50376
1999
241

NUMÉRO D'ORDRE: 2578 ANNÉE 1999

Laboratoire d'Informatique
Fondamentale de Lille

THÈSE

présentée à

L'UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE

pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

par

Gilles Vanwormhoudt

UNIVERSITÉ DES SC. ET TECHN. BIBLIOTHÈQUE UNIVERSITAIRE LILLE

CROME : un cadre de programmation par objets structurés en contextes

Thèse soutenue le 17 septembre 1999, devant la commission d'examen:

Président:	P. Mathieu	Université de Lille 1,LIFL
Directeurs de Thèse:	B. Carré	Université de Lille 1,LIFL
	J.M. Geib	Université de Lille 1,LIFL
Rapporteurs:	I. Borne	École des Mines de Nantes
	C. Dony	Université de Montpellier 2,LIRMM
Examinateurs:	C. Gransart	Université de Lille 1, LIFL
	F. Pachet	Sony CSL - Université Pierre et Marie Curie, LIP6

UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE
LIFL UPRESA CNRS 8022 Bât M3.
59655 Villeneuve d'Ascq CEDEX
Tél. 33 (0)3 20.43.47.24 Fax. 33 (0)3 20.43.65.66

A tous mes proches
"La vie sans eux ne vaut rien et rien ne vaut la vie avec eux"

Remerciements

Je tiens à remercier:

Bernard Carré, qui a été plus qu'un directeur de thèse pour moi. Son amitié, ses "points de vue" et son soutien pendant ces années m'ont beaucoup aidé à parcourir ce long chemin qui arrive aujourd'hui à sa fin. J'espère que cet ouvrage comblera une partie de mon immense dette à son égard.

Monsieur Philippe Mathieu, professeur à l'université de Lille I pour m'avoir fait l'honneur de présider ce jury.

Madame Isabelle Borne, professeur à l'Ecole des Mines de Nantes, et Monsieur Christophe Dony, maître de conférences habilité à diriger des recherches de l'université de Montpellier II, d'avoir accepté de juger ce travail et de m'avoir aidé à améliorer ce mémoire.

Monsieur François Pachet, maître de conférences habilité à diriger des recherches de l'université Pierre et Marie Curie, d'avoir bien voulu s'intéresser à mon travail en acceptant d' être examinateur.

Monsieur Christophe Gransart, maître de conférences à l'université de Lille I, pour l'intérêt porté à mes travaux et pour avoir accepter de participer à ce jury.

Monsieur Jean-Marc Geib, professeur à l'université de Lille I, pour m'avoir accueilli au sein de son équipe et pour la confiance qu'il m'a témoigné.

Laurent Debrauwer, autre apple-maniaque, pour sa gentillesse, sa disponibilité et sa collaboration précieuse.

Les autres membres de l'équipe Goal, Olivier, Philippe et Jean-François pour leur bonne humeur.

Cyrille d'Halluin, mon pote Lillois, pour ses idées farfelues et ses "Saturday night party" dont il a le secret.

Mes potes dunkerquois (Christophe, les deux Nicolas, les deux Stephane, Sam et Richard) avec lesquels nous passons tant de bons moment depuis des années.

Mon frère, ma soeur et toute leur petite famille pour leur complicité.

Mes parents pour leur affection et tout ce qu'ils m'ont donné depuis le debut.

Virginie, mon amour, pour tout ce que nous partageons chaque jour.

Léa, notre "petit bouchon", pour l'immense bonheur qu'elle nous apporte.

Enfin, je terminerai par une pensée pour Lenneke Dekker qui nous a malheureusement quittés peu de temps après la fin de sa thèse.

Table des matières

1

Introduction

Les années 90 ont vu l'explosion de l'utilisation de l'approche orientée objet pour la construction de logiciels aussi bien au niveau des méthodes de conception (OMT, UML) , des langages de programmation (C++, Java), des interfaces graphiques, des bases de données (ODMG) que des systèmes distribués (CORBA). Cette approche, qui aborde la conception de systèmes comme l'assemblage d'objets autonomes représentant les entités du problème, offre à la base des qualités bien connues [Meyer 88][Wegner 90] pour la structuration de logiciels complexes. Ces qualités découlent directement des notions fondamentales telles que l'encapsulation, l'héritage et le polymorphisme.

Toutefois, dans une approche exclusivement objet, il est souvent difficile d'identifier et à plus forte raison de concevoir et de maintenir les différentes fonctions du système. Ces difficultés proviennent d'une part de la distribution des fonctions au sein des objets que sous-tend la démarche. Elles sont dues d'autre part au fait qu'on dispose peu de moyens de structuration avenant à celles-ci [Kristensen 93]. De telles difficultés apparaissent clairement dès qu'il s'agit d'appliquer la démarche au développement d'environnements logiciel complexes de conception et d'ingénierie qui sont généralement organisés autour d'un référentiel d'objets et en fonctions manipulant ce référentiel de façon différenciée. Nous aurons l'occasion de revenir sur les difficultés posées dans ce cadre.

Notre objectif dans cette thèse est de pallier cette insuffisance de structuration de l'approche orientée objet sans pour autant rejeter ses principes fondamentaux et en perdre les bénéfices. Plus précisément, nous cherchons à enrichir l'approche afin de permettre une double décomposition des systèmes (et une double structuration des programmes) prenant compte à la fois des objets et des fonctions de ceux-ci.

Dans cette optique, nous proposons de retenir la notion de point de vue comme outil principal de cette double structuration. Dans les systèmes à objets, la notion de point de vue a souvent été utilisée pour répondre à des besoins de structuration supplémentaires sur les représentations. Ainsi, en représentation des connaissances par objets, plusieurs modèles ont intégré cette notion pour structurer les bases des connaissances et proposer un raisonnement classificatoire selon les points de vue de plusieurs experts [Bobrow 77][Dekker 94][Marino 93]. Dans le cadre des bases de données à objets, les techniques de vue ont été proposées pour structurer les espaces d'informations [Debrauwer 98]. En programmation par objets, qui constitue ici notre cadre d'étude, l'intégration des points de vue [Carré 90b][Harisson 93] a surtout visé à apporter une structuration plus importante des applications pour en augmenter la modularité.

Le travail réalisé dans ce mémoire porte sur l'étude d'une notion de point de vue adaptée à une double structuration objets/fonctions. Ce travail est issu de l'étude de la systématisation des points de vue de ROME [Carré 89] à un niveau de granularité supérieure à celui de l'objet par transversalité des fonctions.

Pour mener cette étude, nous partons du constat général d'orthogonalité entre objets et fonctions [Andersen 92], ce qui se traduit de façon homogène par la structuration interne et fonctionnelle des objets d'une part et dualement par la définition de chaque fonction

transversalement aux objets. Ce constat va nous permettre d'identifier les problèmes orthogonaux de structuration que cela pose et quelles sont les moyens faisant défaut dans les techniques existantes pour les résoudre.

Guidé par ces problèmes et les limites des approches existantes que nous serons amenés à analyser, nous proposons notre solution CROME (Contextes en ROME) qui permet une conception modulaire d'objets selon plusieurs contextes fonctionnels.

L'application de CROME conduit à identifier un référentiel d'objets et à considérer chaque fonction du système comme un contexte d'intervention de ces objets, réclamant de leur part des descriptions (attributs et méthodes) spécifiques. Une telle démarche est systématisée en CROME par une structuration en plans de la description des objets. Le référentiel est introduit par un plan de base sous la forme d'une hiérarchie de classe. Ce référentiel est ensuite enrichi selon les spécificités des fonctions. Pour chaque fonction, cet enrichissement s'effectue par un plan fonctionnel qui fournit pour chaque classe du plan de base concernée sa description dans le contexte. Une stratégie d'héritage modulaire basée sur les plans vient compléter cette structuration afin de déterminer la description des objets dans un contexte. Au sein d'un contexte, les objets se décrivent comme dans un programme objet classique ce qui permet d'en conserver les propriétés: abstraction de données, modularité entre objets, polymorphisme lié à l'envoi de message,... CROME prévoit également des moyens d'articulation entre les fonctions qui sont gérés au sein même des objets.

Une telle démarche permet d'une part de structurer fonctionnellement les descriptions des objets et d'autre part d'isoler orthogonalement les fonctions du système.

Le mémoire est organisé en quatre chapitres. Dans le premier chapitre, nous présentons la problèmatique de prise en compte de la multiplicité des fonctions dans un système d'objets. Cette problématique nous conduit à expliciter l'orthogonalité des objets vis à vis des fonctions et les nouveaux besoins de structuration que cela engendre selon chaque axe. Nous proposons d'examiner les approches existantes pour prendre en charge ces besoins. Dans un premier temps, nous nous focalisons sur les techniques offertes sous l'angle des objets pour structurer leur description. Ensuite, nous étudions les techniques offertes sous l'angle des fonctions, visant principalement à rendre compte de leur transversalité. Ceci nous amenera finalement à constater les limites de ces approches pour articuler conjointement les deux axes.

Notre proposition CROME fait l'objet du second chapitre. Nous y présentons les différents notions et principes mentionnés ci-dessus. D'autres aspects originaux de CROME résultant de l'application systématisée de ces principes y sont également présentés. Nous illustrons l'ensemble de l'approche au travers de l'exemple d'un système de conception de circuits logiques. Enfin, nous étudions les différentes propriétés de l'approche CROME sous l'angle de qualités logicielles comme la modularité, la réutilisabilité et l'extensibilité. Nous verrons notamment que cette approche conduit à l'existence d'une modularité interne aux objets et favorise la réutilisation des structures les organisant.

Le troisième chapitre est consacré à l'application des principes de CROME à ROME qui possède des capacités originales de représentation multiple par points de vue au niveau granulaire de l'objet. Après une présentation des capacités et des aspects réflexifs de ROME, nous précisons comment passer des points de vue aux contextes en les systématisant par transversalité. Cette application permet d'obtenir un langage de programmation par contextes d'objets qui reprend tous les principes et notions du cadre, en préservant ses propriétés. Enfin,

nous montrons comment les capacités réflexives de ROME sont exploitées pour l'implantation de CROME.

Les principes de structuration proposés par CROME sont généraux et se veulent applicables à des environnements variés. Dans le dernier chapitre, nous étudions l'application de CROME à Smalltalk pour obtenir un environnement de conception de classes orientée contexte. Contrairement à l'application à ROME, il s'agit bien ici de se situer au niveau de l'environnement de développement, sans conséquence opérationnelle sur le langage Smalltalk lui-même. Nous commençons par expliciter les difficultés posées pour obtenir une description structurée par contextes des classes Smalltalk. Nous proposons alors une solution qui repose sur une extension de la technique des catégories. Cette solution est exploitée pour offrir un flâneur orienté contexte, que nous présentons. Nous terminons par quelques expériences de programmation dans cet environnement.

2

Des objets et des fonctions

2.1 Introduction

Dans ce premier chapitre, nous cherchons à montrer les besoins de structuration liés à la multiplicité des fonctions dans un système d'objets ce qui constitue notre problématique principale. Cette problématique est abordée à travers l'étude de systèmes logiciels organisés autour d'un référentiel d'objets et de plusieurs fonctions centrées sur ce dernier. Cette étude va permettre de constater l'orthogonalité qui résulte de ces deux axes de structuration. Chaque axe ayant été étudié séparemment, nous les présentons successivement pour arriver à leur étude conjointe. Ceci nous permettra d'introduire des éléments de solution et notre approche CROME.

2.2 Problématique

Les systèmes logiciels où les besoins de structuration objets/fonctions sont particulièrement importants sont les environnements de conception et d'ingénierie tels que ceux rencontrés dans le domaine de la CAO [Gruber92][Trousse96], de l'édition [Decouchant94] ou des ateliers génie logiciel [Sommerville91]. Ces environnements sont généralement destinés à l'élaboration de produits complexes (circuits, objets mécaniques, logiciels, documents...). A ce titre, ils sont conçus pour supporter les différentes activités ou tâches (spécification, conception, analyse, simultation, vérification, test, ...) requises par cette élaboration. Au niveau de leur organisation, ces environnements possèdent une architecture type dans laquelle on distingue:

- d'une part, un référentiel d'entités structuré qui représentent les produits à élaborer. Ces entités sont de types variés et de granularité variable. Elles sont également reliées les unes aux autres à l'intérieur du référentiel formant ainsi des structures complexes, dynamiques (des entités y étant ajoutées et supprimées au fur et à mesure de l'élaboration du produit).

- d'autre part, un ensemble de fonctions[1] qui partagent et manipulent le référentiel (entités et structures) de façon différenciée. Dans le cadre de notre étude, nous allons nous restreindre au cas de fonctions ayant des objectifs complémentaires. Ces fonctions ont des besoins de descriptions spécifiques sur les entités du référentiel.

Pour illustrer cette organisation, considérons un exemple emprunté au domaine de la CAO électronique qui nous servira de support pour la suite de ce chapitre et tout au long du mémoire. L'exemple choisi est le cas d'un système d'aide à la conception de circuits logiques inspiré de ceux décrits dans [VanderMeulen 87] et [Hollant90][2]. Son organisation est schématisée à la figure 1.1.

---

1. Le terme de fonction est à prendre ici au sens de fonction globale d'un système [Meyer 88] et non au sens mathématique. Il faut également distinguer cette notion de fonction de celle correspondant à une unité de traitement en programmation fonctionnelle.
2. Le lecteur peut trouver une description plus détaillée de cet exemple dans le chapitre Annexe.

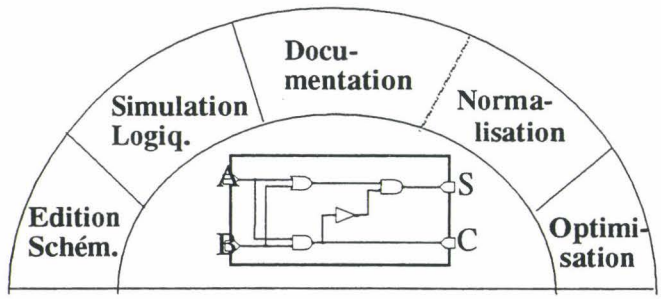


figure 1.1 : Plusieurs fonctions centrées sur un même référentiel

Nous pouvons y distinguer l'ensemble des fonctions supportées par ce système pour permettre une conception riche des circuits. Il s'agit en l'occurence des fonctions d'édition schématique, de simulation logique, de documentation (production de nomenclature), de normalisation et d'optimisation des circuits conçus. On peut noter que ces fonctions répondent à des objectifs différents et ne se recouvrent pas. La fonction d'édition schématique par exemple n'intervient pas directement dans la réalisation des autres fonctions.

Au centre, se trouve le référentiel des entités manipulé à travers l'ensemble des fonctions. Ici, ce référentiel représente les circuits logiques à concevoir. Il est constitué d'entités représentant ces circuits et des différents types d'entités entrant dans leur composition : portes logiques, connexions entre portes, broches des portes... Dans le cas présent, ces entités sont organisées de façon à réfléter l'assemblage en réseau des constituants d'un circuit.

Un système ainsi structuré doit pouvoir être conçu, développé et maintenu relativement à ses fonctions et non globalement. Ceci nécessite de pouvoir décrire chaque fonction séparément les unes des autres. De cette façon, le travail pourra être confié à des programmeurs distincts. Il sera également possible de spécifier, analyser et concevoir chaque fonction de façon modulaire, limitant ainsi la complexité du système.

Par la suite, il ne s'agit pas de présenter le système en soi mais de se focaliser sur la conception orientée objet des entités communes aux différentes fonctions.

La conception par objets offre à la base des qualités bien connues pour représenter les entités du référentiel de ces systèmes en fournissant tout le support nécessaire à leur description, leur création et leur manipulation. Plusieurs travaux ont donc cherché à appliquer l'approche objet pour la conception de ces systèmes en adoptant la plupart du temps une approche par point de vue sur les représentations de façon à rendre compte de la relativité des entités aux fonctions. Dans ce qui suit, nous présentons brièvement quelques uns de ces travaux situés dans le domaine de la CAO et dans celui du génie logiciel en insistant principalement sur l'organisation des systèmes concernés.

Dans le domaine de la CAO robotique, on peut citer le système Systalk [Wolinski 90] dont l'objectif est de fournir une plateforme logicielle commune aux différents spécialistes intervenant dans la conception de robots articulés. Ce système, développé en Smalltalk, supporte la construction de modèles de robots par assemblage des composants robotiques prédéfinis (barres, boites, prismes, pivots, glissières, vis, compas, rotules, ...) et permet l'évaluation de ces modèles selon des activités multiples (visualisation tri-dimensionnelle, calculs d'efforts, simulation de trajectoires, mesure cinématique,...).

Plusieurs études [Carn 92][Marcaillou 93] ont également appliqué la démarche objet pour le développement d'environnements de conception de systèmes spatiaux complexes (satellite, lanceur, téléscope spatial). [Deconninck 92] présente le développement par objets d'une partie de l'environnement Systema qui offre une dizaine d'analyses scientiques (analyse thermique, orbitographique, ..) d'un même système spatial afin de vérifier sa fiabilité avant sa fabrication. Ces différentes analyses sont intégrées autour d'un modèle objet unique basé sur la décomposition hiérarchique du système (sous-système, équipement) et détenant des descriptions spécifiques à chaque analyse.

Mentionnons aussi dans le domaine de la CAO pour l'architecture, l'application décrite dans [Najah 95] qui vise à assister l'architecte et les différents spécialistes dans l'élaboration de plans architecturaux de batiments. Cette application permet l'agencement des différents objets architecturaux (mur, fenêtre, paroi, plancher, ...) entrant dans la constitution d'un tel plan et couvre également les différentes études (thermique, accoustique, électrique, étanchéité, ...) qui sont à effectuer à partir du plan.

Dans le cadre du génie logiciel, [Shilling 89] aborde la conception par objets d'un environemment de développement centré sur des arbres syntaxiques abstraits représentant le programme. Ces derniers sont représentés à l'aide d'objets et sont manipulés à travers plusieurs outils de compilation, d'édition structurée, de mise en forme et d'analyse statique et dynamique.

Un autre travail s'incrivant dans le cadre du génie logiciel est celui présenté dans [Krief 91] autour d'un atelier CASE conçu pour supporter les stades correspondant à l'analyse et la conception de logiciels. Dans cet atelier, les logiciels sont spécifiés à l'aide de diagrammes SADT représentés par des objets et plusieurs outils différents sont proposés pour permettre l'édition graphique de ces diagrammes, leur évalution symbolique et partielle ainsi que la génération de squelette et l'édition de rapport.

En dernier lieu, citons les travaux relatifs à la "programmation par sujets[1]" initiée dans [Harrison 93] qui s'intéresse au développement par objets d'environnements constitués d'une famille d'applications fortement intégrées.

Nous reviendrons au cours du prochain chapitre sur quelques unes des techniques proposées dans ces différents travaux pour remédier aux problèmes de description que cela pose à l'approche objet.

L'étude de tels systèmes fait apparaître un constat d'orthogonalité entre les multiples objets représentant les entités du référentiel, et les multiples fonctions qui constituent autant de contextes de participation de ces objets. Ainsi, chaque objet du référentiel participe à l'ensemble (ou à un sous-ensemble) des fonctions. Dualement, chaque fonction fait participer l'ensemble (ou un sous-ensemble) de ces objets. Pour notre exemple, cette dualité peut être représentée par le diagramme de la figure 1.2. Les axes horizontaux correspondent aux fonctions et les axes verticaux aux objets. La présence d'un cercle à une intersection représente la participation de l'objet à la fonction.

1. "Subject-oriented programming"

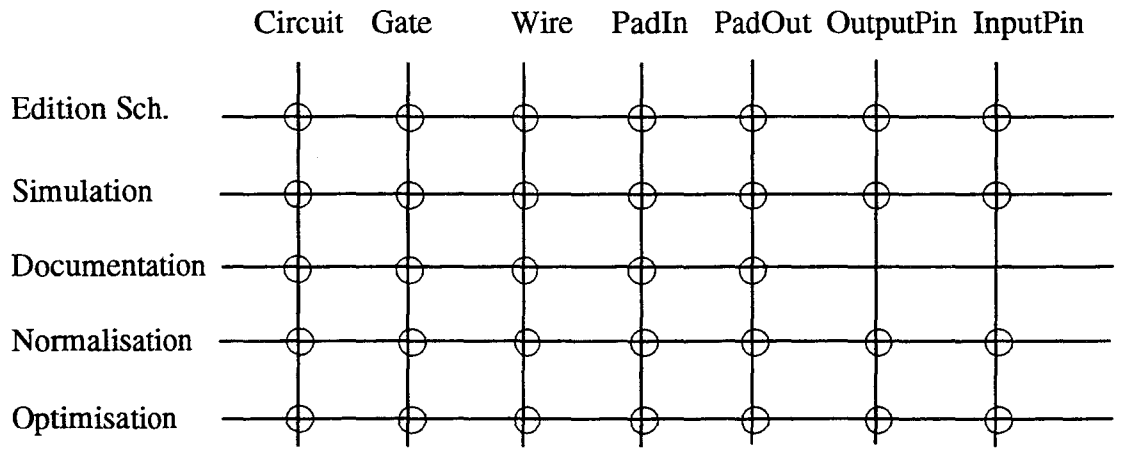


figure 1.2 : Orthogonalité objets/fonctions

Ce diagramme met en évidence l'orthogonalité entre objets et fonctions également évoquée dans [Andersen 92][1].

Verticalement, ce diagramme montre la participation multiple de chaque objet dans plusieurs fonctions. Par exemple, les objets circuits (`Circuit`) prennent ici part dans la totalité des fonctions du système. Il en est de même pour les objets représentant des portes logiques (`Gate`). Chaque participation d'objet à une fonction donne lieu à une description relative pour celui-ci. Ceci révèle le caractère multi-fonctionnel de tels objets.

Horizontalement, ce diagramme rend compte que chaque fonction constitue un contexte de participation pour plusieurs objets. Par exemple, la fonction de simulation fait intervenir toutes les entités. La fonction de documentation quant à elle ne fait pas intervenir les objets correspondant aux broches des portes (`InputPin`, `OutputPin`). La description de la fonction est diffusée dans les objets y contribuant. Ceci permet d'expliciter le caractère transversal des fonctions par rapport aux objets.

Par la suite, nous voulons rendre compte de ces deux axes de structuration et de leur articulation dans la description d'un système d'objets afin d'en améliorer la modularité. Ce qui nécessite de résoudre conjointement les problèmes suivants relatifs à chaque axe.

Sous l'angle des objets, il s'agit de structurer modulairement leur description selon les fonctions où ils participent.

Sous l'angle fonctionnel, il s'agit d'expliciter transversalement chaque fonction de façon à pouvoir circonscrire les objets y participant et leur description dans celle-ci.

Dans les deux prochaines sections, nous examinons successivement chaque problème pour lequel des débuts de solution ont été proposés séparement.

---

1. "In the functional approach, an activity is described orthogonal to the object involved in the activity while in the object-oriented approach objects are described orthogonal to the activities in which they are involved"

2.3 Sous l'angle des objets

2.3.1 Rappel de la démarche objet

Nous rappelons selon [Meyer88][Ferber 91][Riel 94] le changement de perspective et de topologie des systèmes offert par la conception par objets en comparaison à la conception fonctionnelle. Ce rappel nous permettra de bien positionner le problème de la conception de plusieurs fonctions sur un même référentiel d'objets.

L'approche orientée objet repose sur une acception unique du composant logiciel: l'objet disposant de données, sachant effectuer des traitements et communiquant par messages. La démarche objet aborde la conception de systèmes comme un assemblage d'objets autonomes dont l'existence est directement liée aux entités du problème. Par conséquent, la question première est d'identifier les entités du problème avant de s'intéresser aux fonctions du système. En cela, elle s'oppose à la conception fonctionnelle prenant pour point de départ les fonctions du système.

Dans notre exemple, nous identifions un référentiel constitué d'objets représentant des circuits (`Circuit`), des portes logiques Et (`AndGate`), des porte logiques Ou (`OrGate`), des portes logiques Non (`NotGate`), des équipotentielles (`Wire`), des entrées et des sorties de portes (`InputPut` et `OutputPin`), des ports d'entrée et sortie du circuit (`PadIn` et `PadOut`).

Une fois les objets du problème identifiés, la conception des fonctions que doit assurer le système peut être entreprise. L'idée centrale est de faire réaliser ces fonctions par les objets eux-mêmes [Jalote 89]. On y parvient en leur rapportant chaque fonction. Cela se fait en commençant par déterminer parmi tous les objets, ceux qui sont concernés par la fonction considérée. On procède ensuite à la décomposition de la fonction à travers ces objets. Cela revient à ranger dans chaque objet la partie de la fonction qui le concerne, le dotant ainsi de données et de traitements relatifs qui lui sont relatives. Finalement, suite à cette distribution, il reste à obtenir le comportement global souhaité pour la fonction ce qui nécessite la recomposition des traitements locaux fournis par les objets. Cette recomposition est réalisée par coopération de ces derniers.

Par exemple, dans la fonction d'édition schématique, considérons la distribution de la fonctionnalité d'affichage des circuits. Celle-ci est réalisée par les objets Circuit et leurs constituants (portes, équipotentielles) en laissant ces derniers gérer leur propre affichage. Pour les portes, leur affichage est obtenu en rendant leurs broches responsables à leur tour de leur affichage. La fonctionnalité d'affichage est ainsi diffusée au travers des objets du référentiel selon les relations de composition. Les autres fonctionnalités de la fonction (recherche de l'objet du circuit correspondant à une position graphique, vérification du positionnement d'une porte, ...) sont obtenues similairement. La figure 1.3 montre le résultat de ces distributions fonctionnelles dans les objets concernés.

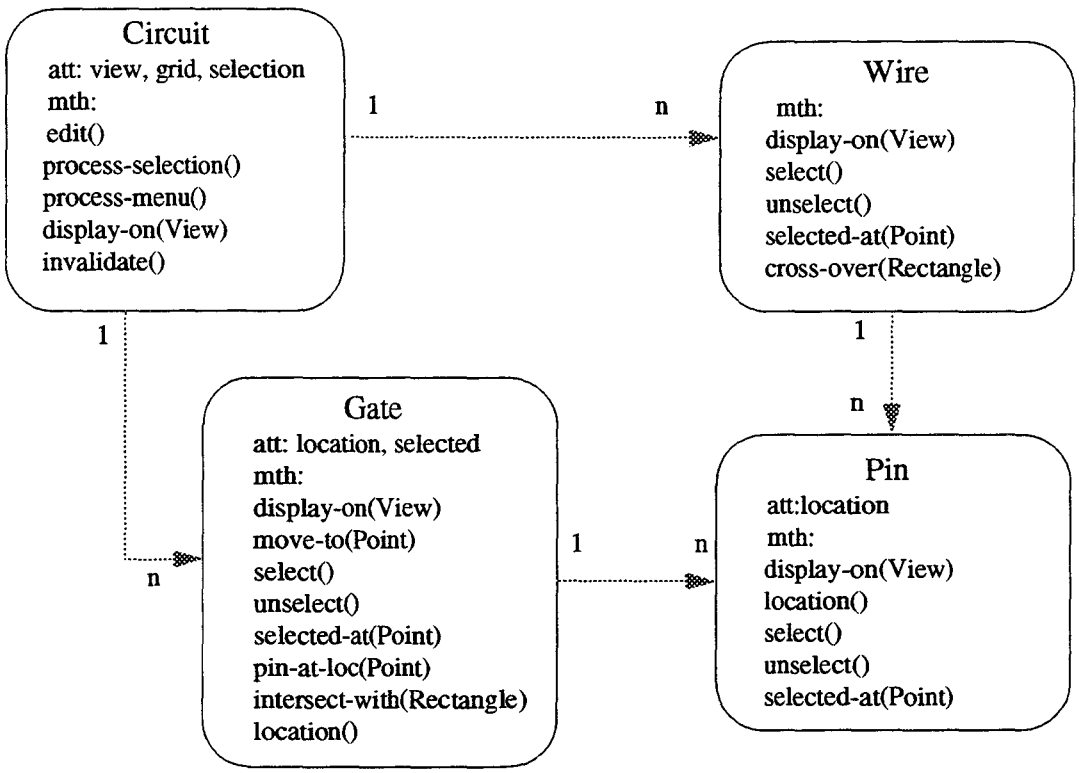


figure 1.3 : Distribution de la fonction d'édition

La conception des différentes fonctions du système revient à itérer la démarche précédente pour chacune d'elle. Ceci amène à décrire chaque fonction rapportée aux objets du référentiel, dotant ces derniers de fonctionnalités (attributs et méthodes) et les faisant intéragir relativement à celle-ci. Notons qu'en raison de la nature disjointe des fonctions, ces fonctionnalités ne se recouvrent pas.

Pour notre exemple, nous avons déjà montré à la section précédente la participation des objets Circuit et de ses constituants dans la fonction d'édition schématique. Considérons de façon analogue la fonction d'évaluation des circuits. Celle-ci est prise en charge, chacun pour leur part, par les objets `Circuit`, `Gate` (évaluation), `Wire` (propagation des signaux), et `Pin` (mémorisation des états).

Les autres fonctions de notre exemple peuvent être conçues selon une démarche analogue, faisant également participer les objets `Circuit`, `Gate`, ... à celles-ci. Précisons toutefois que l'intervention de l'ensemble des objets du référentiel dans chaque fonction n'est pas systématique. Ainsi, la fonction de documentation par exemple ne nécessite pas l'intervention des objets représentant les broches des portes. Cette non-participation des objets à une fonction se traduit pas l'absence de description correspondante.

L'organisation finalement obtenue est un ensemble d'objets qui prennent part dans plusieurs contextes fonctionnels. Ces objets sont directement issus des entités du référentiel, détiennent des descriptions pour chaque fonction et intéragissent localement à celles-ci. La figure 1.4 schématise cette organisation.

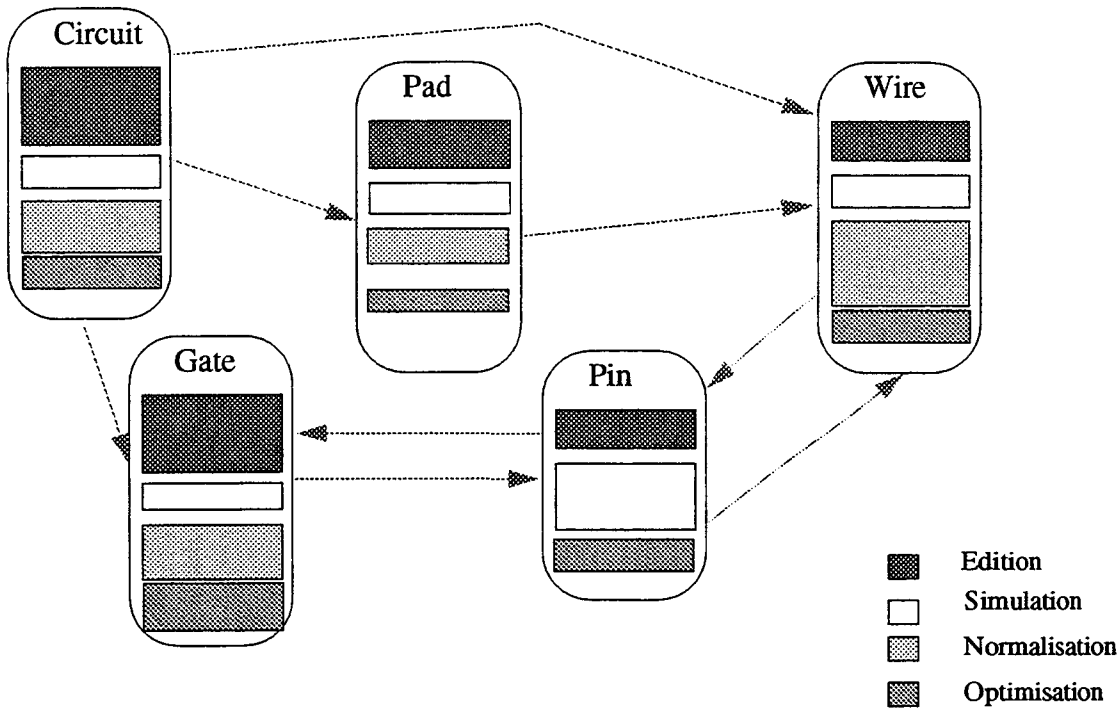


figure 1.4 : Ensemble d'objets multi-fonctionnels

Deux constats peuvent être faits d'après cette figure. Premièrement, la structuration en fonctions du système n'apparait pas en tant que telle dans la structure d'objets mais se retrouve au niveau granulaire de chaque objet. Deuxièmement, chaque fonction est distribuée parmi les objets y prenant part. Sa description n'existe pas en un seul lieu et est exprimée à travers les attributs et méthodes associés à chaque objet participant et les intéractions entre ceux-ci. Un constat similaire est fait dans [Harrison 92][1].

2.3.2 Problèmes posés à la conception par objet

Dans cette section, nous proposons d'étudier le problème de plusieurs fonctions sur un même référentiel d'objets dans l'approche objet classique où les objets sont décrits de façon monolithique par une seule classe[2], leur classe d'instanciation, et où aucune construction descriptive et structurante n'existe en dehors des classes. Compte tenu de ces caractéristiques, l'approche objet classique admet, selon [Harrison 93][3], deux façons d'aborder le problème. Une première approche consiste à concevoir les objets du référentiel en fusionnant leurs descriptions relatives à chaque fonction au sein des classes correspondantes (1). Une seconde approche consiste à externaliser les traitements de la fonction dans de nouveaux objets spécialisés qui intéragissent avec ceux du référentiel (2). Dans les deux prochaines sous-sections, nous

1. "A tool in this context is no longer a single chunck of code. Instead, it is a collection of methods spread accross a number of classes"
2. Précisons que nous nous plaçons ici dans le paradigme classe/instance. Le traitement de la problématique dans les langages à prototypes n'est pas abordé.
3. "With the classical object model, the designer is faced with one of two choices. (2) On one hand, the application can be constructed as a client using the encapsulated methods but forgoing the advantages of encapsulation and polymorphism. (1) On the other hand, the application's function could be integrated into the same class manipulated by other application...The system must manage an ever-expanding collection of state and behavior... Both choices are objectionable"

présentons et critiquons successivement chaque approche.

2.3.2.1 Approche par fusion des descriptions

Dans l'approche objet classique, la description des objets est fournie par leur classe. En règle générale, cette description est faiblement structurée. Elle se divise principalement en deux parties: un ensemble d'attributs et un ensemble de méthodes. Dans la plupart des cas, ces deux ensembles[1] ne sont pas structurés ce qui confère un caractère monolithique à la description des objets. Cette faible structuration de la description des objets impose également une visibilité globale et une unicité de noms au sein de celle-ci.

Cette description monolithique permet de représenter des objets multi-fonctionnels. Cela oblige cependant à fusionner leur description relative à chaque fonction en une seule description non structurée[2]. La figure 1.5 illustre cette approche pour cinq classes de notre exemple. Pour simplifier, seules les méthodes faisant partie de l'interface sont montrées (il existe aussi de nombreuses méthodes privées, cf Annexe) et la fonction de documentation/nomenclature a été omise.

Comme cette figure permet de le constater, cette approche produit des classes volumineuses et peu modulaires. Ces classes concentrent en elles plusieurs descriptions qui sont difficiles à isoler sans une analyse minutieuse du code. Dans l'exemple précédent, les attributs et les méthodes de la classe `Gate` spécifiques à la fonction d'optimisation ne sont pas faciles à identifier sans ambiguité. De fait, le travail sur une description relative à une fonction particulière devient compliqué. Il est de plus rendu délicat dans la mesure où les caractéristiques des autres fonctions qu'il ne faut pas manipuler pour préserver leur intégrité ne sont pas clairement explicitées. Par ailleurs, à cause de l'exigence de nom unique, tout ajout ou modification apporté à la classe pour une fonction ne peut se faire en ignorant les détails existant déjà dans celle-ci pour les autres fonctions.

Pour une classe en relation d'héritage avec d'autres classes, la fusion des descriptions au niveau de chacune d'elles entraine des problèmes supplémentaires. Ces problèmes se posent lorsque dans la classe on veut utiliser ou adapter les caractéristisques héritées des sur-classes relativement aux fonctions où ses objets participent. En l'absence de toute structuration, il faut procéder à un examen minutieux des sur-classes pour identifier ces caractéristiques. Dans l'exemple précédent, les caractéristiques de la classe `Gate` héritées par la classe `And` pour chacune des fonctions ne sont pas clairement explicitées. De façon générale, cela entraine de sérieuses difficultés pour analyser et spécifier l'héritage entre classes relativement à chaque fonction et pose aussi le problème intéressant de localité de l'héritage. Une solution à ce problème d'héritage sera présentée au chapitre 2.

---

1. Précisons que certains langages (comme en C++, Eiffel ou encore JAVA) proposent parfois une structuration plus fine de ces deux ensembles qui sert essentiellement à fixer la visibilité des différentes caractéristiques (publique, privée, ...) par rapport aux autres objets et classes.
2. Bien entendu, il est toujours possible de faire apparaitre cette distinction à l'aide de commentaires informels embarqués dans le code de la classe mais cette solution a comme défaut d'être imprécise et soumise à la discipline du programmeur.

