NOMADE : Un noyau d'environnement pour la programmation globale

Nourredine Belkhatir

▶ **To cite this version:**

Nourredine Belkhatir. NOMADE : Un noyau d'environnement pour la programmation globale. Génie logiciel [cs.SE]. Institut National Polytechnique de Grenoble - INPG, 1988. Français. NNT : . tel-00010437v2

HAL Id: tel-00010437

https://theses.hal.science/tel-00010437v2

Submitted on 6 Oct 2005

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

Noureddine BELKHATIR

pour obtenir le grade de

DOCTEUR
DE L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE
(arrêté ministériel du 5 juillet 1984)

Spécialité «INFORMATIQUE»

OOOOO

NOMADE : UN NOYAU D'ENVIRONNEMENT POUR LA PROGRAMMATION GLOBALE

OOOOO

Thèse soutenue le 2 Décembre 1988 devant la commission d'examen

Président : J. MOSSIERE
Examinateurs : Y. CHIARAMELLA
J. ESTUBLIER
S. KRAKOWIAK
A. Van LAMSWEERDE

Thèse préparée au sein du laboratoire de Génie Informatique-IMAG

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Président : Georges LESPINARD **Année 1988**

Professeurs des Universités

BARIBAUD Michel	ENSERG	JOUBERT Jean-Claude	ENSPG
BARRAUD Alain	ENSIEG	JOURDAIN Geneviève	ENSIEG
BAUDELET Bernard	ENSPG	LACOUME Jean-Louis	ENSIEG
BEAUFILS Jean-Pierre	ENSEEG	LESIEUR Marcel	ENSHMG
BLIMAN Samuel	ENSERG	LESPINARD Georges	ENSHMG
BLOCH Daniel	ENSPG	LONGEQUEUE Jean-Pierre	ENSPG
BOIS Philippe	ENSHMG	LOUCHET François	ENSIEG
BONNETAIN Lucien	ENSEEG	MASSE Philippe	ENSIEG
BOUVARD Maurice	ENSHMG	MASSELOT Christian	ENSIEG
BRISSONNEAU Pierre	ENSIEG	MAZARE Guy	ENSIMAG
BRUNET Yves	IUFA	MOREAU René	ENSHMG
CAILLERIE Denis	ENSHMG	MORET Roger	ENSIEG
CAVAIGNAC Jean-François	ENSPG	MOSSIERE Jacques	ENSIMAG
CHARTIER Germain	ENSPG	OBLED Charles	ENSHMG
CHENEVIER Pierre	ENSERG	OZIL Patrick	ENSEEG
CHERADAME Hervé	UFR PGP	PARIAUD Jean-Charles	ENSEEG
CHOVET Alain	ENSERG	PERRET René	ENSIEG
COHEN Joseph	ENSERG	PERRET Robert	ENSIEG
COUMES André	ENSERG	PIAU Jean-Michel	ENSHMG
DARVE Félix	ENSHMG	POUPOT Christian	ENSERG
DELLA-DORA Jean	ENSIMAG	RAMEAU Jean-Jacques	ENSEEG
DEPORTES Jacques	ENSPG	RENAUD Maurice	UFR PGP
DOLMAZON Jean-Marc	ENSERG	ROBERT André	UFR PGP
DURAND Francis	ENSEEG	ROBERT François	ENSIMAG
DURAND Jean-Louis	ENSIEG	SABONNADIERE Jean-Claude	ENSIEG
FOGGIA Albert	ENSIEG	SAUCIER Gabrielle	ENSIMAG
FONLUPTJean	ENSIMAG	SCHLENKER Claire	ENSPG
FOULARD Claude	ENSIEG	SCHLENKER Michel	ENSPG
GANDINI Alessandro	UFR PGP	SILVY Jacques	UFR PGP
GAUBERT Claude	ENSPG	SIRIEYS Pierre	ENSHMG
GENTIL Pierre	ENSERG	SOHM Jean-Claude	ENSEEG
GREVEN Hélène	IUFA	SOLER Jean-Louis	ENSIMAG
GUERIN Bernard	ENSERG	SOUQUET Jean-Louis	ENSEEG
GUYOT Pierre	ENSEEG	TROMPETTE Philippe	ENSHMG
IVANES Marcel	ENSIEG	VEILLON Gérard	ENSIMAG
JAUSSAUD Pierre	ENSIEG	ZADWORNY François	ENSERG

Professeur Université des Sciences Sociales (Grenoble II)

BOLLIET Louis

Personnes ayant obtenu le diplôme d'HABILITATION A DIRIGER DES RECHERCHES

BECKER Monique
BINDER Zdenek
CHASSERY Jean-Marc
CHOLLET Jean-Pierre
COEY John
COLINET Catherine
COMMAULT Christian
CORNUEJOLS Gérard
COULOMB Jean- Louis
DALARD Francis
DANES Florin
DEROO Daniel
DIARD Jean-Paul
DION Jean-Michel
DUGARD Luc
DURAND Madeleine
DURAND Robert
GALERIE Alain
GAUTHIER Jean-Paul
GENTIL Sylviane
GHIBAUDO Gérard
HAMAR Sylvaine
HAMAR Roger
LADET Pierre
LATOMBE Claudine
LE GORREC Bernard
MADAR Roland
MULLER Jean
NGUYEN TRONG Bernadette
PASTUREL Alain
PLA Fernand
ROUGER Jean
TCHUENTE Maurice
VINCENT Henri

Chercheurs du C.N.R.S
Directeurs de recherche 1ère Classe

CARRE René
FRUCHART Robert
HOPFINGER Emile
JORRAND Philippe
LANDAU Ioan
VACHAUD Georges
VERJUS Jean-Pierre

Directeurs de recherche 2ème Classe

ALEMANY Antoine
ALLIBERT Colette
ALLIBERT Michel
ANSARA Ibrahim
ARMAND Michel
BERNARD Claude
BINDER Gilbert
BONNET Roland
BORNARD Guy
CAILLET Marcel
CALMET Jacques
COURTOIS Bernard
DAVID René
DRIOLE Jean
ESCUDIER Pierre
EUSTATHOPOULOS Nicolas
GUELIN Pierre
JOUD Jean-Charles
KLEITZ Michel
KOFMAN Walter
KAMARINOS Georges
LEJEUNE Gérard
LE PROVOST Christian
MADAR Roland
MERMET Jean
MICHEL Jean-Marie
MUNIER Jacques
PIAU Monique
SENATEUR Jean-Pierre
SIFAKIS Joseph
SIMON Jean-Paul
SUERY Michel
TEODOSIU Christian
VAUCLIN Michel
WACK Bernard

Personnalités agrées à titre permanent à diriger des travaux de recherche (décision du conseil scientifique)

E.N.S.E.E.G

CHATILLON Christian
HAMMOU Abdelkader
MARTIN GARIN Régina
SARRAZIN Pierre
SIMON Jean-Paul

E.N.S.E.R.G

BOREL Joseph

E.N.S.I.E.G

DESCHIZEAUX Pierre
GLANGEAUD François
PERARD Jacques
REINISCH Raymond

E.N.S.H.G

ROWE Alain

E.N.S.I.M.A.G

COURTIN Jacques

E.F.P.

CHARUEL Robert

C.E.N.G

CADET Jean
COEURE Philippe
DELHAYE Jean-Marc
DUPUY Michel
JOUVE Hubert
NICOLAU Yvan
NIFENECKER Hervé
PERROUD Paul
PEUZIN Jean-Claude
TAIB Maurice
VINCENDON Marc

Laboratoires extérieurs

C.N.E.T

DEVINE Rodericq
GERBER Roland
MERCKEL Gérard
PAULEAU Yves

UNIVERSITE Joseph FOURIER (GRENOBLE I)

Président de l'Université :
M. PAYAN Jean Jacques

Année Universitaire 1987 - 1988

MEMBRES DU CORPS ENSEIGNANT DE SCIENCES ET DE GEOGRAPHIE

PROFESSEURS DE 1ère Classe

ARNAUD Paul	Chimie Organique
ARVIEU ROBERT	Physique Nucléaire I.S.N.
AUBERT Guy	Physique C.N.R.S
AURIAULT Jean-Louis	Mécanique
AYANT Yves	Physique Approfondie
BARBIER Marie-Jeanne	Electrochimie
BARJON Robert	Physique Nucléaire ISN
BARNOUD Fernand	Biochimie Macromoléculaire Végétale
BARRA Jean-René	Statistiques-Mathématiques Appliquées
BECKER Pierre	Physique
BEGUIN Claude	Chimie Organique
BELORISKY Elie	Physique
BENZAKEN Claude	Mathématiques Pures
BERARD Pierre	Mathématiques Pures
BERNARD Alain	Mathématiques Pures
BERTRANDIAS Françoise	Mathématiques Pures
BERTRANDIAS Jean-Paul	Mathématiques Pures
BILLET Jean	Géographie
BOELHER Jean-Paul	Mécanique
BONNIER Jane Marie	Chimie Générale
BOUCHEZ Robert	Physique Nucléaire ISN
BRAVARD Yves	Géographie
CARLIER Georges	Biologie Végétale
CAUQUIS Georges	Chimie Organique
CHARDON Michel	Géographie
CHIBON Pierre	Biologie Animale
COHEN ADDAD Jean-Pierre	Physique
COLIN DE VERDIERE Yves	Mathématiques Pures
CYROT Michel	Physique du Solide
DEBELMAS Jacques	Géologie Générale
DEGRANGE Charles	Zoologie
DEMAILLY Jean-Pierre	Mathématiques Pures
DENEUVILLE Alain	Physique
DEPORTES Charles	Chimie Minérale
DOLIQUE Jean-Michel	Physique des Plasmas
DOUCE Roland	Physiologie Végétale
DUCROS Pierre	Cristallographie
FONTAINE Jean-Marc	Mathématiques Pures
GAGNAIRE Didier	Chimie Physique
GERMAIN Jean-Pierre	Mécanique,
GIRAUD Pierre	Géologie
HICTER Pierre	Chimie
IDELMAN Simon	Physiologie Animale
JANIN Bernard	Géographie
JOLY Jean-René	Mathématiques Pures
KAHANE André, détaché	Physique
KAHANE Josette	Physique
KRAKOWIAK Sacha	Mathématiques Appliquées

LAJZEROWICZ Jeanine	Physique
LAJZEROWICZ Joseph	Physique
LAURENT Pierre-Jean	Mathématiques Appliquées
LEBRETON Alain	Mathématiques Appliquées
DE LEIRIS Joël	Biologie
LHOMME Jean	Chimie
LLIBOUTRY Louis	Géophysique
LOISEAUX Jean-Marie	Sciences Nucléaires I.S.N.
LUNA Domingo	Mathématiques Pures
MACHE Régis	Physiologie Végétale
MASCLE Georges	Géologie
MAYNARD Roger	Physique du Solide
OMONT Alain	Astrophysique
OZENDA Paul	Botanique (Biologie Végétale)
PAYAN Jean-Jacques	Mathématiques Pures
PEBAY-PEYROULA Jean-Claude	Physique
PERRIER Guy	Géophysique
PIERRARD Jean-Marie	Mécanique
PIERRE Jean-Louis	Chimie Organique
RENARD Michel	Thermodynamique
RINAUDO Marguerite	Chimie CERMAV
ROSSI André	Biologie
SAXOD Raymond	Biologie Animale
SENGEL Philippe	Biologie Animale
SERGERAERT Francis	Mathématiques Pures
SOUCHIER Bernard	Biologie
SOUTIF Michel	Physique
STUTZ Pierre	Mécanique
TRILLING Laurent	Mathématiques Appliquées
VALENTIN Jacques	Physique Nucléaire I.S.N.
VAN CUTSEM Bernard	Mathématiques Appliquées
VIALON Pierre	Géologie

PROFESSEURS de 2ème Classe

ADIBA Michel	Mathématiques Pures
ANTOINE Pierre	Géologie
ARMAND Gilbert	Géographie
BARET Paul	Chimie
BLANCHI J.Pierre	STAPS
BLUM Jacques	Mathématiques Appliquées
BOITET Christian	Mathématiques Appliquées
BORNAREL Jean	Physique
BRUANDET J.François	Physique
BRUGAL Gérard	Biologie
BRUN Gilbert	Biologie
CASTAING Bernard	Physique
CERFF Rudiger	Biologie
CHIARAMELLA Yves	Mathématiques Appliquées
COURT Jean	Chimie
DUFRESNOY Alain	Mathématiques Pures
GASPARD François	Physique
GAUTRON René	Chimie
GENIES Eugène	Chimie
GIDON Maurice	Géologie
GIGNOUX Claude	Sciences Nucléaires
GILLARD Roland	Mathématiques Pures
GIORNI Alain	Sciences Nucléaires
GONZALEZ SPRINBERG Gérardo	Mathématiques Pures
GUIGO Maryse	Géographie
GUMUCHAIN Hervé	Géographie
GUITTON Jacques	Chimie

HACQUES Gérard	Mathématiques Appliquées
HERBIN Jacky	Géographie
HERAULT Jeanny	Physique
JARDON Pierre	Chimie
JOSELEAU Jean-Paul	Biochimie
KERCKHOVE Claude	Géologie
LONGEQUEUE Nicole	Sciences Nucléaires I.S.N.
LUCAS Robert	Physique
MANDARON Paul	Biologie
MARTINEZ Francis	Mathématiques Appliquées
NEMOZ Alain	Thermodymamique CNRS - CRTBT
OUDET Bruno	Mathématiques Appliquées
PECHER Arnaud	Géologie
PELMONT Jean	Biochimie
PERRIN Claude	Sciences Nucléaires I.S.N.
PFISTER Jean-Claude	Physique du Solide
PIBOULE Michel	Géologie
RAYNAUD Hervé	Mathématiques Appliquées
RICHARD Jean-Marc	Physique
RIEDTMANN Christine	Mathématiques Pures
ROBERT Gilles	Mathématiques Pures
ROBERT Jean-Bernard	Chimie Physique
SARROT-REYNAULD Jean	Géologie
SAYETAT Françoise	Physique
SERVE Denis	Chimie
STOECKEL Frédéric	Physique
SCHOLL Pierre-Claude	Mathématiques Appliquées
SUBRA Robert	Chimie
VALLADE Marcel	Physique
VIDAL Michel	Chimie Organique
VIVIAN Robert	Géographie
VOTTERO Philippe	Chimie

MEMBRES DU CORPS ENSEIGNANT DE L' IUT 1

PROFESSEURS de 1ère Classe

BUISSON Roger	Physique IUT 1
DODU Jacques	Mécanique Appliquée IUT 1
NEGRE Robert	Génie Civil IUT 1
NOUGARET Marcel	Automatique IUT 1
PERARD Jacques	EEA. IUT 1

PROFESSEURS de 2ème classe

BOUTHINON Michel	EEA. IUT 1
CHAMBON René	Génie Mécanique IUT 1
CHEHIKIAN Alain	EEA. IUT 1
CHENAVAS Jean	Physique IUT 1
CHOUTEAU Gérard	Physique IUT 1
CONTE René	Physique IUT 1
GOSSE Jean-Pierre	EEA.IUT 1
GROS Yves	Physique IUT 1
KUHN Gérard, (Détaché)	Physique IUT 1
MAZUER Jean	Physique IUT 1
MICHOULIER Jean	Physique IUT 1
MONLLOR Christian	EEA.IUT 1
PEFFEN René	Métallurgie IUT 1
PERRAUD Robert	Chimie IUT 1
PIERRE Gérard	Chimie IUT 1
TERRIEZ Jean-Michel	Génie Mécanique IUT 1
TOUZAIN Philippe	Chimie IUT 1
VINCENDON Marc	Chimie IUT 1

PROFESSEURS DE PHARMACIE

AGNIUS-DELORD Claudine	Physique	Faculté La Tronche
ALARY Josette	Chimie Analytique	Faculté La Tronche
BERIEL Hélène	Physiologie et Pharmacologie	Faculté La Tronche
CUSSAC Max	Chimie Therapeutique	Faculté La Tronche
DEMENGE Pierre	Pharmacodynamie	Faculté La Tronche
FAVIER Alain	Biochimie	C.H.R.G.
JEANNIN Charles	Pharmacie Galénique	Faculté Meylan
LATURAZE Jean	Biochimie	Faculté La Tronche
LUU DUC Cuong	Chimie Générale	Faculté La Tronche
MARIOTTE Anne-Marie	Pharmacognosie	Faculté La Tronche
MARZIN Daniel	Toxicologie	Faculté Meylan
RENAUDET Jacqueline	Bactériologie	Faculté La Tronche
ROCHAT Jacques	Hygiéne et Hydrologie	Faculté La Tronche
SEIGLE-MURANDI Françoise	Botanique et Cryptogamie	Faculté Meylan
VERAIN Alice	Pharmacie Galénique	Faculté Meylan

MEMBRES DU CORPS ENSEIGNANT DE MEDECINE

PROFESSEURS CLASSE EXEPTIONNELLE ET 1ère CLASSE

AMBLARD Pierre	Dermatologie	C.H.R.G.
AMBROISE-THOMAS Pierre	Parasitologie	C.H.R.G.
BEAUDOING André	Pédiatrie-Puericulture	C.H.R.G.
BEZEZ Henri	Orthopédie-Traumatologie	Hopital SUD
BONNET Jean-Louis	Ophtalmologie	C.H.R.G.
BOUCHET Yves	Anatomie	Faculté La Merci
	Chirurgie Générale et Digestive	C.H.R.G.
BUTEL Jean	Orthopédie-Traumatologie	C.H.R.G.
CHAMBAZ Edmond	Biochimie	C.H.R.G.
CHAMPETIER Jean	Anatomie-Topographique et Appliquée	C.H.R.G.
CHARACHON Robert	O.R.L.	C.H.R.G.
COLOMB Maurice	Immunologie	Hopital sud
COUDERC Pierre	Anatomie-Pathologique	C.H.R.G.
DELORMAS Pierre	Pneumophtisiologie	C.H.R.G.
DENIS Bernard	Cardiologie	C.H.R.G.
GAVEND Michel	Pharmacologie	Faculté La Merci
HOLLARD Daniel	Hématologie	C.H.R.G.
LATREILLE René	Chirurgie Thoracique et Cardiovasculaire	C.H.R.G.
LE NOC Pierre	Bactériologie-Virologie	C.H.R.G.
MALINAS Yves	Gynécologie et Obstétrique	C.H.R.G.
MALLION Jean-Michel	Médecine du Travail	C.H.R.G.
MICOUD Max	Clinique Médicale et Maladies Infectieuses	C.H.R.G.
MOURIQUAND Claude	Histologie	Faculté La Merci
PARAMELLE Bernard	Pneumologie	C.H.R.G.
PERRET Jean	Neurologie	C.H.R.G.
RACHAIL Michel	Hépato-Gastro-Entérologie	C.H.R.G.
DE ROUGEMONT Jacques	Neurochirurgie	C.H.R.G.
SARRAZIN Roger	Clinique Chirurgicale	C.H.R.G.
STIEGLITZ Paul	Anestésiologie	C.H.R.G.
TANCHE Maurice	Physiologie	Faculté La Merci
VIGNAIS Pierre	Biochimie	Faculté La Merci

PROFESSEURS 2ème CLASSE

BACHELOT Yvan	Endocrinologie	C.H.R.G.
BARGE Michel	Neurochirurgie	C.H.R.G.
BENABID Alim Louis	Biophysique	Faculté La Merci
BENSA Jean-Claude	Immunologie	Hopital Sud
BERNARD Pierre	Gynécologie-Obstétrique	C.H.R.G.
BESSARD Germain	Pharmacologie	ABIDJAN
BOLLA Michel	Radiothérapie	C.H.R.G.
BOST Michel	Pédiatrie	C.H.R.G.
BOUCHARLAT Jacques	Psychiatrie Adultes	Hopital Sud
BRAMBILLA Christian	Pneumologie	C.H.R.G.
CHIROSSEL Jean-Paul	Anatomie-Neurochirurgie	C.H.R.G.
COMET Michel	Biophysique	Faculté La Merci
CONTAMIN Charles	Chirurgie Thoracique et Cardiovasculaire	C.H.R.G.
CORDONNIER Daniel	Néphrologie	C.H.R.G.
COULOMB Max	Radiologie	C.H.R.G.
CROUZET Guy	Radiologie	C.H.R.G.
DEBRU Jean-Luc	Médecine Interne et Toxicologie	C.H.R.G.
DEMONGEOT Jacques	Biostatistiques et Informatique Médicale	Faculté La Merci
DUPRE Alain	Chirurgie Générale	C.H.R.G.
DYON Jean-François	Chirurgie Infantile	C.H.R.G.
ETERRADOSSI Jacqueline	Physiologie	Faculté La Merci
FAURE Claude	Anatomie et Organogénèse	C.H.R.G.
FAURE Gilbert	Urologie	C.H.R.G.
FOURNET Jacques	Hépato-Gastro-Entérologie	C.H.R.G.
FRANCO Alain	Médecine Interne	C.H.R.G.
GIRARDET Pierre	Anesthésiologie	C.H.R.G.
GUIDICELLI Henri	Chirurgie Générale et Vasculaire	C.H.R.G.
GUIGNIER Michel	Thérapeutique et Réanimation Médicale	C.H.R.G.
HADJIAN Arthur	Biochimie	Faculté La Merci
HALIMI Serge	Endocrinologie et Maladies Métaboliques	C.H.R.G.
HOSTEIN Jean	Hépato-Gastro-Entérologie	C.H.R.G.
HUGONOT Robert	Médecine Interne	C.H.R.G.
JALBERT Pierre	Histologie-Cytogénétique	C.H.R.G.
JUNIEN-LAVILLAUROY Claude	O.R.L.	C.H.R.G.
KOLODIE Lucien	Hématologie Biologique	C.H.R.G.
LETOUBLON Christian	Chirurgie Générale	C.H.R.G.
MACHECOURT Jacques	Cardiologie et Maladies Vasculaires	C.H.R.G.
MAGNIN Robert	Hygiène	C.H.R.G.
MASSOT Christian	Médecine Interne	C.H.R.G.
MOUILLON Michel	Ophtalmologie	C.H.R.G.
PELLAT Jacques	Neurologie	C.H.R.G.
PHELIP Xavier	Rhumatologie	C.H.R.G.
RACINET Claude	Gynécologie-Obstétrique	Hopital Sud
RAMBAUD Pierre	Pédiatrie	C.H.R.G.
RAPHAEL Bernard	Stomatologie	C.H.R.G.
SCHAERER René	Cancérologie	C.H.R.G.
SEIGNEURIN Jean-Marie	Bactériologie-Virologie	Faculté La Merci
SELE Bernard	Cytogénétique	Faculté La Merci
SOTTO Jean-Jacques	Hématologie	C.H.R.G.
STOEBNER Pierre	Anatomie Pathologique	C.H.R.G.
VROUSOS Constantin	Radiothérapie	C.H.R.G.

Je tiens à remercier les membres du jury :

Monsieur Jacques Mossière, Professeur à l'INPG, qui me fait l'honneur de présider le jury de cette thèse. Qu'il me soit permis de lui exprimer ma profonde reconnaissance pour les appuis et les encouragements qu'il a su m'apporter.

Monsieur Yves Chiaramella, Professeur à l' Université Joseph Fourier de Grenoble, qui a bien voulu donner son avis sur le contenu de cette thèse.

Monsieur Jacky Estublier, Chargé de Recherche au CNRS, responsable du groupe Adèle, dont les idées et encouragements ont largement contribué à la réussite de ce projet. Je profite de cette occasion très formelle pour lui exprimer toute ma gratitude pour la confiance qu'il m'a toujours témoignée et pour son soutien amical et ses nombreux conseils qui m'ont permis de mener à bien ce travail.

Monsieur Sacha Krakowiak, Professeur à l'Université Joseph Fourier et Directeur de cette thèse pour l'intérêt qu'il a manifesté pour ce travail. Qu'il trouve ici l'expression de ma reconnaissance pour m'avoir suggéré cette étude et avoir accepté de lire et corriger les versions de ce document.

Monsieur Axel Van Lamsweerde, Professeur à l'Institut d'Informatique de l'Université de Namur (Belgique) qui a accepté de juger cette thèse. Je le remercie particulièrement d'avoir effectué une lecture soigneuse de ce document et de m'avoir permis par ses remarques de l'améliorer.

Je voudrais aussi remercier

Monsieur Jean Louis Cheval avec lequel j'ai échangé des idées. Je lui suis reconnaissant de m'avoir fait l'amitié de lire et critiquer mon document, pour les nombreuses discussions qui en ont découlé et pour ses conseils judicieux.

Mes camarades de l'équipe Adèle, Jean Michel Adam, Xavier Girod, Philippe Morat et Jean Marie Favre pour leur soutien amical et pour leur contribution à la bonne ambiance dans l'équipe. J'associe à ces remerciements les deux stagiaires Pierre Flener et Bernard Jabas qui se sont joints à notre équipe pour prendre en charge la réalisation de la partie de Nomade consacrée aux droits d'accés.

Toutes les personnes du Laboratoire de Génie Informatique de l'IMAG pour leur amabilité à mon égard et le soutien qu' elles ont pu m'apporter.

A ma mère pour ses nombreux sacrifices

A la mémoire de mon père

RESUME

Dans le développement et la maintenance de logiciels de grande taille, la programmation globale est l'activité la moins formalisée, la moins assistée alors que c'est l'activité prépondérante, celle qui consomme le plus de temps. Nous décrivons les principaux concepts et mécanismes de Nomade, un noyau d'environnement pour supporter le développement et la maintenance de gros logiciels. Nomade constitue une extension et une généralisation de son prédécesseur Adèle. Il fournit un modèle pour la construction, l'intégration et le contrôle des logiciels en versions multiples et de leurs équipes. Nous présentons essentiellement la base d'objets et les mécanismes de contrôle de la structure du logiciel avec la notion de partition et d'activation d'outils basée sur le concept d'événement action. La base d'objets est conçue autour d'un modèle adapté aux besoins de la programmation globale qui inclut des notions "orientées objet". La base conserve la structure du logiciel : les objets logiciels, leurs versions, leurs documents, leurs attributs, leurs relations et leurs objets dérivés. Le mécanisme d'événements actions s'inspire des mécanismes d'activations (déclencheurs, démons, exceptions) développés dans d'autres domaines. Nous montrons comment Nomade permet de maintenir les contraintes de cohérence complexes rencontrées dans la programmation globale, de définir et d'intégrer les stratégies et les outils spécifiques à un environnement de développement et de maintenance.

MOTS-CLES :
génie logiciel, programmation globale, maintenance de logiciel, environnement de génie logiciel, événement, action, déclencheur, bases de programmes, intégration d'outils.

NOMADE : AN ENVIRONMENT KERNEL FOR PROGRAMMING IN THE LARGE

ABSTRACT

For the development and maintenance for large software products, programming in the large is the less assisted,and formalized activity while it is the most time consuming. We describe the main concepts and mechanisms of Nomade, a kernel designed for building environments dedicated to the maintenance of large software products. Nomade is an extension and a generalization of its predecessor Adele . It provides a model to build, integrate and control the multiple versions of software products and their staffs. We present mainly the data base and the mechanisms for the structure control, based on the partition concept, and an events-actions mechanism, based on an activation mechanism. The data base is designed to provide services needed for programming in the large and includes object oriented features. The data base manages the software structure, the software objects, their versions, their documents, their attributes, their relations and their derived objects. The events-actions mechanism borrows ideas from triggers, deamons, exceptions, developped in other fields. We show how Nomade allows to enforce complex consistency constraints as found in programming in the large, to define and integrate the strategies and tools local to a specific software engineering environment.

KEYWORDS:
software engineering, programming in the large, software maintenance, software engineering environment, event, action, trigger, software data base, tool integration.

CHAPITRE 1 :

INTRODUCTION

1. CADRE DE L'ETUDE ET MOTIVATIONS

Les différences de tâches entre la description de logiciels de taille importante et la description de programmes de taille modeste, conçus de manière monolithique, ont vu la séparation de la programmation en deux activités différentes comme le montre [De-Remer76]. La **programmation détaillée** ou *"programming in the small"* et la **programmation globale** ou *"programming in the large"* .

La programmation détaillée concerne le développement, souvent par une seule personne, de programmes de taille modeste (quelques milliers de lignes environ) dont la durée de développement n'excède pas quelques mois. La programmation détaillée utilise des outils comme les éditeurs, les compilateurs et les chargeurs.

La programmation globale (on trouve aussi l'expression*"programming in the many"* [Habermann86]) est l'activité de structuration d'un logiciel de taille importante à partir d'une collection de composants, développés par un nombre important de programmeurs. En plus de la composition de logiciel à partir des composants, la programmation globale pose les problèmes de cohérence induits par les changements et évolutions des besoins d'un projet, par la gestion de versions multiples de composants, de communication et de partage de ressources entre les équipes de développement et, pour lesquels les outils développés pour la programmation détaillée ne peuvent pas s'appliquer [Ramamoorthy86] [Shaw86].

Au cours de ces dernières années, de nombreux travaux de développement d'environnements de support à la programmation globale ont été entrepris. Les raisons qui ont suscité ce développement sont de deux types:

- L'informatique a étendu son champ d'application et a voulu apporter des solutions à des problèmes de plus en plus complexes comme par exemple l'informatisation des centraux téléphoniques, le contrôle de processus industriels... Les logiciels conçus à cet effet sont pour la plupart volumineux (des millions de lignes de code), ont des spécifications

complexes et nécessitent le plus souvent l'intervention de plusieurs personnes pour assurer leur développement et leur maintenance.

- Des langages modulaires ont été conçus pour permettre le développement et la maintenance de logiciels de taille importante;

Les résultats de ces recherches ont permis le développement d'outils puissants qui sous certaines hypothèses construisent automatiquement des logiciels cohérents. Par contre, il y a beaucoup d'aspects de la programmation globale qui n'ont pas été traités convenablement. Une analyse plus détaillée de ces approches nous a conduit à relever certaines lacunes:

- Les aspects liés au contrôle des effets de bord des différentes actions, à l'intégration d'outils, au contrôle de l'évolution de la structure du logiciel, ne sont que partiellement abordés.

- les aspects liés à la construction de logiciels avec l'utilisation de différents outils et différentes notions de niveaux de cohérence plutôt que ceux prédéfinis ne sont pas pris en compte;

- les aspects liés à l'utilisation de différentes stratégies et règles de gestion en fonction des besoins de chaque usager ne sont pas traités;

- beaucoup de solutions proposées ne s'appliquent pas efficacemment et sont en pratique inutilisables.

D'une manière générale, les solutions proposées sont limitées à un langage ou une méthode de programmation et ne permettent pas de s'adapter efficacement à une forme d'hétérogénéité rencontrée dans les situations de développement et de maintenance(plusieurs langages et méthodes).

De ces constatations, nous avons circonscrit le type de problèmes que nous voulions étudier et le degré de généralité et d'efficacité que doivent atteindre les solutions proposées. Cette thèse présente une approche de solutions au problème de la programmation globale. Les fonctions de base sont identifiées et intégrées dans un noyau de support à la programmation globale suffisamment ouvert et extensible.

Le niveau d'assistance offert par un environnement de développement et de maintenance de logiciel dépend largement des mécanismes de base développés pour gérer la quantité importante

d'informations produites durant le cycle de développement et de maintenance du logiciel. Comme les SGBD classiques, développés à l'origine pour les applications commerciales, se sont avérés inadaptés, nous avons cherché à dégager les concepts mis en œuvre dans une base de données adaptée à la gestion d'informations de programmation globale. Ces concepts devant être suffisamment généraux pour couvrir une large variété d'environnements. Nous avons proposé un modèle de données spécifique à la programmation globale et offrant une meilleure structuration et un meilleur contrôle du logiciel. Une nouvelle approche de modèlisation intégrant des concepts récents des bases de données a été définie.

Nous avons complété l'étude par certains aspects de développement et de maintenance de logiciel qui jusqu'à présent ont reçu une moindre attention. Il s'agit en particulier du traitement des effets de changements et évolutions pour lesquels nous proposons une solution permettant à chaque usager ou groupe d'usagers de définir des stratégies différentes en réponse aux changements.

2. TRAVAIL REALISE

Le travail réalisé dans le cadre de cette thèse s'appuie sur la base de programmes réalisée dans le cadre du projet Adèle de développement d'un environnement de programmation pour le langage Pascal et se décompose en deux parties.

La première partie a consisté à poursuivre le travail sur les bases de programmes indépendamment de l'environnement de programmation Adèle et à concevoir et réaliser une base pour des logiciels industriels de grande taille. Ce travail a abouti à la réalisation d'un prototype industriel qui a gardé le nom d'Adèle bien que réalisé dans un cadre différent. Cette version d'Adèle, diffusée industriellement, est centrée sur les fonctions de gestion de versions de logiciels et de construction de configurations.

L'expérimentation d'Adèle en mileu industriel a montré les avantages et limites d'un pareil outil. Les limitations concernent principalement son manque de flexibilité, ce qui rend son adaptation difficile. La connaissance de la méthode de développement ainsi que des objets logiciels et des règles de gestion est figée dans le code. Par ailleurs Adèle est principalement une base de stockage passive dans la mesure où elle ne permet pas l'activation d'outils

Cette expérience nous a convaincus de généraliser les concepts présents dans Adèle pour créer un noyau qui offre une grande adaptabilité et participe activement au développement et à la maintenance de logiciels.

La deuxième partie de ce travail a consisté à faire évoluer la base de programme Adèle vers un noyau de support à la programmation globale: Nomade. Quelques caractéristiques telles que

la flexibilité, l'activation d'outils, la distribution, le contrôle des droits d'accès et de la structure des logiciels sont privilégiées dans la conception et la réalisation du noyau.

Le travail s'est déroulé comme suit:

- Une première phase de consolidation de la base de programmes réalisée pour l'environnement de programmation Pascal a été entreprise afin d'obtenir une base de gestion de logiciels industriels de grande taille. Une version d'Adèle introduisant des extensions et améliorations a été développée sous le système Unix. Le travail mené a été axé principalement sur les aspects techniques améliorant la fiabilité et l'efficacité pour obtenir un produit pré-industriel, portable et diffusable, et une base pour la poursuite de nos travaux. Le prototype a été reprogrammé en C et Pascal afin d'être plus efficace et aisément portable sur les machines supportant Unix.

- La deuxième phase de ce travail a abouti à la conception d'un noyau d'environnement pour la maintenance et le développement de logiciels de grande taille (Nomade).
Plus précisemment, nous avons étendu le modèle de données d'Adèle par de nouveaux mécanismes de structuration, des types d'objets et de relations et nous avons inclus un gestionnaire d'activités pour contrôler la propagation des changements et assurer la cohérence des manipulations. Ces mécanismes s'appuient sur l'approche "orientée objet".
Le noyau a été concu en tant que serveur gérant une base centralisée accessible par un réseau local à partir de machines clients Unix.
La réalisation s'est faite autour du noyau Adèle en expérimentant et en intégrant progressivement les nouveaux concepts. Les concepts développés dans la thèse ont été testés.

3. PLAN DE LA THESE

Le plan de la thèse se décompose comme suit:

- Le chapitre 2 fait le point des travaux menés dans le cadre de la programmation globale. Les caractéristiques, les techniques et les outils sont analysés et les fonctions de base d'un noyau de support à la programmation globale sont identifiées et présentées. Puis nous faisons une étude critique des différentes approches d'implantation du noyau et dégageons les grandes tendances d'évolution. Enfin, l'évolution du domaine développée dans la première partie du chapitre est retracée par la présentation d'outils et environnements représentatifs des différents travaux réalisés, tout en dégageant les idées importantes qu'ils contiennent.

- Le chapitre 3 décrit une première approche qui a permis de mieux comprendre les problèmes de la programmation globale. Elle porte sur l'adaptation de la base de programmes Adèle à la gestion de logiciels industriels de taille importante. Les concepts d'Adèle et l'expérience d'adaptation du prototype est décrite.

Les chapitres suivants présentent les extensions apportées à Adèle pour réaliser le système **Nomade**:

- Le chapitre 4 aborde le problème de la définition d'une base d'objets permettant la programmation globale de systèmes modulaires. Ce chapitre présente le modèle qui a été retenu pour la construction, l'intégration, le contrôle et l'évolution des logiciels et des équipes de développement. Une première partie est consacrée à la présentation des concepts de base et du langage d'expression. La deuxième partie présente les mécanismes de structuration introduits.

- Le chapitre 5 présente une méthode de contrôle des effets des changements basée sur l'activation. L'activation signifie que la base d'objets a été augmentée de mécanismes qui déclenchent des actions lorsqu'une contrainte est violée. Les mécanismes sont exposés et le langage de description des actions est défini. On montre à l'aide d'exemples comment ce simple mécanisme peut être utilisé pour contrôler et gérer les phénomènes de propagation et pour programmer aisément les politiques et les contraintes de gestion de logiciel .

- Le chapitre 6 présente les aspects de réalisation du système Nomade. Les principaux choix techniques sont exposés et les différentes parties du système sont présentées. En conclusion de ce travail, nous suggérons une nouvelle architecture pour un noyau de programmation globale.

CHAPITRE 2 :

SYSTEME DE PROGRAMMATION GLOBALE : CONCEPTS, OUTILS ET TENDANCES D'EVOLUTION

1. PRESENTATION

Les travaux de recherche ont identifié le besoin de formalismes et d'outils pour mieux maitriser les problèmes de la programmation. Les objectifs essentiels sont l'amélioration de la qualité du logiciel et l'accroissement de la productivité des personnels.

Des **environnements de programmation** ont été développés pour atténuer les problèmes de la programmation détaillée. La structure type de ces environnements est constituée d'un ensemble d'outils intégrés incluant un compilateur ou interpréteur et un metteur au point accessibles via un éditeur syntaxique et manipulant une représentation unique des programmes dérivée de la syntaxe abstraite (Mentor [Donzeau80], Cornell Synthesizer [Teitelbaum81], Loipe [Medina81], Adèle [Estublier83]).

Les travaux de [Buxton80], [Notkin85] ont montré la nécessité d'**environnements pour la programmation globale** comme pour la programmation détaillée et ont utilisé le terme d'**environnement de développement et de maintenance**. Les environnements de développement et de maintenance visent l'automatisation des tâches de la programmation globale comme par exemple le contrôle de versions et la gestion de configurations. La prise en compte de ces tâches soulève des problèmes concernant l'architecture et la structure de données d'un tel environnement.

Ce support a d'abord consisté en un ensemble d'outils dépareillés. Ces outils ont résolu quelques problèmes mais de manière totalement cloisonnée. Cette démarche peut engendrer des incohérences entre les données produites par les différents outils car chacun gère séparément la cohérence de ses données. On s'est aperçu plus tard qu'il est important de créer les outils mais

aussi d'offrir une intégration efficace de ces outils. L'idée d'**intégration** indique ici un fondement commun aux outils.

Des efforts de développement tendent à fournir une base de support aux outils variés de développement et de maintenance. L'intégration par l'utilisation d'une base d'objets commune qui maintient les propriétés des composants du logiciel et de leurs relations a été introduite [Huff81],[Tichy82].

Nous nous intéressons plus particulièrement à ce type d'intégration, qui suppose que les outils coopérent et partagent des données. Ces données sont structurées indépendamment des outils particuliers qui les créent et les utilisent.

L'architecture des environnements intégrés fait ressortir par conséquent deux composants essentiels: la base d'objets et un ensemble d'outils. Selon la manière dont ces outils accèdent aux données, il sont répartis en outils natifs et en outils externes [Schwankle88a]. Les outils externes ne connaissent pas l'environnement; ils échangent les informations avec l'environnement via un mécanisme d'intégration d'outils. Les outils natifs utilisent directement les objets dans la base et les travaux actuels tendent à les intégrer dans le noyau de l'environnement.

Les environnements intégrés de génie logiciel font donc la distinction entre les outils externes de l'environnement et le noyau qui constitue la structure d'accueil: PCTE [Gallo86], APSE [Buxton80]. Le noyau constitue l'infrastructure des services communs. Il assiste l'utilisateur de l'environnement en exploitant la connaissance qu'il a de la structure des relations entre composants et, dans le cas d'environnement mono-langage, de la syntaxe et de la sémantique du langage de programmation utilisé. Notre étude est consacrée à la structure d'accueil ou noyau d'environnement.

Ce chapitre est organisé comme suit. Nous commençons par dégager les caractéristiques des informations d'un système de programmation globale et les concepts développés pour les représenter. Puis nous décrivons les fonctions d'un noyau de programmation globale. Nous présentons ensuite les différentes approches pour implanter le noyau. Enfin nous analysons les différents travaux réalisés dans le domaine et ceux représentatifs des tendances actuelles. Pour chacun d'eux, nous présentons leurs champs d'investigation et leurs contributions essentielles.

2. CARACTERISTIQUES DES INFORMATIONS DE PROGRAMMATION GLOBALE

2.1. Introduction

Les logiciels de taille importante changent constamment durant leur cycle de vie et engendrent des composants de différentes natures, produits durant le cycle de vie du logiciel [Boehm76] [Krakowiak82]: modules, documents, jeux de test, données, codes objet. Ces composants sont couramment désignés sous le terme générique d'**objet logiciel**. Ces objets sont caractérisés par des attributs, se développent en versions multiples et ont des relations de différents types (par exemple dépendance entre modules, relation "documente" entre un document et un module...).

La nature évolutive des logiciels de taille importante et le besoin d'un support de stockage est partagée avec d'autres disciplines comme la CAO [Katz85] [Fauvet87], où l'on retrouve quelques concepts de base comme les versions et les configurations. Par contre les logiciels ont des caractéristiques qui posent des problèmes particuliers. Les objets logiciels sont des objets informatiques et peuvent donc être facilement stockés et manipulés dans l'ordinateur. Les changements sont plus aisés qu'avec les autres types de systèmes et tendent à être plus fréquents.

Des base d'objets de programmation globale ont été utilisées pour représenter:

- les différents types d'objets, leur attributs et leurs versions;
- les diverses relations;
- les contraintes de cohérence.

Nous précisons les spécificités de chacun de ces points dans le contexte de la programmation globale.

2.2. Les objets logiciels.

C'est le contenu des objets et la description par des attributs qui sont désignés dans les environnements par le concept unifié d'**objet logiciel**.

- Les attributs de l'objet identifient le type de l'objet et des informations propres à l'environnement de production comme le nom, la date de dernière modification, le langage.

- Le contenu est le texte d'un programme, d'un document. [DOD85] recommande que des attributs contenant des données non interprétées soient supportés par la base.

Les environnements de développement tendent à conserver l'immutabilité des objets. Les objets, une fois créés, ne peuvent être que détruits.

Une classification des objets logiciels est généralement admise [Tichy88] et les distingue selon leur mode de création, en **objets atomiques** et **objets composés** et, selon leur mode de production, en **objets sources** et **objets dérivés**.

- **Les objets atomiques:** Ce sont des objets simples non décomposables. L'objet est manipulé dans son intégralité. Le contenu de l'objet n'est pas interprété par le gestionnaire de l'objet.

- **Les objets composés.** Ce sont des objets complexes souvent structurés et constitués d'un assemblage d'objets atomiques et composés. Les versions comme on le verra par la suite sont des exemples d'objets composés.

- **Les objets sources.** Les objets sources sont produits manuellement et sont par conséquent la matière première de l'environnement comme par exemple le texte d'un programme ou d'un document. Ils doivent être stockés et maintenus précieusement car il n'est pas possible de les regénérer automatiquement.

- **Les objets dérivés.** Ce sont des objets générés automatiquement par un outil de dérivation d'objets. Des exemples d'objets dérivés incluent les binaires de programmes, les exécutables, les documents formattés. Des exemples d'outils de dérivation d'objet incluent les compilateurs, les éditeurs de liens, les outils de formattage de documents... Les objets dérivés peuvent être regénérés à partir des objets sources et n'ont pas besoin d'être stockés. Les environnements utilisent généralement un cache d'objets dérivés pour accélérer le processus de production.

Partant de ces concepts, se pose alors le problème du plus petit composant à considérer comme objet atomique.

2.2.1. Granularité des objets

Dès qu'un logiciel devient complexe, il est convenable de le décomposer en composants plus petits. La difficulté commune des environnements est de définir un niveau de granularité convenable pour les objets manipulés. On trouve une étude des problèmes de granularité dans [Feiler86].

On est tenté d'introduire un niveau fin de décomposition pour augmenter le nombre des informations. Mais plus on décompose, plus le nombre d'objets croît, ce qui pose à terme un

problème d'efficacité. Par exemple un niveau fin de granularité considérant une variable d'un programme comme objet engendre un nombre considérable d'objets et de relations.

Exemple:

Dans le système OMEGA [Linton84] utilisant le SGBD Ingres[Stonebraker76], la granularité des objets est définie au niveau le plus bas (constantes, instructions,variables...).
Les mesures de performance effectuées sur un Vax/750 sous le système Unix donnent les résultats suivants pour un programme de 1000 lignes Pascal :

- l'occupation de l'espace base de donnée est de 418K octets;
- le temps de réponse pour afficher les 1000 lignes à l'écran est de 3 heures en utilisant les techniques standards d'interrogation et de 8 mn après de multiples optimisations incluant des modifications du système Ingres;

Les critères à la base du choix d'une granularité appropriée sont d'ordre technique. On essaie de trouver un compromis entre les fonctions et les performances. Les développements actuels [Clemm86] recommandent la définition d'une entité suffisamment large, stockée comme un objet et à partir de laquelle sont dérivées les informations pertinentes au fur et à mesure des besoins. La plupart des environnements offrant une solution réaliste, ont pris **le module** comme grain de décomposition pour limiter la prolifération d'objets. Nous précisons dans ce qui suit le concept de module.

2.2.2. Module = Interface + Réalisation

Les langages de programmation traditionnels comme PL/1, Pascal ne supportent pas directement la construction modulaire de programmes. Ces langages ne permettent pas de décomposer un programmes en unités compilées séparément et ne fournissent pas de mécanismes pour la vérification de types entre composants. De nouveaux langages modulaires comme ADA [DOD83], Modula2 [Wirth83] et Mesa [Mitchell84] ont été conçus pour supporter le développement et la maintenance de logiciels de taille importante. Dans cette thèse, nous utilisons le concept de **module** dans le sens des **langages modulaires** comme ADA, Mesa, Modula2. Ces langages considèrent un module comme la composition de deux parties qui sont l'nterface du module et la réalisation ou corps et incluent des mécanismes pour les spécifier. L'**interface** indique toutes les ressources du modules qui sont les procédures, les variables et les types visibles à l'extérieur du module. Le **corps** est la partie qui réalise les ressources du module. Chaque module n'est accessible aux autres que par son interface. Tous les détails de réalisation, souvent complexes, sont dans le corps et demeurent cachés aux autres modules. Lorsqu'un module M1 a besoin d'une ressource réalisée par un autre module M2, on dit que M1 **dépend de** M2.

Les modules peuvent être conçus, codés et testés indépendamment les uns des autres par différentes personnes, ce qui facilite le partage du travail.

La séparation de l'interface d'un module de sa réalisation est un concept fondamental dans la programmation modulaire. Nous en listons les principaux avantages.

- Mécanisme d'abstraction.
L'existence d'une interface, séparée de la réalisation, permet de la définir indépendamment de la réalisation associée. On peut vérifier uniquement par les interfaces, la complétude d'un système, c'est à dire que l'ensemble des ressources utilisées sont toutes bien définies dans l'ensemble des interfaces, et de retarder le choix d'une implantation appropriée.

- Minimisation des étapes de recompilation.
Dans le cas d'une granularité de niveau module, la modification d'un module exige la recompilation de tous les modules clients. La séparation entre interfaces et réalisation permet de ne recompiler les modules clients que dans le cas de la modification de l'interface utilisée. La réalisation peut être changée sans déclencher de recompilations.

- Un module supporte les types abstraits de données.
Un module peut encapsuler un type de donnée et ses opérations et cacher ainsi les détails de représentation.

Pour terminer, il faut indiquer que:

- Le niveau de granularité défini par l'interface et la réalisation d'un module peut être raffiné. Avec le développement de la technologie des bases de données et l'augmentation des performances des machines, une granularité plus fine devrait pouvoir être possible.

- Le concept de module, supporté par les langages modulaires, associant une seule réalisation à une interface, reste insuffisant dans un environnement de développement et de maintenance. Il a été étendu pour prendre en compte l'évolution en versions.

2.3. Les versions

Les deux techniques courantes de gestion de données, c'est à dire les SGF et les SGBD ne sont capables de gérer, pour la plupart, que le dernier état d'un objet: le plus récent et le plus cohérent dans le cas des SGBD. Cette limitation n'est pas acceptable dans un environnement de développement et de maintenance. Les logiciels de taille importante évoluent constamment pour des raisons de développement ou pour être adaptés à un large éventail de fonctions différentes.

Par exemple, les logiciels sont souvent adaptés à différentes configurations de machines, à des besoins différents. Il est intéressant de développer des concepts qui expriment les informations retraçant cette évolution. Les environnements ont montré l'avantage de garder cette trace sous deux formes:

- la gestion de l'évolution des objets par le concept de **versions,**
- la gestion d'un objet **historique** pour garder la trace de l'ensemble des opérations effectuées.

2.3.1. Versions de logiciel

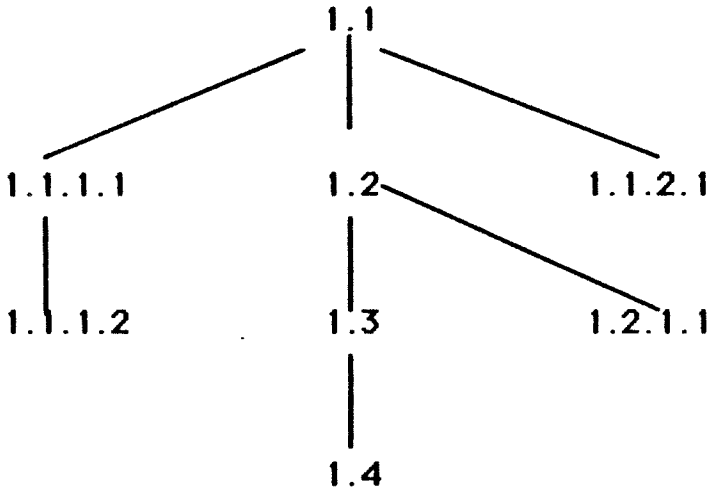
La gestion de versions est généralement appliquée au texte source (unité de compilation) et le concept de version est communément admis [Tichy88] pour regrouper les deux types d'évolutions faites en pratique sans que l'on puisse les distinguer de manière formelle:

- Le concept de **révision** correspond à différentes formes d'évolution d'un objet durant le développement et la maintenance comme par exemple le raffinement progressif et les corrections d'erreurs.

- Le concept d'**alternative** correspond à des développements parallèles indépendants. Les objets gérés en alternatives coexistent dans le temps et correspondent soit à des alternatives fonctionnelles ou d'implantations, soit à des développements parallèles du même objet parce que maintenus en parallèle. Une alternative peut évoluer séquentiellement en révisions.

L'évolution en alternatives et révisions est variable d'un système, contrôlant les versions, à un autre; mais on retrouve globalement les deux schémas d'évolution suivants:

- des outils comme RCS [Tichy85] organisent les révisions en arbre où les branches sont les alternatives. Les alternatives sont représentées par des chemins d'un sous arbre dont la racine est la révision commune comme le montre le schéma suivant :



- des environnements comme Adèle [Belkhatir86b] et Gandalf [Notkin85] gérent les alternatives sous forme de suites parallèles de révisions.

Alt1	**r1-->r2-->r3-->r4**
Alt2	**r1-->r2**
Alt3	**r1-->r2**

Versions composées

Notons que les environnements introduisent un troisième type de versions dénommé **versions composées** [Perry87]. Les versions composées sont définies par une liste de sources définie par l'usager. La liste des sources définit les objets de la composition et spécifie les versions appropriées qui sont utilisées. Les versions composées peuvent à leur tour évoluer en révisions et alternatives. La plupart des environnements supportent les versions composées avec des règles d'assemblage différentes comme on le verra par la suite (§2.6.2). Ce sont les notions de **configuration** dans Adèle, **compositions** dans Gandalf [Habermann81] et **system model** dans Cedar [Lampson83]. Nous utiliserons par la suite indifféremment le terme configuration ou version composée.

Identification des versions

La gestion de versions multiples d'un objet nécessite un schéma d'identification de ces versions. Dans la plupart des environnements, les alternatives et révisions sont identifiées par un triplet (nom d'objet, alternative, révision) où le nom de l'objet correspond au nom usuel, l'alternative sélectionne un membre de l'ensemble des alternatives d'un objet et la révision sélectionne une des révisions de l'alternative. On va examiner les identifications les plus couramment utilisées.

- Adèle et Gandalf désignent l'alternative par un identificateur et les révisions par un nombre attribué séquentiellement par le système.

- Dans Cedar, les versions sont identifiées par un nom et une estampille correspondant à la date de création de l'objet. Il n'y a pas d'autres caractéristiques pour décrire les alternatives et révisions.

- RCS désigne alternative et révision par une numérotation hiérarchique complexe (voir §2.3.1).

- Adèle (Nomade) a introduit en plus de la désignation hiérarchique, une description et une identification des versions basées sur l'utilisation de propriétés définies par l'usager. La description des versions par des propriétés est inspirée de la technologie des bases de données. Elle permet une plus grande richesse sémantique et donne une grande puissance au mécanisme de désignation. Dans Adèle, la désignation par propriétés est utilisée dans le cadre restreint de la gestion de configurations. Nomade l'utilise comme désignation générique pour nommer un ensemble d'objets.

Exemple de désignation générique:

```
version (système= unix et auteur= XXX  et langage= C).
```

S'il est intéressant d'identifier la version dans la base d'objets, il est aussi intéressant d'identifier son contenu qui est manipulé hors base pour éviter des incohérences entre versions successives d'objets issues d'une création de version d'objet à partir d'une version d'un autre objet. Ainsi l'incorporation d'informations identifiant l'auteur, la date et la désignation de l'objet, en tête du contenu de l'objet, sous forme de commentaires adaptés au langage, permet d'éviter des substitutions inappropriées et est réalisée par tous les environnements contrôlant les versions.

