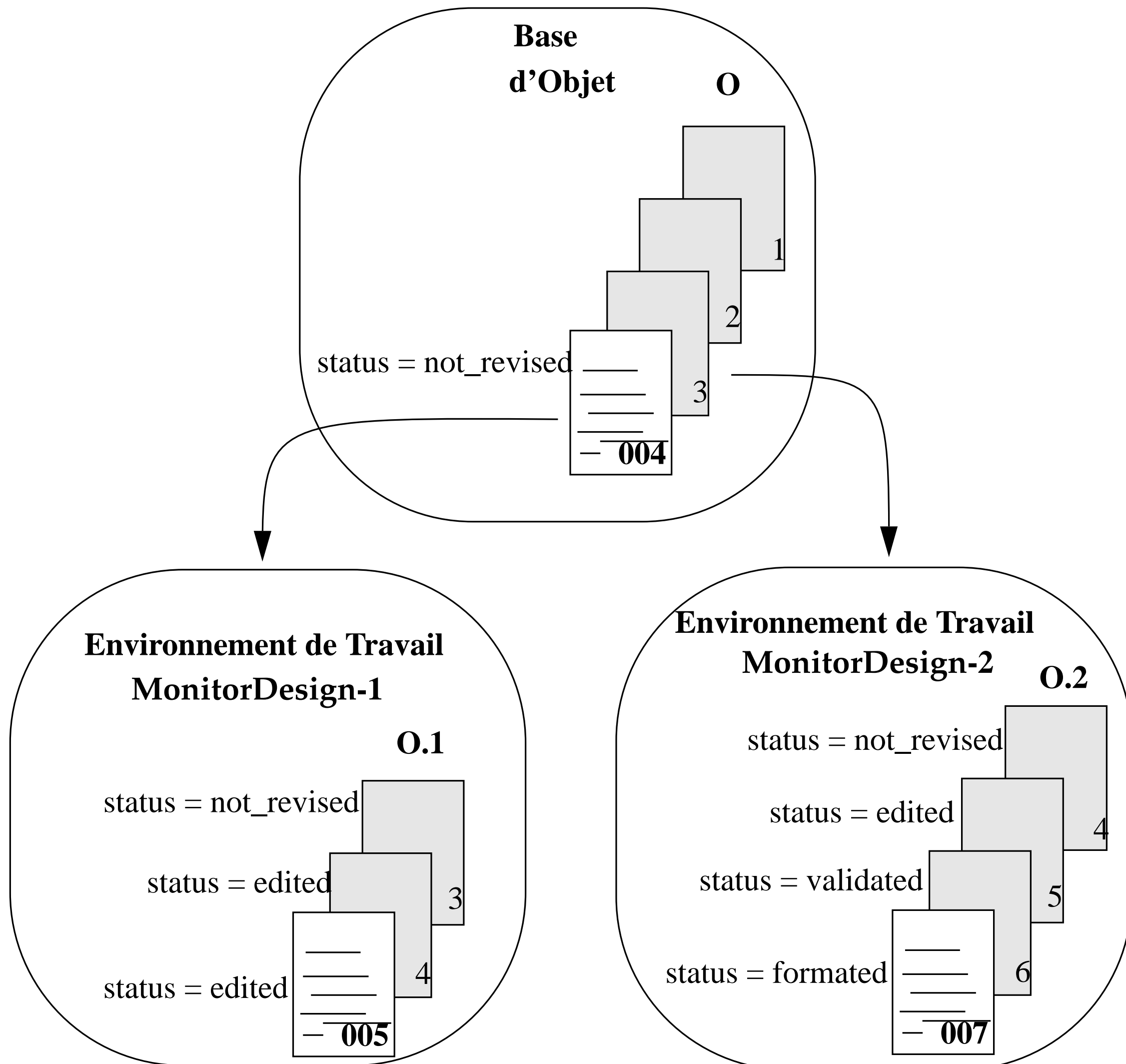
V.4.1.1 La gestion des branches

La figure V.3 montre un scénario où un objet logiciel **O** est partagé par deux environnement de travail, **MonitorDesign-1** et **MonitorDesign-2**. L'objet source stocké dans la base de données a trois révisions. L'attribut **`status`** de la dernière révision est égal à **`not_revised`**. Lorsque cet objet est modifié dans le contexte de l'environnement de travail **MonitorDesign-1**, une alternative (**O.1**) local à cet environnment est créée. Dans cet exemple, l'alternative de l'environnement **MonitorDesign-1** hérite du contenu de la dernière révision ainsi que de l'attribut **`status`** avec sa valeur. Dans l'environnement de travail **MonitorDesign-1**, les valeurs des attributs ainsi que le contenu de l'alternative **O.1** de l'objet **O** peuvent être changés en dérivant de révisions lorsque que le contenu de l'objet **O.1** est mis à jour.

De manière similaire, une alternative de l'objet **O** est également produit et mise à disposition de l'environnement de travail **MonitorDesign-2** lorsqu'une mise à jour est demandée par l'utilisateur responsable de cet environnement. De la même façon que pour l'environnement de travail **MonitorDesign-1**, l'alternative **O.2** de l'objet **O** peut évoluer indépendamment de l'alternative **O.1** et de l'objet source **O**.

Cette solution permet à un ensemble d'utilisateurs de partager un ensemble d'objets logiciels. Chaque environnement de travail appartient à un type de processus logiciel; les activités qui se déroulent dans l'environnement de travail sont donc contrôlées par les descriptions fournies dans ce type.

Figure V.3 La gestion de branches.



V.4.1.2 La décomposition d'un environnement de travail

Un environnement de travail peut se décomposer successivement en plusieurs sous-environnements de travail en respectant les règles de décomposition décrites dans les types de processus logiciel.

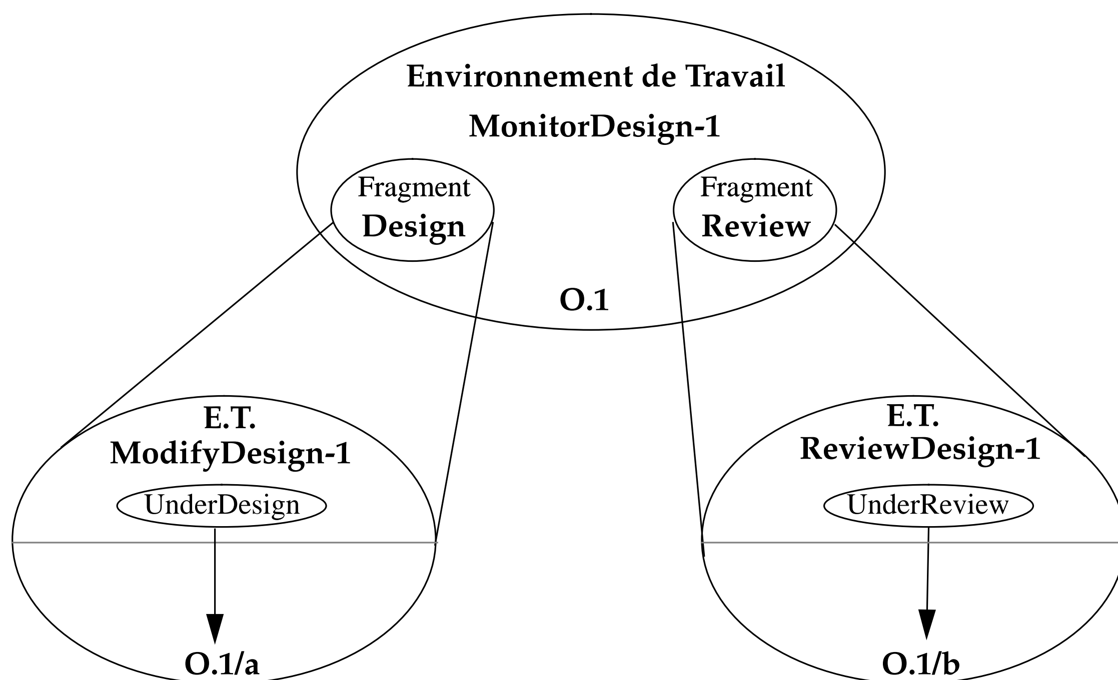
- Chaque environnement de travail fils n'est autorisé à manipuler, au départ, qu'un sous ensemble d'objets de son environnement de travail père.
- Chaque environnement de travail fils correspond à une instance d'un fragment défini dans le type de processus logiciel.

La figure V.4 montre un exemple où l'environnement de travail **MonitorDesign-1** est décomposé en deux sous-environnements de travail.

1. l'environnement de travail **ModifyDesign-1** est une instance du type de processus logiciel **ModifyDesign**. Cet environnement de travail est fils de l'environnement de travail **MonitorDesign-1**. C'est-à-dire que **ModifyDesign-1** correspond donc au fragment **Design** appartenant au type MonitorDesign.

2. De même, l'environnement de travail **ReviewDesign-1** est une instance du type de processus logiciel **ReviewDesign**. Cet environnement de travail correspond au fragment **Review** défini dans ce type.

Figure V.4 La décomposition des environnements de travail.



Les environnements de travail sont donc des instances des types de processus logiciels. Les environnements de travail fils correspondent également aux fragments décrits dans ces types. Ainsi, la décomposition d'un environnement de travail en un ou plusieurs environnements de travail fils est gérée selon les définitions de fragments faites dans les types de processus logiciels.

En conséquence de cette décomposition les sous-environnements **Design** et **Review** ne peuvent manipuler que l'alternative **O.1** de l'objet **O**. Une alternative est alors créée pour chaque sous-environnement lorsqu'une opération de mise à jour est faite dans le contexte de ces sous-environnements.

Selon la procédure de création d'alternatives décrite au paragraphe précédent, les nouvelles alternatives **O.1/**a et **O.1/b** héritent des attributs valeurs et du contenu de l'objet **O.1**.

V.4.1.3 L'unification des environnements de travail

Lorsque l'utilisateur le désire, les modifications effectuées sur les objets dans un environnement de travail fils peuvent être exportées vers l'environnement de travail père. Cette solution pose un problème d'intégration des résultats entre les environnements de travail. C'est-à-dire que lorsque les utilisateurs essayent de propager les modifications effectuées sur les objets partagés, des incohérences peuvent apparaître. A la différence des approches classiques, où une politique est en générale imposée pour prendre en compte ce genre de problèmes (Miller, 1989), nous adoptons dans TEMPO une solution dans laquelle l'utilisateur définit lui même comment réagir dans une situation de conflit. En utilisant les règles temporelles événement-condition-action, il peut décrire quelle politique d'intégration l'AGL doit suivre.

Supposons que l'environnement de travail **MonitorDesign-1** soit une occurrence de type processus logiciel **`MonitorDesign`** et que l'environnement de travail **ModifyDesign-1** et **ReviewDesign-2** soient respectivement des occurrences de type processus logiciel **`ModifyDesign`** et **`ReviewDesign`**. Comme montré dans la figure V.4 les environnements de travail **ModifyDesign-1** et **ReviewDesign-2** sont respectivement contrôlés par les fragments **Design** et **Review** appartenant au type de processus logiciel **MonitorDesign**. Supposons que l'environnement **ModifyDesign-1** a été créé pour modifier la spécification du flux de données de la conception d'un module et que **`ReviewDesign`** est crée pour mettre à jour cette spécification.

En utilisant les règles temporelles événement-condition-action définies dans les types de rôles et dans les types de processus logiciels, il est possible de décrire la politique d'intégration des résultats pour être utilisée par les environnement de travail.
Exemple :

```
design_documment ISA objet;
    ATTRIBUTE
          status = designed, reviewed,approved, none := none;
          no_of_changes = INTEGER := 0;
END_OF design_documment;

ModifyDesign ISA PROCESS;
    ROLE UnderDesign;
          derived_from = design_document;
END_OF ModifyDataFlowDesign;

ReviewDesign ISA PROCESS;
```

```
    ROLE UnderReview;
         derived_from = design_docummment;
END_OF ModifyControlFlowDesign;

MonitorDesign ISA PROCESS;
    RULES
         WHEN promote UPON ccb
         AND PAST
              last(%Modify/UnderDesign.status == designed) <
              last(%Review/UnderReview.status == reviewed)
         DO  change_attr(%ccb/status,approved);
--
-- la fonction last retourne la date de la dernère fois
-- dans l'historique que l'attribut "status" du
-- document "design" a été modifié.
--
    CONTROL Modify;
         Fragment = ModifyDesign;
    CONTROL Review;
         Fragment = ReviewDesign;
    ROLE ccb;
         derived_from = design_docummment;
END_OF MonitorDesign;
```

Dans le type MonitorDesign, une règle est définie pour contrôler l'unification des résultats de l'activité de modification de la spécification du flux de données et de l'activité de mise au point de cette modification. Cette règle impose la contrainte suivante :

> Le document de conception sera considéré comme cohérent (***"approved"***) par l'E.T. **MonitorDesign-1** si et seulement si le document de conception a été modifié (***"designed"***) dans l'environnements de travail **ModifyDesign-1** et validé (***"reviewed"***) par l'environnment de travail **ReviewDesign-1**.

V.4.2 Le protocole de communication

Les activités de génie logiciel sont caractérisées par une forte demande de coordination, collaboration et synchronisation, car les objets logiciels sont partagées par de multiples utilisateurs. Un des problèmes se situe au niveau du contrôle des objets partagés. Par exemple, des questions comme celles ci-dessous doivent être maîtrisées par l'AGL :

- Quand, pourquoi et par qui un objet a été changé?
- Comment et quand ces changements doivent être diffusés aux utilisateurs qui partagent cet objet?
- Quels sont les effets causés pour ce changement?
- Dans quels cas les modifications doivent être acceptées ou interdites?, etc.

Ces problèmes ont fait l'objet de nombreuses études dans différents domaines de recherche, plus particulièrement dans le domaine des bases de données, et du travail coopératif et bien sûr du génie logiciel. Pour les résoudre, de nombreux mécanismes ont alors été proposés. Dans le sous sections suivantes nous montrons comment ces problèmes ont guidé notre recherche et quelles sont les solutions fournies par TEMPO pour les résoudre.

V.4.2.1 La coopération

Pour permettre l'échange d'informations entre les utilisateurs, il est nécessaire de fournir des mécanismes qui assistent et stimulent la collaboration entre ceux-ci. L'environnement doit fournir un protocole de communication par lequel les utilisateurs sont notifiés du déroulement des activités menées dans l'environnement. Grâce à ces notifications, les utilisateurs peuvent savoir à quel moment ils doivent échanger leurs informations et avec qui ils doivent le faire, c'est-à-dire quand et à quelles conditions ils doivent collaborer. Cela est uniquement possible lorsque chaque utilisateur peut être averti de l'état d'avancement des processus logiciels menés par ses collègues dans l'environnement.