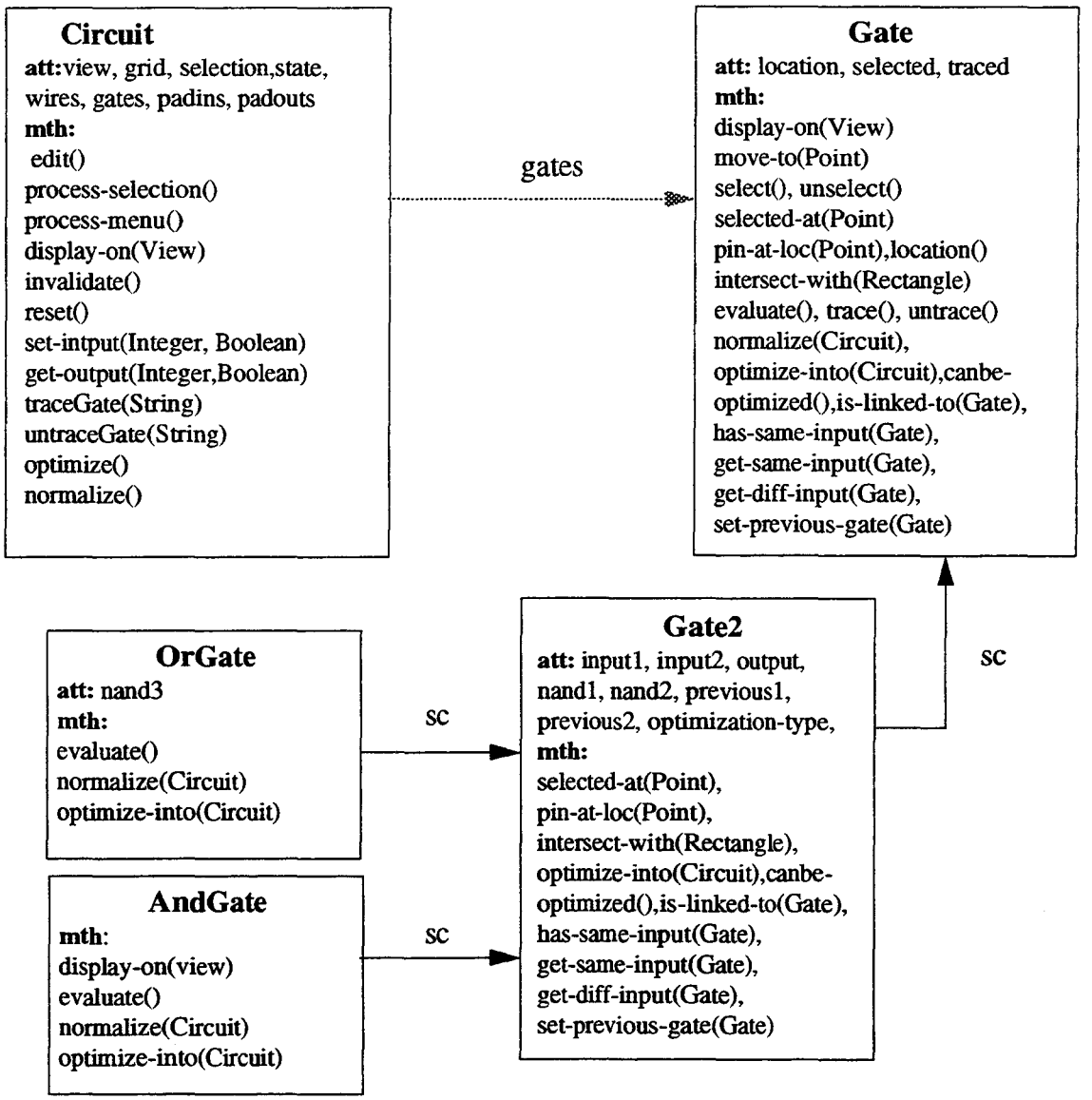


figure 1.5 : Quelques classes de l'exemple dans l'approche objet classique

En plus de ces problèmes relatifs aux objets, d'autres apparaissent lorsqu'il s'agit de prendre en compte les fonctions transversales du système.

Dans l'approche objet classique, l'organisation du système est fournie par les classes. Une telle organisation met en avant la structuration en objets et en classes du système mais elle ne rend pas compte de sa structuration en fonctions. La figure 1.6 montre l'organisation en classes obtenue pour notre exemple.

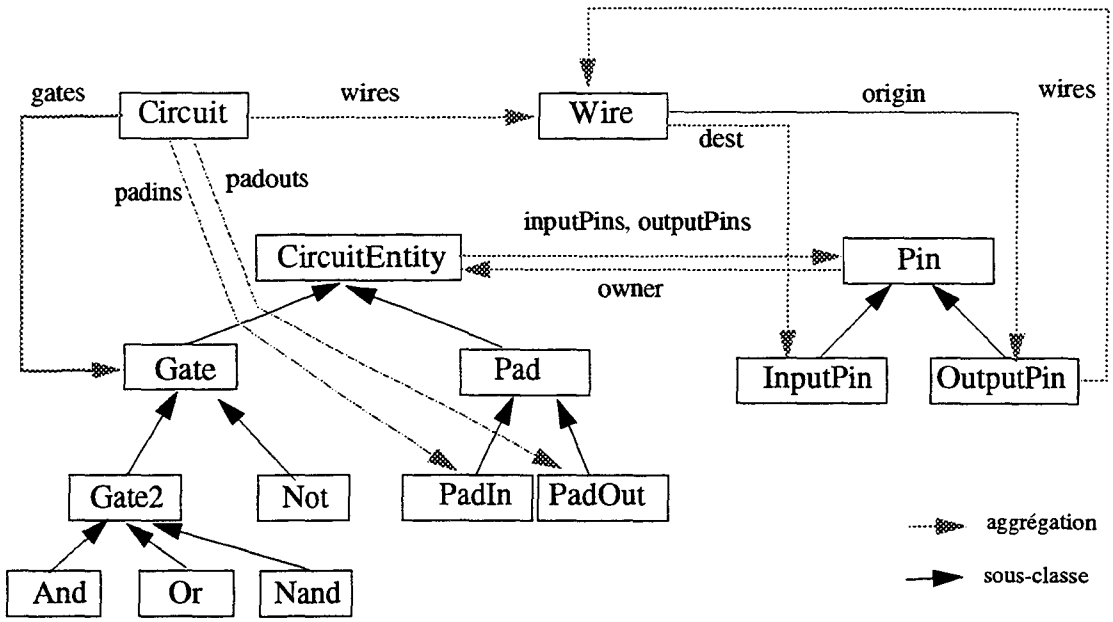


figure 1.6 : Organisation en classes de l'exemple

Selon cette organisation, on peut constater que les différentes fonctions n'apparaissent pas explicitement. Celles-ci se trouvent en fait disséminées à travers les objets. Dans ces conditions, on est amené à appréhender l'ensemble des objets pour déterminer les fonctions qui s'y trouvent. Ce problème est également explicité dans [Larvet 94][1].

On vient de voir qu'il est difficile dans l'approche objet classique de déterminer la structuration en fonctions du système à partir de son organisation en objets et en classes. L'opération[2] consistant à circonscrire les objets et les classes intervenant dans la réalisation d'une fonction n'est guère plus aisée. Il n'y a en effet aucun moyen faisant explicitement le lien entre une fonction et les objets qui la réalisent. Ici, ce lien est enfoui dans le code des objets. De fait, pour retrouver ces objets et ces classes, on n'a pas d'autres solutions que de procéder à un examen approfondi du code, ce qui oblige à affronter la plupart des objets dans leur intégralité et à considérer des aspects non pertinents en raison de la non structuration de leur description. Ce problème est également énoncé dans [Andersen 92][3].

Enfin un dernier problème sous cet angle est lié à la communication inter-objets au sein d'une fonction que la fusion des descriptions au niveau des classes (et par conséquent au niveau des protocoles) rend plus difficile à analyser et à réaliser. En effet, selon cette fusion, les messages que peuvent s'envoyer les objets intervenant dans une même fonction ne sont pas

1. "Une autre critique justifiée est faite à l'approche objet: elle ne permet pas de montrer clairement les fonctions globales du système"
2. Cette opération est importante pour pouvoir comprendre et modifier facilement une fonction. Il faut pouvoir identifier rapidement et précisemment quels sont les objets qui risquent d'être affectés par un changement de spécification de la fonction.
3. "The description of the behavioral effect is distributed over several classes at the same time of the participating objects. To comprehend such an activity the user has to consider several classes at the same time. This is complicated by object of each class participating in several possibly independant activities, and the description of these activities being intermingled in each class"

directement apparents. Ceci est illustré par l'exemple de la figure 1.5 où les méthodes des classes d'objets portes pouvant être activées par envoi de messages dans les méthodes de la classe Circuit définies pour la fonction d'optimisation ne sont pas explicitées. Précisons de plus qu'il n'y a pas de contraintes forçant les communications entre objets au sein d'une même fonction ce qui pose le problème intéressant de localité des envois de message. Une solution à ce problème sera présentée au chapitre 2.

2.3.2.2 Externalisation des traitements et patron de conception visiteur

Comme nous l'avons déjà précisé, l'approche objet autorise une autre solution pour traiter le problème de plusieurs fonctions sur un même référentiel d'objets qui est souvent employé à la place de l'approche précédente à cause de la complexité des objets qu'elle engendre. Cette approche repose sur l'externalisation des traitements associés aux objets du référentiel dans de nouveaux objets spécialement conçus à cet effet et intéragissant avec ces premiers. La figure 1.7 schématise l'organisation en objets obtenue en appliquant cette approche à notre exemple. Pour chaque fonction, on peut identifier un groupe particulier d'objets chargés de réaliser les traitements correspondants. Ainsi, pour la fonction d'édition schématique par exemple, les traitements d'affichage du circuit et du positionnement graphique des composants sont respectivement externalisés dans les objets des classes `CircuitLayoutOrganizer` et `CircuitDisplayer`. Une telle approche est suggérée dans [Halbert 87] et [Gamma 94][1] et peut être rapprochée de l'approche MPVC[2] (Modèle-Point de Vue-Contrôleur) proposée dans [Krief 91]. Elle se retrouve synthétisée dans le patron de conception Visiteur [Gamma 94] que nous présentons dans les prochains paragraphes.

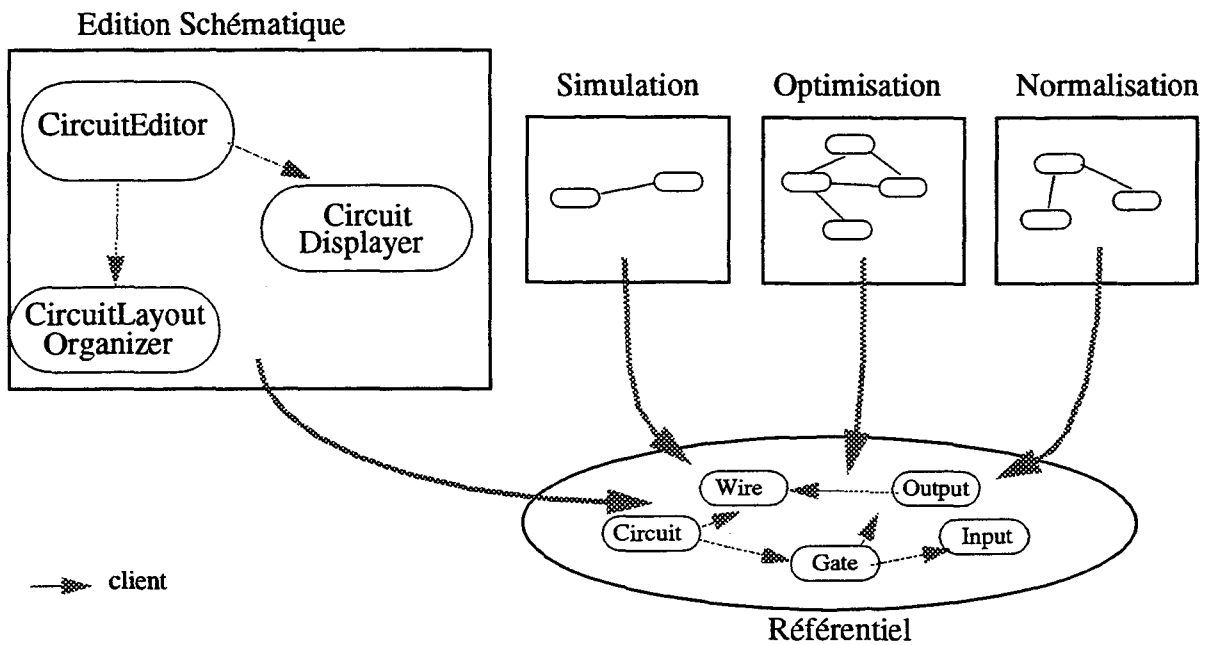


figure 1.7 : Externalisation des traitements liés à chaque fonction

Le patron Visiteur convient quand on veut appliquer plusieurs traitements distincts sur les éléments d'une structure d'objets quelconque, ces traitements se comportant différemment

1. "Avec l'adjonction de chaque analyse, l'interface de l'objet s'alourdit. Au bout d'un certain temps, les opérations d'analyse finiront par nuire à la clarté de l'interface...Cette interface sera perdue dans le bruit...Il ressort de toutes ces indications qu'il faut encapsuler l'analyse dans un objet séparé".
2. qui est toutefois à distinguer de la triade MVC de Smalltalk destinée à la construction d'interface .

suivant chaque type d'élément. Pour ce problème, ce patron donne une solution qui permet la définition de nouveaux traitements sans qu'il soit nécessaire de modifier à chaque fois la classe des éléments visités. Le principe de cette solution consiste d'une part à définir des objets visiteurs de la structure qui implantent chacun un traitement particulier, d'autre part à faire en sorte que ces visiteurs soient activés de façon générique par les objets de la structure qu'ils traversent.

Pour mettre en place cette solution, deux consignes de conception sont données par le patron. La première concerne les classes de visiteurs intervenant sur la structure d'objets. Ces classes doivent être munies du même protocole, lequel est établi en déclarant une méthode pour chaque classe concrète de la structure[1]. L'implantation associée à ces méthodes est cependant spécifique à chaque classe de visiteur et dépend du traitement qu'il réalise. La seconde consigne concerne toutes les classes concrètes de la structure à traverser. Ces classes doivent être dotées d'une méthode particulière généralement nommée `accept` et prenant en paramètre un visiteur répondant au protocole indiqué précédemment. Chacune de ces classes doit alors coder cette méthode de façon à appeler la méthode lui correspondant dans le visiteur. Cette méthode `accept` est destinée à être invoquée par les visiteurs pour chaque objet rencontré lors du parcours de la structure.

Illustrons ces deux consignes à partir de notre exemple en reconsidérant l'affichage d'un circuit considéré en 2.3.1 comme traitement des objets du référentiel eux-mêmes. Ce traitement est ici pris en charge par un objet visiteur qui est nommé `CircuitDisplayer`. La prise en compte de la première consigne conduit à définir dans la classe de cet objet des méthodes pour chaque classe concrète du référentiel: `visitCircuit`, `visitAnd`, `visitNot`, `visitWire`, `visitInputPin`, ..., chacune de ces méthodes implantant l'affichage correspondant. La définition de cette classe en Java par exemple a l'allure suivante:

```
class CircuitDisplayer : CircuitVisitor
{
   public:
      CircuitDisplayer(View v) { }
      void visitCircuit(Circuit c)
      {
            // affichage de la grille
            ...
            // affichage de toutes les portes
            ...eg = c.gates(); while (eg.hasMoreElements()){eg.nextElement().accept(this);}
            // affichage de toutes les équipotentielles
            ...ew = c.wires(); while (ew.hasMoreElements()){e.nextElement().accept(this);}
      }
      void visitAnd(AndGate ag)
      {
               // affichage du symbole graphique représentant une porte ET
               ...v.displayForm(AndForm,pos);
               // affichage des deux broches d'entrée
               ag.input1().accept(this); ag.input2().accept(this);
               // affichage de la broche de sortie
               ag.output().accept(this);
      }

      void visitNot(NotGate ng)
      {
               // affichage du symbole graphique représentant une porte NON
               ......v.displayForm(NotForm,pos);
```

1. nommée généralement à partir du préfixe "visit" suivi du nom de la classe

```
            // affichage de la seul broche d'entrée
            ag.input().accept(this);
            // affichage de la broche de sortie
            ag.output().accept(this);

      }
      void visitWire(Wire w) {
            // affichage de segments de droite entre les deux broches reliés
            ...
      }
      ...
      private:
         View v;
};
```

On peut constater que la méthode associée à chaque classe du circuit prend en paramètre une instance de celle-ci. Grâce à ce paramètre, le visiteur reçoit l'objet visité lors du parcours et peut solliciter ce dernier pour mettre en oeuvre l'opération d'affichage. On peut également noter à plusieurs endroits dans ces méthodes l'invocation de la méthode `accept` sur les objets du référentiel. Ces invocations permettent à l'objet visiteur de parcourir la structure d'objets représentant un circuit.

La prise en compte de la seconde consigne amène à ajouter une implantation de la méthode `accept` pour chaque classe concrète du référentiel. Le code suivant montre l'implantation de cette méthode pour les classes `Circuit`, `AndGate` et `NotGate`.

```
class Circuit {
   public:
      // opération de base
      ...
      // pour le patron
      void accept(CircuitVisitor cv) { cv.visitCircuit(this);}
}
class OrGate: Gate {
   public:
      // opération de base
      ...
      // pour le patron
      void accept(CircuitVisitor cv) { cv.visitOr(this); }
};
class AndGate : Gate {
   public:
      // opération de base
      ...
      // pour le patron
      void accept(CircuitVisitor cv) { cv.visitAnd(this); }
};
```

On remarque à la lecture de ce code que chaque implantation consiste à appeler la méthode correspondante du visiteur obtenu en paramètre ainsi qu'à lui transmettre l'objet courant (designé par `this`). Insistons ici sur le fait que cette implantation suffit pour permettre à tout visiteur de circuit d'opérer sur la structure, y compris ceux développés ultérieurement.

A partir de cette organisation du code, appliquer un traitement sur un circuit revient à créer un objet visiteur de la classe correspondante et à invoquer la méthode `accept` du circuit avec comme argument ce visiteur. Par exemple, pour afficher un circuit, il suffit de lui envoyer le message `accept` avec en paramètre une instance de `CircuitDisplayer`.

Par rapport aux problèmes de structuration et de modularité rencontrés dans l'approche précédente, la solution offerte par le patron visiteur semble de ce point de vue plus satisfaisante car elle permet d'isoler les traitements spécifiques à chaque fonction. Cependant, cette solution reste très "partielle" en présentant les inconvénients suivants.

Un premier inconvénient lié à cette approche est d'ordre méthodologique. Elle conduit à ce qui est appelée dans [Riel 94] la "séparation artificielle des données et des traitements dans des objets séparés". Ici, cette séparation apparait en examinant la nature des objets du référentiel et des objets visiteurs obtenus avec cette approche. D'un coté, on obtient des objets du référentiel qui n'ont aucune composante opérationnelle significative, leur rôle se résumant à l'extrême à celui de simples serveurs de données pour les objets visiteurs. De l'autre coté, les objets visiteurs ont pour seule vocation de réaliser des traitements relatifs aux objets du référentiel ce qui les rend étroitement couplés avec ces derniers[1]. Cette séparation est considérée dans [Meyer 88][Ferber 91] et [Riel 94][2] comme une violation d'un principe cher à l'approche objet, celui du regroupement des données et des traitements associés au sein de la même entité qui garantit son autonomie et sa modularité.

Un deuxième inconvénient concerne l'encapsulation des objets du référentiel que ce patron tend à compromettre [Ossher 98][3]. Le schéma de solution de ce patron repose en effet sur l'assurance que le protocole des objets est suffisamment riche pour permettre aux visiteurs de faire leur travail. Cela impose fréquemment de rendre publiques des opérations pour accéder à l'état interne ou à la structure des objets visités. Par exemple, pour permettre l'affichage de circuits par le visiteur, des moyens d'accès aux différents constituants (porte, équipotentielle) sont forcément à prévoir dans le protocole des objets circuits. Une complication supplémentaire pour établir ces moyens d'accès vient du fait que d'un visiteur à l'autre la manière de parcourir la structure peut varier. En général, une anticipation aux différents projets de parcours consiste à ouvrir complètement la structure d'objet. Cette solution extrême présente cependant comme défaut de ne plus protéger le code clients vis à vis des changements de structure et peut causer des problèmes bien connus de cohérence analysés dans [Blake 87] et [Carré 89].

Un autre inconvénient de ce patron est qu'il convient difficilement pour des traitements qui réclament de nouveaux états aux objets du référentiel (par exemple, une position pour l'affichage). Dans ce cas précis, la seule solution permise sans altérer la définition de ces objets consiste à gérer ces états au niveau du visiteur ce qui peut rapidement devenir complexe lorsqu'ils sont en grand nombre.

Enfin, un dernier inconvénient bien connu de l'approche par visiteur est qu'elle est peu apte à l'évolution de la structure d'objet (et donc du référentiel). Chaque nouvelle classe ajoutée à la structure impose en effet d'introduire une méthode `visitXX` dans chaque classe de visiteur ce qui diminue fortement la modularité et limite l'extensibilité.

---

1. En particulier, ces objets ne peuvent pas fonctionner avec d'autres objets et n'ont pas de raison d'exister indépendamment des objets du référentiel.
2. "This separation should be considered a violation of a fundamental principle of the object oriented, namely data and behavior are bidirectionaly related as a conceptual block"
3. De façon plus générale, ces problèmes d'encapsulation sont inhérents à l'approche par externalisation. Nous les avions également identifiés et étudiés dans le cadre d'interfaçage graphique de systèmes d'objets [Carré 96] où la dimension interfaçage est quasiment toujours déléguée à des objets spécialisés plutôt qu'aux objets du domaine.

2.3.3 Techniques de description modulaire d'objets

Nous avons montré le besoin de décrire des objets détenant une description pour chaque fonction où ils participent. Nous avons critiqué l'approche objet classique en montrant qu'elle ne permet pas d'expliciter la multiplicité des fonctionnalités. Dans cette section, nous passons en revue quelques techniques apportant des éléments de solution dans un cadre programmatique[1]. Parmi les techniques présentées, nous distinguons deux approches principales. Dans la première approche, les objets sont représentés par une seule description (leur classe) dont la structuration est augmentée pour tenir compte des multiples fonctionnalités. Dans la seconde approche, la représentation des objets est obtenue en faisant intervenir plusieurs descriptions qui réflétent les multiples fonctionnalités.

2.3.3.1 Description unique structurée

Les techniques proposant de structurer une classe selon les multiples fonctionnalités de ses objets sont peu nombreuses. Nous en avons recensé principalement trois.

Classes à interfaces multiples

Une première étude visant à s'affranchir de la description monolithique des classes est celle présentée dans [Shilling 89] qui se situe dans le cadre d'environnement de développement logiciel. Deux extensions du modèle à classes sont ainsi proposées.

La première extension est la possibilité d'avoir plusieurs interfaces distinctes pour une même classe où chaque interface définit, de façon indépendante des autres, un ensemble de méthodes liées à une implantation[2]. Dans le cas présent, une interface sert à déterminer une fonctionnalité des objets spécifiques à une activité de développement (compilation, édition, analyse...). Une méthode étant associée à une interface, des méthodes de même nom peuvent exister dans des interfaces distinctes. Pour ces méthodes, des implantations différentes peuvent être fournies. L'existence de méthodes homonymes au niveau de la classe conduit naturellement à des problèmes d'accès au niveau des objets. Ces problèmes sont ici résolus par l'extension de l'envoi de message pour y inclure l'interface à partir de laquelle on veut obtenir la méthode.

La seconde extension est destinée à fournir un contrôle plus fin sur la visibilité des variables d'instance qui s'appuie sur les interfaces. Pour cela, chaque variable d'instance est étiquetée par le nom des interfaces qui y ont accès. A cet étiquetage vient s'ajouter un mode d'accès qui permet de spécifier pour chaque interface si la variable d'instance est visible en lecture ("read-only") ou lecture/écriture ("modifiable"). La combinaison de ces deux moyens de spécification permet de répondre à plusieurs besoins. Cela peut servir par exemple à rendre une variable

1. Signalons que certaines de ces techniques ne sont pas propres à la programmation par objet et ont notamment été appliquées en représentation des connaissances où il s'agit de représenter les objets d'un domaine considéré selon les points de vue de plusieurs experts [Bobrow 77]][Dekker 94][Marino 93]. Quelques variantes de ces techniques ont aussi été employées dans le cadre des bases de données à objets afin de représenter les rôles qui caractérisent l'évolution des objets au cours de leur vie (par exemple un objet représentant une personne peut acquérir le rôle d'étudiant puis celui de salarié) [Albano 93][Wierenga 94].
2. Cette dissociation entre les méthodes et leur implantation permet notamment de factoriser une implantation entre plusieurs méthodes.

d'instance privée à une interface ou à préciser les modalités de partage d'une variable d'instances entre plusieurs interfaces (modification autorisée pour certaines, lecture seulement pour d'autres). Par ailleurs, la définition de variables d'instances de même nom au sein de la classe est également possible à condition qu'elles ne soient pas visibles dans une même interface.

La figure 1.8 montre un exemple de classe étendue selon les deux caractéristiques précédentes. La notation graphique est celle employée dans l'article.

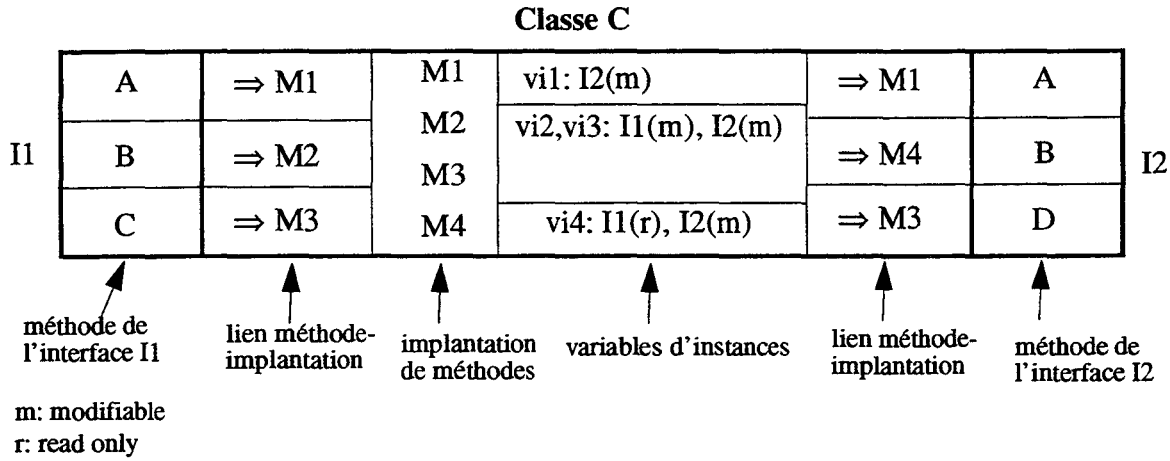


figure 1.8 : Classe à interfaces multiples

Sur cette figure, on peut constater que la classe possède deux interfaces `I1` et `I2`. Chacune de ces interfaces inclut en particulier une méthode de même nom `B` liées à des implantations distinctes `M2` dans `I1` et `M4` dans `I2`. On peut également observer l'étiquetage des variables d'instances par ces deux interfaces. Ainsi, l'ensemble des variables d'instance visibles depuis l'interface `I1` correspond à {`vi2`, `vi3`, `vi4`, `vi5`}et l'ensemble des variables d'instance visibles depuis l'interface `I2` correspond à {`vi1`, `vi2`, `vi3`, `vi4`}. Notons que la variable `vi4` est seulement accessible en lecture pour cette interface.

Cette extension a donné lieu à un prototype en C++ dans lequel une classe étendue est transposée en une famille de classes organisées par rapport aux visibilités des variables d'instance. Selon cette transformation, une instance d'une telle classe est représentée sous la forme d'un objet maitre et de plusieurs sous-objets invisibles pour l'utilisateur.

Composants de classe

[Stata 95] propose d'accroître le modularité au sein d'une classe afin de rendre le sous-classement plus modulaire. La solution proposée repose sur l'introduction d'une nouvelle unité structurelle seulement disponible à l'intérieur de la classe. Cette unité est appelée *composant de classe* dans [Stata 97] et sert à distinguer les composants logiques d'une classe qui peuvent être de nature diverse (fonctionnalités, états abstraits[1], ...). Un composant de classe réunit dans une même unité, la définition d'une partie des variables d'instance de la classe avec celles des méthodes qui les manipulent. Plusieurs composants disjoints peuvent être définis dans une

1. Notons ici que d'autres travaux s'intéressent à un autre type de structuration de la classe liée aux états possibles des objets, appelé aussi "programmation par modes" [Taivalsaari 93].

même classe, conduisant ainsi au partitionnement de l'ensemble des variables d'instance et des méthodes de celle-ci[1]. Il convient de préciser que la structuration en composants d'une classe n'a pas de conséquence pour les clients extérieurs mais intervient par contre pour la programmation des sous-classes.

A titre illustratif, le code d'une classe `C` constituée de deux composants `F1` et `F2` est donné ci-après dans une syntaxe proche de celle de Java. On peut observer que chaque composant est clairement identifié dans le code de cette classe, localisant les variables d'instances et les méthodes associées.

```
class C {
   component F1 {
      private String s1;
      private int i1;

      private int pm() { ... }
      ...
      public void m1() { ... }
      public void m2() { ... }
      public String m3() { ... }
      ...
   }

   component F2 {
      private Array a1;
      private Array a2;
      private String s2;

      public void ma() { ... }
      public void mb() { ... }
      ....
      }
   ...autres composants
}
```

Les composants ne sont pas seulement un moyen de regrouper les méthodes et les variables logiquement liées de la classe. Ils déterminent également la visibilité des variables d'instance au sein de celle-ci. Les méthodes d'un composant sont ainsi les seules à pouvoir accéder directement aux variables d'instances associées. Dans l'exemple précédent, l'accès aux variables d'instances `a1, a2 et s2` du composant `F2` est seulement autorisé aux méthodes `ma` et `mb` du même composant. Lorsqu'une méthode d'un composant a besoin de lire ou de modifier la valeur d'un autre composant, le seul moyen consiste à appeler les méthodes (publiques mais aussi privées au sein de l'objet) fournies par ce dernier.

Les avantages obtenus par cette restriction de visibilité au sein de la classe sont du même ordre que ceux liés à l'encapsulation entre les objets: préservation de la cohérence interne d'un composant, indépendance d'un composant vis à vis des choix d'implantation d'un autre facilitant son remplacement par héritage.

Catégories de méthodes

Dans certains environnements Smalltalk, la technique des catégories de méthodes apporte un

1. Il n'est jamais fait mention de la possibilité de définir des caractéristiques en dehors des composants, ni du partage de caractéristiques entre composants.

début de structuration au sein des classes. Cette technique se situe uniquement au niveau de l'environnement et n'a aucune conséquence opérationnelle sur le langage. Elle permet de partitionner les méthodes d'une classe suivant leurs relations logiques. Une classe possède en général plusieurs de ces catégories qui sont traitées indépendamment de celles des autres classes.

Pour prendre un exemple de l'environnement, l'examen de la classe Date révèle les catégories de méthodes `accessing`, `arithmetic`, `inquiries` et `printing` correspondant à ses différentes fonctionnalités. La structuration de cette classe est représentée à la figure 1.9.

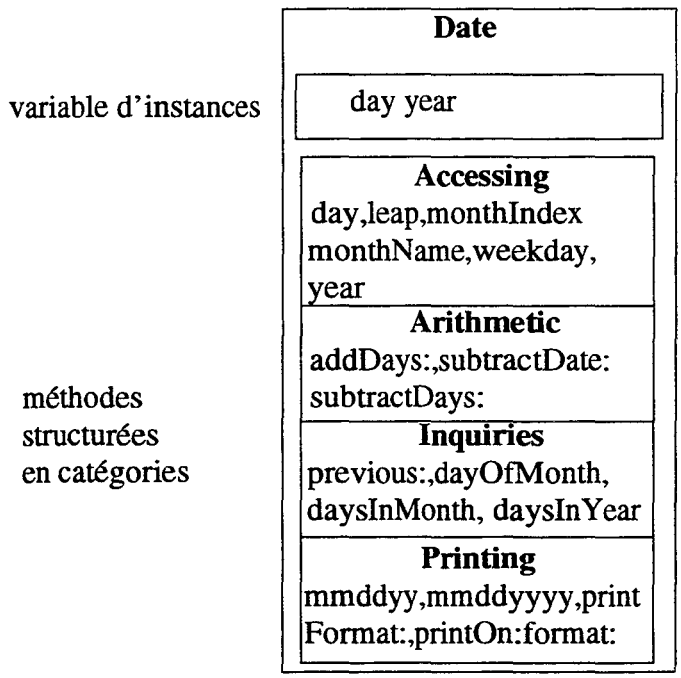


figure 1.9 : Structuration interne d'une classe par les catégories de méthodes

Il est à noter qu'il n'y a pas de catégories équivalentes pour les variables d'instances.

Nous reviendrons sur cette technique au chapitre 5 pour l'enrichir et la systématiser transversalement à plusieurs classes.

2.3.3.2 Description multiple

Comparée à l'approche précédente, l'approche par description multiple a donné lieu à un nombre plus important de techniques. Nous commençons par présenter deux techniques utilisant le pouvoir structurant des mécanismes généraux existants dans les approches à objets que sont l'héritage (multiple) entre classes et l'aggrégation d'objets. Ces techniques reposent sur une interprétation particulière de ces mécanismes. Nous présentons ensuite des techniques dédiées intégrant dès l'origine des mécanismes spécifiques à base de point de vue pour gérer la multiplicité des descriptions d'objets. Nous développerons plus spécifiquement la solution adoptée par ROME qui est notre point de départ.

Héritage multiple

L'héritage multiple peut être mis à profit pour décrire des objets multi-points de vue [Carré 89]. Le principe consiste à concevoir la classe de ces objets comme le résultat de l'héritage de plusieurs classes correspondant aux différentes fonctionnalités.

La figure 1.10 montre l'application de ce principe pour des objets intégrant trois points de vue distincts. Pour chaque point de vue, une classe particulière est introduite dans laquelle sont définis les attributs et méthodes propres à celle-ci. On obtient ainsi trois classes indépendantes[1] nommées `Pdv1`, `Pdv2` et `Pdv3`. De telles classes existent uniquement pour des besoins de structuration et sont uniquement destinées à être héritées. La classe `C` hérite pour ses objets de l'union des caractéristiques définies pour chacun des points de vue. Dans certains cas, cette classe peut également servir à exprimer des relations entre les points de vue par redéfinition et combinaison des méthodes héritées.

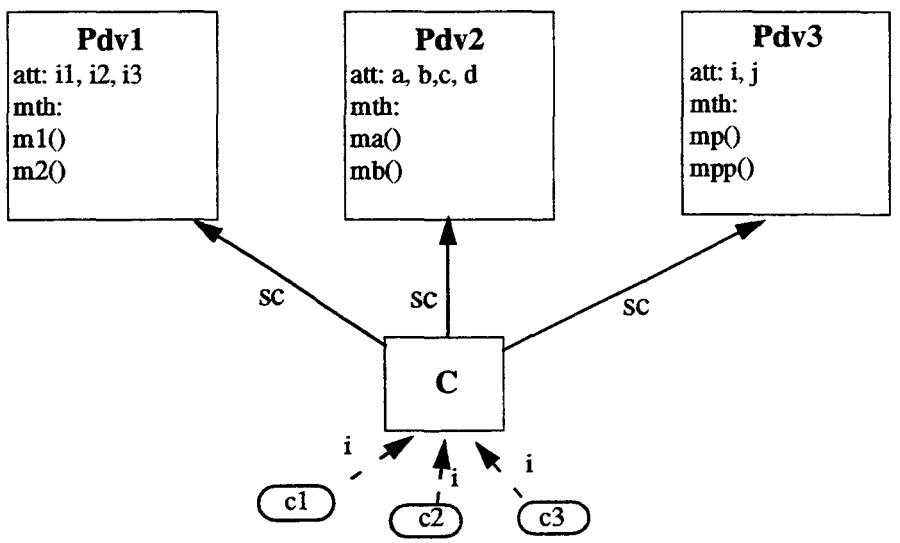


figure 1.10 : Points de vue par héritage multiple

Le graphe d'héritage obtenu pour la classe `C` rend compte de la structuration par points de vue des objets considérés. Ces points de vue sont identifiés en tant que tels dans le graphe par les sur-classes. On peut ainsi facilement localiser les caractéristiques pertinentes à un point de vue.

Cette organisation des points de vue entraine également une modularité importante pour la modification d'une fonctionnalité. Une telle modification n'entraîne ici qu'un minimum de modifications des autres points de vue. Ce fait est bien mis en évidence par le graphe d'héritage précédent où la modification d'un point de vue n'a d'incidence que sur la sous-classe `C`. Par exemple, compléter la classe `Pdv1` avec de nouvelles méthodes fait automatiquement bénéficier la classe `C` sans répercussion sur les autres points de vue décrits.

Un principe important pour modéliser des points de vue par héritage multiple est le principe d'indépendance énoncé dans [Carré 89]. Ce principe stipule que toutes les caractéristiques des

1. Deux classes sont indépendantes si elles ne sont pas sous-classes l'une de l'autre, autrement dit, il n'existe pas de chemin d'héritage de l'une à l'autre.

sur-classes indépendantes doivent être héritées au niveau de la sous-classe même si ces caractéristiques ont le même nom. Ce principe assure l'intégrité de chaque point de vue.

Toutes les stratégies d'héritage multiple ne respectent pas le principe d'indépendance. Il a été montré dans [Carré 90a] que les stratégies dites "linéaires"[1], de CLOS par exemple, ne respecte pas ce principe et se prêtent donc assez mal à la modélisation de plusieurs points de vue.

Le principe d'indépendance est en revanche respecté par les stratégies dites graphiques [Snyder 87] utilisées dans les langages typés statiquement comme C++ ou Eiffel [Meyer 92] et ceux typés dynamiquement comme ROME [Carré 89] ou encore Extended Smalltalk [Borning 82]. Pour gérer les cas d'homonymie (attributs et méthodes), ces stratégies offrent généralement des techniques pour faire un choix. Il existe principalement deux techniques: (1) le renommage des caractéristiques héritées avec le même nom adopté notamment par Eiffel ; (2) la désignation explicite d'une classe levant l'ambiguité lors de l'accès à l'une des caractéristiques (envoi de message étendu d'Extended Smalltalk, sélection de point de vue de ROME, utilisation du typage statique dans C++ [Amiel 95] ou dans Chimera [Bertino 95]). Précisons que ces techniques sont souvent présentées comme un moyen de résoudre les conflits et non comme moyen de conception par points de vue.

Aggrégation

L'aggrégation qui permet de construire un objet complexe par assemblage d'objets plus simples est un autre moyen de structurer la description d'objets multi-points de vue. Le principe consiste à représenter de tels objets comme un objet aggrégat dont chaque sous-objet réifie un des points de vue en encapsulant l'état et en réalisant le comportement associé. On obtient ainsi une représentation éclatée des objets qui rend compte de leur structuration en points de vue et assure une nette séparation entre ces derniers. Cette technique par aggrégation a été adoptée dans l'environnement de développement PIE (Smalltalk-76) [Golstein 80] puis reprise dans Systalk [Wolinski 90] où elle est généralisée sur des objets composites. Plus récemment, elle a aussi été appliquée dans [McAffer 95] pour traduire une implantation structurée d'objets complexes[2].