2.3.2. Les historiques

La relation entre historiques et versions est claire. Les historiques sont des objets de type documents. Ils sont associés aux objets évoluant en versions et sont créés et maintenus automatiquement par l'environnement. Ils conservent une trace des transformations successives effectuées sur un objet. Un historique est constitué d'un ensemble d'informations pertinentes. Ces informations identifient l'auteur du changement effectué, la désignation origine de l'objet à changer, extraite à partir de son contenu et la date du changement. En plus un commentaire est demandé à l'usager pour encourager la documentation des changements effectués.

2.4. Extension du concept de module

Les constructions introduites dans les langages modulaires ne permettent pas de prendre en compte la notion de version comme nous venons de la voir. Nous distinguons deux approches suivant la nature de l'extension envisagée :

- une approche langage qui propose l'extension des langages de programmation par des constructions supportant les versions. A ce titre, nous signalons sans la présenter l'étude faite par [Winkler86]. Cette approche n'a pas été totalement suivie du fait de la complexité de sa mise en oeuvre.

- une approche environnement qui consiste à définir un concept étendu de modules, comme aggrégat d'un ensemble de versions. De nombreux travaux se sont orientés dans cette direction; toutefois la structure interne proposée varie selon le niveau de granularité défini dans chaque environnement. Nous présentons une synthèse des différentes solutions pour intégrer la notion de version dans les modules (voir figure 2.1).

  - Dans Gandalf, le module est composé d'une interface associée à des réalisations qui évoluent en alternatives et révisions.

  - Dans Cedar, un module est une composition d'une interface associée à une réalisation qui évoluent toutes les deux en révisions.

  - Make ne fournit ni définition de module ni version. Il travaille sur des fichiers.

  - DSEE [Leblang84], RCS et SCCS [Rochkind75] ont été développés pour les langages classiques et n'ont pas le concept de module. L'évolution en alternatives et révisions est gérée pour les unités de compilation (les fichiers source).

  - Dans Adèle (actuellement Nomade), la famille (extension de module) est définie comme dans Gandalf et étendue par les concepts de **versions** et **vues** d'interfaces.
  Les versions d'interface sont utiles dans un environnement de maintenance supportant un logiciel ayant une longue durée de vie. Les versions d'interface prennent en compte le fait que les interfaces changent pour s'adapter à l'évolution des besoins. Chacune de ces versions d'interface correspond à une alternative qui a ses propres versions de réalisations.
  Chaque version (alternative) d'interface peut montrer différentes vues d'elle-même. **Une vue d'interface** définit dans une syntaxe donnée un sous-ensemble d'une version d'interface. Les vues sont nécessaires pour protéger la visibilité des interfaces dans la

structuration du logiciel et pour des besoins de compilation (syntaxes) dans un environnement multi-langage. Une interface peut se présenter soit en un sous-ensemble de ressources d'une interface plus complète soit en différents langages dépendant du langage du module qui l'utilise.

SYSTEMES	INTERFACE			REALISATION	
	INTERFACE	VERSIONS	VUES	ALTERNAT	REVISIONS
Make				*	
SCCS					*
RCS				*	*
CEDAR		*			*
DSEE				*	*
Gandalf	*			*	*
Nomade (Adele)		*	*	*	*

Figure 2.1 : Schémas de module : tableau comparatif

2.5. Les relations

La nécessité de relations pour décrire des propriétés importantes du logiciel a été mise en évidence par de nombreuses études [Parnas79]. [Meyer85] a étudié de plus près ces relations. Il montre qu'elles sont nombreuses, variées et que toute création de relation a des conséquences importantes sur les objets qu'elle lie. On peut globalement distinguer deux types de relations :

- les relations de structure. Ces relations expriment la structure d'un objet en le liant à d'autres objets comme par exemple les relations définissant la structure des versions de modules.

- les relations de faits qui lient des objets entre eux mais n'expriment pas la structure comme par exemple une relation de responsabilité qui lie un programmeur au module qu'il maintient.

Cas des systèmes modulaires

Un module peut utiliser les ressources définies dans un autre module, il **dépend** alors de ce module. Les environnements de support à la programmation globale ont étudié le logiciel en terme du graphe de dépendance. Le **graphe de dépendance** est déduit à partir de l'importation de procédures, le partage de variables et l'utilisation de composants partagés ("include") entre modules et représente la structure des systèmes modulaires.

Le graphe de dépendance est utilisé pour:

- effectuer la vérification de types inter-modules;
- déterminer l'ordre de compilation de modules;
- déterminer les modules atteints par la propagation des changements;
- construire les configurations.

Pour décrire la structure du logiciel, les langages modulaires ont été souvent combinés avec les **langages d'interconnexion de modules** dits **MIL** (Gandalf-SVCL [Kaiser83], C/Mesa [Mitchell84]). Les MIL's expriment la structure du logiciel avec une description plus fine des dépendances inter-modules. Ils ont été proposés à l'origine par [De-Remer76]. On trouve une étude complète sur les MIL's dans [Prietro Diaz86].

D'autre relations modélisent des propriétés importantes des objets nécessaires à la programmation globale et sont supportées par les environnements :

- la relation de décomposition hiérarchique d'un logiciel en sous-systèmes;
- la relation d'abstraction représentant des niveaux d'abstraction comme la relation entre interface et réalisation;
- la relation liant des représentations différentes d'un objet comme par exemple un module source et sa documentation;
- la relation de dérivation entre objet source et objets dérivés.

Notons que les environnements de programmation globale ont utilisé une partie des liens et structures d'un projet logiciel, essentiellement ceux répondant à des besoins de compilations et issus des MIL's. En définitive, les MIL's représentent une structure statique du logiciel et n'ont pas souvent la puissance pour s'accomoder à la structure du logiciel lorsque le logiciel évolue. Beaucoup de travaux de développement d'environnements logiciels s'intéressent à la technologie des bases de données pour gérer davantage de types d'objets et de relations ([Belkhatir85], [Ceri83], [Lamsweerde86], [Linton84], [Meyer85], [Snodgrass84]) et pour prendre en compte la grande dynamicité des relations (création et suppression des relations, conséquence de la longue durée de vie du logiciel) qui font évoluer la structure du logiciel.

2.6. Les contraintes sur les objets et les relations.

Une fois définis les objets et les relations, des informations complémentaires sont nécessaires pour indiquer:

- les conditions que leurs valeurs doivent satisfaire;
- les opérations à appliquer aux objets;
- le comportement dans une situation donnée;
- les conditions pour les créer et les conséquences de leur création;

Ces informations appelées **contraintes**, interviennent à différents niveaux du processus de gestion d'un logicel: contrôle de la structure, des modules, des droits d'accés des usagers...

Toutes ces informations sont généralement prises en compte dans les environnements classiques mono-langage comme par exemple Cedar et Gandalf. Les contraintes sur les objets et les relations sont prédéfinies et figées dans le code. Une contrainte importante consiste par exemple à réanalyser tout le logiciel et à recompiler tous les modules dépendant d'un module pour qui une nouvelle version vient d'être créée.

Cette approche a pour avantage d'être efficace mais présente l'inconvénient d'être rigide dans le cas où l'usager veut introduire de nouvelles contraintes ou modifier celles qui éxistent.

Les recherches en cours, inspirées du domaine des bases de données, tendent d'introduire plus de flexibilité dans la formulation des règles de cohérence.

Les travaux identifient trois classes de contraintes à appliquer pour gérer un logiciel.

2.6.1. Les contraintes d'intégrité de la base d'informations

Ces contraintes sont exprimées dans les environnements où le système d'archivage est élaboré, c'est à dire de type systèmes de base de données par opposition aux systèmes d'archivage de base, c'est à dire de type systèmes de fichiers. Ces contraintes expriment la validité permanente des informations stockées dans la base: attributs, relations . Elles sont exprimées à l'initialisation du projet, dans un langage de prédicats combinant les objets, les attributs et les relations. Ce niveau de cohérence est maintenu par le gestionnaire d'objets. On vérifie ainsi que le contenu de la base reste cohérent avec le modèle de définition mis en oeuvre par la base.

Exemple:

définition des attributs valides pour les types d'objets source et code-objet

source:

langage=pascal, pl1, C;

système=unix,vms;

code-objet:

état= test, expérimental, officiel;

mode-compilation= mise-au-point, optimisé

2.6.2. Les contraintes de composition.

Le regroupement des versions de modules dans une version composée (configuration dans notre terminologie) suppose que des règles de compatibilité existent entre les versions. Les critères définissant la compatibilité des versions peuvent différer d'une version composée à une autre et doivent être vérifiés et spécifiés au cas par cas. Une approche intéressante consiste à permettre aux concepteurs d'introduire dans le système les règles qui permettent de regrouper de façon cohérente les versions. En associant les attributs et des conditions sur leur valeurs appelés **contraintes de composition**, il est possible d'assurer automatiquement la maintenance de la cohérence d'une configuration. En particulier il est possible de vérifier ces **contraintes de composition** pour tous les objets dont dépend une configuration.
Les contraintes de composition sont nécessaires pour exprimer le choix d'une version d'objet basée sur des contraintes et des préférences. On distingue deux approches dans l'expression des contraintes de composition selon le niveau de puissance du système d'archivage.

1) Approche fichiers

Le système d'archivage, de type système de fichiers, est rudimentaire. Les versions sont caractérisées par les propriétés gérées par le système d'exploitation (date et auteur) et offrent un niveau d'expression des contraintes de composition très limité, consistant souvent à donner explicitement la liste de composition des configurations. Nous donnons une synthèse de leur formulation dans quelques systèmes représentatifs:

- Le *system modeller* de CEDAR construit des configurations par la définition explicite de la liste de composition. La seule contrainte de composition prédéfinie dans le système est celle qui consiste à prendre par défaut la version la plus récente.

- SVCE de Gandalf définit pour un objet une version dite "standard" sélectionnée automatiquement. La contrainte de composition s'exprime par la spécification de la

version standard à sélectionner lors de la construction d'une configuration. La version standard peut être modifiée dynamiquement.

- Les meilleurs résultats de cette approche sont obtenus par le gestionnaire de configurations de DSEE[Leblang85]. La structure "*configuration thread*" de DSEE définit les contraintes de composition et applique un mécanisme de selection de versions en terme de propriétés statiques portant sur le nom et la date comme les autres systèmes avec en plus des propriétés dynamiques comme par exemple: pour l'objet X choisir la révision en mise au point, pour le reste prendre l'avant dernière...

2) Approche base de données.

Le système d'archivage est plus élaboré et permet la description des objets par des attributs définis par l'usager, offrant ainsi une plus grande richesse sémantique. Les attributs sont divers et une version peut être selectionnée par l'application de contraintes de composition plus élaborées. Ces contraintes sont des prédicats combinant les propriétés comme par exemple la composition d'une configuration avec les versions ayant les attributs (système = unix et langage = C). L'utilisation des contraintes de compositions introduites à l'origine par Adèle, se généralise actuellement [Bernard87], [Pfreundschuh88].

2.6.3. Les contraintes de gestion du logiciel.

Pratiquement tous les environnements de programmation globale font des vérifications basées sur les propriétés sémantiques du logiciel.

Exemple

- Make (Unix) [Feldman79] vérifie que si un objet M2 dépend d'un objet M1, la date de la dernière modification de M1 doit être antérieure à celle de M2;

- les environnements introduisent le concept de configuration cohérente (bien formée) basée entre autres sur la vérification des types inter-modules des modules qui la composent.

La définition de ces contraintes est implicite dans les outils qui font les vérifications comme c'est le cas avec Make et les MIL's.

Une approche alternative qui se développe actuellement consiste à rendre ces contraintes explicites. Les contraintes sémantiques sont exprimées par des règles définissant d'un part, explicitement la sémantique attachée aux objets et aux relations, et d'autre part les actions à appliquer.

Exemple de règle rendant explicite la contrainte Make

```
Pour tout objet S dépendant de D : if (D.date_modification > S.date_modification) alors
Action (compile S)
```

A la modification d'un objet on peut analyser la contrainte Make définie explicitement et dans le cas où elle n'est pas vérifiée, une action qui exécute la compilation est exécutée

Notons que les règles de gestion de logiciel assurent que les propriétés, exprimées par les relations et les objets, sont maintenues de manière cohérente. Leur spécification explicite rend plus fléxibles les stratégies de contrôle désirées. Si on désire un autre type de contrôle, il suffit de modifier la règle. D'un autre côté, les contraintes de gestion de logiciel peuvent être complexes, de différentes natures et nécessitent souvent l'activation d'outils.

La vérification de contraintes de gestion de logiciel et le déclenchement d'actions, définies explicitement, est une des fonctions de la base qui devient un composant actif de l'environnement. Cette fonction est traitée de façon plus détaillée dans (§3.3.3)

Ce point est d'actualité dans les travaux sur la programmation globale [Belkhatir87], [Borison86], [Godart86], [Marzullo86], [Schwanke88a], [Wiebe88]. Cependant, les solutions proposées sont souvent limitées aux problèmes de la recompilation automatique.

2.7. Autorisation et droits d'accès.

Les informations de protection des objets logiciels incluent généralement le concept d'usager, entité active qui déclenche des tâches, et d'objet qui représente les entités passives. Plusieurs mécanismes ont été définis pour réaliser la protection des objets stockés dans la base. On peut les répartir en trois catégories:

- Les systèmes utilisant une protection de base dérivée du système d'exploitation hôte, par exemple du type Unix (lecture, écriture, exécution).

- Les systèmes à base de **listes d'accès**. Une liste d'accès est associée à chaque objet et spécifie les droits possibles, répartis par usager ou groupe d'usagers. La plupart des gestionnaires de base utilisent ce type de protection, comme par exemple Adèle et Gandalf. Adèle spécifie pour chaque module, une liste d'usagers ayant tous les droits sur le module. Gandalf répartit les utilisateurs d'un (ou groupe de) module en trois catégories: l'administrateur qui a tous les droits, y compris la modification des listes d'accès, les programmeurs qui ont le droit de modification des objets et le reste des usagers qui a un droit d'exécution.

- Les systèmes à base de **listes de privilèges** [Minsky84], issus de la technologie de base de données. Par usager (ou classe d') usagers est défini un domaine qui spécifie l'ensemble des objets accessibles. Dans cette approche la désignation des objets et des domaines se fait à partir d'attributs les caractérisant et les droits sont contrôlés par des privilèges. Un privilège est un triplet (opération, attributs, prédicat). Le privilège exprime que l'opération peut être appliquée à tout objet défini par les attributs et satisfaisant le prédicat.

Exemple
usager U :

```
modif  objets (langage=pascal et état= test)
```

L'opération de modification est autorisée pour l'usager sur les sources Pascal en cours de mise au point (test)

L'intérêt de cette approche est la souplesse offerte; les droits ne sont pas figés pour un objet, mais évoluent automatiquement en fonction de son évolution et aucune hypothèse n'est faite sur l'organisation des usagers comme dans les systèmes à base de liste d'accès.

2.8. Conclusion

La base d'objets est fondamentalement le noyau de tout environnement de développement et de maintenance. Elle permet :

- de connaitre la sémantique des objets manipulés (source, binaires, documents..) et de leurs relations;
- de garder un historique de toutes les opérations et les modifications;
- de contrôler les actions exécutées sur les objets.

Les travaux actuels tendent à en faire un composant actif pour mettre en oeuvre des contraintes de gestion logiciel complexes par l'activation d'outils.

Les grandes lignes des caractéristiques des informations de programmation globale étant présentées, nous consacrons la section suivante à la description des fonctions d'un noyau de programmation globale centré sur une base d'objets.

3. NOYAU DE MANIPULATION DE LA BASE

3.1. Notion de noyau

Trois fonctions principales se détachent dans le développement d'environnements de programmation globale centrés sur une base d'objets et constituent le noyau de l'environnement.

1) **La fonction de contrôle de versions.**
La fonction de contrôle de versions implique la création d'une nouvelle versions (alternative ou révision) lors de toute modification d'un objet. Appliquée généralement au code source, elle est appelée **contrôle du code source** et est réalisée par des outils comme SCCS et RCS.

2) **la fonction de gestion de configurations.**
Des noyaux de programmation globale intègrent un gestionnaire de configurations pour aider à reconstituer un logiciel à partir de composants en choisissant la version appropriée.

3) **la fonction de contrôle des changements et d'activation** .
Les propagations d'effets induits par les changements sont analysées et des actions déclenchées. Les environnements offrent actuellement cette fonction dans le noyau de base. Elle permet de contrôler la dynamicité des objets et des relations de la base et sert de canal à la modélisation de méthodes de développement et à l'intégration d'outils externes.

Alors que les deux premières fonctions sont prises en charge par la plupart des environnnements, la troisième fonction est explorée dans de nombreux travaux. Nous allons donné une description détaillée de chacune des fonctions.

3.2. Le contrôle de versions

Les objets sont immuables. La fonction de contrôle de versions crée pour chaque modification une nouvelle version. Les services identifiés et automatisés pour le contrôle de versions sont : l'optimisation du stockage du contenu des versions, l'enregistrement de l'historique d'évolution, la fusion des versions et le contrôle de l'accés aux versions par le mécanisme de réservation-libération. Nous faisons le point sur l'état de développement de chacun des services.

3.2.1. Modifications concurrentes d'un même objet

Dès le développement des premiers environnements de génie logiciel, les études ont conclu que le concept de transaction classique n'etait pas applicable. D'autres environnements de développement et de conception comme la CAO ont conclu de manière analogue [Katz83].
La création d'une nouvelle révision peut nécessiter plusieurs transactions et peut durer assez longtemps. Généralement les programmeurs communiquent avec la base d'objets en déplaçant les données depuis ou vers leur espace de travail .

Le problème à résoudre dans un environnement de développement arrive lorsque deux personnes ont pris chacune une copie du même objet dans leur espace de travail et le modifient. Après une certaine durée, elles décident chacune de modifier l'original par leur copie .

Si des précautions ne sont pas prises, les changements, réalisés par l'une des deux personnes, sont perdus. Ce problème est connu et a été traité convenablement dans l'environnement RCS (*check-in, check-out*). Une demande de réservation de l'objet, avant la modification, donne un accés sur l'objet et positionne un verrou qui bloque l'objet pour la demande de modification suivante. Ce verrou est levé à l'introduction des modifications opérées par l'usager qui a réservé l'objet.

3.2.2. La fusion de développements

Les changements sont opérés, souvent pour des contraintes de temps, en parallèle sur deux versions alors que logiquement ils se suivent. A terme, le problème de la fusion des développements se pose. L'automatisation de la fusion des changements effectués en parallèle pose des problèmes qui restent, pour une large part, ouverts, comme en témoignent les nombreux travaux sur le sujet [Rohrbach88] [Reps88] . Le problème n'a pas de solution générale, puisque toutes les fusions ne peuvent pas se faire automatiquement. Dans des cas simples et souvent fréquents il est possible de fusionner ou d'offrir une assistance à la fusion. Par exemple dans DSEE[Leblang88], la fusion de deux versions d'un système d'exploitation de 100 fichiers sources, développés en parallèle, a été faite automatiquement à 90%. Pour les 10% restants, l'assistance de l'usager était requise.

Exemple de fusion dans RCS:

La commande "rcsmerge" de RCS fait la fusion par comparaison textuelle à une racine commune, la revision rev0. La commande prend comme entrée deux révisions rev1 et rev2 correspondant à deux lignes de développements parallèles.
la commande rcsmerge détermine trois types d'ensemble de lignes:

type1: ensembles communs aux trois révisions;

type2: ensembles identiques dans deux révisions.

type3: ensembles différents dans les trois révisions.

Pour tous les ensembles de type 2, où rev1 est la revision qui diffère par rapport aux deux autres, les ensembles de lignes de rev1 remplacent celles correspondantes de rev2. Le type 3 signale un cas d'erreur et demande l'intervention manuelle des usagers pour sélectionner la révision correcte.

3.2.3. La gestion optimisée de l'espace d'archivage

La prolifération des révisions dans un environnement de maintenance pose un problème de gestion de l'espace disque. Le mécanisme dit de **deltas** [Tichy85] est appliqué. Ce mécanisme fait l'hypothèse que les différences d'une révision à une autre sont de quelques lignes. La technique habituellement appliquée consiste à garder en clair la dernière révision et à recréer automatiquement les révisions antérieures à partir des différences conservées sous forme de commandes d'éditeur de texte. Le mécanisme de delta apporte un gain substantiel dans la gestion de l'espace disque. Les mesures effectuées montrent qu'un fichier de delta dont la taille est celle du code source peut contenir en moyenne de 50 à 100 parfois 200 révisions du même code source[Leblang 88].

3.2.4. Discussion

La fonction de contrôle de versions a été développée sous forme d'opérations séparées gérant des fichiers spéciaux, puis a été intégrée dans le système de fichiers (DSEE) pour protéger les fichiers de versions contre des destructions accidentelles. Plusieurs environnements récents [Zdonik86] proposent d'incorporer la fonction de contrôle de versions dans la base et de la généraliser à tous les objets de l'environnement et non seulement le code source .

3.3. La gestion de configurations

3.3.1. Le problème

Les logiciels de taille importante changent constamment durant leur longue vie et engendrent des composants en versions multiples. Un logiciel typique est composé de plusieurs centaines à plusieurs milliers de composants. Chacun est maintenu en plusieurs versions.
Les composants gérés sous forme de versions rendent possible l'existence d'une famille de logiciels pouvant être configurés de différentes manières. Nous avons défini dans (§2.3.1) la configuration en terme de contenu. D'un point de vue logique une configuration est une sélection d'un des logiciels possibles.

Pour des logiciels de cette taille, les environnements donnent la possibilité d'automatiser le processus de construction de configurations.

3.3.2. La construction de configurations

La fonction de gestion de configurations a besoin :
- d'une base où sont stockés les composants du logiciel,
- d'une description de la structure du logiciel,
- de règles de composition des versions pour chacun des composants de la configuration,
- des règles de traduction à appliquer pour générer les objets dérivés comme par exemple une version exécutable.

Les relations mises en oeuvre dans le processus de construction de configuration dans le cas de systèmes modulaires sont:
- la relation de dépendance,
- les relations liant les révisions d'une alternative,
- la relation "*est_réalisée_par* " entre une interface et un corps dans les bases où la notion d'interface est définie.

Les services fournis par la fonction de gestion de configuration sont la définition et la génération de configuration.

3.3.3. La définition de la configuration

Une configuration est définie par deux types de spécification: la description de la structure de la configuration et les contraintes de composition de versions à appliquer (ce dernier point a été examiné dans §2.6.2). Pour ce qui concerne, la description de la structure de la configuration, deux approches sont envisagées:

- les modules d'une configuration sont automatiquement extraits à partir du code source pour les langages qui spécifient les dépendances directement dans le code. C'est la stratégie adoptée dans Adèle;

- Les modules d'une configuration sont décrits dans une structure séparée identifiant les composants de la configuration et leur structure [Leblang85].

La phase de définition détermine les versions de chaque source de composant à combiner pour créer une configuration. Cette phase est consolidée dans des systèmes comme Cedar et Gandalf parce que mono-langage par la vérification des types inter-modules.

3.3.4. La génération d'une configuration

Cette étape consiste à créer des objets dérivés par l'application d'outils comme par exemple l'exécutable d'une configuration. La plupart des systèmes offrent pour la génération un support dérivé de l'outil Make d'Unix. Les traitements appliqués sont la compilation des objets et l'édition de liens. Cependant d'une manière générale, des traitements divers peuvent être entrepris (utilisation de préprocesseurs, application de traitements répétitifs sur les objets de la configuration...). La maintenance d'un cache d'objets dérivés est utilisée dans la plupart des gestionnaires de configurations pour accélérer la phase de génération.

La phase de génération maintient un historique chargé de conserver les informations utilisées dans la génération d'une configuration comme la liste de composition, les versions d'outils et les paramètres appliqués, la date, l'auteur et un commentaire demandé à l'usager pour documenter la construction faite.

Notons, que nous ne présentons ici qu'une introduction à la fonction de gestion de configurations; voir [Estublier88] [Tichy88] pour plus de détails.

3.3.5. Conclusion

Les mécanismes à la base du gestionnaire de configuration logiciel semblent bien compris . C'est une activité automatisable lorsqu'elle est réalisée par une base de données.
Pour construire un logiciel correctement, les configurateurs ont besoin de comprendre:

- les dépendances entre composants d'un système;
- l'organisation des versions et les contraintes de composition de versions;
- les règles de dérivation pour obtenir par exemple les codes objets et les exécutables;

D'un autre côté, pour plus d'efficacité, les configurateurs opèrent souvent de manière incrémentale en ne reconstruisant que les parties inadaptées (utilisation de configurations partielles lors de la phase de définition d'objets dérivés non altérés lors de la phase de génération). A signaler les travaux actuels où les configurateurs pour une plus grande efficacité, activent des générations parallèles en utilisant la puissance de calcul du réseau local [Leblang88].

3.4. Maintenance globale et activation d'outils

3.4.1. Le problème des données périmées

Le processus de développement de logiciel est long. Les changements sont fréquents parce que la cohérence de certains objets n'est atteinte que progressivement. Ces changements, à

cause de la structure fortement inter-liée du logiciel, invalident une partie du logiciel à développer. C'est un des problèmes importants de la maintenance globale. Pour une meilleure assistance dans le développement, le système doit tolérer ces états incohérents et les gérer. Il y a donc un besoin de mécanismes pour contrôler la propagation des changements [Belkhatir86a].

3.4.2. Relations et propagation.

Les relations mises en œuvre, pour détecter les effets propagés après des changements dans la base, sont le plus souvent implicites. Ainsi, un *Makefile* est une façon indirecte d'exprimer que les fichiers mentionnés ont une relation (indéterminée) entre eux. Le plus souvent, il s'agit uniquement de la relation de dépendance.

La plupart des environnements détectent les effets propagés juste après que l'action qui les a provoqués soit exécutée. Dans cette classe, se trouve les environnements mono-langage tel que Gandalf, Cedar et d'autres systèmes comme FASP[Stuebing84] et LLNL[Blattner84].

3.4.3. L'activation d'outils

Etant donné que généralement seule la relation de dépendance est connue, les environnements se contentent le plus souvent de recompiler automatiquement les modules qui utilisent une interface modifiée. L'intérêt d'une telle approche réside dans le calcul automatique du bon ordre de recompilation, ce qui n'est pas trivial dans un gros logiciel, et l'invocation automatique des compilateurs. La recompilation automatique systèmatique fait déjà apparaître des divergences de points de vue dans la mesure où systèmatiquement à chaque modification de nombreuses compilations sont lancées. Ce dernier aspect est une des préoccupations actuelles comme en témoignent les études de développement de méthodes et d'outils pour minimiser le côut des recompilations [Tichy86], [Schwanke88b].

Le problème est plus délicat encore si des relations autre que la relation de dépendance sont prises en compte pour prendre en compte d'autres dépendances sémantiques.

Il apparait que le traitement des propagations ne peut pas être prédéfini dans l'environnement et qu'il faut dissocier la détection de l'effet propagé de son traitement. Le premier est à la charge de l'environnement, le second permet que soient exécutées des actions explicitées par l'usager via les contraintes de gestion de logiciel. Les actions définissent les outils à activer, les types d'objets, les transformations à appliquer, les conventions et les tâches à lancer. Cette approche permet que toute politique de gestion doit être facilement implémentable. Odin [Clemm85] [Clemm88], Marvel [Kaiser87b] et Nomade [Belkhatir87] sont des exemples de systèmes supportant la gestion de contraintes de gestion de logiciel et l'activation d'outils.

3.4.4. Conclusion

Les mécanismes d'analyse des effets des changements sont rendus plus généraux et utilisent pour ce faire:

- des contraintes de gestion du logiciel explicites pour contrôler la propagation des changements;
- des mécanismes de déclenchement de type événements-actions pour activer les outils.

Des mécanismes similaires d'activation automatique ont été développés et utilisés dans d'autres domaines [Dittrich86a], notamment :

- les bases de données utilisent l'activation automatique pour contrôler des contraintes d'intégrité complexes: mécanisme de *triggers* [Eswaran76]. Les triggers peuvent être utilisés pour déclencher des messages: mécanisme d'*Alerter* [Bunemann77] [Bunemann79].

- les langages de programmation utilisent l'activation pour réagir à des événements particuliers: mécanismes d'exception [Krakowiak85].

4. IMPLANTATION DE LA BASE D'OBJETS DE PROGRAMMATION GLOBALE

Après avoir examiné la nature des informations de programmation globale et les fonctions de base d'un noyau d'environnement de programmation globale, nous présentons ici les systèmes qui ont été développés pour supporter la base d'objets et les fonctions identifiées en retraçant leur évolution.

Les premiers supports de l'activité génie logiciel sont basés sur le concept d'outils : *Toolkits approach* [Kernighan81]. Dans cette approche, il n'y a pas d'outil d'archivage des informations engendrées durant le cycle de production. Les objets sont des fichiers gérés par le SGF de la machine hôte. La gestion des relations et de la cohérence de ces objets est à la charge des usagers [Nestor86].

[Cooprider79] a proposé l'idée de base de stockage persistante pour supporter :

- les objets logiciels,
- l'architecture d'un logiciel,
- le contrôle de versions,
- la construction de systèmes modulaires.

Par la suite, le rapport de [Buxton80] propose une architecture d'environnement de programmation globale pour le langage ADA et donne à la base de données le rôle de noyau central d'un environnement de développement. Le rapport spécifie les besoins d'un environnement de support à la programmation ADA (APSE) et distingue trois niveaux.

- le noyau KAPSE représente les fonctions base de données, de communication et de support à l'éxécution de programmes ADA;

- le niveau intermédiaire MAPSE est constitué d'un ensemble d'outils basiques pour le développement, l'exécution de programmes et la gestion de configurations en ADA;

- la couche haute APSE comporte des outils pour une méthode particulière: mesures de performance, contrôle de projet, aide à la spécification.

Chaque niveau a sa spécificité. Les niveaux KAPSE et MAPSE sont consacrés à l'intégration d'outils de la programmation (détaillée et globale), le niveau APSE pour l'adaptation à un environnement particulier de développement.

Nous distinguons deux approches dans notre présentation selon que la technologie des bases de données a été ou non utilisée dans les environnements. Dans ce dernier cas, c'est l'approche

dite environnement de programmation parce qu'à la base de la plupart des travaux sur les environnements; dans l'autre cas c'est l'approche base de donnée. Nous examinons ensuite les tendances actuelles.

4.1. L'approche environnement de programmation

L'approche environnement de programmation a développé son propre gestionnaire de la base d'objets. C'est l'approche dite *system modelling* [Marzullo86]. Un ***system model*** décrit les relations entre composants, les informations sur les versions et les règles de construction d'un logiciel. Les outils accèdent au *system model* pour créer de nouvelles versions, compiler et construire des configurations. Ces systèmes utilisent à la base un langage d'interconnexion de modules (MIL) comme langage d'expression du *system model* (la figure 2.2 montre un exemple de *system model.*). Les informations du system model sont compilées et stockées dans des fichiers spéciaux qui font référence à l'ensemble des composants du logiciel stockés et gérés par un SGF classique.

```
system S is
   provide
         procedure a;
   require c,d;
 /* le sous systeme S peut être configuré de deux manières: une version sun et une
version macintosh */
            composition    S.sun = (f1.v1.mod, f2.v2.mod)
            composition    S.macintosh= (f1.v2.ada, f2.v1.ada)

module f1                                  module f2
         provide     procedure a;                provide     procedure b;
         require     d,b;                        require     c;
         implementation                          implementation
         v1.mod                                  v1.ada
         v2.ada                                  v2.mod
end f1                                     end f2
end S
Les réalisations du module f1 et f2 sont décrites à l'extérieur du system model
```

Figure 2.2 : Exemple de *system model*

Des systèmes comme Gandalf, Cedar, DSEE et Jasmine[Marzullo86] utilisent l'approche *system model*.. Ces environnements sont simples, efficaces et gèrent un volume important d'informations.

D'un autre côté, la base est restée un outil d'un bas niveau de généralité. Les types d'objets, les relations et les contraintes sont prédéfinis et figés. La relation supportée est uniquement la relation de dépendance. Les attributs connus sont ceux du système hôte comme la date de dernière modification, de protections et dans certains cas ceux définissant l'état d'un objet. La base d'objets développée est une base spécialisée pour les actions de compilation dans laquelle:

- les objets sont les modules,
- les attributs de description sont dérivés du système d'exploitation,
- la relation décrite est la relation de dépendance,
- les contraintes sont prédéfinies comme par exemple dans "*make*" où la date de modification d'un objet doit être plus récente que celles des objets dont il dépend.

La plupart des environnements actuels sont conçus selon cette approche. Une approche alternative consiste à construire l'environnement au dessus d'un SGBD qui supporte des langages de définition des objets, relations et contraintes et des mécanismes systèmatiques de contrôle de cohérence, de journalisation, de droits d'accés et de synchronisation des accés.

4.2. L'approche base de données

4.2.1. Rappel sur les modèles de bases de données

Nous rappelons brièvement les caractéristiques des deux modèles introduits dans les environnements:

- le modèle relationnel déjà expérimenté dans de nombreux projets;
- les modèles sémantiques en cours d'investigation.

1) Le modèle relationnel

Le modèle relationnel [Codd70] [Delobel82] structure les informations sous forme de **relations**. La définition des données utilise les relations et les attributs. Les attributs sont nécessairement atomiques : chaine, entier, réel, booléen. Des langages de manipulation existent (langages algébriques, prédicatif) et permettent d'exprimer des requêtes sur les relations sans se soucier de la manière de rechercher les informations. Certains systèmes permettent la modification dynamique du schéma relationnel comme par exemple l'ajout et la suppression d'attributs.

Les avantages du modèle relationnel sont :

- sa simplicité;

- le concept d'indépendance des données.

Les inconvénients se révèlent dans la manipulation de structures complexes :
- les types de données sont élémentaires;
- la représentation tabulaire des informations est inappropriée à la représentation d'objets structurés;
- l'usager raisonne en terme de représentation et non en terme de signification des données.

Exemple de schéma relationnel :

La description d'une réalisation de module peut s'exprimer par les deux relations suivantes :

`texte (nom, jour, mois, année)` qui définit le texte des composants stockés.

`réalisation-module (sourece, code-objet, doc-réalisation, commentaire)` qui définit la réalisation du module.

Les arguments jour, mois et année sont de type entier; les autres sont de type chaines de caractères.

Les relations entre texte et réalisation de module sont exprimées implicitement (nom<->source, nom<->code-objet et nom<->doc-réalisation). Théoriquement, on peut faire l'opération "join" entre 2 attributs de même type.
Cette opération aura un sens dans le cas d'un "join" entre nom et source. Elle n'aura aucun sens dans le cas de commentaire et nom.

2) Les modèles sémantiques

Partant du modèle relationnel, des extensions ont été apportées et ont donné naissance à des modèles désignés sous le vocable de "modèles sémantiques" [Borgida84][Chen76] [Codd79][Peckham88]. On distingue deux types de modèles :

- le modèle entité-association [Chen76] où une séparation est introduite entre les entités et les relations qui peuvent exister entre entités. Le concept d'entité a introduit une distinction entre les propriétés qui représentent les objets et celles qui représentent les relations entre objets. La notion de type d'entité a permis d'augmenter l'intégrité des données puisqu'elle exprime une contrainte sur les relations entre objets qui doivent être du type spécifié.
La principale critique formulée sur ce modèle est qu'au stade de la conception du schéma des données, le choix n'est généralement pas simple entre attribut d'entité et attribut de relation.

- les modèles orientés objet considèrent la base comme un ensemble d'objets. Par rapport au modèle entité-relation, les relations sont traitées comme des entités dans le sens où elles

peuvent être référencées par d'autres relations. La distinction entre entité et relation est supprimée. Les associations entre entités sont modélisées sous forme d'association entre attributs. Par conséquent, il y a un seul concept d'objet qui peut avoir des attributs qui expriment les relations et qui peut être désigné comme une entité.

Exemple :

Reformulons l'exemple précédent en utilisant les modèles sémantiques.

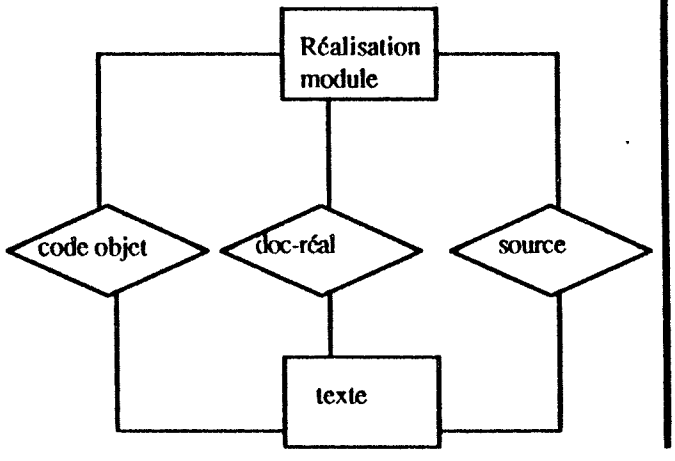


diagramme entité relation

```
objet:  texte
        nom : chaine,
        jour : entier;
        mois:entier,
        année:entier
fin

objet:  Réal-module
        source:texte,
        doc-réal:texte,
        co : texte;
fin     commentaire:chaine
```

représentation objet

clé	nom	Jour	mois	année
$1	a.ada	30	12	87
$2	a.o	10	02	88
$3	a.doc	20	11	86
$4	b.mod	20	01	88
$5	b.o	21	01	88
$6	b.doc	10	08	87

TEXTE

clé	source	co	doc	commentaire
$5	$1	$2	$3	"pile"
$6	$4	$5	$6	"liste"

REALISATION-MODULE

L'utilisation des modèles sémantiques permet de structurer les objets par le typage des attributs.

- Chaque objet est identifié par une clé unique qui est automatiquement instanciée par le système.
- L'opération "join" n'est appliquable que sur les clés.
- Les relations entre les objets texte et réalisation de module sont spécifiées explicitement.

Pour exprimer davantage de sémantique, les modéles sémantiques supportent des abstractions particulières. Les abstractions les plus couramment définies sont : **l'agrégation, la généralisation / spécialisation; la classification et l'association** [Smith77],[Brodie84]. L'agrégation permet de considérer la relation qui existe entre différents objets comme un objet d'ordre supérieur. La généralisation-spécialisation permet de modéliser le fait qu'une classe d'objets est plus générale ou plus spécifique qu'une autre, la relation qui lie la classe générale à la classe plus spécifique est appelée ("*is_a*": "est-sous-classe-de, est-spécialisation-de"). La classification permet de regrouper les objets en classe suivant leurs caractéristiques. L'association permet de définir un objet (*set object)* comme un ensemble d'objets (*member-object*) et par conséquent de définir une classe avec des ensembles d'instances d'autres classes.

Une autre caractéristique des modèles sémantiques permet de considérer l'objet avec l'ensemble des opérations qui lui sont associées. Cette approche type abstrait permet de changer la réalisation sans changer la spécification de l'opération. Par rapport au relationnel qui dissocie opérations et représentation, elle donne un contrôle sur la manière de représenter les données.

4.2.2. Utilisation du modèle relationnel

Des projets se sont intéressés à la technologie des Base de données essentiellement relationnelles. La base d'objet est gérée par un SGBD de type relationnel.

Les informations d'interconnexion du *system model* sont analysées et stockées sous forme de relations [Narayanaswamy87].

Les SGBD rendent plus accessibles les informations stockées (analyse, recherche). Dans ces modèles, les types d'objets et les contraintes ne sont pas figés mais s'expriment par des prédicats combinant relations, attributs et objets.

Par contre les SGBD relationnels ont été développés pour les applications commerciales et les résultats d'expérimentations menées, ont montré qu'ils ne sont pas adaptés au support d'un environnement logiciel [Penedo86], [Bernstein87] [Lamsweerde86].

Outre les temps de réponse prohibitifs (voir §2.2.1), on peut résumer comme suit les fonctions nécessaires pour la gestion du logiciel et non supportées par le modèle relationnel:

- la gestion de versions ;
- la gestion d'objets de taille importante (texte de programmes, documents);
- les mécanisme de structuration (hiérarchie, graphes);
- des opérateurs de calcul sur un graphe comme par exemple la fermeture transitive;
- l'expression et le contrôle de contraintes complexes.

Pour pallier à ces insuffisances les gestionnaires d'objets, utilisant un SGBD relationnel, ont introduit des adaptations.

Des gestionnaires d'objets logiciels hybrides [Boehm82], [Meyer85], combinant un SGBD classsique et des outils propres à la gestion de versions configurations comme RCS, SCCS, Make et le SGF de la machine hôte ont été développés.

- le SGF est le support de stockage des objets,
- des outils comme SCCS ou RCS gèrent les mises à jour successives de chaque produit (deltas),
- le SGBD comme, par exemple Ingres, maintient le modèle de description,
- l'outil Make décrit le processus de génération.

Un des problèmes de cette approche est de garantir la cohérence des informations stockées dans le système de fichiers et celles de la base de donnée en fournissant une interface unique de manipulation. Notons que [Osterweil81] recommande le développement de bases spécialisées pour répondre aux besoins de génie logiciel.

4.2.3. Tendances actuelles : introduction des modèles sémantiques

Les investigations en cours tentent d'unifier les différents mécanismes dans une structure d'accueil présentant davantage de généralité et de flexibilité. La base d'objet intégre, autour d'un schéma de définition, des objets logiciels (sources, documents, données de test); leurs propriétés et des relations de différents types. Cela oblige de repenser la construction de la structure d'accueil.

Parallèlement, les spécialistes des bases de données se sont intéressés au développement de nouvelles applications (CAO, burautique, génie logiciel). Le développement de SGBD extensibles, pouvant être reconfigurés dans le cadre d'applications spécifiques, est à l'étude comme en témoignent de nombreux projets (POSTGRES [Stonebraker86], EXODUS [Carey85], GENESIS [Batory86]). Les modèles de données spécifiés sont plus riches (nouveaux types, dynamicité) grâce au couplage base de données (gestion d'un volume important d'informations) et techniques d'intelligence artificielle (sémantique riche) [Alperin87] [Tichy87].

Les développements entrepris pour explorer la structure et la conception des bases de support à la programmation globale tendent à unifier les résultats obtenus par les communautés bases de données et génie logiciel [Rudmik86], [Dittrich86b], [Kaiser87b] [Wile86]. Le rapprochement des deux domaines a donné naissance à une nouvelle génération de noyaux. Les solutions envisagées dans la plupart des projets de développement de noyau de

programmation globale, semblent adopter un bon compromis entre la technologie base de donnée récente et l'intégration des mécanismes propres aux environnements de programmation.

Cette nouvelle génération développe des gestionnaires de base spécialisés qui:

- intègrent des concepts provenant des modèles "entités relations" et "orientés objets";
- supportent l'activation d'outils [Goldstein80], [Balzer86];
- sont étendus par des caractéristiques propres au logiciel telles que les versions et configurations [Zdonik86], la structuration de données de taille importante et la manipulation de graphes .

Ces travaux sont actuellement au niveau prototype.

5. LES SYSTEMES DE PROGRAMMATION GLOBALE

Plusieurs projets ont exploré le développement d'outils et d'environnements de support à la programmation globale. Nous illustrons ici, par quelques travaux parmi les plus significatifs, l'évolution des systèmes développés et les tendances actuelles de la recherche. En conclusion, quelques voies de recherche, explorées dans cette thèse, sont esquissées.

5.1. Les outils de première génération

SCCS, RCS et Make sont les outils pionniers de la programmation globale. SCCS, RCS et Make ont été développés pour le système d'exploitation Unix et traitent respectivement les problèmes de stockage de versions et de construction de logiciel de taille importante.

Nous décrivons ces outils puisque nous retrouvons leurs fonctionnalités incorporées sous différentes formes dans les environnements qui ont été développés par la suite.

5.1.1. SCCS

SCCS est un système de contrôle de programmes source permettant de produire et de retrouver des versions de fichiers évoluant sous forme de révisions.
SCCS supporte trois fonctions:

- Le stockage des révisions
SCCS gère la trace de toutes les révisions d'un fichier source. L'espace occupé sur mémoire secondaire est minimisé en utilisant le mécanisme de *deltas*. Les révisions associées au code source sont maintenues dans un fichier Unix spécial: *s_file*. Des caractéristiques identifiant l'auteur, la révision et la date de modification constituent l'entête de toute révision. Un historique des modifications est maintenu, enregistrant pour chaque changement de révision, l'auteur, la date et un commentaire. Le commentaire est demandé à l'utilisateur lors de l'archivage d'une révision pour décrire les changements effectués.

- La synchronisation des développements
SCCS permet la synchronisation au niveau de chaque fichier source. Un seul usager peut accéder un *s_file* en modification. La commande explicite de demande de modification (commande "*get*") positionne un verrou sur l'ensemble du fichier *s_file* bloquant ainsi toutes les révisions d'un composant. Ce verrou est levé lors du stockage de la révision (commande "delta").

- la protection des accès aux révisions

SCCS permet de restreindre l'utilisation des révisions à quelques utilisateurs. Le contrôle de l'identification de l'usager a lieu lors de l'accès au *s_file*.

5.1.2. RCS

RCS, développé après SCCS, a les mêmes fonctions et introduit quelques améliorations. Les versions évoluent en alternatives et révisions formant un arbre de révisions (voir §2.3.1). La granularité de la synchronistion est plus fine. Elle porte sur une alternative et non sur l'ensemble des révisions comme dans SCCS. Ce qui permet d'effectuer des développements en parallèles. Une version d'alternative peut être créée à partir de toute révision. RCS aide à la fusion des développements effectués en parallèle en comparant deux révisions de deux alternatives par rapport à leur révision racine.

5.1.3. MAKE

Make [Feldman79] [Feldman88] introduit l'idée de reconstruction automatique d'un logiciel, basée sur l'analyse des dépendances entre ses composants. La reconstruction se fait par l'activation automatique d'outils et produit des objets dérivés. La description du processus de dérivation est décrite dans un fichier spécial maintenu par l'usager "*le Makefile*" . Le langage de commande du système Unix est utilisé pour décrire les outils à activer et les objets sur lesquels ils s'appliquent. Des règles implicites de dérivation facilitent l'écriture des *Makefiles*. L'activation d'outils est basée sur la relation de dépendance et sur la date de dernière modification des objets. L'exécution de Make est demandée explicitement par l'usager et produit l'analyse automatique de la date de dernière modification des objets décrits dans le *Makefile*. Pour tous les objets dépendant d'un objet ayant une estampille plus récente, les activités qui leur sont associées décrites dans le Makefile, sont alors exécutées.

Notons que Make est un outil général qui peut être utilisé dans d'autres applications que la construction du logiciel, telle que la reconstruction d'un document à partir de textes modifiées.

Du fait de sa généralité, les critères de reconstruction appliqués par Make ne sont pas très fins et font relever quelques insuffisances de l'outil:

- le contenu du *makefile* peut ne pas correspondre à la description d'un système si ce dernier évolue sans qu'il soit changé;

- le changement d'un *makefile* n'entraine pas la reconstruction automatique du système jusqu'au moment où un des composants du système est édité;

- Make reconstruit automatiquement tous les objets dérivés qui dépendent d'un composant modifié, si bien que de simples changements de commentaires ont pour effet de déclencher une cascade de recompilations non nécessaires.

5.1.4. Discussion

SCCS et RCS réalisent une base primitive de versions basée sur un stockage efficace du source (utilisation de deltas) et la gestion d'un historique retraçant les modifications successives. Les deux systèmes ont imposé une discipline dans le développement du logiciel, par le mécanisme de reservation-libération, contribuant ainsi à assurer la cohérence des développements.
Make est un outil très bien adapté à la construction de logiciels mais dans un contexte mono-usager, mono-version. Il est difficilement appliquable dans un contexte multi versions où le seul attribut "date" est insuffisant.

Les outils de programmation globale développés par la suite ont repris les caractéristiques de Make, SCCS ou RCS et les ont unifiées et étendues sous forme d'un système de programmation globale offrant une meilleure cohérence par l'utilisation d'une base commune.

5.2. Les systèmes de programmation globale

Nous présentons les résultats atteints par trois projets assez différents dans leur démarche et représentatifs de l'approche *system model*

- CEDAR et GANDALF sont deux systèmes développés dans le cadre de la recherche et offrent un support puissant au développement de systèmes modulaires.

- DSEE est un système développé dans un cadre industriel. DSEE s'inspire fortement de SCCS, RCS et Make mais offre un meilleur support pour la gestion des objets et des configurations.

Parce que la recherche décrite dans cette thèse s'appuie sur Adèle, nous le situons par rapport aux autres environnements et montrons ses limitations qui nous ont amenés à le faire évoluer. Adèle est présenté plus en détail dans le chapitre suivant.

5.2.1. GANDALF-SVCE

Gandalf [Habermann86] [Notkin85] est un environnement de développement de logiciel conçu pour le langage GC (extension de C avec les aspects de modularité). Gandalf est constitué d'outils de programmation détaillée et globale.

- Les outils de programmation détaillée sont basés sur l'édition syntaxique des programmes.

- Les outils de programmation globale comprennent un gestionnaire de versions et un gestionnaire de configurations constituant le système SVCE [Kaiser83], et un gestionnaire de projets. Le gestionnaire de projets supporte la gestion des droits d'accés, la gestion des historiques et les accés concurrents, habituellement intégrés dans le gestionnaire de versions. Nous présentons les grandes lignes de la partie SVCE du système Gandalf.

SVCE contrôle les versions et les relations inter-modules. Le langage SVCL (de type MIL) permet à l'usager de décrire les composants du logiciel, leurs versions et les relations qui les lient. Le logiciel est structuré en un ensemble de modules et de sous-systèmes. La figure 2.3 montre la description d'une structure de logiciel en SVCL. Tout module décrit les ressources fournies par son interface et les versions des modules importés. Une seule version, par module importé, est spécifiée. Un module regroupe plusieurs réalisations d'une même interface. Les réalisations sont organisées en versions spécifiées sous formes d'alternatives et de révisions. Tout module a une alternative et une révision dites "standards" qui sont sélectionnées par défaut dans la construction d'un logiciel.

On distingue deux types d'alternatives: les implémentations et les compositions.
Une implémentation contient le texte du programme et une liste de modules dont elle dépend.
La composition spécifie les modules à utiliser lorsqu'il en existe plusieurs qui fournissent les mêmes fonctionnalités. Une composition est une liste de modules sans source de programme. Chaque composition indique les versions de modules et compositions qui implémentent le module. Gandalf est un des premiers systèmes à gérer uniformément implémentation et compositions comme des versions de réalisation.

Un système décrit en SVCL est dit complet lorsque la clause des ressources qu'il exporte est vide comme le montre l'exemple de la figure 2.3 où M1 est le module de haut-niveau.

```
 Exemple de description en SVCL et Base associée
Box B
 module M1
  standard impl V1
   with M2.C1;
   defaults M2.V1
    rev1
    standard rev2
  impl v2 with M2;
   defaults M2.V1
   standard rev1..
 end M1
 module M2
  provide ( description interface)
  standard comp C1= (M3,M4)
  impl v1 with M3. V4
   standard rev1...
 end M2
end B
```

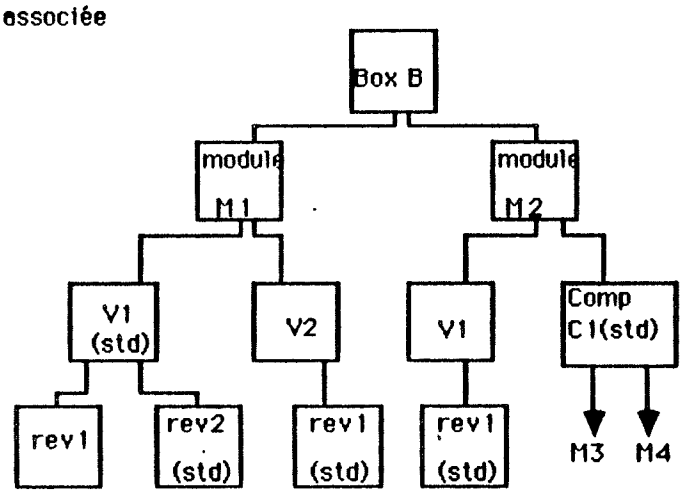


Figure 2.3 : Exemple de description en SVCL-Gandalf

Remarque: A la construction d'un système, si l'option "with" ne spécifie pas le nom de version, le système choisit la version spécifiée par défaut, sinon celle définie avec l'option standard est sélectionnée.

Si l'usager sélectionne le module M1, la configuration est définie comme suit :

conf= M1-V1, M2.C1 + standard de M3 et M4...

Si l'usager sélectionne le module M1. V2 alors conf= M1.V2, M2.V1 + M3.V4...

La base de donnée, créée après la compilation de la description SVCE, est organisée de manière hiérarchique comme une forêt (voir figure 2.3 pour l'exemple de structure produite). Le sommet de tout arbre est une boîte. Une boîte est un ensemble de modules. Les boîtes peuvent inclure d'autres boîtes (sous-système) et modules. Les mécanismes de désignation et de protection sont greffés sur la structure d'arbre. Un chemin d'accés identifie de manière unique un module.

La protection est à base de "liste d'accés". A chaque boite est associée une liste de personnes autorisées à les manipuler et réparties en trois classes (administrateur, programmeur et autres usagers).

SVCE permet à partir d'une description de la structure du logiciel de construire automatiquement une version exécutable. Gandalf a développé un mécanisme d'attachement

procédural: les *actions routines* [Medina82] pour l'analyse sémantique incrémentale des programmes à l'édition. L'édition sysntaxique et les *action routines* ont été généralisées et appliquées à la description en SVCL.

- Dès qu'un composant est modifié , des *actions routines* attachées aux boites et modules permettent le contrôle de versions, la propagation des changements et la regénération des configurations atteintes.

- Dès qu'une composition est modifiée, Gandalf lance une série de contrôles pour vérifier sa validité comme par exemple la concordance des types inter-modules.