Au cours de l'exécution de processus logiciels, nous pouvons avoir un nombre très important de processus logiciels se déroulant en parallèle. Chaque utilisateur doit alors choisir sur quels événements il doit être averti. L'AGL doit permettre à chaque utilisateur de spécifier, en fonction de ce qu'il est chargé d'accomplir, quels sont les événements importants pour l'achèvement de ses activités. En prenant en compte la notification, l'utilisateur peut donc entrer dans le processus d'échange d'informations et collaborer. Le processus de collaboration est donc divisé en trois étapes : notification, décision et échange d'information. Chacune des ces étapes doit être supportée par l'AGL.

V.4.2.2 La synchronisation

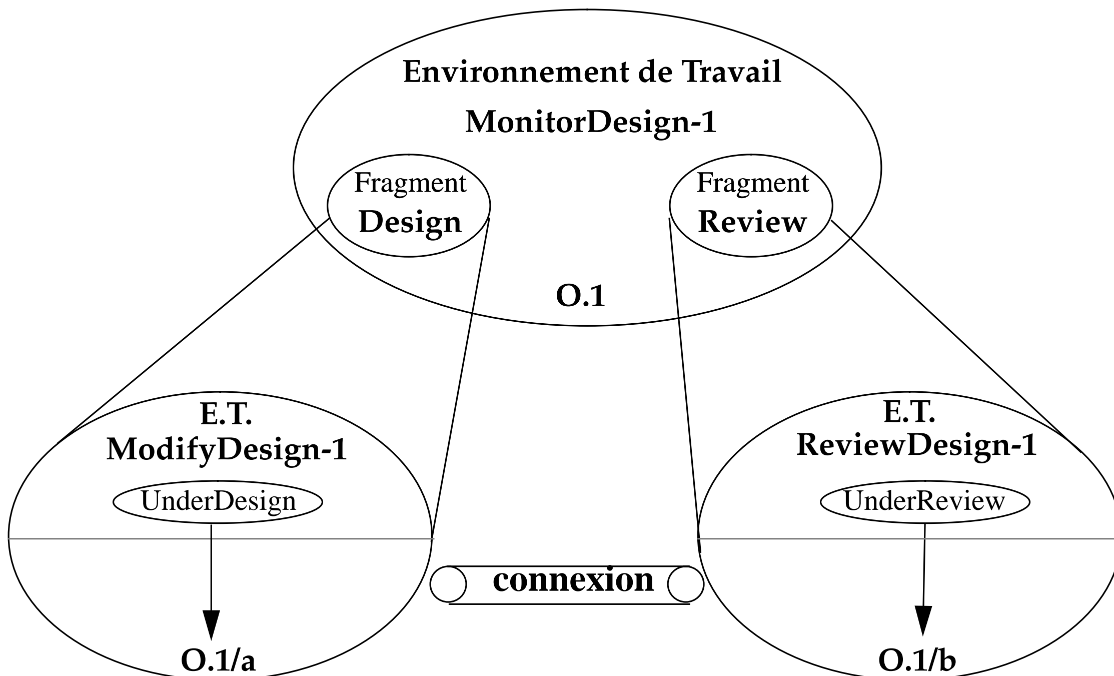
Pour que la communication entre les utilisateurs d'un AGL se fasse de manière contrôlée ceux-ci doivent pouvoir se synchroniser au cours du développement de leurs activités. Sans un mécanisme qui assiste la synchronisation entre les utilisateurs, l'AGL ne peut pas assurer que ces utilisateurs échangent correctement leurs résultats. Dans un cadre où les activités ont une longue durée, il est donc nécessaire que les utilisateurs se synchronisent au cours du développement de leurs activités afin que les résultats obtenus puissent être intégrés. Si l'AGL ne contrôle pas la synchronisation des activités parallèles, les résultats peuvent arriver à un tel niveau de divergence qu'il devient impossible de les intégrer.

Supposons par exemple que deux activités de longue durée, **A1** et **A2**, s'exécutent en parallèle dans l'environnement. Supposons encore que ces deux activités modifient parallèlement un même objet, **O**. Si ces deux activités ne se synchronisent pas au cours de leur exécution, on risque d'arriver à un point où les modifications effectuées sur l'objet **O** deviennent impossibles à intégrer. L'AGL doit alors supporter la synchronisation entre les activités de longue durée ainsi que contrôler cette synchronisation.

V.4.3 Notre proposition pour supporter la communication

Nous trouvons de multiples travaux concernant la communication, la collaboration, la coordination et la synchronisation dans un AGL. Notre attention s'est focalisée sur la description des politiques de coordination, collaboration et synchronisation entre de telles activités. Pour cela, nous avons fourni le concept de *connexion* par lequel le protocole de communication entre les occurrences de processus logiciel peut être décrit, permettant à ces activités de se synchroniser pour augmenter le niveau de coopération et de collaboration entre les utilisateurs de TEMPO (Belkhatir and Melo, 1993).

Figure V.5 L'échange de messages par les connexions.



Les connexions sont utilisées pour permettre à deux occurrences de processus logiciels de se synchroniser au cours de leurs exécutions. Les connexions sont donc un canal de communication entre deux occurrences. La figure V.5 en montre un exemple pour les occurrences **WE-1/a** et **WE-1/b**. En utilisant cette connexion, ces deux occurrences pourront échanger des messages au cours de leurs exécutions.

A travers une connexion, un processus logiciel peut synchroniser la diffusion de ses résultats avec un autre processus qui n'est pas nécessairement un fils ou un père. Les connexions permettent donc à un processus logiciel d'être averti de l'état de déroulement d'autres processus, autorisant ainsi qu'un processus réagisse à des événements provoqués par d'autres processus. Par exemple, une mise à jour de l'objet **O1.b**, dans l'environnement de travail **ReviewDesign-1** peut déclencher des opérations dans l'environnement **ModifyDesign-1**, car ces deux occurrences de processus logiciels sont connectées. Comme les règles TECA peuvent être utilisées pour répondre à ces événements, les connexions peuvent donc être utilisées pour supporter la communication entre deux ou plusieurs processus logiciels.

Pour décrire les politiques d'échange de messages dans un modèle de processus logiciel nous fournissons le concept des *connexions actives et programmables*. Un type de connexion a le style suivant :

```
designing ISA CONNECTION;
    DOMAIN
         ModifyDesign:UnderDesign ->
         ReviewDesign:UnderRevision;

    PLUG-ON-RULES . . .
    ACTIVE-RULES . . .
    PLUG-OFF-RULES . . .
END_OF designing;
```

Le domaine d'une connexion est fourni par la clause **DOMAINE** dans le type de connexion. Les connexions sont toujours binaires, c'est-à-dire qu'elles servent à lier un processus logiciel avec un autre. Le niveau de granularité d'une connexion est celui des rôles. Un type de connexion décrit les politiques de connexion entre un rôle d'un type de processus logiciel avec l'autre. Les instances de connexions sont donc établies entre les rôles d'une occurrence de processus logiciel avec les rôles d'une autre occurrence. Dans l'exemple cité ci-dessus, les occurrences des processus logiciels **ModifyDesign** et **ReviewDesign** peuvent se synchroniser et échanger des informations par la connexion ***designing***. Cette connexion sera établie respectivement entre les rôles **UnderDesign** et **UnderReview**.

V.4.3.1 Les contraintes de mise en connexion

Les conditions sur lesquelles deux occurrences doivent se connecter sont décrites par les contraintes spécifiées dans la clause **PLUG-ON**. Par exemple :

```
designing ISA CONNECTION;
    DOMAIN
         ModifyDesign:UnderDesign ->
         ReviewDesign:UnderReview;
```

```
    PLUG-ON-RULES
(1)  WHEN createprocess UPON (SOURCE OR DEST);
(2)  WHEN allocate_ressouces UPON (SOURCE OR DEST);
(3)  WHEN continue_execution UPON (SOURCE OR DEST)

    ACTIVE-RULES . . .
    PLUG-OFF-RULES . . .
END_OF designing;
```

Dans l'exemple montré ci-dessus, une connexion de type **designing** sera automatiquement établie sur les événements suivants :

1. une occurrence de type de processus logiciel **ReviewDesign** ou **ModifyDesign** est créée.
2. de nouveaux objets sont alloués pour les rôles **UnderDesign** ou **UnderReview**.
3. les rôles **UnderDesign** ou **UnderReview** reçoivent une message permettant de suivre leurs exécutions.

V.4.3.2 Les contraintes de rupture

De manière similaire aux contraintes de mise en connexion, pour chaque type de connexion nous pouvons décrire les conditions dans lesquelles une connexion doit être défaite. Ces contraintes sont décrites dans la clause **PLUG-OFF**. Par exemple :

```
designing ISA CONNECTION;
    DOMAIN
         ModifyDesign:UnderDesign ->
         ReviewDesign:UnderRevision;

    PLUG-ON-RULES
     WHEN createprocess UPON (SOURCE OR DEST);
     WHEN allocate_ressouces UPON (SOURCE OR DEST);
     WHEN continue_execution UPON (SOURCE OR DEST)

    ACTIVE-RULES . . .

    PLUG-OFF-RULES
1)   WHEN stop_execution UPON (SOURCE OR DEST);
2)   WHEN finish_execution UPON (SOURCE OR DEST);
END_OF designing;
```

Cet exemple décrit les contraintes de rupture suivantes :

1. la connexion entre ces deux occurrences est défaite, lorsqu'un message informant de l'arrêt des activités d'une des deux occurrences de processus logiciels connecté est validé,
2. de même, si l'un des deux processus coopérant finit ses activités, la connexion entre eux est également défaite.

V.4.3.3 Les règles de contrôle de collaboration

Pour chaque type de connexion, nous pouvons décrire un ensemble de règles TECA permettant de contrôler l'échange d'informations entre deux occurrences de processus logiciels. Pour que cela soit possible, les règles de contrôle doivent avoir accès aux objets manipulés par les deux occurrences que la connexion serve à lier. La connexion doit également être capable de suivre les opérations effectuées sur ces objets. Cela veut dire qu'une mise à jour sur les objets manipulés par les deux occurrences de processus logiciel, A et B, liées par une connexion C, doit provoquer des événements aussi bien dans le contexte des occurrences A et B que dans le contexte de la connexion C. Les règles TECA définies dans cette connexion peuvent alors répondre à de tels événements. Par exemple :

```
designing ISA CONNECTION;
    DOMAIN
         ModifyDesign:UnderDesign ->
         ReviewDesign:UnderReview;

    PLUG-ON-RULES
     WHEN createprocess UPON (SOURCE OR DEST);
     WHEN allocate_ressouces UPON (SOURCE OR DEST);
     WHEN continue_execution UPON (SOURCE OR DEST);

    ACTIVE-RULES
1)       WHEN design_completed UPON SOURCE
2)       DO  promote(%source);
3)           allocate(%source,occurence_of(%dest));

4)       WHEN design_reviewed UPON DEST
5)       DO  promote(%dest);
6)           IF (%dest.no_of_changes >= 0) THEN
7)           allocate(%dest,occurence_of(%source));

    PLUG-OFF-RULES
     WHEN stop_execution  UPON (SOURCE OR DEST);
     WHEN finish_execution UPON (SOURCE OR DEST);
END_OF designing;
```

Les règles décrites dans la clause **ACTIVE-RULES** informent que :

1. lorsque l'activité de modification du document de conception est achevée dans l'occurrence de processus logiciel responsable de cette modification (ligne 1), l'événement `design_completed` est pris en compte.
2. les modifications effectuées doivent être propagées (ligne 2);
3. ce document doit être alloué à l'occurrence de processus logiciel qui a en charge la révision (ligne 3).
4. une fois l'activité de révision du document de conception achevée, l'événement `design_reviewed` est pris en compte et traité par cette règle (ligne 4)
5. les résultats obtenus par cette révision doivent être promus. L'opération `promote` est fournie pour mettre en oeuvre cette promotion (ligne 5),
6. Après la promotion des résultats de l'activité de révision, la vérification des corrections est faite sur le document de conception (ligne 6),
7. Si des corrections ont été introduites, le document de conception est automatiquement alloué à l'occurrence du processus logiciel responsable de sa modification (ligne 7).

Les mots clés **`UPON SOURCE`** ***`event`***/**`UPON DEST`** ***`event`*** informent que l'opération qui a provoqué l'événement `event` a été faite respectivement sur le rôle source ou sur le rôle destination de la connexion.

V.4.3.4 Définition

Pour donner une définition aux types de connexions nous considérons les ensembles suivants :

> $\boldsymbol{C}_u$ un ensemble dont les éléments sont appelés types de relations et $\boldsymbol{C}_n$ l'ensemble infini de symboles désignant les noms des types de relations. Nous imposons que l'intersection entre les ensemble $\wp_n$, $\mathcal{T}_n$, $\mathfrak{R}_n$, $\mathcal{L}_n$ et $\boldsymbol{C}$ soit vide. Donc, $\wp_n \cap \mathcal{T}_n \cap \mathcal{L}_n \cap \mathfrak{R}_n \cap \boldsymbol{C}_n = \emptyset$.

Considérons l'ensemble de tous les types de connexions noté par $\boldsymbol{C}_t$ chaque type de connexion C de $\boldsymbol{C}_t$ est un n-uplet *(c, S, D, E)* tel que :
$c \in \boldsymbol{C}_n$ est le nom de type de la connexion,
E est un sous-ensemble de $\mathcal{E}_u$ désignant les règles temporelles événements-condition-action de C.
S et D sont chacun un sous-ensemble de $\mathfrak{R}_t$ désignant les types de rôles qui peuvent être connectés par la connexion C. Comme les connexions sont binaires, S fournit les types

de rôles qui peuvent être utilisés comme source de la connexion C tandis que D fournit les types de rôles qui peuvent être utilisés comme destination.

De même que pour les types d'objets, nous désignons l'opérateurs **Rules**(C) qui permet de retrouver la définition des règles TECA appartenant à un type de connexion C. Nous fournissons également l'opérateur **Type**(c) qui permet de retrouver le type de la connexion c et l'opérateur **Extention**(C) qui fournit les instances des connexions qui appartiennent au type de connexion C.

Nous fournissons les opérateurs **Source**(C) et **Destination**(C) qui permettent de retrouve les rôles sources et destinations de la connexion C. Ces opérateurs sont définis ainsi :

$$\textbf{Source}(C) = S \text{ et } \textbf{Destination}(C) = D, \text{ où } S \in \Re_t \text{ et } D \in \Re_t .$$

Chaque type de connexion peut également être spécialisé pour préciser sa structure. Pour permettre la spécialisation de types de connexions, nous fournissons la relation d'ordre partiel **ISAc**. Cette relation peut classer les types de connexions suivant un treillis de types, où chaque type de connexion peut être considéré comme étant un sous-type d'un ou de plusieurs autres types de connexions. Pour définir la relation d'ordre partiel **ISAc**, l'existence d'un symbole *CONNECTION* $\in$ $\mathbf{C}_n$ qui constitue le nom du type de connexion racine du treillis des types de connexions, tel que :

1. pour chaque type de relation C, on C **ISAc** *CONNECTION*.
2. soient les deux types de connexion $L_1=(l_1, E_1, S_1, D_1)$ et $L_2=(t_2, E_2, S_2, D_2)$, on dit que C_1 **ISAc** C_2 si et seulement si E_1 **ISAe** E_2, S_1 **ISAr** S_2 et D_1 **ISAr** D_2.
3. si C_1 **ISAc** C_2 , alors l'extension de C_1 est inclue dans l'extension de C_2.