Dans PIE, le principe précédent est systématisé en introduisant deux classes abstraites: `Node` et `Perspective`. La classe Node définit un modèle d'objets pouvant disposer de plusieurs points de vue appelé ici perspective. Elle spécifie à cet effet la variable d'instance `perspectives` qui contient la liste des perspectives de l'objet, représentants de la classe `Perspective`. Pour permettre un dialogue avec l'objet selon une perspective, elle définit aussi une méthode `as:` qui est paramétrée par un nom de perspective[3] et renvoie le sous-objet correspondant[4] . La classe Perspective définit l'attribut `node` implantant un lien inverse d'une perspective vers l'objet concerné. Ce lien inverse a généralement deux utilités. Il permet depuis n'importe quelle perspective d'accéder indirectement aux autres perspectives de l'objet. Il sert

1. Ces stratégies consistent à transformer l'ordre partiel induit par le graphe d'héritage multiple en un ordre total, se ramenant ainsi à l'héritage simple. Par exemple, une linéarisation possible du graphe précédent est: C -> PDV1 -> PDV2 -> PDV3.
2. Les objets en question sont en fait des méta-objets qui s'occupent de tous les aspects concernant le fonctionnement des objets de base.
3. en générale la classe d'instanciation de la perspective
4. En PIE, la classe Node offre également au travers des méthodes addPersp et deletePersp des facilités pour ajouter et retirer dynamiquement des perspectives

également dans le cas ou il faut transmettre une référence à l'objet dans un envoi de message à un autre objet.

Le schéma de la figure 1.11 donne un aperçu au niveau des classes et des instances de l'organisation `Node/Perspective` et son application pour des objets constitués de trois perspectives.

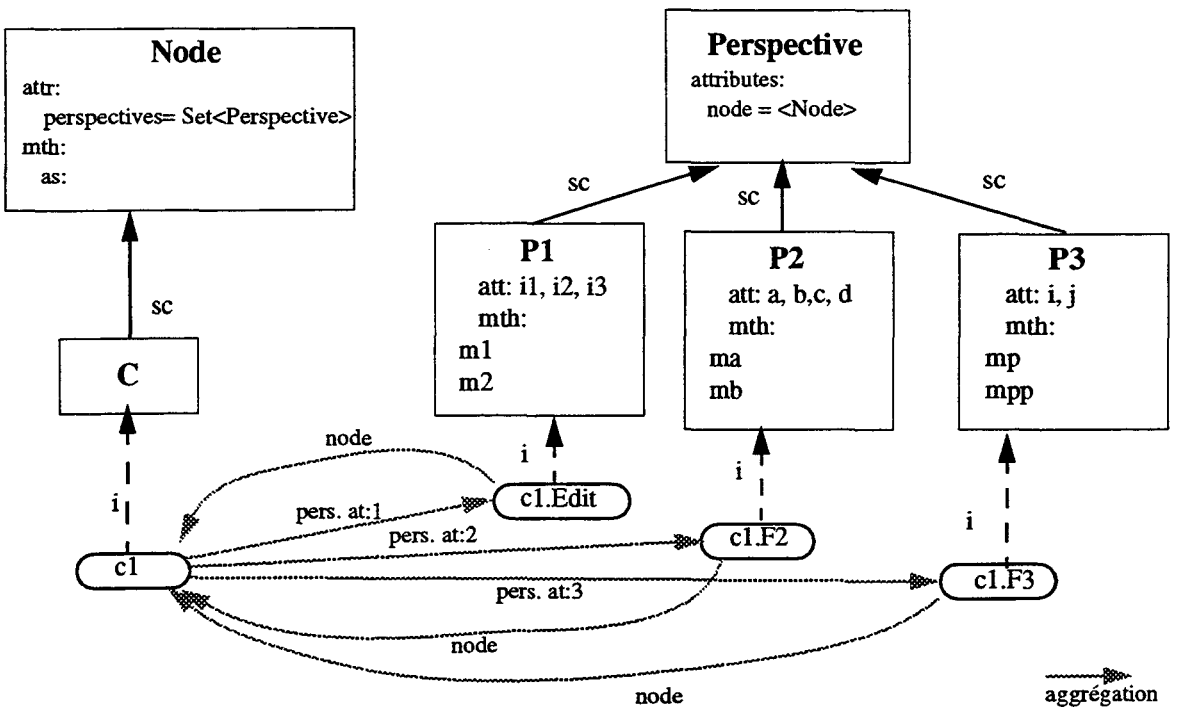


figure 1.11 : Points de vue par aggrégation

La classe `C` de ces objets est ici déclarée sous-classe de `Node`. Cette classe n'introduit aucune caractéristique spécifique relative aux perspectives. Celles-ci sont déterminées par les classes `P1`, `P2` et `P3`, sous-classes de `Perspective`, ce qui permet de les considérer séparemment.

Au niveau des instances, l'objet `c1`, instance de la classe `C`, est un objet aggrégat qui matérialise un objet multi-perspectives pour le reste du programme. A travers son attribut `perspectives` obtenu par héritage, cet objet maintient des références vers une instance de chaque classe de perspective. Ces instances réifient chacune une perspective différente du même objet et ont donc la même référence à `c1` dans leur attribut `node` obtenue par héritage.

A partir de cette représentation, l'accès aux méthodes de l'objet relativement à une perspective se réalise alors en deux temps:

- envoi du message `as:` sur l'objet aggrégat pour récupérer l'objet perspective;
- application de l'opération souhaitée par envoi du message correspondant à cet objet.

A titre d'exemple, pour activer la méthode `m1` de l'objet défini pour la perspective `P1`, on utilisera l'expression suivante: `(c1 as:P1) m1` où le message `as:` permet de récupérer la perspective désirée, à laquelle le message `m1` est ensuite envoyé.

Remarquons que cette solution conduit à exporter des références vers les perspectives à l'extérieur de l'objet aggrégat ce qui pose des problèmes d'identité ("self problem [Lieberman 86], existence de ces objets liée à celle de l'objet aggrégat) et d'intégrité de la représentation (appartenance de ces objets à un seul objet aggrégat). Pour remédier à ces problèmes, le mécanisme de sélecteur composé proposé dans [Blake 87], qui permet d'envoyer un message aux sous-objets sans avoir à les exporter, peut être un début de solution. De même, le mécanisme de délégation tel qu'il est proposé dans les langages à prototypes [Dony 92] peut également être une solution à ce problème en remplaçant les liens d'aggrégation par des liens de délégation.

Amélioration de la technique par aggrégation à l'aide de classe locales

Une amélioration de l'approche précédente peut être obtenue en utilisant la notion de classe locale. Une classe locale est une classe qui a la particularité d'être définie à l'intérieur d'une autre classe au même titre que les variables d'instances et les méthodes de cette dernière. Hormis cette particularité qui lui confère certaines propriétés, une classe locale est une classe comme une autre; elle peut être instanciée et spécialisée. Une classe peut contenir plusieurs classes locales. A notre connaissance, seuls les langages BETA [Madsen 93] et Java sont dotés de la capacité de définir une classe localement à une autre[1].

Le recours aux classes locales pour la représentation de points de vue basée sur l'aggrégation a été étudié et mis en avant dans [Madsen 92][2]. Par rapport à la version de base décrite précédemment, la solution basée sur les classes locales permet:

- de mieux organiser la description des points de vue;
- de mieux gérer le statut des objets représentant les points de vue;
- d'ajouter des possibilités de partage entre les points de vue;
- de faciliter l'expression des intéractions entre points de vue.

Dans le contexte de l'aggrégation, l'utilisation des classes locales procure deux fonctionnalités qui n'ont pas d'équivalent dans un modèle à classes conventionnel où toute classe est globale. Cela permet:

- d'exprimer que des objets ont seulement un sens en tant que composants d'autres objets. Cela se traduit par l'impossibilité de créer une instance de la classe locale de façon isolée. Une telle instance doit toujours exister en relation avec une instance de la classe englobante.

- de définir la classe locale en fonction de la classe englobante. Une instance d'une classe locale a directement accès aux caractéristiques de l'objet (instance) dont elle est un composant. Cet accès est obtenu sans avoir besoin de désigner l'objet englobant. Ce dernier se comporte comme un environnement permanent pour ses composants.

L'emploi des classes locales pour représenter les points de vue ne modifie pas les principes

1. C++ possède également la notion de classe locale mais celle-ci reste peu développée par rapport à ces deux langages. Elle s'adresse essentiellement à des problèmes de visibilité des classes. En outre, les instances d'une classe locale sont traitées comme si celle-ci avait été définie globalement

2. "Part objects may also be used to model that the containing object are characterized by various aspect, where these aspects are defined by other classes...locally defined object enhance this possibility considerably"

de base de la technique par aggrégation. Ce qui diffère par contre, c'est la manière dont les classes de points de vue sont définies. Au lieu d'être définies globalement, celles-ci sont regroupées au sein de la même classe, celle des objets aggrégats. Le code suivant en Java montre la définition de la classe des objets aggrégats pour l'exemple considéré précédemment. Celle-ci inclut la définition de plusieurs classes locales correspondant aux points de vue. Cette classe prévoit également les attributs pour référencer les sous-objets, instances des classes locales.

```
public class C {
   public P1 asP1 = new P1();
   public P2 asP2 = new P2();

   private String shared;

   public class P1 {
      private int i1;
      private int i2;
      ...
      public void m1() { ...i1 = shared ...}
      public void m2() { ... }
      public void m3(int i) { ... }
               ...
   }
   public class P2{
      private int a, b, c
      private int i2;

      public void ma() { ...shared = ...}
      public String mb() { ...asP1.m2().. }
      ....
   }
   ...
}
```

figure 1.12 : Points de vue par aggrégation et classes locales

Le fait de localiser ainsi les classes de points de vue dans la classe procure une organisation plus approprié de celle-ci. En effet, la description d'un point de vue est ainsi directement associée à la classe des objets qu'elle concerne et caractérise. Il en résulte une meilleure modularité. Par ailleurs, cela permet également de prendre en compte le fait qu'un point de vue ne peut exister sans l'objet qu'il représente.

Dans un paragraphe précédent, nous avons mentionné une caractéristique propre à la notion de classe locale qui est la possibilité de faire référence à la classe englobante. Cette caractéristique permet de prendre compte des besoins d'intéraction et de partage de variables d'instance entre points de vue. Concernant l'intéraction entre points de vue, cela est immédiat du fait que tout point de vue dénoté par une variable d'instance de l'objet englobant peut directement être exploité. Dans l'exemple précédent, cette facilité est mise à profit dans la définition du point de vue `P2` pour invoquer la méthode `m2` du point de vue P1 obtenu à travers l'attribut du même nom. Concernant le partage, celui-ci se réalise facilement en plaçant les caractéristiques à partager au niveau de la classe englobante. C'est ce qui est fait dans l'exemple ci-dessus pour la variable d'instance `shared` de la classe `C` qui devient ainsi accessible à l'ensemble des points de vue. Rappelons que dans l'approche par aggrégation sans classe locale, l'intéraction entre points de vue nécessitait de gérer des liens supplémentaires et le partage de variable d'instance était impossible. Ici, le seul fait de localiser les classes des points de vue

dans la classe d'aggrégat suffit.

Représentation multiple

La représentation multiple est une notion propre au langage ROME [Carré 89][Carré 90]. Cette notion permet une description modulaire des objets qui est plus souple que les approches décrites précédemment.

ROME retient la classe comme seul moyen de description des objets. Pour caractériser les objets de manière multiple selon différents points de vue, l'approche est fondée sur le sousclassement. Une première classe déterminant l'identité des objets est d'abord définie. Ensuite, pour décrire un point de vue sur ces objets, on procède en sous-classant cette première classe. Cela mène à introduire une sous-classe pour chaque point de vue à représenter. A la différence de la classe initiale, ces sous-classes n'introduisent pas de nouveaux objets mais représentent un point de vue sur ces derniers. Selon les besoins, une telle sous-classe peut à son tour être sous-classée pour décrire plus finement le point de vue sur les objets (pour des sous-points de vue par exemple). Cela donne lieu à une hiérarchie structurée en plusieurs sous-hiérarchies, chacune représentant la description des objets selon un point de vue particulier. Chaque sous-hiérarchie décrit les objets indépendamment des autres points de vue.

Considérons à titre d'exemple la description des produits logiciels selon le point de vue de leur application, le point de vue de leur maintenance et leur point de vue commercial [Carré 90b]. Cet exemple est illustré par la figure 1.13.

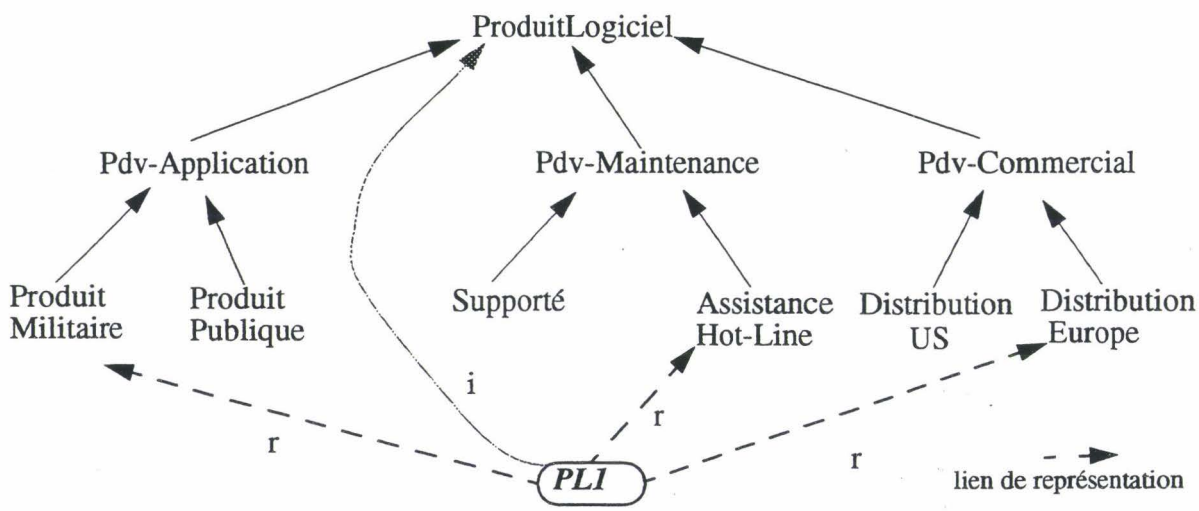


figure 1.13 : Représentation multiple en ROME

La classe `ProduitLogiciel` introduit l'identité des objets. Chaque sous-classe spécialise celle-ci selon un des points de vue donnés précédemment. On peut observer que les classes issues d'un même point de vue sont regroupées dans une même sous-hiérarche. Les différents points de vues sont ainsi séparés les uns des autres, ce qui permet leur description de façon modulaire.

Pour représenter un objet selon les différents points de vue, ROME offre la représentation multiple d'objets par laquelle un objet peut être représentant de plusieurs sous-classes de sa classe d'instanciation. Les différentes sous-classes peuvent appartenir à des sous-graphes distincts et indépendants correspondant aux différents points de vue. La représentation de

l'objet selon un point de vue est alors établie par un lien vers une classe du sous-graphe correspondant.

Dans l'exemple précédent, un produit logiciel particulier, appelé `PL1`, est avant tout instance de la classe `ProduitLogiciel`. Cet objet est aussi représentant de plusieurs sous-classes ce qui traduit sa description multiple selon les points de vue maintenance, application, et commercial. Les liens "r" indiquent les liens de représentation directe, s'ajoutant au lien, unique et invariant, d'instanciation (noté "i"), qui détermine l'identité. L'objet `PL1` possède ainsi toutes les caractéristiques des classes de représentation. Notons que la manipulation des liens de représentation d'un objet est dynamique, ce qui offre des capacités d'évolution de la description de cet objet.

En ROME, la représentation multiple et évolutive d'un objet est contrôlée par le principe de représentation minimale qui limite les possibilités de rattachement et de détachement aux sous-classes strictes de sa classe d'instanciation. Ce principe garantit qu'un objet reste toujours au minimum représentant de sa classe d'instanciation et par conséquent des sur-classes de celles-ci. Ceci assure la cohérence d'objet et empêche par exemple une instance de `ProduitLogiciel` de devenir représentant d'une sous-classe de `Personne` ou de `Table`. Pour plus de détails voir [Carré 89].

La structuration de la hiérarchie selon plusieurs points de vue et la représentation multiple correspondante des objets décrits favorisent la considération de ces derniers sous un point de vue particulier, indépendamment des autres points de vue. En ROME, chaque classe détermine potentiellement un point de vue sur ses représentants selon la définition suivante:

- soit l'ensemble de représentation d'un objet O, Rep(O), l'ensemble de ses classes de représentation directe et leurs super-classes;
- soit l'ensemble des classes dépendantes d'une classe C, Dep(C), l'union de ses super-classes, de ses sousclasses et d'elle-même;

alors le point de vue d'une classe C sur un objet O se définit par

$$Pdv(O, C) = Rep(O) \cap Dep(C)$$

L'ensemble de classes défini comme le point de vue de C sur O constitue une partie de la hiérarchie des classes. Pour illustrer cette notion, reconsidérons l'exemple des produits logiciels. Le point de vue de la classe `Pdv-Maintenance` sur l'instance `PL1` se compose de l'ensemble des classes situées dans la zone grisée de la figure suivante.

```
Pdv(PL1, Pdv-Maintenance )
     = Rep(PL1) ∩ Dep(Pdv-Maintenance)
     = {ProduitMilitaire, Pdv-Application, AssistanceHot-Line, PdvMainte-
       nance, DistributionEurope, Pdv-Commercial, ProduitLogiciel} ∩ {As-
       sistanceHot-Line, Supporté, PdvMaintenance, Produit-Logiciel}
     = {AssistanceHot-Line, PdvMaintenance, Produit-Logiciel}
```

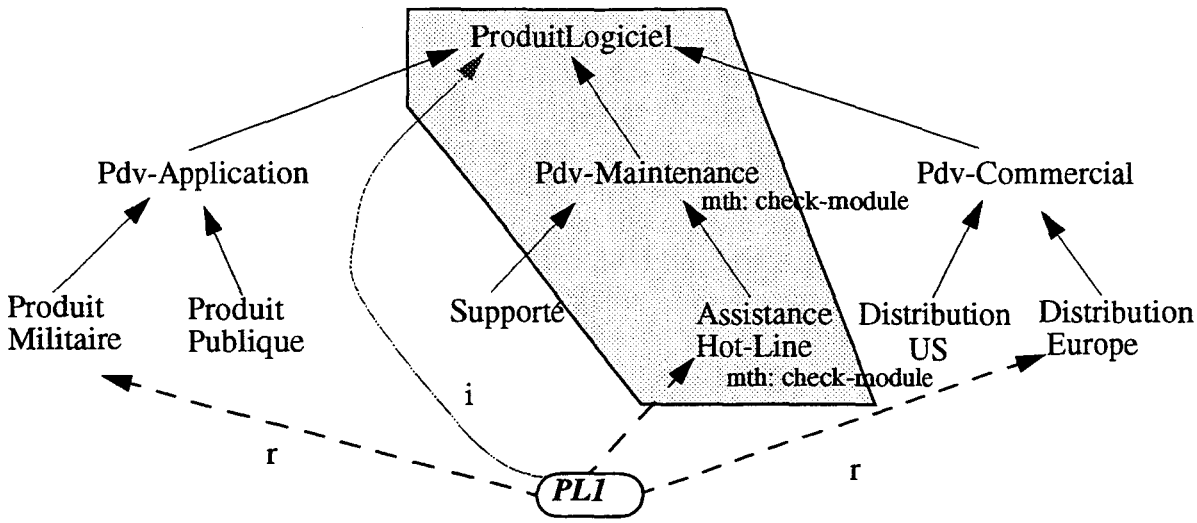


figure 1.14 : Sélection d'un point de vue

La sélection d'un point de vue sur un objet permet de privilégier une partie des classes de la hiérarchie lors de l'accès à l'objet. Le point de vue sélectionné est pris en compte en particulier pour la désignation non-ambigüe des caractéristiques d'un objet. Quand il y ambiguité entre plusieurs caractéristiques, la sélection a pour effet de choisir celle qui est définie dans une classe faisant partie du point de vue choisi, tout en respectant l'affinement. Précisons que cet affinement ne peut jamais être contourné par le choix d'un point de vue. Concrètement, cela signifie que si une caractéristique possède une définition plus affinée en dehors du point de vue sélectionné, alors c'est cette définition qui est retenue. Une sélection de point de vue ne s'applique qu'en cas d'ambiguité.

En ROME, la sélection de point de vue s'exprime par une as-expression qui peut être associée à un envoi de message, un appel de méthode ou un accès d'attributs. Par exemple, pour envoyer un message `check-module`[1] à l'objet `PL1` considéré sous le point de vue maintenance, on écrit:

```
(PL1 'as Pdv-Maintenance 'check-module 2 #t)
```

Ce code a pour résultat l'application de la méthode `check-module` la plus affinée sous le point de vue maintenance, c'est à dire celle de AssistanceHot-Line. Notons que la classe spécifiée dans une as-expression n'est pas nécessairement celle où la caractéristique est définie. Ceci provient du fait que la recherche est accomplie sur tout le graphe correspondant au point de vue.

En plus des points de vue explicites donnés par le programmeur, des points de vue implicites sont automatiquement établis par ROME. Ils sont inférés dans deux cas de figure. Premièrement, en cas d'absence de point de vue explicite sur l'objet, un point de vue équivalent à son graphe de représentation sert de point de vue par défaut. Deuxièmement, à chaque application de méthode, un point de vue implicite est déterminé à partir de la classe où est trouvée la définition de la méthode. Cette notion de point de vue implicite est importante car elle permet de préserver le principe d'indépendance dans l'enchainement des méthodes appliquées par l'objet. Elle assure notamment que l'activation de l'objet se déroulera

1. qui lancerait une procédure de test du logiciel par exemple.

exclusivement sous un point de vue indépendamment des autres. Dans l'exemple précédent, le point de vue implicite inféré suite à l'appel de la méthode est celui de la classe où celle-ci est trouvée, c'est à dire `AssistanceHot-Line`. L'appel à d'autres méthodes ou l'accès à des attributs au sein de cette méthode se fait alors sous ce point de vue.

Autres approches

Dans la lignée de ROME, il existe plusieurs autres travaux qui traite explicitement de description multiple d'objets en programmation. Nous pouvons citer:

Les travaux présentés dans [Bardou 96] et [Malenfant96] sur les objets morcelés dans les langages à prototypes[1]. Ces objets sont issus d'une rationalisation du lien de délégation entre objets existant dans ces langages. Une étude minutieuse a révélé de sérieux problèmes d'identité et d'encapsulation des objets mis en relation par ce lien qui amène à les considérer comme des morceaux d'un seul et même objet de plus haut niveau. La notion d'objet morcelé vise à donner un statut à de tels objets en interprétant les morceaux comme des points de vue de celui-ci. Un objet morcelé se présente comme l'aggrégation d'entités élémentaires, des morceaux (attributs et méthodes) organisés en hiérarchie de partage d'attributs et de délégation selon les modalités assez fine offerte par ces langages. Un morceau appartient toujours à un seul objet morcelé. Il est désigné par un nom unique à l'intérieur de l'objet morcelé et n'a pas d'identité à l'extérieur. Ce nom sert lorsqu'on veut envoyer un message à l'objet selon un point de vue particulier. Dans ce cas, le message est redirigé vers le morceau correspondant ou éventuellement délégué au morceau parent dans la hiérarchie si la méthode n'y est pas trouvée. Des opérations pour ajouter ou supprimer dynamiquement des morceaux à l'objet morcelé sont également proposées. [Bardou 98] aborde l'intégration des objets morcelés dans un langage à classes en respectant les notions sous-jacentes.

[Kristensen 96] traite également de description multiple d'objets dans un cadre de programmation conceptuelle[2]. Ils introduisent la notion de rôle qui est définie comme "l'ensemble des propriétés qui sont importantes pourqu'un objet soit capable de se comporter d'une certaine manière requise par d'autres objets" et propose de l'intégrer dans un modèle à classes. Un rôle est défini par une classe de rôles qui est obligatoirement rapportée à une classe dont elle enrichit la description de ses instances avec de nouvelles caractéristiques ou la spécialise par redéfinition de ses méthodes. Plusieurs classes de rôles peuvent être définies séparement pour une même classe. Une classe de rôle peut aussi être introduite comme la spécialisation d'une autre classe de rôle plus générale. Les liens possibles entre une hiérarchie de classe de rôles et une hiérarchie de classes sont gouvernés par des règles précises (cf [Kristensen 96] pour plus de détails). A partir d'une classe de rôle, des instances de rôles peuvent être créées qui ne peuvent exister de manière indépendante et n'ont pas d'identité propre. Une telle instance doit obligatoirement être rattachée à un objet, instance de la classe concernée dont elle prend l'identité. L'instance de rôle détient alors l'état et réalise le comportement pour cet objet. Plusieurs instances de rôles différents peuvent être associées au même objet et être référencées. Une référence sur une instance de rôle permet de considérer l'objet sous le point de vue correspondant au rôle. Deux façons de mettre en oeuvre cette notion de rôles dans les langages à classes existants (BETA et Smalltalk) sont exposées dans [Osterbye

1. qui s'oppose au modèle fondé sur la distinction entre classes et instances.
2. qui vise à introduire dans les programmes les mêmes mécanismes d'abstraction (classification, aggrégation, généralisation, "exemplification", décomposition) que ceux utilisés lors de la phase d'analyse et de conception du système.

95].

2.3.3.3 Conclusion

Dans cette partie, nous avons étudié plusieurs techniques offrant des éléments de solutions pour structurer fonctionnellement la description des objets. L'analyse de ces techniques révèle que celles-ci sont surtout prévues pour obtenir cette structuration au niveau granulaire de l'objet. Par rapport aux besoins de description structurée d'un système d'objets relativement aux fonctions, cette granularité reste insuffisante pour les deux raisons suivantes.

La première raison est l'absence de systématisation de structuration sur l'ensemble des objets du système. Un moyen ad-hoc pour y parvenir consiste à réaliser "manuellement" cette structuration classe par classe ce qui peut rapidement s'avérer d'une grande complexité. Cette complexité concerne particulièrement les techniques de descriptions multiples où la structuration d'une classe s'obtient en introduisant plusieurs classes représentant des points de vue différents sur les objets. La figure suivante montre ce qu'il en est avec l'approche par aggrégation pour seulement trois classes et deux fonctions (rem: les liens inverse des perspective vers leur objet ne sont pas figurés; les liens entre objets sont représentés en pointillés pour les distinguer de ceux avec les perspectives).

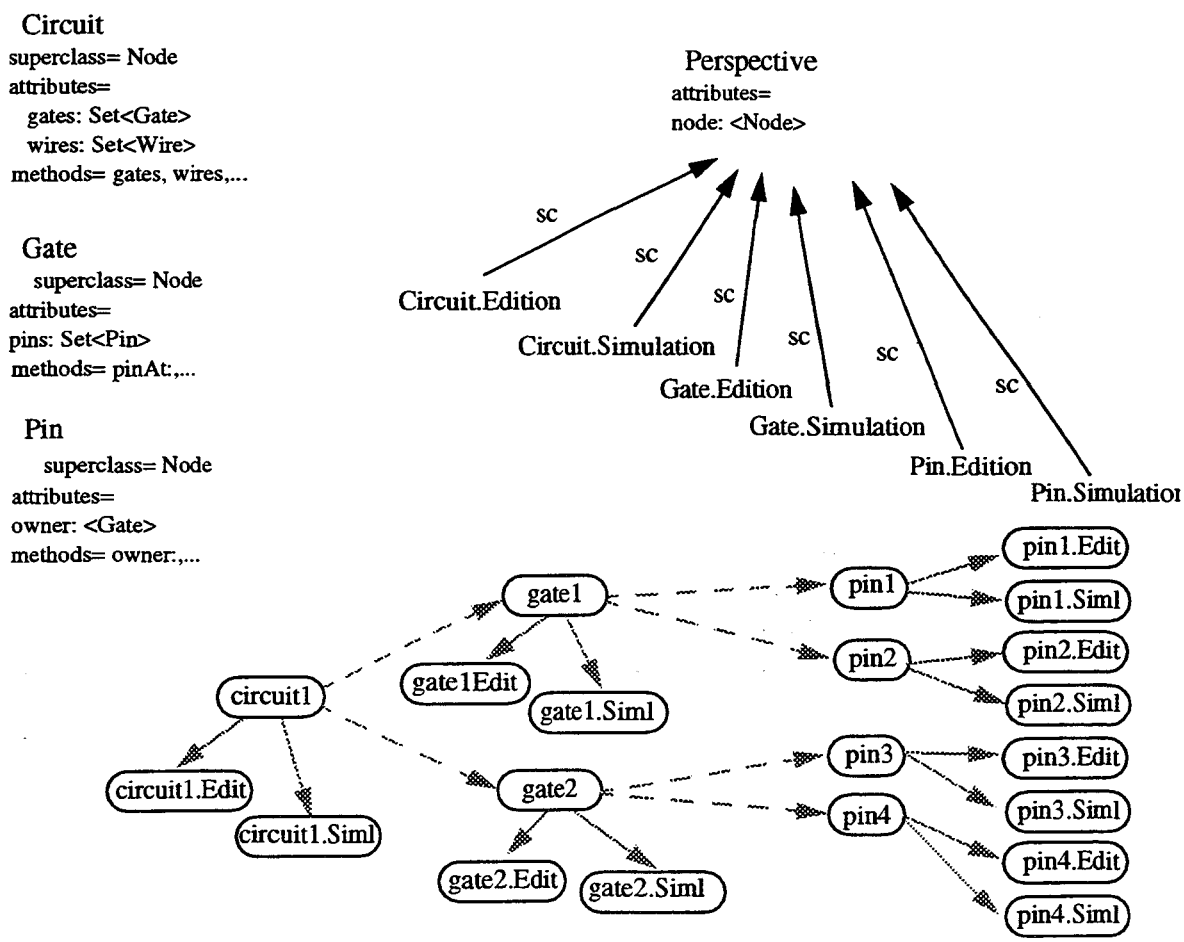


figure 1.15 : Application de l'approche par aggrégation à plusieurs objets

La seconde raison provient de la difficulté de circonscrire les descriptions d'objets, fortement corrélés, relativement aux fonctions du système. Un premier exemple de difficulté mentionné

en 2.3.2.1 est celui de la structuration fonctionnelle d'une hiérarchie de classe par application des techniques de description multiple par points de vue. Dans ce cas, il est nécessaire de gérer la structuration de l'héritage sous une fonction. Un second exemple également évoqué en 2.3.2.1 est celui de la localité des communications entre objets au sein d'une fonction qui ne sont pas garanties systématiquement et doivent alors être assurées par le programmeur.

Un début de prise en charge de la transversalité peut parfois exister. C'est le cas du système Systalk [Wolinksi 90] cité précédemment qui traite la problématique de points de vue et celle d'objets composites en appliquant l'approche par aggrégation. Ce système intègre des mécanismes visant à systématiser la construction de l'objet composite selon tous ces points de vue ainsi que la communication de plusieurs composants selon un même point de vue. Une autre approche gérant un début de transversalité au niveau des relations d'héritage est celle basée sur les composants de classes [Stata 95]. Dans cette approche, l'héritage entre classes se fait selon leur structuration en composants. Insistons cependant sur le fait que chacune de ces approches ne prend en compte qu'un aspect de la transversalité et ne traite pas le problème de manière générale.

En résumé, les techniques présentées ici apportent plus des primitives de structuration qu'une solution globale au problème posé. Ces techniques réclament d'être systématisées transversalement aux objets et d'être gérées au niveau de granularité supérieure des fonctions du système pour mieux circonscrire et corréler la description des objets qui leur est relative. Ce qui fait le lien avec la section suivante.

2.4 Sous l'angle des fonctions

Le problème sous cet angle est d'isoler les axes fonctionnels de description du système orthogonaux aux objets. L'analyse de l'approche objet classique a mis en évidence ses faiblesses vis à vis de ce problème. Ceci nous amène donc à étudier les techniques existantes pour prendre en compte la transversalité de plusieurs fonctions sur un même système d'objets. Les techniques présentées reposent toutes sur l'ajout au modèle objet de nouveaux types d'entité dédiés à l'explicitation des fonctions. Nous allons voir que la nature de ces entités et le support qui leur est consacré varie d'une technique à l'autre.

2.4.1 Activité transversale

[Kristensen 93] traite de la représentation explicite d'activités transversales. Une activité est la combinaison d'actions réalisées localement par plusieurs objets distincts. Pour l'auteur, les activités du système sont des entités aussi importantes que les objets qui les réalisent. Ces activités doivent par conséquent apparaitre dans le développement du système pour mieux le comprendre et mieux le maintenir. Dans ce but, la solution proposée consiste à enrichir le modèle objet avec un nouveau type d'entités appelées "activités transversales".

La structure d'une activité transversale se compose de deux parties:

- une liste de tous les objets intervenant dans l'activité. La nature et le nombre d'objets recensés dans cette partie varient suivant l'activité;

- une directive décrivant le scénario de l'activité, c'est à dire son cycle de vie. Cette directive se présente sous la forme d'une séquence d'envois de message dirigés aux objets participants identifiés dans la première partie. Chaque envoi de message correspond à une action

qu'un participant doit réaliser en local pour supporter l'activité. A travers cette directive, une activité transversale contrôle les objets y participant.

Une activité transversale est définie séparemment des objets sur lesquels elle agit et des autres activités transversales.

Afin d'illustrer cette notion, nous reprenons l'exemple d'"organisation d'une conférence" introduit par l'auteur. Pour simplifier, nous nous intéressons seulement à deux activités qui sont la soumission des papiers et leur évaluation. Dans la premier activité, le président du comité de programme reçoit les papiers des auteurs et leur retourne un accusé de réception. La seconde activité débute par une distribution du papier à trois membres du comité de programme. A partir du papier qu'il a reçu, chaque membre relecteur produit une note d'évaluation. En fonction du score attribué au papier par les relecteurs, le président établit si le papier est accepté ou refusé puis informe les auteurs du résultat.

Après cette brève description de l'exemple, considérons la démarche de conception de l'activité d'évaluation. Cette démarche consiste d'une part à allouer aux objets la partie de l'activité qui leur est spécifique et d'autre part à représenter explicitement l'"activité transversale" correspondante. Pour cette activité transversale, la liste des objets participants est: Auteur, Président, Membre et Papier. Pour sa partie directive, celle-ci se compose de la séquence d'envois de message suivante:

```
the reviewers.evaluate-paper
the chair.make-decision
the paper.accepted or the paper.rejected
   the-author.notify-result
```

Cette directive illustre l'enchainement des messages envoyés aux objets participants pour la réalisation de l'activité d'évaluation des papiers.

La seconde activité est prise en compte de façon similaire à la première ce qui revient à associer une partie de celle-ci à chaque objet y participant et à introduire une activité transversale qui la représente. Pour cette activité, la liste des participants est constituée des objets suivants: Auteur, Président et Papier. On constate que certains des objets mentionnés dans cette liste apparaissent déjà dans celle de la première activité ce qui traduit leur participation multiple.

Le diagramme de la figure 1.16 schématise le résultat pour les deux activités. Celui-ci fait apparaitre les activités comme des entités à part entière (représenté par un cercle), distinctes des objets participants avec lesquels elles coexistent. Chaque activité maintient des liens vers les objets participants (figurés par des flêches pleines) lui permettant ainsi de solliciter ces derniers par envois de message (figurés par des flêches en pointillés) comme indiqué dans sa partie directive (représenté par un rectangle à l'intérieur du cercle).

Par comparaison, les mêmes activités, conçues dans une approche exclusivement objet, sont totalement réparties dans la description des objets participants et existent seulement de façon implicite [Vanwormhoudt 97b]. Ici, celles-ci sont représentées de façon explicite et de manière centralisée, les activités transversales associées localisant tout le comportement collectif dans sa partie directive.

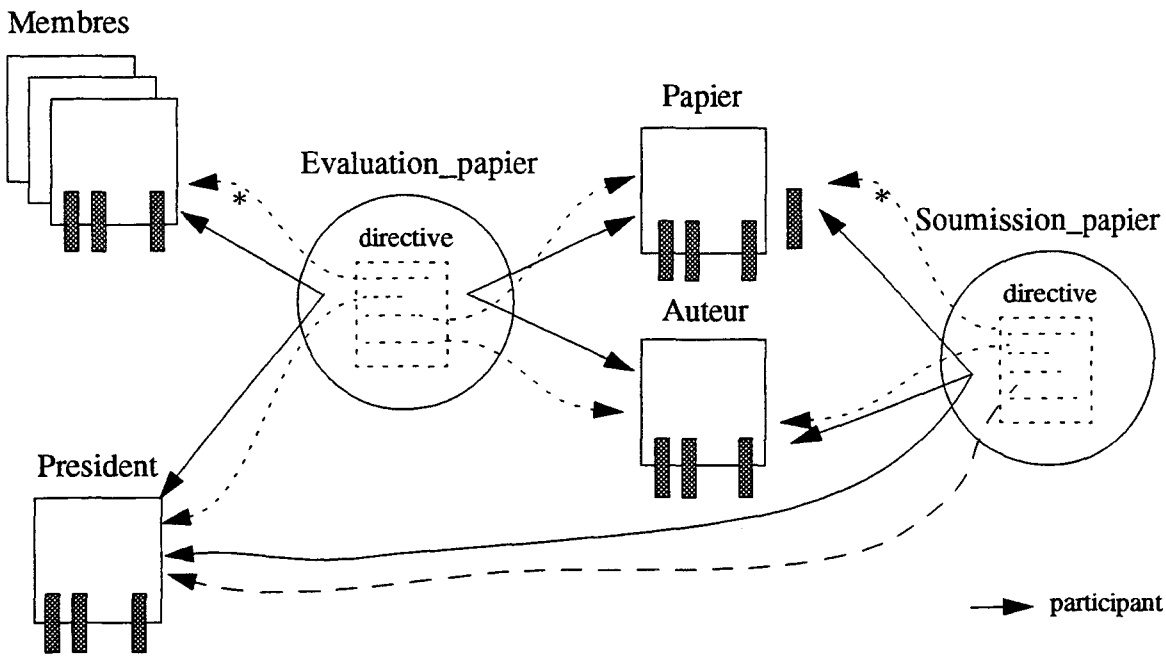


figure 1.16 : Les deux activités transversales de l'exemple

Une nouvelle activité peut être définie par spécialisation d'une autre activité transversale. Dans ce cas, la directive de l'activité est plus spécifique et implique au moins les mêmes participants. Une activité complexe peut aussi être décomposée en activité moins complexe impliquant un sous-ensemble de ses participants..