Dans SVCE, la description de la structure du logiciel et le contrôle de versions sont liés. La coordination est assurée par l'utilisation d'une base de données. Les programmes sont dans la base et les informations pour les recompilations sont dérivées des liens entre programmes représentés dans la base.

5.2.2. CEDAR - system modeller

Cedar [Teitelman84] [Swinehart85] est un système d'exploitation et un environnement de développement distribué développé à Xerox Parc. L'environnement Cedar est conçu spécialement pour le langage modulaire Cedar. Le langage Cedar est bâti autour du concept de module composé d'une partie définition ou interface et d'une partie réalisation. Chaque partie est une entité stockée dans des fichiers distincts et compilés séparemment. Cedar ne fournit pas de fonction de contrôle de versions mais permet à chaque module d'avoir plusieurs copies . Chaque version (copie) de module est identifiée par une désignation unique constituée à partir de son nom et de sa date de création. Cedar a développé un outil de support à la programmation globale: le *"system modeller"* [Lampson83] chargé de la gestion de configurations. Nous présentons, dans ce qui suit, ses carctéristiques essentielles.

Le *system modeller* de Cedar comme le SVCE-Gandalf utilise un langage d'interconnexion de modules (SML dérivé de C/Mesa) pour décrire un système. C/Mesa a été le premier MIL à considérer l'interface et l'implémentation comme deux objets distincts. Cela permet d'une part, de gérer des versions d'interface et d'autre part, de décrire la configuration sans décrire le flot de ressources . Dans Gandalf, comme les interfaces ne sont pas des objets gérés sous formes d'entités séparées, leur description est faite de manière explicite dans le SVCL; ce qui ne permet pas de gérer leur évolution sous forme de versions.

Le *system modeller* est coordonné avec l'éditeur de Cedar qui l'avertit de l'introduction de toute nouvelle version. Le *system modeller* construit une nouvelle description de la configuration à partir de l'ancienne incluant la nouvelle version, rassemble les révisions et les codes objets non

altérés de tous les composants sur le site local et lance la construction de la configuration. La construction de configuration se déroule en plusieurs étapes:

- vérification de la cohérence de la configuration. La vérification de cohérence détecte les modules omis, superflus et les versions incorrectes.

- détermination des dépendances de compilation et génération automatique des commandes nécessaires à la génération de la configuration.

Les objets dans Cedar sont répartis sur le réseau local et évoluent indépendamment les uns des autres. La cohérence des systèmes gérés par Cedar est basée sur le maintien de la cohérence des configurations qui est vérifiée à la construction.

Le *système modeller* apparait difficile à utiliser lorsque l'on configure à partir des révisions antérieures. En effet spécifier le *"system model"* à partir des noms de versions en donnant leur date de création n'est pas une manipulation aisée.

5.2.3. DSEE

DSEE [Leblang84] est un environnement de développement réparti, de niveau industriel, développé sur les ordinateurs Apollo. DSEE intégre de manière cohérente le contrôle de versions et la gestion de configurations. DSEE est composé de cinq outils appelés gestionnaires (*managers*).

- un gestionnaire de versions et d'historiques,
- un gestionnaire de configurations,
- un gestionnaire de tâches,
- un gestionnaire d'instructions,
- un gestionnaire d'événements.

1) Le gestionnaire de versions et historiques.

Il contrôle le code source et maintient les versions d'historiques comme dans RCS, SCCS.

2) Le gestionnaire de configurations

Le configurateur utilise deux structures : le *system model* qui décrit les composants d'un logiciel et leurs dépendances et le *configuration thread* qui spécifie les contraintes de composition à appliquer pour la construction d'une configuration.

Une fois la configuration construite, un historique est associé automatiquement à chaque exécutable et décrit le processus de génération. Le gestionnaire de configuration gère un ensemble

de codes binaires associés aux composants pour accélérer la génération et utilise un recupérateur d'espace qui détruit automatiquement les binaires non utilisées pendant une période donnée.

3) Le gestionnaire de tâches.

Il permet de décrire des tâches pour réaliser une activité de développement ou de maintenance de logiciel. Ces tâches peuvent être de tout ordre: programmation, mise à jour de la documentation, sauvegarde d'un système etc... Les tâches sont enregistrées et affichées par le gestionnaire d'événements sur le poste de l'usager qui est responsable de leur réalisation. Une fois réalisées, ces tâches sont archivées.

4) Le gestionnaire d'instructions.

Il enregistre les instructions utilisées pour décrire un modèle de tâche particulière comme par exemple ajouter une nouvelle option à un programme. Ces instructions sont disponibles aux autres développeurs dans le cas où ils voudraient traiter une tâche identique.

5) Le gestionnaire d'événements

Le gestionnaire d'événements permet de définir des relations et de déclencher des alarmes lorsqu'un des objets de la relation est modifié. Ces alarmes se déclenchent pour informer d'une incohérence potentielle. Des services de différentes natures sont offerts comme:

- indiquer les dangers d'une modification;
- informer d'une modification à introduire dans un autre module;
- indiquer les tâches à réaliser après le changement. Ces tâches sont ajoutées à la liste de tâches du développeur concerné et apparaissent sur son écran;
- lancer une séquence de commandes comme par exemple la reconstruction d'une configuration.

Pour ce faire, les dépendances suivantes sont maintenues:

- programme-document,
- module commun à deux systèmes,
- entre deux modules pour exprimer par exemple qu' un module demande un format d'entrée particulier d'un autre module.

DSEE a des fonctions puissantes pour composer des configurations à partir du *system model* et du *configuration thread* et pour gérer les objets dérivés. Il a été conçu pour supporter les logiciels développés avec les langages classiques. Comme le concept de module n'est pas supporté et pour ne pas lancer une cascade de recompilations dés qu'un source est modifié, deux mécanismes ont été introduits dans DSEE pour limiter la propagation des changements.

- une dépendance entre deux unités de compilation peut être déclarée non critique. Ainsi un changement d'une unité ne cause pas la recompilation de celle qui en dépend.

- un usager peut déclarer qu'une version V1 est compatible (ascendante) avec une autre V2. DSEE en déduit que les codes objets de V2 peuvent être utilisés à la place de ceux de V1.

5.2.4. Discussion

Les systèmes de programmation globale Gandalf et Cedar ont permis de valider certains concepts de la programmation globale. Parmi ces concepts, l'utilisation de la programmation structurée, les idées de [Parnas72] pour la construction de systèmes modulaires, la séparation entre la partie interface et réalisation d'un module et plusieurs idées reprises à partir d'outils Unix (Make pour la génération, SCCS pour la contrôle de versions source, RCS pour les développements parallèles) ont pu être testés et validés, le tout intégré dans un environnement.

Par contre au niveau du contrôle et de la base de données, les mécanismes sont restés très élémentaires et très rigides. Des hypothèses restreignent le champ d'application: mono-langage pour CEDAR, dépendant d'une méthode pour GANDALF. L'ensemble des attributs décrivant les objets étant limité (attributs systèmes), les contraintes de composition dans la constitution de configuration sont restées très rudimentaires (choix de la dernière révision dans le cas de CEDAR et de la version par défaut dans GANDALF). Ceci ne facilite pas la réutilisation de versions lorsque les systèmes évoluent.

D'une manière générale, ces systèmes ont été développés avec une vue particulière de l'environnement qui ne permet pas de les généraliser aisément.

DSEE est un environnement industriel de programmation globale distribué sur un réseau local, testé en vrai grandeur. Le système est basé sur une forte cohérence entre le contrôle de versions et la gestion de configurations. En introduisant les dépendances usagers, DSEE a montré d'autres aspects de la programmation globale. C'est un système très performant car implanté directement dans le système d'exploitation; par exemple le mécanisme de delta est implanté directement dans le SGF. Par contre, ces caractéristiques le rendent dépendant des systèmes et des machines. Comme il a été développé pour les langages classiques du type C plutôt que les langages modulaires, il connait tous les problèmes de Make. Des mécanismes "ad-hoc" ont été développés pour limiter par exemple les recompilations en cascade.

Adèle est dans la lignée des systèmes comme Cedar, Gandalf et DSEE et il n'est pas étonnant de retrouver les concepts et limitations de ces systèmes. Nous en listons les principales caractéristiques:

- comme Gandalf et Cedar, Adèle permet le développement de systèmes modulaires.

- comme dans Cedar, un module dans Adèle est constitué de deux parties distinctes: une interface et une réalisation. La famille, notion de module étendue aux versions est équivalente au module de Gandalf avec en plus les versions et vues d'interface.

- Comme dans DSEE, Adèle supporte des contraintes de composition de versions mais beaucoup plus élaborées. L'expression de la composition des configurations s'exprime par des contraintes sur les attributs plus riches qualifiant les versions.

- la description des composants d'une configuration est issue du graphe de dépendance et n'est pas prédéfinie comme dans Cedar, Gandalf et DSEE.

- Adèle est multi-langage et par conséquent ne permet pas la vérification des types inter-modules.

Adèle est actuellement au stade pré-industriel. Nous résumons dans ce qui suit ses principales limitations qui nous ont amenés à le faire évoluer:

- la structure du logiciel représentée par le graphe de dépendance est figée et ne peut être modifiée. Le traitement des changements évolutions nécessitent des mécanismes de structuration et de contrôle de l'évolution de la structure du logiciel,
- la méthode de développement est nécessairement descendante,
- le mécanisme de droits d'accés est rudimentaire et n'est pas adapté à une équipe de développement,
- Adèle ne permet pas l'activation d'outils, par exemple pour créer des objets dérivés.

5.3. Tendances actuelles

Les investigations en cours dans le développement de systèmes de programmation globale s'orientent vers l'utilisation de la technologie récente des bases de donnée et l'expression de contraintes de gestion de logiciel sous forme de règles associées à des mécanismes d'activation d'outils.

- L'utilisation de la technologie récente des bases de données est illustrée par la présentation du système PCTE-SGO qui a atteint actuellement un niveau de développement avancé.

- Les travaux sur l'activation d'outils sont illustrés par la présentation de deux systèmes : le système ODIN[Clemm86] [Clemm88] basé sur la gestion d'un graphe de dérivation d'objets.

Le système Marvel [Kaiser87b], basé sur la connaissance et développé comme extension d'un systéme fermé: Smile[Habermann86] [Kaiser87a].

5.3.1. PCTE-OMS.

PCTE est un projet ambitieux développé dans le cadre du programme européen Esprit dont le but est de fournir une structure d'accueil pour le développement d'ateliers de génie logiciel. PCTE est basé sur une architecture distribuée réalisée sur un réseau local de type Ethernet et est développé dans l'environnement Unix. PCTE couvre plusieurs aspects :

- des mécanismes d'exécution, de communication (entrées-sorties et inter-processus) et de gestion d'activités concurrentes;
- la distribution sur un réseau local;
- une interface utilisateur ;
- un gestionnaire d'objets.

Nous présentons plus particulièrement dans ce qui suit le gestionnaire d'objets.

Le système de gestion d'objets (SGO) de PCTE permet de gérer une base de données, d'un environnement de génie logiciel, distribuée de manière transparente sur un réseau local.
La conception du SGO est orientée vers la programmation globale.

1) Modèle de données PCTE

Le modèle PCTE est inspiré du modèle entité-relation et intègre des caractéristiques "orientées objets" comme l'héritage et l'organisation en classes. Les concepts du modèle bien que connus, appellent quelques précisions:

- les types d'objets.

Les objets dans la base sons typés. Un type est défini par un nom, des attributs, un ensemble de types de liens dont il est l'origine et un type ancêtre puisque les types d'objets sont organisés en une hiérarchie de sous-types. Le nouveau type hérite des caractéristiques de l'ancien. Un type origine appelé "objet" est la racine de la hiérarchie et est défini avec un ensemble d'attributs prédéfinis. Il y a d'autres sous-types prédéfinis du type "objet" comme par exemple fichier, message... Ces types se caractérisent par un attribut "contenu" dont les occurences ne sont pas interprétées par le gestionnaire de la base.

- les types de liens entre objets.
Un type de lien est spécifié par un nom, des attributs, les types d'objets origine et destination et la cardinalité définissant combien de liens peuvent être créés à partir de l'objet origine. Une séquence d'occurence de liens définit un chemin d'accès dans la base.

- les types d'associations.
Les types d' associations sont définis par des couples de types de liens entre deux objets, l'un étant inverse de l'autre;

- les types d'attributs.
Ils définissent les objets et les liens. Les domaines d'attributs sont des types simples : booléen, entier, chaine de caractères et date.

2) La gestion des schémas.

L'ensemble des types définis sont groupés dans des "SDS" (Schema Definition Sets) qui constituent le schéma de description de données supporté par le gestionnaire de la base.
PCTE-SGO supporte un mécanisme de vue par la notion de schéma de travail (SDS). Le schéma de travail définit les types visibles spécifiques à un usager ou à un outil et permet de contrôler les occurences créées. Des définitions communes de types peuvent être importées d'une SDS vers d'autres SDS.
Des modifications de schémas, contrôlées par des droits d'accés peuvent avoir lieu dynamiquement par application d'opérations d'ajout, de modification et de recopie de définition de types.

Exemple de schéma de travail
Description de la structure hiérachique du système de fichiers d'Unix

---

La base d'objets est vue comme un ensemble de fichiers Unix organisés en arborescence d'annuaires. Les éléments du schéma de travail sont:
- les deux types d'objets **annuaire** et **fichier**;
- le type de l'association de l'objet **annuaire** vers lui-même, avec un type de lien de **sous-annuaire** vers l'**annuaire père** et, un type de lien inverse de **annuaire** vers **sous-annuaire**;
- un type de lien **annuaire de fichier** du type d'objet **annuaire** vers le type d'objet **fichier**.

PCTE-SGO présente des caractéristiques adaptées au développement d'un environnement de programmation globale comme par exemple les schémas de travail permettant d'intégrer des outils externes, la granularité des objets supportés et la gestion des contenus. Cependant quelques mécanismes sont basiques et restent très liés au système Unix comme par exemple la désignation des objets et les droits d'accés. Par ailleurs PCTE-SGO ne supporte pas l'activation.

Notons que PCTE a atteint actuellement un niveau industriel et sert de structure d'accueil à plusieurs projets de développements d'environnements dont PACT et ECLIPSE [Cartmell87a] [Cartmell87b].

5.3.2. ODIN

L'outil Odin a été conçu pour la génération d'objets dérivés. Le processus de dérivation d'objets n'est pas prédéfini mais est décrit par des règles qui sont compilées et représentées par un graphe de dérivation. Dès qu'un objet dérivé est nécessaire ou est rendu périmé, Odin le regénère par l'application des outils nécessaires qu'il détermine par le graphe de dérivation. La dérivation est étendue aux objets composés comme les configurations pour lesquels ODIN construit automatiquement le processus de génération et active les outils pour produire par exemple l'exécutable.

Odin comporte :

- un langage de spécification de règles de dérivation.

Exemple de spécification de règle de dérivation en Odin [Clemm88]

```
fmt "version formattée du code C"
  USER        pol_C.cmd
                      : C
```

pol_C transforme un fichier de type "C" en un fichier de type "fmt",
"version formattée du code C"
décrit l'objet obtenu
pol_C.cmd est le nom de l'outil formatteur,
C est le type d'objet, entrée du formatteur.

- Un langage de commande "orienté objet" pour la manipulation d'objets

Exemple :

obj.c : run pour compiler et exécuter le fichier obj.c
obj.c:fmt pour formater le fichier obj.c

Odin est utilisé dans le projet *Toolpack* [Osterweil83] comme mécanisme d'intégration d'outils.

Il est intéressant de noter qu'Odin est un outil plus puissant que Make puisqu'il permet d'appliquer automatiquement le mécanisme de dérivation dans le cas d'objets composés alors que

pour Make il faut le programmer explicitement. Cependant Odin comme Make ne supportent pas la gestion de versions multiples.

5.3.3. SMILE et MARVEL

Smile est un outil de support à la programmation globale qui a servi au développement du projet Gandalf. L'objectif de ce système est d'assister la reconstruction de logiciels. Smile gère les interactions entre les objets et les outils et contrôle l'activation d'outils. Il est basé sur un modèle prédéfini où tout est figé aussi bien les modules que les différentes procédures à appliquer pour la construction de logiciels.

Marvel, le successseur de Smile, est un système de type expert où la connaissance du processus de construction est rendue explicite par la définition de règles.
Les concepts à la base du système Marvel sont:

- La base d'objets.
Les objets sont de différents types: variables, modules, documents, usagers... Ils sont structurés en classes. Chacun des objets est décrit par des attributs. Marvel supporte l'héritage multiple; ce qui fait qu'un objet peut appartenir à plusieurs classes et par conséquent hérite des attributs des différentes classes. La structure d'un logiciel est décrite par des attributs qui définissent les relations entre objets.

- Les actions.
Une action est caractérisée par des pré et post-conditions et décrit les appels d'outils et les conditions de leur activation. Les pré-conditions doivent être complétement satisfaites avant le déclenchement des outils. Les post-conditions ne sont pas des conditions à vérifier mais des assertions, introduites une fois l'activité exécutée. Les actions sont définies sous forme de règles qui présentent une stratégie de développement de logiciel.

- L'interprétation des règles.
Marvel a deux types d'interprétation des règles selon que l'exécution d'une séquence d'actions soit faite en chaînage avant ou en chaînage arrière.

  - chaînage avant : lorsque les préconditions d'une action sont satisfaites, Marvel l'exécute automatiquement.

Exemple 1 :

Soit la règle de compilation suivante :

```
précondition:    not compiled(module);
activité:        {compile module}
postcondition:   compiled(module);
```

Soit à compiler un module M, si le module M a été déjà compilé, la condition **compiled(M)** est vraie et la compilation ne peut être réexécutée.
Soit une deuxième règle d'édition de programmes dont la post-condition (assertion) valide la pré-condition de la règle de compilation après l' édition du module. Marvel se met en état d'interprétation en chainage avant et compile le module édité.

règle d'édition :

```
précondition:    reserved(module);
activité:        {edit(module)}
postcondition:   not compiled(module);
```

- chaînage arrière : lorsque l'usager active un outil avec des préconditions insatisfaites, Marvel essaye d'exécuter une action dont les post-conditions rendent à "vrai" les pré-conditions de l'activité demandée par l'usager.

Exemple 2 :
Soit la règle suivante de construction de l'exécutable d'une configuration :

```
précondition:    not linked (conf) and for all modules m such
                 that in (conf, module m): compiled (module m)
activité:        {link conf}
post-condition:  linked(conf);
```

Dans l'exemple précédent après une demande de construction de la configuration, si la précondition n'est pas vérifiée, Marvel lance les compilations de modules dont la post-condition satisfait la pré-condition de construction de configuration.

5.3.4. Discussion

PCTE-SGO, ODIN et MARVEL annoncent les tendances actuelles. Le point fort de ces outils et systèmes est de s'appuyer sur des modèles de description des données et des traitements bien établis et s'appuyant sur des concepts simples. Il reste à évaluer les aspects liés à l'efficacité.

- PCTE-SGO est représentatif des investigations actuelles dans la conception d'une base d'objets pour les environnements. Ces développements utilisent la technologie des bases de données notamment les modèles sémantiques.

- Odin est un outil adaptable pour la gestion des objets dérivés mais ne gère pas les versions.

- Marvel est un système basé sur la connaissance. Le systéme rend explicite les règles de reconstruction de logiciels et les traitent par un outil d'activation de type système expert qui supporte une forme de raisonnement. Le système permet à l'usager de structurer son activité de développement en utilisant ses propres méthodes et outils. La base d'objets et le système de conurôle de l'activation sont utilisés comme support pour l'intégration d'outils.

D'autres tendances, que nous n'avons pas traitées dans ce document, concernent les études de développement de systèmes génériques avec une plus grande couverture du cycle de vie [Lamsweerde 88].

6. CONCLUSION

Les développements récents des systèmes de programmation globale amènent les observations et commentaires suivants sur les directions de recherche suivies actuellement :

- Généralité et ouverture. L'indépendance par rapport aux langages, systèmes d'exploitation, outils et méthodes, et l'adaptation aux besoins des usagers et aux outils extérieurs à l'environnement, sont recherchées dans la plupart des projets. Les environnements se développent à partir de noyaux offrant des services puissants de support à la programmation globale comme la gestion de versions, la gestion de configurations et la gestion de la coopération entre programmeurs.

- L'orientation objet. L'utilisation de modèles sémantiques pour définir différents types d'objets logiciels avec leurs opérations et l'organisation de ces objets en classes d'héritage est en exploration dans de nombreux projets.

- L'assistance à la maintenance pour gérer les incohérences dans un environnement de développement logiciels de taille importante commence à être envisagée. [Schwankle88a] montre que les incohérences sont inévitables et que des outils sont nécessaires pour les analyser et les gérer. C'est un domaine ouvert et dans les premiers travaux, l'assistance à la maintenance globlale est perçue via l'activation de la base. L'activation permet de développer des mécanismes de contrôle élaborés sous forme de déclenchement d'actions.

- Des mécanismes de structuration pour le contrôle de l'évolution de la structure sont envisagés. La structure du logiciel est développée au niveau de l'étape de conception. Les logiciels de taille importante du fait de leur longue durée de vie, se déstructurent suite aux changements effectués. Des mécanismes sont nécessaires pour contrôler les changements qui affectent la structure du logiciel durant le développement et la maintenance..

- Une autre caractéristique de l'évolution actuelle des noyaux de programmation globale est liée à celle de l'architecture de support. Actuellement les environnements de développement tâchent de suivre l'évolution du matériel en développant des architectures basées sur un réseau local avec des stations de travail connectées à des machines serveurs puissantes. Les stations de travail contiennent les espaces de travail et les bases publiques sont réparties sur un ou plusieurs serveurs. Les effets de ces changements sont à l'étude dans la plupart des projets actuels. Après le développement d'environnements distribués (Cedar, DSEE) permettant de répartir les données sur plusieurs serveurs de fichiers, les travaux actuels tirent profit de la puissance de calcul disponible sur un réseau pour lancer des traitements parallèles comme les constructions de configurations [Leblang88]. L'hétérogénéité rencontrée dans les réseaux rend ces applications difficiles.

CHAPITRE 3 :

UNE PREMIERE EXPERIENCE AVEC ADELE: UNE BASE DE GESTION DE LOGICIELS DE GRANDE TAILLE

1. INTRODUCTION

A l'origine, Adèle était un projet développé au Laboratoire de Génie Informatique de Grenoble avec comme objectif la conception et le développement d'un atelier de programmation pour le langage Pascal [Estublier83]. Ce projet avait pour thèmes :

- le développement d'outils de programmation bâtis autour de l'édition syntaxique comprenant un interpréteur et un générateur de code multicibles;

- la définition d'une interface usager de haut niveau;

- le développement d'une base de programmes .

En 1984, après l'achèvement du projet Adèle, nous avons repris le travail sur les bases de programmes et poursuivis les développements en conservant le nom d'Adèle pour cette partie. Depuis, le système Adèle a évolué comme base de programmes de gestion de logiciels industriels de grande taille puis comme noyau d'un environnement de programmation globale [Belkhatir86b].

Nous décrivons dans ce chapitre l'évolution d'Adèle comme base de programmes pour la gestion de logiciels de grande taille. Nous avons repris le prototype de 1983 développé sous Multics et introduisit des extensions et des modifications pour aboutir à un produit exploitable industriellement, développé sur machines Unix.

2. OBJECTIFS

La base Adèle gère des programmes d'où la dénomination de base de programmes. Adèle se place dans le cadre de la production de logiciels de taille suffisamment importante pour nécessiter une composition modulaire. La base Adèle permet la description des modules développés et la

manière de les combiner pour produire une configuration. Adèle est essentiellement un support de coordination des développements et de maintenance de logiciels modulaires entrepris par une équipe de programmeurs.

L'architecture d'Adèle a été développée pour supporter deux fonctions essentielles dans un contexte multi-langages:

- une fonction de contrôle de versions qui aide à stocker, à accéder, à protéger, à manipuler les composants du logiciel et à gérer leur évolution.

- une fonction de gestion de configurations pour automatiser la construction de systèmes complexes.

3. NOTIONS FONDAMENTALES

Cett partie introduit la structure et les notations utilisées pour décrire un logiciel géré par la base Adèle.

3.1. Module.

Adèle utilise la notion de module telle qu'elle est définie dans les langages de programmation comme Ada, Modula2 et Mesa. Un module est composé d'une interface et d'une réalisation.

- une **interface** spécifie l'ensemble des ressources fournies par le module: déclarations de constantes, types, variables et procédures;

- une **réalisation** réalise les ressources de l'interface; c'est un texte source compilable (appelé corps dans notre terminologie) qui fournit à lui seul toutes les ressources ou qui utilise des ressources externes, décrites dans d'autres interfaces et fournies par d'autres réalisations.

Le concept de module est étendu dans Adèle par des caractéristiques propres aux environnements de programmation. Nous reviendrons sur cet aspect dans (§3.4).

3.2. Configurations

La réalisation d'une interface est un texte de programme qui généralement utilise des ressources décrites dans les interfaces d'autres modules. Par conséquent la réalisation de l'interface de ce module nécessite un ensemble d'interfaces et de corps.

On appelle **configuration** l'ensemble des interfaces et des réalisations nécessaires pour fournir réellement les ressources d'une interface.

Par définition, une configuration réalise les ressources de son interface, donc c'est une de ses réalisation. La réalisation signifie dans ce cas le corps de la configuration. Adèle comme Gandalf uniformise conceptuellement corps de programme et configuration. Les deux sont des réalisations dans Adèle, distingués par les sous-types corps et configuration :

- le corps est défini par le texte du programme associé à l'interface;

- la configuration est définie par la liste des composants (interfaces et réalisations) nécessaires à l'interface. C'est cette liste de composition qui définit la configuration.

3.3. Les versions

Pour gérer l'évolution d'un logiciel, Adèle supporte la notion de versions et l'applique aussi bien aux interfaces qu'aux réalisations.

3.3.1. Versions de réalisation

Les notions d'alternatives et de révisions sont introduites pour prendre en compte les processus d'évolution d'un logiciel tout au long de son cycle de production et de maintenance. Les définitions suivantes sont appliquées:

- les notions d'alternatives et de révisions sont regroupées sous le concept de versions. Une alternative correspond à des changements significatifs de réalisation qui ont pour origine:
  - des langages ou des systèmes différents;
  - des changements d'algorithme, de structure,
  - des caractéristiques matérielles différentes, etc...

Une alternative évolue en une suite de révisions séquentielles. Une révision correspond à des corrections d'erreurs, des améliorations internes du code source... La désignation d'une révision est maintenue automatiquement par le système qui attribue une numérotation séquentielle.

Une réalisation d'un module est définie par un couple (alternative, révision).

Exemple de désignation :

Deux alternatives V1 et V2 de la même interface I1 du module M sont désignées par M-I1-V1 et M-I1-V2. L'alternative V1 évolue en plusieurs révisions comme par exemple M-I1-V1.01, M-I1-V1.02, M-I1-V1.03 qui sont les 3 premières révisions.

Ce schéma d'évolution en alternatives et révisions est inspiré de Gandalf. Il a été étendu dans Adèle par le concept de version et de vue d'interface.

3.3.2. Versions d'interface

Adèle distingue deux types d'évolution pour les interfaces en introduisant les notions de vue et d'alternative d'interface. Les notions d'alternative et de vue sont regroupées sous le concept de version d'interface. La version d'interface tient compte du fait qu'une interface change durant la vie d'un projet.

- une alternative permet de faire évoluer un module sans en créer un autre. Elle possède ses propres versions de réalisations.

- la vue d'interface. Une alternative d'interface est constituée d'un ensemble de vues; chacune définissant un sous-ensemble de fonctions. Les vues d'interface peuvent différer pour des problèmes de syntaxe de langage ou de protection. Dans Adèle on distingue deux types de vues:

  - une vue "protection". Les ressources d'une interface sont partitionnées pour des raisons de protection. Par exemple, on peut organiser les interfaces en une vue publique qui fournit les ressources accessibles à l'ensemble des modules et des vues privées qui protègent les ressources du module en ne présentant qu'un sous-ensemble accessible à une classe particulière de modules.

  - une vue "syntaxique". Des raisons syntaxiques obligent à décrire les ressources d' une même interface dans plusieurs langages différents. Ceci permet à des modules écrits dans différents langages d'accéder à cette interface, c'est à dire de pouvoir être compilés en incluant cette interface syntaxiquement.

cas particulier: **interface sans corps**.

Une interface peut être une ressource en elle-même: définition de types, de constantes, de macros, de variables globales... Adèle permet la gestion de ces types d'interfaces qui sont sans réalisation.

3.4. Notion de famille

La notion de famille étend la notion classique de module définie dans les langages de programmation.

Les versions d'interfaces et de réalisations qui leur sont associées sont assemblées dans une unité commune appelée **famille**. Ces versions ont en commun un schéma de définition de propriétés structurelles et contextuelles (c'est à dire liées à l'environnement d'utilisation). La figure 3.1 présente la structure étendue de module (Famille), implantée dans Adèle.

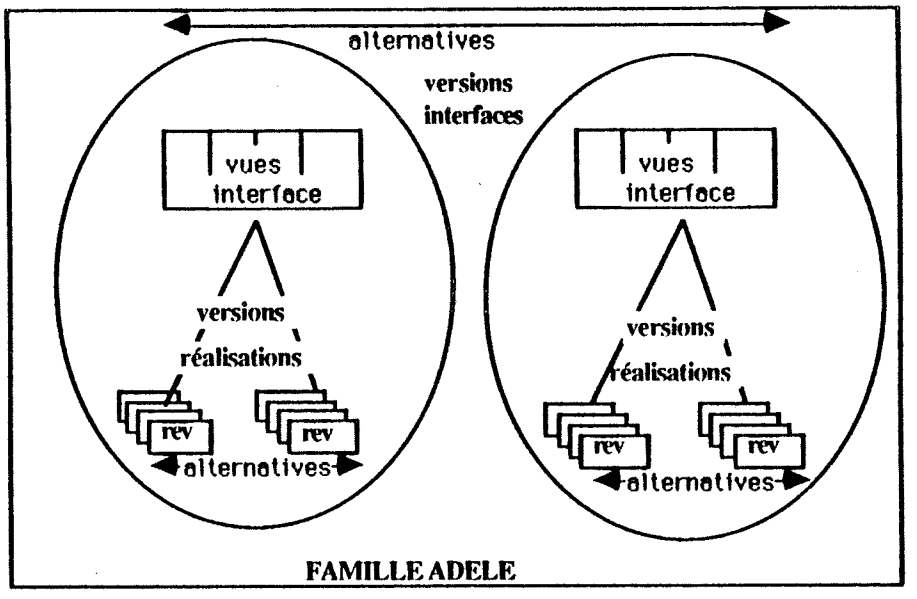


Figure 3.1 : Schéma de module dans Adèle

3.5. Objets atomiques

Les familles, les interfaces et les réalisations sont des objets composés d'objets atomiques. On considère dans Adèle quatre types d'objets atomiques : les révisions, les historiques, les codes objet et les manuels. Un objet atomique "élément" d'un objet composé "Objet" est désigné par Objet.élément.
Nous décrivons dans ce qui suit les différents types d'objets atomiques.

- **Les révisions.**

Ce sont les objets contenant le code source. Les mécanismes de gestion du stockage et d'identification appliqués sont dérivés de RCS. Le mécanisme de delta permet de sauvegarder uniquement les différences entre révisions. L'identification de chaque révision est faite en insérant automatiquement dans le texte une entête identifiant l'auteur, la date et son nom lors du stockage dans la base. La numérotation séquentielle des révisions est maintenue automatiquement. Les révisions sont appliquées aux objets réalisations.

- **Les historiques**

Ils sont créés automatiquement et permettent de garder la trace toutes les actions faites sur l'objet auquel ils sont associés. Toute opération d'ajout ou de modification d'objets dans la base est documentée sous forme d'un commentaire, inséré par l'usager à la fin de toute opération. Les commentaires sont gérés séquentiellement suivant la date et sont accompagnés d'informations insérées automatiquement comme l'auteur, la date de modification et éventuellement l'origine de l'objet.

- **Les codes objets**

Les codes objets peuvent être stockés dans la base et sont par conséquent associés aux révisions. Un récupérateur d'espace ne permet de stocker dans Adèle que les codes objets actifs, c'est à dire utilisés dans les configurations ou associés à la dernière révision. Les codes objets ont une désignation spécifique. C'est un identificateur comme pour les objets atomiques suivi du numéro de la révision d'où ils ont été dérivés.

Exemple :

F1-I1-V1.M68000.02 qui est le code objet M68000 de la 2ème révision du module F1-I1-V1.

- **Les manuels**

Ce sont les descripteurs des objets qui maintiennent toutes les informations sur les objets composés (famille, interface et réalisation). A chaque objet composé correspond un manuel. Les manuels sont créés automatiquement et sont constitués de deux parties: une partie gérée par la base et l'autre par l'usager.

- la partie gérée par la base concerne les informations internes comme par exemple l'état d'un objet, les dépendances avec d'autres objets, ses révisions dans le cas de réalisations. Cette partie peut être accédée en consultation par l'usager.

- la partie gérée par l'usager concerne la description des objets par des attributs, les contraintes de sélection (voir §4.2.2) à appliquer lors de la construction de configurations et la protection de visibilité entre modules(voir §5.1)

Exemple :

désignation des objets atomiques d'une alternative de réalisation F1-I1-V1

- révisions F1-I1-V1.01...F1-I1-V1.nn
- manuel F1-I1-V1.man
- historique F1-I1-V1.hist
- code objet F1-I1-V1.M6809.01, F1-I1-V1.M68000.02

3.6. La structure modulaire et la relation de dépendance

Les vues d'interface et les réalisations dans Adèle utilisent des ressources d'autres vues d'interfaces et par conséquent en dépendent. Dans Adèle cette relation de dépendance est extrapolée au niveau famille. Ainsi si une famille F1 a une de ses vues d'interfaces ou une de ses réalisations qui dépend d'une vue d'interface au moins de la famille F2, alors on considère que la

famille F1 **dépend de** la famille F2 comme le montre la figure 3.2. La relation de dépendance d'un système modulaire définit un graphe où les noeuds sont les familles et les arcs les relations de dépendance. La relation de dépendance est gérée par la base et est soit fournie manuellement par les usagers, soit calculée automatiquement par analyse du source des vues d'interfaces et des réalisations pour les langages la signifiant explicitement dans le source (à partir des *include*, *use*...).

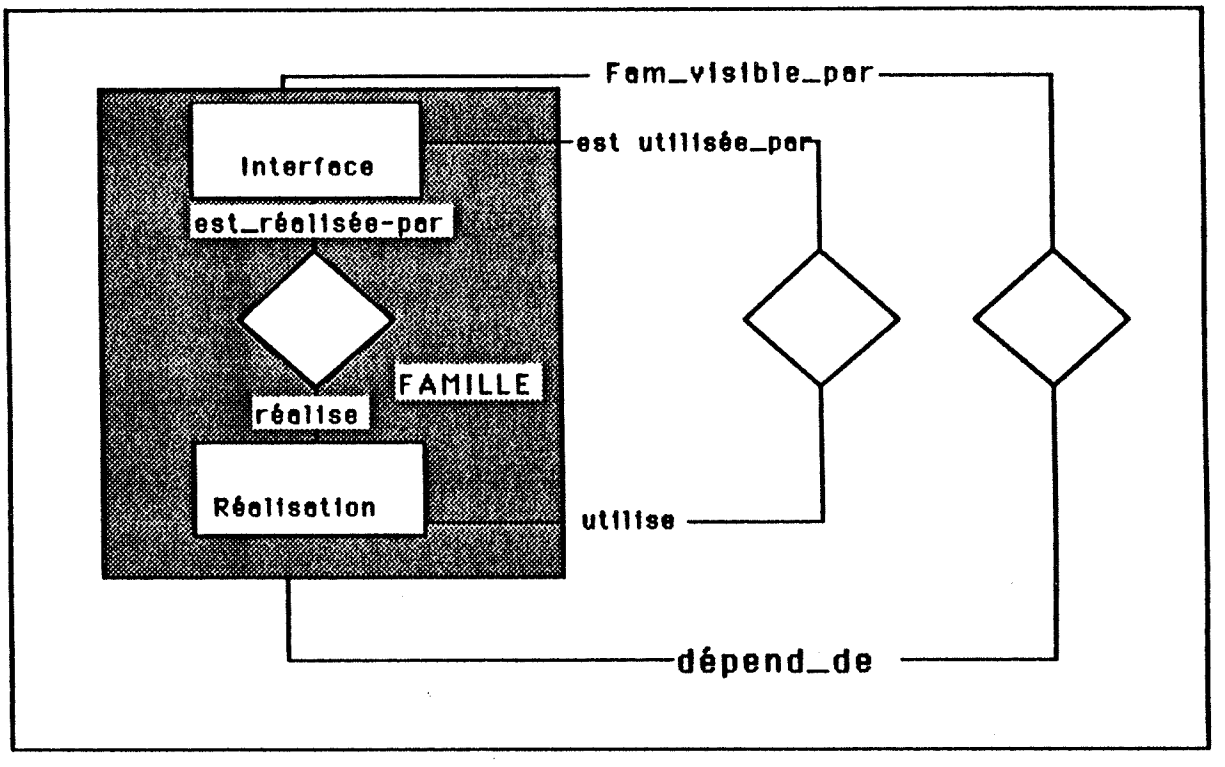


Figure 3.2 : Relation de dépendance dans Adèle

Le graphe de dépendance géré par Adèle présente les propriétés suivantes :

- Adèle décrit et analyse le logiciel et les configurations en terme de graphe de dépendance et impose que ce graphe soit sans circuit.

- L'utilisateur définit manuellement une structure arborescente du logiciel. Cette arborescence résulte de la réduction du graphe de dépendance. Le choix de la structure d'arbre est arbitraire. Cependant il faut noter que l'arborescence permet de définir:

  - une désignation unique des familles sous forme d'une expression de chemin, en partant de la racine. Les noms successifs de familles sont séparés par le caractère '>';

- un modèle de droit des usagers et de protection entre modules qui varie selon la parenté des modules . Cet aspect est décrit dans (§5.2).

- Les dépendances sont communes à toutes les révisions d'une même alternative de réalisation. Si les dépendances d'une réalisation changent, elles sont supposées changer pour toutes les révisions.

- Le graphe de dépendance permet de définir automatiquement une configuration. La définition d'une configuration est le processus de calcul de sa liste de composition. La liste de composition peut être obtenue automatiquement en partant de l'interface de la configuration à construire et en sélectionnant une version de module pour chaque famille traversée en parcourant le graphe de dépendance (fermeture transitive).

4. GESTION DE CONFIGURATIONS

4.1. Introduction

La gestion de configurations dans Adèle est axée principalement sur la fonction de définition de configuration, c'est à dire la description des versions et révisions qui la composent (liste de composition). Un embryon de fonction de génération (production des objets dérivés par exemple) est amorcée dans Adèle via l'outil Exec (voir §4.5)

Adèle supporte trois manières de définir une configuration:

- définition manuelle,
- définition automatique avec des régles de sélection implicites,
- définition automatique avec des règles de sélection explicites.

Nous examinons chacun des cas.

4.2. Définition manuelle.

C'est la solution la plus fréquemment utilisée pour décrire une configuration. Pour les configurations habituelles, l'usager fournit la liste complète des composants d'une configuration. Dans un environnement de programmation globale où un logiciel est composé d'une centaine de modules en plusieurs versions, la composition manuelle est difficilement envisageable. Même si cela est possible, la construction est une tâche hardue et génératrice d'erreurs.

4.3. Définition automatique

Dans un environnement de développement multi-versions, le graphe de dépendance ne définit pas une seule mais plusieurs configurations. Par la suite nous utilisons, pour simplifier, le terme configuration pour désigner une version de configuration.

Pour une configuration donnée, pour des raisons logiques (les modules sont partagés) comme pour des raisons techniques (non duplication des codes objets et des déclarations des noms externes), une seule version (une alternative et une révision) de réalisation d'une famille doit être utilisée. L'existence d'un groupe d'alternatives et de révisions, fonctionnellement équivalentes, nécessite des mécanismes permettant de faire les choix appropriés pour construire une configuration. On distingue deux manières d'effectuer la sélection d'une réalisation.

4.3.1. Règles de sélection implicite

C'est la manière utilisée dans la plupart des environnements de programmation globale. Pour chaque famille, la règle de sélection implicite consiste à choisir soit la dernière révision (Cedar), soit celle qui est réservée sinon celle qui a le statut standard (Gandalf). Dans Adèle, on peut créer

une configuration sans rien spécifier. Implicitement, par famille, l'alternative et la révision, par défaut, sont sélectionnées.

4.3.2. Règles de sélection explicite.

La définition manuelle est inappropriée dans un environnement de programmation globale manipulant une multitude de modules, chacun évoluant en plusieurs alternatives et révisions. D'un autre côté, la définition automatique avec une ou deux règles au plus s'appuyant sur la dimension temporelle, n'est pas assez précise. Or, il faut noter que dans la définition d'une configuration, les choix de versions ne sont pas aléatoires et sont connus par les concepteurs du logiciel.

L'approche originale d'Adèle consiste à spécifier une configuration (composition d'un logiciel multi-versions) à partir d'expressions de contraintes. A partir de l'évaluation de ces contraintes, il devient possible d'automatiser le processus de construction d'une configuration.

Etant donné une interface de module, Adèle parcourt dynamiquement le graphe de dépendance. Il utilise la spécification de configuration pour sélectionner les versions qui la composent. Durant le processus d'analyse, Adèle vérifie que les contraintes des versions sélectionnées sont satisfaites par toutes les versions dépendantes.

Exemple de définition de contraintes :

- choisir les versions écrites en langage Pascal et à l'état test;
- choisir pour la famille F1 des versions écrites uniquement en langage C;
- construire une configuration pour le système Unix_V5...

Nous remarquons que ces règles s'appuient sur des critères de production du logiciel, plus précisemment sur les différentes conditions de réalisation des versions comme le langage, le système, l'état...

La notion d'**attribut** est introduite dans Adèle pour permettre l'expression de ces contraintes de sélection.

4.3.3. Notion d'attribut

Un attribut est une propriété qualifiant un objet. Il est défini par un nom et un ensemble de valeurs.

Certains noms d'attributs sont dérivés du système. Ils sont liés directement au cycle de production comme la date, l'auteur et l'état. Ces attributs sont communs aux versions de tous les logiciels développés dans Adèle.

Exemple :

Le manuel de F1-I1-V1 contient les attributs suivants pour qualifier l'alternative V1.

```
Manuel F1-I1-V1
    attributs
        /* attributs définis par l'usager */
        langage=      C
        système=      unix
        /* attributs définis par Adèle */
        état=         expérimental
        auteur=       x
        date=         01-01-88
```

4.3.4. Expression des contraintes de sélection

Une configuration peut être instanciée automatiquement en fonction des attributs des versions qui constituent les paramètres de configuration. Selon les valeurs des paramètres spécifiées par l'usager et les règles de cohérence fixées, on obtient une version de configuration.

Adèle distingue trois types de règles.

1)Règle de sélection impérative

Soit l'exemple d'une configuration à construire pour la famille F1.

```
configuration F1-I1-Conf
 selimp
            F2-I3-V2.04                      /*contrainte 1 */
            >* (type= test)                  /*contrainte 2 */
            or (   >F1>* (syst= unix_4.3)    /*contrainte 3 */
                   >F1>* (syst= unix_5 ))
 selnot
            >* (état = incohérent)           /*contrainte 4 */
fin
```

Les règles de sélection impérative s'expriment par les deux clauses suivantes :

- ***selimp*** pour décrire les objets à sélectionner impérativement,
- ***selnot*** pour décrire ceux qui ne doivent pas être sélectionnés.

Les versions de réalisations sélectionnées doivent satisfaire l'ensemble des contraintes exprimées par ces deux types de règles.

contrainte 1 : la version à sélectionner est désignée explicitement.

contrainte2 : toutes les versions de réalisations sélectionnées (>* pour désigner toute réalisation) doivent vérifier la contrainte type= test.

contrainte3 : toutes les versions de réalisations du sous-arbre F1 doivent vérifier la contrainte : syst = unix_4.3 ou syst = unix_5.

contrainte4 : les versions de réalisations dans l'état incohérent ne doivent pas être sélectionnées.

Le processus de construction d'une configuration est déterministe. Il conduit à trois situations:

- *L'incomplétude* :

C'est la situation où une ou plusieurs familles n'ont pas de versions de réalisations qui satisfont les contraintes impératives. La configuration est dans l'état **incomplet**..

- *L'incohérence* :

Si dans le processus de construction d'une configuration, une réalisation se trouve sélectionnée à la fois par les clauses *selimp* et *selnot*., la configuration est dans l'état **incohérent**..

- *cas normal* :

Le processus de construction se termine normalement. Tous les objets ont pu être sélectionnés. La configuration définie est supposée cohérente et prend l'état **expérimental**.

2) Règle de sélection conditionnelle

Soit la configuration F1-I1-Conf spécifiée de la manière suivante:

```
configuration F1-I1-Conf
        selimp
                F2-I3-V2.04
        selnot
                >* (état.= incohérent)
        selcond
                >* (type= test)
                or (    >F1>* (syst= unix_4.3)
                        >F1>* (syst= unix_5 ))
fin
```

La contrainte décrite par la clause de sélection conditionnelle ***selcond*** n'est appliquée à un famille que si au moins une des versions de réalisations est définie par les attributs spécifiés dans la contrainte. Ceci permet d'exprimer des contraintes conditionnelles. Ces contraintes ont un sens pour un sous-ensemble de familles (celles qui sont caractérisées par les attributs exprimés dans les contraintes) et aucun sens pour d'autres.

Dans l'exemple précédent, la contrainte conditionnelle:

```
(type= test et ( syst=unix_4.3 ou unix_5))
```

n'est appliquée que pour les familles ayant des réalisations avec les attributs *type* et *syst*.

3) Sélection par défaut

Les contraintes exprimées par la clause de sélection par défaut ***seldef*** sont appliquées après les deux autres types de contraintes. Ce type de contrainte est utilisé pour exprimer une préférence parmi plusieurs réalisations potentielles. Les contraintes par défaut ne conduisent pas à des situations de configuration incomplète et/ou incohérente.

4.3.5. Expression de contraintes décentralisées.

Jusqu'à présent, les contraintes de construction de configurations étaient supposées toutes décrites dans le manuel des configurations. Adèle apporte une plus grande souplesse. Les contraintes peuvent être décentralisées (locales) au niveau d'une famille, interface et réalisation. Elles s'expriment dans les manuels, de la même manière que pour les configurations et avec la même sémantique. Ceci permet à une famille particulière d'exprimer des choix de ressources logicielles particulières dont elle a besoin.

Le gestionnaire de configurations applique les propriétés suivantes pour l'application des contraintes décentralisées au moment de la construction :

- les contraintes exprimées au niveau d'une famille F sont ajoutées à l'ensemble des contraintes à satisfaire avant la sélection d'une version de réalisation de F;

- les contraintes exprimées au niveau d'une interface I sont ajoutées à l'ensemble des contraintes à satisfaire avant la sélection d'une version de réalisation de I;

- les contraintes exprimées au niveau d'une réalisation R sont ajoutées à l'ensemble des contraintes à satisfaire pour toutes les versions de réalisations dépendantes de R.

La décentralisation des contraintes a l'avantage de permettre une expression fine des contraintes jusqu'au niveau où elles sont applicables.

Exemple :

Expression de contraintes applicables aux versions qui dépendent des réalisations de F2-I1-V2.

```
manuel (de la réalisation) F2-I1-V2
        attributs
                etat=           expérimental
                syst=           unix
                langage=        C
                date=           01_01-88
        selimp
                F4-I1-V5.02
        selcond
                >* (syst=unix)
                >* ( langage = Yacc)
Fin
```

4.4. Exemple de description de logiciel

L'exemple suivant adapté de [Parnas72] illustre les concepts d'Adèle. L'exemple présente un système de production d'index: KWIC. En gros, le système accepte en entrée un ensemble de lignes constituées d'un ensemble de mots et fournit en sortie une liste de tous les décalages circulaires de mots de toutes les lignes triés par ordre alphabétique. 7 modules ont été identifiés dans ce système comme le montre la figure 3.3.

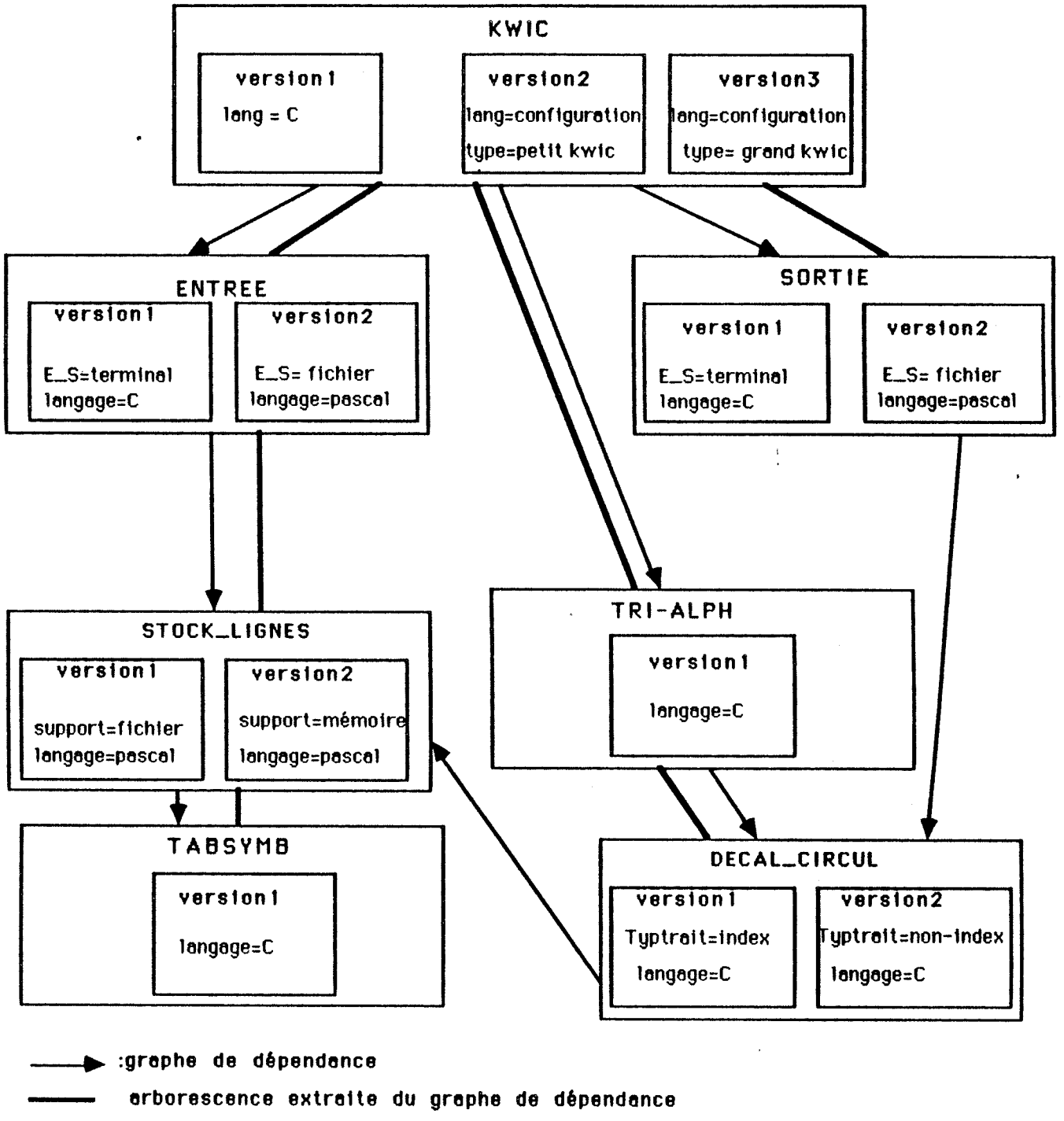


Figure 3.3 : Exemple du Kwic : représentation du graphe de dépendance

Le module Kwic est composé de 3 versions:

- version1: représente le programme principal du Kwic;

- version2 : représente une configuration du Kwic où tous les traitements se font en mémoire et les E/S via les terminaux. C'est le cas d'un Kwic avec peu de lignes à trier.

```
selimp
        stock_lignes (support=mémoire)
        decal_circul ( typtrait =non-index)
selcond
        >* E_S= terminal
```

Le processus de construction aboutit à la configuration suivante:

LISTE DE COMPOSITION =

```
entrée-v1,
sortie-v1,
stock_lignes-v2,
tri_alph-v1,
tabsymb-v1,
decal_circul-v2.
```

- version3 : représente une configuration du Kwic où tous les traitements ont besoin d'un support fichier et les E/S se font via les fichiers. C'est le cas d'un Kwic avec un volume de données importants à traiter.

```
selimp
        stock_lignes (support=fichier)
        decal_circul ( typtrait =index)
selcond
        >* (E_S= fichier)
```

Le processus de construction aboutit à la configuration suivante:

LISTE DE COMPOSITION =

```
entrée-v2,
sortie-v2,
stock_lignes-v1,
tri_alph-v1,
tabsymb-v1,
decal_circul-v1
```

4.5. Cas particuliers

Dans Adèle, le gestionnaire de configuration sélectionne une seule interface et réalisation par famille sauf dans les cas particuliers suivants:

4.5.1. Les vues d'interface

Lors du processus de configuration, plusieurs vues d'une alternative d'interface peuvent être sélectionnées pour une même réalisation.

4.5.2. les bibliothèques

Un famille de type bibliothèque signifie que les versions d'interfaces avec chacune de leurs versions de réalisations sont considérées comme compatibles dans une configuration. Une configuration peut contenir plusieurs versions d'interfaces chacune réalisant une fonction de la bibliothèque. Ce qui permet dans Adèle de regrouper les fonctions dans une même famille au lieu de définir une famille pour chacune.

Exemple :

On peut définit une famille bibliothèque regroupant les fonctions de services avec 3 versions d'interfaces plutôt que de créer trois familles. Par exemple

- 1 version d'interface définissant les fonctions de manipulation de pile;
- 1 version d'interface définissant les fonctions de manipulation de tableaux,
- 1 version d'interface définissant les fonctions de manipulations de liste.