Nous désignons l'opérateur *SuperTypes(C)* = $\{C_1, ..., C_n\}$ qui permet d'obtenir la liste de super types de C, où pour chaque i <=n, on a C_i **ISAc** *CONNEXTION* et C **ISAc** L_i

V.5 Conclusion

Dans ce chapitre, nous avons montré la façon dont le travail coopératif est supporté dans TEMPO. Il est basé principalement sur deux composants :

1. un gestionnaire de ressources pour réglementer le partage des ressources contrôlées par un modèle unique de données unifiant les informations descriptives et les relations. Ce partage est basé sur la gestion d'un arbre de version de composants

2. un gestionnaire d'activités contrôlé par un formalisme exécutable permettant de décrire des modèles de processus logiciels. Ce modèle structure les activités en unités de base appelées types de processus logiciels et fragments, instanciées par des environnements de travail à l'exécution. Le déroulement du processus de production logiciel est contrôlé par des règles temporelles événement-condition-action.

Les politiques qui gouvernent la synchronisation et la coopération entre les différentes occurrences de processus s'exécutant en parallèle sont spécifiées par les connexions dites *actives et programmables*. Une connexion est une relation binaire liant deux rôles. La description de la politique de communication se fait par des règles définissant des stratégies spécifiques de synchronisation entre les *rôles* en propageant des effets dès qu'une action est exécutée sur l'un ou l'autre des points de la connexion.

La gestion de la cohérence des objets manipulés par les activités de longe durée est faite par les environnements de travail. L'union entre les connexions et les environnements de travail permet de supporter les processus coopérants et le partage d'objets entre ces processus. Les activités de longue duré dans TEMPO sont supportées par une hiérarchie de transactions longues. TEMPO permet que ces activités puissent se synchroniser pour échanger les résultats obtenus au cours de leur exécution. L'utilisateur chargé d'une occurrence de processus logiciel, donc une activité de longue durée, peut demander au cours de l'exécution de ce processus de se synchroniser avec le processus logiciel immédiatement au-dessus de lui dans la hiérarchie.

D'autre part, TEMPO peut automatiquement forcer une synchronisation en se basant sur les politiques de communication exprimées par les règles TECA. La modalité temporelle des règles TECA permet de supporter les transactions longues (activités de longue durée), car elles peuvent être utilisées pour raisonner sur des activités qui ont eu lieu dans le passé.

CHAPITRE VI La Réalisation de TEMPO 157

VI.1 Introduction 157

VI.2 Le gestionnaire de ressources 158

VI.2.1 La modélisation des données 158

VI.2.1.1 Attributs 158

VI.2.1.2 Objets 159

VI.2.1.3 Relations 161

VI.2.2 L'utilisation de modèle de données d'Adèle pour la représentation des processus logiciels 163

VI.2.2.1 Les types de relations pré-définies 163

VI.2.2.2 Les types de processus logiciels 164

VI.2.2.3 Les fragments de processus logiciels 165

VI.2.2.4 Les types de rôles 167

VI.2.2.5 Les connexions 170

VI.3 L'historique des objets 171

VI.3.0.1 Le type HISTORIQUE 171

VI.3.0.2 L'association des historiques avec les objets 173

VI.3.0.3 La manipulation des historiques 173

VI.4 Le gestionnaire de processus 174

VI.4.1 Les règles événements-condition-action 174

VI.4.1.1 Les événements 174

VI.4.1.2 Les actions 175

VI.4.1.3 Les règles 175

VI.4.1.4 Les règles sur les relations 177

VI.4.2 Les événements sur le passé 177

VI.5 Implémentation de TEMPO : état de lieu 178

VI.6 Conclusion 179

CHAPITRE VI La Réalisation de TEMPO

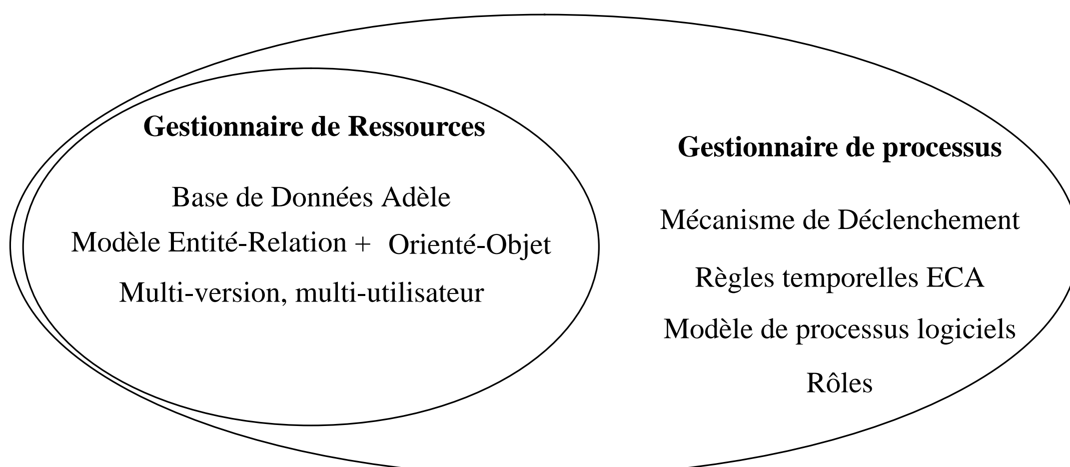
VI.1 Introduction

Comme le montre la figure VI.1, TEMPO comprend deux parties principales :

1. un gestionnaire de ressources qui utilise Adèle comme base d'objets pour stocker objets et activités ainsi que pour tracer le déroulement du projet.

2. un gestionnaire de processus. Les règles temporelles événement-condition-action (TECA) et le mécanisme de déclenchement (“*trigger*”) sont sollicités par le gestionnaire de processus. Ce gestionnaire permet la définition et la structuration d'activités par le concept de processus logiciels. Les objets impliqués dans un processus sont définis par le concept de rôle également supporté par ce gestionnaire. L'exécution des processus logiciels est supportée par le biais des environnements de travail.

Figure VI.1 Les principaux composants de TEMPO.



VI.2 Le gestionnaire de ressources

La base de données Adèle a été adopté comme le gestionnaire de ressources de TEMPO. L'environnement Adèle est un système spécialement conçu pour supporter la programmation-globale. Il est dédié à la gestion des objets, des versions et des configurations logicielles (Estublier et al., 1984) ainsi qu'à la structuration des gros logiciels (Estublier and Favre, 1989). De plus, la base de données Adèle permet la gestion des différents types d'objets produits durant le développement d'un logiciel (spécifications, cahiers des charges, code source, code binaire, jeux de tests, etc) ainsi que des relations entre ces objets.

Le système Adèle (Belkhatir et al., 1991; Belkhatir et al., 1992) est articulé autour des composants suivants :

- un gestionnaire d'objets (Belkhatir and Estublier, 1987). Les objets, distribués sur différents sites, peuvent être manipulés par une interface graphique, un langage de commandes ou par une interface programmatique. Ce gestionnaire supporte un modèle de données voisin du modèle entité-relation étendu avec les concepts d'héritage multiple et de versionnement; tout objet peut être versionné.
- un gestionnaire de configurations (Estublier et al., 1984) permet de construire des systèmes logiciels à partir de plusieurs composants logiciels (modules), chacun d'eux pouvant exister en plusieurs versions. La configuration est construite à partir d'une requête sur l'état des objets stockés dans la base.

Dans les sections suivantes nous analysons le modèle de données d'Adèle (Belkhatir and Estublier, 1987).

VI.2.1 La modélisation des données

Le modèle de données d'Adèle fournit les concepts d'objet, d'associations entre ces objets, d'attributs et de versions d'objets.

VI.2.1.1 Attributs

Tous les objets et les relations peuvent avoir des attributs associés. Les attributs sont typés. Le modèle de données supporte cinq types d'attributs : entier, réel, chaîne de caractères, date et énuméré. Par défaut, les attributs sont multi-valués. Une caractéristique MONO est cependant fournie pour les rendre mono-valué. Ils se présentent sous la forme d'un couple `nom_attribut/valeur_attribut`. Il existe trois types d'attributs :

1. les attributs pré-définis, tels que le nom et le type de l'objet ou de la relation, la date de création, le nom de l'auteur;

2. Les attributs calculés qui fournissent des informations relatives aux objets (nom des objets englobants, fichiers associés, ...) et au contexte d'utilisation (commande courante, utilisateur courant...).

3. Les attributs définis par l'utilisateur. Comme par exemple :

a. `delivered MONO = true, false := true;`
L'attribut `delivered` est défini comme étant un attribut mono-valué. Cet attribut peut pendre pour valeur `true` ou `false`. Lorsque l'objet ou la relation est créé cet attribut est initialisé avec la valeur `true`.

b. `target = sun-3, sun-4, hp := sun-3, sun4;`
L'attribut `target` peut pendre pour valeur : `sun-3, sun-4` ou `hp`. Lorsque l'objet ou la relation est crée cet attribut est initialisé avec les valeurs `sun-3` et `sun-4`.

c. `lines MONO = INTEGER;`
L'attribut `lines` est définit comme étant un attribut mono-valué de type entier.

d. `comments = STRING;`
L'attribut `comments` est un attribut multi-valué de type chaîne de caractères.

VI.2.1.2 Objets

Les objets manipulés dans un environnement sont typés. A tout type d'objet on peut associer des attributs, des relations ou des contraintes d'intégrité. Tous les types d'objet ont pour racine un type pré-défini nommé **`object`**.

```
TYPEOBJECT object;
    DEFATTRIBUTE
       creatdate = DATE; -- date de creation de l'objet
       author = STRING;  -- nom de l'utilisateur
                         -- qui a créé l'objet
END object;
```

Le type **`Elem`** est fourni pour permettre le stockage des objets logiciels, tels que des sources de programmes, des textes. Ce type est câblé par le système et ses composants correspondent à des fichiers.

```
TYPEOBJECT Elem ISA object;
    DEFATTRIBUTE
       moddate MONO = DATE;--date de la dernière modification
END Elem;
```

Les objets logiciels, une fois stockés dans la base de données, sont considérés comme immuables, c'est-à-dire qu'ils ne peuvent être modifiés, mais seulement créés ou éventuellement détruits. La modification d'un objet provoque donc la création d'une nouvelle révision de cet objet. Une révision est un objet composé d'un objet source, par exemple, un programme Pascal, et une liste d'objets dérivés de cet objet source, par exemple les fichiers correspondant au code binaire et exécutable du programme Pascal. Le type révision est défini de la manière suivante :

```
TYPEOBJECT Revision ISA object;
    STRUCT
       sc : Elem;          -- l'object source
       * : List_Of Elem;   -- les objects dérivés
END Revision;
```

Cette définition exprime qu'une révision est constituée d'un élément de nom **`sc`** et d'un nombre quelconque d'objets dérivés de nom quelconque "**`*`**".

Un ensemble de révisions d'un même objet constitue un **`document`**. Un document représente donc l'évolution d'un élément logiciel dans le temps. Un document est une succession ordonnée de révisions. Le noyau d'Adèle fournit le type pré-défini **`Rset`** pour permettre la modélisation d'un document quelconque.

```
TYPEOBJECT Rset ISA object;
    DEFATTRIBUTE
       reservedby MONO = STRING;
    STRUCT
       [0-9][0-9][0-9] : List_Of Revision;
END Rset;
```

Un document est une suite de révisions ayant pour nom un entier de trois chiffres maintenu automatiquement par Adèle.

Le dernier type pré-défini d'Adèle est l'objet composé :

```
TYPEOBJECT Cobj ISA object;
    STRUCT
       main : Rset;
       * : List_Of Rset;
       * : List_Of Cobj;
END Cobj;
```

Ceci exprime qu'un objet complexe est composé d'un document de nom **`main`** et d'une liste non limitée d'autres documents et d'autres objets complexes. Il convient de remarquer que tout objet a un identificateur structuré basé sur les concepts de révision, d'objet source et d'objet dérivé :

`Cobj.Rset.Révision.Element`

Par exemple : `Module5.main.012.sc`.

Ces quatre types d'objets sont pré-définis par le système. D'autres types peuvent être gérés par spécialisation d'un de ces quatre types. Par exemple, un module peut être défini de la manière suivante :

```
TYPEOBJECT Module ISA Cobj;
    STRUCT
1)     main : Rset;
2)     * : List_Of Rset;
3)     * : List_Of interface;
END Module;

TYPEOBJECT Interface ISA Cobj;
    STRUCT
4)     main : Rset;
5)     * : List_Of Rset;
6)     * : List_Of Body;
END Interface;

TYPEOBJECT Body ISA Cobj;
    STRUCT
7)     main : Rset;
8)     * : List_Of Rset;
END Body;
```

Un module est composé d'un document décrivant la fonctionnalité du module (ligne 1), d'un ensemble de documents de conception (ligne 2), et d'un ensemble de variantes d'implémentation du module (ligne 3).

Chaque variante est définie comme étant un type d'interface. Une interface est composée d'un document décrivant sa fonctionnalité (ligne 4), d'un ensemble de documents contenant les vues de l'interface (ligne 5), par exemple une vue C, une vue Pascal, une autre vue C++ de la même interface, etc, et un ensemble d'alternatives aux corps implantant l'interface (ligne 6). Enfin chaque corps est également un objet composé d'un document contenant un corps de programme (ligne 7) et d'un ensemble de versions (ligne 8).

VI.2.1.3 Relations

Adèle permet de représenter les liens existants entre objets. Il y a les associations inhérentes au modèle de produit, comme par exemple celle qui lie une réalisation à l'interface qu'elle réalise ou encore celle qui lie un objet dérivé à son objet source. Il existe également les relations qui permettent d'exprimer des structures et des faits. On distingue les relations pré-définies et les relations définies par l'utilisateur. Les relations pré-définies sont créées et maintenues par le noyau d'Adèle. Ces relations font partie du modèle de produit câblé par le système. Le gestionnaire de configuration connaît ces

relations, et à partir de cette connaissance et de l'états des objets il peut construire des configurations de programmes.

Les relations définies par l'utilisateur permettent d'exprimer les associations sémantiques existantes entre les objets dans le monde réel. Par exemple, l'association sémantique suivante "Le programme X dépend de l'interface Y" est définie comme suit :

```
TYPERELATION dep;
    DOMAIN
       [ type = body ]     -> [ type = interface ] OR
       [ type = interface ] -> [ type = interface ]
    CARD N:N;
    SYNTH_BY use;
    DEFATTRIBUTE
       no_of_functions MONO = INTEGER;
END dep;
```

Ceci exprime le fait que la relation dep est définie entre un corps d'un module et une vue d'interface ou entre deux vues d'interface. CARD indique la cardinalité : des relations dep en nombre N quelconque peuvent arriver sur un noeud donné ou en partir. La relation dep définit donc un graphe quelconque.