Afin d'obtenir une implantation du système qui soit proche de sa conception par activité transversale, il est important de supporter cette notion au niveau du langage. Plutôt que de chercher à introduire des mécanismes spécifiques pour obtenir cette notion, l'approche préconisée dans [Kristensen 93] est de la simuler en réutilisant les concepts de base du modèle objet : classes, objet, envoi de message, ... La solution proposée consiste à représenter une activité transversale par un objet ad-hoc dont les attributs servent à référencer chaque participant à l'activité, la directive étant implantée par une méthode. Le lancement de l'activité est réalisé en activant cette méthode par envoi de message.

2.4.2 Classes de Vue

Dans la partie 3.3.1, nous avons présenté la proposition de [Shilling 89] pour étendre la notion de classe avec plusieurs interfaces. Dans cette même étude, cette extension intervient pour élaborer la notion de classe de vue et d'instance de vue (view class, view instance). Une classe de vue[1] représente une activité globale du système dans laquelle participent plusieurs classes d'objets[2].

---

1. Il ne faut pas confondre non plus cette notion avec celle de classe-vue existant dans le domaine des bases de données à objet [Debrauwer 98] où une classe-vue est définie comme le résultat d'une requête et constitue un raccourci pour exploiter cette requête sans devoir la réexprimer lors de chaque utilisation.
2. "A view class is a global abstraction which uses many object classes to provide a unified global behavior"

Une classe de vue est définie par une suite d'associations de la forme (classe d'objet, interface)[1]. Cette liste recense les classes d'objets et l'interface de celles-ci nécessaires à la réalisation de l'activité[2]. Une classe de vue ne détient aucune description d'objets associée à l'activité, ni de code concernant l'activité elle-même. Elle ne fait qu'expliciter l'activité et certains de ses ingrédients. En particulier, la réalisation de l'activité reste intégralement du ressort des objets. Une même classe d'objets peut intervenir dans plusieurs classes de vue distinctes. Cela s'interprète comme la participation de ses instances dans plusieurs activités à la fois. En général, l'interface associée à une classe d'objets varie d'une activité à l'autre. La figure 1.17 donne un exemple dans lequel trois classes `C1`, `C2` et `C3` ont leurs objets qui participent à deux activités `A` et `B` distinctes représentées par les classes de vue `CVA` et `CVB`.

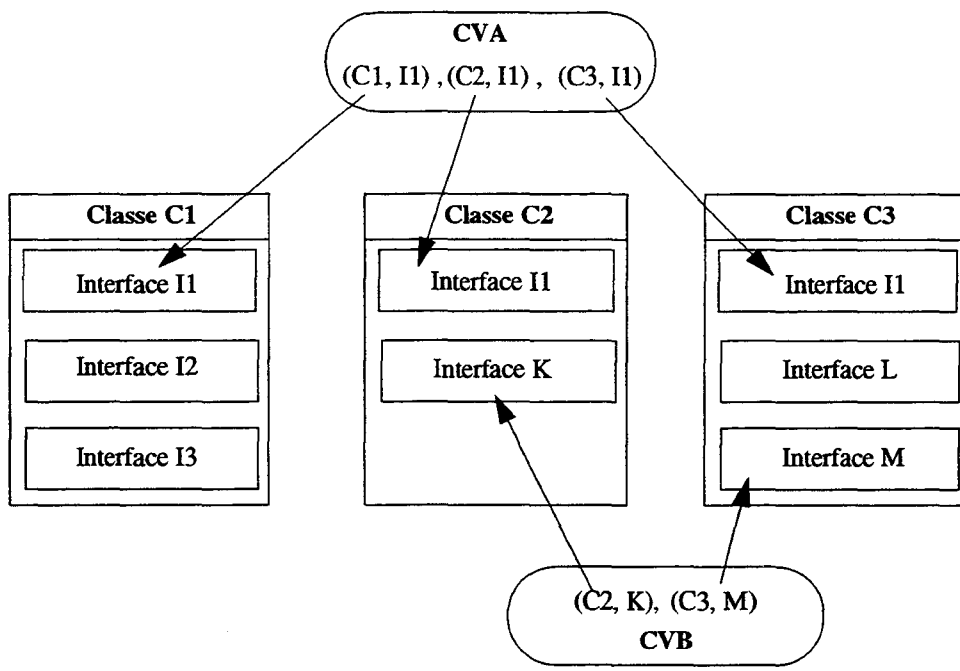


figure 1.17 :Classes de vue et classes d'objets

Une classe de vue est plus qu'une simple notion structurante. Elle sert à créer des instances de vue qui correspondent à des occurences d'activité. Une instance de vue est composée par un ensemble d'objets, instances des classes précisées dans la définition de la classe de vue. Un même objet peut faire partie de plusieurs instances de vue issues de la même classe de vue. Ce cas conduit à la duplication des variables d'instances au niveau de la structure de l'objet. Celui-ci possède alors plusieurs copies du même jeu de variables d'instance, une pour chaque occurence de l'activité. Cette caractéristique permet à l'objet de fonctionner simultanément dans plusieurs occurences de la même activité sans que cela provoque des interférences dans la manipulation des variables d'instances.

La figure 1.18 schématise le cas d'une instance de la classe `C1` participant à deux occurences `A1` et `A2` de la même activité `A` (classe-vue `CVA`) et le cas d'une instance de la classe `C2` participant à `A1` et une occurence `B1` d'une autre activité `B` (classe-vue `CVB`). On peut voir sur

1. C'est la seule information donnée en ce qui concerne la structure d'une classe de vue. Le schéma exact d'une telle classe n'est pas présenté dans l'article.
2. "The object classes are expected, through the interfaces, to provide a unified abstraction for some activity".

cette figure que, du fait de sa participation dans deux occurences de la même activité, l'instance de `C1` dispose d'un double jeu de variables d'instance relatif à l'interface `I1` sélectionnée pour l'activité. Cette figure montre également la participation conjointe de ces deux instances à la même occurence d'activité `A1` avec leurs interfaces propres.

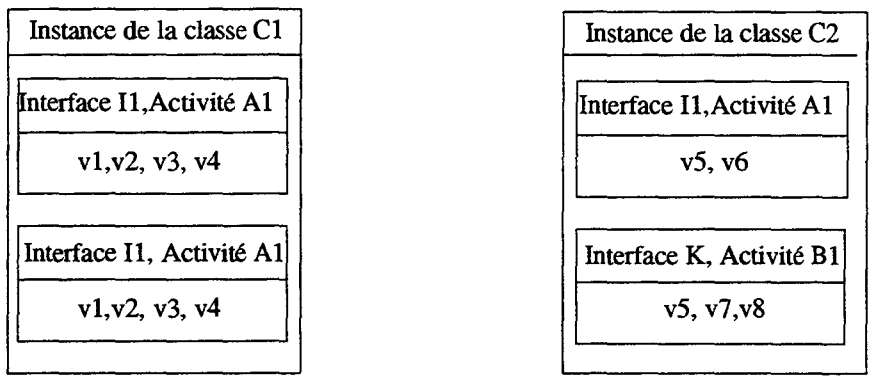


figure 1.18 : Instances participant à plusieurs occurences d'activités

Dans une activité, l'interface spécifiée par la classe vue est la seule disponible pour utiliser l'objet. Pour invoquer une méthode sur un objet dans une activité particulière, il est nécessaire de préciser l'instance de vue dans l'envoi de message. Cette instance de vue est alors exploitée par le système pour déterminer à la fois l'interface à considérer pour le choix de la méthode et le jeu de variables d'instance à utiliser pour l'exécution de la méthode sélectionnée. Toute communication entre objets liés à une même occcurence d'activité repose sur l'utilisation de la même instance de vue.

Selon cette approche, un objet est crée dans le contexte d'une instance de vue. Il existe également une opération spécifique pour rattacher un objet à d'autres instances de vue que celle de sa création. Lorsque l'objet devient membre d'une instance de vue, un jeu de variables d'instance étiqueté par cette instance lui est ajouté. Un objet peut devenir membre d'une instance de vue seulement si sa classe apparait dans la spécification de la classe vue de cette dernière.

2.4.3 Modèles de rôles

Conception

La modélisation par rôle (role modeling) [Andersen 92] est une technique de conception basée sur l'étude séparée des différentes tâches/activités/fonctions d'un système d'objets, celles-ci étant plus ou moins indépendantes entre elles. L'unité de conception est appelée modèle de rôles ("role model"). Un modèle de rôle décrit une tâche particulière sous la forme d'un ensemble de rôles en interaction. Un rôle est une spécification d'un participant à la réalisation de cette tâche. Il exprime les besoins et les responsabilités de celui-ci vis à vis des autres participants de la même tâche, spécifiés par les rôles du même modèle. La description d'un rôle se compose des parties suivantes:

- un nom permettant de le distinguer dans le modèle;
- un domaine spécifiant les autres rôles du modèle avec lesquels il est en relation;
- une interface d'entrée déterminant les messages qu'il reçoit;

- une interface de sortie déterminant les messages envoyés par le rôle vers ceux de son domaine.

Un modèle de rôles peut être défini séparement pour chaque tâche que doit effectuer le système. Pour former le modèle de rôle correspondant au système dans son ensemble à partir des modèles de rôles, une opération de synthèse est proposée. Cette opération consiste à combiner plusieurs modèles des rôles dans un nouveau modèle dont les rôles sont obtenus soit par fusion, soit par conservation des rôles de départ. La fusion est appliquée lorsque plusieurs rôles issus de modèles à combiner doivent correspondre au même participant dans le nouveau modèle[1]. Ces rôles sont regroupés dans un rôle aggrégat au niveau de ce dernier, éventuellement sous un nouveau nom. Ce rôle aggrégat est formé de façon à satisfaire les besoins et les responsabilités exprimées au travers des rôles de départ, en prenant l'union de leurs domaines et de leurs interfaces. Dans le cas où un rôle d'un modèle ne peut être mis en correspondance avec ceux des autres modèles à combiner, ce rôle est conservé dans le nouveau modèle.

La figure 1.19 illustre la synthèse de trois modèles de rôles en reprenant l'exemple donné par les auteurs d'un système de gestion des bouteilles consignées. Les modèles de rôles représentés correspondent à trois des activités d'un tel système: dépot de bouteille (`UserDepositRM`), demande et réception de reçu (`UserReceiptRM`), gestion du niveau de stockage (`StoreLevelRM`). La notation graphique utilisée est celle adoptée par les auteurs. Les ellipses représentent les rôles, ceux obtenus par synthèse étant grisés. Les lignes pleines illustrent les liens entre rôles. Un cercle plein (resp. vide) à une extremité indique une interface de sortie (resp. une référence) vers le rôle situé à l'autre extrémité. Les lignes en pointillés montrent les liens entre les rôles des différents modèles.

A travers cet exemple, nous pouvons identifier plusieurs fusions de rôles. C'est le cas notamment des rôles `User` des différents modèles combinés en un rôle `Customer`. Les rôles `Conveyor` et `Hole` sont également combinés pour former un rôle également nommé Conveyor dans le nouveau modèle. Les rôles `Receipt` et `Compressor` montrent la conservation des rôles d'un modèle au niveau du modèle synthétisé.

Dans le cas particulier où les rôles fusionnés sont en relation avec le même rôle, le domaine du rôle aggrégat n'en contient qu'un seul et son interface de sortie vers ce rôle regroupe les parties des interfaces de sortie de ceux dont il est issu. Dans l'exemple, ce cas se présente pour le rôle `Customer` dont les rôles `User` de départ possèdent respectivement dans leur domaine les rôles `Convoyer` et `Hole` fusionnés dans le modèle final.

Précisons par ailleurs que dans les rôles du modèle synthétisé, les références aux rôles ayant été fusionnés sont substituées par des références aux rôles aggrégats correspondants. Une telle substitution est illustrée dans l'exemple pour le rôle `Account` par la substitution dans son domaine du rôle `User` initial par le rôle `Customer`, résultat de la fusion.

1. La façon dont cette mise en correspondance est précisément effectuée (manuellement ou automatiquement sur la base de noms par exemple) n'est pas indiquée dans l'article.

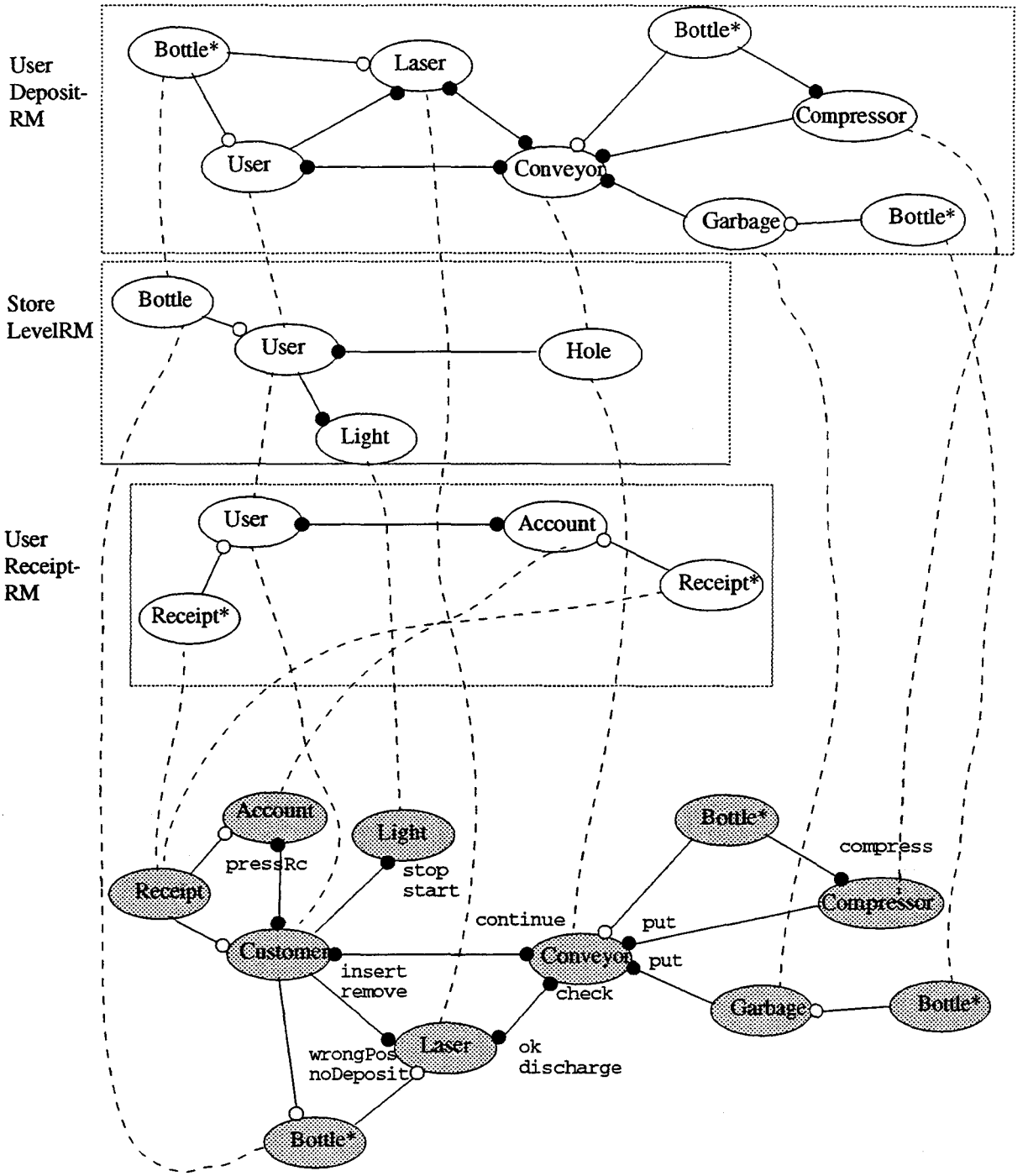


figure 1.19 : Synthèse de trois modèles de rôles

En synthétisant ainsi l'ensemble des modèles de rôles, ce qui peut être fait itérativement, on aboutit à un modèle correspondant au système dans sa globalité. Ce modèle peut alors être utilisé pour déduire l'implantation du système dans un langage à objets traditionnel. Cette implantation peut être obtenue en faisant correspondre une classe à chacun des rôles de ce modèle final, leur domaine et leurs interfaces permettant de déterminer l'interface et l'implantation de ces classes.

L'inconvénient d'une telle implantation est de ne pas faire apparaitre explicitement les modèles de rôles initiaux au niveau de l'implantation. Il en résulte des difficultés lorsqu'il faut isoler la partie de l'implantation correspondant à un modèle de rôle. Pour cette raison, plusieurs travaux ont cherché à préserver les modèles de rôles et à rendre possible leur composition au niveau de l'implantation, en faisant appel à des techniques spécifiques à certains langages. Ces travaux sont exposés dans la prochaine partie.

Implantation

Une première approche est celle de [VanHilst 96] basée sur un usage particulier des classes paramétrables telles qu'elles sont supportées en C++ (e.g templates). L'idée de l'approche est de représenter un rôle par une classe qui admet un paramètre pour la sur-classe[1] et autant de paramètres que de rôles avec lesquels il intéragit dans le modèle de rôle, ces autres rôles étant joués par d'autres classes. Par exemple, le rôle `User` dans le modèle de rôle relatif à la gestion du reçu sera exprimé de la façon suivante:

```
template <class RoleSuper, class AccountRole, class ReceiptRole> class User-
ReceiptRM: public RoleSuper
{
.../* implantation du rôle en utilisant AccountRole, ReceiptRole */
};
```

Les rôles ainsi représentés peuvent alors être composés de manière multiple. Cela se fait en paramétrant les classes correspondantes. Par exemple, pour obtenir la classe `User` correspondant à la synthèse des deux premiers modèles de rôles, on écrira les lignes suivantes:

```
class User1: public UserDepositRM< Laser, Conveyor1, Bottle1>{};
class User: public UserReceiptRM< User1, Account1, Receipt1> {};
```

La première ligne montre le paramétrage de la classe correspondant au rôle User dans le modèle de dépot de la bouteille, `Conveyor1` et `Bottle1` étant elle-mêmes des classes obtenues en paramétrant les classes `ConveyorDepositRM` et `BottleDepositRM` représentant des rôles du premier modèle. La seconde ligne utilise la classe résultante `User1` comme premier paramètre ce qui en fait la sur-classe de la classe `User`.

Cette approche présente comme défaut de nécessiter une paramétrisation des classes représentant les rôles qui devient extrêment complexe dès que le nombre de classes augmente[2].

Une amélioration de l'approche précédente est proposée dans [Smaragdakis 97] en combinant classes paramètrables et classes imbriquées selon la sémantique de C++. L'idée repose sur l'encapsulation des classes paramètrables représentant les rôles d'un même modèle dans une classe englobante représentant le modèle. Une telle classe admet en paramètre une classe de modèle de rôle qui permet de paramétrer les sur-classes de classes de rôle. Par exemple, le modèle de rôle gérant le niveau de stockage peut être exprimé selon cette approche comme suit:

---

1. Une classe dont la sur-classe n'est pas fixée lors de sa définition est appelé "mixin" (dans le sens général de [Bracha 90])
2. Dans le pire des cas, le nombre de paramétrisation nécessaire est égal à m^n pour n modèles composés de m rôles chacun.

```
template <class RoleModelSuper> class StoreLevelRM: public RoleModelSuper
{
public:
   class Bottle : public RoleModelSuper::Receipt {...};
   class User : public RoleModelSuper::User{...};
   class Conveyor: public RoleModelSuper::Conveyor {...};
   class Light {...}; // classe spécifique à ce rôle
};
```

Cette représentation des modèles de rôles permet de le composer selon des ordres multiples[1]. Une composition particulière de modèles est réalisée en paramétrant une suite de classes englobantes comme suit:

```
typedef < StoreLevelRM < ReceiptRM < DepositRM >>> finalRM;
```

Ce paramétrage revient à placer les classes englobantes dans une hiérarchie d'héritage, ce qui entraine également un héritage entre leurs classes de rôle ayant le même nom. Ainsi, selon l'exemple de composition donné ci-dessus, la classe de rôle `User` définie dans la classe `StoreLevelRM` a pour sur-classe directe la classe `User` définie dans la classe `ReceiptRM`, celle-ci héritant à son tour de la classe `User` de `DepositRM`. Les classes contenant les rôles de chaque modèle sont alors accessibles à travers le type `finalRM` qui est la classe englobante résultant de la paramétrisation.

Comparée à la solution précédente, nous constatons que cette approche a l'avantage de rendre plus explicites les modèles de rôle dans l'implantation et facilite leur composition par le biais du paramétrage. La généralisation de cette approche à d'autres langages comme CLOS et Java est également discutée par les auteurs.

2.4.4 Programmation par sujets

La programmation par sujets ("Subject-Oriented Programming") introduite dans [Harrisson 93] est une extension de la programmation par objets pour la construction de systèmes complexes faisant intervenir plusieurs applications opérant sur les mêmes objets. Cette technique de programmation propose d'élaborer ces systèmes en composant un ensemble de sujets où un sujet détermine les objets et leur description selon le point de vue d'une application. Au niveau d'un sujet, les objets sont introduits et décrits de façon classique par un ensemble de classes organisées selon la relation d'héritage. Chaque sujet définit et organise ses classes indépendamment de celles des autres sujets.

Plusieurs sujets peuvent être composés pour produire un nouveau sujet incluant tous les détails de ceux dont il est issu. Cette composition, réalisée de façon semi-automatique, est exprimée par des règles de composition intervenant à des niveaux de détails différents. Les règles de bas-niveau permettent de spécifier finement les correspondances entre les sujets à composer:

- la correspondance d'opérations: des sujets différents peuvent utiliser des opérations distinctes pour le même objectif ou introduire des opérations qui sont complémentaires. Cette correspondance permet de spécifier que l'activation d'une de ses opérations dans un sujet entraine aussi l'activation des opérations correspondantes dans les autres sujets.

1. Les noms de classes de rôles correspondant aux mêmes objets doivent rester identiques.

- la correspondance de classes : celle-ci permet d'indiquer que des classes déclarées dans des sujets différents avec des noms et des caractéristiques différents concernent les mêmes objets.

- la correspondance de variables d'instance : certaines variables d'instances déclarées dans des classes de sujets différents peuvent s'avérer être les mêmes. Cette correspondance permet le partage de ces variables d'instances entre les sujets.

- la combinaison de méthodes : lorsque plusieurs sujets déterminent des implantations différentes pour des opérations correspondantes, la façon dont ces implantations sont combinées peut être spécifiée précisément.

Des règles de composition de plus haut-niveau construites à partir des règles précédentes et d'opérateurs de combinaison[1] de règles sont également proposées. Deux règles ont été isolées comme étant particulièrement utiles: il s'agit des règles de fusion (merge) et de surchage (override). La première permet de fusionner plusieurs sujets en une seule hiérarchie de classes globale, alors que la seconde permet d'étendre un sujet de façon non destructive.

Pour illustrer la notion de sujet et la composition de plusieurs sujets, considérons l'exemple donné dans [Ossher 95] d'un système de location de voitures faisant intervenir deux applications, une (`Renting`) destinée à l'établissement d'une location par un client et une autre (DMV) pour les vérifications administratives du conducteur. Chaque application amène à définir un sujet séparé comme le montre la figure 1.20 et la figure 1.21.

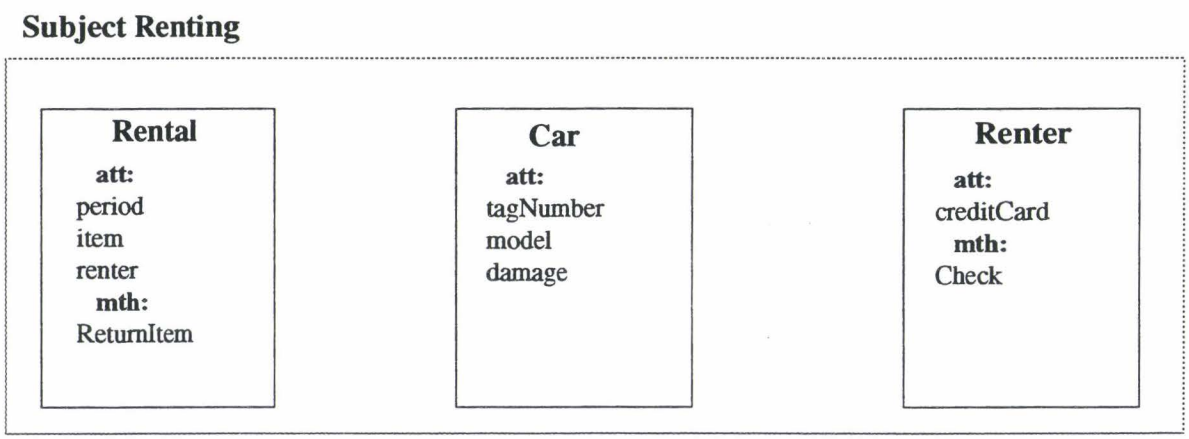


figure 1.20 :Sujet pour l'application établissant la location

1. Contrairement aux règles qui opèrent sur le contenu des sujets, ces opérateurs opèrent sur les règles elles-mêmes.

Subject DMV

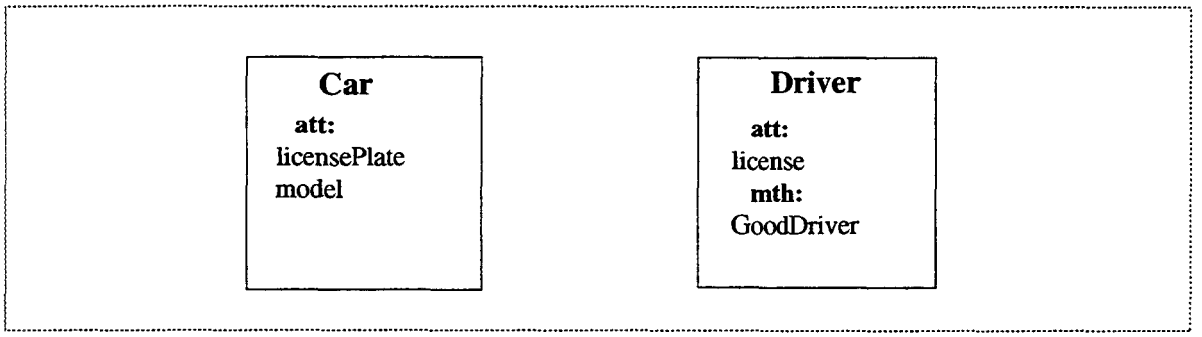


figure 1.21 : Sujet pour les vérifications administratives

Ces sujets peuvent ensuite être composés pour obtenir un sujet satisfaisant les besoins des deux applications et prenant en compte les correspondances entre les sujets. La figure 1.22 montre le sujet résultant d'une telle composition.

Subject CarRenting

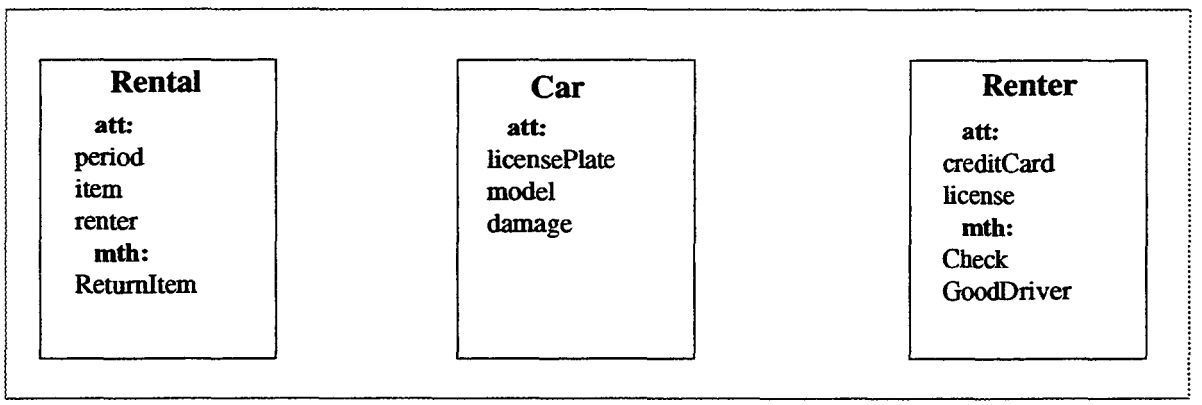


figure 1.22 : Sujet résultant de la composition

L'expression produisant le sujet présenté ci-dessus est la suivante:

```
merge_s(CarRenting, <Renting, DMV>)
   except merge_c(CarRenting.Renter, <RentingRenter, DMV.Driver>)
          merge_v(CarRenting.Car.licensePlate,
                  <Renting.Car.tagNumber, DMV.Car.licensePlate>)
          equatek[compose_o[equivalent],1]
                 (CarRenting.Check, <Renting.Check, DMV.GoodDriver>)
```

La règle de base utilisée est celle de fusion (`merge-s`) qui fait la correspondance sur l'identité des noms. Trois sous-règles additionnelles viennent toutefois spécifier des exceptions à la sémantique standard de cette fusion. La première spécifie que les classes `DMV.Driver` et `Renting.Renter` sont à fusionner même si leur nom ne correspondent pas. La seconde exception stipule que les variables d'instances `tagNumber` et `licensePlate` des classes `Car` de chaque sujet sont à fusionner dans la classe `Car` correspondante. La troisième exception spécifie que l'opération `Check` dans le sujet combiné est une combinaison de l'opération `Check` initiale avec l'opération `GoodDriver`.

Précisons qu'à la base il n'existe pas de règles pour résoudre les conflits de structure entre sujets qui peuvent se produire du fait de leur conception séparée. Le cas où les mêmes objets

sont décrits par une classe dans un sujet et par plusieurs dans un autre est un exemple d'un tel conflit. Dans ce cas, la solution passe par l'élaboration de nouvelles règles de composition qui peuvent être obtenues en utilisant les règles et les opérateurs de combinaison existants.

2.4.5 Conclusion

Dans cette partie, nous avons passé en revue quatre techniques qui montrent la problématique de l'identification d'un ensemble de fonctions transversales aux objets et la recherche de solutions pour la résoudre. L'analyse de ces techniques révèle cependant des lacunes pour règler le problème de la structuration interne des objets relativement aux fonctions auxquelles ils participent.

Pour les deux premières techniques présentées, nous avons vu que l'explicitation est réalisée à l'aide d'entités spécifiques (activité transversale et classe de vue) définies parallèlement aux objets et classes. Cette expression des fonctions pose deux problèmes. Premièrement, cela ne permet pas de rapporter la structuration en fonctions au sein des objets participants. Deuxièment, les ingrédients de chaque fonction qui peuvent ainsi être représentés sont relativement réduits. C'est ce que nous avons pu constater par exemple pour la technique des classes vues où les seuls éléments explicités par ces classes sont les classes participantes et les interfaces utilisées. Une solution pour représenter davantage d'ingrédients liés aux fonctions au niveau de ces entités consiste à externaliser certains ingrédients rangés dans les objets. C'est ce qui est fait dans l'approche par activités transversales où les collaborations entre objets sont décrites par les entités introduites à cet effet. Cependant, cette explicitation est obtenue au détriment des objets qui sont ainsi vidés de leur composante collaborative.

Les dernières approches présentées (modèles de rôles et programmation par sujets) montrent bien la nécessaire prise en compte des multiples dimensions fonctionnelles d'un système pour la conception des objets. Les modèles de rôles se situent essentiellement à un niveau préalable de spécifications. Les descriptions d'objets sont ensuite obtenues par fusion des modèles. La programmation par sujets permet également de réaliser une telle fusion par des procédés de composition de sujets. Cependant, le problème de la structuration en fonctions du système d'objets résultant reste entier.

2.5 Conclusion générale

Dans ce chapitre, nous avons examiné la problématique de plusieurs fonctions sur un même système d'objets qui se rencontre dans les environnements logiciels de conception et d'ingénierie. Nous avons montré que cette problématique réclame une double structuration prenant en compte à la fois les objets et les fonctions transversales à ces objets en les articulant.

Pour prendre en compte ces besoins de structuration, nous avons étudié des débuts de solutions sous les deux angles. Il s'est avéré qu'en général les solutions proposées permettent surtout la prise en charge de la structuration au niveau d'un seul axe laissant l'articulation avec l'autre axe à la charge du programmeur. Ainsi, nous avons vu que les techniques de description modulaire d'objets permettent de structurer fonctionnellement les descriptions des objets mais ne tiennent pas compte de la transversalité des fonctions. Inversement, nous avons vu que les approches explicitant les fonctions transversales ne font pas le lien avec la structuration interne des objets.

Fort de cette analyse et de ces constats, nous voulons proposer une approche qui comble ces

lacunes et réalise l'intégration des deux axes de structuration. Cette approche que nous avons appelée CROME est présentée dans le prochain chapitre. Nous verrons que celle-ci tire profit des moyens de structuration offerts par les deux axes.

3

L'approche CROME

3.1 Introduction

Guidé par le problème de l'orthogonalité objets-fonctions analysé dans la première partie de ce mémoire, nous présentons dans ce chapitre notre proposition CROME [Carré96, Debrauwer97, Vanwormhout97a-b] (Contextes en ROME [Carré89, Carré90a-b]). CROME est issu de l'étude de la systématisation des points de vue dans la lignée du langage ROME. Il a été généralisé comme cadre programmatique pour concevoir et programmer modulairement des objets intervenant dans des contextes fonctionnels multiples.

Dans ce chapitre, nous présentons les notions et principes essentiels de CROME. Les principaux points qui seront développés sont les suivants:

- CROME introduit une structuration en plans de la description des objets du référentiel, relativement aux fonctions. Cette structuration est exposée dans les sections 3.2 à 3.4.

- Il s'en déduit une stratégie d'héritage modulaire, basé sur la notion de plan, pour déterminer la description des objets pour une fonction particulière. Cet héritage modulaire est expliqué dans la section 3.5.

- La communication entre objets peut alors être circonscrite relativement aux fonctions offrant une modularité de programmation des traitements fonctionnels multiples assurés par les objets. Ceci est traité dans la section 3.6.

- Systématisés sur un graphe d'objets, les principes précédents autorisent la fonctionnalisation multiple de tels graphes (structure commune d'un système, objets complexes). Ceci fait l'objet de la section 3.7.

- Nous détaillons ensuite en section 8 les bénéfices de l'héritage modulaire par plans qui permet la factorisation de code aux niveaux des fonctions prises dans leur ensemble, c'est à dire à un niveau de granularité plus riche que celui de la classe.

- Enfin nous montrons en section 9 comment les problèmes d'articulations entre fonctions se déclinent et sont résolus originalement au sein des objets eux-mêmes. L'objet offrant par là un espace d'articulation entre fonctions du système.

Après la présentation de ces différents points, nous proposons dans la section 3.10 d'évaluer CROME. Ses propriétés structurantes et ses qualités modulaires y sont explicitées. CROME est également analysé sous l'angle de la réutilisibilité et de l'extensibilité.

Tout au long de ce chapitre, nous présentons les différentes notions en retenant les propriétés du modèle de programmation par objets le plus général, celui des langages à classes, dans lequel les objets sont caractérisés par un ensemble d'attributs privés et de méthodes et communiquent par messages. Pour les méthodes, nous faisons la distinction entre méthodes publiques qui sont accessibles par envoi de message aux autres objets et les méthodes privées seulement

applicables par l'objet lui-même.

Les questions liées à la mise en oeuvre des propositions de CROME ne sont pas abordées dans ce chapitre. Elles seront traitées dans la dernière partie du mémoire où nous verrons son application à ROME en tant que langage de programmation et à Smalltalk, comme environnement de conception.

L'ensemble des sections s'appuie sur l'exemple du système d'aide à la conception de circuits logiques introduit au premier chapitre. Le code complet de cet exemple exprimé en ROME peut être trouvé dans le chapitre Annexe.

3.2 Programmation par plans

La solution proposée par CROME est fondée sur une description par plans des objets du référentiel. La notion de plan n'est pas une nouvelle construction du modèle objet. C'est une notion structurante intervenant à la conception qui systématise transversalement la description des objets et permet de rendre compte de la double structuration objets/fonctions.

CROME introduit deux sortes de plans pour décrire les objets: le plan de base et les plans fonctionnels.

Le plan de base est le premier plan à définir. Celui-ci détermine:

- l'identité des objets du référentiel;
- la description de ces objets partagée par toutes les fonctions.

Les objets introduits par le plan de base sont ensuite décrits selon les spécificités des fonctions. Pour chaque fonction, cette description s'effectue par un plan fonctionnel qui fournit pour les objets du référentiel y intervenant leur description spécifique à celle-ci.

La figure 3.1 montre le plan de base et les plans fonctionnels correspondants à notre exemple. Au niveau du plan de base, les objets identifiés pour le référentiel sont ceux qui entrent dans la constitution des circuits (`Circuit`, `Wire`, `Gate`, ...). Nous pouvons constater qu'il y a un plan fonctionnel associé à chaque fonction du système: Édition, Simulation, Documentation, Normalisation et Optimisation. Ces plans participent conjointement à la description des objets du référentiel. Pour l'ensemble (ou un sous-ensemble) de ces objets, chaque plan fonctionnel détermine leur description relativement à la fonction correspondante. Ainsi, le plan fonctionnel associé à la fonction de simulation détient une description pour les objets `Circuit`, `Wire` et `Gate` qui est distincte de celles fournies pour ces mêmes objets par le plan de base et les autres plans fonctionnels.