4.6. Génération des configurations

La génération aborde la manière dont les opérations peuvent s'exécuter sur une configuration après la phase de définition. Les opérations les plus classiques sont la compilation et l'édition de liens, mais d'une manière générale, on peut appliquer d'autres opérations comme par exemple le transfert des sources sur un support externe, leur formattage ...

C'est le rôle de l'outil **Exec** d'Adèle qui à partir d'un nom de configuration et d'un bloc de commandes génériques, exécute des actions sur les composants de la configuration. L'outil Exec agit en deux étapes :

1) remise à niveau de la liste de composition d'une configuration. Cette fonctionnalité est équivalente au *"configuration thread"* de DSEE. Les manipulations permises sont :

- le remplacement des révisions d'une configuration par les dernières (option *new*);

- le remplacement de certaines révisions par celles actuellement en mise au point dans l'espace usager (option *res*). Il suffit pour ce faire, de réserver les révisions devant être substituées par celles en mise au point.

- les deux manipulations peuvent être combinées comme par exemple prendre les dernières révisions sauf celles qui sont réservées.

2) Une fois la remise à niveau effectuée, l'utilisateur peut appliquer en bloc une liste de commandes sur tous les composants de la configuration.

Exemple :

Soit la commande exec suivante:

```
exec M-I1-conf ,res ,new
     read F-I-C.o.R.
```

qui permet d'amener dans l'espace de travail de l'usager les codes objets de la configuration M-I1-conf non reservés, associés à la dernière révision. F, I, C, R sont des métavariables. Pour chaque composant de la configuration, F est substitué par le nom de famille, I par le nom d'interface, C par le nom de réalisation et R par le numéro de révision. C'est la manière utilisée dans Adèle pour effectuer une édition de liens en vue de tester une configuration.

5. COOPERATION ET PROTECTION

5.1. Relation de dépendance et visibilité entre modules

On entend le plus souvent par structure des logiciels celle définie par la relation de dépendance. Dans la plupart des approches, un graphe acyclique est proposé. Le contrôle de la relation de dépendance(structure du logiciel) se réduit au contrôle de la visibilité des interfaces des modules. Dans Adèle, chaque famille peut indiquer lesquelles de ses interfaces sont visibles, et les familles d'où elles le sont. Les vues d'interface permettent d'exprimer différents niveaux de visibilité. Chaque module client a accés à des vues d'interface en fonction de la protection souhaitée. Ceci montre l'intérêt d'avoir la notion de vue d'interface.
Par défaut, dans Adèle, une famille a une visibilité sur les ressources de ses familles filles. La visibilité définit la protection attachée à chacune des interfaces. La tentative d'accès d'une réalisation vers une interface non visible est refusée.
La visibilité s'exprime dans chaque famille par une liste d'accés indiquant pour chacune de ses interfaces, les familles pouvant l'accéder (clause "**utilisable_par**").

Exemple :
Dans la famille F1, on peut définir la visibilité suivante

```
manuel F1
        utilisable-par
                * ( -istandard)
                F2 (-init_reset)
```

Supposons que F1 soit une famille définissant une gestion de mémoire libre. L'exemple ci-dessus permet d'exprimer que l'interface init_reset définit les procédures init et reset, dont l'utilisation est réservée à la famille F2, que -istandard définit les procédures "malloc" et "free", dont l'utilisation est générale (*).

La clause "**utilisable-par**" est vérifiée à la création d'une relation de dépendance entre deux familles. Dans l'exemple précédent, pour que F2 puisse créer une dépendance sur une famille externe F1 , il faut au préalable que l'usager crée la relation visible entre F2 et F1. Puis la validité de la clause **utilisable-par** est vérifiée pour la famille réceptrice de la relation de dépendance. La relation visible vérifie surtout que des cycles ne sont pas introduits.

5.2. Droits d'accés

Adèle utilise le concept d'espace de travail pour définir les droits des usagers sur les familles. Le concept d'espace de travail s'appuie sur l'arborescence définie dans Adèle, extraite du graphe de dépendance. Un espace de travail est le sous-arbre défini par le nom de la famille racine. Un espace de travail est associé à un usager qui peut opérer sur l'ensemble des objets du sous-arbre. L'espace de travail permet de renommer et par conséquent d'alléger la désignation d'une famille en remplaçant un chemin d'accés par un identificateur. Par définition dans Adèle, tout usager créant une famille a un espace de travail défini sur cette famille. Implicitement c'est le nom de la famille qui est pris comme nom d'espace de travail. La pratique montre que seule la désignation par espace de travail et utilisée pour nommer les familles.

Exemple :

```
dclesp (x) >F1>F2    esp1
```

L'exemple montre la définition d'un espace de travail pour l'usager x, désigné par esp1. Ce qui permet à l'usager d'avoir accés au sous-arbre dont la racine est F2.

5.3. Modifications concurrentes

Les objets ne sont pas directement modifiés dans la base mais copiés dans un espace privé afin de pouvoir appliquer les outils classiques de traitement et qui ne sont pas contrôlés par la base. Ce problème est devenu classique et a été posé et bien résolu dans RCS.

Deux commandes **réserver** et **libérer** permettent le développement sans interférences d'actions concurrentes de manipulation de la base. Un objet dans Adèle ne peut être changé (commande catal) que par l'usager qui l'a réservé. Adèle verrouille l'objet réservé mais également les objets directement dépendants.

Exemple :

Dans le cas de la modification d'une réalisation, les interfaces utilisées et réalisées sont vérouillées; ce qui évite qu'une interface utilisée soit changée entre le moment où la réalisation est compilée et le moment où elle est replacée dans la base.

5.4. Les effets de bord des modifications

Adèle avertit l'utilisateur des effets potentiels des opérations de modification d'un objet sur tous les objets qui en dépendent. L'opération est défaite si elle n'est pas validée par l'usager.

Adèle gère chaque effet de bord en produisant des listes et en maintenant des attributs d'état pour informer les usagers des effets des modifications d'objets.

Trois listes sont maintenues:

- une liste des interfaces modifiées constituée de toutes les interfaces utilisées ou réalisées par une réalisation et qui ont été modifiées;

- une liste d'incomplétude: créée dans le cas d'une construction de configuration. Elle donne la liste des composants de la configuration qui n'ont pas de texte associé ou qui ont une liste de composition incomplète;

- une liste des périmés: créée dans le cas d'une configuration. Elle donne la liste des composants de la configuration qui ont été modifiés depuis sa création.

A partir des listes, la base déduit automatiquement la valeur de deux noms d'attributs: "état" et "étatconf" pour les configurations. Les règles de déduction des valeurs de ces états sont les suivantes:

- état = expérimental : liste des interfaces modifiées = nil;
- état = vide : objet créé mais non initialisé;
- état = interf-modifiée : liste des interfaces modifiées <> nil;

- étatconf = ok : liste d'incomplétude et des périmés = nil;
- étatconf = incomplet : liste d'incomplétude <> nil;
- étatconf = périmé : liste des périmés <> nil;
- étatconf = pér-incomp : liste des périmés et d'incomplétude <> nil;

6. EXPERIENCE AVEC LA BASE ADELE

Cette partie décrit l'expérience de transfert d'Adèle dans un environnement d'utilisation industriel utilisant des mini-ordinateurs. Cette expérience nous a amenés à revoir les fonctions principales tout en réalisant le portage. Le prototype initial développé sur Multics en Pascal et en PL1 a été largement modifié. Adèle en tant qu'outil de gestion de configuration a atteint actuellement un niveau de développement avancé favorable à son industrialisation. Nous résumons dans ce qui suit les principales améliorations et extensions apportées .

6.1. Améliorations et extensions apportées.

Les extensions et améliorations apportées au prototype ont porté sur des aspect fonctionnels et techniques pour rendre la base de programme réellement exploitable.

6.1.1. Aspects fonctionnels

Les caractéristiques fonctionnelles modifiées sont une gestion plus complète des interfaces de modules et l'ajout de fonctions d'interface usager.

1) La gestion des interfaces

- Le développement des vues d'interface, de la notion d'interface sans corps, de bibliothéque et la prise en compte de ces aspects dans le processus de construction de configuration.

- Le calcul automatique des dépendances. A partir de conventions de désignation fixées par Adèle, les dépendances sont analysées et extraites automatiquement de tous les sources d'interfaces ou de réalisations de modules introduits dans la base.

2) L'interface usager

De nouveaux outils ont été introduits pour améliorer l'interface usager:

- la commande **exec** pour exécuter un bloc de commandes sur la liste des composants d'une configuration (cf §4.6)

- la commande **install** permet d'installer (création) une base de programmes à partir des sources contenus dans l'espace privé d'un usager. Cet outil permet de gérer avec Adèle des logiciels développés de manière classique. La commande prend comme entrée un nom de base et un nom de fichier source qui permet de créer la famille racine. Les dépendances de ce fichier

source sont analysées et récursivement les autres familles sont créées à partir des sources dépendants. Ce processus est contrôlé de manière interactive par l'usager.

- la commande **resuser** donne la liste de l'ensemble des objets réservés par l'usager. La commande **replace** prend la liste des objets réservés et les replace dans la base. Ceci permet à l'usager de ne pas quitter la base avec des objets réservés.

6.1.2. Aspects techniques

La tâche essentielle a consisté à adapter le prototype à un cadre d'utilisation industrielle pour les systèmes Unix. Les caractéristiques techniques qui ont été développées sont :

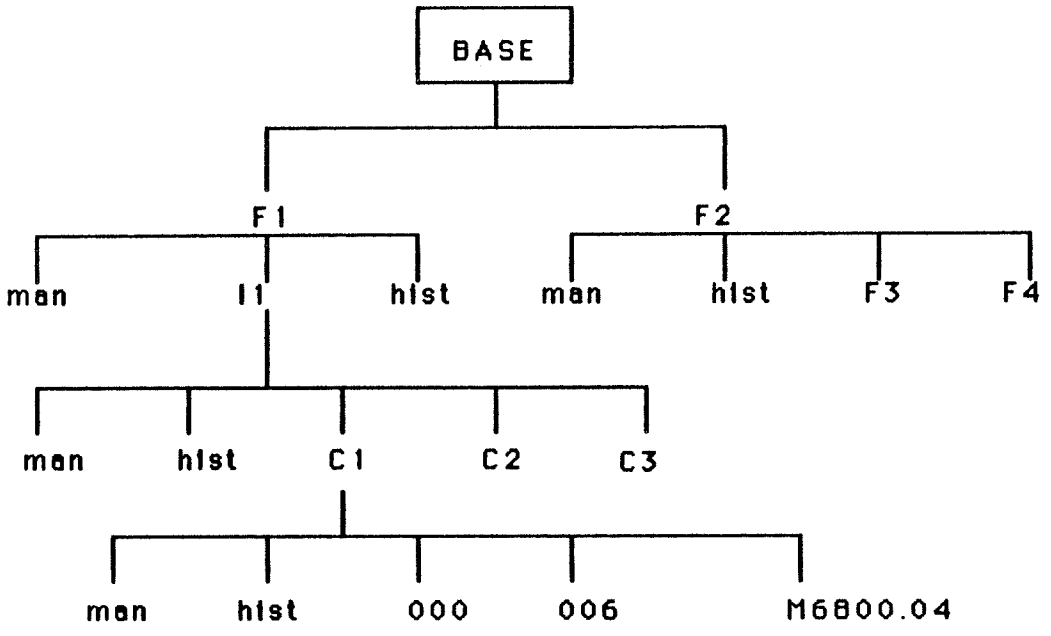
- la gestion de l'accés multiple à la base de programmes pour permettre son utilisation simultanée par plusieurs personnes;

- l'amélioration de la portabilité et de l'efficacité: La partie qui interface le système d'exploitation a été modifiée et entièrement réécrite en C afin d'être plus efficace et plus facilement portable sur les machines supportant le système Unix. Adèle offre actuellement un haut niveau de portabilité.

- la gestion des deltas a été introduite pour une occupation optimisée de l'espace mémoire : 40 à 50 révisions peuvent être conservées dans un espace de la taille d'une révision;

- la reprise en cas de panne.

- la structure du SGF du système Unix a été utilisée pour le stockage des objets de la base Adèle. Chacun des objets composés famille, interface et réalisation est représenté par un annuaire. Les annuaires sont organisés hiérarchiquement et regroupent chacun les objets atomiques associés, représentés sous forme de fichiers comme le montre la figure 3.4.



I_i sont les interfaces, F_i les familles et C_i les réalisations,
man et hist sont respectivement les fichiers descripteurs et historiques.
000 fichier des deltas, 006 est la dernière révision et M6800.04 est le code objet associé à la révision 4.

Figure 3.4 : Structure physique de la base Adèle

6.2. Conclusion de cette expérience

Le prototype réalisé a atteint un niveau exploitable et a permis d'engager plusieurs opérations de transfert vers l'industrie. Parmi les utilisateurs potentiels, Adèle a été utilisé par:

- la SFENA pour la gestion du logiciel de l'Airbus A320 composé de 800 modules et 200.000 lignes de programmes;
- la société Bull qui a expérimenté Adèle sur 150 modules en vue de son utilisation pour la maintenance du système Gecos7 (4 millions de lignes de code);
- la société norvégienne Norsk_Data pour la maintenance de systèmes d'exploitation ;
- la société Robinson (Grande Bretagne) pour un portage d'Adèle sur PC (MS/DOS) en vue de sa commercialisation;
- la société CRIL(France) avec un transport sur VAX-VMS en vue de sa commercialisation.

Nous résumons dans ce qui suit les caractèristiques et les limitations importantes d'Adèle qui ont été confirmées par les premières utilisations.

6.2.1. Le schéma de données d'Adèle

- La modularité est une notion fondamentale dans Adèle. Adèle étend les principes de la modularité à tout langage de programmation classique comme C et Pascal qui ne sont pas modulaires. Il permet de créer une interface pour tout programme. Dans ce dernier cas, l'interface correspond à la partie déclarative des modules et est référencée dans les réalisations comme des fichiers *include*. Les attributs sont appliqués uniquement aux versions de réalisations et ne sont pas généralisés à l'ensemble des objets de la famille. La structure de l'entité famille a été réétudiée dans Nomade. Elle a été enrichie par de nouvelles informations permettant de mieux contrôler sa structure.

- La relation de structuration du logiciel est la relation de dépendance prédéfinie dans Adèle. Une fois le graphe de dépendance défini, la structure est figée, et il n'est pas possible d'insérer ou de déplacer une famille à cause de la forte dépendance de la structure logique par rapport à l'implantation physique. La désignation des modules est étroitement liée à un gestionnaire de fichiers hiérarchique (voir la figure 3.4 qui représente la structure physique de la base). Toute insertion ou suppression de famille nécessite des changements importants portant sur la structure physique de la base et la désignation des objets de la base (chemin d'accés). D'un autre côté, cette contrainte de structuration oblige à une conception descendante de la base de programmes.

6.2.2. La gestion de configurations

L'utilisation des attributs rend aisée le processus de définition de configurations et incite à décrire les versions en terme d'attributs. C'est un aspect puissant d'Adèle puisque seules quelques lignes suffisent pour définir une configuration complexe. Cependant Adèle ne supporte pas la description du processus de génération (application d'outils dans notre terminologie) pour la dérivations d'objets comme par exemple la construction de l'exécutable d'une configuration.
L'absence d'un mécanisme de reconstruction nous a amenés à introduire l'outil *Exec* pour la manipulation interne des configurations. C'est un outil limité puisqu'il ne permet pas d'appliquer des opérations différentes aux objets en fonction de leur caractéristiques particulières. Seule la manipulation en bloc est permise. Un processus plus général d'utilisation et de génération d'objets dérivés par l'activation d'outils est introduit dans Nomade.

6.2.3. Les mécanismes de coopération et de protection

- Le contrôle de la relation de dépendance:

Le contrôle de la relation de dépendance par les règles de visibilité s'applique seulement à la dépendance entre familles et ne nous permet pas de contrôler aisement la structure entre entités plus générales. Pour ceci, il faut donner une existence réelle aux entités conceptuelles d'un plus grand niveau d'abstraction. D'un autre côté, la relation de dépendance n'est pas la seule relation structurante, il en existe d'autres.
Ces aspects sont étudiés dans Nomade dans un cadre plus général de définition et du contrôle des relations et de l'organisation des familles en classes pour obtenir des entités conceptuelles de plus haut niveau que les familles comme par exemple tous les modules d'une configuration.

- Les droits d'accés .

La notion d'espace de travail est utile pour renommer des familles puisqu'elle permet d'utiliser un nom unique plus pratique qu'une désignation arborescente. Par contre, c'est une définition statique des droits d'un usager qui restent liés à la structure arborescente avec une granularité de contrôle trop importante (sous-arbres des familles).
Les objets évoluent mais les droits sur les objets restent les mêmes. Cet aspect sera réétudié dans Nomade en introduisant une politique de droit plus flexible.

- La gestion des effets de bord.

Les effets de bord dans Adèle donnent un compte rendu des effets propagés mais ne permettent pas des déclenchements d'actions.
La gestion des effets de bord a été réétudiée et généralisée dans Nomade :

- Il est possible de définir d'autres types de relations et de les rendre génératrices d'effets propagés;

- Dans Nomade, les effets propagés permettent que des actions soient exécutées.

7. CONCLUSION

Les limitations d'Adèle et les derniers développements de la recherche nous ont amenés à revoir les concepts de la base de programmes et à l'étendre vers un noyau de programmation globale.

7.1. Vers un environnement ouvert et adaptable

Des environnements de programmation globale ont été développés et ont fourni un support puissant pour les étapes clés du développement et de la maintenance. Ces environnements tirent souvent leur puissance et leur efficacité d'un langage de programmation ou d'une méthode de développement qu'ils imposent comme préalable à leur utilisation. Nous avons vu précédemment quelques exemples de systèmes qui encouragent et supportent cette vue. Les systèmes développés sont dépendants d'un langage, d'une méthode et sont difficiles à modifier. Mais très vite, ces environnements ont eu à manipuler des logiciels écrits en différents langages et à s'adapter à différentes organisations et méthodes de développement et soulevèrent des problèmes d'accomodation. La nécessité d'un environnement flexible, indépendant de tout langage de programmation, outil ou méthodes semble admise comme en témoignent les nombreux projets actuels. Une fois admise l'extensibilité des environnements, il devient nécessaire de repenser leur architecture pour permettre l'intégration de nouvelles caractéristiques comme de nouveaux langages, méthodes avec peu d'effort de développement.

Les développements entrepris sur Adèle nous ont permis de valider un certain nombre de concepts et nous ont amenés à changer notre approche concernant le niveau de généralité a atteindre. Le projet Nomade tend à combler ces lacunes. Il s'appuie sur un certain nombre d'idées qui découlent de l'expérience d'Adèle dans des environnements industriels notamment son adaptation à un environnement particulier qui prenait un temps non négligeable.

Plus précisemment l'objectif final de Nomade est la définition et la réalisation d'un noyau d'environnement de programmation globale. Nomade propose les concepts généraux à mettre en oeuvre dans un noyau et une structure générale qui permette de résoudre les points suivants:

- la gestion des composants logiciels,
- la gestion et le contrôle des versions et des configurations,
- l'activation d'outils permettant le contrôle de contraintes de cohérences complexes

De nombreux outils et méthodes sont spécifiques à une entreprise et il est illusoire de vouloir imposer quelque méthode, outil et convention que ce soit. Un des objectifs de Nomade est de permettre d'une part de définir les conventions et méthodes utilisées et d'autre part de garantir qu'elles seront effectivement respectées.

Nomade constitue un progrés sensible par rapport a Adèle, en augmentant la puissance de modèlisation de la base, sa flexibilité et sa généralité. Il semble que ce soit un bon compromis entre une base de donnée générale et une base de programmation globale. Ce compromis est obtenu en figeant dans le modèle le concept de Module (cf. chapitre 4). Nomade intégre une base d'objets et un ensemble de mécanismes. La base d'objet supporte les objets logiciels, leurs versions et les informations qui leur sont associées. Les mécanismes permettent d'une part le contrôle des configurations, la définition des méthodes, l'intégration et l'activation d'outils et d'autre part d'imposer l'utilisation des outils, méthodes et stratégies précédemment définis.

Les deux évolutions majeures pour arriver à un tel noyau sont:

1) L'extension du modèle de structuration du logiciel Adèle.
Le modèle est enrichi par de nouveaux types de données et des mécanismes de structuration pour contrôler l'évolution du logiciel. Le chef de projet peut ainsi définir des classes d'objets qu'il veut gérer avec un contexte approprié de traitement. Cette extension est nécessaire pour s'adapter à différentes organisations, chacune ayant sa manière de décrire ses objets et de conduire les opérations.

2) L'extension de la base de programmes pour la rendre active.
Un gestionnaire d'activité est construit, basé sur le concept d'événements-actions. Ce mécanisme est nécessaire pour supporter les outils de génération de configuration, pour traiter les effets de bord et imposer des procédures de travail.

Ces deux aspects introduisent de nouvelles fonctions en réponse aux critiques formulées après les premières expériences d'utilisation d'Adèle et intègrent les nouveaux développements de la recherche. Ils doivent amener Adèle vers un noyau de développement et de maintenance. Plus précisément, Nomade est un noyau qui fournit un modèle pour la construction, l'intégration, le contrôle et l'évolution des logiciels et de leurs équipes de développement.

Les grandes lignes du noyau Nomade étant esquissées, nous présentons ses caractéristiques plus en détail dans les deux chapitres qui suivent.

CHAPITRE 4 :

UN MODELE DE CONSTRUCTION MODULAIRE DE LOGICIELS EN VERSIONS MULTIPLES

1. PRESENTATION

L'objectif de ce travail est de développer une structure d'accueil (un noyau)[Belkhatir 88] favorable à l'intégration d'outils de programmation globale. Le gestionnaire d'objets est un des éléments de cette structure dont le but est de gérer les objets logiciels et leurs versions. Comme dans les SGF, le gestionnaire d'objets Nomade permet de stocker et d'accéder le contenu des objets. Comme dans les SGBD, il permet de gérer leurs propriétés et leurs relations.

Le gestionnaire d'objets est basé sur un modèle de données qui tend à intégrer dans un même formalisme (1) la description de la structure du logiciel (définition des types d'objets et de relations) (2) la structure des équipes de développement et de support, (3) les règles de gestion (outils utilisés, activités de maintenance, droits d'accès) et (4) les opérations pouvant s'exécuter sur la base. Les fondements de ce modèle visent à supporter la construction modulaire de logiciels en versions multiples, la construction de configuration et les activités de production et de maintenance. Cette approche tranche avec celles où le modèle décrivant le logiciel est indépendant des objets logiciels comme dans Make, dans [Meyer 85], dans les langages d'interconnexion de modules (MIL) [Prietro_Diaz 82] ou la plupart des bases de données logiciel. Dans ce cas une modification du logiciel n'est pas forcément répercutée dans le modèle, et on ne peut pas contrôler l'usage qui est fait des outils et des objets.

Nous avons dégagé dans le chapitre 2 l'ensemble des concepts qu'il nous parait important de modéliser pour la production, la configuration et la maintenance de logiciels développés en versions multiples et décrit une première approche avec Adèle. Ce chapitre décrit la partie du modèle permettant de structurer le logiciel et les équipes de développement et de support.
Dans le chapitre suivant, nous présentons la partie du modèle décrivant les règles de gestion du logiciel, et le mécanisme d'événements actions développé pour permettre leur interprétation.

2. CRITERES DE CHOIX POUR UN MODELE UNIFIANT DONNEES ET ACTIVITES LOGICIEL

Dans Nomade, les critères qui ont été retenus pour la conduite de l'étude sont les suivants:

2.1. Généralité et adaptabilité

Les environnements de génie logiciel qui imposent des méthodes et des stratégies ont été constamment abandonnés par la communauté industrielle. D'autre part, il n'est pas possible de construire entièrement un nouvel environnement pour chaque projet. Un environnement doit être général et adaptable. La généralité suppose l'indépendance de l'environnement par rapport aux méthodes et langages de programmation. L'adaptabilité est la capacité de l'environnement à décrire aisément de nouveaux outils, de nouvelles méthodes et à permettre de contrôler leur suivi effectif. Ces deux critères constituent un aspect fondamental qui favorise l'utilisation d'un environnement dans des cadres différents. Pour obtenir cette généralité et flexibilité, nous introduisons un modèle de données rendant programmable l'environnement et son comportement adapté aux besoins des usagers. De nouvelles commandes et options de manipulation de ces objets peuvent être rajoutées. Tous ces changements sont assez simples pour être introduits directement par l'administrateur de l'application sans changer l'environnement lui même. Les outils utilisés sont intégrés et associés étroitement au comportement de l'environnement.

Il y a plusieurs avantages liés à ce type d'environnement:

- les usagers peuvent combiner et utiliser les outils provenant de différentes sources, incluant ceux venant du système d'exploitation, ceux qui sont propres à une organisation et ceux acquis à l'extérieur;

- l'environnement peut être adapté pour supporter des stratégies de gestion propres aux organisations.

- l'environnement est extensible.

2.2. Expression de la sémantique des objets

Pour décrire la sémantique d'un objet, il faut non seulement lui associer ses propriétés (sous la forme d'attributs) et ses relations avec les autres objets, mais il faut s'assurer, lors de tout changement, que ces propriétés et ces relations restent cohérentes. La description d'un objet comprend une partie statique : la définition de ses propriétés et de ses relations, et une partie dynamique : la définition des procédures et outils permettant de vérifier la validité de la sémantique des objets. Par exemple, si une propriété ou une relation devient invalide, quels sont les traitements à appliquer pour restaurer la cohérence.

Nomade s'appuie sur un modèle facilitant la description des propriétés des objets logiciels, leur vérification et la définition des actions à entreprendre en cas de détection d'incohérence.

2.3. Modélisation de la structure du logiciel

Le support à la programmation globale nécessite le maintien de la structure du logiciel. Généralement, l'évolution incontrôlée, résultant de la maintenance, éloigne le logiciel de sa conception originale et entraine progressivement sa destructuration jusqu'à rendre tout changement impossible et par conséquent provoque son abandon. La méconnaissance de l'état courant d'un logiciel, des raisons à l'origine des décisions de conception, des relations entre les différentes parties sont des facteurs qui entrainent cette entropie du logiciel.

Un produit logiciel est en constante évolution. Sa structure n'est pas statique et seul un contrôle rigoureux de celle-ci permet d'éviter sa destructuration.

Il nous a semblé justifié de proposer un ensemble de moyens destinés à contrôler les conditions d'évolution. Dans Nomade, ce contrôle s'appuie sur la définition de la structure logicielle supportée par les concepts de famille, partition; des relations qui les lient, et sur la définition de règles exprimant les structures valides et la façon dont elles peuvent évoluer.

2.4. Mécanisme de support aux activités de programmation globale

La plupart des outils logiciel sont dédiés à une tâche précise de développement. Cependant, ils ont une connaissance limitée des objets qu'ils manipulent; ce qui ne leur permet pas de "comprendre" le pourquoi de cette tâche ni comment elle s'intègre dans un processus de production. Ils ne sont activables qu'explicitement par des actions usagers.

Un environnement peut intégrer la connaissance des objets manipulés et être ainsi amené à participer plus activement à la gestion du logiciel en lançant automatiquement des activités. Par exemple, la définition explicite d'une relation de dérivation entre deux objets peut l'amener lors de toute modification de l'objet source à activer automatiquement l'outil de dérivation et à propager la modificaton à tous les objets utilisant l'objet dérivé.

L'activation automatique, le contrôle des outils et des propagations sont parmi les besoins importants identifiés dans les environnements de programmation globale. Les approches actuelles développent des environnements qui fournissent une assistance par une participation active. Ces environnements sont développés autour :

- d'une base d'objets qui maintient les objets et leurs relations (objet dérivé, objet utilisé par un autre...).
- de règles (actions dans notre terminologie) décrivant les activités de développement et de maintenance. Les règles sont un moyen simple et pratique pour décrire une tâche ou pour effectuer un contrôle de cohérence. Par exemple, on peut définir une règle spécifiant qui a le droit de modifier un objet source, une autre décrivant le processus de dérivation,

- d'un contrôleur de l'activation pour choisir les règles à activer et l'ordre de leur activation.

L'environnement devient un élément actif capable d'exécuter des activités en parallèle avec les actions usagers.

Un mécanisme d'événements-actions permettant l'activation automatique, le contrôle des activités et l'intégration d'outils a été défini dans Nomade. Ce mécanisme s'appuie fondamentalement sur une base d'objets et sur un modèle décrivant les objets, leurs relations et les actions ou opérations sur les objets. Cette approche présente des avantages :

- les actions et les opérations que l'usager peut appliquer aux objets sont unifiées et donc un seul mécanisme suffit.
- il est possible de redéfinir les actions et les opérations pour s'adapter à un environnement particulier.

2.5. Intégration de la gestion de versions et configurations.

Pour les aspects gestion des versions et des configurations, Nomade s'appuie sur Adèle d'où il tire ses fondements et un savoir faire. Dans Nomade, la gestion des versions et des configurations est intégrée dans le noyau.

En ce qui concerne la gestion de configurations, l'expérimentation d'Adèle a montré qu'aucune stratégie n'est intrinsèquement la meilleure. La première version d'Adèle maintenait une cohérence forte qui entrainait la reconstruction des configurations après chaque modification d'un composant avec une cascade de recompilations qui ne satisfaisait pas tous les usagers. Dans la seconde version d'Adèle, la configuration était définie automatiquement (composants identifiés) mais la phase de génération (application d'outils dans notre terminologie, cf. chapitre2) n'était pas supportée. Dans Nomade, nous avons considéré que la stratégie à adopter et les outils à appliquer sont propre à chaque organisation de projet et que le noyau doit fournir des mécanismes pour les intégrer.

2.6. Intégration d'outils

Les problèmes de la programmation globale incluent l'intégration d'outils car autour du noyau, il existe un ensemble d'outils de production et de documentation qui fournissent des services communs et sont par conséquent des constituants de l'environnement. On distingue deux approches d'intégration :

- une intégration dépendante du langage. C'est une intégration étroite, monolythique où tous les outils sont adaptés à l'utilisation de la nouvelle interface du noyau. En particulier les environnements mono-langage sont fortement intégrés. Ces environnements ont une vision uniforme des objets à traiter via la définition d'une interface standard. Les objets ont une

représentation commune dérivée de la syntaxe abstraite pour les composants logiciels. Ce type d'intégration a l'avantage d'encourager l'utilisation correcte d'une méthodologie. Les outils sont dépendants l'un de l'autre; ce qui permet une plus grande efficacité. L'effet produit par un outil consiste à appeler un autre. Par exemple la modification d'un source déclenche l'analyse sémantique puis l'interprétation. Une erreur d'interprétation rappelle l'éditeur.
L'inconvénient de cette approche est le manque de flexibilité dans la mesure où elle ne permet pas d'intégrer les outils classiques développés auparavant.

- une intégration "ouverte" où les outils classiques qui interfacent le système d'exploitation, accèdent au noyau de l'environnement [Snodgrass 86]. Cette intégration regroupe toutes les informations produites et utilisées par l'environnement sans hypothèses préalables sur leur contenu. Ces environnements acceptent d'intégrer des outils, chacun travaillant avec sa propre représentation; le noyau offrant des services puissants de gestion de versions, de configurations et de coopération entre programmeurs. L'usager d'un outil classique doit opérer les transformations d'objets telles que nécessitées par l'outil avant son invocation. Ce type d'intégration est plus flexible car il est indépendant d'une méthode donnée de développement. Chaque outil crée sa représentation et la maintient dans une base qui n'interprète pas le contenu. Cependant, il y a peu de partage de données entre les outils puisqu'ils ne se connaissent pas mutuellement. C'est l'approche d'intégration adoptée dans Nomade; elle est basée sur un mécanisme d'encapsulation et d'activation d'outils.
L'adaptation de Nomade à une chaine de production se fait aisément. Des actions représentant des modèles de production peuvent être définies sous forme de procédures cataloguées. Ces actions activent des processeurs dont les fichiers d'entrée sont issues de la base d'objets et dont les fichiers de sortie créent des objets gérés par Nomade. L'indépendance vis à vis des outils de production se double évidemment d'une indépendance vis à vis de la nature des objets logiciels gérés par la base.

2.7. Quelques aspects d'implantation

L'environnement Nomade est le noyau d'un environnement expérimental plus large et doit supporter des outils intéractifs. Son implantation prend en compte certaines contraintes.

- La distribution.

Nomade est utilisé par un ensemble de stations de travail reliées par un réseau local. Des mécanismes d'accès à distance à la base et des transferts transparents d'objets sont implantés. Des outils et des actions peuvent être exécutés sur différents sites permettant des développements croisés.

- Le partage.

Les mécanismes d'accès multiple et le contrôle multi-usager existent pour permettre à des équipes de développement de travailler simultanément sur un même produit logiciel.

2.8. Concepts de base

Après avoir examiné les caractéristiques des informations d'un environnement de programmation globale, nous avons constaté que certains concepts comme les versions, la gestion de contenu (textes de programmes, documents, entres autres... (cf chap2 §4.2.2)) ne sont pas supportés par les SGBD actuels. Nous avons donc défini un modèle de données logiciel enrichi de concepts spécifiques à la programmation globale et avons augmenté son pouvoir descriptif en introduisant des fonctionnalités nécessaires à l'activation de la base.

Les différentes recherches ont proposé l'utilisation de modèles sémantiques pour représenter les données d'un environnement de développement.
Notre modèle est basé sur le modèle entité-relation binaire simplifié dans lequel s'insérent des caractéristiques "orientées objets" comme la classification, l'héritage et la possibilité de supporter l'activation. Les activités appliquées sont définies comme des méthodes associées aux objets ou aux classes d'objets.

Pour simplifier la construction et la manipulation de logiciels, nous avons introduit dans le modèle deux abstractions; d'une part celle de Famille permettant de prendre en compte la structure du module et son évolution en versions multiples et d'autre part celle d'Usager pour la déclaration de la partie opérative.

Les deux ensembles structurés *Famille* et *Usager* peuvent être enrichis et explicités de plusieurs façons par des caractéristiques qui décrivent leurs aspects externes comme par exemple le langage de programmation, le système d'exploitation...

Le modèle ainsi retenu est spécialisé pour la programmation globale car il est orienté vers la manipulation des modules et la gestion de données non interprétées comme les textes de programmes et les documents. La gestion des objets à un niveau plus fin que celui de module n'est pas supportée directement par la base.
Le modèle est général dans le sens où il est indépendant de tout langage et méthode excepté pour la méthode de construction de configuration qui est intégrée directement dans le noyau.
Après avoir précisé les principales caractéristiques de notre modèle, nous présentons dans ce qui suit nos choix en matière de modélisation.

3. UN MODELE DE SUPPORT A LA PROGRAMMATION GLOBALE

3.1. Présentation

Un environnement de programmation globale manipule des informations complexes concernant le logiciel et le processus de leur développement. L'approche adoptée dans Nomade est de supporter un modèle qui permet la représentation et la manipulation du logiciel en tant qu'objets privilégiés dans un environnement de programmation globale. Le modèle spécifie les entités de l'environnement comme une collection de types de données définis au moyen de types de base et de constructeurs particuliers et exprimés dans un langage spécifique. Comme Nomade est conçu pour le développement des logiciels modulaires mettant en oeuvre plusieurs personnes pour leur développement et leur support, le modèle prend en compte ces spécificités en permettant la manipulation et la représentation de deux objets structurés : famille et usagers en tant qu'objets privilégiés d'un environnement de programmation globale.

3.2. Objets structurés et versions

On appelle objets structurés des objets qui incluent d'autres objets. On distingue deux objets structurés dans Nomade dénommés Famille et Usager.

3.2.1. L'entité Famille

Le modèle Nomade repose sur la notion de Famille, un extension de la notion classique de module adaptée à la gestion de logiciels de taille importante et évolutifs. La Famille regroupe toutes les informations nécessaires pour constituer et décrire le logiciel y compris la documentation et les objets dérivés. Nous reprenons les propriétés structurelles de la notion de Famille telle que définie dans Adèle (cf chap3 §3.4) que nous enrichissons par des informations spécifiques à l'environnement du projet où elle évolue.

Nomade gère les versions de modules regroupées dans l'entité *Famille* et structurées en alternatives et révisions. Les versions permettent de représenter plusieurs instances d'un même objet. Chaque version appartient exactement à une *Famille* : son objet générique avec lequel elle partage une structure commune et un même ensemble de contraintes. La structure commune aux versions d'une famille est donnée par des règles de structuration que nous étudions en §3.3.

3.2.2. L'entité Usager

La tendance en matière de modélisation des usagers dans les environnements consiste habituellement à imposer des profils prédéfinis d'utilisateurs. Il nous a semblé fondamental pour plus de généralité et de flexibilité de décrire de façon homogène le logiciel et les usagers. Nomade

propose une approche d'intégration des notions de *famille* et d'*usager* dans un même modèle et leur applique en général les mêmes opérations. Le terme usager s'applique à toute personne ou groupe de personnes. L'introduction de l'objet *usager* est motivée par plusieurs raisons :

- permettre de modèliser l'organisation des équipes de développement et de maintenace propre à chaque organisation;

- permettre d'établir des relations entre les usagers et les objets logiciels comme par exemple des relations de responsabilté qui définissent les objets sous la responsabilité d'un usager donné;

- rendre possible la définition d'un schéma de protection en liant les usagers à leurs droits d'accès (voir §3.6).

Nous donnons ci-après une définition plus précise de l'organisation globale des entités Famille et Usager.

3.3. Les types structurés prédéfinis Famille et Usager

Pour donner une définition rigoureuse des entités Famille et Usager, nous introduisons la notion de *type structuré*. Le type structuré définit le mode de construction des entités. Les informations sont représentées à l'aide de schémas définissant le type structuré, c'est à dire des entités de forme prédéfinie dont il suffit de remplir les cases. Appliqué aux entités d'une famille et usager, il permet d'exprimer les différents composants qui les constituent. Ceci permet d'appliquer une même présentation aux composants et par conséquent de mieux contrôler leur développement. Les types structurés Famille et Usager sont prédéfinis dans le modèle Nomade (par abus de langage, nous employons par la suite entité Famille ou Famille et entité Usager ou Usager respectivement à la place d'entité de type Famille et d'entité de type Usager). Nous nous appuyons sur les types de base et sur la notion de constructeur pour les définir.

3.3.1. Type de base et granularité des objets

Dans les environnements mono-langage, le grain (le plus petit objet manipulable) peut être variable. Une approche réaliste consiste à ne prendre en compte que les aspects les plus pertinents pour la programmation globale. Habituellement, les informations constituant l'interface d'un module (ressources exportées et importées) sont considérées comme les objets de base et peuvent être classifiées en procédures, variables, types et constantes. Cette approche est intéressante car, à partir de ces informations, la relation de dépendance peut être dérivée, et la vérification de types inter-modules peut être facilement traitée. Par contre les problèmes syntaxiques ne sont pas aisément résolus dans un environnement multi-langage, et des problèmes d'efficacité apparaissent car tout changement conduit à manipuler un volume important d'informations.

Nomade est principalement un système de construction modulaire. L'information apportée par le module est un ensemble structuré d'objets correspondant à la notion classique de fichiers. Ces fichiers sont des unités textuelles représentant les éléments algorithmiques du module comme par exemple le code source d'une réalisation, le code objet dérivé, le texte de l'interface...

Dans Nomade, chacun de ces objets constitue une unité à part entière. C'est à ce niveau de granularité que les objets sont manipulés dans Nomade sans interprétation de leur contenu. Comme conséquence, le grain des objets analysés dans Nomade est assez gros; c'est le type primaire, choisi ainsi pour plus de généralité, d'efficacité et de simplicité. Généralité parce que la base est réellement indépendante du langage et peut intégrer tout type de document. Efficacité parce que le nombre d'objets est raisonable. Simplicité parce que le noyau est maintenu à une taille raisonnable.

Une granularité plus fine ne serait adaptée qu'à certains langages et obligerait notre modèle à avoir une orientation mono-langage prononcée.
Par conséquent, dans Nomade, un module est représenté par un groupe d'unités textuelles; cette dernière étant le type base (niveau de granularité le plus fin). Nous désignons par la suite ce type de base par **Text.**

3.3.2. Les constructeurs

Un constructeur est un opérateur de composition permettant de définir des types. Deux constructeurs semblent nécessaires, chacun décrivant un type d'objet structuré.

- Le constructeur **agrégation** (constructeur de type produit cartésien : bloc **begin...end**) définit le type d'un objet structuré à partir d'un ensemble ordonné d'objets de base ou structurés de différents types.

Exemple :

Une procédure peut être considérée comme un objet structuré formé par aggregation de deux objets de types de base : sa définition et son corps. Ceci s'exprime de la manière suivante.

```
procedure =
 begin
   définition-proc = Text;
   corps-proc = Text;
 end
```

- Le constructeur **liste** définit un type d'un objet structuré à partir d'une liste ordonnée d'objets de base ou structurés de même type. (**list** [cardinalité min.. max] **of**)

Exemple :

Une alternative d'interface est constituée d'une liste de vues; une alternative de réalisation d'une liste de révisions.

```
alternative-réalisation =   list[1..*] of révisions;
alternative-interface =     list[1..*] of vues;
```

L'analogie entre les constructeurs dans les modèles de données et les constructeurs de types dans les langages de programmation est évidente. De même que la structure de données en programmation correspond au modèle de données, de même certains constructeurs ont des correspondances. Ainsi l'*agrégation* correspond au *record* de Pascal et la *liste* à la structure de *tableaux*.

Avec ces deux constructeurs, on peut définir les corps des types structurés Famille et Usager tels que prédéfinis dans le modèle Nomade.

Les informations apportées par le type d'objet structuré *Famille* sont représentées par un ensemble d'objets de différents types et assimilant totalement le contenu du logiciel décomposé en modules. La structure est associée aux texte des versions de modules grâce à une arborescence qui permet d'identifier les objets correspondant à chaque partie (interface, réalisation, révision..). Ces objets sont à leur tour soit structurés soit des objets de base par conséquent non décomposables. Les objets de base sont des unités textuelles représentant le contenu de programmes. Les informations apportées par le type d'objet structuré *Usager* représentent à ce stade l'identification de l'usager.

Deux types d'objets de structure prédéfinie sont intégrés dans les structures Famille et Usager. Il s'agit du type d'objet *manuel* qui contient des informations de gestion, définies dans ce chapitre et du type d'objet *historique* qui trace l'ensemble des opérations effectuées, répertoriées par objet de la structure.

La partie commune aux types structurés Famille d'une part et Usager d'autre part est définie comme suit dans le modèle Nomade :

Structure prédéfinie du type Famille

```
structure Famille =
begin
INTERFACES= list [1..*] of  (ALTERNATIVE-INTERFACE =
          begin
          vues_interface = list[1..*] of Text;
          REALISATIONS = list [0..*] of (ALTERNATIVE =
                begin
                REVISIONS = list [1..999] of  (source-dérivés =
                          begin
                          code_source= Text;
                          objets_dérivés = list[0..*]  of Text;
                          end)
                manuel =Text;
                historique =Text;
                end )
          manuel =Text;
          historique =Text;
          end)
manuel =Text;
historique =Text;
end;
```

Structure prédéfinie du type Usager

```
structure Usager =
    begin
    nom_usager = Text;
    manuel =Text;
    historique =Text;
    end;
```

3.3.3. Discussion

Nous avons considéré les types Famille et Usager en se limitant aux aspects purement structurels. Cependant les parties structurelles à elles seules sont insuffisantes pour décrire toutes les caractéristiques d'un objet structuré dans la mesure où elles ne permettent pas de rendre compte

de l'utilisation qui en est faite. D'autres annotations constituant les parties descriptives des objets ont été introduites et utilisées dans la définition des types structurés Famille et Usager comme par exemple la représentation de l'environnement dans lequel le logiciel est développé et exploité. La partie descriptive est représentée par les entités *document, attributs, relations, actions* et *droits d'accès* dans le cas des usagers qui ont été regroupées pour constituer avec la partie structurelle la définition des types structurés Famille et Usager. L'ajout de ces entités permet d'envisager des définitions de types structurés adaptées à chaque projet, famille ou usager. C'est pourquoi nous proposons un langage permettant de définir des types structurés adaptables à une organisation de projet.

Nous représentons dans la figure 4.1 les types structurés Famille et usager telles que prédéfinies dans Nomade combinant la partie structurelle et la partie descriptive. Cette dernière étant étendue et adaptée à la spécificité de l'environnement.

```
structure Famille
structure FAMILLE=
begin
nom-famille =Text;
INTERFACES = list [1..*] of
   (ALTERNATIVE-INTERFACE=
   begin
   VUES-INTERFACES = list[1..*] of
      begin
      nom-vue =Text;
      contenu_vue =Text;
      end
   REALISATIONS=list[1..*] of (ALTERNATIVE=
      begin
      nom-alternative=Text;
      REVISIONS=list[1..999] of (source-dérivés =
         begin
         num-révision=integer;
         code-source = Text;
         objets-dérivés=list[0..*] of (od=
               begin
               nom_objet_dérivé=Text;
               contenu_od =Text;
               end)
            documents;
            end)    /* fin revision*/
         manuel =Text;
         historique =Text;
         documents;
         end)            /* fin alternative */
      manuel =Text;
      historique =Text;
      documents;
      end )              /* fin interface*/
documents =list[0..*] of (document =
   begin
   nom-document= Text;
   contenu_document = Text;
   end);
manuel =Text;
historique =Text;
attributs;    /*clause de définition d'attributs*/
relations;    /*clause de définition de relations*/
actions;      /*clause de définition d'actions */
end;          /* fin de la structure Famille*/
```

```
structure Usager
structure Usager =
begin
nom-usager =Text;
documents = list[0..*] of (document =
         begin
nom_document = Text;
         contenu_document = Text;
         end)
manuel =Text;
historique =Text;
attributs;  /*clause définition d'attributs*/
relations; /*clause définition de relation*/
actions;  /*clause  définition d'actions*/
end;      /*fin de la structure usager*/
```

Figure 4.1 : Types structurés Famille et Usager

Nous explicitons dans ce qui suit chacune des annotations de la partie descriptive, c'est à dire les entités attributs, relations, documents et actions.

3.4. Les éléments associés document

L'organisation d'une famille et d'un usager peut comporter d'autres objets de différentes natures qui répondent à des besoins divers de représentation d'informations de l'environnement autres que programme comme par exemple des jeux d'essai, des documents de spécification, une liste de tâches associée à l'usager....

La description de ces types objets a été intégrée aux types prédéfinis Famille et Usager et est représentée par le type prédéfini **document**, ayant une position fixée dans la structure comme le montre la figure 4.1. En effet les objets de type document sont intégrés aux structures Famille et Usager et ne peuvent pas par conséquent être décrits indépendamment de ces dernières.

3.5. Les attributs

L'aspect purement structurel est insuffisant en lui-même pour décrire les caractéristiques des objets manipulés dans un environnement de programmation globale. Ceci a conduit à enrichir la description structurelle en introduisant dans le modèle la notion d'attribut. Les attributs permettent de caractériser les objets en terme de leur structure logique et de leur lien avec l'environnement du projet. Par exemple des attributs qui expriment les propriétés des objets comme le langage, le système d'exploitation ou des informations liées au cycle de production comme la date, l'auteur, l'état d'un objet. Le modèle permet une définition globale de ces attributs par type d'objet en mentionnant leur nom et leur domaine sous forme d'une liste de valeurs permises. Ceci permet d'accepter ou de rejeter un attribut lors de l'instanciation .

On distingue deux types d'attributs, ceux pouvant être dérivés automatiquement comme par exemple la date et l'auteur d'un objet et ceux définis explicitement par l'usager. Ces derniers sont définis avec leurs valeurs possibles respectives dans chaque type structuré famille et usager et sont spécifiés par type d'objet composant la structure.

3.6. Les relations

Dans une famille, il existe d'autre relations qui viennent s'ajouter à celle de l'organisation hiérarchique résultant de la décomposition modulaire et qui peuvent lier des éléments pour exprimer des structures et ajouter des faits. Suivant en cela l'approche de [Meyer85], le modèle définit les objets et les relations sous forme binaire. Les relations peuvent être définies entre composants d'une même type structuré famille ou usager ou entre composants de types structurés différents. La définition des types structurés Famille et Usager comprend la clause **relation** qui regroupe les types de relations définis pour un projet. Chacune des relations a un type défini par le

nom de la relation et les types des objets origine et destination (domaines). Les types d'objets de tout niveau peuvent participer dans la définition du type d'une relation. Une relation peut posséder plusieurs définitions relatives à des listes de domaines différents.
L'instanciation d'une relation est faite en donnant son nom, les objets source et destination. La validité du type de l'objet que ce soit source ou destination de la relation est vérifiée par rapport à la définition de la relation dans le type structuré correspondant à l'objet destinataire de la relation.
Dans Nomade, la différence est faite entre de simples relations binaires qui exprime des faits comme par exemple la relation de responsabilité entre un usager et une version de module et celles qui expriment une structure comme par exemple la relation de dépendance entre modules. Ces relations définissent un type de relation appelé **relation structurante** et permettent de décrire une organisation entre usagers ou familles. On rattache des propriétés particulières à ce type de relation dans Nomade.

3.6.1. Notion de relation structurante

La relation structurante est un moyen utilisé pour représenter et contrôler la structure d'un produit logiciel. Elle définit un graphe où les noeuds sont des objets et les arcs des relations. Des graphes multiples peuvent être définis. Par exemple, on peut avoir la structure exprimée par le graphe de dépendance, une autre en appel de processus, une autre en flot de données... En conséquence, la structure peut être appréhendée comme un ensemble de graphes existant simultanément. C'est une généralisation de l'approche d'Adèle basée sur le graphe de dépendance. Dans ce cas la relation structurante peut être n'importe quelle relation (dépendance ou autre).

D'autre part, il existe d'autres types de structures dès lors que l'on a introduit différents types d'objets dans le modèle. Ainsi les usagers, les documents définissent des structures différentes qui coexistent avec la structure des programmes.

3.7. Les actions

Pour décrire la sémantique d'un objet, il faut non seulement lui associer ses attributs et ses relations avec les autres objets, mais il faut aussi s'assurer, lors de tout changement, que ses propriétés et ses relations restent cohérentes. On définit le mécanisme d'événement actions qui permet de vérifier la validité de la sémantique des objets et de spécifier les actions à entreprendre en cas de détection d'incohérence. Les actions sont activées par des événements produits :

- soit de manière externe par des usagers;
- soit de manière interne par la violation de droits d'accès et de contraintes.

A tout événement et objet peuvent être associées des actions. Le mécanisme d'événements-actions est à la base du contrôle de la propagation et de l'activation d'outils dans Nomade.

La définition des actions et le mécanisme d'interprétation qui modèlisent la partie dynamique de l'environnement est présenté plus en détail dans le chapitre suivant.

3.8. Les droits d'accès

A partir du moment où les ensembles d'informations (objets) sont partagées entre usagers, il faut pouvoir les protéger. Le problème de la protection consiste à prémunir les objets des accès interdits. Les environnements actuels offrent des moyens primitifs de contrôle d'accès essentiellement dérivés du système d'exploitation. Plus récemment [Minsky 84] a proposé une protection à base de privilèges. Au schéma de données qui décrit l'organisation des données et leur utilisation est associé un schéma de protection de ces données adapté à chaque organisation. Ce schéma de protection définit les droits associés à chaque usager de l'environnement. Nous adoptons cette démarche dans Nomade.

Les usagers jouent un double rôle dans le système : ils sont soit à l'état passif (objet sur lequel est exécuté une action comme la création d'un nouvel usager ou la modification de ses droits), soit à l'état actif (usager en cours de session). L'entité Usager comme l'entité Famille est décrite entre autres par des attributs, ce qui permet d'exprimer de manière uniforme les droits aussi bien sur les opérations manipulant les entités Familles que les Usagers.

Les droits sont exprimés pour chaque type Usager au moyen de couples (opération, domaine) où opération est le nom de l'action et domaine définit les objets sur lesquels l'opération peut porter.

Exemple de privilèges d'un programmeur :

```
cataloguer        *:*:*. (language=pascal and état = test)
créerusager       * ( type=programmeur and système =unix)
```

Ces privilèges expriment que le programmeur peut effectuer l'opération de catalogage de révisions écrites en pascal et à l'état "test" et qu'il peut créer des programmeurs ayant une compétence sur le système Unix.

L'intérêt principal de cette approche est la puissance et la souplesse offerte; les droits ne sont pas figés pour un type d'Usager, mais évoluent automatiquement en fonction de l'évolution des objets et aucune hypothèse préalable n'est prise sur l'organisation des usagers.

On peut ainsi exprimer que les programmeurs peuvent manipuler les objets dans l'état "test", les intégreurs ceux dans l'état "testé", et les personnes chargées du support technique peuvent lire et copier ceux dans l'état "livré". La dernière action que peut avoir un programmeur sur un objet est de lui affecter l'état "testé" pour le rendre accessible aux intégreurs.

Le contrôle des droits d'accès est mis en oeuvre par la commande de vérification de droits (**checkright**) appelée explicitement lors de l'accès à tout objet. On vérifie que l'objet référencé est inclus dans les privilèges de l'usager pour la commande en cours.

3.9. Classification des objets structurés

Nous allons étudier comment dans Nomade, il est possible de structurer le logiciel en sous-ensembles et d'organiser les usagers en classes.

3.9.1. Notion de partition

Il est pratique, pour réduire la difficulté de développement de logiciels de taille importante de le décomposer en plusieurs parties. Des entités de haut-niveau comme des sous-systèmes sont définies pour organiser le logiciel. Ces entités correspondent :

- soit à des regroupements fonctionnels. Ceci est particulièrement évident pour une décomposition en niveaux d'abstraction où chaque niveau fournit une interface, utilisée pour écrire les modules du niveau supérieur.

- soit à des regroupements de parties de logiciel, dus à des structures de construction communes (par exemple l'utilisation des mêmes opérations, outils...).