La clause SYNTH_BY indique que dep doit être synthétisée par la relation use. Ainsi, lors de la création d'une relation dep, la relation use doit être créée sur des objets englobant ceux de la relation dep.

Les relations peuvent, comme les objets, être caractérisées par des attributs. Par exemple, la relation dep possède l'attribut no_of_functions indiquant le nombre de fonctions exportées par l'interface et utilisées par le programme.

Les types de relations peuvent être spécialisés pour mieux exprimer la sémantique des associations entre les objets. Analysons le cas suivant :

```
TYPERELATION dep_pascal ISA dep;
    DOMAIN
       [ type = body_pascal ] ->
       [ type = interface_pascal ]
END dep_pascal;
```

La relation dep_pascal sert à lier les corps des programmes en Pascal avec ses interfaces respectives .

VI.2.2 L'utilisation de modèle de données d'Adèle pour la représentation des processus logiciels

Un compilateur a été conçu permettant la traduction les concepts de processus logiciel, de fragment de processus, de rôle et de connexion vers les concepts utilisés par la base de données Adèle : types d'objets et types de relations. Le mécanisme de recouvrement (*"late-binding"*) d'Adèle a été largement utilisé. Grâce à ce mécanisme les types de rôles peuvent remplacer les définitions des méthodes faites dans les types d'objets. Dans cette section nous donnons un aperçu synthétique de ce compilateur. Avant cela, nous montrons les types pré-définies utilisés par TEMPO. Tous les types de processus logiciels, fragments de processus et rôles sont définies comme sous-types des types pré-définies.

VI.2.2.1 Les types de relations pré-définies

Les contraintes associées aux rôles "**modifiable**", "**visualizable**" et "**new**" ainsi qu'aux types de processus, sous-processus sont décrites dans des types de relations pré-définies dans TEMPO :

```
TYPERELATION isbranch;
    DOMAIN
         [ type = rset ] -> [ type = rset ];
    CARD N:1;
    POST ORIGIN promote DO
         "rsrset !destname";
         "rpe !destname -f ~!sourcename!filename";
END isbranch;

TYPERELATION sub_process;
    DOMAIN
       [ type = Cobj ] -> [ type = process ];
 CARD 1:N;
 PRE ORIGIN rmprocess DO
    IF [ ~!sourcename%type = process ] THEN {
         "#";
         "# remove before the sub-processes
         of the !optvalname process occurence";
         "#";
         ABORT; };

 POST DEST rmprocess DO
    IF [ ~!sourcename%type = process ] THEN
         "chprocess !sourcename"
    ELSE "chct -ct !sourcename";
END sub_process;

TYPERELATION role;
    DOMAIN
    [ type = process ] -> [ type = role ];
```

```
END role;

TYPERELATION new-role ISA role;
    PRE DEST modification DO
    1) If destination object does not exist yet:
         a) create object according to process name rules
         b) insert object in context
    2) otherwise,
       if destination object exist and
         it does not belong to the current process
       then abort.
END new-role;

TYPERELATION role-visualizable ISA role;
    PRE DEST modification DO
       ABORT;
END role-visualizable;

TYPERELATION role-modifiable ISA role;
 PRE DEST modification DO
    1) create branch
    2) copy source relationships from original object to
         the created branch.
    3) up-date current context (remove original object and put the
         new one)
    5) copy attributes from original object to the created
         branch.
END role-modifiable;
```

VI.2.2.2 Les types de processus logiciels

La traduction d'un type de processus logiciel vers le modèle de données d'Adèle est aisée, car pour chaque type de processus logiciel défini dans le modèle TEMPO nous créons simplement un type d'objet Adèle.
Exemple :

TEMPO:

```
MonitorDesign ISA PROCESS;
ATTRIBUTES
. . .
METHOD
. . .
RULES
. . .
END_OF MonitorDesign;
```

ADELE :

```
TYPEOBJECT MonitorDesign ISA PROCESS;
ATTRIBUTES
. . .
METHOD
. . .
RULES
. . .
END MonitorDesign;
```

VI.2.2.3 Les fragments de processus logiciels

Pour chaque type de fragment de processus logiciel, défini dans un modèle de processus logiciels, nous générons un type d'objet, un type de relation et une liste d'attributs.

1. Le type d'objet est utilisé pour représenter des informations définies dans le type de fragment de processus logiciel (les attributs, les méthodes et les règles TECA).
2. Le type de relation est utilisé pour contrôler les opérations de création, de suppression et de navigation dans les occurrences des processus logiciels.
3. Les attributs sont utilisés par TEMPO pour récupérer les définitions des fragments d'un type de processus logiciel. La présence de ces attributs est nécessaire car Adèle ne fournit pas de concepts qui supportent la décomposition arbitraire des types d'objets. Ces attributs sont donc utilisés par TEMPO pour gérer ces décompositions. Dans l'exemple ci-dessous, TEMPO prend connaissance des fragments du type de processus logiciel **MonitorDesign** en consultant les attributs qui correspondent à l'expression régulière "**sub:*:type**". De manière similaire, en utilisant les relations **REL:MonitorDesign:*,** TEMPO peut déduire les fragments (les instances) appartenant à une occurrence de processus logiciel.

Exemple :

Définition en TEMPO:

```
ModifyDesign ISA PROCESS;
    ...
END_OF ModifyDesign;

ReviewDesign ISA PROCESS;
    ...
END_OF ReviewDesign;

MonitorDesign ISA PROCESS;
    ATTRIBUTES
```

```
        . . .
    METHOD
        . . .
    RULES
        . . .
CONTROL Modify;
    fragment := ModifyDesign;
    card := 1;
CONTROL Review;
    fragment := ReviewDesign;
    card := N;
END_OF MonitorDesign;
```

Traduction en ADELE :

```
TYPEOBJECT ModifyDesign ISA PROCESS;
    . . .
END ModifyDesign;

TYPEOBJECT ReviewDesign ISA PROCESS;
    . . .
END ReviewDesign;

TYPEOBJECT MonitorDesign ISA PROCESS;
    ATTRIBUTES
       listofsub := Modify, Review;
       sub:Modify:type := ModifyDesign;
       sub:Modify:card := 1;
       sub:Review:type := ReviewDesign;
       sub:Review:card := N;
       . . .
    METHOD
       . . .
    RULES
       . . .
END MonitorDesign;

TYPERELATION REL:MonitorDesign:Modify ISA REL:sub-process;
    DOMAIN
       [ type=MonitorDesign ] -> [ type=MonitorDesign:Modify ];
END REL:MonitorDesign:Modify;

TYPERELATION REL:MonitorDesign:Review ISA REL:sub-process;
    DOMAIN
       [ type=MonitorDesign ] -> [ type=MonitorDesign:Review ];
END REL:MonitorDesign:Review;

TYPEOBJECT MonitorDesign:Modify ISA sub-process,modify-design;
END MonitorDesign:Modify;

TYPEOBJECT MonitorDesign:Review ISA sub-process,review-design;
END MonitorDesign:Review;
```

VI.2.2.4 Les types de rôles

Pour chaque type de rôle défini dans un modèle de processus, nous générons un type d'objet et un type de relation. Nous définissons également une liste d'attributs dans le type d'objet représentant le type de processus logiciel où les rôles sont déclarés. Cette liste permet à TEMPO de connaître :

1. les types de rôles d'un type de processus logiciel.

2. la cardinalité de ces rôles;

3. les types d'objets auxquels ces rôles font référence.

Exemple :

Définition en TEMPO:

```
MonitorDesign ISA PROCESS;
    ROLE ccb;
         derived := spec
         mode := visualizable;
         cons := elementof;
         card := 1, 1;
END_OF MonitorDesign;
```

Résultat en Adèle:

```
TYPEOBJECT MonitorDesign ISA PROCESS;
    ATTRIBUTES
         role:ccb:derived := type/spec;
         role:ccb::mode := visualizable;
         role:ccb:cons := elementof;
         role:ccb:card:min := 1;
         role:ccb:card:max := 1;
         . . .
    METHOD
         . . .
    RULES
         . . .
END MonitorDesign;
TYPERELATION REL:MonitorDesign:ccb ISA role-visualizable;
    DOMAIN
         [ type = MonitorDesign ] -> [ type = spec ]
END REL:MonitorDesign:ccb;
```

Le type de rôle **ccb** est défini comme étant dérivé du type d'objet **spec**. Le mode d'utilisation des objets qui vont jouer ce rôle est "**visualizable**". Cela signifie que les

objets jouant ce rôle ne pourront être manipulés par les occurrences de type **MonitorDesign**.

Le type de rôle `ccb` est traduit dans Adèle par une relation (**REL:MonitorDesign:ccb**) et par un ensemble d'attributs préfixés par **role:ccb**. Les attributs sont définis dans le type d'objet représentant le type de processus logiciel nommé **MonitorDesign**. La relation **REL:MonitorDesign:ccb** est utilisée pour lier les occurrences de type **MonitorDesign** avec les objets jouant le rôle **ccb**.

Les contraintes d'utilisation propre au rôle "visualizable" sont définies dans la relation **role-visualizable** (voir section VI.2.2.4). Dans le cas où nous décidons de modifier ces contraintes, les modèles de processus logiciel déjà traduits ne seront pas affectés, car ces modifications seront automatiquement héritées de la relation **role-visualizable**. Nous utilisons ainsi intensivement les facilités fournies par le modèle orienté-objet. Cela nous permet plus de souplesse dans la traduction et dans la modification d'un modèle de processus logiciels.

Nous donnons ensuite un autre exemple où les rôles sont définis comme étant "**modifiable**". Les objets jouant ces rôles pourront être modifiés dans les occurrences des processus logiciels.

Définition en TEMPO :

```
ModifyDesign ISA PROCESS;
1)  ROLE ccb;
        derived := spec
        mode := visualizable;
        cons := elementof;
        card := 1, 1;
2)  ROLE UnderDesign;
        derived := spec "%role.owner=!userid";
        mode := modifiable;
        cons := setof;
        card := 0, N;
3)  ROLE OldDesign;
        derived := under_design "%role.changes > 50";
        mode := modifiable;
        cons := elementof;
        card := 1, 1;
END_OF ModifyDesign;
```

Résultat en ADELE:

```
TYPEOBJ ModifyDesign ISA PROCESS;
    ATTRIBUTES
1)        role:ccb:derived := type/spec;
          role:ccb:mode := visualizable;
          role:ccb:cons := elementof;
          role:ccb:card:min := 1;
          role:ccb:card:max := 1;
2)        role:UnderDesign:derived := type/spec;
          role:UnderDesign:spec:exp := "~dest%owner = !userid";
          role:UnderDesign:mode := modifiable;
          role:UnderDesign:cons := setof;
          role:UnderDesign:card:min := 0;
          role:UnderDesign:card:max := N;
3)        role:OldDesign:derived := role/under_design;
          role:OldDesign:UnderDesign:exp := "~dest%changes > 50";
          role:OldDesign:mode := modifiable;
          role:OldDesign:cons := elementof;
          role:OldDesign:card:min := 1;
          role:OldDesign:card:max := 1;
          . . .
    METHOD
          . . .
    RULES
          . . .
END ModifyDesign;

TYPERELATION REL:ModifyDesign:ccb ISA role-visualizable;
    DOMAIN
          [ type = ModifyDesign ] -> [ type = spec ]
END REL:ModifyDesign:ccb;

TYPERELATION REL:ModifyDesign:UnderDesign ISA role-modifiable;
    DOMAIN
          [ type = ModifyDesign ] -> [ type = spec ]OR
          [ type = ModifyDesign ] ->
                    [ type = ModifyDesign:UnderDesign ];
END REL:ModifyDesign:UnderDesign;

TYPEOBJ ModifyDesign:UnderDesign ISA role-modifiable, spec;
END ModifyDesign:UnderDesign;

TYPERELATION REL:ModifyDesign:OldDesign ISA role-modifiable;
    DOMAIN
          [ type = ModifyDesign ] -> [ type = spec ]OR
          [ type = ModifyDesign ] ->
               [ type = ModifyDesign:OldDesign ];

    AFTER DEST modification DO
          IF [ "~!O%role:OldDesign:roles:UnderDesign:exp" != ""]
          THEN
               IF [ ~!O%role:OldDesign:UnderDesign:exp ]
               THEN {
                    "allocate(!O,OldDesign,!D)";
                    "kill(!O,UnderDesign,!D)";
               }
END REL:ModifyDesign:OldDesign;
```

```
TYPEOBJ MonitorDesign:OldDesign ISA role-modifiable, spec;
END MonitorDesign:OldDesign;
```

VI.2.2.5 Les connexions

Les connexions sont traduites par des relations dans Adèle. Les règles actives dans un type de connexion sont traduites directement par des règles TECA dans le type de relation représentant la connexion dans le modèle de données d'Adèle. Les règles "**PLUG-ON-RULES**" et "**PLUG-OFF-RULES**" sont traduites par des règles TECA dans les types d'objets représentant les types de rôles que la connexion sert à lier.
Exemple :

Description en TEMPO :

```
designing ISA CONNECTION;
    DOMAIN
         ModifyDesign:UnderDesign ->
         ReviewDesign:UnderRevision;

    PLUG-ON-RULES
     WHEN createprocess UPON (SOURCE OR DEST);
     WHEN allocate_resources UPON (SOURCE OR DEST)
         USING %dest.!fname == %source.!fname;
     WHEN continue_execution UPON (SOURCE OR DEST);

    ACTIVE-RULES
         WHEN design_completed UPON SOURCE
         DO   promote(%source);
              allocate(%source,occurence_of(%dest));

         WHEN design_reviewed UPON DEST
         DO   promote(%dest);
              IF (%dest.no_of_changes >= 0) THEN
              allocate(%dest,occurence_of(%source));

    PLUG-OFF-RULES
     WHEN stop_execution    UPON (SOURCE OR DEST);
     WHEN finish_execution  UPON (SOURCE OR DEST);
END_OF designing;
```

Résultat ADELE :

```
TYPERELATION designing ISA CONNECTION;
 DOMAIN
    [ type = ModifyDesign:UnderDesign ] ->
    [ type = ReviewDesign:UnderRevision ]
 CARD N:N;
 DEFATTRIBUTE
    ProcessSource := ModifyDesign;
    ProcessDestination := ReviewDesign ;
```

```
        RoleSource := UnderDesign;
        RoleSource := UnderRevision;
    ON SOURCE design_completed DO
        promote ~!source%name;
        allocate ~!source%name
              -d ~(**|REL:ReviewDesign:UnderRevision|~!source%name)!O;

    ON DEST design_reviewed DO
        promote ~!dest%name;
        IF (~(~!dest%name)%no_of_changes >= 0) THEN
              allocate ~!dest%name
                   -d ~(**|REL:ModifyDesign:UnderDesign|~!dest%name)!O;
    END designing;

    TYPERELATION REL:modify-design:UnderDesign ISA role-modifiable;
     DEFATTRIBUTE
        CON:designing:SOURCE := ModifyDesign:UnderDesign;
        CON:designing:DEST := ReviewDesign:UnderRevision;
        CON:designing:PLUG-ON:allocate:USING :=
              «~%source%fname = ~%dest%fname»;
        CON:designing:PLUG-ON:allocate:WHEN := allocate;
        CON:designing:PLUG-ON:allocate:ABOVE := target;
     ...
     AFTER DEST allocate DO
        IF [ CON:designing:SOURCE != «» ] THEN {
              SUB = ~!relship%CON:designing:**;
              PARENT = ~(**|%SUB|!O)%!O;
              TARGET = !D;
              FORALL SOURCE
              IN «(%PARENT!(~!relship%CON:designing:SOURCE)!**)»
              DO {
                   IF [ ~!relship%CON:designing:PLUG-ON:allocate:USING ]
                   THEN
                         mkcon %SOURCE -d %TARGET -r designing;
              };
        };
    END REL:modify-design:under-design;
```

VI.3 L'historique des objets

La base de données d'Adèle a été étendue de manière à permettre la gestion des historiques des objets de manière explicite. Un historique est associé à chaque objet stocké. L'historique a pour but de garder la trace des mises-à-jours faites sur un objet.