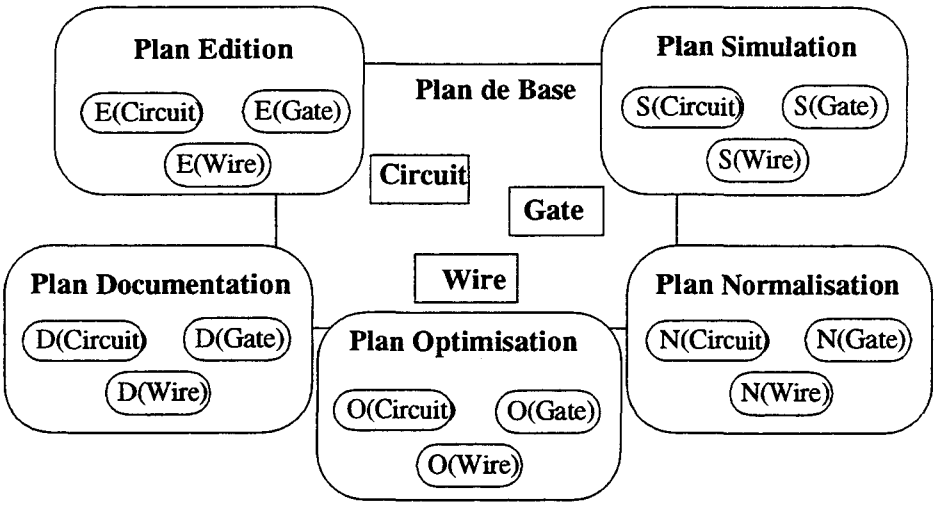


figure 3.1 Structuration par plans de l'exemple

Cette description par plans des objets permet de prendre en compte chaque axe de structuration de l'orthogonalité objets/fonctions. Ainsi, le plan de base permet d'établir les axes des objets. Dualement, les plans fonctionnels permettent d'expliciter et de décrire isolément chaque axe fonctionnel de description du système. Selon ces plans, l'articulation entre les deux axes de structuration se traduit de la façon suivante:

- la participation des objets aux différentes fonctions se traduit par un ensemble de descriptions, chacune située dans un plan fonctionnel différent.

- la participation de plusieurs objets à une fonction se décrit par un ensemble de descriptions respectives localisées modulairement dans le même plan.

Par la suite, nous appelons *partie fonctionnelle* la description des objets relative à une fonction.

3.3 Plan de base

Ce premier plan détermine la structuration en classes du référentiel: les objets y sont à la fois identifiés et décrits par des classes organisées selon des relations d'héritage et des relations entre objets. Chacune de ces classes détient la description de base des objets qui correspond à leur partie partagée par toutes les fonctions.

Dans le cas de notre exemple, le plan de base est représenté graphiquement à la figure 3.2. La structuration en classes est semblable à celle obtenue pour l'approche objet classique (cf chapitre 2).

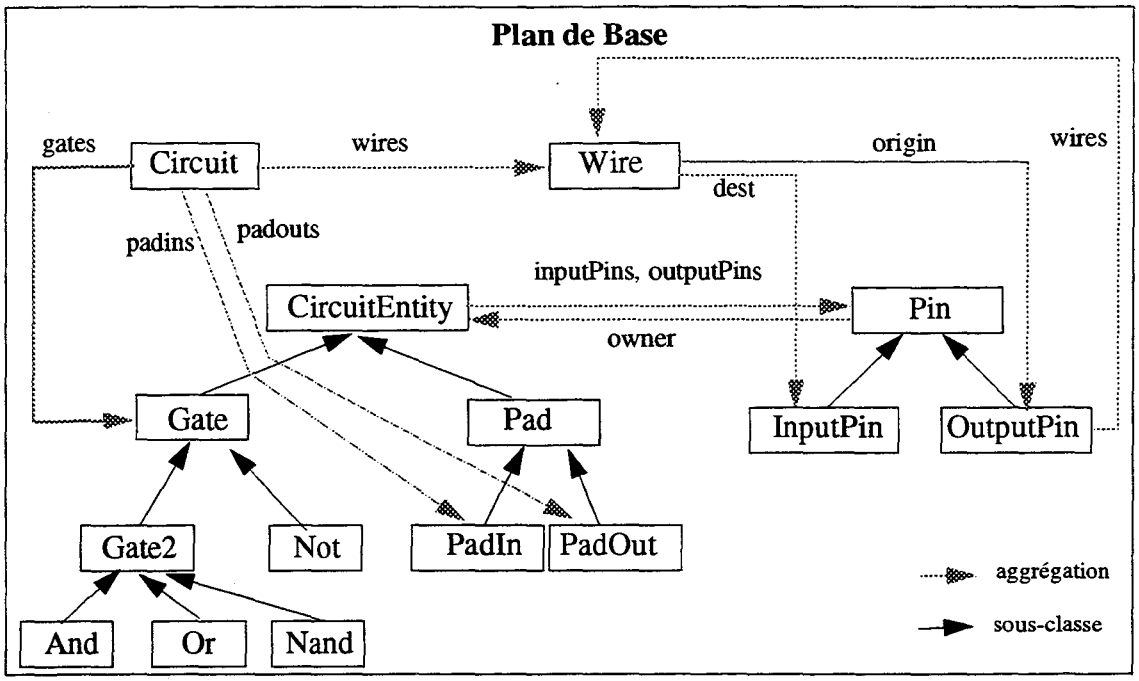


figure 3.2 Plan de base de l'exemple

Description de base

La description de base fournie par les classes du plan de base a un objectif bien précis: elle sert à introduire les caractéristiques (attributs et méthodes) des objets partagées par les différentes fonctions. Ces caractéristiques peuvent être issues de la mise en facteur entre fonctions ou représentées des aspects des objets existant indépendament des fonctions. Elles n'ont pas forcément besoin d'être pertinentes pour toutes les fonctions. Certaines de ces caractéristiques peuvent notamment être ignorées dans le cadre d'une fonction particulière.

CROME n'impose pas de contrainte sur la description des objets dans le plan de base. Cette description peut être plus ou moins riche suivant le niveau de partage requis entre fonctions. Par exemple, dans le cas où le partage entre fonctions porte uniquement sur des données ou des relations, les objets peuvent y être décrits uniquement par un ensemble d'attributs sans qu'il leur soit conféré de méthodes, celles-ci étant apportées par les plans fonctionnels.

Pour notre exemple, la description de base associée aux différentes classes montrée précédemment comprend toutes les caractéristiques nécessaires pour représenter un circuit du point de vue de sa structure interne. Cette structure est en effet partagée par la plupart des fonctions sans être spécifique à l'une d'entre elles. Il faut souligner que c'est souvent le cas dans les applications de CAO notamment où la description de la structure des entités à concevoir sert de reférence aux fonctions de l'environnement (voir par exemple [Carn 92][Deconninck 92][Wolinski 90]).

Relation d'héritage entre classes du plan de base

Dans le plan de base, toute classe peut être liée par la relation d'héritage à une autre classe dont elle devient une sous-classe[1]. Ceci permet d'organiser les classes du référentiel en hiérarchie d'héritage plus ou moins complexe. Dans le plan de base de notre exemple illustré

précédemment, deux hiérarchies de classes peuvent être identifiées: celle introduite par la classe `CircuitEntity` correspondant aux types d'entités pouvant composer la structure interne d'un circuit et celle introduite par la classe `Pin` correspondant aux types de broches appartenant aux portes.

Dans la construction d'une hiérarchie de classes, l'héritage peut être utilisé de diverses façons [Carré 89] qui peut aller d'une approche conceptuelle (héritage respectant le principe d'inclusion) à une approche purement utilitaire (héritage de code uniquement). Dans le cadre de CROME, l'héritage entre classes du plan de base est employé selon une approche conceptuelle, principalement pour traduire une hiérarchie d'abstraction/concrétisation pour laquelle une sur-classe (par ex. `Gate`) est l'abstraction de ses sous-classes (`AndGate`, `OrGate`, `NotGate`). Dans le cas présent, la construction d'un hiérarchie de classes se fait en effet selon une analyse ascendante à partir des classes des objets intervenant directement dans le référentiel. Les sur-classes (`Gate`, `Gate2`) sont définies conjointement avec les sous-classes (`AndGate`, `OrGate`, `NotGate`) pour répondre à des points précis: factoriser les caractéristiques et rendre possible la liaison dynamique dans chacune des fonctions.

L'utilisation de l'héritage pour l'abstraction amène à considérer deux types de classes: classe abstraite et classe concrète. En CROME, ces deux types de classes sont distingués et coexistent dans la hiérarchie du plan de base. Par exemple, la classe `Gate` mentionnée précédemment est un exemple de classe abstraite, elle n'a que le rôle d'abstraction de ses sous-classes et n'est pas succeptible d'être instanciée. Une autre classe abstraite dans notre exemple est la classe `Pin` qui est l'abstraction des classes `InputPin` et `OutputPin`.

L'héritage entre classes du plan de base peut être utilisé pour exprimer la description de base des objets. Les possibilités offertes dans ce cas sont les mêmes que celles rencontrées classiquement entre une classe et ses sur-classes. Dans la description de base d'un classe, on peut utiliser les attributs et méthodes hérités de la description de base des sur-classes mais aussi redéfinir ces dernières.

Cette utilisation de l'héritage pour la description de base conduit à articuler deux types de factorisation: l'une étant liée aux classes, l'autre aux fonctions. Cette articulation est illustrée dans le plan précédent par la relation `owner` définie dans la description de base de la sur-classe `Pin`. Des exemples mettant à profit cette double factorisation offerte par le plan de base seront développés en section 3.7 et 3.8.

Relation entre objets dans le plan de base

Des relations entre les objets des différentes classes peuvent également être introduites au niveau du plan de base, en utilisant les attributs des descriptions de base. Ces relations donnent la possibilité d'organiser les objets du référentiel selon une structure partagée par toutes les fonctions. L'exploitation de cette structure partagée pour supporter plusieurs fonctions sera décrite en section 3.7.

Dans le cas de notre exemple, plusieurs relations inter-objets sont introduites au niveau du

1. Nous faisons l'hypothèse d'un héritage simple entre les classes. L'héritage multiple peut être envisagé mais sa pertinence et son intégration dans le cadre de CROME reste à étudier. Cela nécessite en particulier de re-examiner les problèmes classiques dus à l'héritage multiple selon les spécificités de ce dernier.

plan de base. Ces relations permettent d'agencer les objets selon un graphe qui réflète la structure interne du circuit. La figure 3.2 met en évidence ces relations inter-objets entre les classes. La figure 3.3 donne une représentation d'un graphe d'objets issu de ces relations correspondant à un circuit Nand conçu par combinaison d'une porte ET et d'une porte Non.

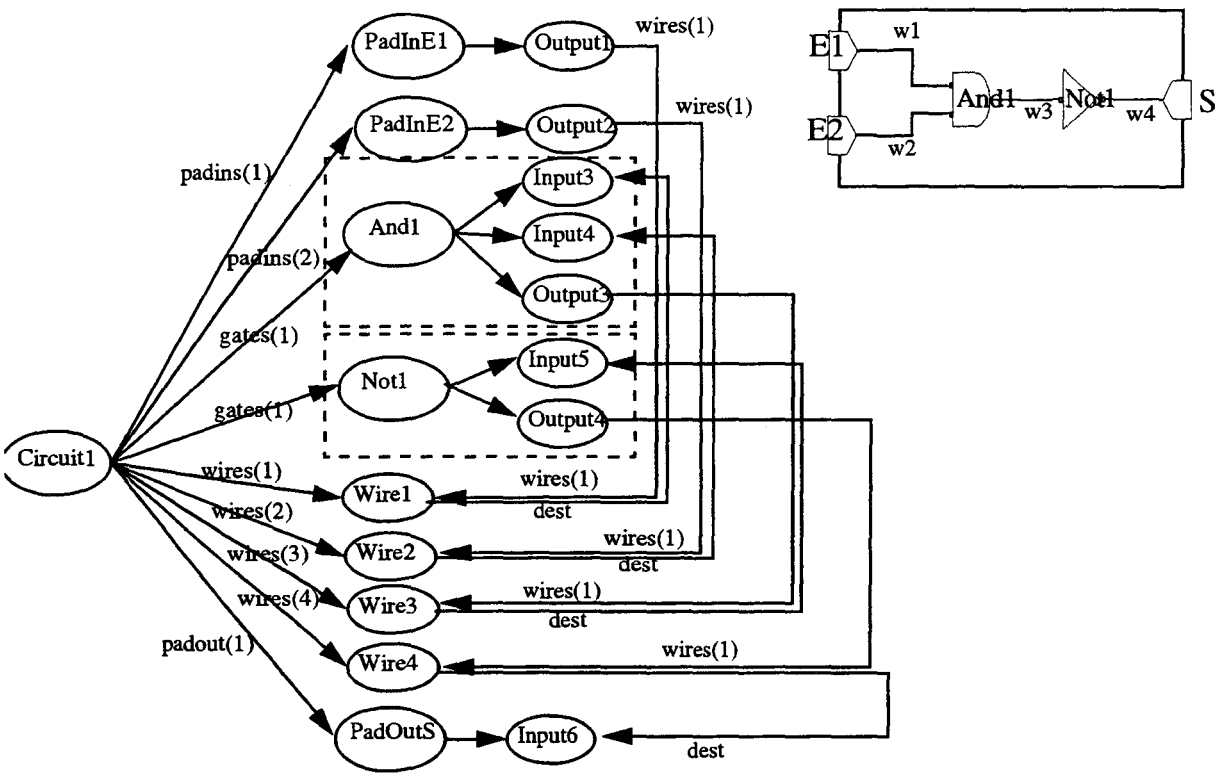


figure 3.3 Graphe d'objets représentant un circcuit Nand

Les relations inter-objets introduites dans la description de base d'une classe peuvent être utilisées pour programmer des traitements à ce niveau. Nous reviendrons plus en détail sur la description de ces traitements et les capacités offertes pour les contextualiser en section 3.8.

3.4 Plans fonctionnels

3.4.1 Structure d'un plan fonctionnel

Les plans fonctionnels sont associés aux fonctions du système et interviennent une fois le plan de base déterminé. Comparé à ce plan, les plans fonctionnels n'ont pas vocation à introduire de nouveaux objets. Leur but est d'enrichir les classes du plan de base pour les contextes fonctionnels correspondants.

Au niveau d'un plan fonctionnel, cet enrichissement est spécifié à l'aide de parties fonctionnelles qui sont associées aux classes du plan de base concerné par la fonction associée. Une partie fonctionnelle détermine les caractéristiques (attributs et méthodes) qu'il faut ajouter à une classe pour que ses objets (représentants pour une classe abstraite) interviennent dans la fonction associée.

Pour chaque classe introduite dans le plan de base, un plan fonctionnel ne peut détenir qu'une et une seule partie fonctionnelle. Mis à part cette contrainte d'unicité, il n'y a pas de restriction

sur les classes du plan de base pouvant être enrichies. Autrement dit, toute classe peut recevoir une partie fonctionnelle, indépendamment de sa nature abstraite ou concrète et de ses relations (héritage et inter-objet) avec d'autres classes dans le plan de base.

Par ailleurs, il peut exister des fonctions qui ne requièrent pas l'enrichissement de toutes les classes du plan de base. Pour ces classes, les plans fonctionnels correspondants n'introduisent aucune partie fonctionnelle. Ajoutons toutefois que, pour une classe, l'absence de partie fonctionelle associée dans un plan fonctionnel ne signifie pas obligatoirement la non participation de ses objets à la fonction correspondante du fait d'un enrichissement implicite que nous présenterons en 3.4.3.2.

Illustrons à présent la structure d'un plan fonctionnel en considérant celui qui est défini pour la fonction d'édition de notre exemple. Ce plan est représenté graphiquement à la figure 3.4 sans montrer tous les détails des parties fonctionnelles ni les relations implicites existant entre celles-ci. Afin de faire la distinction entre le plan de base (resp. les classes du plan de base) et un plan fonctionnel (resp. les parties fonctionnelles) nous faisons le choix pour le reste du chapitre de représenter ces derniers par des rectangles aux coins arrondis. Les parties fonctionnelles des classes sont nommées selon la convention F(Classe), F étant la première lettre de la fonction.

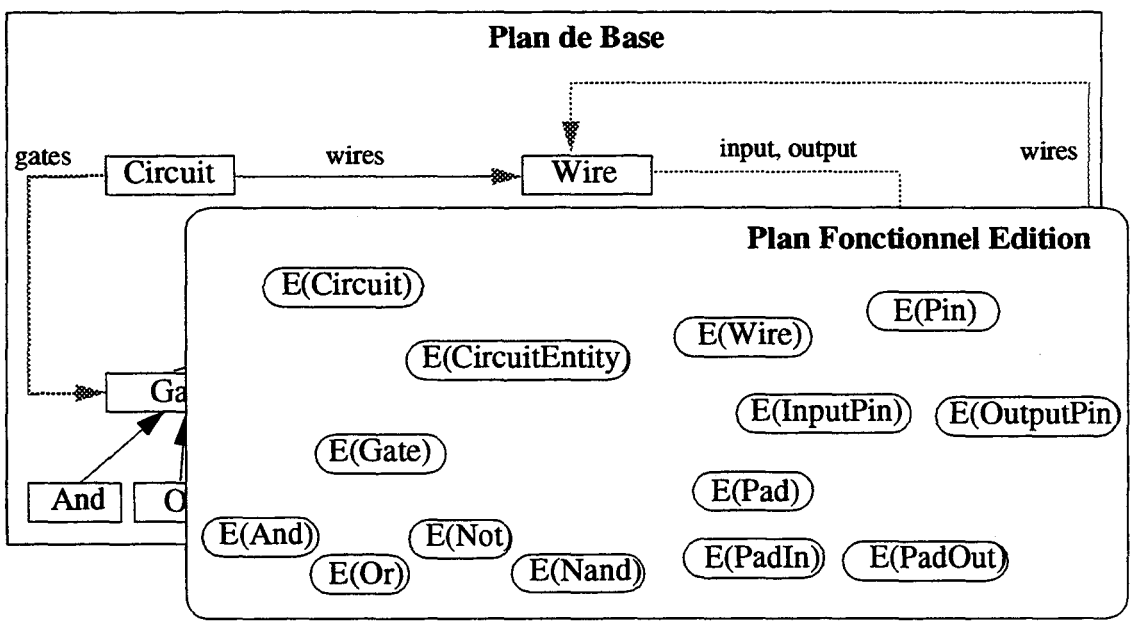


figure 3.4 : Plan fonctionnel relatif à la fonction d'édition des circuit

Dans la section 2, nous avons vu que cette fonction fait participer l'ensemble des objets du référentiel. Le plan fonctionnel correspondant confère par conséquent des parties fonctionnelles à leurs classes. Des parties fonctionnelles sont également apposées à leurs sur-classes (cf 3.4.3). Pour chaque classe ces parties fonctionnelles ne contiennent que des attributs et des méthodes spécifiques à la fonction d'édition.

Un plan fonctionnel détermine un ensemble de parties fonctionnelles pour les classes du plan de base, indépendamment des autres plans. Cela donne la possibilité dans un plan fonctionnel d'enrichir les classes du plan de base sans se préoccuper de leur enrichissement par les autres plans. Par conséquent, une classe peut être enrichie dans un plan, sans l'être dans un autre plan. D'autre part, deux plans fonctionnels peuvent enrichir un sous-ensemble distinct de classes.

Illustrons ces points en donnant un aperçu d'un second plan fonctionnel, celui qui est fourni pour la fonction de documentation. Ce plan est représenté graphiquement comme le précédent par la figure 3.5.

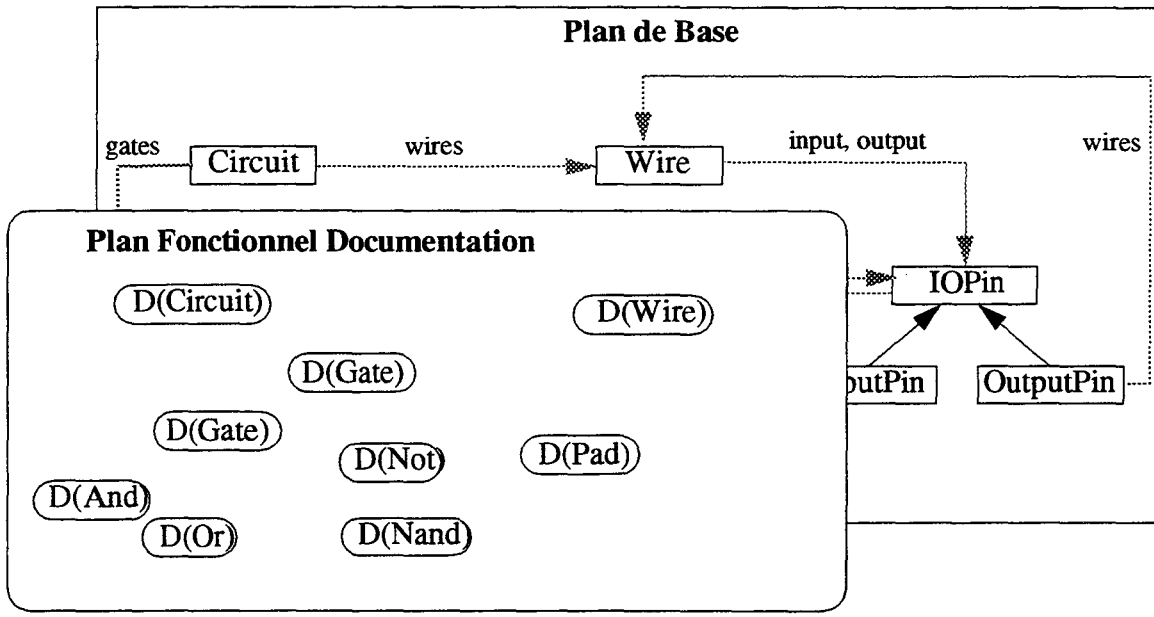


figure 3.5 Plan fonctionnel relatif à la fonction d'édition des circuit

En examinant ce plan, nous pouvons constater qu'il ne détient pas de partie fonctionnelle pour les classes `Pin`, `InputPin` et `OutpuPin` du plan de base. Ceci est dû à la non participation de ces objets dans cette fonction. Les autres classes du plan de base sont par contre également enrichies au niveau de ce plan ce qui leur confère des parties fonctionnelles respectives.

Les autres plans fonctionnels de notre exemple peuvent être développés de façon analogue.

Par rapport aux difficultés pour expliciter les fonctions transversales et leurs ingrédients dans l'approche classique (cf 2.3.2.1), les plans fonctionnels présentent les intérêts suivants:

Premièrement, ils permettent de rendre compte de la structuration en fonctions du système tout en préservant la structuration en objets et en classes. Chaque dimension fonctionnelle, circonscrite par un plan, diffuse sa description transversalement aux objets, selon les principes chers à la programmation par objets.

Un autre intérêt des plans est qu'ils permettent de circonscrire immédiatement les objets et les classes intervenant dans la réalisation de chaque fonction. Ce lien est ici fourni par l'existence des parties fonctionnelles associées aux classes dans le plan correspondant. En examinant ces parties fonctionnelles, on sait déterminer si certains objets et certaines classes participent ou non à la fonction. Rappelons que le besoin de circonscire les participants à une fonction est important du point de vue de la maintenance pour avoir connaissance des objets concernés par un changement de spécification fonctionnelle.

3.4.2 Partie fonctionnelle

Dans la section précédente, nous avons examiné la structure des plans fonctionnels qui se

compose essentiellement de parties fonctionnelles associées aux classes du plan de base. Cette section précise la structure d'une partie fonctionnelle.

Une partie fonctionnelle est définie pour un seul plan fonctionnel et une seule classe du plan de base. Par rapport à la description de la classe fournie par le plan de base, une partie fonctionnelle étend cette description en venant ajouter de nouvelles caractéristiques (attributs et méthodes) ayant seulement un sens dans la fonction associée. Une partie fonctionnelle ne doit pas être confondue avec une sous-classe. En effet, à la différence de celle-ci, une partie fonctionnelle n'a pas vocation à introduire de nouveaux objets. Son rôle est purement descriptif. Une partie fonctionnelle a comme propriété essentielle de préserver l'identité des objets de la classe pour laquelle elle est définie tout en réalisant leur enrichissement fonctionnel.

Structure d'une partie fonctionnelle

La structure d'une partie fonctionnelle se résume à un ensemble d'attributs et de méthodes privées et publiques. Les attributs permettent de prendre en compte des données spécifiques à la fonction ainsi que des relations inter-objets locales à celle-ci. Ces attributs ne sont accessibles qu'aux méthodes de la partie fonctionnelle[1]. Les méthodes permettent d'implanter les traitements spécifiques à la fonction. Les méthodes publiques font partie du protocole de l'objet dans la fonction associée. Les autres objets intervenant dans celle-ci y ont accès par envoi de message. Les méthodes privées sont réservées pour des besoins d'implantation de la partie fonctionnelle et ne sont pas disponibles à l'extérieur de l'objet. Nous verrons en section 3.9.2 que ces dernières sont accessibles aux parties fonctionnelles de la classe définie dans les autres plans selon des modalités que nous préciserons.

Dans une partie fonctionnelle, les noms de caractéristiques à définir (attribut et méthode) peuvent être choisis indépendamment des autres parties fonctionnelles de la classe. La présence de caractéristiques de même nom dans des parties fonctionnelles de la même classe est autorisée, chacune possédant alors un sens dans son contexte.

A la figure 3.6, nous donnons la définition de la partie fonctionnelle associée à la classe AndGate pour la fonction d'optimisation.

1. Cet accès limité sera assoupli lorsque nous aborderons les questions d'héritage au sein d'un plan fonctionnelle

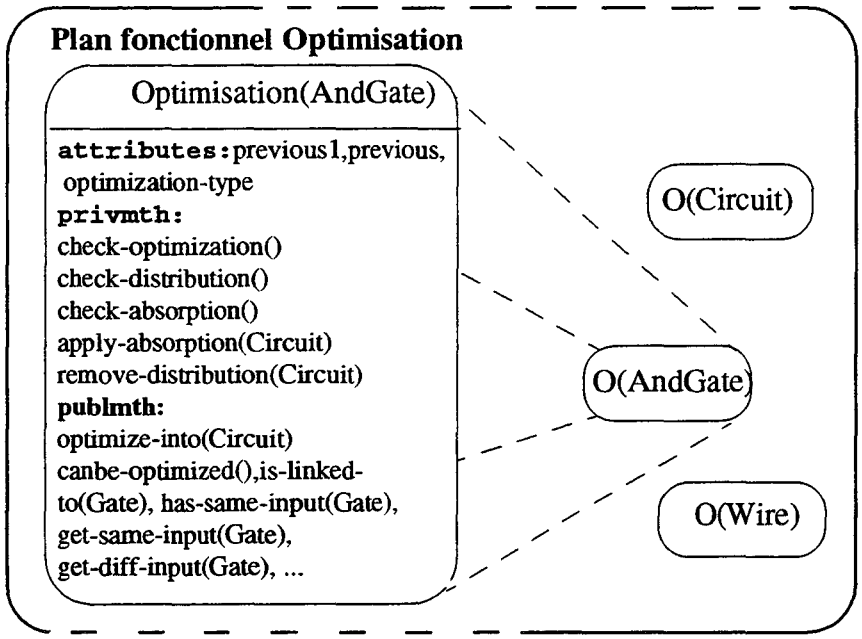


figure 3.6 Structure d'une partie fonctionnelle

Cette figure rend compte de la différence entre la structure d'une partie fonctionnelle et celle d'une classe. On peut aussi noter que l'identification d'une partie fonctionnelle est donnée par deux paramètres, le plan où elle est définie et la classe qu'elle enrichit.

Programmation à l'intérieur d'une partie fonctionnelle

La programmation à l'intérieur d'une partie fonctionnelle reste analogue à celle au sein d'une classe. Cette programmation s'effectue par l'intermédiaire des méthodes. On dispose notamment d'une pseudo-variable représentant l'objet courant. Cette reférence est invariante au travers des fonctions. Elle permet à l'objet de transmettre une référence vers celui-ci lors d'un envoi de messsage.

Les méthodes de la partie fonctionnelle peuvent être combinées entre elles. Une méthode (publique ou privée) peut faire appel à une autre méthode de la partie fonctionnelle (publique ou privée). Dans le corps des méthodes des envois de message à d'autres objets intervenant dans le même fonction peuvent aussi être spécifiés. Les envois de messages permis seront précisés en section 3.6.

Les caractéristiques auxquelles on peut faire référence au sein d'une partie fonctionnelle ne se limitent pas exclusivement aux caractéristiques définies localement par celle-ci. D'autres caractéristiques issues notamment du plan de base mais aussi des parties fonctionnelles associées aux sur-classes y sont également accessibles. La nature et les modalités de ces accès seront précisées dans la prochaine section pour le premier cas et dans la section 3.4.3 pour le second.

Articulation avec la description de base

Une partie fonctionnelle définie pour une classe dans un plan entretient systématiquement une relation d'héritage avec la description de base de celle-ci. Elle permet dans la partie

fonctionnelle:

- d'avoir accès à l'ensemble des caractéristiques (attributs et méthodes) introduites dans la description de base;

- de redéfinir les méthodes de la description de base pour les adapter ou les enrichir de façon spécifique à une fonction.

L'accès obtenu à travers cette relation d'héritage permet dans le corps des méthodes de la partie fonctionelle d'utiliser les attributs et méthodes introduites par la description de base. L'accès aux attributs est permis à la fois en lecture et en écriture. L'accès aux méthodes prend la forme d'une combinaison de méthode entre deux parties d'une même classe. L'ensemble des méthodes (publiques et privées) de la description de base peuvent être appelées dans les méthodes (publiques et privées) de la partie fonctionnelle correspondant à la même classe.

Nous insistons sur le fait que ces accès ne remettent pas en cause l'encapsulation des objets en ce qu'elle concerne des descriptions des mêmes objets.

La redéfinition dans une partie fonctionnelle des méthodes de la description de base permet de leur attribuer une nouvelle implantation spécifique à la fonction. Un telle redéfinition a un caractère local: elle n'affecte pas les autres parties fonctionnelles de la classe et par conséquent les autres fonctions. Une même méthode de la description de base peut être redéfinie localement dans plusieurs parties fonctionnelles. Nous verrons en section 3.8 des exemples de redéfinition multiple d'une même méthode et son intérêt pour prendre en compte des besoins de généricité entre fonctions.

Nous insistons ici sur le fait qu'il ne s'agit pas d'une redéfinition par rapport à une sur-classe. Il s'agit d'une redéfinition pour une fonction d'une méthode appartenant à la description de base de l'objet et factorisée pour les fonctions.

Dans le corps d'une redéfinition locale, les caractéristiques de l'objet spécifique à la fonction sont accessibles tout comme ceux de la description de base. L'appel à la méthode de la description de base qui est masquée est également rendu possible. Cet appel doit passer cependant par l'introduction d'un mécanisme spécifique puisque la redéfinition en interdit l'accès par self message. Le mécanisme en question peut être rapproché du mécanisme de super-message disponible classiquement pour l'héritage entre classes mais s'applique ici entre descriptions (fonctionnelles et de base) d'une même classe.

3.4.3 Héritage local aux plans fonctionnels

Un plan fonctionnel préserve également les relations d'héritage issues du plan de base. Cette préservation de l'héritage fait qu'au sein d'un plan , la partie fonctionnelle enrichissant une classe est héritée localement par ses sous-classes. De cet héritage local à un plan fonctionnel résulte deux traits caractéristiques de l'approche. Premièrement, cela donne lieu à un héritage contextualisé entre les classes au travers de leurs parties fonctionnelles respectives. En second lieu, même si une classe n'est pas explicitement enrichie par une partie fonctionnelle dans le plan, elle l'est implicitement en héritant localement des parties fonctionnelles associées à ses sur-classes. Cet héritage local apporte une solution au problème de structuration d'héritage explicité au cours du premier chapitre en section 3.2.1.

3.4.3.1 Héritage contextualisé entre classes

La contextualisation de l'héritage se produit quand des classes en relation d'héritage dans le plan de base sont conjointement enrichies dans un plan fonctionnel. La relation d'héritage entre ces classes est alors préservée entre leurs parties fonctionnelles respectives associées à ce plan. Il en résulte un héritage des attributs et des méthodes entre les parties fonctionnelles de ces classes qui est local au plan.

Afin de montrer un exemple d'héritage contextualisé, reprenons le plan fonctionnel pour la fonction d'édition déjà schématisé à la figure 3.4 mais en montrant en plus l'héritage contextualisé entre les parties fonctionnelles. La représentation d'une partie de ce plan est donnée à la figure 3.7. Les flêches en pointillés indiquent la contextualisation des relations d'héritage au sein de ce plan fonctionnel. Les pointillés rendent compte que cet héritage est implicitement établi selon la hiérarchie d'héritage déterminée par le plan de base.

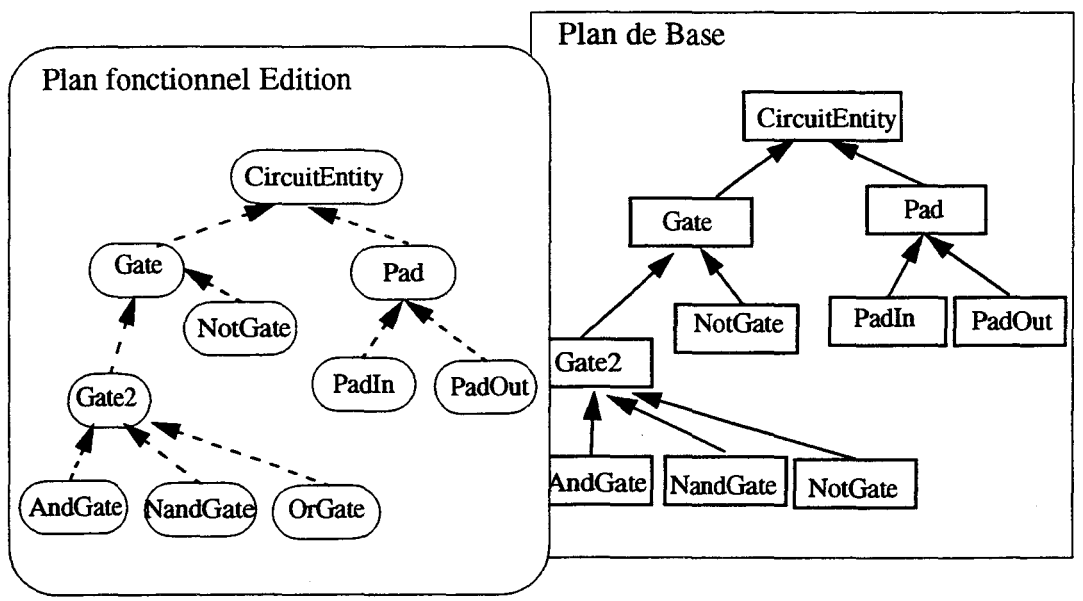


figure 3.7 : Héritage local dans le plan Edition

Comme le montre cette figure, les relations d'héritage entre classes déterminées par le plan de base sont préservées entre les parties fonctionnelles associées. Il en résulte un héritage des attributs et des méthodes entre les parties fonctionnelles de ces classes dans le plan. Ainsi, dans cet exemple, les parties fonctionnelles des classes `AndGate` et `OrGate` héritent des attributs et des méthodes introduits par la partie fonctionnelle de leur sur-classe Gate2 mais aussi transitivement de ceux définis par la partie fonctionnelle relative à la classe `CircuitEntity`.

Cet héritage offre pour les parties fonctionnelles les mêmes capacités de spécialisation entre classes que pour la description de base. Ainsi, une partie fonctionnelle d'une classe peut spécialiser la partie fonctionnelle associée à la sur-classe pour le même plan de façon classique:

- en déclarant de nouveaux attributs;
- en définissant de nouvelles méthodes publiques et privées;
- en redéfinissant les méthodes héritées localement que celles-ci soient publiques ou privées.

Concernant ce dernier point, il convient d'insister sur le caractère local de la redéfinition.

Une méthode d'une partie fonctionnelle associée à une classe peut seulement être redéfinie dans les parties fonctionnelles des sous-classes correspondant au même plan. Hormis son caractère local, la redéfinition de méthodes au sein du plan reste classique. Ajoutons que la modalité d'accès publique ou privée d'une méthode ne peut être modifiée lors de sa redéfinition.

Afin de montrer un exemple de spécialisation, la figure suivante donne une vue détaillée des parties fonctionnelles associées aux classes `InputPin` et `OutputPin` et à leur sur-classe commune Pin pour le plan considéré précédemment. La partie fonctionnelle associée à la classe Pin détient les caractéristiques (attributs et méthodes) communes aux sous-classes mais seulement pertinentes pour la fonction. Celles-ci sont héritées par les parties fonctionnelles des sous-classes dans le même plan. Ces parties fonctionnelles viennent ajouter pour la fonction des caractéristiques propres à leur classe respective. Dans le cas présent, celles-ci ne comportent qu'une redéfinition locale de la méthode `propagate-tick` de propagation d'unité de temps. Cette méthode étant abstraite au niveau de la partie fonctionnelle associée à `Pin`, elle est concrétisée dans `InputPin` pour propager une unité de temps à la porte et dans `OutputPin` pour propapager une unité de temps à un ensemble d'équipotentielles.

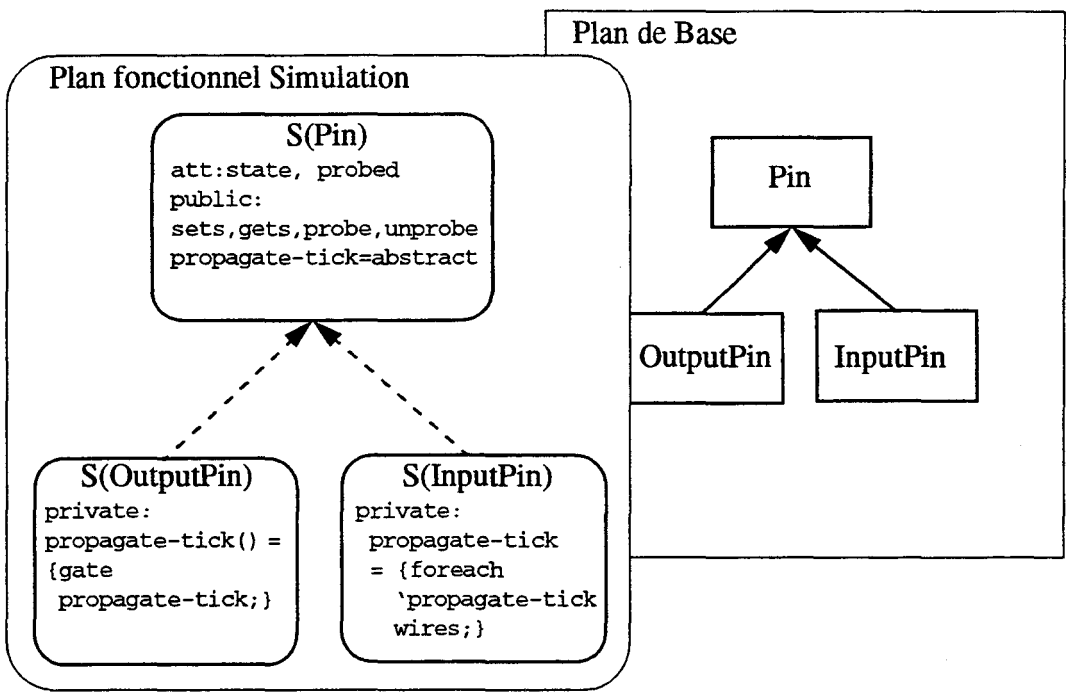


figure 3.8 : Héritage local pour les broches dans le plan Simulation

Lorsqu'une partie fonctionnelle hérite d'une autre partie fonctionnelle, les caractéristiques définies ou héritées par cette dernière peuvent être exploitées pour l'implantation de ses méthodes. De ce point de vue, les possibilités offertes restent classiques. Ces méthodes peuvent ainsi directement accéder aux attributs hérités de la même manière que des attributs locaux. Elles peuvent également faire appel aux méthodes héritées. Dans ce cas, l'appel à ces méthodes se fait de la même façon que pour les méthodes locales à la partie fonctionnelle. Quand une méthode héritée est redéfinie au niveau d'une partie fonctionnelle, son ancienne définition reste accessible de manière interne par super-message. Ceci permet l'affinement incrémental des méthodes entre parties fonctionnelles.