La structure des logiciels et des usagers est basée sur la notion de **partition**, une extension des entités Famille et Usager pour définir ces entités de haut niveau. En pratique plusieurs relations permettent de définir ces entités de haut niveau. Pour chacune des relations, une structuration adaptée peut être définie. Par exemple on peut définir la relation de décomposition pour structurer le logiciel ou organiser les usagers par fonctions. La **partition** définit des sous-ensembles de familles et usagers liés par une **relation structurante** partageant des propriétés structurelles et descriptives communes. Ceci permet de contrôler leur évolution car, lors de toute modification, pour des raisons de maintenance ou d'extension, les répercussions sur la structure générale sont détectées. Il y a une correspondance biunivoque entre une classe et une partition puisque toutes les familles et usagers d'une partition ont en commun une même définition de type structuré. Nous définissons dans ce qui suit plus précisemment la partition et détaillons les services qui lui sont rattachées pour un bon contrôle de la structure du logiciel et des usagers.

3.9.2. Définitions

Soit le graphe G(N, A) sans circuit défini par la relation structurante R où N est l'ensemble des familles (noeuds) et A l'ensemble des arcs de la relation structurante R. Soit F une famille particuliere du graphe. La partition F pour la relation R, notée R.F est l'ensemble des familles f tels que tout chemin sur G entre une source du graphe et f passe par F.
La figure 4.2 montre un exemple de structuration en partitions à partir d'une relation structurante R. On définit ainsi 4 partitions P0, P1, P2 et P3 ayant respectivement pour racines F1, F2, F3, F9.

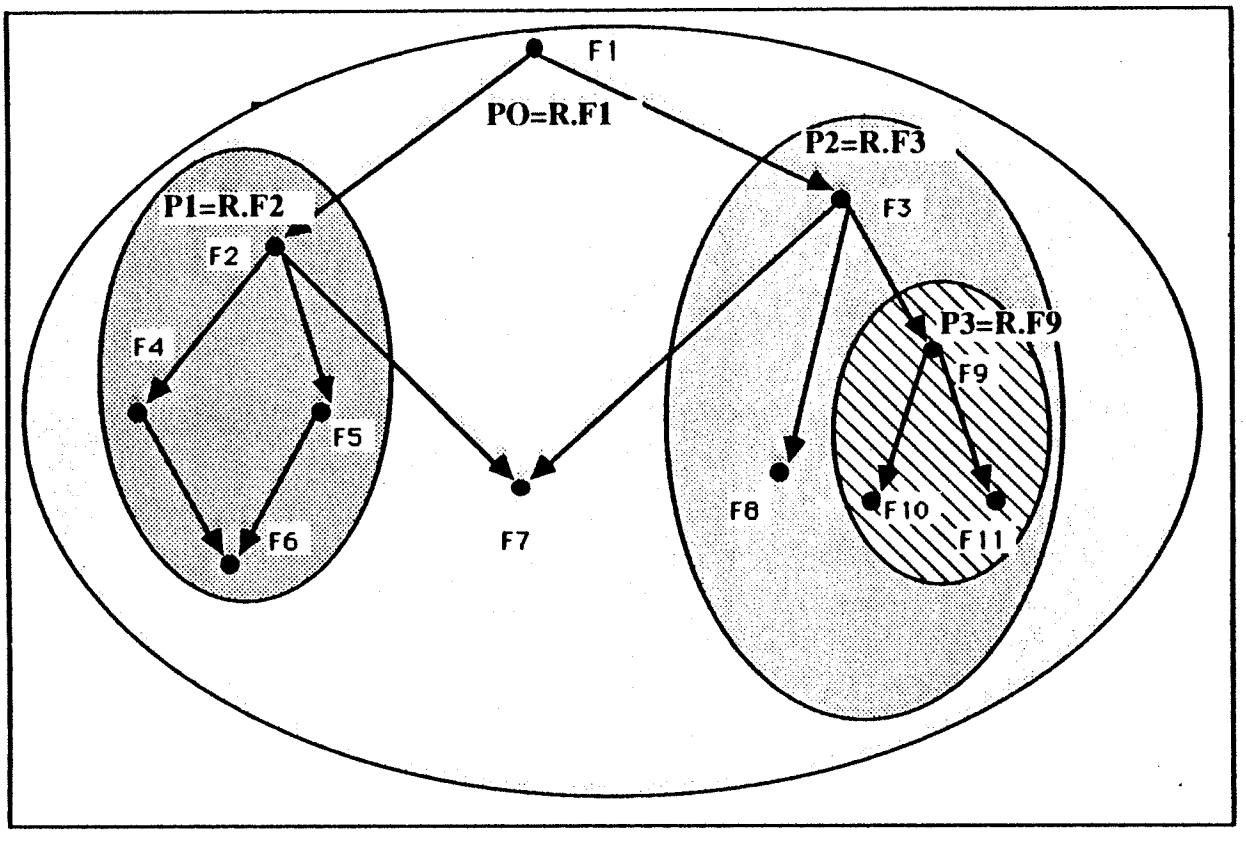


Figure 4.2 : Structuration en partitions

R.F est un sous-graphe de G à point d'entrée unique F.

On note <a,b> ∈ A un arc du graphe G

Nous donnons une définition formelle de la partition:

Soit G(N,A), le graphe sans circuit défini par la relation R;

soit $F \in N$

soit S, l'ensemble des sources du graphe G

$S = \{ s \in N \mid (\nexists x \in N)\ \langle x, s \rangle \in A \}$

Soit C, l'ensemble des chemins du graphe G issus de S

$C = \{ \langle a_0, \ldots, a_j, \ldots, a_k \rangle \in N^* \mid [(a_0 \in S) \wedge (\forall i \in [0, k-1])\ (\langle a_i, a_{i+1} \rangle \in A)] \}$

La partition associée à F pour la relation R, notée R.F, est le sous-graphe :

G_F (N(F),A(F)) tel que:

$$N(F) = \{ n \in N \mid (\exists \langle a_0, \ldots, a_j, \ldots, a_k \rangle \in C)\ [\ a_k = n \wedge (\forall i \in [0, k])\ a_i = F)] \}$$

$$A(F) = \{ \langle n, m \rangle \in A \mid n \in N(F) \wedge m \in N(F) \}$$

Ainsi une partition correspond à la notion, classique en théorie des graphes, d'un graphe connexe qui admet un point d'entrée unique. Ce graphe pouvant à son tour être décomposé en sous-graphes ayant des propriétés identiques.

Cette définition appelle plusieurs remarques :
- Par construction, l'ensemble des partitions relatives à une même relation R constitue une arborescence. Deux partitions de R sont soit disjointes soit imbriquées.

- Une même famille peut appartenir à plusieurs partitions différentes (c'est à dire définies à partir de relations structurantes différentes.

Par convention, La famille racine du projet est implicitement la racine de toutes les partitions.

Exemple de décomposition modulaire et de partitions pour un projet imaginaire de développement de compilateur (figure 4.3)

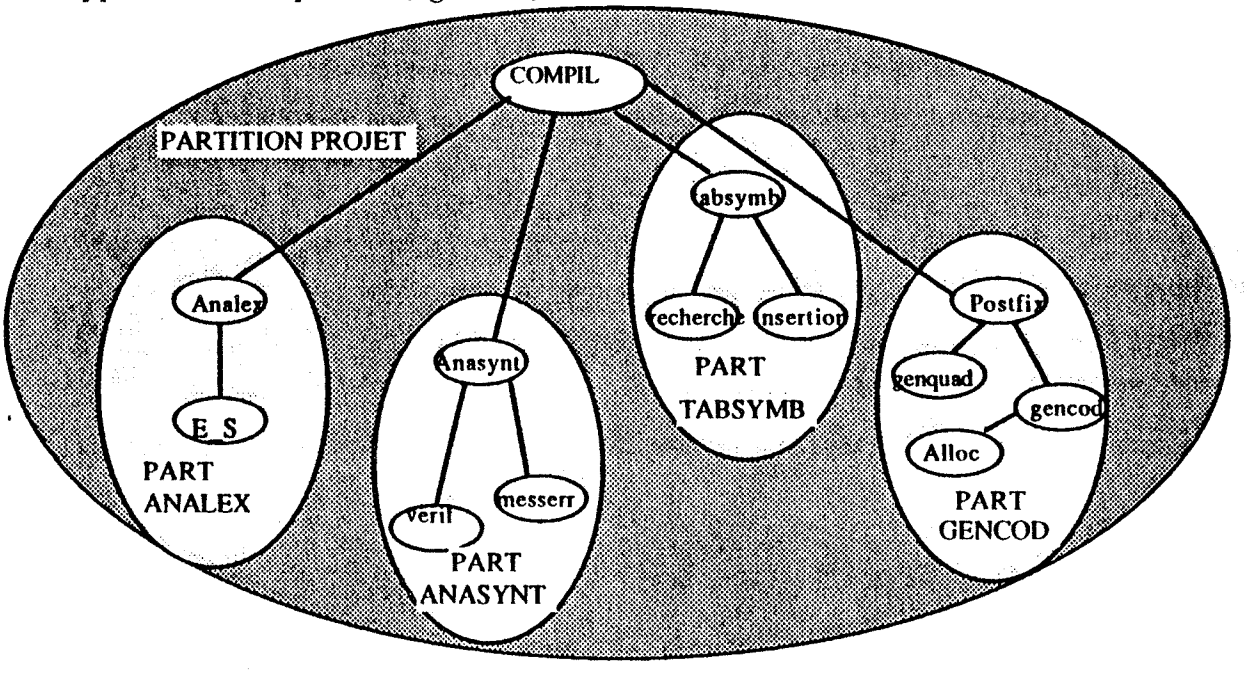


Figure 4.3 : Exemple de structuration en partitions

5 partitions pour la relation structurante de décomposition hiérarchique
- PROJET : Compilateur
- ANALEX : Analyse lexicale (2 modules)
- ANASYNT : Analyse syntaxique (3 modules)
- TABSYMB : Table des symboles (3 modules)
- GENCOD : Génération de code (4 modules)

3.9.3. Construction de partition et lien d'héritage

La partition est chargée de deux sens : un sens conceptuel et un sens ensembliste. Une partition est un concept (par exemple les familles définissant l'entité conceptuelle analyseur syntaxique, la partition définissant la fonction programmeur) et un ensemble (familles ou usagers) qui la composent. En tant qu'ensemble les familles et les usagers d'une partition ont les propriétés de la partition. Si F est une Famille de la partition P, par définition F est un raffinement de P. Le lien de raffinement dans ce cas représente un lien de hiérarchie entre les types structurés. Il traduit les Familles et Usagers satisfaisant la description de la partition. C'est un lien d'héritage.
On distingue deux niveaux de lien d'héritage dans le modèle Nomade :

1) un lien d'héritage résultant de l'appartenance des familles ou usagers à une partition (partage)

Ce lien d'héritage est particulier dans la mesure où il signifie que l'ensemble des propriétés d'une famille ou usager doit être inclus dans celui de la partition. La famille ou usager se contentant de reprendre les types structurés de la partition et de les appliquer entiérement ou par réduction comme par exemple invalider des attributs, des relations... Ce lien d'heritage induit par l'appartenance d'une famille ou d'un usager à une partition ne correspond pas à l'héritage classique mais à un **partage** (terme plus approprié) de propriétés puisqu'une famille ou un usager reprennent les propriétés communes offertes par la partition.

2) un lien d'héritage résultant de la représentation de la hiérarchie entre partitions;

Ce lien traduit une relation "sous-classe" qui signifie une inclusion des propriétés. C'est le cas classique de l'héritage ("isa") entre un type et ses sous-types. Dans Nomade, l'inclusion des propriétés entre hiérarchie de partitions n'est pas systématique mais sélective, c'est à dire contrôlée par l'usager. En effet, on distingue globalement deux cas de construction de partition avec des variantes entre les deux selon la manière dont on veut que les partitions coopèrent.

- On peut définir les partitions en assurant indépendance logique et structurelle. Une partition est un univers protégé et encapsulé. Elle se développe indépendamment des critères, caractéristiques et conventions définis à l'extérieur tout en définissant son propre jeu de caractéristiques et conventions. Aucune connaissance des familles qui la composent n'est directement accessible de l'extérieur de la partition. La partition se présente à ses utilisateurs comme une entité unique et fournit une interface clairement définie. Ceci est utile pour la définition de machines abstraites.

- La partition joue pour la programmation globale, le même rôle que le bloc pour les langages de programmation possédant cette structure. Les caractéristiques utilisées suivent des comportements globaux déclarées dans les partitions englobantes. La partition se

présente à ses utilisateurs comme un ensemble de modules et fournit plusieurs interfaces aux niveaux supérieurs.

3.10. Discussion du modèle

Le modèle est bâti à partir des types structurés prédéfinis Famille et Usager. Chaque type structuré est composé de deux parties : une partie descriptive et une partie structurelle correspondant à l'organisation hiérarchique des entités Famille et Usager. Dans le modèle Nomade, la partie structurelle est prédéfinie dans le modèle. Comme la partie descriptive est adaptable, il est possible de spécifier un type structuré pour chacune des familles et usagers.

La notion de partition permet de contrôler la structure des logiciels en définissant des sous-ensembles de Familles et les relations entre ces sous-ensembles. Comme les familles organisées en partitions partagent des caractéristiques communes, chaque partition contrôle, pour ses composants, les quatre mécanismes de base offerts par Nomade : les attributs, les relations permises, les droits dans le cas de partitions usagers et les actions. Une partition peut hériter de la définition du type structuré de sa partition englobante, l'enrichir ou en être indépendante.

- Si une partition hérite des caractéristiques de ses partitions englobantes, la notion de partition est semblable à celle de classe;

- Si aucun héritage n'est permis et que les protections sont maximales, la notion de partition est semblable à celle de niveau d'abstraction ou de machine abstraite et le logiciel sera alors structuré en couches étanches.

- Entre les deux extrêmes, toutes les variantes sont permises.

En utilisant des relations différentes, plusieurs structures peuvent être définies sur le même logiciel; pour chacune de ces relations, le type de structuration peut être différent. Par exemple, on peut structurer les programmes en niveau d'abstraction (en utilisant la relation de dépendance), structurer les documents en classe (en utilisant la relation "documente"), structurer les usagers en classe également (en utilisant les relations "équipe" et "fonction"). C'est l'un des buts de la notion de partition que de permettre de définir et de contrôler des structurations sans faire d'hypothèses à priori sur le type de structuration désiré.

Un langage de définition permet la définition des objets associés "document" et des clauses (attributs, relations, droits d'accès et actions) utilisées dans la définition des types structurés Famille et Usager. Cette définition peut être faite au niveau de chacune des familles, usagers et partitions avec la spécification du type d'héritage à appliquer via la hiérarchie partition / sous-partition et partition / famille.

4. LE LANGAGE DE DEFINITION DES TYPES STRUCTURES

Nous présentons les caractéristiques du langage de définition qui offre des possibilités adaptées à la désignation des objets et à la définition des types structurés Famille et Usager.

4.1. Organisation générale du schéma de définiton des types structures.

Chaque entité Famille et Usager peut adapter et contrôler pour ses composants les mécanismes de base suivants offerts par Nomade: les attributs, les relations, les droits d'accès dans le cas des usagers ainsi que les actions qui vont permettre de les manipuler. L'organisation du schéma permet une définition modulaire des différents types d'informations sous forme de clauses définies dans l'objet manuel de la Famille ou de l'Usager selon le cas. La définition syntaxique[1] suivante est appliquée pour la définition du schéma qui est développé dans ce chapitre.

Schéma de définition des types structurés Famille et Usager :

```
<FamUser-schema> :: = family | user
                    [<Attributes-clause>]
                    [<Relations-clause>]
                    [<rights-clause>]
                    [<Actions-clause>]
```

Dans le cas d'une famille racine de partitions, les clauses sont définies de manière identique. Elles ont la même sémantique mais leur portée est étendue à l'ensemble des Familles de la partition et des sous partitions. Un bloc partition est défini dans le manuel de la famille ou usager racine de la partition comme suit:

```
<Partition-schema> :: = partition <relation>
                    [<Attributes-clause>]
                    [<Relations-clause>]
                    [<Rights-clause>]    /* si usager */
                    [<Actions-clause>]
```

---

[1]Conventions utilisées pour la définition: **gras**: symboles terminaux, <>: symboles non-terminaux de la grammaire, ::= séparation partie gauche et droits d'une production, |: séparation de différents choix, [ ]: contient une partie optionnelle dans une production, { } indique une partie qui peut apparaitre zéro ou plusieurs fois.

Exemple :

Le schéma définissant les types structurés d'une famille F ou d''un usager U et des partitions R1 et R2, dont F et U sont les racines, est organisé comme suit :

```
schéma de structure
manuel de Famille F ou Usager U
   /* définitions ayant pour portée F ou U */
   Family F ou User U
         <Attributes-clause>
         <Relations-clause>
         <rights-clause>      /* si usager */
         <actions-clause>

   /*définition ayant pour portée F.R1 ou U.R1*/
   partition  R1
         <Attributes-clause>
         <Relations-clause>
         <rights-clause>      /* si usager */
         <actions-clause>

   /* définitions ayant pour portée F.R2 ou U.R2*/
   partition R2
         <Attributes-clause>
         <Relations-clause>
         <rights-clause>      /* si usager */
         <actions-clause>
end;
```

Pour pouvoir utiliser les objets, il faut introduire certains moyens de les désigner. Avant de décrire les procédés de construction de chacune des quatres entités, nous précisons les modes de désignation des objets introduits dans Nomade et utilisés dans la définition des différentes clauses.

4.2. Les modes de désignation

4.2.1. Désignation simple

La désignation simple est basée sur une structure de nom construite en correspondance avec la structure hiérarchique des entités Famille et Usager.

La définition syntaxique appliquée est la suivante :

```
<simple_name>:  ::=   <Object>] [• [  <Element>] | [<User>][•  <Document>]
<Object>        ::=   <Family> [ : <Interface> [ : <Realization> ] ]
<Element>       ::=   <Document> [ • <Revision> ] | <Revision> |
<Revision>      ::=   1..999
```

<User>, <Family>, <Interface>, <Realization>, <Document> sont des **identificateurs**.

"Object" est constitué d'un ensemble d'objets de base notés "element":

element = rev — désigne la révision de numéro rev;

element = • — désigne la révision la plus récente;

element = doc.rev — désigne le document de nom doc associé à la révision rev;

element = doc — désigne le document doc. Les documents hist (pour historiques) et man (pour manuel) sont créés automatiquement et sont gérés par Nomade.

élément = man<identifier> désigne une partie du manuel

Exemple de structure de noms :

`F1:I2:C1.012` : désigne la révision 12 de l'alternative C1 de l'interface I2 de la famille F1

`Prog.spécif` : désigne le document spécif de l'usager Prog

`F1.manattr, F1.manrel, F1.manact` : désigne respectivement la partie du manuel définissant les attributs, les relations et les actions pour la famille F1.

`F.manpart` : désigne la partie du manuel définissant le schéma de la partition dont F est la racine.

`U.manrights` : désigne la partie du manuel définissant les droits de U.

Remarque1:

Famille, Interface et Réalisation peuvent être omis dans la désignation d'un objet; ils sont alors remplacés dynamiquement par la famille courante, l'interface et la réalisation spécifiées par défaut sinon les plus récentes.

Exemple:

.man	désigne le manuel de la famille courante
:I1	désigne l'interface I1 de la famille courante.
: :	désigne la réalisation par défaut de l'interface par défaut de la famille courante.

Remarque2:

Nomade permet la définition d'identificateurs génériques. Des méta-caractéres sont utilisés en reprenant les principales conventions Unix[2]. Ceci permet de désigner simultanément plusieurs identificateurs[3].

Exemples :

F:*:c*:	désigne toutes les réalisations de la famille F dont le nom commence par la lettre "c",
* :	désigne toutes les familles et tous les usagers du projet,
** :	désigne tous les objets du projet.

4.2.2. Désignation simple étendue aux objets de la partition

La désignation précédente est étendue aux familles et usagers d'une partition en préfixant le schéma de désignation simple par le nom de la partition.

Syntaxe :

```
<Name>          ::= [<partition>] [<simple_name> ]
<partition>     ::= / <Relation> • <struct-obj> /
<struct-obj>    ::= <Family> | <user>
<relation>      ::= <identificateur>
```

Exemple :

/dep.F/**	désigne tous les éléments de la partition créée par la relation structurante "dep" avec comme famille racine F.
/dep.F/ *	désigne les familles de la partition F pour la relation dep.

---

[2]Nous précisons les plus importants pour la compréhension des exemples :'*' désigne une chaine éventuellement vide de caractéres sauf les séparateurs suivants (:, ., \ et /). '**' désigne une chaine (éventuellement vide) de caractéres.'?' désigne n'importe quel caractère sauf les séparateurs. [...] désigne n'importe quel caractère ou intervalle de caractères entre crochets.

[3]<identifier> ::= <L> | <métachar> { <L> | < C> | <metachar>}

<metachar> :: = * | ? | [|] <L> est l'ensemble des lettres y compris '-' et '_' et <C> l'ensemble des chiffres.

/dep.F/vms*:*.specif

désigne les documents de nom "spécif" des interfaces des familles de la partition dep.F dont le nom commence par vms.

/programmeur.U/*

désigne les usagers de la partition créée par la relation structurante "programmeur" avec comme famille racine U.

4.2.3. Désignation par la qualification des objets

Les attributs rendent possible la définition d'une désignation à base d'une expression d'attributs qualifiant les objets des entités Famille et Usager. On la définit comme suit :

Syntaxe :

```
<QualN>            ::= [not] <Name> [ ( <LsAndQual> ) ]
<LsAndQual>        ::= <Qualification> { and <Qualification> }
<Qualification>    ::= <name_Attribute> <RelOp> <Value>
<value>            ::= $U\ <name_attribute> | <identifier>
<name_attribute>   ::= <identifier>
<RelOp>            ::= < | > | <= | >= | <> | =
```

Exemple :

F1:I2:* (state = archive and syst=Unix4.2) : désigne toutes les réalisations de l'interface I2 de F1 archivées et associées au système unix4.2.

*(author = $U\name) : désigne toutes les familles dont l'attribut auteur est le nom (indiqué par l'attribut name) de l'usager en cours de session (indiqué par $U).

4.2.4. Définition de l'expression de désignation de Nomade

Une définition généralisée à l'ensemble des familles et usagers d'un projet logiciel est définie sous forme de conjonctions de disjonctions de désignations simples et par qualification des objets dans la base. Ceci correspond à la définition de l'expression de désignation des objets manipulés par Nomade. Cette désignation peut être utilisée comme langage d'interrogation ou pour exécuter toute commande sur un ensemble d'objets. Cette expression peut apparaître dans une expression booléenne. Elle est considérée comme une proposition logique dont la valeur est vraie s'il existe un objet dans la base qui satisfait cette proposition, faux sinon.

Syntaxe : Expression généralisée Nomade

```
<NomadeExpr>     ::= <LsOrLsAndQN> | not ( <LsOrLsAndQN>)
<LsOrLsAndQN     ::= <LsAndQN> | ( <LsAndQN> )  { or ( <LsAndQN> ) }
<LsAndQN>        ::= <QualN> { and <QualN> } | not ( <QualN> { and <QualN>})
```

Exemple :

F1:I2:* (state = archive and syst = Unix4.2) or F2:I2:C1.12 (syst = vms)

4.3. Définition des attributs

Chaque attribut est défini par son nom suivi du signe égal et d'une énumération de valeurs possibles. Les valeurs peuvent contenir des méta-caractères. La déclaration indique, pour une Famille ou un Usager, les types d'objet valides et les attributs permis (opérateur =), imposés et hérités (opérateur ==) par les instances.

Syntaxe :

```
<Attributes_clause> ::= attributes [ def <NomadeExpr > <list_attributes>]
<list_attributes>   ::= [ <name_attribute > = [= ] <value> [ , <value> ] ; ]
<nam-attribute>     ::= <identifier>
<value>             ::= <identifier>
```

Exemple :

Définition de la clause attributs d'un type structuré appliqué à la famille F

```
manuel Famille F
attributes
def **                  /* attributs valides pour tout type d'objet (**) */
    syst= unix???       /* ? désigne un métacaractère pouvant être substitué par un
                           caractère quelconque différent de / , : et . */
    temps_réponse =[0-9][0-9][0-9]ms;
def :*                  /* attributs valides pour les objets de type interface */
    language = h, c;
def:*:*                 /* attributs valides pour les objets de type réalisation
    language = pascal, C, pl1;
def : *.specif          /* attributs valides pour les documents de spécification
                           d'interface */
    type == specif;     /* cette déclaration indiquent les attributs permis imposés
                           et hérités par les instances du document spécif */
    format ==ODA;
    editeur= *;
    def: *:*.docreal    /* attributs valides pour les documents de réalisation */
    * = *               /* tout attribut est valide */
end
```

La création d'un document dans une Famille F n'est acceptée que s'il est défini dans le modèle valide pour F. La création d'un attribut pour tout objet de la famille F n'est valide que s'il figure dans le type structuré définissant F.

Exemple :

Les attributs comme :

syst = unix_4.2,

syst = unix_v5,

temps_réponse = 422ms

sont valides dans la Famille F. Les interfaces ne peuvent avoir que langage = h ou c.

Cas des attributs prédéfinis

Nomade gère implicitement des attributs. Certains sont modifiables, d'autres ne peuvent être que consultés.

Attributs modifiables:

state = exp /* spécifie l'état pour les révisions (cf chap 3. § 5.4)*/

stateconf = ok /* spécifie l'état pour les révisions de configurations (cf chap.3 § 5.4) */

language = * /* spécifie le langage pour les révisions */

Attributs non modifiables et prédéfinis:

date = date de la création ou date de la dernière modification;

author = nom de l'usager ayant créé l'objet;

object = family, user, interface, realization, body, conf;

elem = sourcetext, objcode, doc, revdoc, hist, man, manattr, mansel, manaction, manrights, manpart, manother;

partition= noms des partitions auxquels un objet appartient

family = nom de la famille à laquelle un objet appartient;

interface= nom de l'interface d'un objet;

realization= nom de la réalisation à laquelle appartiennent les objets associés;

name = nom de l'objet.

Exemple :

L'objet F:I:C.specif a comme attributs implicites (object = realization, elem = doc, family = F, interface =I, realization= C, name = specif)

4.4. Définition des relations

Deux objets dans Nomade peuvent être mis en relation. Par définition, les relations structurantes sont définies dans la famille projet qui est racine des partitions pour chacune des relations structurantes. Les relations définies dans une partition sont dites associées à la relation structurante définissant la partition.

La déclaration d'une relation est exprimée par les règles suivantes:

syntaxe :

```
<Relations-Clause> ::= relations { <Source> <Relation> <Destination>;}
<Source> :: =              <NomadeExpr> | reflex
<Destination> ::=    <NomadeExpr>
<relation> ::=             <identifier>
```

Si le domaine de l'objet source est exprimé par le mot clé **reflex**, alors la relation est réflexive. L'intérêt des relations reflexives est la création de commandes associées à l'objet porteur de la relation. Cet aspect est présenté dans la chapitre 5 §4.1.

Exemple:

Soit les définitions suivantes de relations dans le manuel de la famille F:

```
manuel F
relations
**.specif                  specifie    :*;
```

La relation spécifie est définie entre un document spécif attaché à n'importe quel objet et une interface quelconque de F.

```
* *                       dep         :Istd;
(F4:*)  or  (F4:*:*)            dep          :IspecialF4;
**(syst=unix*)            dep         (:*(syst=unix))
```

Les 3 relations indiquent respectivement que l'interface F:Istd est accessible par tout objet de la base; l'interface F:IspecialF4 ne peut être accédée que depuis les interfaces ou réalisations de F4; enfin seuls les objets ayant l'attribut syst=unix* peuvent dépendre des interfaces de F ayant l'attribut syst= unix.

```
reflex                     compile     ** (elem = Sourcetext);
```

La relation **compile** est reflexive et peut être appliquée à tous les textes sources de F.

4.5. Définition des droits d'acces

Un usager est syntaxiquement désigné comme une famille. Dans la plupart des commandes, un objet usager est indiscernable d'un objet famille. Comme tout objet, il possède un manuel dans lequel apparait en plus la clause "*rights*" qui précise ses droits.

Les droits d'un usager sont définis sous forme d'étiquettes associées à des expressions Nomade de la manière suivante :

syntaxe :

```
<Rights-clause>::=  rights (<Label> : <NomadeExpr> (, <NomadeExpr>);)
<Label> ::=         <identifier>
```

Chaque commande Nomade est définie par un programme d'action(voir chap5). Le programme d'action peut contenir comme pré-condition une fonction définie pour la vérification de droits :

checkright (étiquette, O)

qui analyse si l'usager courant peut exécuter la commande. Cette analyse se fait en vérifiant si l'expression Nomade définie par **étiquette** dans la clause *rights* du manuel de l'usager en cours de session désigne l'objet **O** .

Exemple:

Prenons l'exemple d'un programmeur P dont les droits sont les suivants.

manuel P

P a peu de droits :

```
Catal       :     not (*.manrights);
```

Il lui est interdit de modifier les droits de quiconque.

```
res_catal: /dep.F/ *:*:*. (syst = unix) , ( /dep.F/ *:*:*. .
(syst=unix ) );
```

Il ne peut que réserver et cataloguer dans la partition /dep.F/ des réalisations pour le système Unix (syst=unix) et les codes objets correspondants.

```
creer       :     /dep.F/ *:*:* ;
voir_lire   :     /dep.F/ ;
*           :     ** (author = P)
```

Il peut voir et lire tout objet de la partition dep.F, créer des réalisations dans dep.F et exécuter les autres commandes uniquement sur les objets qu'il a créés.

4.5.1. La modification de droits

Les droits usagers peuvent évoluer de manière dynamique. Un usager peut définir, ajouter ou modifier les droits d'un autre usager. Nous indiquons ici les principes directeurs pour la distribution et la revocation des droits entre usagers.

Les droits peuvent être transmis individuellement à chaque usager ou collectivement à un groupe d'usagers via les partitions. Nous indiquons les deux privilèges particuliers que doit avoir un usager pour ajouter, retirer ou modifier les droits d'un autre usager.

- Un usager doit avoir le droit de cataloguer la partie "droits" du manuel pour modifier les droits d'un autre usager : "*catal U.manrights*".

- Tout usager qui posséde des droits sur une commande peut céder ou retirer ses droits (ou des droits plus faibles) à un autre usager. Ce contrôle est effectué en comparant, pour chaque commande de modification de droits, ceux avant et après modification. Chaque étiquette de droits retirée ou ajoutée, doit être incluse dans les droits de l'auteur de la modification.

Exemple :

```
manuel usager U1
    rights
          catal      :     U2.manrights /* peut modifier les droits de U2 */
          create1    :     /dep.F/  *:*:*;
          create2    :     /dep.F/  *:*;
          delete     :     /dep.F/ (author = P);
end
```

```
manuel usager U2
/* AVANT modification des droits*/


rights
   delete : /dep.F/ (author = P);

end
```

```
manuel usager U2
/*APRES modification des droits par U1,
opération autorisée car le privilège create1
appartient à U1 */
 rights
      create1  : /dep.F/  *:*:*;
      delete   :  /dep.F/(author=P);
end
```

5. HERITAGE : REPRESENTATION ET REGLES DE RECHERCHE

5.1. Présentation

L'héritage est une propriété logique attachée à la hiérarchie entre partitions d'une part, et entre partitions et familles ou usagers (s'il s'agit de partitions usagers) d'autre part. Se pose le problème de savoir comment hériter?

Après avoir décrit le langage de définition des types structurés Famille et Usager, nous présentons l'héritage défini et les règles de recherche appliquées pour le partage des caractéristiques entre entités liées hiérarchiquement.

Nous avons mis en place deux types d'héritage :

- entre une partition et ses partitions englobantes. L'héritage est simple. Il y a raffinement successif des caractéristiques;
- entre une famille et les partitions auxquelles elle appartient. L'héritage est multiple.

Des exceptions à l'héritage sont possibles; une partition pouvant masquer les comportements hérités. Nous donnons une représentation de la hiérarchisation de l'héritage sous forme d'un graphe (figure 4.4) distinguant les cas d'application de l'héritage simple et multiple.

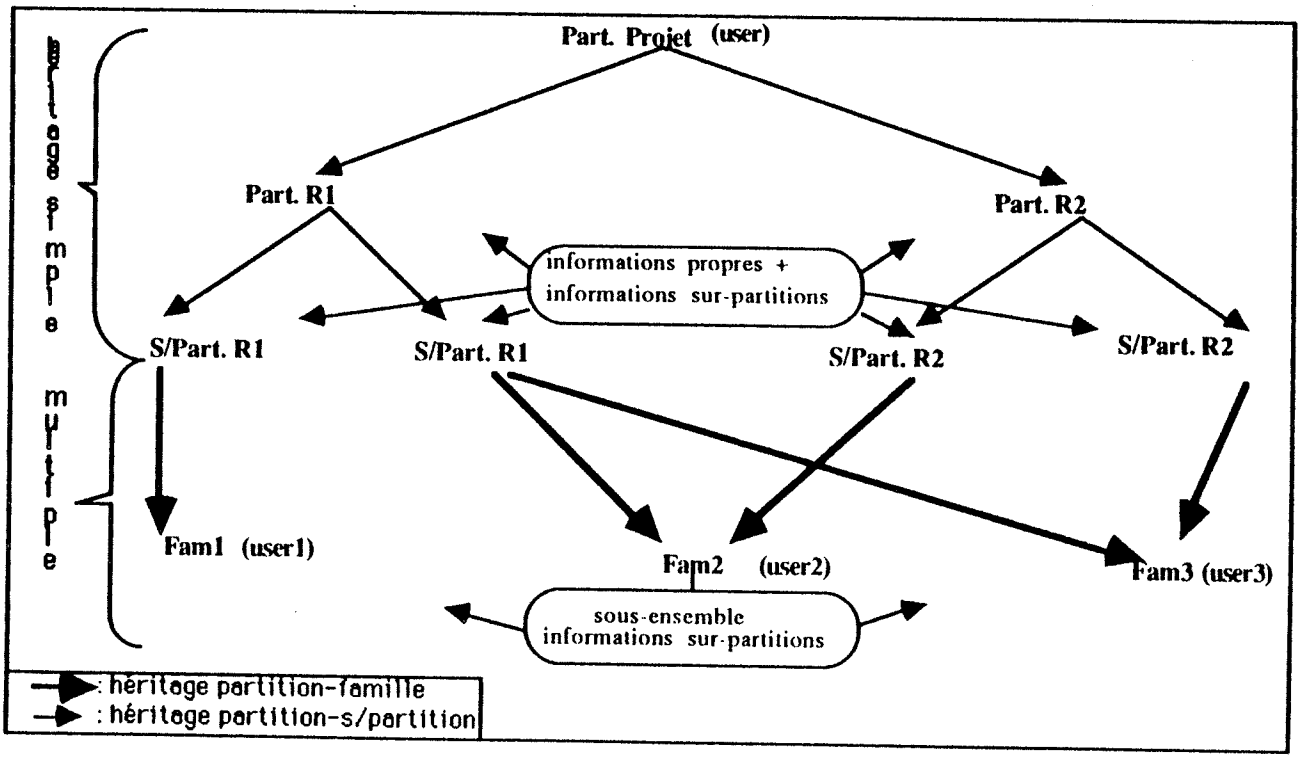


Figure 4.4 : Principe de l'héritage dans Nomade

5.2. L'héritage entre partitions

Pour une partition R.F donnée, la valeur d'une de ses clauses (attributs, relations, droits) est l'**union** (au sens ensembliste) de celles qui lui sont propres et de celles de ses partitions ancêtres. Le tableau de la figure 4.5 spécifie la sémantique de l'opérateur d'*union* pour chacune des clauses:

CLAUSES	Partition R1	Partition S/R1	Déf validesPart S/R1
ATTRIBUTS	attr1 = val1.1 attr2 = val2.1	attr1 = val1.1, val1.2 attr3 = val3.1 attr2 = val2.2	attr1 = val1.1, val1.2 attr2 = val2.1, val2.2 attr3 = val3.1
RELATIONS	S1 R1 D1 S2 R2 D2 S3 R3 D3	S1 R1 D2 S2 R3 D1	S1 R1 D1 S1 R1 D2 S2 R2 D2 S2 R3 D1 S3 R3 D3
DROITS	L1 : ExprN 1.1 L2 : ExprN2.1	L4 : ExprN4.1,ExprN4.2 L2 : ExprN2.2	L1 : ExprN1.1 L2 : ExprN2.1,ExprN2.2 L4 : ExprN4.1,ExprN4.2

Figure 4.5: Héritage partitions - sous/partition

Cette règle systèmatique d'héritage peut être modifiée pour devenir sélective. Ceci permet de contrôler l'héritage pour l'ensemble d'une clause ou individuellement pour un attribut, une relation ou un droit. La rencontre du mot clé ***stopinh*** lors d'une évaluation arrête l'héritage. Nous montrons sur des exemples le fonctionnement de l'héritage.

Soit une partition R.F1 incluant une partition R.F2 et définie de la manière suivante:

```
manuel F1
        partition R
                attributes
                        syst =          unix, vms, msdos;
                        Optcompil =     test, optimise;
                relations
                        **(syst=unix_4.2)  dep          *:*(syst=unix)
                        **.specif          specifie     : *
                rights /* si partition usager */
                        catal :         *(syst = unix*)
                        create :        *:*:* (syst = vms)
end
```

R.F2 hérite implicitement des déclarations de R.F1. Mais la partition R.F2 peut refuser l'héritage des déclarations comme elle peut ajouter de nouvelles déclarations.

Exemple 1: Soit R.F2 définie comme suit:

```
manuel F2
        partition R
                attributes
                        syst = unix*;
                relations
                        **(syst = unix_V5) dep  *:*(syst = unix)
                rights /* si partition usager */
                        catal : *(syst = msdos)
end
```

Dans R.F2:

cas des attributs :

- les attributs définis dans R.F1 sont hérités, et la valeur "unix*" est ajoutée à celles déjà définies;

- les attributs instanciés syst=vms et syst= msdos sont valides ainsi que les valeurs de syst comme "unix_5.2" et "unix_4.2".

cas des relations :

- les relations définies dans R.F1 sont héritées, et des déclarations nouvelles sont ajoutées. Les déclarations induites sont :

```
**(syst =unix_4.2) or **(syst = unix_V5)  dep          *.*(syst=unix),
**.specif                                 specifie    : *
```

cas des droits d'accès :

- les droits d'accès définis dans R.F1 sont hérités et cumulés soit :

```
catal :       * (syst=unix* ) or  * (syst =msdos),
create:       * :*:* (syst= vms)
```

Exemple 2 : Soit R.F2 défini comme suit :

```
manuel F2
        partition R
                attributes
                        syst=Unix*;
                        syst= stopinh;
                relations
                        * dep stopinh;
                rights /* si partition usager */
                        catal : *(syst=unix*) or stopinh;
        end
```

Dans R.F2 :

Cas des attributs:

- les attributs "Optcompil" sont hérités et valides,
- pour l'attribut "syst", seule la valeur "unix*" est valide.

Cas des relations:

- seule la relation "spécifie" est héritée,
- l'héritage est stoppé pour la relation "dep". Cette relation n'est pas valide.

Cas des droits d'accès

- les droits pour la commande créer sont hérités,
- l'héritage est stoppé pour la commande catal, seul les droits de catalogage des familles unix et de création des réalisations pour le système vms sont autorisés.

Exemple 3 : Soit R.F2 défini comme suit:

```
manuel F2
        partition R
                attributes
                        syst=Unix*;
                        stopinh;
                relations
                        **(syst=unix_4.2)   dep *:*(syst=unix);
                        stopinh;
                rights /* si partition usager */
                        catal : *(syst=unix*);
                        stopinh;
        end
```

Dans R.F2 :

Cas des attributs :

- les déclarations d'attributs des partitions englobantes ne sont pas héritées (indiqué par le mot clé stopinh);
- seule la déclaration d'attributs de R.F2: " syst=unix*" est valide.

Cas des relations :

- les déclarations de relations des partitions englobantes ne sont pas héritées;
- seule la déclaration de la relation "dep" dans R.F2 est valide.

Cas des droits d'accès :

- les déclarations des droits des partitions englobantes ne sont pas héritées;
- seule la déclaration de catalogage de R.F2 est valide.

5.3. L'héritage Partitions-Famille

5.3.1. Le cas d'une famille ou d'un usager appartenant à une seule partition

Dans une famille, les instanciations d'attributs, de relations et les autorisations d'exécution de commandes sont acceptées que si elles sont validées par rapport à leur définition dans la famille et la partition à laquelle le module appartient au sens de l'intersection ensembliste. Par conséquent une famille hérite des caractéristiques de la partition à laquelle elle appartient, et ne peut que restreindre les caractéristiques qu'elle hérite. Nous précisons dans le tableau de la figure 4.6 la sémantique de l'opérateur d'intersection pour chacune des clauses.

CLAUSES	Partition S/R1	Famille Fam1 de S/R1	Déf valides Fam1
ATTRIBUTS	attr1 = val1.1 attr2 = val2.1, val2.2	attr1 = val1.1, val1.2 attr3 = val3.1 attr2 = val2.2	attr1 = val1.1 attr2 = val2.2
RELATIONS	S1 R1 D1 S2 R2 D2 S3 R3 D3	S1 R1 D2 S2 R3 D1	S1 R1 (D1 and D2) (S2 and S3) R3 (D1 and D3)
DROITS	L1 : ExprN 1.1 L2 : ExprN2.1	L4 : ExprN4.1,ExprN4.2 L2 : ExprN2.2	L2 : ExprN2.1 and ExprN2.2

Figure 4.6 : Héritage partition-famille ou partition-Usager

Exemple :

Considérons une partition R.F1 définie comme suit :

```
manuel F1
        partition R
                attributes
                        syst= unix, vms, msdos

                relations
                        /dep.F/ *           dep     *:*;
        end
```

Soit une famille f appartenant à la partition R.F1 définie comme suit :

```
manuel f
        module
                attributes
                        syst=unix;
                relations
                        **.                 dep     -lstd;
                        **. (syst =unix*)   dep     :*(syst=unix)
        end
```

Cas des attributs :

- la déclaration valide pour les objets de la famille f est "syst=unix" uniquement.

Cas de relations :

- les déclarations de la famille f précisent la protection que f désire sur la visibilté de ses interfaces: l'interface f:lstd est accessible par tous les objets. Les interfaces de f possédant l'attribut "syst=unix" ne peuvent être référencées que par des versions possédant l'attribut "syst=unix". La relation devant être valide dans f et dans la partition R.F1, la protection effective sur dep est la suivante :

```
(/dep.F/ * dep      *:*)   end   (**. dep -lstd)      end
(**. (syst =unix*) dep :*(syst=unix))
```

ce qui est équivalent pour f aux déclarations suivantes :

```
/dep.F/ **.                 dep     :lstd,
/dep.F/ **. (syst=unix*)    dep     :*(syst=unix)
```

En réalité, l'accès des interfaces de f a été restreint aux familles de la partition R.F1, ce qui permet l'encapsulation.

Cas des droits d'accès:

Exemple:

Soit la partition <fonction.programmeur> définie comme suit :

```
manuel programmeur
        partition fonction
                rights
                        catal           :       not *.mandroits;
                        res_catal       :       **(elem=sourcetext);
                                                **(elem=docrev);
                        create_del      :       **(object = realization);
                        *               :       ** (author = $U\name);
        end
```

Les droits exprime qu'un programmeur ne peut pas : modifier les droits d'autres usagers, réserver ou modifier des objets élémentaires autre que les textes sources et les documents de révision, mais peut créer ou détruire des réalisations. Cependant il possède tous les droits sur ses objets.

Soit un programmeur Lambda appartenant à la partition /fonction.programmeur/ définie précédemment et dont les droits d'accès sont les suivants:

```
manuel Lambda
        user
                rights
                        *               :       /dep.F/ ** (syst = vms);
end
```

Les droits précisent les parties du logiciel sur lesquelles le programmeur Lambda est autorisé à travailler en l'occurence la partition dep.F. Le type de commande autorisé est précisé dans sa partition fonction qui définit le profil programmeur.

5.3.2. Le cas d'une famille ou d'un usager appartenant à plusieurs partitions

L'instanciation d'un attribut ou l'exécution d'une commande sont acceptés s'ils sont valides par rapport à la famille ou l'usager selon le cas et par rapport à l'une au moins des partitions auxquelles ils appartiennent.

Une famille ou un usager hérite donc des caractéristiques des partitions auxquelles ils appartiennent. La famille ou l'usager ne peut que restreindre l'**union** des caractéristiques qu'ils héritent. Seuls les attributs et les droits d'accès sont susceptibles d'avoir des héritages multiples. En effet pour éviter les conflits de noms, une relation ne peut être définie que dans les partitions associées à une même relation structurante. Par conséquent l'héritage des relations se ramène à un cas d'héritage simple.

Cas des attributs :

Exemple :

Soit la famille F appartenant aux partitions R1.F1 et R2.F2 définies comme suit :

```
manuel F1
partition R1
        attributes
                syst=unix;
end
```

```
manuel F2
partition R2
        attributes
                syst=vms;
end
```

Dans F, la définition d'attribut valide est syst= unix, vms

Cas des droits d'accès:

Exemple :

Supposons que l'usager Lambda décrit précédemment appartienne également à la partition **spécifieur.équipe**.

```
manuel fonction
partition programmeur
rights
catal       : not *.mandroits;
res_catal   : ** (elem=sourcetext),
              **(elem=docrev);
create_del :** (object = realization);
*           : ** (author = $U\name);
end
```

```
manuel fonction
 partition spécifieur
rights
catal : not *.mandroits;
create_del : ** (object=interface);
* : ** (author= $U\name);
end
```

```
manuel Lambda
user
rights
*   :       /dep.F/ **
end
```

L'usager Lambda acquiert en plus des droits des programmeurs, les droits des spécifieurs. Mais la partie du logiciel sur laquelle il peut travailler reste limiter à la partition dep.F.

Un usager peut appartenir à plusieurs partitions simultanément, ses droits subissent donc un héritage multiple comme nous venons de le voir sur l'exemple précédent. Les droits effectifs d'un usager sont donc l'union des droits des partitions auxquelles il appartient. Le fait, pour un usager, d'entrer dans une nouvelle partition a pour conséquence de lui affecter les droits de cette nouvelle partition usager. C'est pourquoi la création d'une relation faisant entrer un usager dans une nouvelle partition est considérée comme un ajout de droits et n'est acceptée que si les droits de l'usager créant la relation englobent les droits contenus dans la partition.

Finalement il est à remarquer que la définition de la partition (cf §3.7.2) implique qu'une famille f entre dans une partition R.F dès qu'une relation R est créée entre une famille de R.F et f; inversement cette famille f peut sortir de la partition R.F si une autre relation R est créée entre une famille n'appartenant pas à R.F et f. La stabilité d'une partition impose un contrôle rigoureux des relations structurantes R permises.

6. INITIALISATION DE LA BASE

A l'initialisation de la base, une famille "**project**" est créée implicitement et possède le statut implicite de racine de toutes les partitions. Par conséquent toutes les partitions sont déclarées dans la famille "project" et définissent toutes les relations structurantes possibles. Par défaut une seule relation structurante notée "**struct**" est définie et permet de lier tous les usagers et familles d'un projet. Les attributs et les relations prédéfinies sont indiqués dans la partition "struct". Trois autres familles sont créées :

- une famille du nom de la base (*basename* sur le schéma) racine de toutes les familles logiciel redéfinissant la partition *struct* avec des caractéristiques propres au logiciel,
- un usager *user* redéfinissant la partition *struct* avec des caractéristiques propres aux usagers,
- un usager Administateur de la base initialisé avec le nom du créateur de la base (*Superuser* sur le schéma).

Cette structure est représentée dans la figure 4.7.

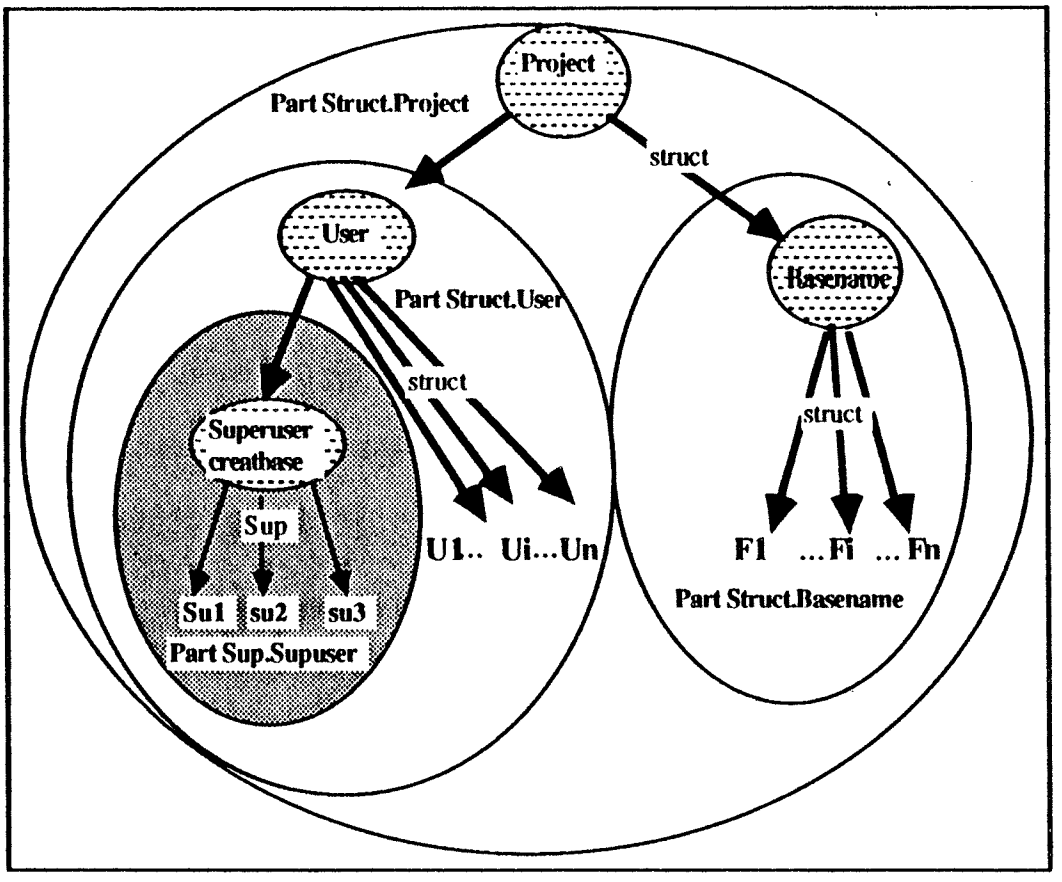


Figure 4.7 : Structure mise en place à l'initialisation de la base

Les types structurés définis implicitement pour chacune des familles et partitions sont initialisés lors de la création de la base de la manière suivante:

```
manuel project
    partition struct
/* définition de la partition "struct" qui définit tout ce qui est commun aux usagers et
familles*/
    attributes
          def **
                partition = *;
                family=*;
                interface = *;
                realization = *
                name = *;
                author = *;
                object= body, realization, interface, conf, family, user;
                elem= sourcetext, objcode, Revdoc, doc, man, hist, manattr, mansel,
                manaction, manpart,manother;
                langage = *;
                state = *;
                stateconf=*;
    relations
    /* la relation "struct" liant la famille basename et l'usager user à la famille projet */
                project        struct (Basename or User);
    /* définition de la relation de dépendance */
                **.            dep     * :*;
    /* définition de la relation structurante "sup" pour organiser les administrateurs */
                superuser      sup   * (object =user);
                ...        /* (toute relation structurante introduite dans Nomade) */
end;
```

```
manuel Basename
    partition struct
/* redéfinition de la partition struct à partir de la famille Basename définissant tout ce qui est
commun aux familles logiciel */
    attributes
                object = body, realization, interface, conf, family;
                object = stopinh;
```

```
    relations
        /* définition de la relation prédéfinie "struct" liant toute famille à la famille
        Basename*/
            Basename    struct      * ;
            *           struct      stopinh;
        /* définition de la relation de composition associée à la relation dep entre les
        configurations et les interfaces et réalisations composant la configuration */
            **.(object = conf) comp    **.;
end;
```

```
manuel User
    partition struct
/* redéfinition de la partition struct à partir de la famille User définissant tout ce qui est
commun aux usagers*/
    attributes
        def **
            name = *;
            partition = *;
            family =*;
            author =*;
            elem= doc, man, hist, manattr, manaction, manrights, manpart;
            object = user;
            stopinh;
    relations
/* redéclaration de la relation prédéfinie "struct" liant les usagers à la famille User */
            User   struct      * ;
            *      struct      stopinh;
    rights
/* droits des usagers sur les objets qui leur appartiennent */
            * : ** ( author = $U\name);
end;
```

Le créateur du projet devient son administrateur (racine de la partition Sup) doté de tous les privilèges sur la base. Sa clause de droits est initialisée de manière à lui donner un contrôle total sur le projet.

```
manuel nom-usager (Administrateur)
    partition sup
        rights
            *: **;
        relations
            Superuser Sup * ( object= user)
end;
```

6.1. Exemple

Nous présentons un exemple d'adaptation de la structure d'initialisation pour prendre en compte l'organisation des usagers dans le système Unix comme le montre la figure 4.8. Nous décrivons les grandes lignes du schéma de protection des fichiers à mettre en place pour exprimer les droits d'accès tels que définis dans le système Unix .