VI.3.0.1 Le type HISTORIQUE

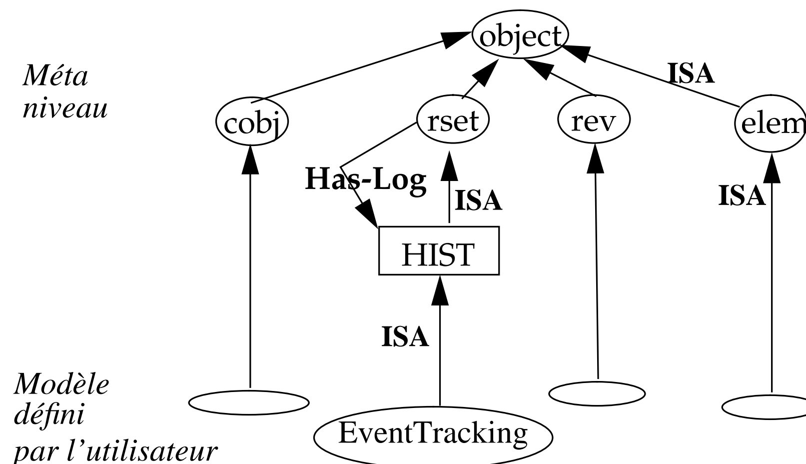
Un type pré-défini nommé **HIST** fournit un ensemble d'attributs communs à tous les historiques de la base Adèle (méta-type)

```
TYPEOBJECT HIST;
    DEFATTRIBUTE
    name      INIT = STRING := %name;
    Author    INIT =STRING := !username;
```

```
    Date      INIT = STRING := !date;
    type      := HIST ;
    merged_from = STRING;
END HIST;
```

- L'attribut "name" fait la liaison entre l'objet et son historique
- L'attribut "Author" a pour valeur l'identification de l'utilisateur qui a créé l'historique.
- L'attribut "Date" garde la date de création de l'historique
- L'attribut "type" indique le type de l'historique;
- L'attribut "merged_from" indique si l'objet a été créé comme le résultat d'une opération d'unification entre deux objets.

Figure VI.2 Exemple de spécialisation d'un type d'historique



En utilisant l'héritage, d'autres types d'historiques peuvent être définis par l'utilisateur. Cela permet de bâtir une base historique adaptable pour chaque produit logiciel. De cette façon, l'utilisateur peut décider, en fonction du produit, les informations qu'il considère importantes à conserver au cours de l'exécution des processus logiciels (voir figure VI.2).

Par exemple :

```
TYPEOBJECT EventTracking ISA HIST;
    source   INIT = STRING := !sourcename
    relation INIT = STRING := !relship
    destinationINIT = STRING := !destname
    command INIT = STRING := !cmd
    event   INIT = STRING := !eventid ;
END EventTracking ;
```

L'historique **EventTracking** est défini comme une spécialisation de type pré-défini **HIST**. Cet historique stocke les événements issus des manipulations effectuées sur les objets.

- L'attribut **source** est utilisé pour mémoriser l'identificateur de l'objet source de la relation où l'événement a été déclenché. Dans le cas de TEMPO, l'attribut source sauvegarde l'identificateur de l'objet représentant le processus logiciel dans lequel l'événement a eu lieu.
- L'attribut **relation** sauvegarde le nom de la relation. Dans le cas de TEMPO, cet attribut garde le nom du rôle.
- L'attribut **destination** sauvegarde l'identificateur de l'objet sur lequel l'opération qui a déclenchée l'événement a été faite.
- L'attribut **command** sauvegarde le nom de la commande Adèle/Tempo.
- L'attribut **event** sauvegarde l'identificateur de l'événement.

VI.3.0.2 L'association des historiques avec les objets

L'attribut **DEFHIST** est définit comme étant le type d'historique que nous voulons associer à chaque type d'objet ou de relation dans le modèle de données. Par défaut le type d'objet **OBJECT** est associé au type d'historique **HIST**.

```
TYPEOBJECT OBJECT;
    ...
    DEFHIST := HIST;
END OBJECT;
```

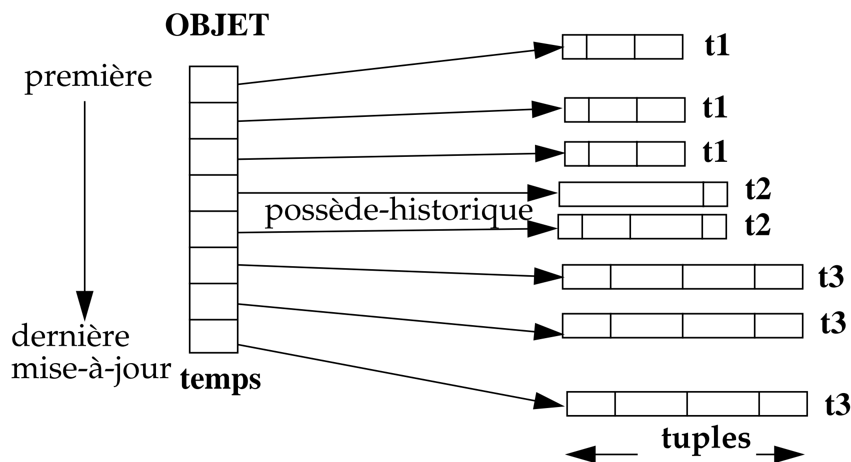
Nous pouvons re-définir cet attribut pour les sous-types d'objet ou de relation. Ainsi au type de relation **ROLE** nous pouvons associer le type d'historique **EventTracking** de la manière suivante :

```
TYPERELATION ROLE;
    DOMAIN
       [ type = PROCESS ] -> [ TYPE = OBJECTROLE ]
    DEFATTRIBUTE
       DEFHIST := EventTracking;
END ROLE;
```

VI.3.0.3 La manipulation des historiques

Les historiques sont stockés dans la base de données Adèle comme des tuples au sens des bases de données relationnelles (voir figure VI.3).

Figure VI.3 Manipulations des historiques



ti ::= type de l'historique

La manipulation de ces tuples se fait à l'aide des opérateurs suivantes :

- **mkhist** pour la création des historiques.
- **rmhist** pour la suppression des historiques.
- **lshist** pour la consultation des historiques. Cette opération est similaire à une requête SQL utilisée dans les bases de données relationnelles.

VI.4 Le gestionnaire de processus

Le gestionnaire de processus permet de gérer les contraintes d'intégrité et de contrôler la cohérence des opérations sur les objets et sur les relations. Ce gestionnaire supporte un modèle d'activités basé sur des règles temporelles événement-condition-action (Belkhatir et al., 1991; Belkhatir et al., 1993). Pour permettre une exécution efficace de ces règles, ce gestionnaire utilise un mécanisme de déclencheur ("*trigger*") fortement intégré au gestionnaire d'objets.

VI.4.1 Les règles événements-condition-action

VI.4.1.1 Les événements

Contrairement à d'autres systèmes tels que HiPac et Damokles, dans le système Adèle un événement modélise un changement d'état de la base d'objets ainsi qu'une condition sur cet événement.

Un événements survient chaque fois qu'une action est exécutée sur un objet dans la base de données. Cet événement est défini par l'action, ses paramètres et ses options. Les événements pertinents sont déclarés de la façon suivante :

```
DEFEVENT
    e1 = (command = delete, state = official);
    e2 = (command = replace);
```

L'événement **`e1`** est généré à chaque application de la commande de **`suppression`** sur un objet dont l'état est **`officiel`**. L'événement **`e2`** survient également lorsqu'une révision d'un objet logiciel est créée par l'application de la commande **`replace`**.

VI.4.1.2 Les actions

Pour permettre la manipulation des objets et des relations stockés dans la base de données, Adèle fournit un ensemble de commandes. L'utilisateur peut associer plusieurs commandes dans une *action*. Une action est donc un ensemble de commandes contrôlées par un ensemble d'instructions de contrôle (IF-THEN-ELSE, WHILE-DO, DO-UNTIL et FOREACH-DO). Une action ressemble à un programme "*Shell-Unix*". Par exemple :

```
DEFACTION FailOfficial;
    IF (%name.state == official) THEN {
       echo "Cannot replace an official object";
       ABORT; };
END FailOfficial;

DEFACTION Compile
    IF (%name.state != compiled) THEN
       IF `cc -c -g %name` THEN
         replace_binary %name
       ELSE ABORT;
END Compile;
```

VI.4.1.3 Les règles

Les règles forment la partie active du langage Adèle. Une règle associe toujours un événement à une action dans un type d'objet ou de relation. Par exemple :

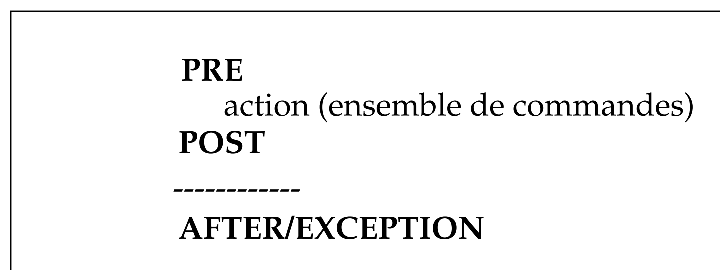
```
TYPEOBJECT Body ISA Cobj;
    . . .
    RULES
(1)    PRE ON e1 DO
         ABORT;
(2)    PRE ON e2 DO
         Compile;
(3)    POST ON e2 DO
         PutBinary %name;
(4)    EXCEPTION ON e2 DO
         modify_attribut %name -a state not_compiled;
```

```
(5)     AFTER ON e2 DO
          mail ~%name%author -s "body changed";
```

Le texte ci-dessus indique que :

1. un objet dans un état officiel ne peut pas être détruit;
2. un code source doit être d'abord compilé avant qu'il ne soit stocké dans la base de données.
3. dès qu'un code source est stocké dans la base, son code binaire doit l'être aussi.
4. si la compilation d'un code source échoue, son état devient `not_compiled`;
5. une fois le code source compilé et stocké dans la base avec son code binaire, un message est envoyé à l'auteur de l'objet pour l'avertir de cette modification.

Comme montré dans cet exemple, les actions associées aux règles peuvent être exécutées avant ou après le déclenchement de l'événement, de la façon suivante :



A chaque règle est associé un mode de couplage indiquant l'ordre dans lequel elles doivent être exécutées (voir figure VI.4). Les modes de couplage sont décrits ci-dessous **:**

mode = **PRE**. Lorsqu'un événement est signalé et sa condition évaluée l'action est éventuellement exécutée immédiatement avant l'exécution même de l'opération ayant produit l'événement. Ce mode de couplage permet de vérifier l'état de la base d'objets avant l'exécution de l'action et éventuellement ne pas l'exécuter.

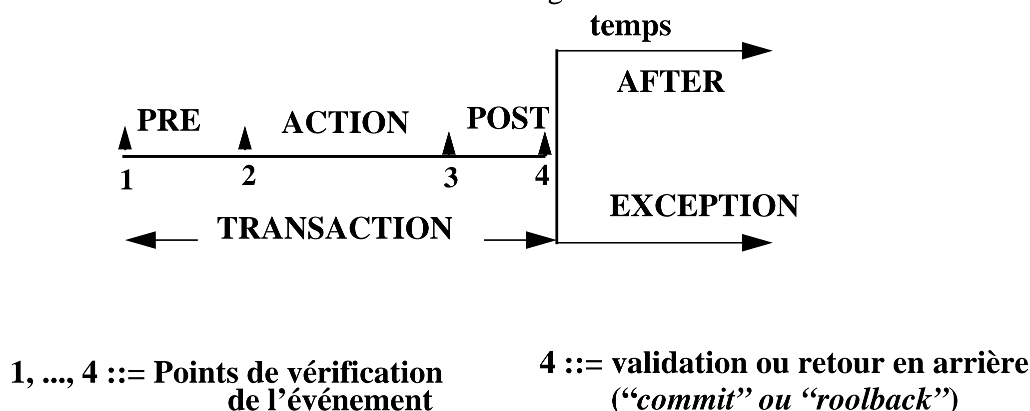
mode = **POST**. Lorsqu'un événement est signalé et sa condition évaluée l'action est éventuellement exécutée à la fin de la commande. Ce mode de couplage permet de vérifier les effets propagés sur l'état de la base d'objets après

l'exécution de l'action et éventuellement de la défaire (restauration de l'état de la base après l'exécution de l'action)

mode = AFTER. Lorsqu'un événement est signalé, les actions définies en mode "`after`" sont évaluées après la validation de l'opération en cours. Quelque soient les conditions d'exécution des actions en mode "`after`", l'action est validée.

mode = EXCEPTION. Lorsqu'un événement est déclenché et que les règles en exécution sont avortées, les règles définies dans la clause EXCEPTION sont exécutées.

Figure VI.4 L'exécution des règles ECA et les transactions courtes.



VI.4.1.4 Les règles sur les relations

Les règles associées à une relation sont déclenchées à chaque fois qu'une action est exécutée sur une relation (créer, détruire une relation,...). Elles sont utilisées pour gérer les actions effectuées sur des objets liés par des relations. On peut ainsi ajouter pour chaque objet un comportement selon les relations le liant à d'autres objets; c'est ainsi que sont gérés les agrégats par exemple.