Cette capacité de contextualisation de la hiérarchie d'héritage trouve tout son sens dans la multiplicité liée aux fonctions. Il devient possible de prendre en compte des besoins de

factorisation, de généricité et d'abstraction entre classes du plan de base spécifiques à chaque fonction.

Montrons ceci à travers la contextualisation multiple de la hiérarchie des portes logiques.

Le plan fonctionnel lié à l'édition permet d'illustrer la contextualisation de cette hiérarchie pour des besoins de factorisation spécifiques à la fonction correspondante. Pour les besoins de cette fonction, toute porte doit possèder un attribut représentant sa position graphique et une méthode de déplacement pour modifier cette position dont l'implantation est identique pour chaque type de porte. Ces deux caractéristiques réclament par conséquent d'être factorisé entre tous les types de porte. Dans le cas présent, cette factorisation est réalisée en spécifiant ces deux caractéristiques (`location` et `move-to`) dans la partie fonctionnelle de la classe `Gate` définie dans le plan d'édition schématique. La figure 3.9 montre cette factorisation dans le plan en question.

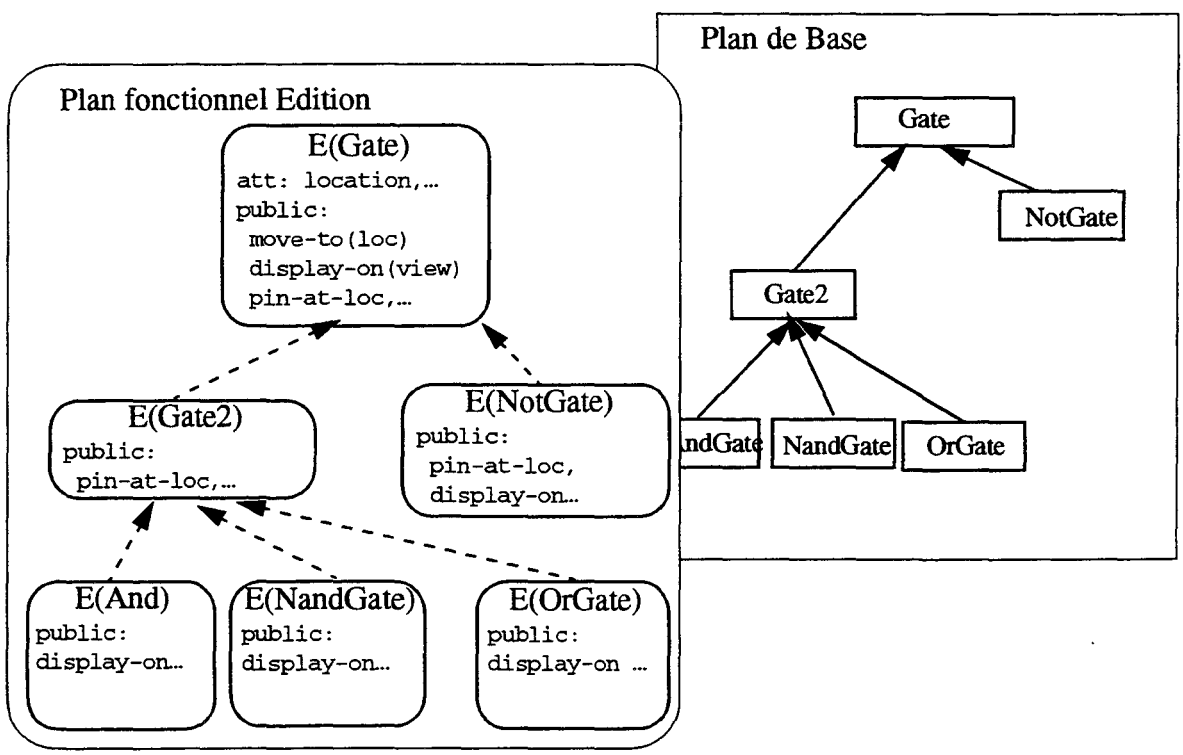


figure 3.9 : Factorisation locale au plan Edition

Sur cette figure, on peut voir que la partie fonctionnelle `E(Gate)` détermine les caractéristiques partagées entre classes dans la fonction tandis que chaque partie fonctionnelle associée aux sous-classes `E(And)`, `E(NandGate)` et `E(Or)` détermine seulement les caractéristiques des objets spécifiques à la fonction mais qui leur sont propres et non partagée par les autres classes.

Le plan fonctionnel lié à la simulation permet d'illustrer la contextualisation de cette même hiérarchie pour programmer des méthodes génériques. Rappelons qu'une méthode générique est une méthode faisant appel à d'autres méthodes et dont la définition introduite dans une sur-classe n'est pas redéfinie explicitement dans les sous-classes mais l'est implicitement par rédéfinition des méthodes auxquelles elle fait référence.

Dans ce plan, la partie fonctionnelle de la sur-classe `Gate` définit la méthode `propagate-tick` de manière générique. Cette méthode, non redéfinie ailleurs, fait en effet référence aux méthodes d'évaluation `evaluate` et de calcul du délai `delay` pour laquelle des définitions plus affinées sont fournies dans les parties fonctionnelles des sous-classes `AndGate`, `OrGate`, `NandGate`, et `NotGate`. La figure 3.10 schématise ce cas de généricité locale au plan de simulation.

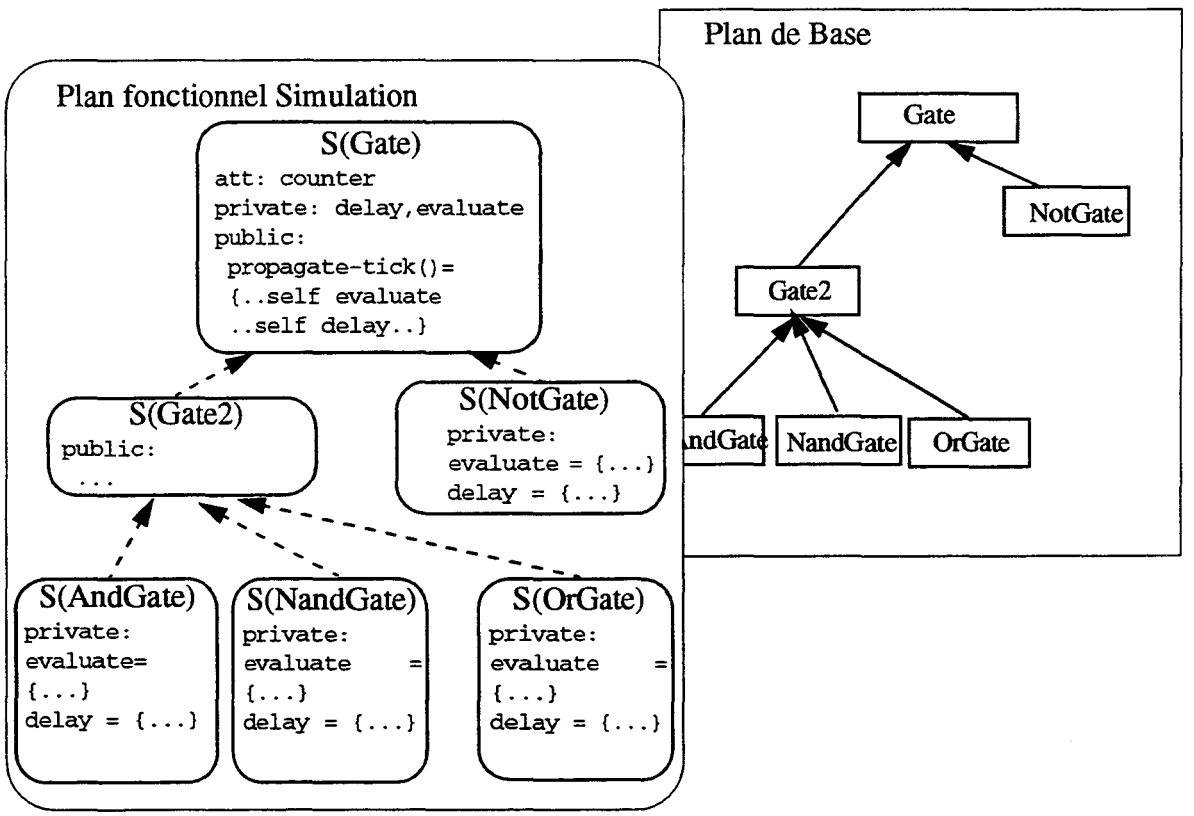


figure 3.10 : Généricité locale au plan Simulation

Dans cette figure, la méthode `propagate-tick` est implicitement redéfinie au niveau des parties fonctionnelles du fait de la redéfinition de la méthode `evaluate` et `delay`.

Pour terminer, le plan fonctionnel lié à l'optimisation permet d'illustrer la contextualisation de la hiérarchie des portes pour supporter des besoins d'abstraction entre classes spécifiques à une fonction. Il s'agit ici d'exprimer des méthodes dont la spécification est commune à plusieurs classes mais dont l'implantation diffère pour chaque classe.

Pour la fonction de normalisation, toutes les portes doivent savoir répondre au message `optimize` et partagent donc la même spécification de méthode correspondante. En revanche, l'action associée dépend du type de porte concernée (porte Et, porte Ou, ...). La figure 3.11 montre la partie du plan fonctionnel correspondant.

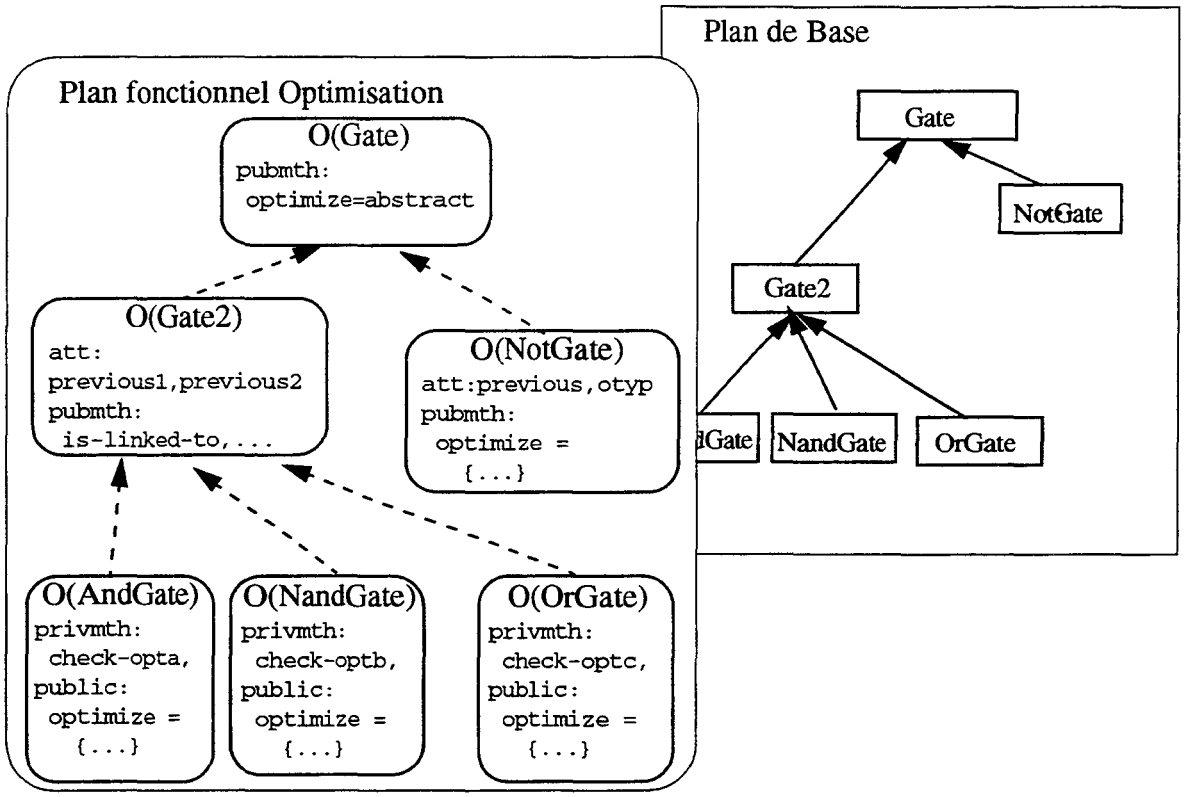


figure 3.11 : Abstraction locale au plan d'Optimisation

Sur cette figure, nous observons que la méthode `optimize` est seulement spécifiée dans la partie fonctionnelle associée à la classe Gate, sans lui donner d'implantation (méthode abstraite). Elle est redéfinie dans les parties fonctionnelles des sous-classes `AndGate`, `NandGate`, `OrGate` et `NotGate` afin de recevoir une implantation propre à celles-ci.

3.4.3.2 Enrichissement implicite des classes

Dans la sous-section précédente, nous avons examiné une première caractéristique liée à la préservation des relations d'héritage au sein du plan fonctionnel lorsque les classes sont conjointement enrichie dans le plan. L'enrichissement implicite des classes au sein d'un plan fonctionnel est la seconde caractéristique liée à cette préservation. Cet enrichissement se produit lorsqu'une classe enrichie explicitement dans un plan fonctionnel possède des sous-classes qui n'ont pas de partie fonctionnelle associée dans ce dernier. Dans ces conditions, la préservation de l'héritage au sein du plan fait que les sous-classes en question héritent localement de la partie fonctionnelle de la sur-classe, conduisant ainsi à l'enrichissement implicite de celles-ci.

Montrons un exemple de cet enrichissement implicite en considérant à nouveau le plan Documentation vue précédemment. La partie de ce plan qui nous intéresse dans le cas présent concerne l'enrichissement des classes représentant les ports d'entrée et de sortie (PadIn et PadOut). Cette partie est représentée par la figure 3.12.

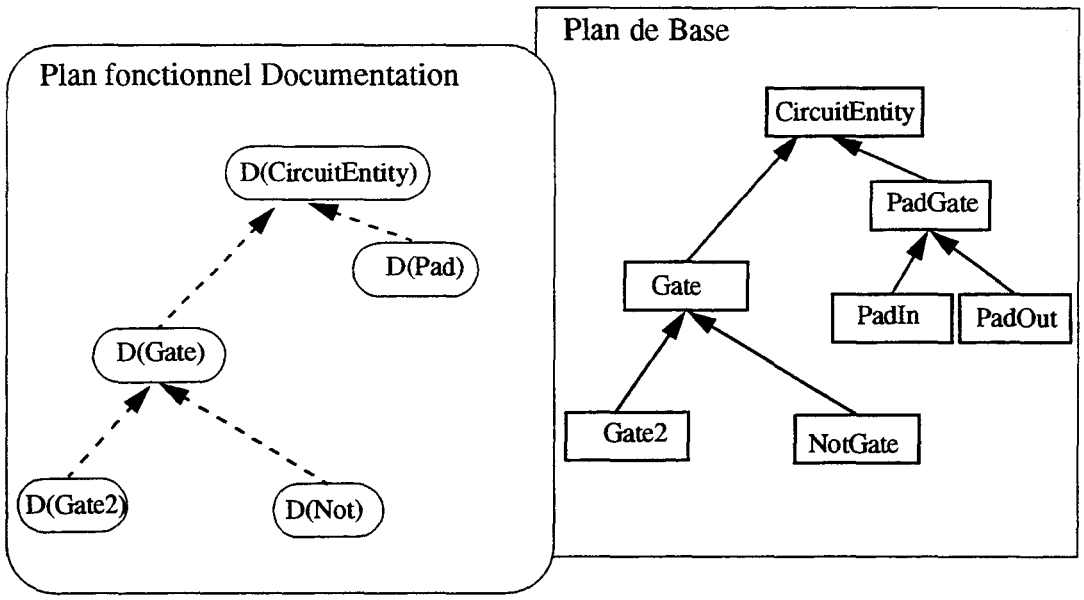


figure 3.12 : Enrichissement implicite des classes PadIn et PadOut

L'enrichisssement implicite apparait ici pour les classes `PadIn` et `PadOut`. Bien que ne possédant pas de partie fonctionnelle au sein de ce plan, ces deux classes héritent pour leurs objets de la partie fonctionnelle associée à la surclasse `Pad`.

Il est intéressant de noter que la sur-classe enrichie explicitement peut être abstraite. Ici, `Pad` est abstraite et n'est pas succeptible d'être instanciée. La partie fonctionnelle associée à cette classe doit par contre être complètement déterminée. Ceci est ici requis dans la mesure où cette partie fonctionnelle est héritée par les sous-classes qui sont elles concrètes. Si ce n'était pas le cas, les objets des sous-classes enrichis implicitement disposerait dans ces conditions d'une caractérisation incomplète pour la fonction.

Une autre caractéristique intéressante de l'enrichissement implicite est que cela permet d'avoir un certain degré de liberté vis à vis de la hiérarchie d'héritage du plan de base. En mettant à profit le fait qu'une partie fonctionnelle héritée par une classe enrichie implicitement est également héritée par ses sous-classes, il devient possible d'exprimer l'héritage local au plan fonctionnel sans être obligé de respecter de façon totalement isomorphe la hiérarchie du plan de base.

Pour illustrer ce point, le cas du plan normalisation peut être pris en exemple. Au niveau de ce plan, la classe `Gate2` ne possède pas, contrairement à sa surclasse `Gate` et ses sous-classes `NotGate`, `OrGate`, `AndGate` et `NandGate`, de partie fonctionnelle associée. Pour ces sous-classes, les attributs et méthodes hérités localement au niveau de leur partie fonctionnelle sont par conséquent ceux de la sur-classe `Gate`. L'héritage local ainsi obtenu peut être interprété comme à la figure 3.13.

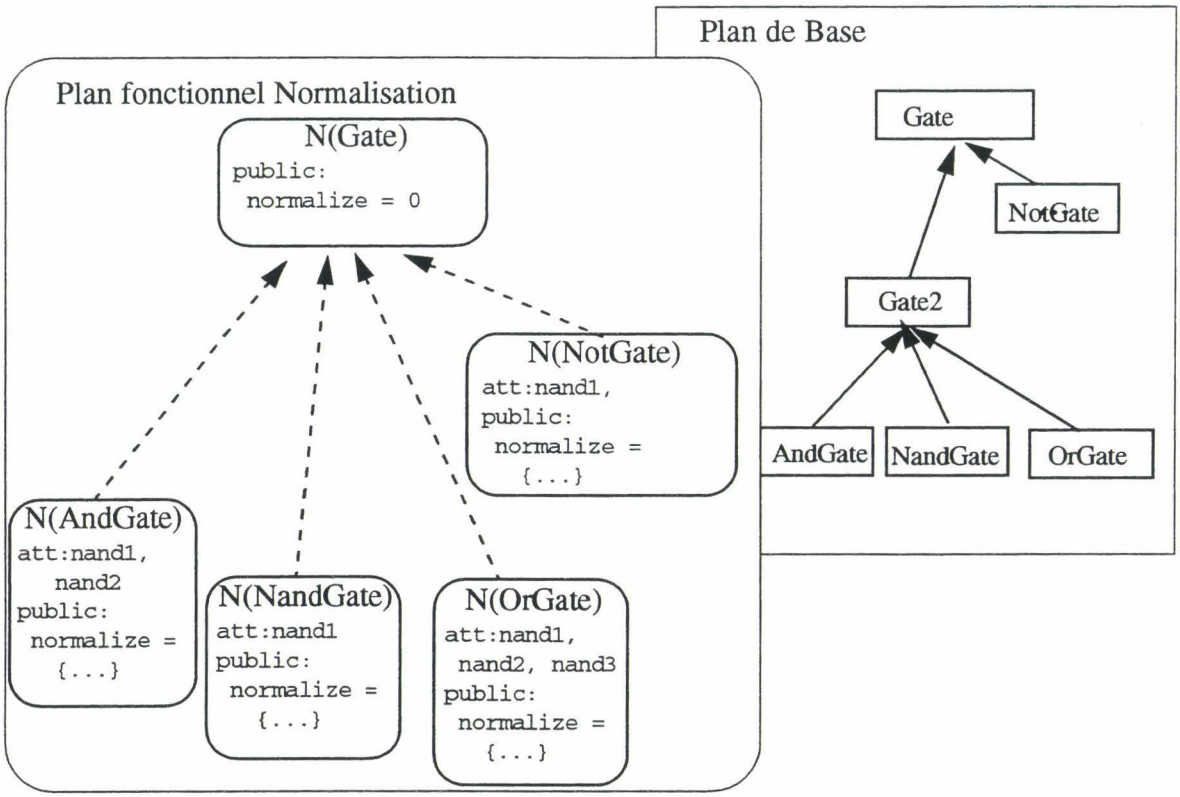


figure 3.13 : Enrichissement implicite des sous-classes de Gate

Ici, en raison de l'absence de partie fonctionnelle associée à la classe `Gate2`, les parties fonctionnelles associées à ces sous-classes sont présentées comme si elles héritaient directement de la classe Gate. Soulignons que l'héritage obtenu continue de respecter la hiérarchie déterminée par le plan de base.

3.4.4 Objets résultant des plans de descriptions

La description complète des objets d'une classe est déterminée par sa description de base et l'ensemble des parties fonctionnelles associées à celle-ci dans les plans. Pour une classe munie de liens d'héritage, la description de ses objets est déterminée en faisant intervenir en plus la description de base de ses sur-classes ainsi que que leurs parties fonctionnelles. La figure 3.14 représente la description obtenue pour les objets des classes Circuit, AndGate et Wire. Les traits horizontaux expriment le fait que les plans correspondants déterminent cette description.

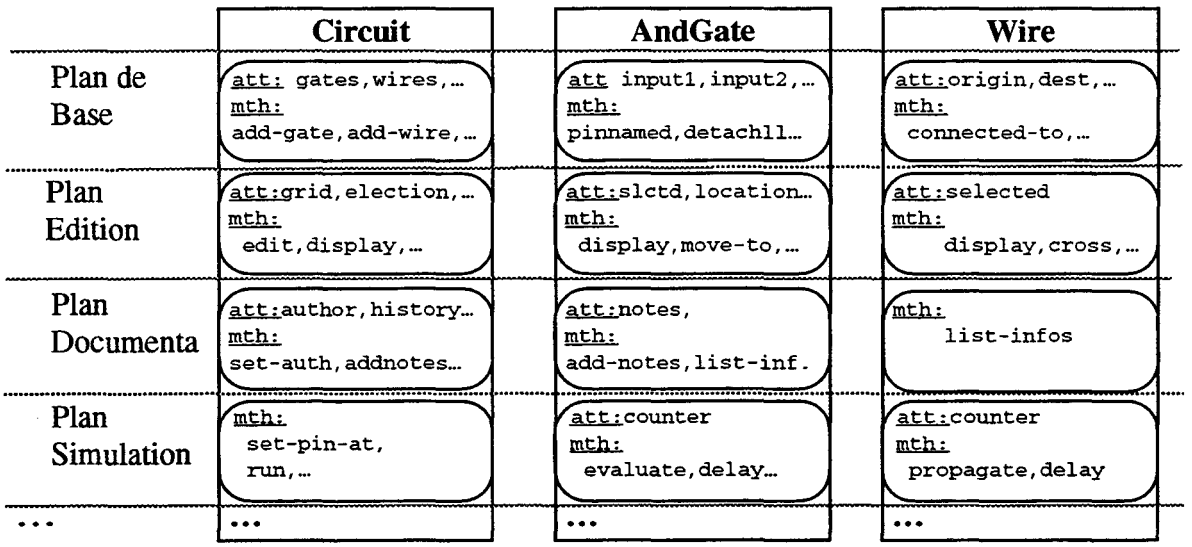


figure 3.14 Structuration transversale aux objets

Sous l'angle des objets, cette figure met en évidence que chaque objet détient toutes les caractéristiques nécessaires à l'ensemble des fonctions où ils participent. Elle montre également que la description de base et les parties fonctionnelles déterminent chacune un sous-ensemble des attributs et des méthodes.

Sous l'angle des plans, on s'aperçoit que chaque dimension fonctionnelle est distribuée au travers des objets.

Cette description selon les fonctions est obtenue de manière systématique pour l'ensemble des objets du référentiel quelque soit leur classe.

3.5 Héritage modulaire

Les contextes ont pour but de déterminer toute la description des objets nécessaire à l'exécution d'une fonction. Un *contexte* est l'association d'un plan fonctionnel et du plan de base. Ils viennent compléter les plans qui ne font que détenir et systématiser les descriptions. Un contexte peut être considéré comme un véritable programme. Les objets caractérisés selon un contexte sont notamment capables de fonctionner et de coopérer par envoi de message. Dans cette section, nous précisons de quelle façon la description des objets pour un contexte est établie à partir des plans. Ceci va nous amener à présenter un héritage modulaire.

En section 3, nous avons vu que le plan de base introduit, pour tous les objets, la partie de leur description partagée entre tous les contextes tandis que chaque plan fonctionnel détermine pour tous les objets intervenant dans un contexte, la partie de leur description spécifique à ce dernier. A partir de cette répartition des descriptions, la description des objets dans un contexte s'obtient en considérant exclusivement celles fournies par le plan de base et le plan fonctionnel correspondant.

Montrons sur un exemple la description des objets déterminée pour un des contextes de notre système, celui lié à l'édition. Les classes choisies pour cet exemple sont `Circuit` et `NotGate`.

La partie du plan de base et la partie des plans fonctionnels destinées à l'édition et à la simulation correspondant à ces classes sont représentées à la figure 3.15.

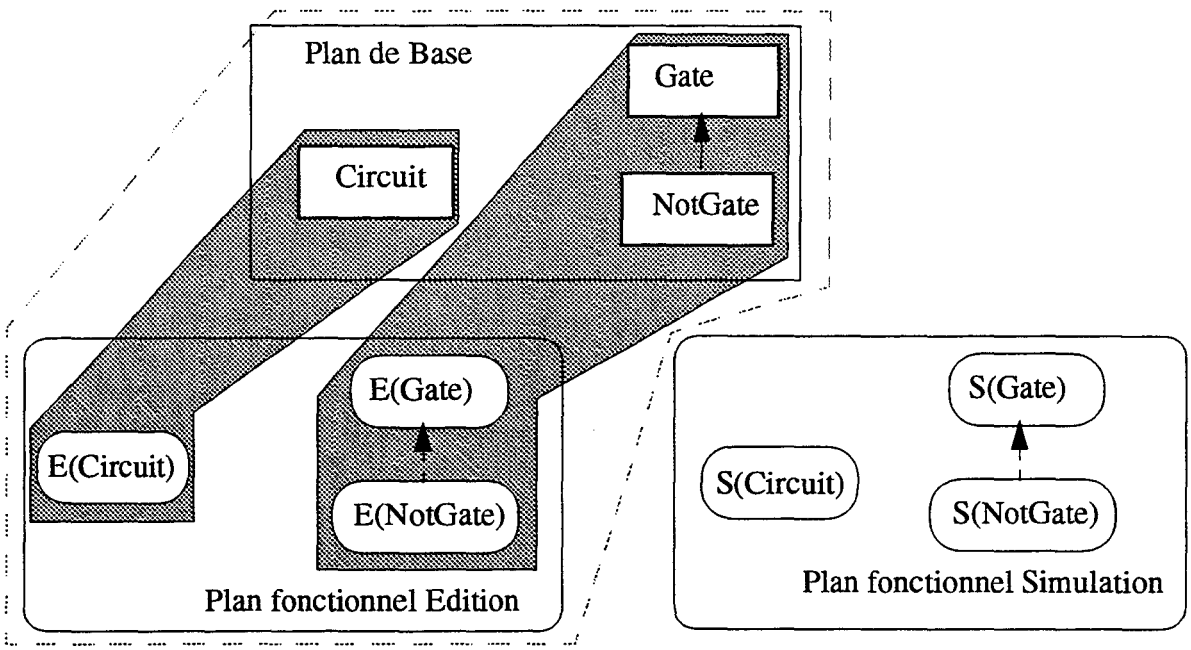


figure 3.15 Descriptions des objets Circuit et NotGate retenues pour la fonction d'édition

Sur cette figure, sont indiqués en pointillés les plans considérés pour obtenir la description des objets dans le contexte d'édition. Les zones grisées délimitent les unités de descriptions d'objets pour un contexte. L'analyse de la zone relative à `Circuit` montre que, pour une classe sans relation d'héritage, cette description est déterminée à partir de sa description de base et de sa partie fonctionnelle associée dans le plan d'édition. Pour une classe héritant d'une autre classe, la zone relative à `NotGate` montre que cette description est déterminée en faisant également intervenir la description de base et la partie fonctionnelle associée aux sur-classes[1]. Il est en de même pour toute autre classe dont les objets participent à ce contexte. De manière générale, la description des objets relativement au même contexte est obtenue en considérant les parties fonctionnelles du même plan.

Pour le contexte de simulation, la description déterminée pour les objets des mêmes classes Circuit et NotGate sera obtenue en considérant leur description de base et leur partie fonctionnelle dans le plan associé. Soulignons que, quelque soit le contexte considéré, la description correspondante des objets inclut toujours leur description de base ce qui correspond à leur partage par toutes les fonction.

A ce stade, nous avons seulement indiqué le sous-ensemble d'entités qui entre dans la description des objets relatives à un contexte sans préciser leur exploitation. La structuration en plans suggère une stratégie d'héritage modulaire rendant compte des contextes. Cette stratégie consiste à privilégier les descriptions des objets déterminées par le plan fonctionnel.

---

1. Bien que la figure ne la fasse pas apparaitre pour une raison de place, la description de base de la sur-classe indirecte CircuitEntity et la partie fonctionnelle associée dans le plan d'édition interviennent également dans la description des objets NotGate relative à ce contexte.

Appliquée à l'exemple des objets porte Non considéré précédemment, cette stratégie revient à examiner dans l'ordre : la partie fonctionnelle de la classe `NotGate`; la partie fonctionnelle asssociée à `Gate`; la description de base de `NotGate` et enfin la description de base de `Gate`. Cette stratégie est représentée à la figure 3.17. Les flêches grisés indiquent l'héritage systématique entre une partie fonctionnelle et la description de base de chaque classe.

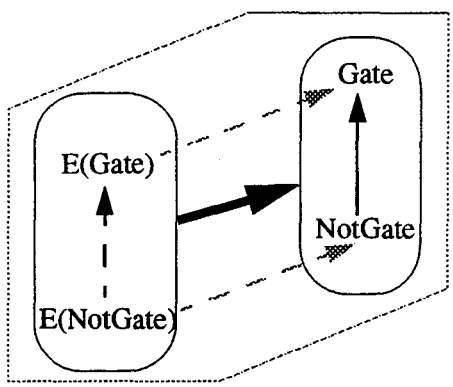


figure 3.16

Cet exemple fait apparaitre une exploitation modulaire des descriptions. Plus généralement, cette stratégie amène à dissocier pour chaque contexte et pour chaque classe deux modules:

- le *module de base* incluant les descriptions de base;
- le *module spécifique* incluant les parties fonctionnelles.

Le module de base et le module spécifique obtenus pour la classe `NotGate` considérée dans le contexte d'édition schématique sont présentés à la figure 3.17. D'après cette figure, la stratégie revient au parcours dans l'ordre, du module spécifique puis, si la caractéristique n'est pas trouvée au sein de ce premier module, du module de base. Au sein d'un module, l'héritage reste classique.

Ce parcours ordonné de modules, qui détermine la description des objets pour le contexte, garantit:

- qu'un objet dispose des caractéristiques définies par le module spécifique ainsi que celles du module de base
- la règle du plus affinée relativement au contexte: pour une caractéristique donnée, c'est toujours la définition la plus affinée pour le contexte qui est prise en compte.

Ceci permet d'expliciter un héritage entre modules. Sur le schéma précédent, cette relation d'héritage entre les deux modules est symbolisée par une flêche de plus grande taille.

Montrons les principales caractéristiques de cette héritage modulaire à travers différents cas de figure primitifs. Chaque cas présenté est donné pour un objet d'une classe `B` considérée dans le contexte `C1` (figure 3.17). La structure d'héritage est constituée de 2 modules correspondants respectivement au plan de base (`A`, `B`) et au plan fonctionnel C1 (`C1(A)`, `C1(B)`). Nous considérons uniquement le cas des méthodes.

1) Héritage du plan de base: Cet héritage est illustré à la figure 3.18. La méthode m n'étant

pas redéfinie localement, la définition choisie pour un message correspondant envoyé à un objet B dans le contexte C1 est donc celle de B, celle-ci étant la plus spécifique dans le plan de base.

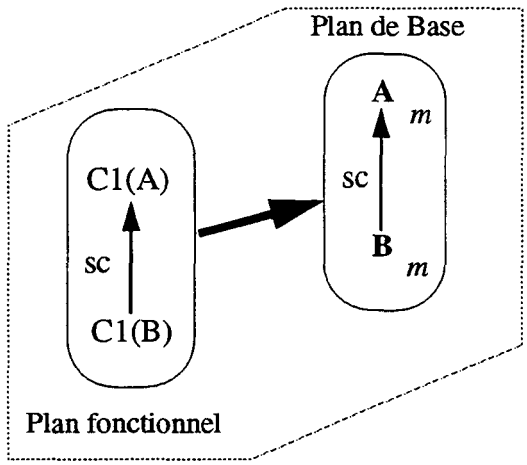


figure 3.17

2) Référence générique au plan de base. Dans une méthode d'une partie fonctionnelle , il est possible de faire référence aux méthodes de la description de base (cf 3.4.2). Ce qui est illustré ci-dessous pour la méthode mp de C1(A) qui fait appel à m par self-message.

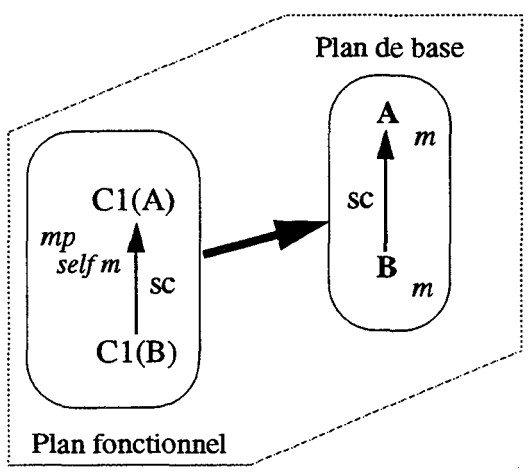


figure 3.18

Pour cette méthode mp, la référence à m résultera en l'application de la méthode m telle que définie dans B. Il convient d'insister sur le fait que les caractéristiques du module de base auxquelles une classe du module spécifique peut faire référence sont déterminées par la relation d'héritage qu'elle entretient avec sa classe de base et non en fonction du parcours des modules. Dans le cas précédent, cela signifie par exemple que la méthode mp de C1(A) ne peut faire appel aux méthodes locales à B mais seulement celles définies par A.

3) Redéfinition dans un contexte d'une méthode de base. Toute méthode du plan de base peut être redéfinie localement dans une partie fonctionnelle (cf 3.4.2). Selon la stratégie adoptée, c'est cette redéfinition qui est choisie pour le contexte même si il existe une définition de cette méthode dans la description d'une sous-classe. Soit une méthode m introduite par A, et deux redéfinitions de celles-ci, l'une au niveau de B et l'autre au niveau de C1(A) comme indiqué à la figure 3.19.

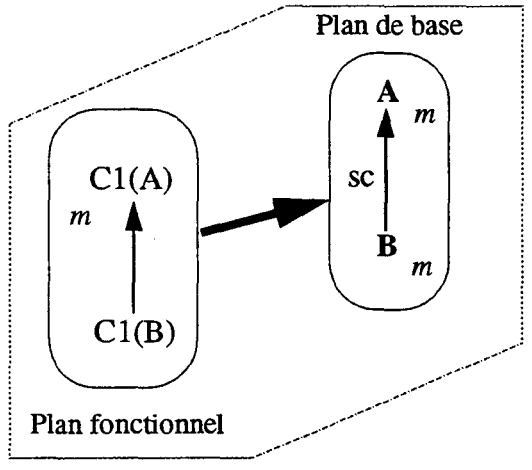


figure 3.19

Ici, la stratégie donnant la priorité au contexte, celle-ci conduit à choisir la définition issue de `C1(A)` pour un envoi de message m à `B` dans C1. Cette définition masque toutes les définitions de `m` dans le plan de base.

4) Référence masquée au plan de base. Pour une méthode redéfinie localement, la méthode du plan de base qui est masquée peut être appelée à l'aide d'une opération spécifique (designée par `base` ici). La figure suivante présente cet appel pour la méthode `m`.

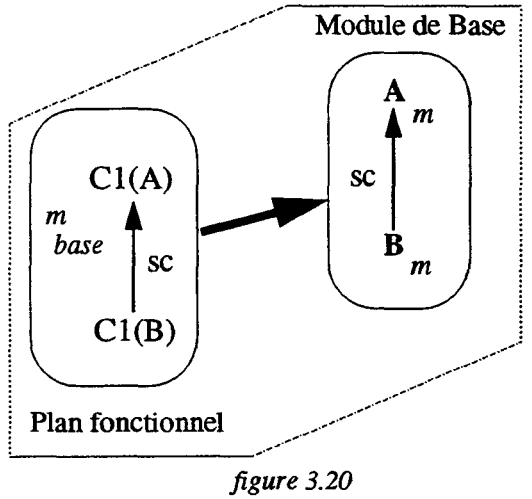


figure 3.20

Dans ce cas, la méthode masquée, activée par l'utilisation de `base` dans la redéfinition de `m` est la plus affinée pour `B` dans le plan de base selon la stratégie adoptée. Il s'agit donc de la méthode `m` appartenant à la description de base de `B` ce qui permet de bénéficier de l'affinement de m au niveau du plan de base. Ceci rend compte de la différence avec un super-message où la référence à la méthode masquée est figée.