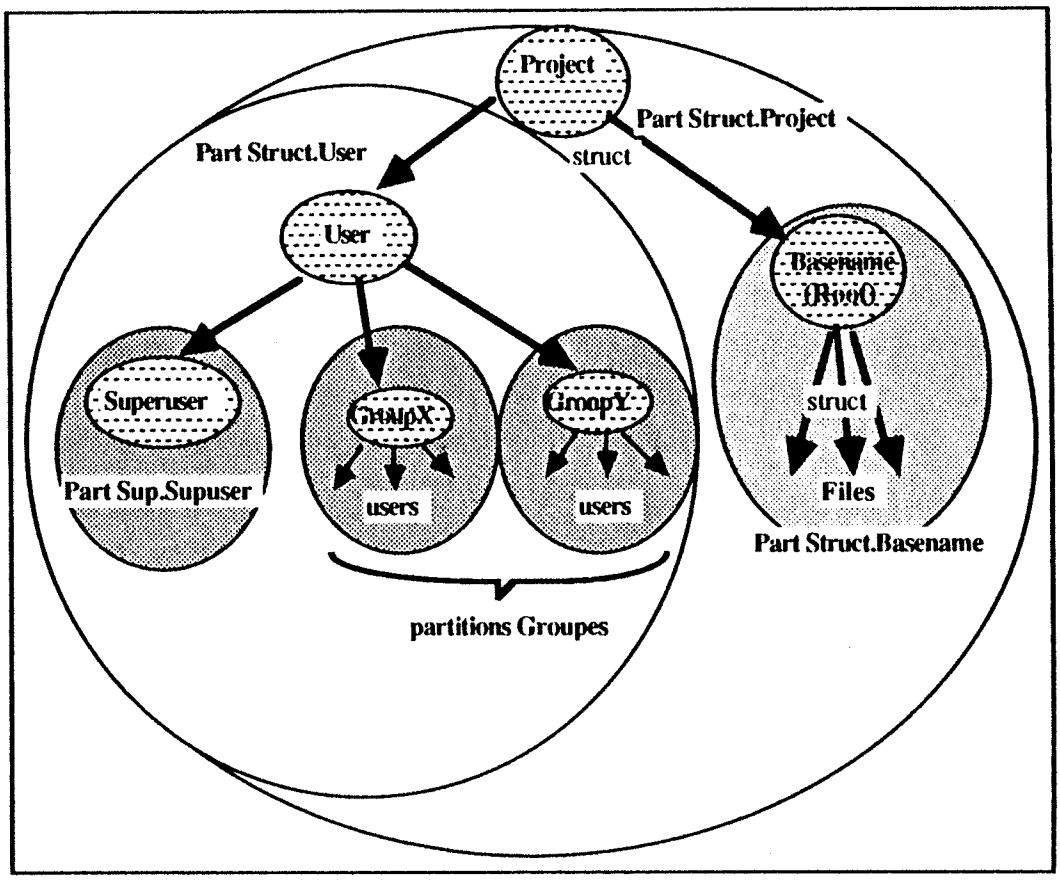


Figure 4.8: Adaptation en Nomade du schéma de protection Unix

A chaque objet dans Nomade sont associés les attributs suivants:

- 3 attributs avec les valeurs possibles correspondant au droits de lecture d'écriture et d'exécution du système Unix :

```
User-protection     = R, W, X
Group-protection    = R, W, X
Other-protection    = R, W, X
```

- l'attribut "author" définissant le propriétaire de l'objet
- l'attribut "group" définissant le groupe.

Cette structure est définie comme suit dans Nomade:

```
manuel basename (root)
    partition struct
        attributes
            User-protection     = R,W, X
            Group-protection    = R, W, X
            Other-protection    = R,W, X.
            author              = *
            groupe              = *
            ...
end
```

Nous définissons une partition groupe regroupant l'ensemble des usagers et se spécialisant en différents sous-groupes pouvant être créés progressivement. Nous associons à la partition groupe les trois types de privilèges étiquetés R,W,X dans la clause "rights" correspondant aux droits standards d'Unix pour la lecture, l'écriture et l'exécution d'un fichier et un privilège particulier PRIV-CHMOD qui définit la contrainte de modification des droits d'un fichier et un attribut "group-name" définissant le nom du groupe de tout usager.

Ces priviléges sont initialisés comme suit dans la famille user de la partition groupe regroupant tous les usagers :

```
manuel user
        partition groupe
            attributes
                group-name = *;
            rights
                R : ** (author = $U\name and User-protection= R) or
                **(group= $U\group-name and Group-protection =R) or **(
                Other-protection =R)

                W : ** (author = $U\name and User-protection= W) or
                **(group= $U\group-name and Group-protection =W) or **(
                Other-protection =W)

                X : ** (author = $U\name and User-protection= X) or
                **(group= $U\group-name and Group-protection =X) or **(
                Other-protection =R)
                  /* privilége associé à la commande chmod */

                PRIV-CHMOD : **( author = $U\name)
                stopinh;

            relations
                  /*relation structurante de définition de partitions groupes */
                  *(object = user) groupe * (object =user)
end
```

```
manuel group X
    /* créé à chaque définition de groupe par spécialisation de la partition groupe */
            partition groupe
                    attributes
    /* valeur du nom de groupe imposée et héritée (==) par tous les usagers du groupe */
                            group-name == nom de groupe ;
                            ...
end
```

La vérification des droits s'éffectue avant l'exécution de chaque commande par la fonction *checkright*, soit :

- checkright (object, R) pour toute commande de lecture de l'objet;
- checkright (object, W) pour toute commande d'écriture de l'objet;
- checkright (object, X) pour toute commande d'exécution de l'objet;
- checkright (object, PRIV-CHMOD) pour la commande "chmod".

La programmation de la commande "chmod" d'Unix en Nomade permet la modification des trois attributs User-protection, Group-protection et Other-protection.
Le privilège **PRIV-CHMOD** permet de vérifier avant la modification des attributs de protection si l'objet appartient effectivement à l'usager en cours.

7. CONCLUSION

Nous avons présenté dans ce chapitre un modèle de structuration du logiciel applicable dans un contexte de programmation globale. La définition d'un modèle général et adaptable, intégrant données et utilisateurs est une des contributions significatives de ce travail. Dans la description qui vient d'être faite, on a associé à des modèles syntaxiques prédéfinis (famille et usager), des mécanismes pour adapter le schéma de la base à l'organisation réelle dans laquelle le noyau est utilisé. La base conceptuelle de ces mécanismes a été calquée sur les problèmes réels de la programmation globale, dégagés de l'expérimentation d'Adèle. Le modèle repose sur les concepts suivants:

- les attributs et les relations constituent les concepts de base pour modèliser les informations contextuelles, c'est à dire liées à l'environnement matériel et humain dans lequel le noyau est utilisé.

- les droits d'accès reposant sur les usagers et la désignation, permettent la définition d'un schéma de protection propre à chaque organisation,

- la notion de partition s'appuie sur les relations structurantes et les entités familles et usagers. La notion de partition permet de définir des classes de modules et d'usagers, de préciser les règles d'héritage entre partitions, et de contrôler la structure des logiciels.

Il reste à présenter les aspects dynamiques du modèle développés à partir du mécanisme d'événements-actions avec la partie du langage permettant de définir les actions et la façon de les interpréter.

CHAPITRE 5 :

UN MECANISME D'EVENEMENTS-ACTIONS POUR UN CONTROLE DES ACTIVITES DE PROGRAMMATION GLOBALE

1. INTRODUCTION

Il est usuel de distinguer deux aspects dans un environnement : la base d'objets et les opérations pouvant s'effectuer sur les objets. Le premier aspect correspond au modèle de structure présenté précédemment, le second au modèle de comportement abordé dans ce chapitre et permettant de compléter la description du logiciel par la définition des actions susceptibles d'être effectuées sur les objets. Ces actions vont permettre la gestion des tâches de programmation globale. L'objectif est de développer des mécanismes qui permettent d'automatiser la gestion de l'évolution de gros logiciels complexes en respectant les méthodes, les habitudes et les conventions des équipes qui en sont chargées.

Le besoin se fait donc sentir de disposer d'une part d'un noyau regroupant toutes les informations produites et utilisées par l'environnement, et d'autre part de fonctionnalités permettant de rendre ce noyau facilement programmable et extensible afin de définir simplement diverses stratégies de gestion en fonction du site d'utilisation. Ces fonctionnalités sont supportées par un gestionnaire d'activité qui s'apparente aux "triggers" et "démons" mis en évidence respectivement dans le domaine des bases de données et de l'intelligence artificielle.

L'une des premières utilisations de ce gestionnaire concerne la définition d'opérations sur les objets, la définition et la vérification de contraintes de cohérence complexes, l'activation automatique et le contrôle des outils; les relations étant utilisées pour calculer la propagation des modifications. Le développement de ce gestionnaire d'activité est basé sur un mécanisme d'événements-actions [Belkhatir87].

Après une discussion sur les principes du gestionnaire d'activité, ce chapitre présente le mécanisme d'événements-actions proposé et discute de son utilisation et de son intérêt en montrant, par des exemples, comment peuvent être implantées simplement les diverses stratégies habituelles de programmation globale.

2. PRINCIPES DE L'ACTIVATION

2.1. Motivations

Le développement et la maintenance de logiciel est un processus continuel de changements et d'évolutions mettant en oeuvre différents types de tâches: compiler les sources, construire des versions cohérentes de configuration, développer de nouvelles versions, mettre à jour la documentation associée...). Durant ces développements, l'état de certains objets n'est plus cohérent à cause de leur forte interdépendance avec les objets modifiés. Les contraintes à satisfaire dans ce cas sont complexes, de différentes natures et nécessitent que des actions soient exécutées et que des outils de l'environnement soient activés. Ces activités sont largement répétitives et obéissent généralement à des procédures et conventions concernant les tâches à valider. Ces procédures sont le plus souvent spécifiques à toute organisation de projet. La gestion manuelle de ces activités pour le développement d'un logiciel de taille importante est fastidieuse, génératrice d'erreurs et entraine la multiplication des efforts et par conséquent une dépense considérable d'énergie, chacun des usagers développant ses propres descriptions de tâches en l'absence d'un outil faisant coopérer les développements.

Pour être libérés de ces tâches, les usagers ont besoin de mécanismes qui permettent de les assister durant le développement, par exemple avoir une évaluation des objets atteints par une modification, réagir à la violation de contraintes d'intégrité et à des accès non autorisés ou erronés. Ces possibilités sont fournies par des mécanismes qui permettent de définir des actions à entreprendre sur certains événements (erreur de manipulation, modification inhibée, violation de droits d'accès..) pendant le déroulement des opérations. L'impossibilité de prévoir toutes les utilisations possibles nous amène à l'idée qu'il faudrait fournir un mécanisme permettant de définir explicitement ces actions et de les adapter aux besoins des usagers. Les actions sont des entités actives qui ont besoin d'un certain nombre d'objets pour s'exécuter. Ce mécanisme peut être défini efficacement s'il est construit à partir d'une base d'objets. La propriété de description des objets et des actions peut être résolue par la définition d'un schéma de donnée approprié et celle de l'activation par un mécanisme d'événements-actions.

Ainsi il est possible de clarifier les rapports qui existent entre le noyau de l'environnement et l'usager de cet environnement. L'usager peut décrire les objets manipulés et les actions qu'il désire avec les outils à appliquer pour régir leur manipulation et le noyau gére les objets et contrôle l'activation des actions.

2.2. Descriptions séparées de la structure et des activités logiciel

L'approche adoptée dans Nomade consiste à définir un schéma regroupant des entités décrites séparément notamment celle de la structure et des activités(dans notre terminologie actions)

logiciel. L'approche décrivant séparément les deux entités présente plusieurs avantages par rapport à celle, caractéristique de l'outil Make, mixant les deux . On peut les résumer comme suit :

- chaque abstraction est à son niveau approprié. Dans les approches mixant structure et actions, il apparaît une certaine confusion pour l'usager entre les objets ayant un rapport avec la description de la structure et ceux (intermédiaires) qui sont nécessaires uniquement au déroulement de l'action. Par ailleurs, la description des actions se fait pour l'activation d'outils, à des niveaux de détails (comme par exemple les options) qui sont pertinents pour l'action en question mais qui ne doivent pas interférer avec la structure du logiciel.

- Il est possible de réduire la description redondante des actions. Par exemple les actions d'impression du contenu d'un objet et de compilation d'un texte source sont définies une seule fois, paramétrées par les types d'objets en entrée et sortie. Lors de l'appel d'une action les paramètres sont substitués par les valeurs réelles puis l'exécution est poursuivie.

- Un des avantages majeurs réside dans le fait que de nouvelles actions peuvent être définies sans pour cela porter atteinte à la structure du logiciel.

2.3. Le contrôle des activités logiciel et le traitement des effets propagés

L'approche généralement utilisée pour l'activation des tâches et basée sur l'estampillage des objets. Le système exécute, sur la base des dates de modification, seulement les étapes nécessaires pour rétablir la cohérence des objets. C'est l'approche Make qui présente énormément d'efficacité mais qui est approximative dans la mesure où il est possible de modifier l'estampille d'un objet sans que l'objet change réellement. Une approche plus sûre, tirant avantage de la base d'objets, permet d'enregistrer les renseignements de modification d'un objet, ce qui apporte plus de possibilité. Ceci permet de mieux cerner les changements apportés et de réagir automatiquement en exécutant les actions appropriées. Cette dernière approche a été systématisée dans Nomade. Ainsi, la base se comporte comme un mécanisme de base qui permet de définir les relations pertinentes entre les objets, et de déclencher des actions suite aux événements définis par l'usager. Chaque usager ou groupe d'usager peut définir les événements et des actions réponses différentes tout en utilisant la même base.

Les relations jouent un rôle capital dans ce mécanisme. Sans relation, il n'y a pas d'événements et par conséquent pas d'actions. Chaque relation définit une étape de propagation. Utilisée récursivement, elle définit une manière de propager les changements. Le noyau Nomade peut vérifier ainsi qu'un objet est créé ou modifié conformément aux règles de production et déterminer les objets atteints par la propagation des changements en utilisant les propriétés et les relations attachées à chaque objet.

3. LE MECANISME D'EVENEMENTS-ACTIONS

3.1. Présentation générale

Le concept de déclencheurs (Triggers) a été développé dans les bases de données. Un déclencheur est une condition portant sur le changement d'état d'un objet permettant de déclencher l'exécution d'une action. La condition peut être nommée et lorsqu'elle est satisfaite, elle produit un événement qui déclenche une action. La condition peut être satisfaite après l'exécution d'une opération sur un objet (déclencheur externe) ou après l'exécution d'actions associées à la condition (déclencheur interne entrainant la propagation). Ce mécanisme est particulièrement adapté au contrôle de la cohérence puisqu'il permet de définir les opérations qui d'une manière conditionnelle ou non succédent à des opérations sur la base. Le gestionnaire d'activité introduit dans Nomade est basé sur un mécanisme d'**événements-actions** inspiré des déclencheurs et que nous avons adapté à notre contexte.

Le mécanisme d'événements-actions définit la notion d'événements qui peuvent être produits explicitement et qui entrainent le déclenchement d'actions usagers.

Le modèle retenu conduit à une représentation de la dynamique (structure de comportement) de l'environnement sur la base des événements. Les événements déclenchent des actions qui modifient les objets et les relations; certains changements d'états produisent à leur tour des événements. La dynamique est représentée par l'interconnexion de 3 entités : les objets et leurs relations, les événements et les actions. Nous proposons une description graphique du mécanisme montrée sur la figure 5.1.

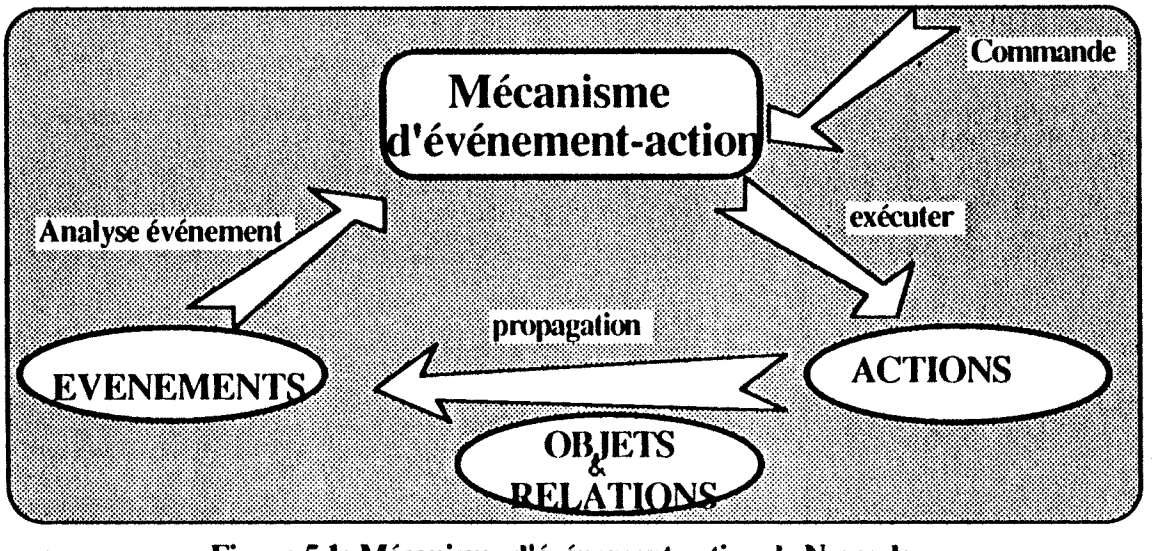


Figure 5.1: Mécanisme d'événement-action de Nomade

Nous allons essayer de voir plus en détail ce qu'est le mécanisme d'événements-actions, c'est à dire de définir avec précision les différents composants, les primitives permettant d'exprimer l'association événement-action et son application à la gestion des activités logiciel.

3.2. Les différents composants du mécanisme et les primitives associées

3.2.1. Les événements

Les relations jouent un rôle important dans Nomade. Elles permettent de structurer le logiciel ou de représenter des faits; elles peuvent être utilisées pour informer sur les objets affectés par un changement et rendre possible le traitement des propagations.
Partant de cette constatation, un événement dans Nomade représente le changement remarquable de l'état d'un objet qui rend invalide un ou plusieurs objets qui en dépendent. L'événement est défini par référence aux objets et aux relations dont il est la cible. Les conditions dans lesquelles surviennent les événements sont connues des responsables de projet qui peuvent par conséquent programmer explicitement leur production. Notre hypothèse de base est la suivante: si aucune relation n'existe entre deux objets, un changement de l'un des objets n'affecte pas l'autre objet. C'est le mécanisme de base utilisé dans Nomade pour produire les événements et contrôler les phénomènes de propagation. La relation est considérée comme l'élément conducteur, le canal par lequel sont acheminés les effets propagés. S'il existe une relation R entre un élément source S et destination D et si D ou R est modifié, il peut y avoir des effets sur S. Un événement E peut être produit sur demande de l'usager entrainant l'exécution d'une action associée à S.
Dans Nomade, la primitive permettant de produire un événement est la suivante:

propagate (S, R, D, C)

Cela signifie qu'une production d'événement, demandée par l'usager, a lieu s'il existe une relation S R D. Le schéma de la figure 5.2 montre les différents paramètres intervenant dans la production d'un événement.

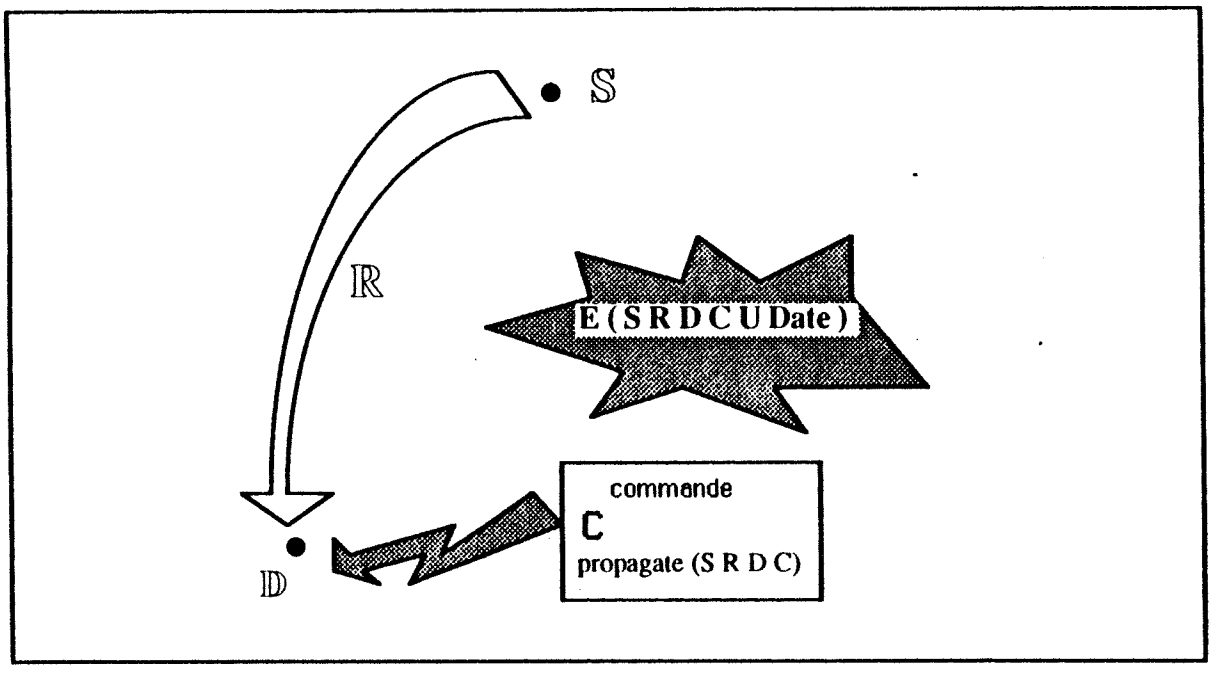


Figure 5.2 : Schéma de production d'événements

L' état d'un événement produit est défini par les informations suivantes :

E (S, R, D, C, U, Date)

- **S** est le nom de l'objet source de la relation **R**
- **C** est un paramètre par convention le nom de la commande informant de l'action sur **D** ayant produit l'événement (destruction ou modification ou tout autre action définie par l'usager) ,
- **U** est le nom de l'usager déclenchant l'événement,
- **Date** l'instant où l'événement a été déclenché.

Exemple :

Si une relation de dépendance (R=dep) existe entre un programme (S=P) et une interface (D=I), une production d'événement suite à la modification d'une l'interface I par un usager Lambda peut être produite par la fonction:

```
propagate ( P, "dep", I, "modifier")
```

L'état de cet événement est défini comme suit :

```
E(P,dep, I, modifier, Lambda, date_modification)
```

Des actions associées à S et définies par l'usager expriment comment réagir lorsqu'un événement survient. Le mécanisme d'événements-actions a en charge la détermination de l'action à exécuter sur l'objet S. Dans l'exemple précédent , il faudrait spécifier une action de recompilation de P associée à l'événement propagé sur P suite à la modification de l'interface I.

La définition de la notion d'événement propose les intervenants :

- l'action qui provoque l'événement,
- les relations ayant cet objet pour cible,
- les objets source de ces relations atteints par les effets propagés,
- Enfin, éventuellement une ou des actions en réaction à l'événement; c'est l'action réponse à l'effet propagé.

Cette définition assez générale doit être précisée :

- Quelles sont les relations mises en oeuvre?
- Quelles actions exécuter et quand les exécuter ?

3.2.2. Les relations utilisées pour la propagation d'événements

La relation mise en oeuvre dans les environnements est le plus souvent la relation de dépendance qui peut être explicitement fournie ou directement extraite des textes source grâce à un compilateur (Gandalf, Cedar), ou grâce à un analyseur ad-hoc (Nomade). Deux systèmes, Domain et Nomade, ont étendu ce concept, permettant ainsi de définir d'autres types de relations entre objets. Dans Nomade, toute relation peut être fournie explicitement ("dépend_de" entre familles, "spécifie" entre un document et une famille, "activités" entre un programmeur et un planning...). Ces relations permettent de tenir compte de certaines dépendances d'ordre sémantiques, même si la granularité des objets est importante dans Nomade (en pratique : un fichier).

3.2.3. Les actions

La généralisation des relations permet de définir des actions autre que l'unique action de compilation associée à la relation de dépendance. Les actions peuvent être définies dans des contextes aussi différents que la gestion de projet, le contrôle de cohérence, la production...

Dans Nomade les actions peuvent être indéxées par les informations définissant l'événement à savoir S, R, D, C, U, Date et substituées par leur valeurs respectives avant l'exécution de l'action. Ceci permet de donner plus de flexibilité aux programmes d'actions puisqu'il est possible de définir des actions appliquables à plusieurs objets. La figure 5.3 montre des modèles d'actions de compilation et d'impression indéxés par le type d'objet. Le langage de support à la description dans l'exemple est le langage de commandes *Shell* d'Unix.

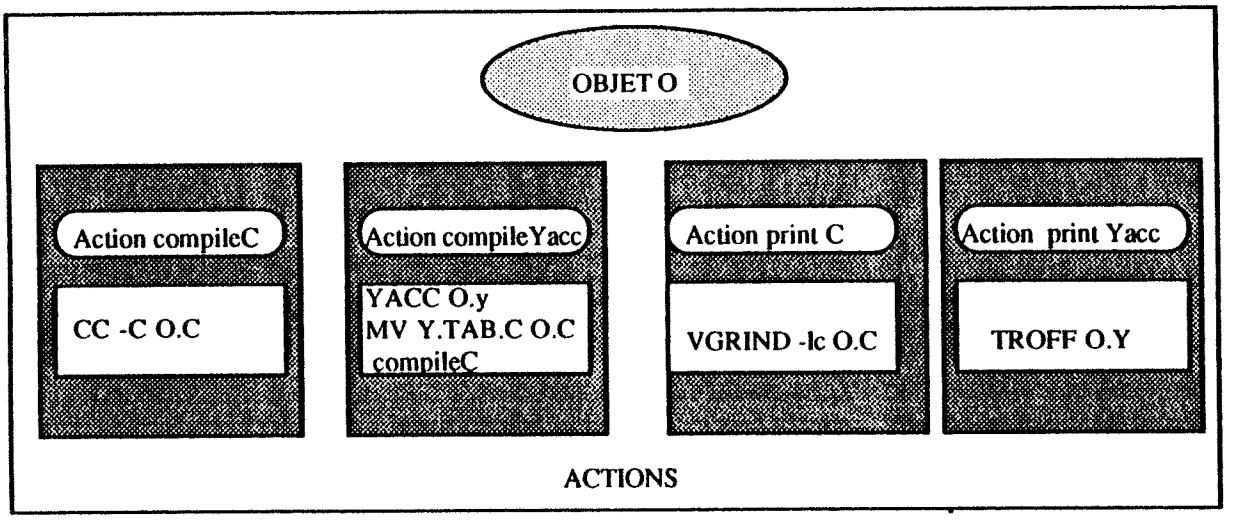


Figure 5.3 : Exemple d'actions associées à un objet

Nous présentons dans ce qui suit notre approche pour traiter les problèmes de l'écriture, de l'exécution et du stockage des actions.

a) Ecriture

Nous avons voulu dans une première phase valider les idées avancées aussi le développement d'un langage adapté à l'écriture des actions n'a pas été une priorité. Pour cet aspect, nous nous sommes contentés de tirer avantage des possibilités très riches existant sur le système d'exploitation hôte Unix notamment la puissance de son langage de commandes *shell* pour l'écriture de programmes d'actions simples et des langages de programmation classiques pour l'écriture de programmes d'actions élaborées en permettant l'accès à la base via une interface programmable.

b) Exécution

Comme la plupart des systèmes d'exploitation ne permettent pas l'édition de liens dynamique notamment dans notre cas le système Unix, nous avons distingué plusieurs possibilités dans l'exécution des programmes d'actions. La figure 5.4 résume ces différentes possibilités d'exécution.

L'action est décrite par une séquence de fonctions. Ces dernières pouvant être :

- des soumissions de commandes à la base d'objets. La fonction qui permet d'exécuter une commande Nomade sur la base d'objets en interne est définie comme suit :

exec(nomade-commande paramètres);

- des programmes usagers écrits dans un langage classique (C, pascal) pour décrire des traitements élaborés. Comme alternative à l'édition de liens dynamique, nous permettons aux actions écrites dans un langage classique d'être stockées sous forme exécutable et d'être

appelées par la fonction : *rexec*. Une communication est établie entre le programme d'action est le gestionnaire de la base d'objets pour la soumission de commandes à la base et la reception des résultats via une interface programmable. Par aillleurs comme les architectures types dans ces applications sont constituées d'un site central ou serveur de base d'objets et d'un réseau de stations locales, nous avons généralisé le mécanisme de communication pour prendre en compte cette répartition et permettre ainsi l'exécution d'actions sur des sites distants. Cet aspect est décrit dans le chapitre 6. Il faut pouvoir indiquer dans quel environnement le processus "action" doit s'exécuter. Ceci est effectué par la fonction rexec définie comme suit:

rexec (site : programmes arguments).

où *site* désigne le site de résidence et d'exécution des actions.

- des appels d'outils disponibles sur la machine hôte comme les compilateurs, editeurs, formatteurs via l'interpréteur de commandes. La fonction qui permet d'appeler l'interpréteur de commandes du système d'exploitation (dans notre cas le *shell* du système *unix)* , soit :

system (chaine).

Nous avons introduit cette possibilité du fait de la puissance du *Shell d'Unix* en tant que langage de programmation et langage de commande. Ces commandes sont exécutées par un processus *Shell Unix* indépendamment de la base, par conséquent sans établir la communication avec la base d'objets. Ce cas est distingué du précédent pour des raisons d'efficacité, plus précisemment pour faire une économie d'une communication qui aurait été inutile.

- des fonctions pour le contrôle du programme d'actions. Nous présentons les principales qui vont permettre la compréhension des exemples qui vont suivre:

Exemple de fonctions :

- propagate (S, R, D, C) : produit l'événement E(S, R, D, C...);
- display ("message") : affiche le message à l'écran;
- displayconf ("message") : affiche message avec demande de confirmation à l'usager. Selon la réponse de l'utilisateur, un booléen est positionné;
- record : enregistre dans S l'événement courant;
- reset (S, R, D, C) supprime l'événement correspondant de la liste des événements à traiter;
- fail : cette fonction est utilisée dans les post-actions et entraine le refus de la commande;
- exit : sortie immédiate du programme d'action.

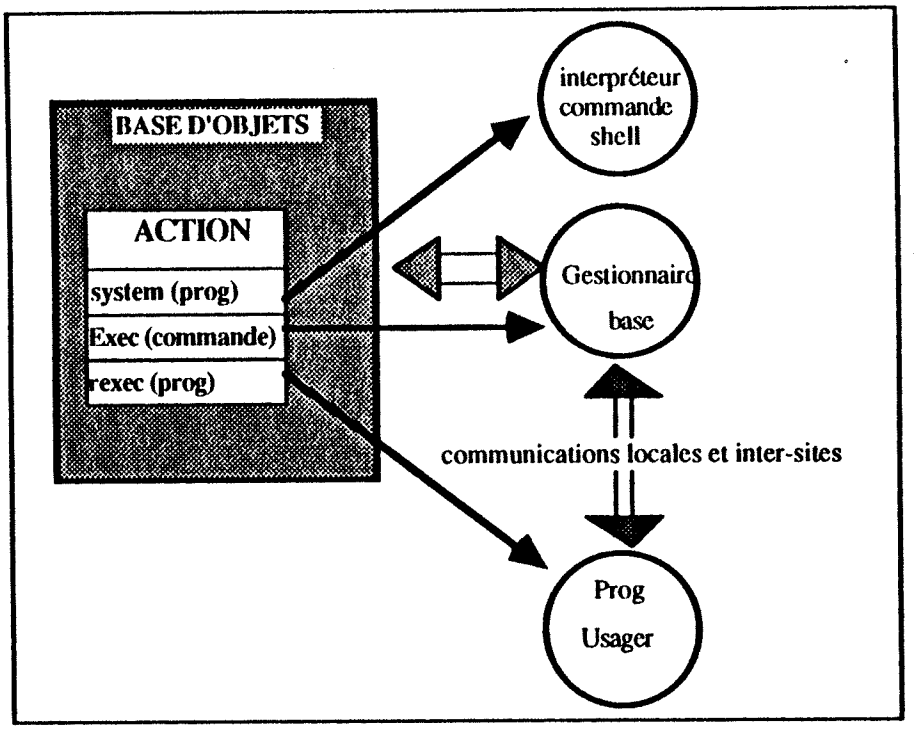


Figure 5.4 : Récapitulation des différents modes d'exécution des actions

c) Stockage

Les actions sont définies dans les manuels des objets auxquels elles s'appliquent. Par contre les programmes mis en oeuvre par les actions que ce soit les fichiers de commandes pour l'interpréteur de commandes ou les programmes sous forme exécutable, peuvent être stockés chez l'usager ou insérés dans la base et stockés sous forme d'objets de type document de manière totalement indépendante du mécanisme d'événements-actions.

Après avoir défini les entités Evénements et Actions nous décrivons la manière dont s'exprime leur association.

3.3. Formulation de l'association événement-action

Le couple (E: événement, A: action) traduit le fait que le déclenchement de l'action A est provoqué par l'événement E(S, R, D, C, U, Date). Cette association est exprimée dans la clause *actions* du manuel de l'objet S, source de la relation R comme suit :

For R (C) do action done;

où R(C) constitue un filtre portant sur le nom de la relation R; le paramètre C décrivant l'action ayant produit l'événement (par exemple le nom de la commande).

Ceci exprime que pour un événement E(S, R, D, C, U, Date), ou les paramètres ont été substitués par les valeurs réelles, il faut rechercher, dans le manuel de l'objet S une action : *For R(C)....* comme le montre la figure 5.5.

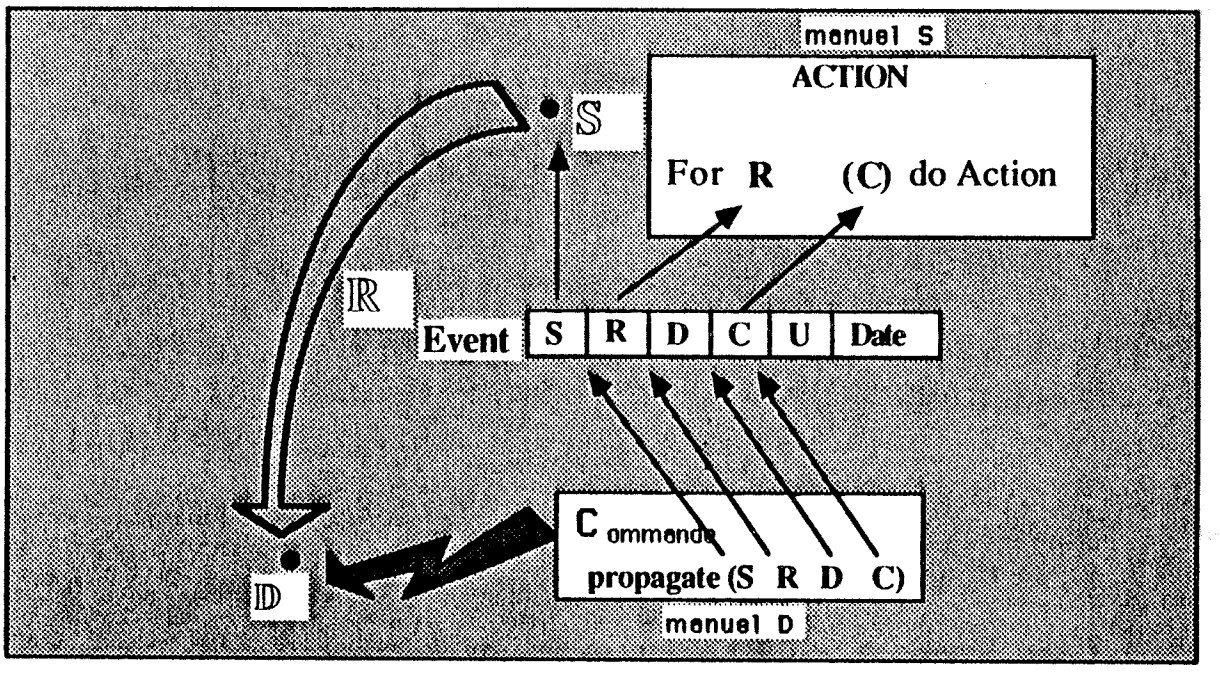


Figure 5.5 : Schéma d'association événement-action

Exemple :

Reprenons l'exemple précédent de modification d'une interface, l'association événement-action peut être définie de la manière suivante dans le manuel de la famille de la réalisation P :

```
manuel P
actions
        for dep(modifier) do exec(compile $S) done;
end
```

compile est le nom d'une commande Nomade effectuant la compilation de P.

Commentaire sur le déroulement:

Supposons qu'une commande "modifier I" soit exécutée par un usager U. La commande "modifier" va exécuter "propagate (* ,dep, I, "modifier") qui produit entre autres l'événement E(S = P, R = dep , D =I , C = modifier...) . Une action de type : for dep (modifier) est recherchée dans le manuel de P. Les paramètres de l'événement, utilisés dans l'écriture du programme d'actions sont substitués par leurs valeurs respectives et le programme d'action prend la forme suivante avant son interprétation:

`for dep(modifier) do exec (compil P) done;` ce qui permet de compiler P

3.3.1. Les conditions sur les actions

Nomade permet le déclenchement conditionnel des actions. Avec l'introduction des conditions, l'association événement/action prend la forme suivante:

```
for R(C) do
        if condition then action fi;
        ...
done;
```

Cette condition peut être décrite par une expression Nomade ou une fonction du programme d'action retournant une valeur booléenne. En effet, toutes les fonctions Nomade retournent un booléen positionné à "vrai" si elle se déroule correctement, "faux" sinon;

Exemple :

Nous reprenons l'exemple précédent et modifions la stratégie de recompilation définie précédemment suite à une modification d'interface. Le programme d'action est défini comme suit :

```
manuel P
for dep (modifer) do
if not exec (compile $S) then record; exec (attr $S state "interface-modifiée");
fi;
done
end
```

où

- dep (valeur du paramètre R) est la relation de dépendance entre les réalisations (S) et les interfaces (D) qu'elles utilisent,
- modifier (valeur du paramètre C) qui informe de l'action effectuée sur l'interface,
- compile est une commande définie par l'usager qui compile un programme.
- attr est une commande Nomade qui affecte à l'attribut état de S la valeur "interface-modifiée".

Ces actions signifient que chaque fois qu'une interface est modifiée, les réalisations qui l'utilisent sont compilées; si la compilation échoue, l'événement est enregistré et l'attribut état de S prend la valeur "interface-modifiée".

4. LES DIFFERENTES FORMES DE DECLENCHEMENT D'ACTIONS

Soit un événement E(S, R, D, C, U, Date) , les actions associées peuvent être déclenchées de cinq manières différentes. On distingue :

- les **commandes** qui sont les commandes usagers classiques et les commandes de la base. Le mécanisme d'événement/action permet aux usagers de définir des commandes plus élaborées à partir des primitives de base fournies par le noyau. Ceci est réalisé par l'association d'actions aux relations refléxives définies comme nous le verrons dans le paragraphe suivant.

- les **post-actions** qui valident ou provoquent l'abandon de la commande; les postactions agissent comme des règles d'intégrité (décentralisées) qui sont mises en oeuvre pour interdire les actions et commandes qui provoquent un état incohérent;

- les **obligations** qui sont exécutées après que les mises à jour d'une commande soient validées. Les obligations agissent comme des règles mises en oeuvre pour rétablir l'intégrité des objets. Par exemple lorsqu'on modifie un objet, tous ceux qui l'utilisent directement ou indirectement voient leur validité remise en cause. En conséquence, les obligations peuvent comporter les tâches de revalidation correspondantes;

- les actions sur **demande** qui s'exécutent à la demande explicite par la commande *make* à partir des événements enregistrés,

- les actions sur **accès** à un objet qui s'exécutent lors de l'accès à un objet absent ou sur lequel des événements sont enregistrés.

Des actions appropriées à chacune des formes de déclenchement peuvent être définies dans le manuel de tout objet source d'une relation. Nous montrons sur la figure 5.6, l'enchainement des différentes formes de déclenchement d'actions puis nous décrivons dans ce qui suit les caractéristiques de chacune des formes de déclenchement.

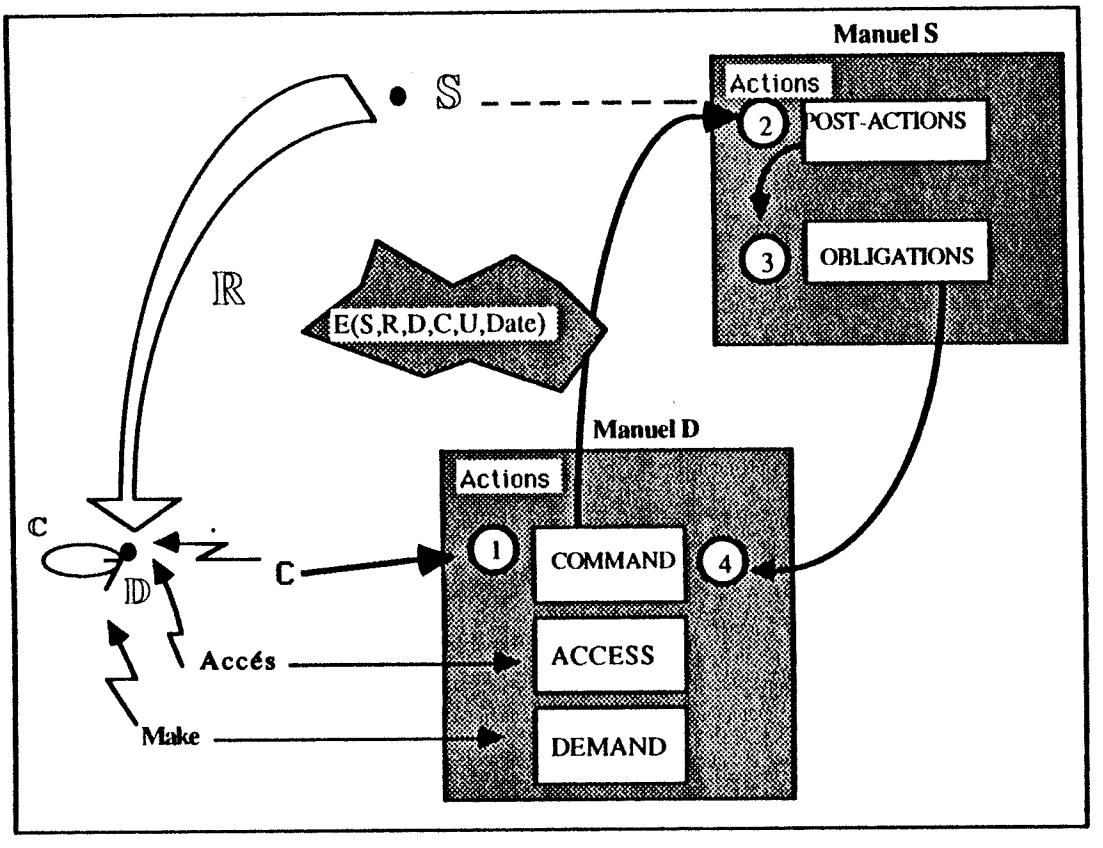


Figure 5.6 : Enchaînement des différentes formes de déclenchement d'actions

4.1. Les commandes

Lorsqu'une commande est exécutée sur un objet D, des événements peuvent être propagés à tous les objets reliés à D. Pour appliquer des actions sur D et utiliser uniformément le mécanisme d'événements/actions, Nomade permet à l'usager de définir des relations refléxives sur D (D R D) (cf chap.4).

Les actions réflexives permettent de contrôler l'utilisation d'un objet et de définir ainsi les commandes associées à un objet.

Si une relation reflexive *CMD* a été définie sur l'objet D, exécuter la commande

propagate(D, CMD, D...)

ou plus simplement introduire:

||CMD D ....

provoque l'événement E (D,**CMD**, D, C, U, Date); ce qui permet d'exécuter des actions associées à D pour cet événement. Ces actions sont définies dans le bloc "**command**" de la clause "**actions**" du manuel de l'objet D sous la forme suivante:

manuel D

actions

command

for CMD do

texte de l'action réalisant la commande;

done;

end

Exemple 1 :

L'exemple suivant décrit l'implémentation d'une commande compile qui transfère une réalisation dans l'espace usager (read), la compile et la recopie avec le code objet associé dans la base (catal).

```
actions
command
for compile do
/* lecture de la réalisation dans le fichier local usager fichloc */
        exec(read $D ) ;
        if $D (language = pascal) then
                system ( pc  -c -g $fichloc); fi;
        if $D (language = c) then
                system ( cc -c -g $fichloc );  fi;
        if $D (language= ...  then
                display ("langage inconnu"); exit; fi;
        exec(catal $D);
done;
```

Exemple 2 :

Définition d'une commande **print** d'impression adaptée au type du contenu de l'objet :

```
command
        for print do
        exec(read $D );
/* formattage des scripts shell et impression utilisant troff */
        if $D (elem= doc and language = shell) then system(vgrind -lsh  $fichloc);
        exit;  fi;
/* impression des sources yacc en utilisant directement troff  */
        if $D (language=yacc) then system(troff  $fichloc); exit; fi;
/* formattage des codes source c */
        if $D (language = c) then system( vgrind -lc $fichloc);exit; fi;
        if $D (elem= doc and language * shell) then system (cat $fichloc); fi;
done;
```

Les commandes **compile** D et **print** D exécutent les actions "command" pour les relations réflexives "compile" et "print" définies sur D de la manière suivante :

```
reflex  compile     ** (elem = sourcetext)
reflex  print       ** (elem =sourcetext or elem=doc)
```

Les actions de type *command* sont similaires à l'envoi du message "compile" à l'objet S dans les langages objets. Ces actions sont exécutées lorsqu'elles sont introduites par un usager ou sont appelées à partir d'une autre action.

Toutes les commandes de Nomade sont implantées ainsi. Ce sont des relations réflexives prédéfinies avec la commande elle-même comme action par défaut. L'usager peut donc les modifier, les étendre, les redéfinir ou en définir d'autres. C'est aussi un moyen de protection des objets qui consiste simplement à réduire l'utilisation de certaines commandes.

Une commande est une transaction c'est à dire une opération considérée comme atomique. Ainsi la base passe par des états cohérents correspondant à chaque fin de commande.

Exemple :

L'usager peur définir une commande release comme une suite d'actions :

```
for release do (create...; attr... ; transfert...); done;
```

Si une action de la commande release échoue, la commande release est défaite dans sa totalité.

4.2. Les post-actions

Les post-actions sont définies par l'usager dans le bloc **post-actions** pour vérifier des contraintes à la suite d'une commande.

```
manuel S
  actions
    Post-actions
        for R(C) do
        texte de l'action ;
        done;
end
```

Elles sont exécutées avant la validation de la commande qui a provoqué l'événement. Les post-actions évaluent si l'état des objets après l'exécution de l'action est satisfaisant. Dans le cas contraire, l'action est défaite. Plus exactement, tout objet, directement ou indirectement atteint par les effets d'une commande, peut demander de la défaire.

Les post-actions peuvent être utilisées pour différentes raisons. Nous en donnons deux exemples:

- Les post-actions permettent de montrer interactivement à l'usager les conséquences d'une commande . La commande est validée après confirmation par l'usager. Ceci permet déviter des erreurs dramatiques en montrant les répercussions des actions effectuées.

Exemple :
La modification d'une interface peut avoir comme post-action associée à la relation de dépendance la visualisation de la liste de toutes les réalisations et interfaces atteintes par la modification pour évaluer le coût des recompilations. Ceci peut s'exprimer de la manière suivante :

```
manuel S
actions
        post-actions
        for dep(modifier) do
                if not displayconf( "l'objet $S est atteint par la modification de $D,
                confirmez-vous?") then fail; done;
end
```

- Pour vérifier des contraintes violées par une commande.

Exemple :
La relation prédéfinie "S stabilise D " de l'environnement PCTE signifie que tant que l'élément source S existe, l'élément D ne doit pas être modifié, et ce, quelquesoit la commande lancée. Dans Nomade, la relation stabilise n'a pas besoin d'être prédéfinie comme dans PCTE; ceci s'exprime de la manière suivante :

```
manuel
actions
        post-actions
        for stabilise do fail; done;
end;
```

Ainsi toute commande, qui directement ou indirectement provoque une modification de D ou de la relation "stabilise", est refusée .

Par définition toute commande est défaite si au moins une des post-actions n'est pas satisfaite. Ceci a pour conséquence de refuser la commande et d'annuler toutes les modifications effectuées dans la base par la commande. En effet comme le contrôle de cohérence est décentralisé, il est impossible de connaitre statiquement les conséquences d'une action avant son exécution.

4.3. Les obligations

Les événements produits déterminent les objets affectés par une commande. Une fois la commande validée, il faut éventuellement restaurer la cohérence de ces objets, modifier leur état ou compléter la commande par d'autres actions. Ces actions sont dites obligations car elles sont exécutées obligatoirement après la commande qui a provoqué l'événement. Elles sont identifiées par le mot clé **obligations** dans la clause actions du manuel d'un objet.

manuel
actions
Obligations
for **R(C)** do
texte de l'action;
done;
end

La plupart des environnements détectent les effets propagés juste après l'exécution de la commande qui les a provoqués. Dans cette classe se trouve les systèmes langage dépendant tel que Gandalf, Cedar, Mesa et la plupart des autres environnements comme Fasp, LLNL. Les actions qu'ils exécutent sont du type obligations.

Exemple :

```
manuel
actions
    obligations
            for dep (modifier) do
            if $S(author = $U\name) then exec(compile $S); exit; fi;
            exec (attr state "interface-modifiée");
            record;
            done;
end
```

Ce qui exprime que chaque fois qu'une interface est modifiée, le texte source S est compilé immédiatement si l'auteur de la modification est l'auteur du texte source sinon l'événement est enregistré (record) dans le manuel de S et l'attribut "état" de S prend la valeur "interface modifiée".

4.4. Les actions sur demande

Un usager peut demander si un objet donné a été atteint par un événement. De nombreux environnements fournissent ce service en s'appuyant sur les dates de dernière modification données par le système d'exploitation. L'outil *Make d'Unix* illustre bien cette classe de systèmes. Ce n'est que lors de l'exécution explicite d'un *Makefile* que les effets propagés sont détectés et éventuellement traités. Nomade fournit également ce service par les actions sur demande, non pas sur la base des dates de modification mais en utilisant les renseignements enregistrés dans l'objet lui-même, ce qui fournit plus d'informations.
Les actions sur demande sont exécutées lorsqu'un usager active la commande :

```
make S R C
```

Ce qui a pour effet d'exécuter les actions pour les événements de type E(S, R, *, C,...) enregistrés dans le manuel de l'objet S.

Ces actions sont exprimées par le mot clé **demand** dans le manuel de l'objet S.
Remarque:

Le méta-caractères '*' est admis. Ainsi *make* * * * prend en compte tous les événements.

```
manuel S
   actions
      Demand
            for R(C) do
            texte de l'action;
            done;
end
```

Les actions sur demande permettent à l'usager de programmer des stratégies de gestion de la cohérence exécutées sur demande et non à chaque fois que l'événement est levé.
Dans l'outil Make, la relation implicite entre deux objets est enregistrée dans le Makefile et le seul événement reconnu est "le fichier D est plus récent que le fichier S". Ce sont des actions sur demande puisque l'usager doit activer le makefile pour que les actions soient exécutées.
Les stratégies *à la Make* peuvent être implantées aisément en Nomade puisqu'il suffit de programmer des actions sur demande sur un objet pour le reconstruire proprement d'où le nom "make" de la commande.

Exemple : Make d'Unix programmé avec le mécanisme d'événements-actions de Nomade.

Le traitement équivalent à ce que fait l'outil make d'Unix peut être obtenu par le mécanisme d'événements-actions à l'état "demand".

L'exemple de reconstruction automatique d'une configuration lors des modifications d'éléments la composant se programme de la manière suivante dans Nomade :

```
manuel  S
actions
demand
    /* pour la configuration où un événement a été enregistré sur R=comp avec
    C=obsolète faire l'action suivante */
for comp(obsolete) do
    /* si l'objet est une réalisation qui a été modifiée, le compiler */
    if ($D (object = realization and elem = sourcetext) then exec(compil $D); exit;fi;
    /* si D est une configuration englobante qui a été modifiée, appliquer le make
    récursivement pour les objets modifiés qui dépendent de cette configuration
    englobante */
    if $D(object = conf and elem=sourcetext) then exec (make $D comp obsolete);
    exec(rebuild $S) ;fi;
done
end
```

Ce programme est activé par la commande Nomade:

```
make nomdeconfig (= S) comp ( =R)  obsolete (=C)
```

Note : Ce programme d'action est écrit une seule fois et est valide pour toute configuration de tout système ou sous-système géré par Nomade

4.5. Les actions sur accès

Lorsque des événements sont enregistrés sur un objet , les actions associées peuvent être exécutées lorsqu'on y accède. Ces actions sont exprimées dans le bloc "**access**" de la clause **actions** du manuel de l'objet.

```
manuel S
  actions
    access
        for R(C) do
        texte de l'action;
        done;
end
```

Il y a peu d'environnements qui offrent ce service parce qu'il nécessite l'enregistrement des événements. Ce type d'action permet de reporter le traitement des événements lors de l'accès. Les actions "sur accès" peuvent être utilisées :

- pour monter aux usagers la liste des événements enregistrés sur l'objet et non encore traités. Ce mécanisme évite l'utilisation d'un objet à l'état incohérent. Une action définie en accès peut consister à avertir l'usager en indiquant les événements associés à cet objet et non encore pris en compte.

- permettre une évaluation par nécessité en exécutant les actions seulement lors d'un besoin. Par exemple un programme peut être automatiquement recompilé seulement lors qu'il est utilisé ou lorsqu'on a besoin de son code objet.