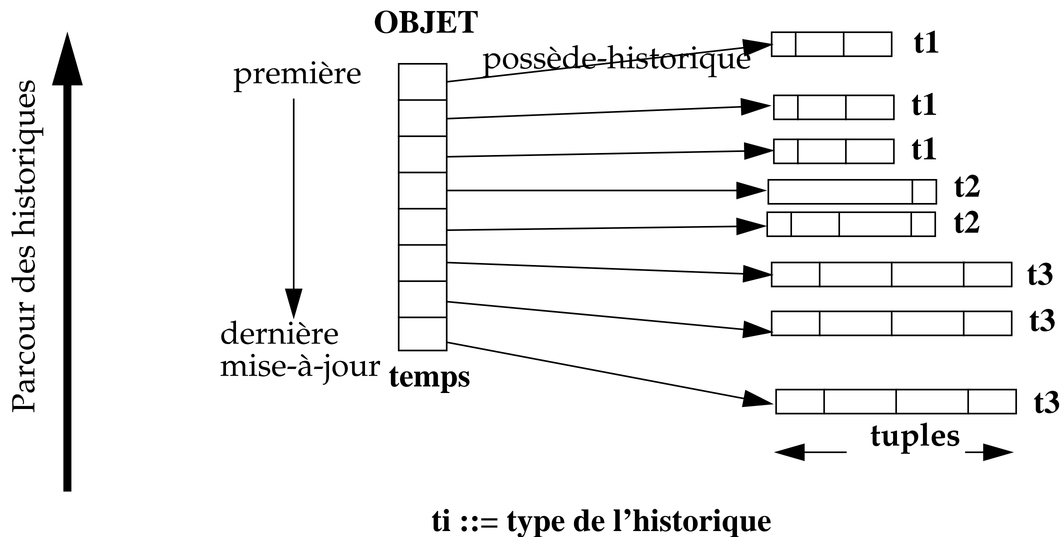
Une action effectuée sur un objet X déclenche les règles ECA définies sur les relations dont l'objet X est source (elles sont identifiées par le mot-clé ORIGIN) et les règles ECA définies sur les relations dont l'objet X est destination (elles sont identifiées par le mot-clé DEST).

VI.4.2 Les événements sur le passé

Basé sur les historiques des objets, le gestionnaire de processus peut analyser les règles TECA sur le passé. Comme décrit dans la figure VI.5, les historiques sont stockés dans la base de données en gardant l'ordre chronologique des mises-à-jour. Pour analyser les

règles TECA, l'historique de l'objet sur lequel la règle TECA a été déclenchée est parcouru depuis la dernière mise-à-jour jusqu'à la première mise-à-jour.

Figure VI.5 Parcours des historiques pour les règles TECA.



La règle TECA est considérée comme ayant été satisfaite lorsque la condition associée à l'événement temporel a été satisfait au cours du processus de parcours de l'historique.

VI.5 Implémentation de TEMPO : état de lieu

L'implémentation de TEMPO comporte les aspects suivants :

1. La réalisation d'un compilateur permettant de traduire les concepts de processus, rôle, fragment de processus et connexion vers les concepts d'Adèle. Ce compilateur a été réalise en utilisant les outils Lex, Yacc et C.

Un travail est visant l'incorporation des concepts de processus coopérant et de vue d'objets dans le noyau d'Adèle dans le cadre du projet ESPRIT PERFECT un cours de spécification. Cela impliquera l'extension du modèle de données d'Adèle pour prendre en compte ces concepts ainsi que la modification du noyau. Ces modifications donneront probablement lieu à une nouvelle version d'Adèle.

2. L'implémentation d'une bibliothèque de fonctions permettant de manipuler les concepts de TEMPO. Les fonctions de cette bibliothèque utilisent l'interface programmatique d'Adèle. Cette bibliothèque comprend les opérations de mise-à-jours des modèles de processus logiciels ainsi que les

fonctions de manipulation des processus, rôles, fragments de processus, connexions et gestion des environnements de travail.

3. La spécification de la base d'historiques. La spécification détaillée des historiques est en cours de validation au sein de l'équipe et doit être fournie avec la nouvelle version d'Adèle.

VI.6 Conclusion

Nous avons décrit dans ce chapitre les deux principaux composants de TEMPO. Le gestionnaire de ressources utilise la base de données Adèle et le gestionnaire de processus adopte le système de déclenchement (*"trigger")* et l'étend avec la capacité de manipuler des contraintes temporelles.

La base de données Adèle étendue par la capacité de gérer les historiques des objets, fournit un noyau de fonctionnalités permettant la gestion des processus logiciels, à savoir :

1. gérer les versions des objets;

2. supporter des multiples utilisateurs.

3. permettre la distribution des objets sur un réseau local;

4. fournir une architecture client-serveur. Dans la réalisation actuelle, TEMPO est un client d'Adèle.

5. modéliser des objets complexes. Le modèle de données fournit des concepts venant de l'approche orienté objet (héritage multiple, encapsulation, etc).

Cependant, certaines difficultés ont été rencontrées lors de la mise en oeuvre du prototype de TEMPO sur le système Adèle, à savoir :

1. Le langage utilisé pour programmer les actions dans les règles ECA est de bas niveau. Contrairement aux langages persistants orientés-objet, comme par exemple CO2 et O++, qui fournissent un langage fortement typé proche de C++, le langage Adèle demeure un langage non typé très proche du langage Unix.

Une étude a été lancé dans l'équipe dans le cadre d'un projet de DEA dans le but de proposer un nouveau langage où la sémantique pourra être validée statiquement.

2. La modélisation des objets complexes n'est pas aisée car le modèle ne fournit pas explicitement le concept d'attribut de composition. L'utilisateur est obligé d'utiliser la structure figée fournie par le type **Cobj**. La tâche de modélisation des produits logiciels est par conséquent plus difficile à mettre en oeuvre.

En effet, un autre modèle de données est en cours de développement dans le cadre d'une thèse de doctorat au sein de l'équipe. L'objectif est de supporter de manière plus orthogonale les objets composites en intégrant à ceux-ci le concept de versionnement.

CHAPITRE VII Conclusion et Perspectives

En guise de conclusion, nous décrivons la contribution de ce travail par rapport aux travaux dans ce domaine. Puis, nous présentons les perspectives et les nouveaux thèmes de développement et de recherche abordés actuellement dans notre équipe et dont TEMPO est l'élément fédérateur dans le domaine de processus logiciel.

VII.1 Contribution du travail

Au vu du modèle conceptuel proposé par (Wasserman, 1989) analysé dans la section I.1.2, notre travail porte essentiellement sur les points 3 (intégration par le contrôle) et 4 (intégration par le processus). Un formalisme exécutable a été conçu permettant de décrire différents modèles de processus logiciels coopérants.

Les aspects multi-comportementaux liées à l'utilisation des objets par les processus coopérants sont définies via le ***concept de rôle d'objets***. Une maquette a été construite pour supporter ce concept au-dessus d'Adèle. Un compilateur a été également bâti pour traduire les définitions de rôle dans TEMPO vers les concepts propre à la base Adèle.

Les contraintes d'utilisation des objets dans les processus logiciels sont décrites via les ***règles temporelles événement-condition-action*** (TECA) associées aux types de rôles. Pour cela, nous avons adopté les règles événement-condition-action d'Adèle en les étendant avec une modalité temporelle. Pour supporter l'interprétation des règles TECA, un gestionnaire des historiques d'objets a été spécifié. Ce gestionnaire est intégré au mécanisme de déclencheurs d'Adèle.

Les politiques gouvernant la coopération entre processus logiciels sont spécifiées à l'aide du ***concept de connexions actives et programmables***. Les échanges de messages entre les processus coopérants sont également contrôlés par des règles TECA.

La gestion de la cohérence des objets manipulés par les activités de longue durée est faite par le ***concept d'environnement de travail***. L'union entre les connexions et les environnements de travail permet de supporter l'exécution des processus coopérants ainsi que le partage d'objets entre ces processus.

Les aspects liés à l'évolution des processus logiciels sont également abordés. Nous avons montré comment cette dimension est intégrée aux concepts et mécanismes à la base de mon approche.

VII.2 Perspectives

A partir de cette étude, des activités de développement et de recherche centrées sur Adèle/TEMPO ont démarré ou sont en voie d'initialisation. Elles son conduites selon deux axes : un volet recherche et un volet développement.

VII.2.1 Des activités de recherche

De nouveaux horizons sont apparus suite à cette étude. On peut résumer comme suit les principaux travaux envisagés :

1. TEMPO et la gestion de l'évolution des produits et des processus logiciels. Comme les activités de génie logiciel ont une longue durée, les politiques de coordination et de synchronisation peuvent changer au cours de leur exécution. Le modèle intégré permettant la description des processus et des produits logiciel doit offrir des capacités d'évolution et adaptation. Nous avons à peine abordé ce point. L'utilisation des rôles comme concept pour gérer l'évolution et l'adaptation et poursuit actuellement dans l'équipe Adèle dans le cadre d'un thèse de doctorat
2. TEMPO et le modèle données Adèle. Le but ici est d'unifier les concepts de rôle pour structurer les produits et les processus logiciels. Dans ce dernier cas, on devra vraisemblablement se tourner vers des solutions de modèles à bases de rôle des d'objets pour le support du produit. Ce qui permet d'uniformiser les visions processus et produit. Le niveau processus est un niveau d'abstraction au dessus du niveau du produit permettant de supporter les activités en les organisant autour du concept de rôle.
3. TEMPO et l'inter-opérabilité des outils. Un AGL doit permettre l'intégration aisée des nouveaux outils. Pour que cela soit possible, des mécanismes sont nécessaires pour l'activation, le contrôle, la gestion, la surveillance et la communication entre les outils. Au vu des travaux récents dans le domaine,

un support formel à l'inter-opérabilité des outils se fait sentir. Ce domaine de recherche a commencé a peine à être traité de manière conceptuelle par la communauté de génie logiciel. Aujourd'hui on trouve de nombreux outils supportant l'inter-opérabilité des outils, comme par exemple l'outil Field. Cependant, peut d'effort a été consacré sur la formalisation de l'inter-opérabilité en proposant un modèle de communication entre les outils.

VII.2.2 Des activités de développement

La première priorité est la valorisation de l'étude décrite dans ce rapport par la réalisation d'un prototype industrielle capable de supporter les processus logiciels en vraie grandeur. Les concepts proposés par TEMPO doivent être intégrer dans le noyau d'Adèle permettant ainsi l'utilisation transparente des concepts de rôles, processus, connexion et environnement de travail pour les applications construits sur Adèle

La nature fortement active des processus logiciels fait sentir la nécessité d'une interface utilisateur graphique conviviale pour TEMPO. L'interface proposée va dans le sens d'une conception d'une interface de style objet ("pointer et cliquer") permettant aux utilisateurs d'exécuter leurs activités avec l'aide d'un support graphique.

Pour conclure, il faut dire que les points énoncés ainsi que leur intégration constituent un plan de recherche ambitieux qui devrait amener Adèle/TEMPO vers une plateforme où des environnements de développement et de maintenance pourront être aisément bâtis. les ateliers de génie logiciel dites centrés processus sont à notre avis le challenge de la décennie à venir pour.

CHAPITRE VIII Références

Adiba, M. and Collet, C. (1993). *Objets et Bases de Données : Le SGBD O2*. Hermès.

Albano, A., Bergamini, R., Ghelli, G., and Orsini, R. (1993). An object data model with roles. In *Proc. of the 19th Int'l Conf. on Very Large Data Bases*. Dublin, Ireland, August 1993. pages 667-675.

Ambriola, V., Ciancarini, P., and Montangero, C. (1990). Software process enactment in Oikos. In *Proc. of the 4th ACM SIGSOFT Symposium on Software Development Environments*, volume 15 of *ACM SIGSOFT Software Engineering Notes*, Irvine, CA.

Ambriola, V. and Montangero, C. (1992). Oikos at the age of threee. In (Derniame, 1992), pages 84–93.

Arbaoui, S. (1993). *Peace*. Thèse de doctorat, Université Pierre Mendès-France, Laboratoire CRISS, Grenoble, France.

Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., and Zdonik, S. (1989). The object-oriented database system Manifesto. In *Deductive and Object Oriented Databases*, Kyoto, Japao.

Bachman, C. and Daya, M. (1977). The role concept in data models. In *Proc. of the 3rd Int'l Conf. on Very Large Data Bases*, pages 464–467.

Balzer, R. (1991). Tolerating inconsistency. In *Proc. of the 13th Int'l Conf. on Software Engineering*, pages 158–165, Austin, TX. IEEE Press.

Balzer, R. (1993). Generic process support. In (Schafer 1993).

Bancilhom, B., Delobel, C., and Kanellakis, P. (1992). *Building an Object-Oriented Database: The Story of O2*. Morgan Kaufmann.

Banerjee, Kim, W., Kim, H. J., and Korth, H. F. (1987). Semantics and implementation of schema evolution in Object Oriented Databases. In *Proceeding of the ACM SIGMOD conference*.

Barghouti, N. and Kaiser, G. E. (1990). Modeling concurrency in rule-based development environments. *IEEE Expert*, 5(6):15-27.

Barghouti, N. S. (1992). Supporting cooperation in the Marvel process-centered SDE. In (Weber, 1992), pages 21–31.

Barghouti, N. S. and Kaiser, G. E. (1991a). Concurrency control in advanced database

applications. *ACM Computing Surveys*, 23(3):269–317.

Barghouti, N. S. and Kaiser, G. E. (1991b). Scaling-up rule-based development environments. In Ghezzi, C., editor, *Proc. of the 3rd European Software Engineering Conf.*, volume 550 of *LNCS*, pages 380–395, Milan, Italy. Springer-Verlag.

Barghouti, N. S. and Kaiser, G. E. (1992). Scaling-up rule-based development environments. *Int'l Journal on Software Engineering & Knowledge Engineering*, 2(1):59–78.

Basili, V. (1992). Software modeling and measurement: The goal/question/metric paradigm. Technical Report CS-TR-2956,UMIACS-TR-92-96, University of Maryland.

Basili, V. R. and Rombach, H. D. (1988). The TAME project: towards improvement-oriented softwre environments. *IEEE Transactions on Software Engineering*, 14(5):758–773.

Belkhatir, N. (1988). *Nomade : un noyau d'environnement pour la programmation globale*. Thèse de doctorat, INPG, Grenoble, France.

Belkhatir, N. and Estublier, J. (1987). Experience with a database of programs. *ACM SIGPLAN Notices*, 22(1):84–91.

Belkhatir, N., Estublier, J., and Melo, W. L. (1991). Adele 2: a support to large software development process. In (Dowson, 1991), pages 159–170.

Belkhatir, N., Estublier, J., and Melo, W. L. (1993). Software process model and work space control in the Adele system. In (Osterweil, 1993), pages 2–11.

Belkhatir, N. and Melo, W. L. (1992a). The object role software process model. In (Derniame, 1992).

Belkhatir, N. and Melo, W. L. (1992b). TEMPO: a software process model based on object context behavior. In *Proc. of the 5th Int'l Conf. on Software Engineering & its Applications*, pages 733–742, Toulouse, France.

Belkhatir, N. and Melo, W. L. (1993). Supporting software maintenace processes in TEMPO. In *Proc. of the Conference on Software Maintenance*, Montreal, Canada. IEEE Press.

Belkhatir, N., Melo, W. L., Estublier, J., and Nacer, M. A. (1992). 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.*, pages 165–172, Raleigh, NC.