Précisons que . Ainsi, si nous modifions l'exemple précédent en ajoutant une redéfinition locale de m au niveau de C1(B), l'emploi d'une opération base dans cette redéfinition ne prend pas en compte la redéfinition locale de m introduite dans C1(A). Cette redéfinition peut néanmoins être activé en exprimant un super-message avant ou après l'opération base.

5) Référence générique par rapport aux plans fonctionnels. Une méthode du plan de base

peut se décomposer en appelant d'autres méthodes. La possibilité de redéfinir localement les méthodes appelées dans les parties fonctionnelles provoque l'affinement implicite de la méthode appelante. Un exemple de ce cas de figure est donné par la figure 3.21 pour les méthodes `m` et `mp`.

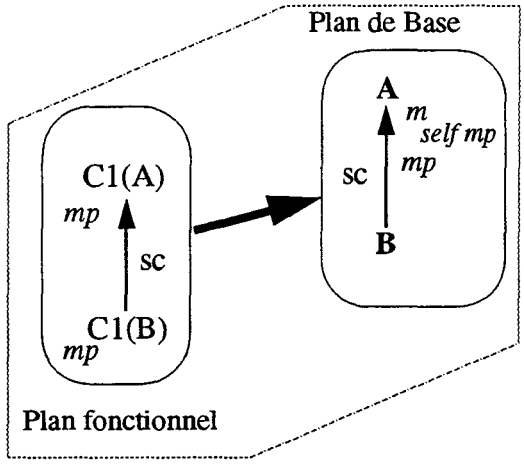


figure 3.21

Dans le cas présent, la méthode `m` du plan de base est décomposée en une méthode `mp` qui est redéfinie localement dans les parties fonctionnelles `C1(A)` et `C1(B)`. Lors de l'appel à `mp` dans `m`, c'est la définition de `mp` détenue par `C1(B)` qui est appliquée, celle-ci étant la plus affinée pour le contexte. Il en résulte un affinement implicite de la méthode `m` pour le contexte sans que celle-ci soit effectivement redéfinie.

Des exemples d'application des cas 3, 4 et 5 qui constituent les principales originalités de cet héritage modulaire seront donnés dans la section 3.8.

3.6 Communication entre objets au sein d'un contexte

La communication entre objets prend toujours place dans un contexte. Lorsqu'il intervient dans un contexte, un objet peut communiquer par envoi de message avec tous les objets qu'il référence dans celui-ci. Les messages qu'il peut alors envoyer à ces objets sont basés sur leur description dans le même contexte: ils correspondent aux méthodes publiques de ces objets telles que définies par le plan fonctionnel associé et le plan de base. Autrement dit, un objet ne perçoit les autres objets qu'au sein du même contexte et n'en a qu'une vision partielle. Il ne connait pas notamment les autres contextes auxquels ces objets participent.

A la figure suivante, sont données la description d'un objet de la classe `Circuit` et celle d'un objet de la classe `NotGate` pour le contexte d'édition considéré précédemment. Ces descriptions sont obtenues en composant pour chaque type de caractéristiques celles de la description de base et celles de la partie fonctionnelle correspondant à leur classe (dissocié par un rectangle grisé) dans le plan correspondant. La partie en pointillée située au bas de chaque description indique la détention par les objets d'autres caractéristiques n'intervenant pas pour ce contexte.

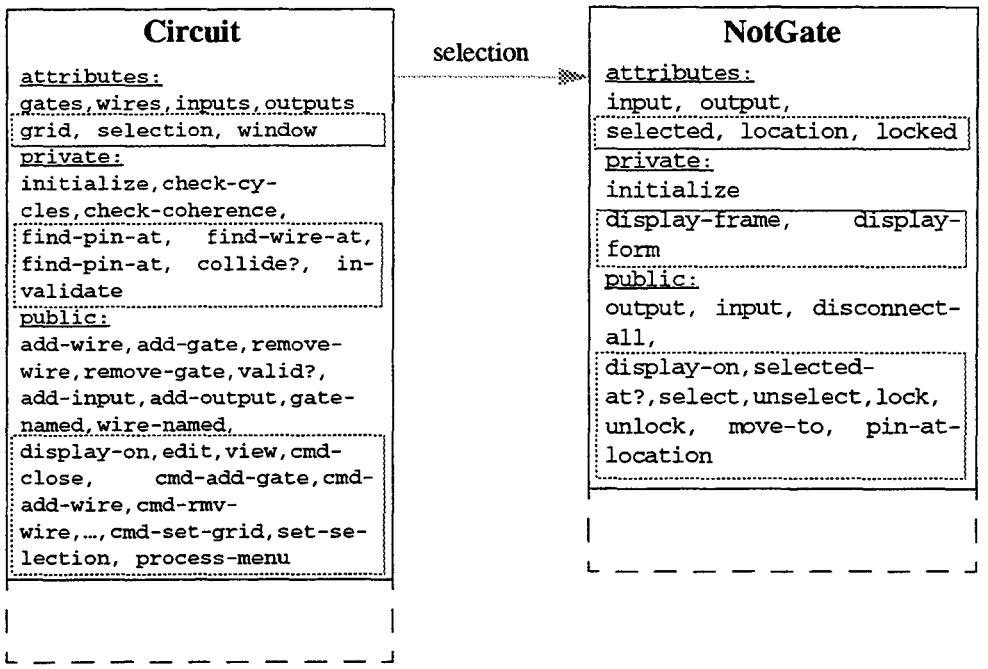


figure 3.22 Description d'un objet Circuit et d'un objet porte Non pour le contexte d'édition

Ici, les méthodes publiques de chacune de ces descriptions sont celles qui sont accessibles par envoi de message aux autres objets intervenant dans le contexte d'édition. Par exemple, en supposant que l'objet circuit possède une référence vers l'objet porte Non dans ce contexte comme représenté à la figure 3.22, les messages pouvant être envoyés à travers cette référence sont par exemple `output`, `input`, ...ainsi que les messages `display-on`, `selected-at?`, `selected`, ...correspondant respectivement aux méthodes publiques de la description de base et de la partie fonctionnelle.

Dans la description d'un objet déterminée pour un contexte, les envois de messages aux autres objets s'effectuent selon les règles suivantes:

- Dans les méthodes publiques et privées de la description de base, les seules méthodes publiques des autres objets pouvant être activées sont celles issues de leur description de base.

- Dans les méthodes publiques et privées d'une partie fonctionnelle (y compris dans les redéfinitions locales), l'ensemble des méthodes publiques déterminées pour les autres objets dans le contexte sont accessibles par envoi de messages.

Le polymorphisme qu'induit classiquement l'envoi de message est conservé pour les contextes. Dans un contexte, la réponse à un message dépend avant tout de l'objet receveur. L'héritage local fournit implicitement un support pour ce polymorphisme puisque toute méthode d'une partie fonctionnelle associée à une classe peut être redéfinie dans celles des sous-classes.

L'intérêt de cette préservation du polymorphisme est de permettre la définition de méthodes génériques par rapport aux autres objets intervenant dans le contexte. Un exemple de méthode générique exploitant ce polymorphisme dans le contexte d'édition est la méthode `display` des circuits qui réalise leur affichage. Cette méthode est implantée de façon à propager le message

`display` aux objets portes du circuit sans tenir compte de leur identité plus fine en tant que porte ET, porte OU, porte NON..., chacune s'affichant évidemment de façon différente.

Dans la section précédente, nous avons insisté sur le fait que la description déterminée pour un objet n'est pas identique d'un contexte à l'autre. L'ensemble des messages qu'il est possible d'envoyer à un même objet varie par conséquent suivant les contextes. Cette différence d'accès est montrée à la figure 3.23 pour l'objet porte Non considéré précédemment et présenté ici selon un autre contexte, celui de simulation. Cette figure donne également la description d'un objet InputPin[1] qui détient une référence vers cet objet à travers son attribut `owner` et l'utilise pour communiquer avec lui dans ce même contexte, notamment lui envoyer le message `propagate-tick` de prise en compte d'une unité de temps.

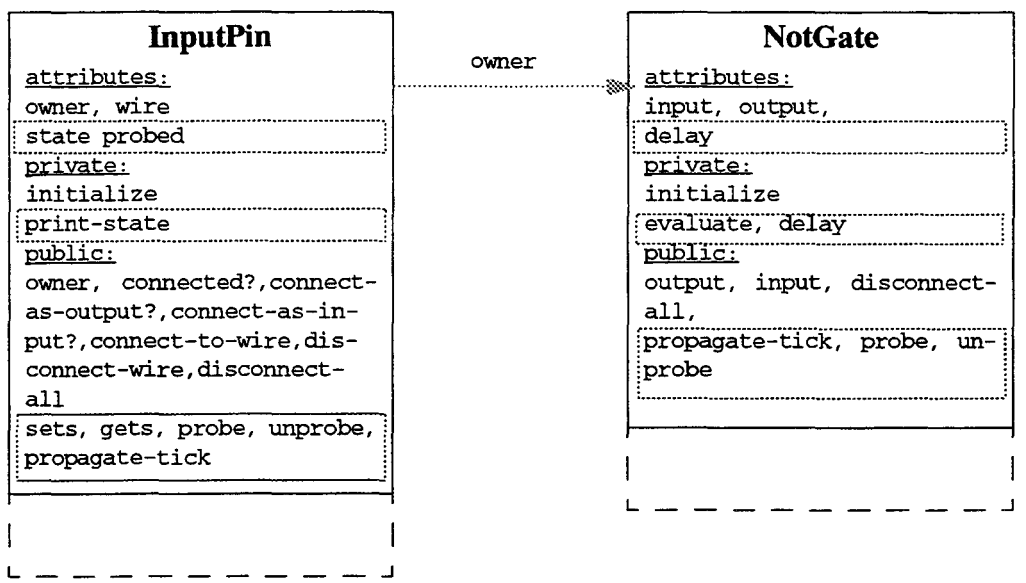


figure 3.23 Description d'un objet InputPin et d'un objet porte Non pour le contexte de simulation

En comparant cette figure avec la précédente, deux constats peuvent être faits.

Premièrement, l'objet porte Non présente pour ce contexte un protocole différent de celui déterminé pour l'édition. Les deux objets Circuit et Entrée communiquant avec cet objet dans deux contextes différents en ont par conséquent chacun une perception différente. Il en est de même pour un objet communiquant avec le même objet dans deux contextes différents.

Deuxièmement, nous pouvons constater que les méthodes de la description de base font partie du protocole de l'objet porte Non dans chacun des contextes. Le même message peut donc être envoyé au même objet dans des contextes différents. Dans ce cas, la réponse de l'objet peut différer suivant le contexte puisque des redéfinitions locales peuvent exister pour la méthode de la description de base correspondante. Supposons par exemple qu'il existe une redéfinition locale de la méthode `disconnect-all` dans le contexte d'édition uniquement, l'envoi du message correspondant à l'objet porte Non donnera lieu à une réaction différente de celui-ci dans le contexte d'édition et dans le contexte de simulation.

1. La raison pour laquelle nous n'avons pas repris un objet circuit vient du fait qu'un tel objet ne communique pas avec ses portes dans le contexte de simulation. Le cas où un objet communique avec le même objet dans plusieurs contextes sera considéré dans la section suivante.

De cette caractéristique découle une nouvelle forme de polymorphisme d'envoi de message basée sur les contextes. Ce polymorphisme est disponible uniquement sur les méthodes du plan de base. Il convient d'insister sur le fait que ce dernier ne remet pas en cause le polymorphisme classique et les propriétés induites, de généricité notamment. Dans des contextes différents, le même message lié à une méthode de la description de base peut être envoyé à des objets différents. Dans ce cas, la réponse dépend de l'objet mais aussi du contexte.

L'intéret de ce polymorphisme est de permettre l'écriture de code générique par rapport aux fonctions où interviennent les objets. Un exemple exploitant ce polymorphisme sera développé en 3.8 pour mettre en oeuvre un parcours de la structure d'un circuit, générique par rapport à plusieurs fonctions.

3.7 Contextualisation de graphes d'objets

Dans la section consacrée à la présentation du plan de base (3.3), nous avons vu que les descriptions de base permettent de factoriser un graphe d'objets entre les fonctions. Dans cette section, nous montrons qu'il est possible d'exploiter ce graphe pour mettre en oeuvre des collaborations entre objets spécifiques à chaque contexte ce qui donne lieu à sa contextualisation.

La contextualisation de graphe d'objets provient du fait qu'il y à la fois héritage de ses relations et adaptation conjointe de ses objets dans un plan fonctionnel. Les relations peuvent alors servir pour supporter les messages liés au contexte.

Soit le graphe d'objets représentant une porte ET considérée ici dans le contexte de simulation chargé de son évaluation. La partie du plan fonctionnel correspondante représentée à la figure 3.24 rend compte de la contextualisation des relations du plan de base relatives à cette exemple (indiquée en pointillés).

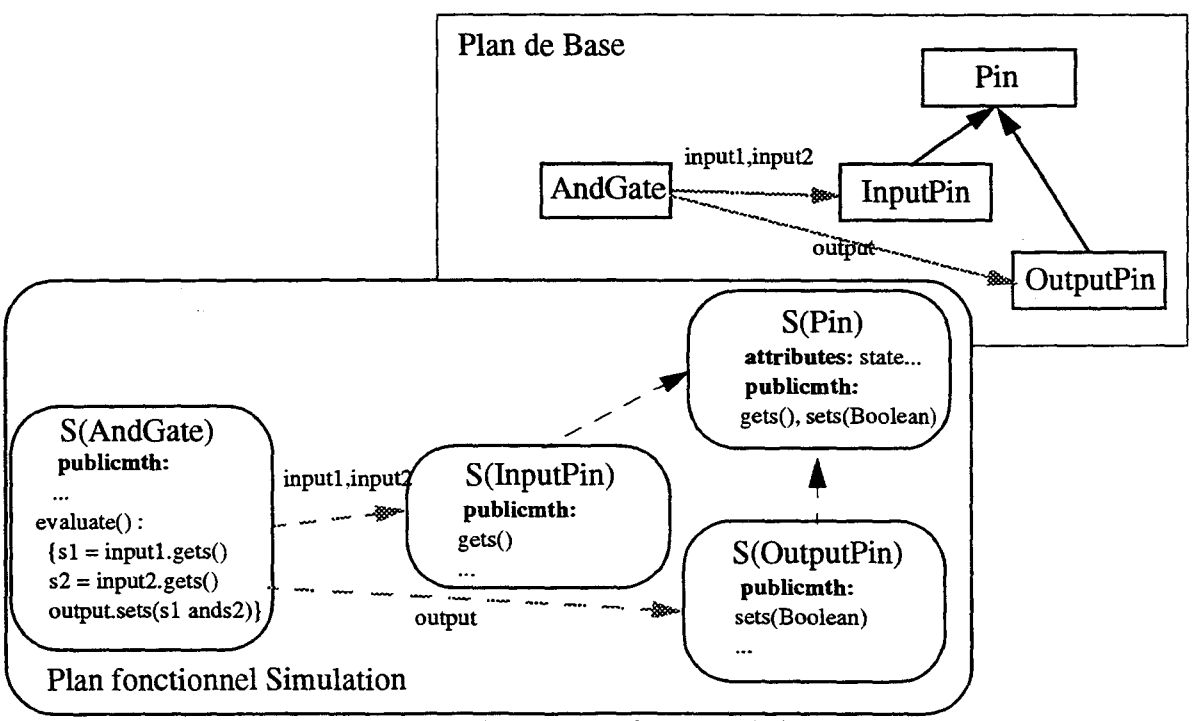


figure 3.24 : Contextualisation pour la simulation du graphe d'objets représentant une porte ET

Dans la partie fonctionnelle associée à la classe `AndGate`, l'accès aux relations issues du plan de base est mis à profit pour définir la méthode d'évaluation `evaluate` d'une telle porte. Les relations `input1` et `input2` sont ici utilisées afin d'invoquer la méthode `gets` d'accès à l'état de chaque broche d'entrée alors que la relation `output` permet de mettre à jour l'état de la broche de sortie en invoquant sa méthode `sets` correspondante.

A travers cet exemple de contextualisation, nous pouvons noter la préservation de l'héritage entre classes localement aux plans ce qui est mis en évidence dans les envois de message `gets` aux deux broches d'entrée. Cette méthode est en effet définie dans la partie fonctionnelle associée à la classe Pin et héritée localement par les sous-classes.

Cette capacité de contextualisation de graphe d'objets exprimés dans le plan de base existe systématiquement pour chaque plan fonctionnel en raison de l'héritage systématique des parties fonctionnelles avec les descriptions de base correspondantes. Un même graphe du plan peut ainsi être contextualisé en parallèle et de façon indépendante dans plusieurs plans fonctionnels. Cette contextualisation multiple trouve tout son sens dans la multiplicité liée aux fonctions. Il devient alors possible à un objet d'utiliser la même relation du plan de base pour communiquer localement avec le même objet dans toutes les fonctions où il intervient.

Montrons ceci en considérant la contextualisation du graphe précédent dans une autre fonction, l'édition schématique, où les portes sont chargées de leur propre affichage graphique. Cette seconde contextualisation est illustrée par la figure 3.25.

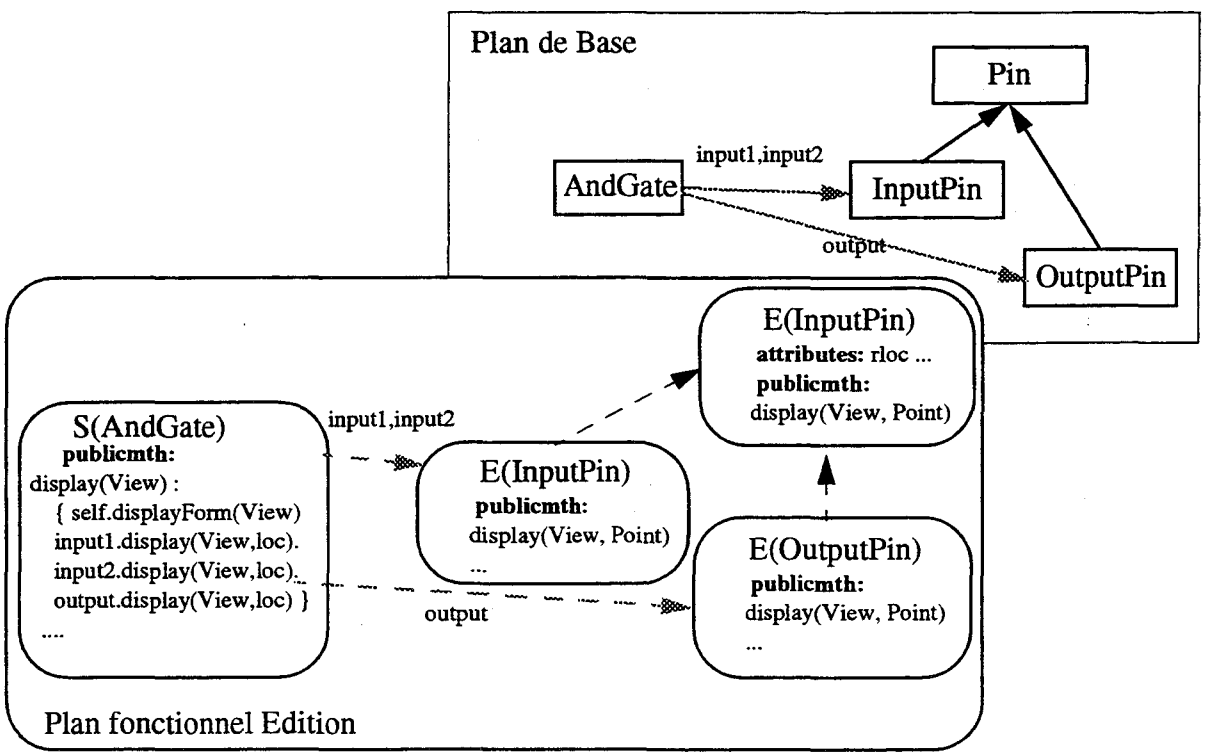


figure 3.25 : Contextualisation pour la simulation du graphe d'objets représentant une porte ET

Sur cette figure, l'utilisation des relations du plan de base est montrée au niveau de la méthode `display` de la partie fonctionnelle associé à `AndGate`. Cette méthode qui implante l'affichage des portes ET traite l'affichage de chaque broche en envoyant à celles-ci le message

`display`.

Suite à ces deux exemples de contextualisation, on s'aperçoit que le même graphe d'objets suffit pour supporter à la fois la fonctionnalité d'évaluation et celle d'affichage. Celui-ci revêt un caractère générique vis à vis des contextes.

En exploitant la capacité de contextualisation des relations au niveau de chaque partie fonctionnelle du même plan, l'ensemble (ou un sous-ensemble) du graphe d'objets déterminé au niveau du plan de base peut de cette façon être contextualisé.

Dans notre exemple où le graphe d'objets défini dans le plan de base vise à refléter la structure interne du circuit, les possibilités de contextualisation de ce graphe sont mises à profit pour réaliser les fonctions d'optimisation, de simulation et de normalisation des circuits. La figure 3.27 présente la contextualisation de ce graphe pour la normalisation des circuits.

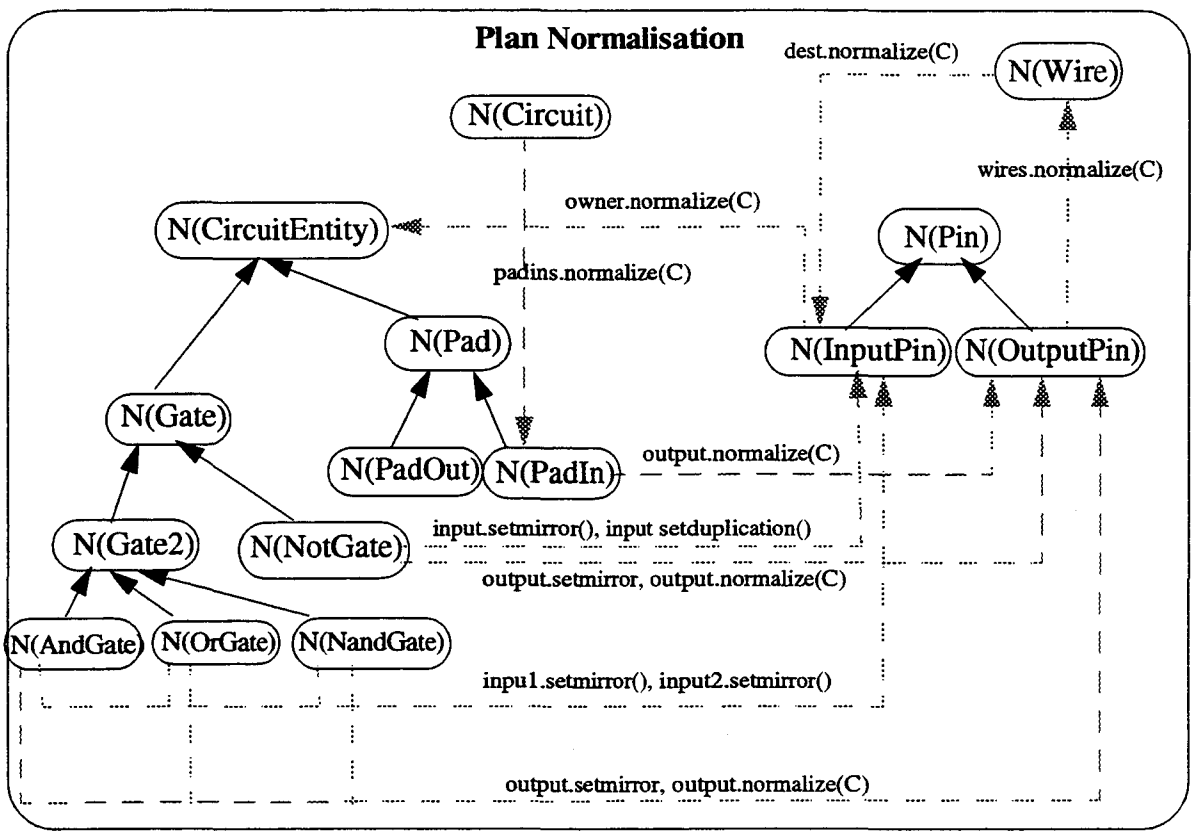


figure 3.26 : Contextualisation pour la normalisation du graphe d'objet représentant un circuit

Cette contextualisation est ici rendue compte par l'utilisation des relations du plan de base pour propager le message `normalize` aux différents objets du graphe.

Cet exemple permet aussi d'apporter deux précisions supplémentaires sur les possibilités de contextualisation des relations du plan de base.

Premièrement, la contextualisation s'applique également pour des relations de base qui sont héritézs. La contextualisation peut dans ce cas avoir lieu dans les parties fonctionnelles des sous-classes de la classe où la relation est introduite. Dans l'exemple ci-dessus, ce point est illustré au niveau des parties fonctionnelles associées aux classes `AndGate`, `OrGate`, `NandGate`

et `NotGate`. Celles-ci font en effet usage des attributs `input1`, `input2` et `output` déclarés dans la description de base de leur sur-classe `Gate` et `Gate2` pour envoyer des messages à leurs broches référencées par ces attributs.

Deuxièment, les possibilités de contextualisation s'appliquent également sur des relations représentées par des références polymorphes. Un exemple d'une telle contextualisation apparait au niveau de la partie fonctionnelle associée à la classe InputPin. Celle-ci exploite en effet l'attribut `owner` du plan de base contenant une référence vers le propriétaire d'une telle broche pour lui envoyer le message `normalize`. Ce message est envoyé sans tenir compte de l'identité plus fine de ce propriétaire en tant que porte Et, porte Ou, port d'entrée, ... Notons au passage que l'attribut `owner` est déclaré dans la description de base de la classe Pin ce qui fait référence au point précédent.

Ces deux points montrent la préservation des propriétés du graphe d'objets, notamment de polymorphisme, lors de sa contextualisation.

Un autre aspect également mis en évidence par ce plan est la contextualisation partielle du graphe. En effet, bien que tous les objets du graphe prennent part à la fonction, seulement une partie des relations est effectivement contextualisée. En particulier, les relations reliant un circuit à ses portes et ses équipotentielles (matérialisés par les attributs `gates` et `wires` dans le plan de base) sont ici inexploitées[1] .

Un autre cas menant à une contextualisation partielle est celui où une partie des objets du graphe n'interviennent pas dans la fonction. C'est le cas par exemple du contexte documentation où les broches des portes n'y participent pas et n'ont pas besoin d'être enrichies dans le plan correspondant.

A travers ces exemples, nous constatons que CROME permet de fonctionnaliser différemment un même graphe d'objets factorisé dans le plan de base.

3.8 Contextualisation de code du plan de base

Après avoir présenté la contextualisation du graphe d'objets factorisé dans le plan de base, cette section expose une autre forme de contextualisation que permet l'approche. Il s'agit de la contextualisation du code du plan de base. Cette caractéristique est illustrée à travers une problématique qui se rencontre fréquemment [Krief 91][Gamma 94]: la description de plusieurs traitements d'un même graphe selon le même processus de parcours. Cette problématique est abordée ici selon deux approches différentes, chaque approche visant à mettre l'accent sur des caractéristiques distinctes de CROME. La première approche est basée sur la factorisation d'un parcours de base de la structure enrichi différemment selon les contextes. La seconde approche repose sur la factorisation d'un parcours de la structure implicitement paramétré pour les contextes. La section se termine par une comparaison avec le patron visiteur (cf chapitre 1).

3.8.1 Factorisation d'un parcours de base et enrichissement selon les contextes

L'implantation des fonctions de simulation, de normalisation et d'optimisation met en évidence que les objets constituant la structure interne du circuit sont activés par envoi de

1. Ces relations sont en revanche contextualisées pour l'édition et la documentation.

message selon un même parcours (méthodes `propagate-tick`, `normalize`, `optimize`). Ce parcours commence par les entrées du circuit (instance de `PadIn`) puis se poursuit à travers la structure selon les relations entre les objets. Ainsi, une entrée du circuit entraine l'activation de la broche de sortie associée puis celle d'une équipotentielle puis celle d'une broche d'entrée puis celle d'une porte et ainsi de suite. Dans ce dernier cas, l'activation d'une porte se produit uniquement si toutes les broches qu'ils possèdent en entrée ont été préalablement activées.

La similitude de parcours de la structure d'objets entre ces différentes fonctions incite à le factoriser dans le plan de base. La description de ce parcours à ce niveau pose néanmoins un problème. Il faut en effet le décrire tout en donnant la possibilité d'associer des traitements différents suivant les contextes.

Cette contextualisation du parcours peut être obtenue en ayant recours aux possibilités de redéfinitions des méthodes localement à chaque contexte. Une première démarche possible consiste à factoriser un parcours de base (i.e sans traitement spécifique) dans le plan de base et à enrichir par redéfinition locale les méthodes implantant ce parcours dans les plans fonctionnels correspondant.

En appliquant cette démarche à notre exemple, le parcours de base de la structure du circuit est implanté en ajoutant aux descriptions de base des classes concernées une méthode `traverse` spécifique prenant en paramètre un objet de type circuit. Cette méthode est codée au niveau de chaque description de base selon un schéma similaire consistant à propager le message `traverse` aux objets de la structure restant à parcourir à partir de l'objet courant.

La figure 3.27 schématise la propagation du message `traverse` entre les objets représentant un circuit. Le signe de multiplication indique l'envoi du même message à plusieurs objets.

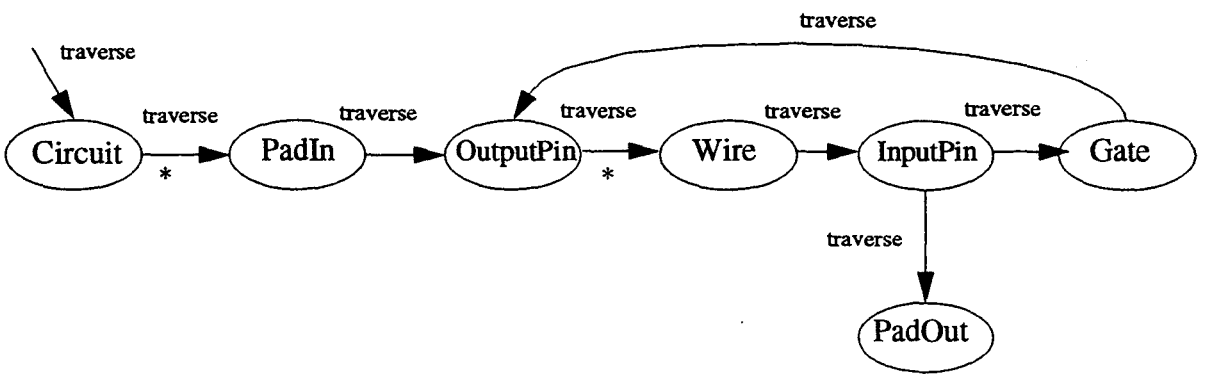


figure 3.27 : Propagation des messages "traverse" pour le parcours d'un circuit

La figure 3.28 montre les ajouts apportés à la description de base des classes pour implanter le parcours de base. Pour ces ajouts, voici quelques précisions:

- la propagation des messages s'appuie sur les relations entre objets introduites au même niveau.

- le début du parcours correspond à la méthode `traverse` de la description de base de la classe Circuit. La terminaison du parcours correspond à la méthode `traverse` de PadOut.

- La description de base de `InputPin` comporte un attribut `visited` de type booléan et des

méthodes pour accéder (`visited`) et modifier (`unmark`) cet attribut. Ces ajouts permettent à une porte de déterminer si ces deux entrées ont été activées auquel cas le reste de la structure à parcourir est entrepris. Les méthodes en question sont exploitées dans la méthode `traverse` déterminée par la partie fonctionnelle de la classe `Gate2`.

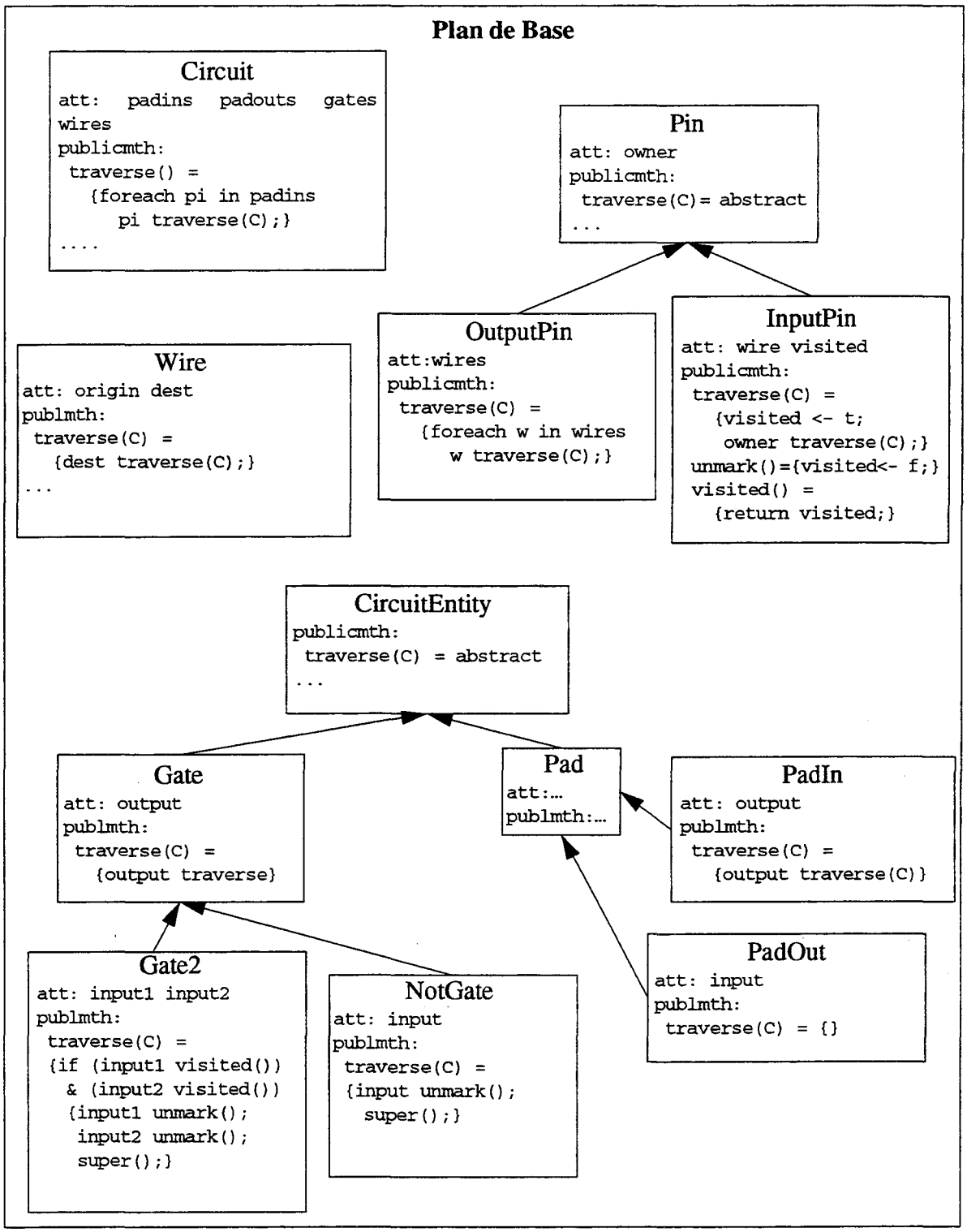


figure 3.28 : Plan de base factorisant un parcours de la structure sans traitements

Nous attirons l'attention sur l'utilisation de l'héritage et des relations entre objets pour

décrire du code factorisé entre plusieurs fonctions. Ces possibilités que nous avions évoquées en section 3.3 sont ici mises en valeur à travers les différentes factorisations et redéfinitions de `traverse` mais aussi l'exploitation des relations pour propager les messages correspondant.

Pour obtenir un contextualisation du parcours factorisée dans le plan de base, il suffit d'enrichir les méthodes `traverse` dans le plan fonctionnel correspondant. Pour chaque méthode, cette enrichissement revient à hériter de la définition tout en la masquant pour lui ajouter les traitements spécifiques au contexte. Les possibilités de redéfinition locale et d'appel au méthode masquée du plan de base permettent de réaliser ces enrichissements.

Comme premier exemple illustrant l'enrichissement de ce parcours, considérons le contexte de normalisation où le circuit résultat est construit selon un parcours de la structure interne du circuit initial. Au cours de ce parcours, chaque objet activé doit intervenir localement en ajoutant au circuit normalisé obtenu en paramètre les éléments lui correspondant.

La figure 3.29 montre les parties fonctionnelles du plan normalisation suite à la factorisation du parcours. Précisons que l'activation du parcours de la structure avec les enrichissement fournies par ce plan est réalisée dans la méthode `normalize` de la partie fonctionnelle associée à la classe Circuit. Cette méthode commence par créer le circuit à élaborer puis lance le parcours en envoyant le message `traverse` à chacune de ses entrées avec une référence vers le circuit en argument.