4.6. Récapitulation de la définition de la clause Actions

Pour compléter la definition des types structurés Famille et Usager, on définit ainsi la clause actions en reprenant les constructions présentées précédemment :

```
<Actions-clause>     ::= actions  [ command  ( < ForStatement> ) ]
                                  [post-actions     ( <ForStatement> )]
                                  [obligation       ( <ForStatement> )]
                                  [demand     ( <ForStatement> )]
                                  [access     ( <ForStatement> )]
<ForStatement>       ::= for <Relation> [(<command> )]
                            do ([IfStatement>] | <Action>) done;
<IfStatement>        ::= if <Condition> then <Action> fi;
<Relation>           ::= <identifier>
<Condition>          ::= <NomadeExpr> | [ not] ( Nomade-function )
<Action>             ::= (<Nomade-function> ; )
<Nomade-function>    ::= <Name-function> [( <string> ) ]
<Name-function>      ::= exec | rexec| system | propagate | display |
                         displayconf | record | option | reset | fail | exit
```

5. HERITAGE D'ACTIONS

La paramétrisation des actions par les objets intervenants dans la production des événements permet la définition d'actions communes à un projet, une partition, une famille et l'extension du mécanisme d'héritage de manière uniforme à la clause *actions*. La recherche d'une action associée à un événement utilise (comme pour les attributs, les droits d'accès et les relations) la structure hiérarchique projet / partition imbriquées / famille. Cependant comme les actions sont relatives à des relations, il leur est appliqué comme pour les relations un héritage simple. En effet les relations ne peuvent être définies que dans les partitions associées à une relations structurante.

Pour un événement E (S, R, D, C, U, date), l'action correspondante est recherchée:

- dans la famille de S,
- puis récursivement dans la partition de S où R est défini jusqu'à atteindre la partition projet. La première action possible pour cet événement est exécutée.

Ainsi une action définie :

- dans les partitions ayant la famille "*Project*" comme racine est valide pour tous les objets dans ce projet,
- dans une partition quelconque est valide pour toutes les familles ou usagers de la partition,
- dans la famille ou usager est valide pour tous les objets de la famille ou de l'usager selon le cas.

Ceci permet de définir des comportements communs à un ensemble de familles, d'usagers et de partitions et des comportements spécifiques uniquement où cela s'avère nécessaire.

Par exemple les relations réflexives (commandes) "print" et "compile" et leurs actions associées peuvent être définies au niveau global projet, mais toute famille peut redéfinir ces actions.

Exemple :

On définit une action **compile** commune à l'ensemble du projet. Une des stratégies de compilation pourrait être la suivante :

```
manuel
    actions:
/* Lors de la modification d'un objet utilisé par une configuration à l'état officiel, la
commande de modification est refusée.*/
        ||post-actions
              for comp(modifier) do
                    if $S (state = officiel) then fail;
              done
/* La relation comp (composition) relie les composants d'une configuration à l'objet
configuration*/

/* L'événement est enregistré, la recompilation n'est pas immédiate pour les réalisations
qui en dépendent */
        ||obligations:
              for dep (modifier) do record; done;

/* la compilation est exécutée lors de l'accès au code source */
        ||access : for dep(modifier) do exec(compile $S); done;

/* la compilation est exécutée sur demande explicite de l'usager */
        ||demand: for dep(modifier) do exec(compile $S); done;
end
```

- Inversement, on peut avoir une stratégie de recompilation définie globalement pour le projet comme par exemple celle exprimée ci-dessus. Mais l'action compile peut être définie différemment dans différentes familles comme par exemple:

```
        ||command :
        ||for compile do
              if $D (state = test and language = c) then
                                    exec(read $D);
                                    system (cc -g -o $fichloc);
                                    exec(catal $D); fi;
        ||done;
        ||/* recompilation en mode mise au point : option "-g " */
```

6. PROPAGATION

Jusqu'ici nous avons considéré seulement une étape de propagation et un seul événement. D'autres problèmes apparaissent lorque l'on considère plusieurs événements, plusieurs relations et la propagation avec plusieurs étapes.

6.1. La propagation de plusieurs événements sur une relation unique

Supposons que l'on a défini dans un projet la stratégie suivante de traitement applicable aux interfaces et réalisations :

- l'action commande "modifier" pour modifier une interface. La modification d'une interface propage des événements sur la relation de dépendance *dep* pour toutes les interfaces et réalisations utilisant l'interface.

- l'action obligation sur la relation de dépendance qui définit le traitement à effectuer sur les objets (interface ou réalisation) atteints par une modification d'une interface dont ils dépendent.

Ces actions sont définies plus précisemment de la manière suivante :

```
manuel project
actions
    command
    for modifier do
        if $D(object = interface) then
        /* verification du droit de modication */
                    checkright(catal, $D);
        /* réalisation de la modification effective par la commande catal */
                    exec(catal $D);
        /* propagation aux réalisations et interfaces utilisant l'interface modifiée */
                    propagate ( *,dep, $D, "modifier"); fi;
    done;
```

```
obligations
for dep (modifier)  do
        /* cas d'une interface */
                if $S(object =interface) then
                /* si interface compilable */
                                exec(compile $S)
                /*propagation aux interfaces et réalisations utilisant cette interface*/
                                propagate(*,dep, $S, modifier);
                /* sortie du programme d'action */
                                exit; fi; done;
        /* cas d'une réalisation */
                if $S(object = body) then
                /* compilation du corps sans propagation */
                                exec (compile $S); exit; fi;
done
end
```

Appliquons ces actions à l'exemple de structure représentée par la figure 5.7 qui montre une réalisation P dépendant de deux interfaces I1 et I2.

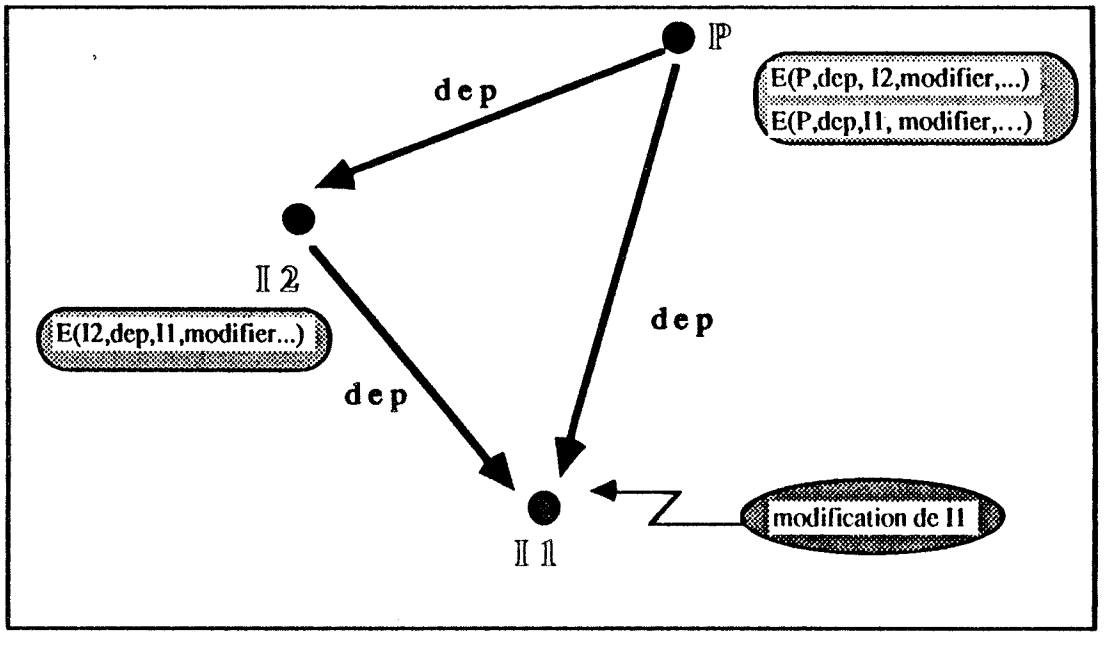


Figure 5.7 : Ordre de traitement des actions sur une même relation.

- Le changement de l'interface I1 produit l'événement E(S, dep , I1, modifier,...) sur toutes les interfaces et réalisations utilisants I1 en l'occurence I2 et P soit :
E(P, dep, I1,modifier,...) et E(I2,dep, I1,modifier,...). I2 et P sont à recompiler et à recataloguer dans la base.

- Le premier problème est celui du bon ordre de compilation. L'interface I2 utilisée par P doit être compilée avant celui-ci. Autrement dit, l'événement E(I2, dep, I1, modifier) doit être traité avant E(P, dep, I1, modifier,...). Le traitement de l'événement E(I2, dep, I1, modifier,...) produit de même l'événement E(P, dep, I2, modifier,...) car P utilise I2.

- La réalisation P dépendant de I1 et I2 comme le montre la figure 5.7, il apparait un second problème. En effet deux événements sont enregistrés sur P qui pourraient entrainer deux compilations de P, la seconde étant inutile.

Pour résoudre ces problèmes, Nomade calcule un ordre partiel sur tous les objets liés par une même relation en considérant le graphe de cette relation. La racine du graphe est numérotée 0, et le niveau d'un objet est le nombre maximum de noeuds entre la racine et l'objet. Les graphes avec circuits ne sont pas acceptés.

Ainsi pour une commande C sur un objet D, le mécanisme de propagation calcule tous les événements pour une relation R et construit une liste de tous les événements E(S, R ,D, C,...) dans l'ordre décroissant des niveaux associés à S pour la relation R. Puis les actions associées à chacun des événements sont exécutées dans l'ordre de la liste. Les actions produites par cet événement peuvent produire d'autres événements. Les nouveaux événements sont insérés dans la liste déjà existante pour R. Les événements sont successivement pris en compte jusqu'à épuisement de la liste.

Cet algorithme traite le premier problème puisqu'il assure que des compilations sont exécutées dans le bon ordre, mais n'empêche pas des compilations multiples pour une même réalisation. Cet aspect doit être traité dans le programme d'action. Lorsque l'événement E(P, dep, I, modifier,...) demande de recompiler P, l'événement E(P, dep, I2, modifier,...) doit être annulé.
La fonction :

reset (S, R, D, C)

permet de supprimer un événement E(S, R, D, C,...) de la liste pour qu'il ne soit pas pris en compte.

Exemple:

Reprenons l'exemple précédent pour montrer les différentes phases du traitement en tenant compte de cette nouvelle possibilité.

L'action obligation de la relation de dépendance qui permet de supprimer les compilations multiples s'écrit de la manière suivante:

```
manuel project
    Actions :
obligations
    for dep (modifier)  do
    /* cas d'une interface */
          if $S (object =interface) then
          /* si interface compilable */
                    exec(compile $S);
                    reset( $S, dep, *, modifier);
          /*propagation aux interfaces et réalisations utilisant cettte interface */
                    propagate(*,dep, $S,modifier); exit; fi; done;
    /* cas d'une réalisation */
          if $S (object = body) then
                /* compilation du corps sans propagation */
                    exec (compile $S); exit; fi;
                    reset( $S, dep, *, modifier);
    done
end
```

La fonction reset (S, dep, *, modifier) supprime tous les événements de type E(S, dep,...) déjà insérés dans la liste des événements à traiter.

déroulement

L'évaluation, pour les actions à exécuter, suit les étapes suivantes :

1) Traitement de la relation de dépendance:

- La modification de I1 produit la propagation de deux événements sur la relation de dépendance enregistrés dans l'ordre suivant:

E(I2, dep, I1, modifier,...) et E(P, dep, I1, modifier,...).

- Traitement de E(I2, dep, I1, modifier,...): compilation de I2; production de l'événement E(P, dep, I2, modifier,...) déclenché par propagation;

- Traitement de E(P, dep, I, modifier,...): compilation de P; le traitement de l'événement E(P, dep, I2, modifier,...) est annulé par la fonction *reset* pour ne pas recompiler une seconde fois;

6.2. La propagation de plusieurs événements sur plusieurs relations

Supposons que la stratégie de gestion adoptée dans un projet, consiste à reconstruire automatiquement les configurations lorsqu'un des composants est changé ou recompilé. Les deux relations mises en oeuvre sont:

- la relation de composition (*comp*) entre une configuration et ses composants;
- la relation de dépendance (*dep*).

L'action associée à la relation de dépendance est identique à celle définie dans le paragraphe précédent; celle associée à la relation de composition est définie comme suit:

```
manuel
    actions
    obligations
        for comp (périmé) do exec (rebuild $S); done;
end
```

La configuration doit être reconstruite automatiquement seulement après la recompilation de tous ses composants. Autrement dit, les actions associées aux événements sur la relation *"comp"* doivent être exécutées après les actions associées aux événements sur la relation *"dep"*. La relation *"dep"* doit être plus prioritaire que la relation *"comp"*.

En effet, un objet D pouvant être cible de plusieurs relations, le déclenchement des événements peut être quelconque si un ordre de priorité des relations n'est pas imposé. Ce dernier est défini en fonction du sens que l'on veut accorder à l'ordre de déclenchement des événements. Certaines relations ont un rapprochement évident, par exemple dépendance et composition avec traitement de la relation de dépendance (recompilation des objets) avant la composition (constitution d'un exécutable).

Dans Nomade, un ordre partiel peut être défini entre les relations pour exprimer cette priorité. C'est l'ordre de déclarations dans la clause "relations" qui définit l'ordre des relations. Dans Nomade, les relations *"dep"* et *"comp"* sont prédéfinies comme suit dans la partition projet:

```
**.  dep   *:*.
**(object=realization and language=conf)   comp          **.
```

Une configuration peut être un composant d'une autre configuration. La commande *"rebuild"* de reconstruction d'une configuration est suivie de la fonction *"propagate"* qui produit des événements E(...,comp,...) et entraine récursivement la reconstruction de toutes les configurations englobantes. Reprenons l'écriture de l'action *"comp"* permettant de reconstruire les configurations englobantes, soit :

```
manuel project
    actions
    obligations
        for comp (périmé) do
                exec (rebuild $S);
                propagate( *, comp, $S, périmé);
                done;
end
```

Exemple :

Nous illustrons par cet exemple les problèmes soulevés précédemment montrant la necéssité d'ordonner les relations pour un contrôle cohérent de l'ordonnancement des événements.
Soit un exemple d'une structure où *"Conf"* est une configuration composée de l'interface *I* et de la réalisations *A* . Cette dernière dépend de l'interface *I* comme le montre la figure 5.8:

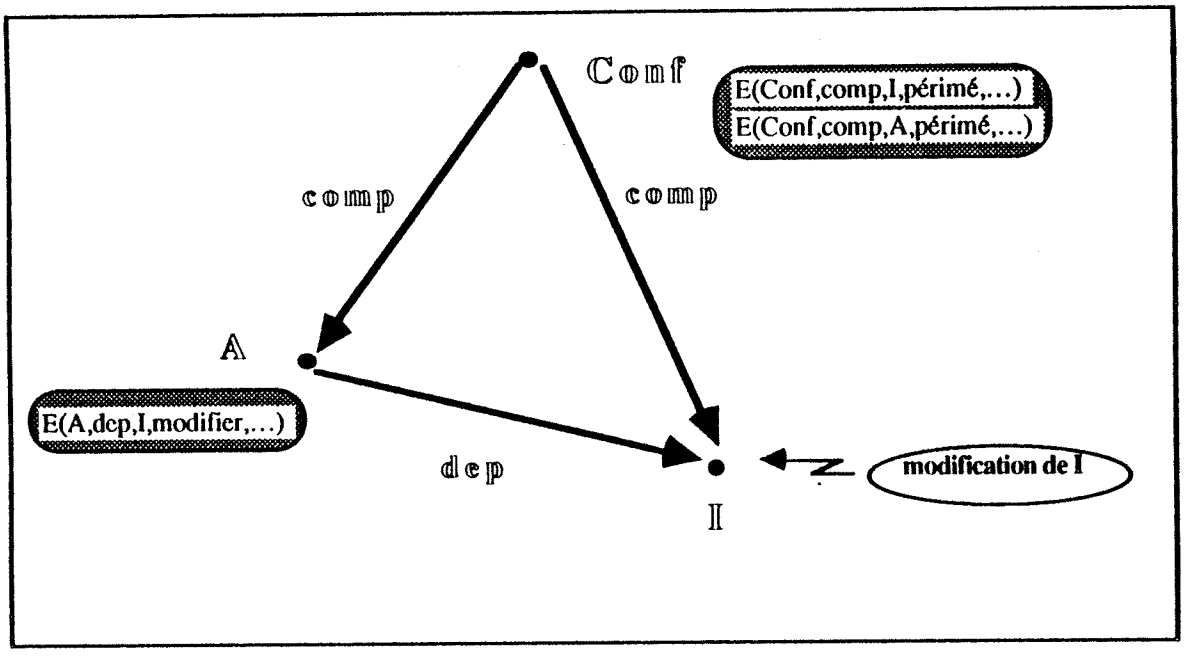


Figure 5.8 : Ordre de traitement des actions sur plusieurs relations différentes

L'interface *I* est modifiée, observons les effets propagés sur les objets en relation et le déroulement du traitement :

hypothèse :

- la relation de dépendance est plus prioritaire que la relation de composition;
- *Conf* est une configuration composée de la réalisation *A* et de l'interface *I*.

Actions

```
manuel project
    actions
        for dep(modifier) do
            exec( compile $S );  /* si interface compilable */
            reset( *, dep, $S, modifier);
        /*propagation aux interfaces et réalisations utilisant cettte interface */
            propagate(*,dep, $S, modifier);  exit; fi; done;
            propagate(*, comp, $S, périmé);
        done;

        for comp(périmé) do
            exec (rebuild $S);
            reset ($S, comp, *, périmé);
            propagate(*, comp, $S, périmé);
        done;
end
```

déroulement:

L'évaluation des actions à exécuter se fait comme suit :

- Traitement de la commande: modification de *I* avec propagation de 2 événements sur les relations "dep" et "comp" : E(A, dep, I, modifier,...) et E(conf, comp, I, périmé,...):

1) Traitement de l'événement associé à la relation de dépendance en priorité:

- Traitement de l'événement E(A, dep, I, modifier,...): compilation de *A*; l'événement E(Conf, comp , A, périmé,...) est produit;

2) Traitement des événements associés à la relation de composition :

- Traitement de l'événement E(Conf, comp, I, périmé) : reconstruction de la configuration *Conf*; puis annulation par la fonction *reset* de l'événement E(Conf, comp, A, périmé,...) pour ne pas reconstruire une seconde fois.

6.3. Le traitement du bouclage d'actions

Un des problèmes important de tout environnement qui gère les propagations est celui du bouclage d'actions, autrement dit une succession d'actions qui s'enchainent. On distingue plusieurs solutions à ce problème :

- le principe de l'horloge de garde où l'on compte les événements et au delà d'une certaine limite, on décide qu'il y a bouclage d'actions;

- la conservation de tous les événements qui sont produits et le contrôle à chaque nouvel événement s'il n'a pas déjà été traité. Nous avons choisi cette dernière approche dans le développement du prototype.

7. EXEMPLES RECAPITULATIFS

Nous présentons deux exemples pour donner une vue d'ensemble du mécanisme d'événements-actions de Nomade.

7.1. Exemple1: la reconstruction d'une configuration

Nous exprimons dans cet exemple une stratégie de reconstruction de configurations définie par les règles suivantes:

- les composants utilisés par des configurations officielles ne doivent pas être modifiés;

- lorsqu'un composant est mis à jour, les configurations de test sont immédiatement reconstruites (pour tester les changements), les autres types de configurations deviennent périmés et sont reconstruits sur demande;

- lors d'un accès à une configuration périmée, l'usager en est informé et doit décider intéractivement de la reconstruction;

- l'usager demande de reconstruire à la demande les configurations périmées par la commande suivante :

```
make nomdeconfig comp périmé
```

Cette stratégie s'exprime comme suit dans Nomade sur la relation de composition (*comp*) qui lie toute configuration à ses composants (relation calculée par le gestionnaire de configuration et prédéfinie dans Nomade.

```
post-actions :
        for comp (périmé) do
                if $S(state = officiel) then fail; fi;
                propagate(*, comp, $S, périmé);
        done

obligations:
        for comp (périmé) do
                if $S (state = test) then exec (rebuild $S); fi;
                if $S(state <> test) then record; exec(attr $S state "périmé");
                fi;
        done
```

```
access:
        for comp (périmé) do
                if displayconf("la conf $S  comporte des éléments modifiés,
                voulez-cous la reconstruire ?") then exec( rebuild $S);  fi;
        done;
demand
        for comp (périmé) do
                if $S(state = périmé) then  exec (rebuild $S); fi;
        done;
```

7.2. Exemple 2 : le traitement des options d'une commande

Cet exemple montre la définition de la commande de lecture de réalisations avec deux options : *edt* et *pr*; L'option *edt* permet, après la lecture d'une réalisation, l'activation de l'éditeur de texte préféré de l'usager défini dans le fichier de commande *EDT* ; l'option *pr* permet l'impression d'un listing. *EDT* est un fichier de commandes implanté dans l'espace local de l'usager. Le programme d'action permettant de traiter ces options est défini comme suit :

```
manuel objet
  actions
  command
     for read do
     /*exécution de la  de lecture nomade*/
           exec ( read $D);

     /*traitement de l'option "pr" qui demande l'impression d'un listing, la fonction
     "option" vérifie si l'option est présente, $fichloc est le nom d'une variable contenant
     le nom du fichier local où est stocké le résultat de la lecture */
           if option("pr"...) then system( imag ppr +Hln $fichloc); fi;

     /*traitement de l'option "edt" d'appel de l'éditeur de l'usager à partir du fichier de
     commande EDT dont le contenu peut être par exemple l'appel de l'édteur  "vi" ou
     "emacs"...*/
           if option ("edt"...) then system( EDT $fichloc);fi;
     done;
  end
```

8. CONCLUSION

Nous avons présenté principalement des exemples de recompilation et de construction de reconfigurations parce qu'ils sont simples et sont très utilisés dans un environnement de support à la programmation globale. Tous nos exemples ont été écrits en quelques lignes mais des stratégies de gestion plus complexes nécessitent plus de puissance d'expression. Les actions peuvent être des commandes de la base, des commandes du système d'exploitation (Unix dans nos exemples) ou des programmes.

Nous avons essayé de montrer la versalité du mécanisme d'événements-actions. Il peut être utilisé à plusieurs fins comme :

- personnaliser les commandes et en ajouter d'autres;
- contrôler la sémantique des relations et des attributs, les accès aux objets et les propagations;
- définir des pré et des post-conditions à des commandes;
- définir des transactions, des stratégies de gestion de configurations, de gestion de la base et les enchainements d'outils;

Le système Nomade utilise les relations définies explicitement et le typage des objets pour guider le gestionnaire d'activités et permettre l'activation d'outils. Dans Nomade, il y a plusieurs niveaux de vérification et une séparation entre la détection des événements et les actions. Toute activité Nomade est décrite par des pré-conditions et des post-conditions. Ces dernières ont la caractéristique d'être décentralisées et exprimées au niveau des objets ayant un lien avec l'objet en cours de modification. Les activités sont automatiquement exécutées en fonction de l'état des objets. La paramétrisation des actions par les objets et les relations de la base donne plus de flexibilité au mécanisme.

Il reste à aborder la question de savoir comment construire et valider de tels programmes d'actions et notamment d'étudier plus profondemment les problèmes liés au contrôle du mécanisme par le système, sa maîtrise par l'usager et de son efficacité face à la prolifération d'actions. Cela nécessitera sans doute le contrôle de la cohérence même de l'ensemble des événements produits en analysant plus finement la structure de cet ensemble.

CHAPITRE 6 :

LE NOYAU NOMADE : LES ASPECTS DE REALISATION

1. PRESENTATION

Dans ce chapitre, on va traiter de quelques aspects de réalisation du noyau Nomade en essayant de justifier les solutions prises. La démarche adoptée dans la réalisation de Nomade consiste à prendre comme point de départ Adèle et à l'étendre par de nouvelles fonctionnalités. C'est ainsi que la plupart des décisions de conception d'Adèle sont maintenues ainsi que la sémantique de la plupart des modules quoiqu'ils puissent avoir des structures internes différentes. Dans cette optique, un effort important de réalisation a été déployé sur Adèle pour l'amener à un niveau d'utilisabilité satisfaisant.

Nous présentons simultanément l'architecture fonctionnelle du système Nomade et son organisation générale en montrant les différentes parties qui le composent. Puis nous examinons les aspecs liés à l'organisation interne de la base et la structure interne du manuel. Enfin, nous développons quelques aspects techniques permettant une exploitation cohérente de la base d'objets.

2. L'ARCHITECTURE FONCTIONNELLE DE NOMADE

2.1. Principe des solutions retenues

Un aperçu des environnements logiciels en général montre que les architectures matériel généralement envisagées, sont constituées d'un serveur central base de donnée et d'un réseau de stations de travail. La solution que nous présentons est développée dans le cadre d'un environnement Unix.

Pour pouvoir exploiter les différentes possibilités de ce type d'architecture, nous avons adopté une solution basée sur un modèle client-serveur. Nomade est développé comme un serveur implanté sur un site central. Il est accessible à distance par un ensemble d'usagers clients via un

réseau local de type Ethernet. Cette approche implique cependant de traiter plusieurs aspects notamment les problèmes de communication et la conception d'une architecture fonctionnelle et d'un protocole de communication adaptés.

Dans la recherche d'une solution pour traiter les problèmes de communication, nous avons pris pour modèle le mécanisme des *sockets* du système Unix 4.2 BSD qui nous a semblé adapté à nos besoins. En effet ce mécanisme présente l'avantage de règler les problèmes de communication permettant ainsi un accès à distance transparent. Le *socket* est un port de communication attaché à chaque processus Unix. Les *sockets* de deux processus Unix peuvent être connectés de manière bi-directionnelle sur des sites différents. Sur chaque *socket*, il est possible d'exécuter des opérations de lecture, d'écriture et de contrôle.

2.2. La structure en processus

Le gestionnaire de la base Nomade s'exécute comme un processus (au sens du système Unix) indépendant du processus client pour des problèmes de protection de la base. La structure mise en oeuvre utilise un processus gestionnaire de la base par client. La structure fonctionnelle de Nomade est représentée par la figure 6.1. Les processus mis en jeu sont les suivants :

- Le processus Serveur Nomade est un processus "démon", chargé de l'écoute des demandes de communication du client. Il est chargé de lancer les processus Nomade gestionnaire de la base et d'initialiser le dialogue avec les processus clients

- Le processus gestionnaire de la base Nomade est créé pour chaque client et est chargé de traiter les commandes émanant des clients.

- Le processus client est soit un terminal, soit un programme d'application usager.

- Le processus "préambule" est activé sur tout site référencé par les programmes d'actions, pour l'établissement du dialogue entre d'une part le processus ACTION et d'autre part le couple de processus : gestionnaire de la base Nomade et Client.

Deux *sockets* sont nécessaires pour faire dialoguer un client avec le gestionnaire de la base :
- un *socket* de contrôle pour faire transiter les commandes du client vers Nomade et pour recevoir le statut de fin de commande;
- un *socket* pour l'échange d'informations générées par l'éxécution de la commande.

2.3. Les fonctionnalités

Un processus serveur accessible par tous les sites est en attente continuellement sur le site central. A la demande d'un processus client sur un site distant et après son identification, il crée un processus Nomade et initialise la communication entre le gestionnaire de la base et le processus client. Nous fournissons une interface qui gère la communication avec le processus Nomade et offre la possibilité d'exécuter les commandes de Nomade pour la consultation des informations dans la base. Cet interface se présente sous forme d'une bibliothèque de fonctions à attacher avec le programme client ou action. Cet interface permet entre autres le transfert d'objets de la base publique implantée sur le site central vers l'espace privé de l'usager sur le site distant et inversement. Pour un utilisateur, le processus gestionnaire de la base Nomade est unique et gère tous les événements produits et toutes les actions déclenchées. Une interface simplifiée permet l'accès à des programmes Pascal, C et Prolog.

Des actions Nomade peuvent correspondre à des fichiers exécutables qui ne sont déterminés qu'à l'exécution en raison du fait que les actions sont insérées et modifiées à tout instant. Dès qu'une action est déclenchée un processus est créé et travaille en co-routine avec le processus gérant la base.

Le mécanisme de communication via l'interface *socket* a été appliqué dans le cas des actions. Ceci permet de déporter des actions sur des sites distants, déterminés dynamiquement. Sur chaque site distant potentiel, un processus "Préambule" est implanté. Sa fonction est de prendre en charge les aspects liés à l'initialisation de la communication entre le processus Action et d'une part le processus Nomade, d'autre part le processus client.

Tous les processus apparaissent comme des tâches standard pour le système Unix échangeant des informations via l'interface *sockets*.

Du fait de la complexité de ce schéma, une maquette de ce système a été réalisée et expérimentée sur le réseau éthernet de l'IMAG avant son intégration dans le noyau Nomade.

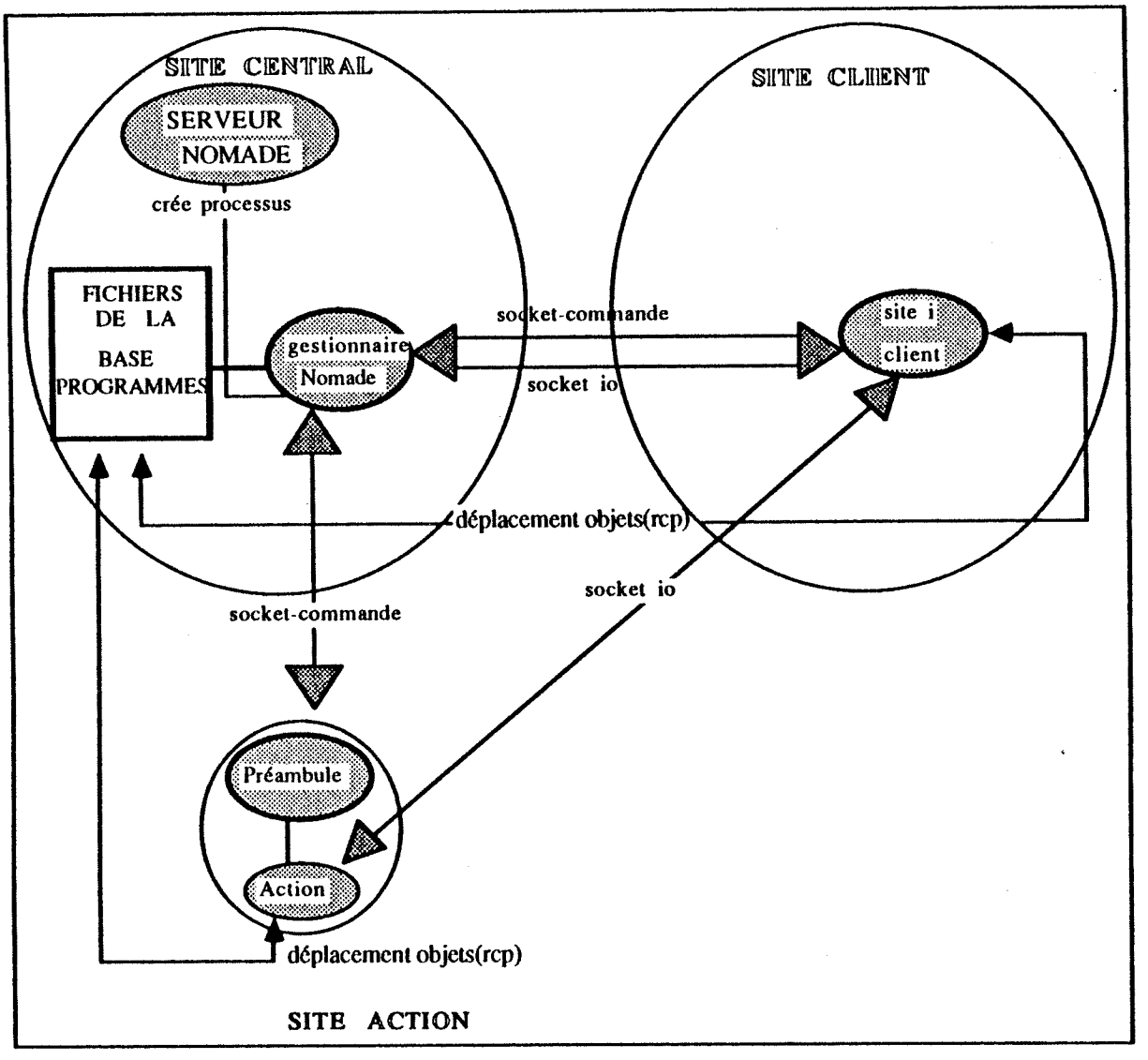


Figure 6.1. Architecture fonctionnelle de Nomade

Commentaire sur l'architecture représentée :

La figure 6.1 montre les interactions mises en jeu dans un état de développement donné représenté par un processus " gestionnaire de la base Nomade" s'exécutant pour le compte d'un processus "client" et le déclenchement d'un processus "action".

3. L'ORGANISATION GENERALE DE NOMADE

L'architecture générale du noyau Nomade est représentée par la figure 6.2. Elle montre les différentes parties qui le composent.

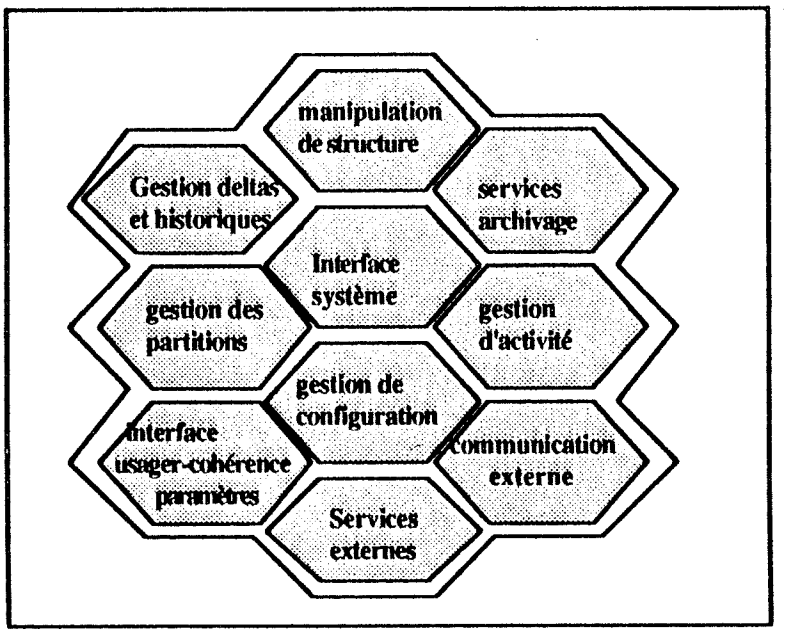


Figure 6.2 : Architecture générale du noyau Nomade

3.1. Les différentes parties

3.1.1. L'interface usager et la cohérence des paramètres.

L'interface usager est une interface de commandes. Elles est accessible soit directement à partir du terminal soit à partir de programmes.

Cette partie est composée de modules qui assurent:

- l'analyse des commandes;
- l'évaluation des expressions Nomade et la vérification des types des objets;
- la gestion des objets par défaut (Ceux figurant dans la clause *Seldef* sont pris par défaut sinon le plus récent);
- la gestion des informations associées aux usagers (paramètres, options, famille courante...);
- la récupération et l'analyse des options des commandes et des paramètres;
- la gestion et la récupération d'erreurs et la journalisation;
- la gestion des messages d'erreurs;
- la gestion des noms externes, c'est à dire la transformation entre les noms des objets dans la base et leur correspondant (noms externes) dans l'espace local de l'usager et inversement.

3.1.2. La manipulation de structures.

C'est un ensemble de modules chargés de :

- la création et la destruction des objets dans la base;
- la gestion de projet (création de base).

3.1.3. Les services d'archivage.

Ce sous-système est composé d'un ensemble de modules et assure des services aussi divers que :

- la détermination des interfaces et des réalisations attachées à une famille,
- la liste des familles,
- la manipulation des objets associés à une famille ou un usager,
- la vérification de la non-cyclicité des relations et le calcul des niveaux des objets,
- la gestion des manuels notamment la lecture du manuel de la mémoire secondaire vers la mémoire centrale et inversement. Les manuels en mémoire secondaire sont sous forme compactée, en mémoire ce sont des listes.

3.1.4. La gestion de la communication externe

Ce sous système regroupe les modules pour la gestion des interactions clients-Nomade et action-Nomade. Ceci comprend :

- le module *préambule* pour l'initialisation de l'exécution d'actions sur les sites distants;
- les bibliothéques des fonctions *clients* et *actions* pour la gestion de la connexion-déconnexion et la soumission de commandes au gestionnaire de la base Nomade sur le site central;
- les modules de communication externe du gestionnaire de la base Nomade.

3.1.5. La gestion des historiques et des deltas

On retrouve les fonctionnalités du système RCS avec les modules assurant :

- la constitution et la lecture des historiques;
- la réalisation des commandes de réservation, de libération, de lecture et de catalogage des objets dans la base;
- la gestion des deltas.

3.1.6. La gestion des partitions

Ce sous-système est constitué des modules gérant la manipulation des types structurés, c'est à dire les quatre aspects suivants :

- la gestion de la clause *actions*,
- la gestion de la clause *attributs*,
- la gestion de la clause *relations*,
- la gestion de la clause *droits d'accès*.

3.1.7. La gestion de configurations

Ceci regroupe les modules chargés de :

- la construction de configurations,
- la gestion de l'interactivité dans le cas de la construction interactive de configurations,
- la compilation de manuels,
- la gestion des modifications dont la fonction principale est la gestion de la commande de catalogage. Ce module prend en compte les répercussions engendrées par la modification de chaque type d'objet. En particulier la modification d'une configuration entraine sa reconstruction et la modification d'un manuel sa compilation.

3.1.8. La gestion d'activités

Ce sous- sytème gère le mécanisme d'événements-actions.

Il regroupe les modules chargés de :

- l'ordonnancement des événements par forme de déclenchement et relations (constitution des listes d'événements);
- la détermination de l'action à exécuter;
- l'interprétation des actions avec une exécution locale ou à distance sur le réseau local.

3.1.9. L'interface système

Ce sous-système est constitué de modules assurant l'interface avec le système d'exploitation.

On retrouve les modules permettant :

- la création de fichiers et d'annuaires, la copie et recopie, la visualisation sur un terminal, le test de l'existence d'un objet, le changement d'annuaire...
- la gestion des interruptions, la récupération d'erreur et la réalisation de l'exclusion mutuelle.

3.1.10. Les services externes

Ce sous-système est constitué d'outils permettant :
- la manipulation des configurations (l'outil *Exec*); ceci permet d'exécuter des commandes d'un bloc sur les composants d'une configuration,
- l'installation et la constitution d'une base gérée par Nomade à partir d'une liste de modules constituant un logiciel (l'outil *Install*).

3.2. La représentation de la base d'objets

Nomade et Adèle utilisent le système de gestion de fichiers de la machine hôte en l'occurence Unix pour la représentation interne des objets. La base est organisée à partir de deux répertoires :

3.2.1. Le répertoire des fichiers de contrôle

A partir de cet annuaire, sont implantés les fichiers de contrôle qui sont :

- Le fichier contenant les projets (*base*) connus avec un format interne sous forme d'un tuple: (nom_de_base, système hôte =unix, Auteur).

- Les fichiers usagers. Chaque usager connu de la base dispose d'un fichier à son nom. Ce fichier contient la famille courante par défaut, les paramètres et options par défaut et les informations sur les objets réservés par l'usager.

- le fichier *mutex* qui est un verrou global pour les commandes qui manipulent le fichier *base*.

3.2.2. Le répertoire des objets de la base

Les bases sont implantées à partir de cet annuaire.

- Pour chaque base, un annuaire est créé et contient les liens vers les interfaces de cette base. Cet annuaire est disponible pour compiler directement avec les interfaces de la base.

- chaque objet atomique de type Text est représenté par un fichier Unix.

- chaque objet composé famille et usager, interface et réalisation est représenté par une organisation hiérarchique d'annuaires représentant la partie structurelle prédéfinie dans le modèle.

La figure 6.3 montre l'organisation des objets en utilisant le SGF d'Unix.

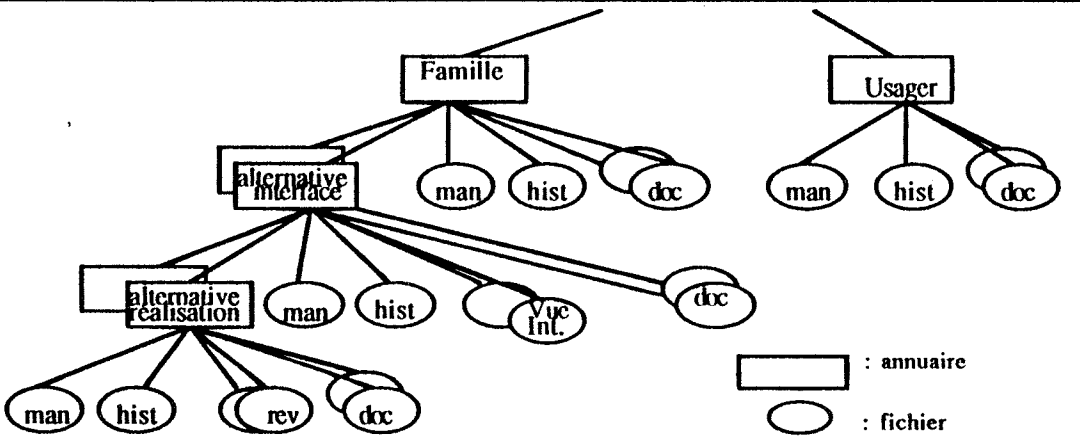


Figure 6.3 : Principe de la représentation interne des objets dans Nomade

3.2.3. La création des objets composés et atomiques

a) Cas des objets composés

C'est le cas des objets famille, usager, interface, réalisation. Ces objets sont créés par la primitive : "*create objet*". La syntaxe de l'objet indique le type de l'objet à créer.

- Pour une famille F est créée la relation "*struct*" entre la famille nom-base et F.
- Pour un usager U est créée la relation "struct" entre la famille User et l'usager U.

Ces relations ne peuvent pas être retirées; elles assurent que toute famille ou usager a une partition de référence.

Exemple :

```
create F     : crée la famille F dans le projet courant,
create U     : crée un usager dans le projet courant,
create F-I   : crée l'interface I pour la famille F,
create --C   : crée la réalisation C pour l'interface et la famille par défaut.
```

c) Cas de la famille projet

Un projet est créé par la primitive "*creatbase nom-projet*".

Un projet est une famille possédant comme prérogative d'être la racine de toutes les partitions. C'est la seule famille capable de déclarer des relations structurantes. la primitive *creatbase* met en place une structure à base de trois familles: la famille *project*, et deux familles qui lui sont reliées

par la relation *struct* : la famille du nom de projet (*Basename*) et la famille *user* (voir chapitre 4 §6.)

b) Cas des objets atomiques

Les objets atomiques sont créés implicitement par les opérations de catalogage. Les objets atomiques "manuels" et "historiques" sont créés implicitement lors de la création de l'objet composé auquel ils appartiennent.

Exemple :

Soit la primitive : **catal F-I-C.specif**

Si le document de nom "specif" n'existe pas, Nomade produit le message suivant :

```
Le document specif n'existe pas dans la réalisation F-I-C.
Désirez-vous créer spécif(O/N) :
```

3.3. La structure de l'objet descripteur manuel

3.3.1. La représentation externe du Manuel

La principale structure de données manipulée par le noyau Nomade est le manuel. Un manuel est un objet particulier de type descripteur représenté par un fichier Unix. Les manuels sont le support de description des types structurés et recueillent toutes les informations nécessaires à la gestion de la base. Ce sont des objets consultables et éditables (seulement une partie) par les utilisateurs. Chaque famille, usager, interface et réalisation est représentée par un manuel. Pour exécuter une commande, le gestionnaire de la base parcourt le manuel où il trouve l'ensemble des informations concernant un objet.

- Le manuel d'une famille a la structure suivante :

```
        manuel famille
        family

attributes
        relations
        actions
partition /* cas de partition */
        attributes
        relations
        actions
        ...
selections
        selimp
        selcond
        seldef
library
I-attributes
namext
```

Les informations du bloc *family* et *partition* ont été précisées dans le chapitre 4. Le mot-clé *library* indique que cette famille est une bibliothèque, donc que ses interfaces sont toutes considérées comme étant compatibles c'est à dire qu'un nombre quelconque d'entre elles peuvent faire partie d'une même configuration.

Le bloc *sélection* est utilisé par le configurateur, et pour la détermination des objets par défaut (*seldef*). Le bloc *namext* est défini dans §4.1. Le bloc *I-attributes* définit les attributs instanciés.

- Le manuel d'un usager a la structure suivante :

```
        manuel usager
User
        attributes
        relations
        rights
                actions
partition /* cas de partition */
        attributes
        relations
        rights
        actions
        ...
I-attributes
namext
```

- Le manuel des interfaces et des réalisations a la structure suivante

```
manuel interface/ réalisation

selections
        selimp
        selcond
        seldef
sans-corps/* cas d'interface */
I-attributes
namext
```

3.3.2. La représentation interne du manuel

Les manuels, une fois compilés, sont représentés en mémoire sous forme d'une structure de donnée constituée de plusieurs listes. La figure 6.4 montre la représentation interne en mémoire centrale du contenu d'un manuel compilé.

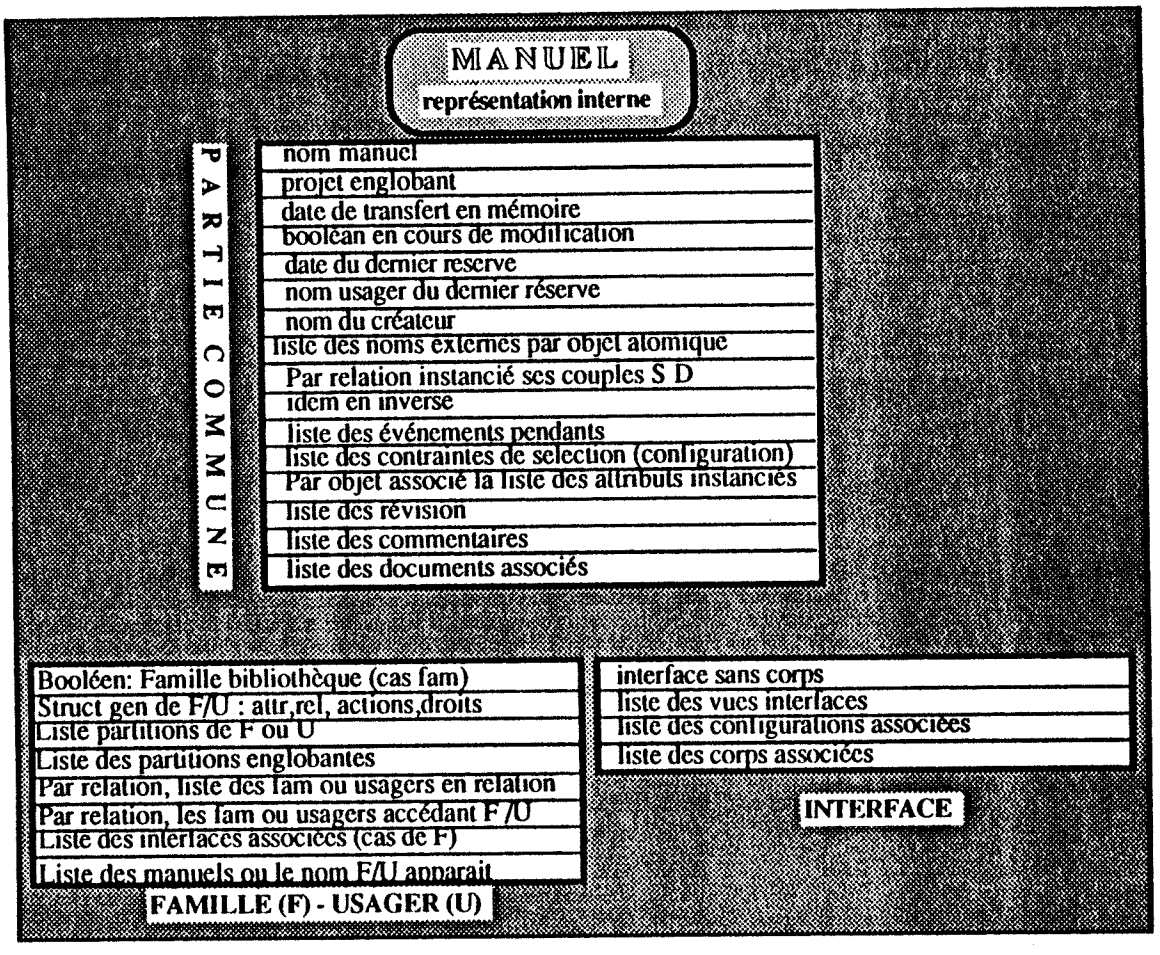


Figure 6.4 : Structure interne du Manuel

Cette représentation retient principalement les informations suivantes:

a) *Pour tous les objets composés famille usager interface et réalisation:*

- des informations générales communes à l'ensemble des objets : nom du manuel, l'auteur de sa création, le projet auquel appartient l'objet décrit par le manuel, un booléen indiquant si le manuel a été modifié, la date de dernière réservation et le nom de l'usager ayant réservé l'objet;

- Par objet atomique, une liste de noms externes (cf § 4.1).

- la liste représentant les relations instanciées. Pour chaque relation instanciée, le nom de l'objet source et destination est conservé;

- la liste représentant les événements pendants. C'est la liste des événements E(S, R, D, C, U, Date) enregistrés par la fonction *record* et en attente de traitement sur accès ou par la commande *Make*.

- les listes représentant les contraintes de sélection à appliquer lors de l'évaluation d'une configuration. Elles correspondent à la représentation interne des clauses *selimp, selcond et seldef*;

- la liste des attributs instanciés. Par objet associé sont conservés les attributs instanciés;

- la liste des révisions dont chaque élément de liste est composé du nom de la révision, des attributs prédéfinis *state, stateconf, date et author* de la révision, de la liste des configurations englobantes, de la liste des codes objets et de la liste des dépendances.

- la liste des noms de documents associés ;

- la liste des commentaires;

b) *Contenu du manuel spécifique à une interface*

- une valeur représentant l'information que l'interface est définie sans corps associé;

- une liste des noms des vues d'interfaces;

- une liste des noms de configurations associées à l'interface;

- une liste des noms de corps associés à l'interface.

c) *Contenu du manuel spécifique à une famille et usager* [1]

- une valeur booléenne indiquant que la famille est de type bibliothéque (si famille);

- la description des types structurés Famille ou Usager comprenant :

  - par type d'objet, une liste de définitions d'attributs;

  - une liste représentant la définition de relations. chaque élément de la liste donnant le nom de la relation et les deux expressions nomade définissant les domaines de la source et de la destination de la relation;

[1] La même structure que famille est allouée pour la structure usager. Les informations n'ayant aucun sens pour la structure usager sont réservées mais non exploitées.

- une liste représentant la définition des droits dans le cas d'un usager. Chaque élément de liste donnant un identificateur de droits ou le mot-clé d'arrêt de l'héritage "stopinh" et deux listes associées : une liste d'expressions nomades liées implicitement par l'opérateur logique "or" définissant les droits propres à l'usager et une liste identique mais de droits hérités.

- des listes représentant les programmes d'actions, une liste par forme de déclenchement d'actions (commandes, post-actions, obligations, définitions d'actions en accès et définitions d'actions sur demande). La structure des cinq listes est définie de manière identique. chaque élément représente de façon interne la structure : *"For R(C) do if Expression do action"*. La structure comprend : le nom de la relation, le nom de l'identificateur de commande et une liste de couple (expression, action).

- une liste représentant les déclarations des types structurés de partitions dans le cas de manuels d'usagers ou de familles racines de partition. Chaque définition correspond à une définition de partition. En effet, une même famille ou usager peuvent être racines de plusieurs partitions différentes.

- une liste des noms de partitions englobantes.

- une liste de relations dont la famille ou l'usager sont les sources.

- de manière inverse une liste de relations dont la famille ou l'usager sont les cibles.

- la liste des noms des interfaces associées (si famille) ;

- la liste des manuels où le nom de la famille apparaît (si famille).

La syntaxe et la description détaillée du manuel est présentée en annexe 2.

4. LES AUTRES ASPECTS DE NOMADE

Nous allons essayer de décrire briévement ici les aspects abordés dans le cadre du développement d'Adèle et Nomade.

4.1. La gestion des noms externes

Il s'agit de la gestion de la correspondance entre les noms d'objets atomiques dans la base et les noms de fichiers dans l'espace privé de l'usager (l'annuaire de l'usager). Les commande *"read"* et *"catal"* ont deux paramètres : le nom d'un objet atomique dans Nomade et le nom d'un fichier local dans l'annuaire de l'usager courant. Le plus souvent, le nom du fichier local dépend uniquement de l'objet élémentaire qui va lui être affecté : le couple (objet Nomade, nom fichier local est stable). La liste "namext" dans chaque manuel définit les couples (nom objet atomique/nom fichier local) pour les objets atomiques gérés par ce manuel. Plusieurs noms de fichiers peuvent être associés à un même objet atomique.

Les commandes suivantes permettent de gérer ces associations :

- *adnamext nom-objet nom_fichier* : établit une correspondance entre *nom-objet* et *nom_fichier*
- *namext nom-objet nom_fichier* : répond *"vrai"* si cette association existe sinon rend la liste des noms de fichiers associés à *nom-objet* si *nom_fichier* est omis.

La stratégie par défaut de gestion des noms pour les commandes *"read"* et *"catal"* est la suivante :

- si le nom du fichier est omis, Nomade prend par défaut le premier nom de fichier associé au nom Nomade de la commande.

4.2. Le contrôle multi-usager

C'est la possibilité d'utiliser la base d'objets par plusieurs utilisateurs. Toutes les informations utilisées par Nomade et Adèle pour la réalisation des activités de gestion de la base sont stockées dans le *"manuel"*. Pour une commande donnée, les manuels nécessaires sont lus en mémoire et verrouillés. Les conflits d'accès interviennent lorsque des usagers travaillent simultanément avec Nomade et par conséquent accèdent ou modifient les mêmes manuels. Cette constatation montre que le contrôle multi-usager se situe au niveau du contrôle de l'accès multiple aux manuels. Il est effectué en plaçant des verroux sur les manuels en cours de consultation ou de modification par un usager. Ces verroux sont des liens Unix de nom *"man.LCK"* sur les fichiers manuels dans l'annuaire les contenant. Ils sont détruits après que le manuel correspondant soit libéré. Tout accès à un manuel sur lequel est positionné un verrou échoue. Un message est envoyé à l'usager indiquant que l'objet est bloqué. Ce message est répété trois fois toutes les cinq secondes. En cas de non débloquage la commande en attente est défaite.