Benali, K., Boudjlida, N., Charoy, F., Derniame, J.-C., Godart, C., Griffiths, P., Gruhn, V., Legait, A., Oldfield, D., and Oquendo, F. (1989). Presentation of the Alf projet. In *Proc. of the 1st Int'l Conf. on System Development Environments and Factories*, pages 75–90, Berlin.

Bézivin, J. (1990). Teaching object-oriented methods. In *Proc. of the 2nd Int'l Conf. Technology of Object-Oriented Languages and Systems*, Paris, France.

Boehm, B. (1988). A spiral model for software development and enhancement. *IEEE Computer*, pages 61–72.

Boudier, G., Minot, R., and Thomas, I. M. (1988). An overview of PCTE and PCTE+. In Henderson, P., editor, *Proc. of the 3rd ACM Software Enginnering Symposium on Practical Software Development Environments*, volume 24 of *ACM SIGPLAN Notices*, pages 107–109, Boston, MA.

Bratko, I. (1990). *Prolog — Programming for Artificial Intelligence*. Addison-Wesley. 2nd ed.

Brown, A. W. and MacDermid, J. A. (1992). Learning from IPSEs mistakes. *IEEE Software*, 8(2):23–28.

Brownston, L., Farrell, R., Kant, E., and Martin, N. (1985). *Programming Expert Systems in OPS5*. Addison-Wesley.

Bruynooghe, R., Parker, J., and Rowles, J. (1991). PSS: a system for process enactment. In (Dowson, 1991), pages 128–141.

Bruynooghe, R. F., Greenwood, R. M., Robertson, I., Sa, J., Snowdon, R. A., and Warboys, B. C. (1993). Towards a total process modelling system: a case study using ISPW-6. In (Derniame, 1993). To appear.

Butterworth, P., Otis, A., and Stein, J. (1991a). The Gemstone object database. *Communications of the ACM*, 34(10):64–77.

Butterworth, P., Otis, A., and Stein, J. (1991b). The gemstone object database management. *Comm. of the ACM*, 34(10):64–77.

B.Zdonik, S. (1990). Object-oriented type evolution. In Kim, W., editor, *Advances in Databases Programming Languages*, pages 277–293. ACM Press.

Cagan, M. R. (1990). The H. P. SoftBench environment: an architecture for a new generation of software tools. *Hewlett-Packard Journal*, 41(3).

Canals, G., Boudjlida, N., Derniame, J.-C., Godart, C., and Lonchamp, J. (1993). A short tour through ALF. In (Derniame, 1993). To appear.

Chen, P. S. (1976). The entity-relationship model- towards a unified view of data. *ACM Transaction on Database Systems*, 1(1):9–36.

Chikofsky, E. J. (1988). CASE: reliability engineering for information systems. *IEEE Software*, 5(2):8–16.

Clemm, G. M. and Osterweil, L. (1990). A mechanism for environment integration. *ACM Transactions on Programming Languages and Systems*, 12(1):1–15.

Conradi, A. R., Didriksen, T. M., Karlsson, E.-A., Hallsteinsen, S., and Holager, P. (1989). Change oriented versioning in a software engineering. *ACM SIGSOFT Software Engineering Notes*, 17(7):56–65. Proc. of the 2nd Int'l Workshop on Sofware Configuration Management.

Conradi, R., Fernstrom, C., and Fuggetta, A. (1993). A conceptual framework for envolving software processes. In (Derniame, 1993). To appear.

Conradi, R. and Holager, P. (1990). Change-oriented versioning: rationale and evaluation. In *Proc. of the 3rd Int'l Workshop on Software Engineering and its Applications*, pages 97–109, Toulouse, France.

Conradi, R., Jaccheri, M. L., Mazzi, C., Aarsten, A., and Nguyen, M. N. (1992). Design,

use, and implementation of Spell, a language for software process modeling and evolution. In (Derniame, 1992).

Conradi, R., Liu, C., and Jaccheri, M. (1991a). Process modeling paradigms: an evaluation. In *Proc. of the 7th Int'l Software Process Workshop*, San Francisco, CA.

Conradi, R., Osjord, E., Westby, P., and Liu, C. (1991b). Initial software process management in Epos. *IEE Software Engineering Journal*, 6(5):275–284.

Conradi, R., Osjord, E., Westby, P., and Liu-Beijing, C. (1990). Software process modelling in EPOS: design and initial implementation. In *Proc of the 3rd Int'l Workshop on Software Engineering and its Applications*, pages 365–381, Toulouse, France.

Courington, W. (1989). *The Network Software Environment*. Sun Microsystems, Inc.

Curtis, B., Kellner, M. I., and Over, J. (1992). Process modeling. *Communications of the ACM*, 35(9):75–90.

DeRemer, F. and Kron, H. (1976). Programming-in-the-large verus programming in the small. *IEEE Transactions on Software Engineering*, 2:80–86, June.

Derniame, J.-C., editor (1992). *Proc. of the 2nd European Workshop on Software Process Technology*, volume 635 of *LNCS*, Trondheim, Norway. Springer-Verlag.

Derniame, J.-C., editor (1993). *Software process research in Europe: The Promoter project*. To appear.

Derniame, J.-C., Godart, C., Gruhn, V., and Lonchamp, J. (1992). Process-Centered IPSEs in ALF. In G. Forte, N. H. M. and Muller, H. A., editors, *Proc of the 5th Int'l Workshop on Computer-Aided Software Engineering (CASE'92)*, pages 179–190, Montréal, Québec, Canada. IEEE Computer Society Press.

Dewan, P. and Riedl, J. (1993). Toward computer-supported concurrent software engineering. *IEEE Computer*, 26(1):17–27.

Dittrich, K. (1989). The Damokles database system for design applications: its past, its present, and its future. In Bennett, K. H., editor, *Software Engineering Environments: Research and Practice*, pages 151–171. Ellis Horwood Books, Durhan, UK.

Dittrich, K., Gotthard, W., and Lockemann, P. C. (1987). Damokles – a database system for software engineering environments. In R. Conradi, T.M. Didriksen, D. W., editor, *IFIP Int'l Workshop on Advanced Programming Environments*, volume 244 of *LNCS*, pages 353–371. Spring-Verlag, Trondheim, Norway.

Dowson, M., editor (1991). *Proc. of the First Int'l Conf. on the Software Process*, Redondo Beach, CA. IEEE Computer Society Press.

Dowson, M. (1993). Process variables and process change. In (Schafer, 1993).

Dowson, M., Nejmeh, B., and Riddle, W. (1991). Fundamental software process concepts. In (Fuggetta et al., 1991), pages 16–37.

Emmerich, W., Junkermann, G., Peuschel, B., Schafer, W., and Wolf, S. (1991). Merlin: knowledge-based process modeling. In Proc. of the *First European Workshop on Software Process Modeling*, pages 181–186, Milan, Italy.

Estublier, J. and Belkhatir, N. (1988). Nomade: Un noyau de maintenance et de

developpement. In *Proc. of the Workshop on Software Engineering and its applications*, Toulouse.

Estublier, J. and Belkhatir, N. (1989). Un gestionnaire d'activities pour la programmation globale pour nomade. In *Proc. of the 2nd Int'l Workshop on Software Engineering and its applications*, Toulouse.

Estublier, J., Belkhatir, N., Nacer, M., Melo, W., and Wadouh, K. (1993). Support a l'evolution des procedes de production logiciel dans un AGL centré processus. Technical report, L. G. I., Adele project.

Estublier, J. and Favre, J. (Sept 20–22, 1989). Structuring large versioned software products. In *Proc. of the 30th Annual Int'l Conf. Computer Software and Applications (COMPSAC'89)*, Orlando, FL.

Estublier, J., Ghoul, S., and Krakowiak, S. (1984). Premilinary experience with a configuration control system for modular programs. *ACM SIGPLAN Notes*, 9(3):149–156.

Feiler, P. (1991a). Configuration management models in commercial environments. Technical Report CMU/SEI-91-TR-7, Carnegie-Mellon University, Software Enginnering Institure.

Feiler, P. H., editor (1991b). *Proc. of 3rd Int'l Workshop on Software Configuration Management*, Trondheim, Norway. IEEE Press.

Feiler, P. H. and Humphrey, W. S. (1993). Software process development and enactment: Concepts and definitions. In (Osterweil, 1993), pages 28–40.

Feldman, S. I. (1979). Make: a program for maintaining computer programs. *Software–Practice and Experience*, 9(4):255–265.

Fernstrom, C. (1993). Process Weaver: adding process support to Unix. In (Osterweil, 1993), pages 12–26.

Fernstrom, C. and Ohlsson, L. (1991). Integration needs in process enacted environments. In (Dowson, 1991), pages 142–158.

Floyd, C., Reisin, F., and Schmidt, G. (1989). STEPS to software development with users. In *Proc. of the 2nd European Software Engineering Conference*, Warwick, England.

Fuggetta, A., Ghezzi, C., and Conradi, R., editors (1991). *Proc. of the 1st European Workshop on Software Process Modeling*, Milan, Italy. AICA Press.

Garlan, D. and Ehsan, I. (1990). Adaptable tool integration policies for integrated environments. In *Proc. of the 4th Int'l Workshop on Computer-Aided Software Engineering*, Irvine, CA. ACM SIGSOFT 15(6):1–10, December 1990.

Gatziu, S., Geppert, A., and Dittrich, K. R. (1991). Integrating active concepts into an object-oriented database systems. In *Proc. of the 3rd Int'l Workshop on Database Programming Languages: Bulk Types & Persistent Data*, pages 399–415, Nafplion. Morgan Kaufmann.

Gehani, N. H., Jagadish, H. V., and Shmueli, O. (1992). Event specification in an active object-oriented database. In Stonebraker, M., editor, *Proc. of the ACM SIGMOD 92*, volume 21, no. 2 of *ACM SIGMOD Record*, pages 81–90. ACM Press.

Girod, X. (1991). *Conception par objets*. PhD thesis, Université Joseph Fourier.

Godart, C. (1993). *Contribution á la modeélisation des procédes de fabrication de logiciel : support au travail coopératif*. Thèse de doctorat d'état, Université de Nancy I, Nancy, France.

Goldman, N. and Narayanaswamy, K. (1992). Software evolution through interative prototyping. In Montgomery, T., editor, *Proc. of the 14th Int'l Conf. on Software Engineering*, Melbourne, Australia. IEEE Computer Society Press.

Habermann, A. and Notkins, D. (1986). Gandalf: Software development environment. *IEEE Transaction on Software Engineering*, SE-12(12):1117–1127.

Hahn, U., Jarke, M., and Rose, T. (1991). Teamwork support in a knowledge-based information system environment. *IEEE Transactions on Software Engineering*, 17(5):467–482.

Heimbigner, D. (1992). The ProcessWall: a process state server approach to process programming. In (Weber, 1992), pages 159–168.

Henderson, P., editor (1988). *Proc. of the 3rd ACM SIGSOFT Software Engineering Symposium on Software Practical Development Environments*, volume 24 of *ACM SIGSOFT Software Engineering Notes*, Boston, MA.

Horowitz, E. and Williamson, R. (1986). Sodos: a software documentation support environment. *IEEE Transactions on Software Engineering*, SE-12(8).

Hudson, S. E. and King, R. (1988). The Cactis project: database support for software environments. *IEEE Transactions on Software Engineering*, 14(6):201–260.

Huff, K. E. (1989). *Plan-Based Intelligent Assistance: An Approach to Supporting the Software Development Process*. PhD thesis, University of Massachusetts.

Huff, K. E. and Kaiser, G. E. (1991). Change in the software process. In (Thomas, 1991), pages 10–13.

Huff, K. E. and Lesser, V. R. (1988). A plan-based intelligent assistant that supports the software development process. In (Henderson, 1988).

Humphrey, W. (1988). Characterizing the software process: a maturity framework. *IEEE Software*, pages 73–79.

Humphrey, W. S. and Kellner, M. I. (1989). Software process modeling: principles of entity process models. In *Proc. of the 11th Int'l Conf. on Software Engineering*, pages 331–342, Pittsburgh, Pennsylvania.

Jarwah, S. (1992). *Un modéle Générique pour la Gestion des Informations Complexes et Dynamiques*. Thèse de doctorat, Université Joseph Fourier, LGI, Grenoble, France.

Kaiser, G. E. (1990). A flexible transaction model for software engineering. In *Proc. of the 6th Int'l Conf. on Data Engineering*, pages 560–567, Los alamitos, CA. IEEE CS Press.

Kaiser, G. E., barghouti, N. S., Feiller, P. H., and Schwanke, R. W. (1988a). Database suport for knowledge-based engineering. *IEEE Expert*, 3(2):18–32.

Kaiser, G. E. and Ben-Shaul, I. Z. (1993). Process evolution in the Marvel environment. In (Schafer, 1993).

Kaiser, G. E., Feiller, P. H., and Popovich, S. (1988b). Intelligent assistance for software development and maintenance. *IEEE Software*, 5(3):40–49.

Katayama, T. (1989). A hierarchical and functional software process description and its enaction. In *Proc. of the 11th Int'l Conf. on Software Engineering*, pages 343–352, Pittsburgh, Pennsylvania.

Kim, W. (1988). Features of the Orion object-oriented DBMS. In Kim, W. and Lochovsky, E., editors, *Object-Oriented Concepts, Databases, and Applications*. Addison-Wesley.

Kim, W., Garza, N. B. J., and Woelk, D. (1991). A distributed object-oriented database system supporting shared and private databases. *ACM Transactions on Information Systems*, 9(1):31–51.

Kimball, J. (1992). The EIS execution engine and the AAA process engine. In Penedo, M. H., editor, *Process Sensitive SEE Archicture Workshop*, Boulder, CO.

Leblang, D. and Chase, R. P. (1985). Configuration management for large scale software development efforts. In *Workshop on Software Engeneering Environment for Programming in Large*, Harwichport, Massachussets.

Leblang, D. and Chase, R. P. (1988). Parallel building: experience with a case for workstations networks. In *Proc. of the Int'l Workshop on Software Version and Configuration Control*, Grassau, FRG.

Lehman, N. M. and Belady, L. A. (1985). *Program Evolution — Processes of Software Change*. Academic Press.

Lerner, B. S. and Habermann, A. N. (1990). Beyond schema evolution to database reorganization. In *ECOOP'90 Proceedings*.

Liu, C. (1990). A software process planner in Epos. In *Proc. of the Norsk Informatikk Konferanse 1990*, pages 203–215, Bergen, Norway.

Lonchamp, J. (1993). A structured conceptual and terminological framework for software process engineering. In (Osterweil, 1993), pages 41–53.

Lonchamp, J., Godart, J., and Derniame, J.-C. (1992). Les environnements intégrés de production de logiciel. *Technique et Science Informatique*, 11(1).

Long, F., editor (1989). *Proc. of Int'l Workshop on Software Engineering Environments*, volume 467 of *LNCS*, Chinon, France. Springer-Verlag, Berlin, 1990.

Madhavji, N. (1992). Environment evolution: The Prism model of changes. *IEEE Transactions on Software Engineering*, 18(5):380–392.

Madhavji, N. and Schafer, W. (1991). Prism — methodology and process-oriented environment. *IEEE Transactions on Software Engineering*, 17(12):1270–1283.

Madhavji, N. H. (1991). The process cycle. *IEE Software Engineering Journal*, 6(5):234–242.

Malm, C. C. and Conradi, R. (1991). Cooperating transactions against the Epos database. In (Feiler, 1991b), pages 98–101.

Marzullo, K. and Wiebe, D. (1986). A software system modelling facility. In *Proc. of the ACM Software Engineering Symposium on Practical Software Development Environments*, pages 121–130.

McCarthy, D. R. and Dayal, U. (1989). The architecture of an active database management system. In *Proc. of ACM SIGMOD 89*, pages 215–224, Portland, OR.

Mi, P. and Scacchi, W. (1990). A knowledge base environment for modeling and simulating software engineering processes. *IEEE Trans. Knowledge and Data Engineering*, 2(3):283–294.

Mi, P. and Scacchi, W. (1992). Process integration in CASE environments. *IEEE Software*, 8(2):45–53.

Miller, T. (1989). Configuration management with the NSE. In (Long, 1989), pages 99–106.

Minsky, N. H. (1991). Law-governed systems. *IEE Software Engineering Journal*, 6(5):285–302.

Montangero, C. and Ambriola, V. (1993). Process-centred software development environments in Oikos. In (Derniame, 1993). To appear.

Narayanaswamy, K. (1993). Enactment in a process-centered softwre engineering environment. In (Schafer, 1993).

Oberndorf, P. A. (1988). The commom Ada programming support environment (APSE) interface set (CAIS). *IEEE Transactions on Software Engineering*, 14(6).

Oivo, M. and Basili, V. R. (1992). Representing software engineering models: the TAME goal oriented approach. *IEEE Transactions on Software Engineering*, 18(10):886–898.

Oquendo, F., Boudier, G., Gallo, F., Minot, R., and Thomas, I. (1991). The PCTE+'OMS: A software engineering database system for supporting large-scale software developpement environments. In *Proc. of the 2nd Int'l Symp. on Database Systems for Advanced Applications*, Tokyo, Japan.

Oquendo, F., Zucker, J.-D., and Tassart, G. (1990). Support for software tool integration and process-centered software engineering environments. In *Proc. of the 3rd Int'l Workshop on Software Engineering and its Applications*, pages 135–155, Toulouse, France.

Osterweil, L., editor (1993). *Proc. of the 2nd Int'l Con.f on the Software Process*, Berlin, Germany. IEEE Press.

Osterweil, L. J. (1987). Software processes are software too. In *Proc. of the 9th Int'l Conf. on Software Engineering*, pages 2–13, Monterey, CA.

Peckhan, J. and Maryanski, F. (1988). Semantic data models. *ACM Computing surveys*, 20(3):153–190.

Pedersen, C. H. (1989). Extending ordinary inheritance schemes to include generalization. In *Proc. of the OOSPSLA'89*, New Orleans.

Penedo, M. (1986). Prototyping a project master data base for software engineering environments. In *Proc. of the 2nd ACM Software Engineering Symposium on Software Practical Development Environments*, SIGPLAN Notices, Palo Alto, CA.

Penedo, M. (1991). Acquiring experiences with the modeling and implementation of the project life-cycle process. *IEE Software Engineering Journal*, 6(5):285–302.

Penny, D. and Stein, J. (1987). Class modification in the Gemstone Object-Oriented

DBMS. In *Proc. of the ACM Conf. on Object-Oriented programming*, Orlando.

Perry, D. E. (1987). Software interconnection models. In *Proc. of the 9th Int'l Conf. on Software Engineering*, pages 61–69, Monterey, CA.

Perry, D. E. and Kaiser, G. E. (1987). Infuse: a tool for automatically managing and coordinating source changes in large systems. In *Proc. of the ACM Computer Science Conf.*, pages 17–19, St. Louis, Missouri.

Peuschel, B., Schafer, W., and Wolf, S. (1992). A knowledge-based software development environment supporting cooperative work. *Int'l Journal of Software Engineering and Knowledge Engineering*, 2(1):79–1–6.

Ramamoorthy, C., Garg, V., and Prakash, A. (1988). Support for reusability in Genesis. *IEEE Transactions on Software Engineering*, 14(8):1145–1154.

Ramanathan, J. and Sarkar, S. (1988). Providing customized assistance for software lifecycle approaches. *IEEE Transactions on Software Engineering*, 14(6):749–757.

Reiss, S. P. (1990). Connecting tools using message passing in the Field environment. *IEEE Software*, 7(4):57–66.

Richardson, J. and Schwartz, P. (1991). Aspects: Extending objects to support multiple, independent roles. In *Proc. of the Int'l Conf. on Management of Data*, volume 20 of *ACM SIGMOD Record*, pages 298–307.

Rochkind, M. (1974). The source code control system (SCCS). *IEEE Transactions on Software Engineering*, 1:370–376.

Roddick, J. F. (1992). A query language extension for databases supporting schema evolution. *ACM SIGMOD Record*, 21(3).

Romamoorthy, C. (1986). Programming in the large. *IEEE Transactions on Software Engineering*, 12(7):1145–1154.

Rombach, H. D. and Verlage, M. (1993). How to assess a software process modeling formalism from a project member's point of view. In (Osterweil, 1993), pages 147–158.

Rose, T., Jarke, M., Gocek, M., Maltzahn, C., and Nissen, H. W. (1991). A desision-based configuration process environment. *IEE Software Engineering Journal*, 6(5):332–346.

Rosenblum, D. S. and Krishnamurthy, B. (1991). An event-based model of software configuration management. In (Feiler, 1991b), pages 94–97.

Royce, W. (1970). Managing the development of large software systems. In *Proc. IEEE Western Computer Conf.*, Los Alamitos, CA. IEEE Computer Society Press.

R.Zicari (1989). A framework for schema updates in an Object Oriented database system. In *Proc. of the 7th Int'l Conf. on Data Engineering*, Kobe, Japan.

Sarkar, S. (1989). *The Design of a Software Environment Architecture Based on Executable Process Description*. PhD thesis, Ohio-State University.

Sarkar, S. and Venugopal, V. (1991a). A language-based approach to building CSCW systems. In *Proc. of the 24th Annual Hawaii Int'l Conf. on System Sciences*, pages 553–567, Kona, HI. IEEE Computer Society, Software Track, v. II.

Sarkar, S. and Venugopal, V. (1991b). Transaction mechanisms for software

environment databases. In *Proc. of the 24th Annual Hawaii Int'l Conf. on System Sciences*, pages 511–518, Kona, HI. IEEE Computer Society, Software Track, v. II.

Schafer, W., editor (1993). *Proc. of the 8th Int'l Software Process Workshop*, Germany. IEEE Computer Society Press.

Schwanke, R. W. and Kaiser, G. E. (1988). Living with inconsistency in large systems. In Winkler, J. F. H., editor, *Int'l Workshop on Software Version and Configuration Control*, Grassau, Germany. B. G. Teubner, Stuttgart, 1988.

Shilling, J. and Sweeney, P. (1989). Three steps to view: Extending the object-oriented paradigms. In *Proc. of the OOPSLA'89*, volume 24, no. 10 of *ACM SIGPLAN Noticies*, pages 353–361.

Simmonds, I. (1989). Configuration management in the PACT software enginnering environment. *ACM Software Engineering Notes*, 14(7):118–121.

Skarra, A. H. and Zdonik, S. B. (1986). Type evolution in an object-oriented database. In Shriver, B. and Wegner, P., editors, *Research Directions in Object-Oriented Programming*.

Snowdon, R. A. (1989). An introduction to the Ipse 2.5 project. In (Long, 1989).

Sugiyama, Y. (1993). The process programming system OPM. In (Schafer, 1993).

Sugiyama, Y. and Horowitz, E. (1990). Language support for object process modeling. In *Proc. of the 6th Int'l Software Process Workshop*, Hakodate, Japan.

Sugiyama, Y. and Horowitz, E. (1991). Building your own software development environment. *IEE Software Engineering Journal*, 6(5):317–331.

Sutton, S. M. (1993). Opportunities, limitations, and tradeoffs in process programming. In (Osterweil, 1993), pages 135–146.

Sutton, S. M., Heimbigner, D., and Osterweil, L. J. (1990). Language constructs for managing change in process-centered environments. In Taylor, R., editor, *Proc. of the 4th ACM Software Engineering Symposium on Software Practical Development Environments*, volume 15 of *ACM SIGSOFT Software Engineering Notes*, pages 206–217, Irvine, CA.

Sutton, S. M., Ziv, H., Heimbigner, D., Yessayan, H., Maybee, M., Osterweil, L. J., and Song, X. (1991). Programming a software requirements-specification process. In (Dowson, 1991), pages 68–89.

Suzuki, M., Iwai, A., and Katayama, T. (1993). A formal model for re-execution in software process. In (Osterweil, 1993), pages 84–99.

Suzuki, M. and Katayama, T. (1991). Meta-operations in the process model HFSP for the dynamics and flexibility of software process. In (Dowson, 1991), pages 202–217.

Taylor, R. N. and et al (1988). Foundations for the Arcadia environment architecture. In (Henderson, 1988), pages 1–13.

Thomas, I. (1989a). PCTE interfaces: supporting tools in software engineering environments. *IEEE Software*, 6(6):15–23.

Thomas, I. (1989b). Version and configuration management on a software enginnering database. *ACM Software Engineering Notes*, 14(7):23–25.

Thomas, I., editor (1991). *Proc. of the 7th Int'l Software Process Workshop*, San

Francisco, CA. IEEE Computer Society Press.

Thomas, I. and Nejmeh, B. (1992). Definitions of tool integration in software engineering environments. *IEEE software*, 8(2):29–35.

Tichy, W. (1982). Design, implementation, and evaluation of a revision control system. In *Proc. of the 6th Int'l Conf. on Software Engineering*, Tokyo, Japan. IEEE Computer Society.

Tichy, W. (1985). Rcs — a system for version control. *Software—Practice and Experience*, 15:637–654.

Warboys, B. (1989a). The IPSE 2.5 project: a process model based architecture. In Bennett, K. H., editor, *Software Engineering Environments: Research and Practice*, pages 313–331. Ellis Horwood Books.

Warboys, B. (1989b). The Ipse 2.5 project: process modelling as the basis for a support envrionment. In Madhavji, N., Schafer, W., and Weber, H., editors, *Proc. of the 1st Int'l Conf. on System Development and Factories*, Berlin, Germany. Pitman Publishing, London, March 1990.

Wasserman, A. I. (1989). Tool integration in software engineering environments. In (Long, 1989).

Waters, R. C. (1989). Automated software management based on structural models. *Software–Practice and Experience*, 19(10):931–955.

Weber, H., editor (1992). *Proc. of the 5th ACM Software Engineering Symposium on Software Practical Development Environments*, volume 17, no. 5 of *ACM SIGSOFT Software Engineering Notes*. ACM Press.

Williams, L. (1988). Software process modeling: a behavioral approach. In *Proc. of the 10th Int'l Conf. on Software Engineering*, pages 174–186. IEEE Computer Society Press.

ANNEXE A Publications dans le cadre de la thèse

I.1 Introduction

Les résultats des recherches effectuées dans le cadre de cette thèse ont été diffusés sous la forme de publications à des revues et des conférences internationales.

1. N. Belkhatir et W. L. Melo.
TEMPO: a Process-Centered Software Developoment Environment.
In *Journal of Systems and Software*. A aparaître.

2. N. Belkhatir et W. L. Melo.
Supporting Software Maintenance Processes in TEMPO.
In *Proc. of the Conference on Software Maintenance*, Montreal, Canada. Sept. 1993. IEEE Press.

3. N. Belkhatir, J. Estublier, et W. L. Melo.
TEMPO: Enhancing O.O. Paradigm for Modeling Software Engineering Processes.
In W. Schafer, editor, *Proc. of the 8th Int'l Software Process Workshop*, Berlin, Germany, February 1993. IEEE Press.

4. N. Belkhatir, J. Estublier, et W. L. Melo.
Software Process Model and Work Space Control in the Adele System.
In L. Osterweil, editor, *Proc. of the 2nd Int'l Conf. on the Software Process*, pages 2–11, Berlin, Germany, February 1993. IEEE Press.

5. N. Belkhatir et W. L. Melo.
A Software Process Model Based on Object Context Behavior.
In *Proc. of the 5th Int'l Conf. on Software Engineering & its Applications*, Toulouse, France, December 7–11 1992.

6. N. Belkhatir et W. L. Melo.
An Object Driven Process Model.
In E. Gelenbe, editor, *Proc. of the 7th Int'l Symposium on Computer and Information Sciences*, pages 463–470, Antalya, Turkey, November 2–4 1992.

7. N. Belkhatir et W. L. Melo.
The Object Role Software Process Model.
In J.-C. Derniame, editor, *Proc. of the 2nd European Workshop on Software Process Technology*, volume 635 of *LNCS*, Trondheim, Norway, 7–8 September 1992. Springer-Verlag.

8. N. Belkhatir et W. L. Melo.
Supporting Software Processes: A Preliminary Evaluation of an Event Trigger Mechanism.
In M. H. Penedo, editor, *Process Sensitive SEE Archicture Workshop*, Boulder, CO, September 1992. Rmise Press.

9. N. Belkhatir, J. Estublier, et W. L. Melo.
Cooperative Work in Large–Scale Software Systems.
Journal of Software Maintenance: Research and Practice. Wiley.
A aparaître.

10. J. Estublier, N. Belkhatir, W. L. Melo et M. A. Nacer.
Process-Centered SEE and Adele.
In N. H. Madhavji G. Forte and F. Coalier, editors, *Proc of the 5th Int'l Workshop on Computer-Aided Software Engineering (CASE'92)*, pages 156–165,
Montréal, Québec, Canada, July 6–10 1992. IEEE Press.

11. N. Belkhatir, W. L. Melo, J. Estublier et M. A. Nacer
Supporting Software Maintenance Evolution Processes in the Adele System.
In C. M Pancake and D. S. Reeves, editors, *Proc. of the 30th Annual ACM Southeast Conf.*, pages 165–172, Raleigh, NC, April 8-10 1992. ACM Press.

12. N. Belkhatir, J. Estublier, et W. L. Melo.
Activity Coordination in a Software Production Kernel.
In I. Thomas, editor, *Proc. of the 7th Int'l Software Process Workshop*, San Francisco, CA, October 16–18 1991. IEEE Press.

13. N. Belkhatir, J. Estublier, et W. L. Melo.
Adele 2: a Support to Large Software Development Process.
In M. Dowson, editor, *Proc. of the First Int'l Conf. on the Software Process*, pages 159–170, Redondo Beach, CA, October 21–22 1991. IEEE Press.