4.3. Implantation de la gestion des deltas et historiques.

Notre hypothèse sur l'immutabilité des objets a nécessité la mise en place d'un mécanisme pour minimiser l'occupation de l'espace disque. Ce mécanisme appelé mécanisme de deltas est appliqué aux révisions dont sont conservées les différences successives, la dernière révision étant stockée en clair. L'implantation est basé sur les deux outils d'Unix *"diff"* et *"ed"*. L'outil *"Diff"* permet l'analyse de différences entre deux fichiers; le résultat est donné sous forme d'une liste de commandes éditeur *"ed"* . Ce dernier permet, lors d'une demande de révision, de la restituer en clair en appliquant les suites de commandes à partir de la dernière révision. Nous avons adapté ces deux outils à nos besoins et avons fait deux modules dans la base appelés sous forme procédurale pour plus d'efficacité.

Par ailleurs, des traces des catalogages dans la base pour tous les types d'objets et de détermination de l'origine d'une réalisation lorsqu'elle est créée à partir d'une autre réalisation sont conservées automatiquement dans un fichier spécial de type historique associé à chacune des entités famille, usager, interface et réalisation.

4.4. Pannes et procédures de reprise

Pour une commande réussie, les manuels modifiés sont recopiés sur disque, sinon ils sont ignorés. Lorsque les manuels sont recopiés sur disque les interruptions sont désarmées mais des incohérences peuvent intervenir en cas de panne de système. Pour cette raison un mécanisme de reprise en cas de défaillance logicielle et matérielle a été développé. Nous présentons les grandes lignes du mécanisme basé sur les copies double d'objets avec :

- une copie active en cours de modification;
- une copie stable composée de valeurs cohérentes.

Le passage d'une copie active à une copie stable est contrôlé par une procédure de validation.

Pour chaque usager *"U"* un fichier *"2U"* est créé avant le transfert des manuels modifiés en mémoire secondaire. Initialement le fichier contient un statut positionné à la valeur "avant validation" de la commande et une liste de manuels à recopier en mémoire secondaire. Chaque manuel est recopié en mémoire secondaire dans un fichier temporaire *"&man"* dans l'annuaire de son original. Lorsque tous les manuels ont été recopiés en mémoire secondaire, le fichier *"2U"* est modifiéc c'est à dire que la valeur du statut prend la valeur "validé". Puis les fichiers *"&man"* sont transformés en manuels réels *"man"*. Finalement le fichier *"2U"* est détruit.

Cette procédure de validation permet de déterminer, en cas de panne, l'état du système et de décider s'il faut poursuivre ou défaire la commande.

La procédure de reprise est mise en oeuvre à chaque initialisation si le fichier "2U" existe ou après une erreur majeure à l'exécution d'une commande. Elle est destinée à remettre la base dans un état cohérent. Elle se déroule de la manière suivante:

- si le fichier "*2U*" a un statut ayant la valeur "avant validation", les fichiers &man sont détruits,
- si le fichier "*2U* " a un statut ayant la valeur "validé", les fichiers *"&man"* restants sont recopiés dans les fichiers manuels correspondants.

Une reprise explicite pour toute la base ou pour un usager particulier est faite par la commande *restaure*. La reprise explicite ne peut se faire que sous l'identité de l'usager privilégié.

La méthode présente l'avantage qu'en cas de panne, il n'est pas nécessaire de défaire les mises à jour mais il suffit d'ignorer la commande en ignorant les manuels avant validation ou de refaire la dernière étape après la validation.

5. ETAT DE DEVELOPPEMENT

La réalisation de Nomade n'est pas terminée. Les développements entrepris sur Adèle ont pris du temps pour améliorer la portabilité et la performance (trois réécritures). Après la stabilisation d'une version d'Adèle pour une diffusion pré-industrielle, nous avons continué le développement de Nomade par l'extension progressive des fonctionnalités d'Adèle. Les concepts présentés ici ont été testés soit avec Adèle soit séparément. Il s'agit plus précisemment des aspects suivants :

- La structure de partition, les documents et la généralisation des relations ont été testés et intégrés dans Adèle;
- les droits d'accès ont été réalisés et testés séparément;
- Une première approche de l'accès à distance à une base de programme a été testé et intégré dans Adèle pour un dialogue entre des programmes écrits en Prolog et Adèle.
- Une maquette simplifiée du mécanisme des événements-actions avec répartition des clients et des usagers sur des sites distants a été développée séparément pour mieux comprendre ce type de système.

Comme Nomade est réalisé dans le cadre d'une collaboration avec des industriels initiée avec les équipes génie logiciel de Bull et Norsk-Data (Norvège), tous les concepts testés sont actuellement repris, intégrés et complétés pour la fabrication d'un véritable système opérationnel où les contraintes de fiabilité, d'évolution et la taille importante du logiciel à maintenir et des équipes sont parmi les critères importants. En effet, le noyau Nomade doit remplacer à terme le système Adèle qui est actuellement utilisé par ces même industriels.

CONCLUSION ET PERSPECTIVES

En guise de conclusion, nous décrivons les aspects abordés dans cette étude puis nous situons la contribution de notre travail par rapport aux travaux actuels dans ce domaine. Enfin, nous présentons les perspectives et les nouveaux thèmes de développement et de recherche abordés actuellement dans notre équipe et dont le noyau Nomade est l'élément fédérateur.

1. ASPECTS ABORDES DANS L'ETUDE DE NOMADE

Nous avons présenté la base de programmes Adèle puis décrit le noyau Nomade issu d'Adèle. Le principe directeur de nos travaux est de proposer une approche permettant de traiter de façon cohérente divers aspects de la programmation globale. Ce travail résulte de notre expérience avec Adèle et de son utilisation en milieu industriel. Les concepts et les mécanismes d'Adèle ont été réétudiés et étendus.

Les aspects étudiés dans nos travaux sont liés à la gestion des objets logiciels, au contrôle des effets propagés des différentes actions, à l'intégration et l'activation d'outils, à la structuration du logiciel et au contrôle de son évolution. Les critères de développement privilégiés sont la flexibilité pour intégrer les outils et méthodes que l'usager souhaite utiliser et la possibilité de définir et d'imposer des stratégies de gestion du logiciel et les contraintes utilisées dans toute organisation de projet. Plus précisément, nous avons essayé dans nos travaux d'apporter quelques réponses concrètes aux questions suivantes:

- Comment spécifier les composants d'un logiciel pour pouvoir le maintenir et le faire évoluer?
- Quels sont les mécanismes permettant de contrôler l'évolution d'un logiciel?
- Un logiciel est un ensemble d'entités dépendantes les unes des autres. Comment représenter cet ensemble?
- Quels sont les mécanismes nécessaires pour apprécier les impacts d'une modification d'un composant sur le logiciel?
- Comment expliciter les règles de gestion et les contraintes de développement et rendre par conséquent l'environnement plus général?

- Comment faire coopérer les efforts d'une équipe de développement?
- Comment supporter l'activation d'outils?
- Comment faire en sorte que les manipulations s'inscrivent dans une logique d'évolution et ne soient pas une séquence de changements apparemment incohérents entrainant la destructuration du logiciel et à plus ou moins long terme son abandon?
- Comment intégrer les réponses aux questions précédentes dans un noyau d'environnement?

2. CONTRIBUTION DE NOTRE TRAVAIL

Cette étude essaye d'apporter une solution à des aspects de la programmation globale. Le but principal de Nomade est de proposer un modèle pour la construction, l'intégration, le contrôle et l'évolution de logiciels et de leurs équipes. Nous allons essayé de dégager briévement la contribution et les résultats de cette étude.

2.1. La base d'objets

Nous proposons le développement d'une base de données construite à partir d'un modèle de support à la programmation globale. Ce modèle est dérivé du modèle entité-relation binaire et intègre quelques caractéristiques "orientées objet". Notre modèle est bâti autour de concepts décrivant plusieurs caractéristiques de la programmation globle notamment la structure du logiciel, les modules, les configurations et les versions, les opérations et les informations nécessaires au support de l'activité de maintenance et d'évolution . L'utilisation de la base de données permet de tracer tous les événements déclenchés dans la base; de garder un historique des opérations et des modifications, de contrôler les actions exécutées sur les objets, la structure du logiciel et les droits des usagers. La base de donnée Nomade est spécialisée pour la programmation globale. Cette spécialisation nous a amenés à prédéfinir dans le modèle les types structurés module (Famille) et Usager. Les travaux sur Nomade rejoignent les approches actuelles de développement de base de données spécialisées pour le génie logiciel car l'expérience a montré les limites des SGBD classiques concernant l'efficacité et la puissance de modélisation. Les travaux menés actuellement mettent à profit les méthodes de bases de données notamment les modèles sémantiques et les méthodes de représentation et de réutilisation des connnaissances issues des travaux en intelligence artificielle [Barstow87]. Les connaissances sur les processus de développement sont intégrées de manière explicite sous forme non procédurale pour augmenter le niveau de généralité et de flexibilité tout en maintenant l'efficacité.

2.2. La structuration du logiciel

Les problèmes de la programmation globale notamment la destructuration du logiciel nous ont amenés à fournir des mécanismes de structuration, de protection et de contrôle. La notion de partition est définie dans Nomade. Une partition correspond à un sous-système tel qu'un

composant appartienne à une seule partition de même type. Il est possible d'offrir une vue unifiée d'un composant en le considérant soit comme un module simple, soit comme un sous-système soit comme une partition d'objets complexes d'une manière complètement transparente. Les partitions doivent permettre de contrôler la structure du logiciel. L'utilisation des partitions multiples et de l'héritage devra être évaluée dans des bases d'objets de taille importante.

2.3. Le mécanisme d'événements actions

Nous nous sommes penchés sur les aspects liés au contrôle des activités logicielles notamment la propagation des effets des changements qui est un problème crucial dans les environnements de programmation globale à cause de la forte inter-dépendance des données. Nous avons développé un mécanisme d'événements-actions permettant de rendre la base active et de l'intégrer dans l'environnement de développement et de maintenance. Le mécanisme d'événements-actions est utile pour définir différentes stratégies pour le contrôle de la propagation des changements, pour exprimer des contraintes de cohérence complexes, pour régler les dépendances objets-outils et pour imposer des procédures de travail. L'évaluation du comportement du mécanisme en vrai grandeur avec des logiciels de taille importante et l'utilisation de différentes stratégies de gestion ainsi que le développement d'un véritable langage d'actions plus élaboré restent à faire.

Ce travail aborde des aspects importants que l'on retrouve dans la plupart des études actuelles comme la gestion de la cohérence, l'intégration d'outils et le contrôle des activités dans les environnements de développement de logiciel de taille importante et qui demeurent des problèmes ouverts.

2.4. La gestion de versions et des configurations

La gestion des versions comprend la gestion des historiques, le contrôle des développements parallèles et la gestion des *deltas* . Nous n'insistons pas sur ces aspects qui sont actuellement bien compris. L'aspect intéressant est l'intégration de la gestion de versions dans la base; elle n'est plus accomplie de manière indépendante. Nomade rejoint les travaux actuels ou la gestion des versions est intégrée dans la base et en plus généralisée à l'ensemble des objets. Cependant il reste une voie ouverte liée à la gestion de versions et que nous n'avons pas abordée dans Adèle et Nomade concernant la fusion automatique de versions actuellement en investigation dans de nombreux travaux.

Pour ce qui concerne la gestion de configuration, l'idée de construction automatique de configurations d'Adèle et Nomade à partir d'une description de logiciel en termes de contraintes sur les versions plutôt qu'une liste de composition, est reprise actuellement dans la plupart des travaux. La configuration est évaluée à partir du graphe de dépendance et de la description des modules. Dans les approches classiques un objet spécial, le *system model* est utilisé pour définir les composants et l'architecture d'un logiciel.

2.5. La distribution

Nous avons initié une première expérience qui n'est pas entièrement validée et qui a consisté à développer des accès distants à un serveur de base logicielle avec la possibilité de répartir les activités de gestion logiciel sur le réseau local. Cependant nous avons gardé une certaine homogénéité puisque les environnements (systèmes et outils) des différents sites sont identiques. L'utilisation des stations de travail avec des outils et des systèmes hétérogènes est actuellement en investigation dans de nombreux travaux.

2.6. Discussion

Les réalisations et les prototypages ont montré la faisabilité des mécanismes proposés. Cependant, il semble un peu tôt pour juger des résultats de l'expérience car l'implémentation de Nomade n'est pas achevée et des améliorations et des validations restent à faire; notre but étant d'obtenir un noyau performant utilisable dans un milieu industriel.

3. PERSPECTIVES

A partir de cette étude, des activités de développement et de recherche centrées sur Adèle et Nomade ont démarré ou sont en voie d'initialisation. Elles son conduites selon deux axes :

a) Des activités de recherche

De nouveaux horizons sont apparus suite à cette étude. On peut résumer comme suit les principaux travaux envisagés:

- Des améliorations et extensions à apporter à Nomade. Ce travail comporte entre autres des études sur :
  - la fusion de versions à partir des informations stockées dans les historiques et les deltas;
  - l'adjonction d'une interface usager avec l'utilisation des possibilités de l'interface graphique;
  - le langage d'action actuellement de bas niveau doit être remplacé par la définition d'un langage plus élaboré inspiré des travaux sur Odin et Marvel;
  - la généralisation du modèle Nomade pour supporter la définition de différents types d'objets structurés et en versions multiples. Le but recherché est une plus grande couverture du cycle de vie du logiciel.

- Une ouverture vers d'autres projets dont Nomade constitue la structure d'accueil. Nous présentons ici briévement les principales études amorcées.

- Les méthodes de conception. L'appoche choisie prétend résoudre les problèmes de la liaison conception-maintenance. Ce travail qui a démarré porte sur l'intégration dans la base des informations qui proviennent de l'étape de conception. L'objectif est de conserver durant la maintenance le lien entre les objets logiciels et les entités utilisées durant la conception. Pour ce faire les diverses conceptions doivent être traduites en modèle Nomade.

- Le contrôle des interfaces. C'est un premier travail qui démarre avec l'étude plus fine du contenu des interfaces. Ce travail comprend la définition d'un forme interne commune aux interfaces et indépendante des langages de programmation. Ceci permet d'effectuer des vérifications de types inter-module et multi-langages, de consulter des références croisées inter-modules (généralisation du Masterscope d'Interlisp pour le rendre multi-langages).

- En relation avec la généralisation du modèle, une étude est menée sur l'étude des problèmes liès à la documentation technique avec l'implantation d'un gestionnaire de document sur Nomade en collaboration avec l'équipe Systèmes d'Informations (IOTA) du LGI-IMAG.

b) Des activités d'industrialisation et de développement d'outils.

Des partenaires industriels se sont intéressés à nos travaux. Des plans de développements avec les sociétés Bull et Norsk Data sont actuellement effectifs et où Adèle est utilisé comme précurseur. Ce travail a pour prolongement des améliorations et extensions importantes à apporter à Adèle et Nomade.

Pour conclure, il faut dire que les points énoncés ainsi que leur intégration constituent un plan de recherche ambitieux et cohérent qui devrait amener Nomade vers un environnement de développement et de maintenance et pour lequel il reste beaucoup à faire car la recherche dans ce domaine est loin d'être close.

ANNEXE 1:

LE LANGAGE DE PRESENTATION DU MANUEL

Cette annexe regroupe les règles de syntaxe décrites au chapitre 4 et 5 utilisées pour la présentation du manuel :

<manual> ::= **manual** <objectype>
[<Attributes-clause>]
[<Relations-clause>]
[<Rights-clause>]
[<Actions-clause>]
{< Partition-schema>}

- Dans le cas de Famille et Usager

[<Selimp-Clause >]
[<Selcond-Clause >]
[<Seldef-Clause >]

- Les clauses suivantes de selection concerne le gestionnaire de configuration.

{ <Other> }

- Informations pour qualifier une famille de type bibliothèque ou une interface sans corps

end

<Objectype> ::= **Family | Interface | Realization | User** *- objets avec manuels*

< Partition-schema> ::= **partition** <Relation>
[<Attributes-clause>]
[<Relations-clause>]
[<Rights-clause>]
[<Actions-clause>]

<Relation> ::= <identifier>

<Selimp-Clause > ::= **Selimp** <NomadeExpr>
<Selcond-Clause > ::= **Selcond** <NomadeExpr>
<Seldef-Clause > ::= **Seldef** <NomadeExpr>

<Other> :: = **Library | Withoutbody**

<NomadeExpr> ::= <LsOrLsAndQN> | not (<LsOrLsAndQN>)
<LsOrLsAndQN ::= <LsAndQN> | (<LsAndQN>) { **or** (<LsAndQN>) }
<LsAndQN> ::= <QualN> { **and** <QualN> } | **not** (<QualN> { **and** <QualN>})

<QualN> ::= [**not**] <Name> [(<LsAndQual>)]
<LsAndQual> ::= <Qualification> { **and** <Qualification> }
<Qualification> ::= <name_Attribute> <RelOp> <Value>
<value> :: = **$U** <name_attribute> | <identifier>
<name_attribute> :: = <identifier>
<RelOp> ::= < | > | <= | >= | <> | =
<Name> ::= [<partition>] [<simple_name>]
<partition> ::= / <Relation> • <struct-obj>/
<struct-obj> ::= <Family> | <User>
<simple_name>::= ::= <Object>] [• [<Element>] | [<User>][• <Document>]
<Family> ::= <identifier>
<User> ::= <identifier>
<Object> ::= <Family> [**:** <Interface> [**:** <Realization>]]
<Element> :: = <Document> [• <Revision>] | <Revision> |
<Revision> ::= **1..999**

<Attributes_clause> ::= attributes { **def** <NomadeExpr > <list_attributes>}
<list_attributes> ::= { <name_attribute > = [=] <value> { , <value> } ; }
<nam-attribute> ::= <identifier>
<value> ::= <identifier>
<identifier> ::= <L>| <métachar> { <L> |<C> | <métachar>}
<métachar> ::= ***** | **?** | **[** | **]**
<L> est l'ensemble des lettres y compris les caractères '-' et '_ ' et <C> l'ensemble des chiffres.

<Relations-Clause> ::= **relations** { <Source> <Relation> <Destination>;}
<Source> :: = <NomadeExpr> | **reflex**
<Destination> ::= <NomadeExpr>

<relation> ::= <identifier>

<Rights-clause> ::= **rights** {<Label> **:** <NomadeExpr> {, <NomadeExpr>};}

<Label> ::= <identifier>

<Actions-clause> ::= **actions** [**command** { < ForStatement> }]
[**post-actions** { <ForStatement> }]
[**obligation** { <ForStatement> }]
[**demand** { <ForStatement> }]
[**access** { <ForStatement> }]

<ForStatement> ::= **for** <Relation> [(<command>)]
do ({IfStatement>} | {< Action>}) **done**;

<IfStatement> ::= **if** <Condition> **then** <Action> **fi**;

<Relation> ::= <identifier>

<Condition> ::= <NomadeExpr> | [**not**] (Nomade-function)

<Action> ::= {<Nomade-function>;}

<Nomade-function> ::= <Name-function> [(<string>)]

<Name-function> ::= **exec** | **rexec**| **system** | **propagate** | **display** | **displayconf** | **record** | **option** |
reset | **fail** | **exit**

ANNEXE 2 :

LA REPRESENTATION DE LA STRUCTURE DE DONNEES MANUEL

Cette annexe présente la structure de donnée décrivant la structure interne du manuel après sa compilation :

```
const
        CNLg        = 64 ;          (*Max lengh of a name /R.F/F:I:C.doc.004 *)
        CLongNLg    = 128 ;         (* for real names of files & buf. only     *)
        CFalse      = 0 ;           (* booleans represented by integers        *)
        CTrue       = 1 ;
        CDifferent  = 0 ;           (* relational op. represented by integers  *)
        CEqual      = 1 ;
        CInferior   = 2 ;
        CSuperior   = 3 ;
        CInfEq      = 4 ;
        CSupEq      = 5 ;

        CNot        = 0 ;
        CYes        = 1 ;
        CStopInh    = 2 ;
        CGenN       = 3 ;
        CScript     = 4 ;
        CCommand    = 5 ;

        CUnknown    = -1 ;
        CUser       = 0 ;
        CFamily     = 1 ;
        CInterface  = 2 ;
        CRealization= 3 ;
        CBody       = 4 ;
        CConfiguration= 5 ;
        CTextConf   = 6 ;
        CObject     = 7 ;
        CElement    = 8 ;
        CDocument   = 9 ;
        CRevDoc     = 10 ;
        CObjCode    = 11 ;
        CHistory    = 12 ;
        CTextBody   = 13 ;
        CTextInt    = 14 ;
        CText       = 15 ;
        CManual     = 16 ;
        CManSel     = 17 ;
        CManDefAttr= 18 ;
        CManRel     = 19 ;
        CManAct     = 20 ;
        CManRight   = 21 ;
```

```
        CManPart   = 22 ;
        CManAttr   = 23 ;
        CManOther  = 24 ;                (* library, no body, included*)

type
        ObjectGen  = CUser .. CRealization ;
        NumRel     = CEqual .. CSuperior ;
        NumRelOp   = CDifferent .. CSupEq ;
        PtN        = ^NElem ;            (* pointers          *)
        Pt2N       = ^R2NElem ;
        PtQualN    = ^QualNElem ;
        PtRel      = ^RelElem ;
        PtRight    = ^RightElem ;
        PtGenN     = ^GenNElem ;
        PtLsAndQN  = ^LsAndQNElem ;
        Pt2GenN    = ^R2GenNElem ;
        PtForRel   = ^ForRelElem ;
        PtPart     = ^PartElem ;
        PtEvent    = ^EventElem ;
        PtRev      = ^RevElem ;
        PtMan      = ^ManElem ;
        TyN        =  packed array [CNLg] of char ;
        TyLongN    =  packed array [CLongNLg] of char ;

NElem = record
        Next       : PtN ;
        Name       : TyN ;
        IntegVal   : Integer ;
        LsRev      : PtN ;               (* not saved on disk *)
end ;

R2NElem = record
        Next       : Pt2N ;
        Name       : TyN ;               (* source family (space)  base relation attribute *)
        Value      : TyN ;               (* target locname (space) author family value    *)
        IntegVal   : Integer ;           (* level (relations)      rel.op.          *)
end ;

RelElem = record
        Next       : PtRel ;
        Name       : TyN ;               (* relation ; element          *)
        Pair       : Pt2N ;              (* source & target  of the rel.        *)
                                         (* attributes of element       *)
end ;

QualNElem = record                       (* [NOT] Name (LsAndQual)          *)
        Next       : PtQualN ;
        Name       : TyN ;
        No         : Integer ;
        LsAndQual  : Pt2N ;
end ;

LsAndQNElem = record
        Next       : PtLsAndQN ;
        No         : Integer ;
        LsAndQN    : PtQualN ;           (* list of QualN's linked by AND's  *)
end ;
```

```
GenNElem = record                    (* [NOT] GenN ; [NOT] script ; [NOT] cmd ; StopInh *)
        Next          : PtGenN ;
        No            : Integer ;          (* NOT ; StopInh      *)
        LsOrLsAndQN: PtLsAndQN ;           (* list of LsAndQN's linked by OR's*)
end ;

R2GenNElem = record                  (* IF GenN/command/script THEN GenN/command/script*)
        Next          : Pt2GenN ;
        Condition     : GenNElem ;         (* GenN or text or NOMADE Cmd  *)
        Action        : GenNElem ;         (* GenN or text or NOMADE Cmd  *)
end ;

RightElem = record                         (* Identifier : GenN ' OR GenN ' own      *)
                                           (*              GenN ' OR GenN ' inh      *)
        Next          : PtRight ;
        Name          : TyN ;              (* ident. of a right ; "stopinh"          *)
        LsOwnGenN: PtGenN ;                (* own GenN's , linked by impl. OR's *)
        LsInhGenN  : PtGenN ;              (* inh GenN's , linked by impl. OR's *)
        InhPoss       : Integer ;          (* work - variable ; not saved on dk *)
        Already       : Integer ;          (* work - variable ; not saved on dk *)
end ;

ForRelElem = record                        (* For R (A) DO (IF GenN THEN GenN)  *)
                                           (* Def R (source, target)      *)
        Next          : PtForRel ;
        Name          : TyN ;              (* name of the relation        *)
        Action        : TyN ;              (* name of the action          *)
        Ls2GenN       : Pt2GenN ;          (* IF GenN THEN GenN           *)
end ;

PartElem = record                          (* block partition             *)
        Next          : PtPart ;
        Name          : TyN ;              (* name of the partition rel.  *)
        LsAttrDef     : PtRel ;            (* attribute definitions       *)
        LsRelDef      : PtForRel ;         (* relation definitions        *)
        LsRights      : PtRight ;          (* rights definitions          *)
        LsCmd         : PtForRel ;         (* command definitions         *)
        LsPostAct     : PtForRel ;         (* post action definitions     *)
        LsObl         : PtForRel ;         (* obligation action definitions*)
        LsAcc         : PtForRel ;         (* access action definitions   *)
        LsDem         : PtForRel ;         (* demand action definitions   *)
end ;

EventElem = record
        Next          : PtEvent ;
        Name          : TyN ;              (* S   *)
        Relation      : TyN ;              (* R   *)
        Target        : TyN ;              (* T   *)
        Action        : TyN ;              (* A   *)
        Date          : TyN ;              (* D   *)
        Author        : TyN ;              (* U   *)
        FreeN         : TyN ;
end ;

RevElem = record
        Next          : PtRev ;
        Name          : TyN ;
        State         : TyN ;
```

```
        StateConf    : TyN ;
        Date         : TyN ;          (* creation date          *)
        Datem        : TyN ;          (* date last change in user directory *)
        Author       : TyN ;
        Free         : Integer ;
        LsFreeN      : PtN ;
        LsEmbConf    : PtN ;
        LsObjCode    : PtN ;
        Lsdocrev     : PtN;
        LsDep        : PtN ;
end ;

ManElem = record
        Next         : PtMan;
        Name         : TyN ;          (* name of the manual          *)
        Project      : TyN ;          (* embedding base              *)
        ReadDate     : Integer ;      (* hour of read from disk      *)
        Empty        : Integer ;      (* empty boolean               *)
        Modified     : Integer ;      (* Bool  Modified manuel in central memory *)
        ResDate      : TyN ;          (* date of last reservation    *)
        ResUser      : TyN ;          (* user of last reservation    *)
        Author       : TyN ;          (* creator name                *)
        Language     : TyN ;          (* free                        *
        Future2      : TyN ;          (* free                        *)
        LsNamext     : PtN ;          (* list of external names for each element*)
        LsRel        : PtRel ;        (* S T pairs for each relation *)
        LsRelInv     : PtRel ;        (* idem inversed               *)
        LsEvents     : PtEvent ;      (* list of pending events      *)
        LsFree       : PtN ;          (* free list                   *)
        LsImpSel     : PtGenN ;       (* list of imposed selections  *)
        LsCondSel    : PtGenN ;       (* list of cond. selections    *)
        LsDefSel     : PtGenN ;       (* list of default selections  *)
        LsAttr       : PtRel ;        (* list of attributes          *)
        LsRev        : PtRev ;        (* list of revisions           *)
        LsComm       : PtN ;          (* list of comments            *)
        LsDoc        : PtN ;          (* list of associated documents*)

        case Manual : ObjectGen of
        CBody , CConfiguration : () ;
        CInterface   : (
              NoBody       : Integer ;   (* interface without body     *)
              MainView     : TyN ;       (* embedding interface        *)
              LsViews      : PtN ;       (* list of included interfaces *)
              LsConf       : PtN ;       (* list of associated conf.   *)
              LsBodies     : PtN ;       (* list of associated bodies  *)
              LsFree       : PtN ;       (* free list                  *)
                    ) ;
        CFamily , CUser    : (
              Library      : Integer ;   (* boolean value              *)
              DefFam       : PtPart ;    (* dcl. of attr, rel , act & rights for this fam *)
              LsDefPart    : PtPart ;    (* list of defined part.      *)
              LsEmbPart    : Pt2N ;      (* list of emb. part. : (/R.F/)*)
              LsRelFam     : PtN ;       (* list of fam rel. to source F for each rel *)
              LsRelFamInv  : PtN ;       (* list of fam rel. to target F for each rel *)
              LsInt        : PtN ;       (* associated interfaces      *)
              LsInvConstr  : PtN ;       (* list of man. to which Name belongs *)
                    ) ;
 end ;
```

REFERENCES BIBLIOGRAPHIQUES

[Alperin 87] L. B. Alperin, B. I. Kedzierski.
AI-Based Software Maintenance.
IEEE AI Applications Conference, February 1987.

[Balzer 86] R.M. Balzer.
Living in the next Generation of the Operating System
Proceedings of the 10th Word computer Congress, Dublin, IFIP-Sept- 1986.

[Barstow 87] D. Barstow.
Artificial Intelligence and Software Engineering.
Proc. of the 9th Int. Conf. on Soft. Eng., Monterey, CA, March 1987.

[Batory 86] D.S. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C. Twichell and T.E. Wise.
GENESIS: A Reconfigurable Database Management system', Tech. Rep. 86-07, Department of Computer Sciences, The University of Texas at Austin, 1986.

[Belkhatir 85] N.Belkhatir, J.L. Cheval.
Un Modèle de Gestion de Gros Logiciels en Versions Multiples : ADL2
Actes des journées AFCET-Informatique. Nouveaux langages pour le Génie Logiciel Cnam-Evry, 28 et 29 octobre 1985.

[Belkhatir 86a] N. Belkhatir, J. Estublier.
Protection and Cooperation in a Software Engineering Environment.
IFIP WG2.4 International Workshop on Advanced Prog. Env., Trondheim Norway, 16-18 June 1986. Springler Verlag(ed.) LNCS 244, Feb. 1987.
Présenté également à la 3ème Conférence-Exposition de Génie logiciel CGL3- Versailles- 27-30 mai 1986. pp 69-79.

[Belkhatir 86b] N.Belkhatir, J. Estublier.
Experience with a Data base of Programs.
Proc. of the ACM SIGPLAN/SIGSOFT Soft. Eng. Symposium on Practical Soft. Dev. Env.. Palo Alto December 1986. SIGPLAN Notices Vol. 22, No 1 Jan. 1987. pp 84-91. Also submitted for publication in ACM-TOPLAS.

[Belkhatir 87] N. Belkhatir, J. Estublier.
Software Management Constraints and Action Triggering in Adele Program Database. ESEC'87-1st European Soft. Eng. Conf., 9-11 sept 1987-Strasbourg-France pp 47-57. in John N. Buxton (ed.).

[Belkhatir 88] N. Belkhatir, J. Estublier.
Nomade : A Kernel for Programming In The Large.
IFIP WG2.4 Dauville-France fevrier 1988.

[Bernard 87] Y. Bernard, M. Lacroix, P. Lavency, M. Vanhoedenaghe.
Configuration Management in an Open Environment .
ESEC'87-1st European Soft. Eng. Conf.. 9-11 sept 1987-Strasbourg-France, in John N. Buxton (Ed.).

[Bernstein 87] Ph. A. Bernstein.
Database System Support for Software Engineering. An Extended Abstract.
Proceedings of the 9th International Conference on Soft. Eng., IEEE, Monterey, CA, March 1987.

[Blattner 84] M.M. Blattner, K.I Joy, B. E. Kelly, D.L. Zimmerman.
An Environment for the Integration of Existing Software Tools.
Lawrence Livermore National Laboratory and The University of California, Davis, November 13, 1984. UCRL-91752.

[Boehm 76] B. W. Boehm.
Software Engineering.
IEEE Transactions on Computers. December 1976- Vol. C-25 No 12.

[Boehm 82] B.W. Boehm and al.
The TRW Software Productivity System.
Proceedings of The 6th Int. Conf. on Soft. Eng., Tokyo 1982.

[Borison 86] E. Borison.
A Model of Software Manufacture.
IFIP WG2.4 Int. Workshop on Advanced Prog. Env . Trondheim Norway, 16-18 June1986. Springler Verlag(ed.) LNCS 244, Feb. 1987.

[Borgida 84] A. Borgida, J. Mylopoulos, H. K. T. Wong.
Generalisation/Specialisation as a basis for software specification.
On conceptual Modelling, Perspectives from Artificial Intelligence, Databases and Programming Langages, M.L. Brodie, J. Mylopoulos and J.W. Schmidt, Edts Springer Verlag, New York, pp 87-114.

[Brodie 84] M.L. Brodie.
On the development of Data models.
On conceptual Modelling, Perspectives from Artificial Intelligence, Databases and Programming Langages, M.L. Brodie, J. Mylopoulos and J.W. Schmidt, Edts Springer Verlag, New York, pp 87-114.

[Bunemann77] O.P. Bunemann, H.L. Morgan.
Implementing Alerting Techniques in Database Systems.
Dept of Decision Science. The Wharton School, University of Pensylvania, PA 19174, 1977.

[Bunemann 79] O. Bunemann, E. Clemons.
Efficiently Monitoring Relational Databases.
ACM-TODS. Sepember 1979.

[Buxton 80] J.N. Buxton (Ed.).
Stoneman : Requirements for ADA Programming Support Environments.
Tech. Rep., U.S. Department of Defense. February 1980.

[Carey 85] M. Carey, D. Dewitt.
Extensible Database System.
Proceedings of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems. February 1985.

[Cartmell 87a] J. Cartmell, A. Alderson.
The Eclipse Data Model - A Functional Account.
KEELE Conf. on Soft. Eng. Env.. Ellis Horwood, April 1987.

[Cartmell 87b] J. Cartmell, A. Alderson.
The Eclipse Two-Tier Database Interface.
ESEC'87-1st European Software Engineering Conference. 9-11 sept 1987-Strasbourg-France. in John N. Buxton (ed.).

[Ceri 83] S. Ceri, S. Crepi-Reghizi.
Relational Databases in the Design of Program Construction Systems.
SIGPLAN Notices. Vol.18, No 11, November 1983.

[Chen 76] P. Chen.
The Entity -Relationship Model. Toward a Unified View of Data.
ACM-TODS, Vol.1, No1, March 1976.

[Clemm 86] G.M. Clemm.
The Odin system : An Object Manager for Extensible Software Environments. CU-CS-314-86, The University of Colorado, Boulder, CO 80309, February, 1986.

[Clemm 88] G.M. Clemm.
The Odin Specification Language.
Int. Workshop on Software Version and Configuration Control. 27-29 January- Grassau-Germany-J. F.H. Winkler (Ed.).

[Codd 70] E. Codd.
A Relationnal Model of Data For Large Shared Data Banks.
Comm. ACM, Vol.14, No 6, pp 377-387, 1970.

[Codd 79] E. Codd.
Extending the Database Relationnal Model to Capture More Meaning.
ACM-TODS, Vol.4, No 4, 1979.

[Cooprider 79] L.W. Cooprider.
The Representation of Families of Software Systems.
Phd Thesis, Carnegie-Mellon University, Computer Science Department, April 1979. CMU-CS-79-116.

[Delobel 82] C. Delobel, M. Adiba.
Base de données et systèmes relationnels
DUNOD (ed.), Paris 1982.

[De-Remer 76] F. De Remer, H. H. Kron.
Programming-In-The-Large versus Programming-In-The-Small.
IEEE Transactions on Software Engineering, SE-2(2): pp 80-86, June 1976.

[Dittrich 86a] K.R Dittrich, A. M. Ktz, J.A. Mulle.
An Event / Trigger Mechanism to Enforce Complex Consistency Constraints.
ACM SIGMOD RECORD, Vol. 15, No 3, September 1986.

[Dittrich 86b] K. R. Dittrich, W. Gotthard, P. C. Lockemann.
DAMOKLES. A Database System for Software Engineering Environments.
IFIP WG2.4 Int. Workshop on Advanced Prog. Env. . Trondheim Norway, 16-18 June1986. Springler Verlag(ed.) LNCS 244, Feb. 1987.

[DOD 83] U.S. Department of Defense. Ada Joint Project Office.
Reference Manual for the ADA Programming Language.
ANSI/MIL-STD-1815A- 22 January 1983.

[DOD 85] U.S. Department of Defense.
Requirements and Criteria for CAIS II. 1985.

[Donzeau 80] V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang.
Programming Environments Based on Structured Editors : the Mentor Experience.
Proc. Workshop on Programming Environments, Ridgefield, Conn. 1980.

[Estublier 83] J. Estublier, S. Krakowiak, J. Mossière, Y. Rouzaud.
Design Principles of the ADELE Programming Environment.
International Computing Symposium on Application Syst. Dev., Nuremberg. March 1983.

[Estublier 88] J. Estublier.
Configuration Management : the Notation and the Tools.
Proceedings of the Int. Workshop on Software Version and Configuration Control. 27-29 January 1988, Grassau-FRG, J. F.H. Winkler (Ed.).

[Eswaran 76] K. Eswaran.
Specifications, Implementations and Interaction of a Trigger Subsystem in an Integrated Database System.
IBM research, RJ 1820, San Jose, Ca. , August 1976.

[Fauvet 87] M.C. Fauvet, D. Rieu.
CADB : un Système de Gestion de Bases de Données et de Connaissances pour la CAO. MICAD, 6ème Conférence, Paris, février 1987.

[Feiler 86] P. Feiler, G.E. Kaiser.
Granularity Issues in a Knowledge Based Programming Environment.
Technical Memorandum SEI-86-TM-11,September 1986, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213.

[Feldman 79] S. I. Feldman.
Make - A Program for Maintening Computer Programs.
Software Practice and Experience, Vol. 9, No 3, March 1979.

[Feldman 88] S.I. Feldman.
Evolution of Make.
Proceedings of the Int. Workshop on Software Version and Configuration Control. 27-29 January 1988, Grassau-FRG, J. F.H. Winkler (Ed.).

[Gallo 86] F. Gallo, R. Minot, I. Thomas.
The Object Management System of PCTE as a Software Engineering Database Management System.
Proc. of the ACM SIGPLAN/SIGSOFT Soft. Eng. Symposium on Practical Soft. Dev. Env.. Palo Alto December 1986. SIGPLAN Notices Vol. 22, No1, January 1987.

[Godart 86] C. Godart, K. Benali, J.C. Derniame.
Propositions pour un Système de Gestion d'Objets.
3ème Colloque-Exposition de Génie Logiciel-CGL3, 27-30 mai 1986, Versailles-France.

[Goldstein 80] I. Goldstein.
Description for a Programming Environment.
Proceedings of the 1st Annual Conference of the American Association for Artificial Intelligence, AAAI, 1980.

[Habermann 81] A. N. Habermann, D.E. Perry.
System Composition and Version Control for ADA.
Software Engineering Environments, Huenke (Ed.), Amsterdam- 1981.

[Habermann 86] A.N. Habermann, D. Notkins.
Gandalf : Software Development Environments.
IEEE Transactions on Software Engineering. Vol. SE-12, No12, Dec. 1986.

[Huff 81] K.E. Huff.
A Database Model for Effective Configuration Management in the Programming Environment.
Proceedings of the 5th Int. Conf. on Software Engineering, March 1981.

[Kaiser 83] G.E. Kaiser, A.N. Habermann.
An Environnement for System Version Control.
Proceedings of the 26th IEEE Computer Society International Conference, COMPCON '83., San Francisco, CA., February 1983.

[Kaiser 87a] G.E. Kaiser, P. H. Feiler.
Intelligent Assistance without Artificial Intelligence.
32nd IEEE Computer Society International Conference, pp 236-241, San Francisco, CA, February 1987.

[Kaiser 87b] G.E. Kaiser, P. H. Feiler.
An Architecture for Intelligent Assistance in Software Development.
9th International Conference on Software Engineering, pp 180-188, Monterey, CA, March 1987.

[Katz 83] R.H. Katz.
Transaction Management for Design Databases.
Computer Sciences Technical Report 496, February 1983.

[Katz 85] R.H Katz, T.J. Lehman.
Database Support for Versions and Alternatives of Large Design Files.
IEEE Transactions on Software Engineering SE-10, No 2, MArch 1984.

[Kernighan 81] B.W. Kernighan, J.R. Mashey.
The Unix Programming Environment.
IEEE Computer, 14, no 4, April 1981, pp 12-24.

[Krakowiak 82] S. Krakowiak.
Systèmes Intégrés de Production de Logiciels : Concepts et Réalisations.
Actes des Journées Bigre 82 Systèmes Intégrés de Production de Logiciels.27 au 29 Janvier 1982, Grenoble France.

[Krakowiak 85] S Krakowiak.
Principes des Systèmes d'Exploitation des Ordinateurs.
Dunod (Ed.), Paris, 1985.

[Lampson 83] B. W. Lampson, E.E. Schmidt.
Organizing Software in a Distributed Environment.
Sigplan Notices, June 1983.

[Lamsweerde 86] A. Van Lamsweerde and al.
The Kernel of a Generic Software Development Environment.
Proceedings of the ACM SIGPLAN/SIGSOFT Software Engineering Symposium on Practical Software Development Environments. Palo Alto December 1986. SIGLAN Notices Vol.22, No 1 January 1987.

[Lamsweerde 88] A. Van Lamsweerde and al.
Generic Lifecycle Support in the ALMA Environment.
IEEE Transactions on Soft. Eng., Vol. 14, No 6., June 1988.

[Leblang 84] D. Leblang, R. P. Chase.
Computer Aided Software Engineering in a Distributed Workstation Environment. Proceedings of the ACM SIGPLAN/SIGSOFT Software Engineering Symposium on Practical Software Development Environments. April 1984.

[Leblang 85] D. Leblang, R. P. Chase.
Configuration Management for Large Scale Software Development Efforts. Proceedings of the Workshop on Soft. Eng. Env. for Prog. in the Large, GTE Laboratories, Harwichport, Massachussets, June 1985.

[Leblang 88] D. Leblang, R. P. Chase.
Parallel Building : Experience with a Case for Workstations Networks.
Int. Workshop on Software Version and Configuration Control.
27-29 January, Grassau-FRG, J. F.H. Winkler (Ed.).

[Linton 84] M.A. Linton.
Implementing Relationnal Views of Programs.
Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, April 1984. ACM SIGPLAN Notices, Vol.19, No5, May 1984, pp 132-140

[Marzullo 86] K. Marzullo, D. Wiebe.
Jasmine : A Software System Modelling Facility.
Proceedings of the ACM SIGPLAN/SIGSOFT Sof. Eng. Symposium on Practical Soft. Dev. Env.. Palo Alto December 1986. SIGPLAN Notices Vol.22, No 1 January 1987.

[Medina 81] R. Medina-Mora, P. Feiler.
An Incremental Programming Environment.
IEEE Transactions On Soft. Eng., Vol. SE-7, No 5, September 1981.

[Medina 82] R. Medina-Mora.
Syntax-Directed Editing : Toward Integrated Programming Environment.
Carnegie-Mellon University, Tech. Rep. CMU-CS-82-113, (March 1982).

[Meyer 85] B. Meyer.
The Software Knowledge Base.
Proceedings of the 8th Int. Conf. on Soft. Eng. IEEE, August 1985, London, UK.

[Minsky 84] N. Minsky.
The Darwin Software Evolution Environment.
Proceedings of the ACM SIGSOFT/SIGPLAN Soft. Eng. Symposium on Practical Soft. Dev. Env., Pittsburgh, April 1984. ACM SIGPLAN Notices, Vol.19, No 5, May 1984.

[Mitchell 84] J.G. Mitchell and al.
Mesa Language Manual, Version 11.0.
Xerox Office Systems Division, June 1984.

[Narayanaswamy 87] K. Narayanaswamy, W. Scacchi.
Maintening Configurations of Evolving Software Systems.
IEEE Transactions on Soft. Eng., Vol. SE-13, No 3, March 1987.

[Nestor 86] J.R. Nestor.
Toward a Persistent Object Base.
IFIP WG2.4 International Workshop on Advanced Prog. Env., Trondheim Norway, 16-18 June1986. Springler Verlag(ed.) LNCS 244, Feb. 1987.

[Notkin 85] D. Notkin.
The Gandalf Project.
The Journal of Systems and Software, Vol. 5, No 2, May 1985.

[Osterweil 81] L. Osterweil.
Software Environment : Research Direction for the Next Five Years.
IEEE Computer, Vol. 14, No 4, April 1981.

[Osterweil 83] L. Osterweil.
Toolpack : An Experimental Software Development Environment Research Project. IEEE Transactions on Soft. Eng., SE-9, No 6, November 1983.

[Parnas 72] D.L. Parnas.
On the Criteria To Be Used in Decomposing Systems into Modules.
CACM, Vol. 15, No 12, December 1972.

[Parnas 79] D.L. Parnas.
Designing Software for Ease of Extension and Contraction. IEEE Transactions on Software Engineering, Vol. SE-5, No 2, March 1979.

[Peckham 88] J. Peckham, F. Maryanski.
Semantic Data Models.
ACM computing Surveys, Vol. 20, No 3, September 1988.

[Penedo 86] M.H. Penedo.
Prototyping a Project Master Data Base for Soft. Engineering Environments.
Proceedings of the ACM SIGPLAN/SIGSOFT Software Engineering Symposium on Practical Soft. Dev. Env., Palo Alto December 1986.
SIGPLAN Notices Vol. 22, No 1 January 1987.

[Perry 87] D.E. Perry.
Version Control in the Inscape Environment.
Proc. of the 9th. Int. Conf. on Soft. Eng., 30 March 1987, Montery- CA.

[Pfreundschuh 88] M. Pfreundschuh, R. Ford.
Using Attribute Grammar to Control Incremental Concurrent Build of Modular Systems.
Int. Workshop on Software Version and Configuration Control. 27-29 January- Grassau-Germany-J. F.H. Winkler (Ed.).

[Prietro-Diaz 86] R. Prietro-Diaz, J. M. Neighbors.
Module Interconnection Languages.
The Journal of Systems and Software, 6, pp 307-334, 1986.

[Ramamoorthy 86] C. V. Ramamoorthy, V.Garg, A. Prakash.
Programming in the Large.
IEEE Transactions on Software Engineering, Vol. SE-12, No 7, July 1986.

[Reps 88] T. Reps, S. Horwitz, J. Prins.
Support For Integrating Program Variants in an Environment for Programming in the Large.
Proceedings of the Int. Workshop on Software Version and Configuration Control. 27-29 January- Grassau-Germany-J. F.H. Winkler (Ed.).

[Rochkind 75] M. Rochkind.
The Source Code Control System.
IEEE Transactions on Software Engineering, Vol. SE-1, No 4, December 1975.

[Rohrbach 88] R. Rohrbach, Ch. Seiwald.
GALILEO : A Software Maintenance Environment.
Int. Workshop on Software Version and Configuration Control. 27-29 January- Grassau-Germany-J. F.H. Winkler (Ed.).

[Rudmik 86] A. Rudmik.
Choosing an Environment Data Model.
IFIP WG2.4 International Workshop on Advanced Programming Environments . Trondheim Norway, 16-18 June1986. Springler Verlag(ed.) LNCS 244, Feb. 1987.

[Shaw 86] M. Shaw.
Beyond Programming in The Large.
IFIP WG2.4 International Workshop on Advanced Prog. Env., Trondheim Norway, 16-18 June1986. Springler Verlag(ed.) LNCS 244, Feb. 1987.

[Schwankle 88a] R.W. Schwanke, G.E. Kaiser.
Living With Inconsistency in Large Systems.
Proceedings of the Int. Workshop on Software Version and Configuration Control. 27-29 January, Grassau-FRG, J. F.H. Winkler (Ed.).

[Schwankle 88b] R.W Schwankle, G. E. Kaiser.
Smarter Recompilation.
ACM Trans. on Prog. Lang. and Systems, Submitted for publication.

[Smith 77] J.M. Smith, D.C.P. Smith.
Database Abstractions: Aggregation and Generalisation.
ACM Trans. Database Syst. Vol 2, No 2, 1977.

[Snodgrass 84] R. Snodgrass.
Monitoring in a Software Development Environment : A relationnal Approach. Proceedings of the ACM SIGSOFT/SIGPLAN Soft. Eng. Symposium on Pract. Soft. Dev. Env.. SIGPLAN Notices, Vol.19, No 5 May 1984.

[Snodgrass 86] R.Snodgrass, K. Shannon.
Supporting Flexible and efficient Tool Integration.
IFIP WG2.4 International Workshop on Advanced Prog. Env., Trondheim Norway, 16-18 June1986. Springler Verlag(ed.) LNCS 244, Feb. 1987.

[Stonebraker 76] M. Stonebraker, E. Wong, P. Kreps, G. Held.
The Design and Implementation of INGRES.
ACM-TODS, Vol. 1, No 3, September 1976.

[Stonebraker 86] M. Stonebraker, L.A. Rowe.
The design of POSTGRES.
Proc. of the ACM SIGMOD Conf. on Management of Data, June 1986.

[Stuebing 84] H.G. Stuebing.
A Software Engineering Environment for Weapon System Software.
IEEE Transactions on Software Engineering, Vol. SE-10, No 4, July 1984.

[Swinehart 85] D.C Swinehart, P.T. Zellweger, E.B. Hagmann.
The Structure of CEDAR.
Proceedings of the ACM SIGPLAN 85 Symposiim on Language Issues in Prog. Env. (Jun.1985). SIGPLAN Notices 20,7, July 1985.

[Teitelbaum 81] T. Teitelbaum, T. Reps.
The Cornell Program Synthesizer : A Syntax-Directed Programming Environment. CACM, Vol. 14, No 9, September 1981.

[Teitelman 84] W. Teitelman.
The Cedar Programming Environment : A Midterm report and Examination.
XEROX PARC, Rep CSL-83-11, 1984.

[Tichy 82] W.F. Tichy.
A Data Model For Programming Support Environment and its Application.
Automated Tools For Information System Design and Development.
A. I. Wasserman (Ed.), North Holland Publishing co., Amsterdam-1982.

[Tichy 85] W. F. Tichy.
RCS- a System for Version Control.
Software Practice and Experience, Vol. 15, No 7, July 1985.

[Tichy 86] W. F. Tichy.
Smart Recompilation.
ACM Transactions on Prog. Lang. and Systems, Vol.8, No 3, July 1986.

[Tichy 87] W. F. Tichy.
What Can Software Engineers Learn From AI?
IEEE Computer, Vol. 20, No11, November 1987.

[Tichy 88] W. F. Tichy.
Tools for Software Configuration Mangement.
Proceedings of the Int. Workshop on Software Version and Configuration Control. 27-29 January- Grassau-Germany-J. F.H. Winkler (Ed.).

[Wiebe 88] D. Wiebe.
Specifying and Verifying Semantic Properties of Software Configurations.
Proceedings of the Int. Workshop on Software Version and Configuration Control. 27-29 January- Grassau-Germany-J. F.H. Winkler (Ed.).

[Wile 86] D.S. Wile, D. G. Allard.
Worlds : an Organizing Structure for Object-Bases.
Proceedings of the ACM SIGPLAN/SIGSOFT Soft. Eng. Symposium on Practical Soft. Dev. Env.. Palo Alto December 1986. SIGPLAN Notices Vol.22, No 1 January 1987.

[Winkler 86] J.F.H. Winkler.
The Integration of Version Control into Programming Languages.
IFIP WG2.4 International Workshop on Advanced Prog. Env., Trondheim Norway, 16-18 June1986. Springler Verlag(ed.) LNCS 244, Feb. 1987.

[Wirth 83] N. Wirth.
Programming in Modula-2
Springler-Verlag, Berlin (1983)

[Zdonik 86] S. B. Zdonik.
Version Management in an Object-Oriented Database.
IFIP WG2.4 Int. Workshop on Advanced Prog. Env., Trondheim Norway, 16-18 June1986. Springler Verlag(ed.) LNCS 244, Feb. 1987.

TABLE DES MATIERES

CHAPITRE 3 UNE PREMIERE EXPERIENCE AVEC ADELE : UNE BASE DE GESTION DE LOGICIELS DE GRANDE TAILLE 55

LISTE DES FIGURES

INPG département des études doctorales

AUTORISATION de SOUTENANCE

VU les dispositions de l'article 15 Titre III de l'arrêté du 5 juillet 1984 relatif aux études doctorales

VU les rapports de présentation de Messieurs

. Axel van Lamsweerde , Professeur

. Yves Chiaramella , Professeur

Monsieur BELKHATIR Noureddine

est autorisé(e) à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, spécialité " Informatique "

Fait à Grenoble, le 22 novembre 1988

Pour le Président de l'I.N.P.-G.
et par délégation,
le Vice-Président
P. VENNEREAU

RESUME

Dans le développement et la maintenance de logiciels de grande taille, la programmation globale est l'activité la moins formalisée, la moins assistée alors que c'est l'activité prépondérante, celle qui consomme le plus de temps. Nous décrivons les principaux concepts et mécanismes de Nomade, un noyau d'environnement pour supporter le développement et la maintenance de gros logiciels. Nomade constitue une extension et une généralisation de son prédécesseur Adèle. Il fournit un modèle pour la construction, l'intégration et le contrôle des logiciels en versions multiples et de leurs équipes. Nous présentons essentiellement la base d'objets et les mécanismes de contrôle de la structure du logiciel avec la notion de partition et d'activation d'outils basée sur le concept d'événement action. La base d'objets est conçue autour d'un modèle adapté aux besoins de la programmation globale qui inclut des notions "orientées objet". La base conserve la structure du logiciel : les objets logiciels, leurs versions, leurs documents, leurs attributs, leurs relations et leurs objets dérivés. Le mécanisme d'événements actions s'inspire des mécanismes d'activations (déclencheurs, démons, exceptions) développés dans d'autres domaines. Nous montrons comment Nomade permet de maintenir les contraintes de cohérence complexes rencontrées dans la programmation globale, de définir et d'intégrer les stratégies et les outils spécifiques à un environnement de développement et de maintenance.

MOTS-CLES :
génie logiciel, programmation globale, maintenance de logiciel, environnement de génie logiciel, événement, action, déclencheur, bases de programmes, intégration d'outils.