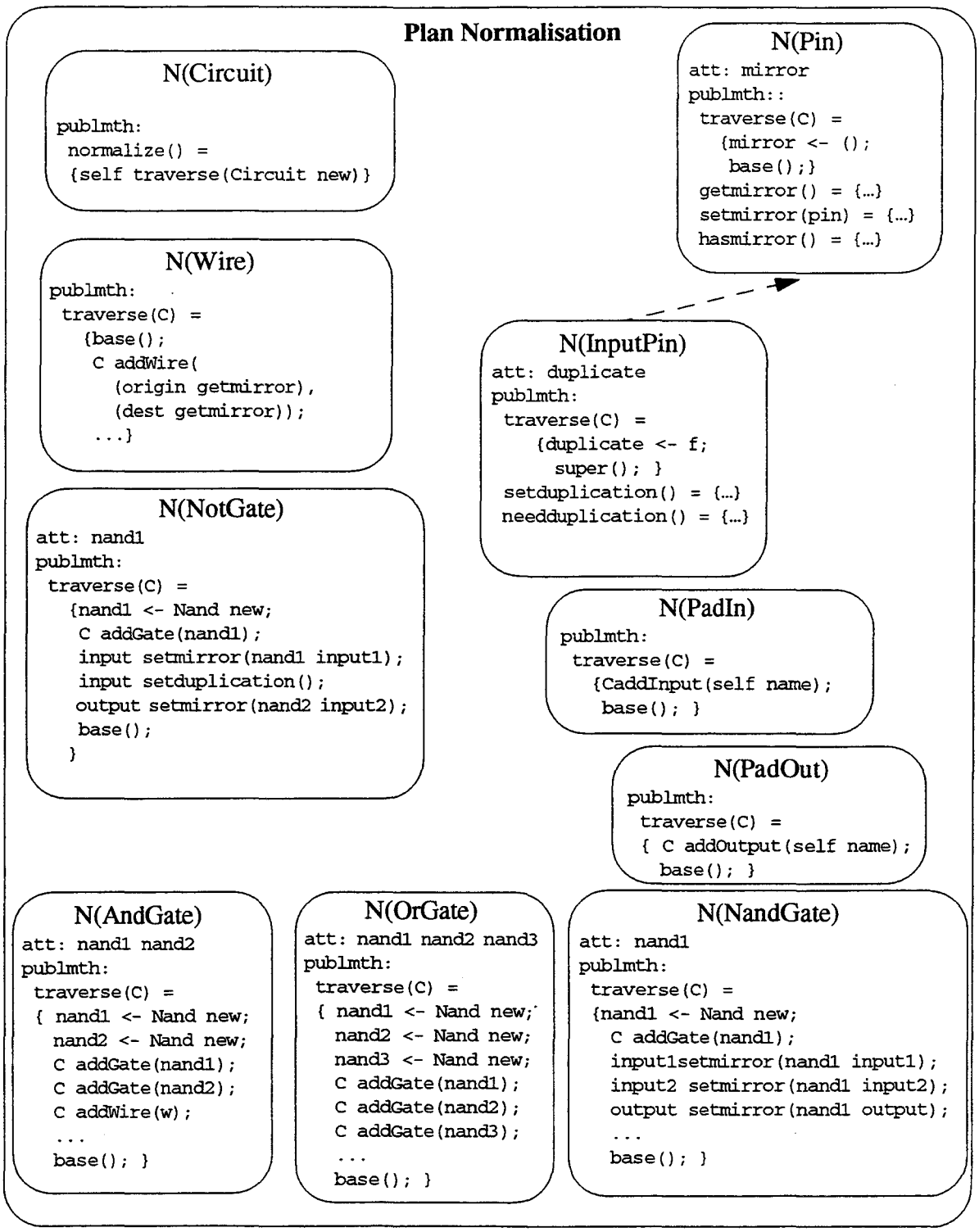


figure 3.29 : Plan Normalisation enrichissant le parcours de la structure

En analysant ce plan, on s'aperçoit que chaque partie fonctionnelle procède à une redéfinition locale de la méthode `traverse`. Ces redéfinitions réalisent l'enrichissement des méthodes correspondantes du plan de base, chacune d'elles apportant des traitements supplémentaires spécifiques au contexte de normalisation. Dans chacune de ces redéfinitions, nous pouvons observer l'utilisation du mécanisme (designé ici par l'opération `base`) pour rappeler la méthode masquée du plan de base et ainsi continuer le parcours.

L'existence de ces redéfinitions dans le plan fonctionnel provoque la contextualisation du graphe de collaboration exprimé dans le plan de base. Selon cette contextualisation, les redéfinitions locales précédentes sont implicitement prises en compte pour les messages `traverse` envoyés dans ce contexte.

Intéressons-nous maintenant à des parties de ce plan qui permettent de montrer plus précisemment certains traits caractéristiques de l'approche.

Pour les portes, nous pouvons remarquer qu'il existe une redéfinition locale de `traverse` dans la partie fonctionnelle associée à chaque type. Ceci provient du fait que chaque type de porte possède une forme normalisée spécifique. Chacune de ces redéfinitions masquent la méthode `traverse` factorisée dans `Gate2` au niveau du plan de base qui définit le comportement de toute porte à deux entrées dans le parcours de base. Cet exemple montre qu'une méthode mise en facteur dans le plan de base peut être redéfinie localement selon plusieurs classes.

Les redéfinitions locales de `traverse` dans les parties fonctionnelles associées à `Pin` et à `InputPin` illustrent d'autres caractéristiques.

La première de ces redéfinitions donne un exemple d'héritage modulaire dans lequel une redéfinition locale associée à une sur-classe masque la définition dans la description de base des sous-classes (cas 3). En l'occurence, la définition `traverse` masquée est celle de `OutputPin` qui n'a ici pas de partie fonctionnelle associée mais par enrichissiment implicite hérite de la redéfinition locale de `Pin`. Ce qui rend compte de la priorité du contexte.

Une autre caractéristique de l'héritage modulaire montrée par cette redéfinition est le traitement d'une référence masquée au plan de base (cas 4). Ici, la définition de `traverse` activée par l'appel à la méthode masquée du plan de base dépend de l'objet : il s'agit de celle de `InputPin` pour un broche d'entrée et de celle de `OutputPin` pour une broche de sortie. Ceci révèle le caractère générique de ce mécanisme dont la résolution n'est pas figée comme c'est le cas pour un super-message.

La seconde redéfinition quant à elle illustre les possibilités de spécialisation des méthodes redéfinie localement à un contexte: une méthode redéfinie localement au niveau d'une classe peut également l'être dans les parties fonctionnelles des sous-classes. Cette possibilité est ici exploitée de façon à ajouter un traitement supplémentaire à la redéfinition locale de `traverse` pour prendre en compte le cas particulier où une broche d'entrée et l'équipotentielle lui étant rattachée sont à dédoubler dans la forme normalisée de sa porte. La capacité à appeler la définition masquée localement au plan par un super-message est également illustrée par cette seconde redéfinition.

Pour finir, cet exemple nous permet d'insister sur la conservation du polymorphisme classique dans la prise en compte des redéfinitions locales pour le contexte. Dans le cas présent, cette conservation est montrée par le message `traverse` exprimé dans la description de base de InputPin. En effet, ce message est envoyé au propriétaire[1] indépendamment de son identité plus fine en tant que composant du circuit (Porte Ou, Porte ET, ...) car celle-ci peut varier d'une broche d'entrée à l'autre. Dans ce contexte, la redéfinition locale de `traverse` appliquée pour

1. référencé par l'attribut owner

ce message sera cependant choisie en fonction de la classe de ce propriétaire.

Pour un autre contexte basé sur le même parcours de structure d'objet, la démarche reste analogue. On contextualise le parcours en procédant à une redéfinition locale des méthodes `traverse` dans le plan correspondant.

Comme exemple de seconde contextualisation du parcours, considérons le contexte de simulation. Dans ce contexte, le parcours des objets du circuit est réalisé pour chaque unité de temps de la simulation de façon à mettre à jour les compteurs associés localement aux objets Porte et Equipotentielle pour prendre en compte leur délai spécifique dans la propagation des signaux.

La figure 3.30 fait apparaitre les parties fonctionnelles du plan simulation (pour des raisons de place seules les méthodes utiles à la discussion sont présentées) quand le parcours est factorisé dans le plan de base. L'activation du parcours est ici réalisée dans la méthode `run` de la partie fonctionnelle associé à la classe `Circuit`. Cette méthode permet de simuler le circuit pendant une periode de temps fournie en paramètre. Elle consiste à itérer le parcours, chaque parcours correspondant à une unité de temps de la simulation.

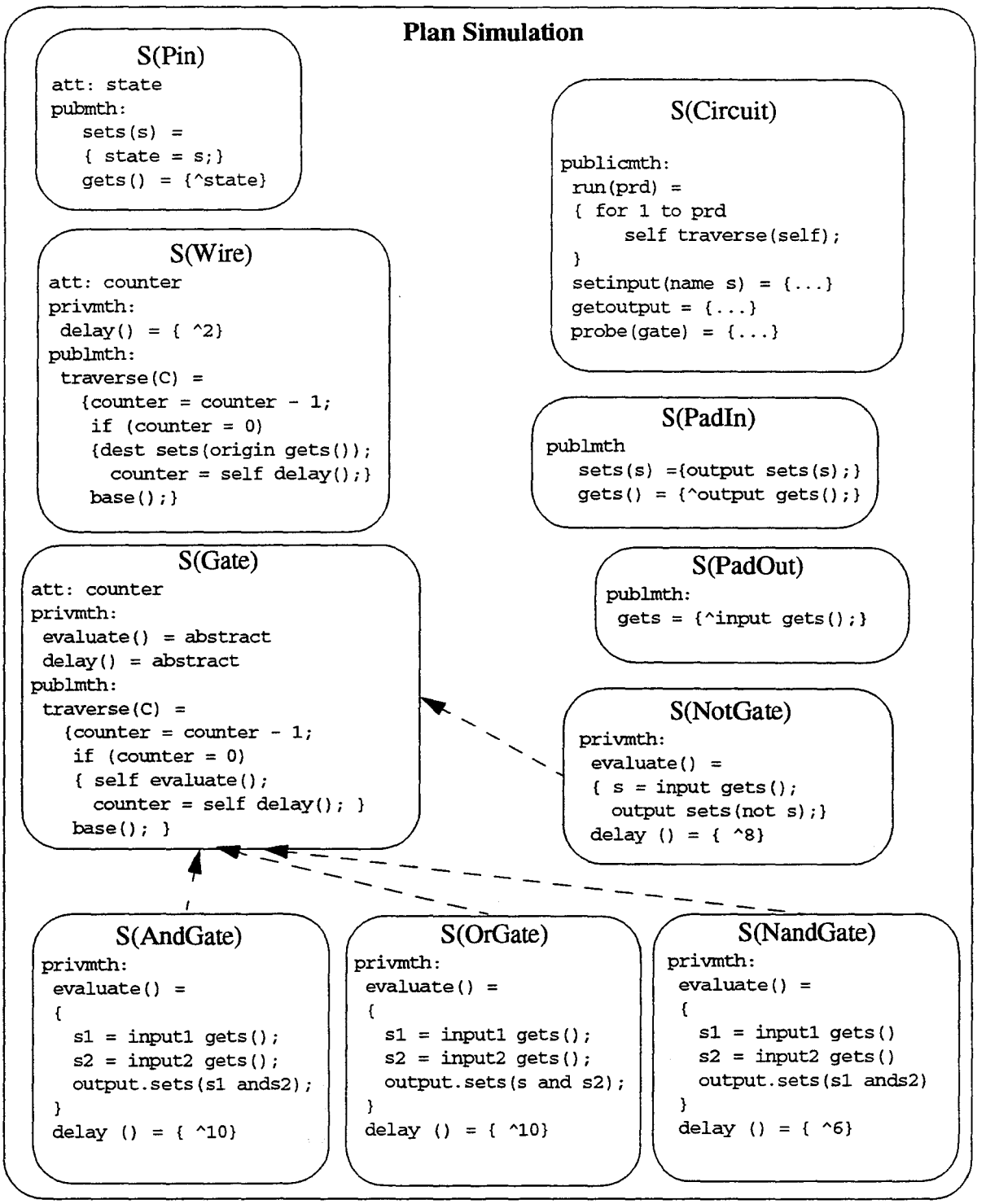


figure 3.30 : Plan Simulation enrichissant le parcours de la structure

Pour ce plan, nous constatons que seules les parties fonctionnelles associées aux classes `Gate` et `Wire` redéfinissent localement la méthode `traverse` afin de procéder à leur enrichissement. Pour les autres classes, la méthode `traverse` qui sera appliquée pour ce contexte est celle fournie par le plan de base.

Comme pour le plan précédent, ces redéfinitions entrainent la contextualisation du graphe de collaboration du plan de base. Cette contextualisation se traduit ici par la prise en compte des

redéfinitions précédentes pour les messages `traverse` envoyés dans ce contexte. Soulignons que cette prise en compte s'applique également lorsque les objets émetteurs de ces messages n'ont pas leur méthode `traverse` redéfinie localement. C'est le cas par exemple des objets `InputPin` dont la méthode `traverse` appliquée pour le contexte de simulation, celle du plan de base, conduit à envoyer le message `traverse` au propriétaire référencé par l'attribut `owner`. Si ce propriétaire est une porte (instance d'une sous-classe de Gate) et non un port du circuit (PadIn ou PadOut), alors ce message résultera en l'application de la méthode `traverse` telle que définie dans `S(Gate)`. Une observation similaire peut être faite pour les broches de sortie qui sont amenées à envoyer le message `traverse` à leurs équipotentielles, une redéfinition locale de la méthode correspondante existant pour ces objets dans ce contexte.

Parmi les redéfinitions locales de `traverse` fournies par ce plan, celle appartenant à S(Gate) montre un aspect intéressant de l'approche. Cette redéfinition est factorisée pour tous les types de portes et masque leur définition de `traverse` déterminée par le plan de base. Les traitements ajoutés par cette redéfinition consistent à décrémenter le compteur d'unité de temps associé à la porte et à déclencher l'évaluation lorsque ce compteur a atteint zéro signifiant la fin du délai. La prise en compte du délai initialisant le compteur et de l'évaluation qui sont spécifiques à chaque type de porte est obtenue en concevant cette redéfinition de manière générique par rapport aux sous-classes. Cela se traduit par les appels aux méthodes `delay` et `evaluate` toutes deux abstraites au niveau de la partie fonctionnelle associée à Gate mais redéfinie dans celles des sous-classes, entrainant ainsi l'affinement implicite de `traverse`.

Comparé à la version de ce plan sans factorisation du parcours, il est intéressant de noter que des simplifications en découlent. Ces simplifications se traduisent notamment ici par l'absence d'enrichissement pour certaines classes comme OutputPin et InputPin. Ceci s'explique par le fait que dans la version précédente l'enrichissement de ces broches était principalement destiné à la réalisation du parcours à travers les éléments du circuit (cf 3.4.3.1). Ce code existe toujours mais il est à présent factorisé au niveau du plan de base. Une autre simplification apportée par cette factorisation est de rendre le code des parties fonctionnelles plus concis.

Concluons cette section en insistant sur les deux propriétés importantes que fait apparaître cette contextualisation multiple du parcours de base accomplie par enrichissement.

La première propriété est la généricité vis à vis des contextes des envois de messages exprimés dans le plan de base. Cette généricité est obtenue grâce au polymorphisme de contexte (cf 3.6) selon lequel un message envoyé au même objet dans des contextes différents peut conduire à des activations de méthodes différentes dépendamment des redéfinitions locales. Cette généricité a été mis en évidence ici pour l'ensemble des messages `traverse` exprimés dans la description de base des objets. En effet, nous avons vu que pour les mêmes envois de message, les méthodes `traverse` activées diffèrent selon les contextes. Ceci confère un caractère générique aux graphes de collaboration exprimés dans le plan de base.

La seconde propriété est la capacité à factoriser partiellement du code entre les fonctions, celui-ci étant complété par redéfinition locale, différemment selon les contextes.

Il importe de préciser que ces deux propriétés sont obtenues tout en continuant à bénéficier des propriétés classiques. Nous avons ainsi montré pour la première propriété que le polymorphisme classique s'applique dans la prise en compte des redéfinitions locales. Pour la seconde propriété, nous avons montré que les affinements exprimés pour le code factorisé sont correctement pris en compte.

3.8.2 Factorisation et contextualisation d'un parcours implicitement paramétré

Lorsque la manière dont les traitements spécifiques sont activés lors du parcours de la structure est identique pour chaque contexte, une seconde approche peut être adoptée. Celle-ci consiste à factoriser dans le plan de base un parcours paramétré implicitement par des méthodes à redéfinir localement selon le contexte. Cette solution met en oeuvre une caractéristique de l'héritage modulaire qui est l'utilisation de référence générique par rapport aux plans fonctionnels (cas 5).

Appliquée à notre exemple, cette seconde démarche conduit à ajouter une méthode `traverse` à la description de base des objets similairement à l'approche précédente. Cependant, pour introduire le paramétrage, ces méthodes y sont codées selon un schéma différent consistant dans l'ordre:

- à appeler une méthode `doactions` définie au même niveau. Cette méthode est destinée à être redéfinie localement suivant chaque contexte. Elle prend en paramètre un objet de type circuit.

- à propager le message `traverse` aux objets de la structure restant à parcourir à partir de l'objet courant. Le paramètre associé à ces messages est le même que celui obtenu en paramètre.

Remarquons que cette façon de procéder mène à séparer clairement les traitements spécifiques à chaque contexte du reste de l'implantation du parcours.

Schématiquement, le parcours de la structure du circuit obtenu est représenté à la figure 3.31. La propagation du message `traverse` entre les objets représentant un circuit est identique à l'approche précédente.

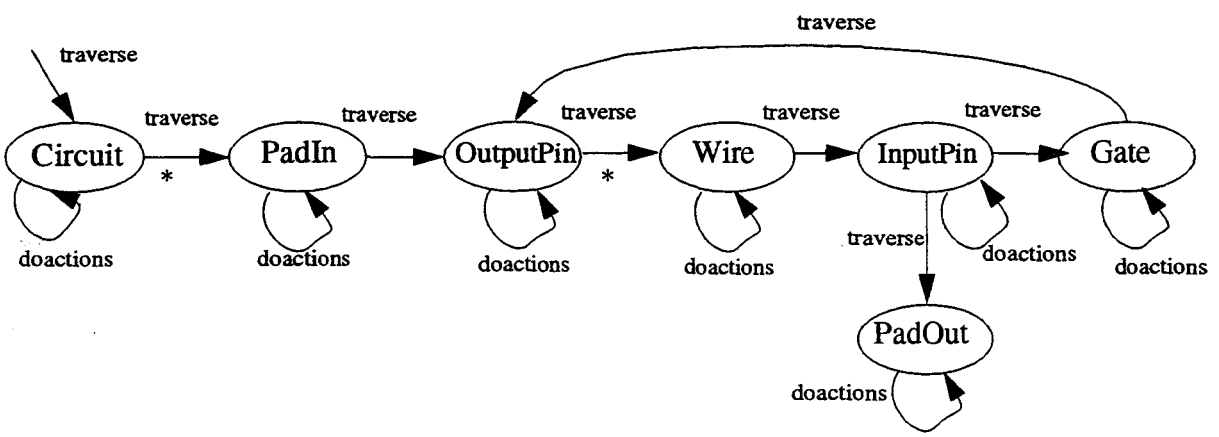


figure 3.31 : Application des méthode "doactions" lors du parcours d'un circuit

La figure 3.32 donne la description de base des classes quand cette solution est appliquée. Comparé au plan de base précédent, on remarque l'ajout d'une méthode `doactions`[1] à chaque description de base et la modification de leur méthode `traverse` pour faire référence à cette méthode. Précisons qu'en procédant ainsi, on précise à quel moment les traitements spécifiques au contexte doivent être effectué lors du parcours.

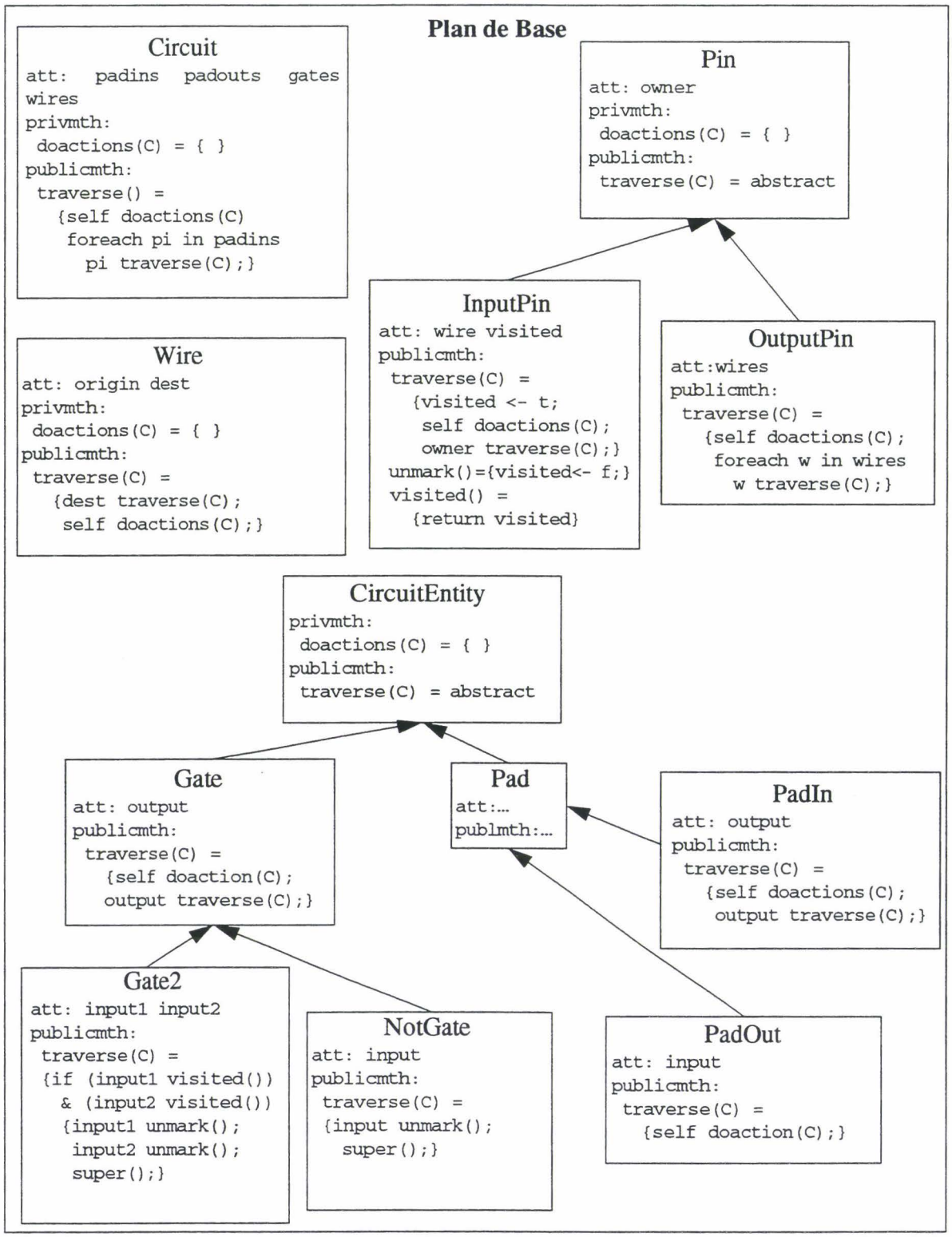


figure 3.32 : Plan de base factorisant un parcours de la structure implicitement paramétré

1. qui ne fait rien par défaut dans le cas présent car l'exemple ne permet d'identifier un traitement par défaut pour les actions activées lors du parcours. A l'extrême, ces méthodes pourrait être déclarées abstraite obligeant ainsi à les redéfinir localement au niveau de chaque contexte.

A partir de ce parcours implicitement paramétré par les méthodes `doactions`, il suffit alors de redéfinir localement ces dernières pour obtenir un parcours adapté à un contexte.

Comme exemple, prenons le plan normalisation reconçu selon cette approche. Les parties fonctionnelles obtenues pour ce plan sont présentées à la figure 3.33.

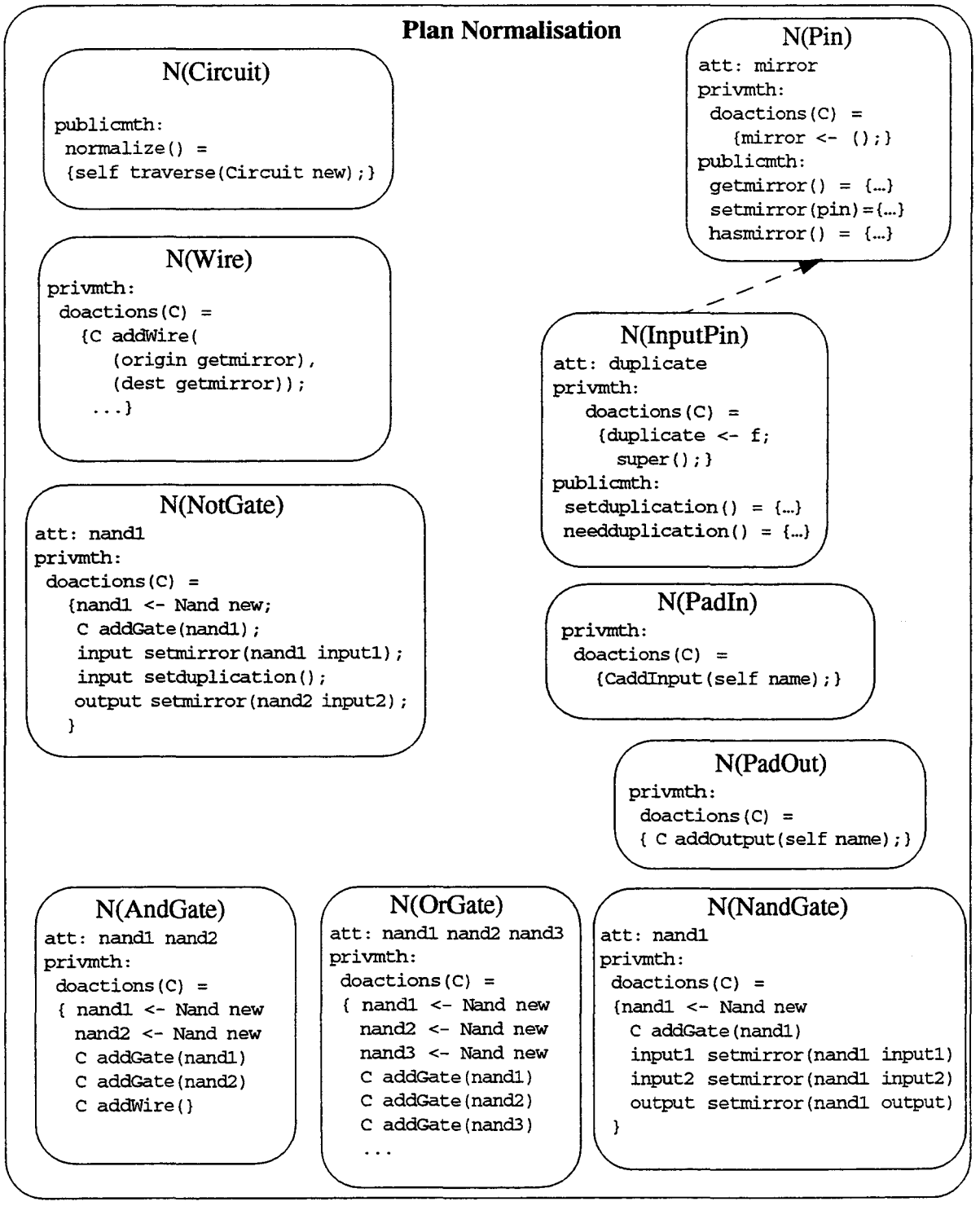


figure 3.33 : Plan Normalisation paramétrant le parcours factorisé

A l'examen de ce plan, nous observons que les parties fonctionnelles procèdent à la redéfinition des méthodes `doactions` sans redéfinir les méthodes `traverse` appelantes. Ces

redéfinitions implantent les traitements spécifiques à réaliser lors du parcours.

L'existence de ces redéfinitions a pour conséquence l'affinement implicite des méthodes `traverse` appelantes pour le contexte sans que celles-ci soient redéfinies. Cet affinement provient du fait que les appels de `doactions` exprimés dans ces méthodes sont résolus selon l'héritage modulaire en tenant prioritairement compte des redéfinitions locales au contexte.

Pour un autre contexte, celui de simulation par exemple, on procédera de façon analogue, c'est à dire par redéfinition locale des mêmes méthodes `doactions` dans le plan fonctionnel correspondant, ces redéfinitions causant un nouvel affinement implicite des méthodes `traverse`.

Comme précédemment, nous terminons en insistant sur une propriété que fait apparaître cette contextualisation multiple du parcours implicitement paramétré. Il s'agit de la possibilité au niveau du plan de base de définir des méthodes génériques par rapport aux contextes. Cette généricité des méthodes est ici rendue compte par les affinements implicites des méthodes `traverse`, différents selon les contextes.

Nous attirons l'attention sur le fait que cette généricité est obtenue sans perdre la généricité classique relativement aux sous-classes. Ainsi, dans l'exemple précédent, la méthode `traverse` définie dans la description de base de `Gate2` est à la fois générique par rapport aux contextes mais aussi par rapport aux sous-classes. Celle-ci est en effet adaptée implicitement pour les objets des sous-classes `AndGate`, `OrGate` et `NandGate` dans le contexte de normalisation à travers la redéfinition de la méthode `doactions`.

3.8.3 Comparaison avec le patron de conception Visiteur

Au cours de la section 2.3.2.2, nous avons présenté le patron de conception Visiteur qui constitue une solution partielle au problème de la réalisation de plusieurs traitements sur une même structure considérée ici en exemple. Nous proposons d'expliciter dans cette section les apports de notre approche comparée à cette solution. Pour cette comparaison, nous prenons la version du patron visiteur dans laquelle les objets de la structure prennent en charge le parcours pour permettre son partage et notre seconde approche qui s'apparente le plus avec ce patron.

Les deux approches diffèrent radicalement sur la manière de traiter le problème. La démarche proposée par le patron visiteur est fondée sur l'externalisation des traitements dans des objets spécialisés activés par les objets de la structure à parcourir. De son coté, CROME propose une démarche où les traitements sont directement rapportés aux objets de la structure.

Pour faciliter la comparaison, il nous parait utile d'établir un parallèle entre les méthodes élaborées dans les deux approches:

- les méthodes `traverse` peuvent être rapprochées des méthodes `accept`, ces méthodes étant dans les deux cas associées aux objets de la structure pour réaliser le parcours;

- les méthodes `doactions` peuvent être rapprochées avec les méthodes `visit`, ces méthodes mettant en oeuvre dans les deux cas les traitements spécifiques activés lors du parcours;

A partir de cette mise en correspondance, nous pouvons dégager des différences notables sur

deux points précis:

- la description des traitements: Avec l'approche par visiteur, les méthodes implantant un traitement sont décrites dans une même classe de visiteur. Avec CROME, de telles méthodes sont décrites dans les parties fonctionnelles d'un même plan. Dans les deux approches, on remarque que les traitements sont isolés modulairement les uns des autres. Pour le patron visiteur, cette modularité des traitements est obtenue physiquement en les localisant dans des classes d'objets distinctes. Pour CROME, cette modularité est obtenue au niveau logique par la structuration en plans tout en préservant la modularité liée aux objets.

- le paramétrage du parcours: Pour le patron visiteur, ce paramétrage est obtenu dans les méthodes `accept` par envoi de message `visit` approprié aux objets visiteurs passés en paramètre lors du parcours. Dans l'approche CROME, ce paramétrage est remplacé par la généricité de la méthode `traverse` qui fait appel à la méthode `doactions` redéfinie de manière appropriée dans chaque contexte.

Ces différences procurent à notre approche plusieurs avantages que nous proposons d'expliciter.

Contrairement au visiteur, notre approche n'aboutit pas à une séparation entre les traitements et les objets auxquels ils se rapportent. Ces traitements sont directement associés aux objets qu'ils concernent et ont ainsi directement accès à leur environnement, à leur structure interne notamment. L'encapsulation des objets de la structure est donc préservée par notre approche. Ceci contraste avec l'approche par visiteur où il faut souvent briser cette encapsulation du fait de l'externalisation pour permettre aux objets visiteurs de fonctionner (cf chapitre 2).

Le patron visiteur pose également des difficultés dès qu'il s'agit d'ajouter de nouveaux états aux objets nécessaires à la réalisation des traitements. On est obligé dans ce cas de les gérer globalement au niveau des objets visiteurs. Dans notre approche, l'ajout de nouveaux états est directement supporté par les capacités d'enrichissement des parties fonctionnelles. Ces dernières permettent de plus de ranger ces états dans les objets concernés.

A l'inverse de notre approche, l'externalisation des traitements dans les objets visiteurs ne permet pas de réutiliser la hiérarchie d'héritage des classes de la structure pour mettre en oeuvre ces derniers. Dans une classe de visiteur, les méthodes `visit` associées aux objets de la structure sont mises à plat conduisant à une perte importante des bénéfices de l'héritage (factorisation complète, factorisation partielle, généricité). A ce titre, on peut prendre en exemple la méthode générique `traverse` factorisée au niveau de la partie fonctionnelle associée à la classe `Gate` pour la simulation. Pour parvenir à un comportement analogue avec les visiteurs, la seule solution dont on dispose consiste à recopier le code générique dans la méthode `visit` prévue pour chaque type de porte et à remplacer les appels aux méthodes `delay` et `evaluate` contenus dans ce dernier par le code correspondant.

Par manque de généricité, certains cas de figure requièrent des efforts de programmation plus importants avec les visiteurs que dans notre approche. Ainsi, une factorisation similaire à celle de la méthode `traverse` du plan de base au niveau de `Gate2` s'obtient plus difficilement avec les visiteurs. En effet, selon un schéma analogue, la solution consistant à envoyer un message `visit` dans la méthode `accept` factorisée dans `Gate2`, ne permettrait pas l'activation de traitements différents pour des objets des sous-classes. Dans ces conditions, la solution consiste à paramétrer la méthode `accept` factorisée dans `Gate2` par une seconde méthode destinée à être

BU LILLE 1

redéfinie dans les sous-classes pour y invoquer la méthode `visit` appropriée.

Enfin, finissons cette comparaison en soulignant une dernière caractéristique spécifique à notre approche procurant des facilités impossibles à obtenir avec le patron visiteur. Il s'agit de la possibilité d'adapter modulairement la façon dont les traitements spécifiques sont activés lors du parcours de la structure. Il suffit pour cela de procéder à la redéfinition locale des méthodes `traverse` comme indiqué dans notre première approche. Par comparaison, dans l'approche par visiteur, les activations des traitements à réaliser lors du parcours sont cablées une fois pour toutes et ne peuvent être changées pour des besoins particuliers.

3.9 Programmation entre contextes au sein de l'objet

Dans cette dernière partie, nous présentons les réponses de CROME au problème d'articulation entre les fonctions du système centrées sur les mêmes objets. Le besoin d'articulation entre les fonctions peuvent exister pour diverses raisons. Il peut s'agir :

- d'échanger des données entre les fonctions;
- de coordonner une fonction par rapport à un état fourni par une autre;
- ou encore de gérer la cohérence entre plusieurs fonctions.

En CROME, où les fonctions sont diffusées tranversalement dans les objets, cette articulation est directement traitée au sein de ces derniers, en prenant en compte leur structuration interne. De cette façon, chaque objet du référentiel constitue un lieu implicite d'articulation entre les contextes.

Pour gérer cette articulation au sein des objets, deux moyens sont proposés:
- le partage entre contextes obtenu à travers la description de base;
- l'appel de méthodes entre contextes au sein de l'objet.

Les deux sections suivantes présentent successivement ces deux moyens d'articulation. Le partage ayant déjà été abordé à travers plusieurs sections, nous le présentons rapidement pour nous attarder plus amplement sur l'appel de méthodes entre contextes.

3.9.1 Partage entre contextes

Les parties fonctionelles d'une même classe entretiennent une relation d'héritage avec la description de base de celle-ci et ont accès à toutes ses caractéristiques selon des capacités identiques. Plusieurs parties fonctionnelles peuvent ainsi être amenées à exploiter conjointement les mêmes caractéristiques de la description de base ce qui conduit à leur partage. La figure 3.34 schématise ce partage.

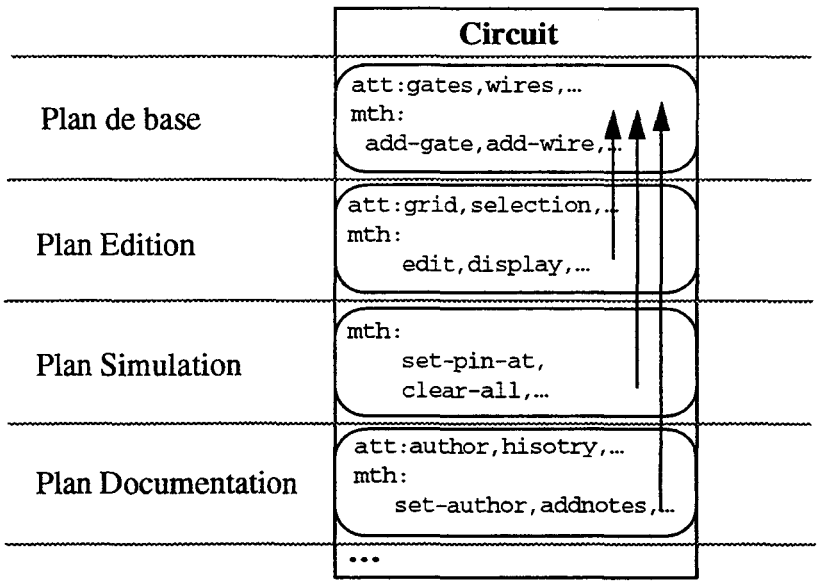


figure 3.34 : Partage entre contextes au sein des objets Circuit

Le partage d'une méthode se produit lorsque des méthodes appartenant à des parties fonctionnelles distinctes font appel à une même méthode de la description de base. Nous n'insistons pas sur ce cas qui s'apparente à celui de plusieurs méthodes d'une classe appelant une même méthode de celle-ci. Dans notre système, un exemple d'un tel partage est donné par la méthode `check-completion` appartenant à la description de base de la classe Circuit. Cette méthode vérifie si le circuit est complètement construit, c'est à dire ne contient pas de broches non reliées. Elle est appelée par les parties fonctionnelles de cette même classe déterminée pour les fonctions de simulation, normalisation et d'optimisation de façon à éviter la réalisation de traitements correspondants sur un circuit incomplet.

Le partage d'un attribut se produit lorsque des méthodes appartenant à des parties fonctionnelles distinctes manipulent le même attribut de la description de base[1]. Ce cas s'apparente au cas d'une classe dont plusieurs méthodes d'une classe en appellent toutes une unique autre. Il en découle un partage de la valeur des attributs de la description de base entre les parties fonctionnelles concernées. Selon ce partage, la modification de la valeur d'un attribut par une partie fonctionnelle[2] est immédiatement effective pour toutes les autres parties fonctionnelles.

Ce partage d'attribut entre parties fonctionnelles a déjà été illustré dans la section 3.7 à travers la contextualisation multiple de graphes d'objets factorisés dans le plan de base.

Plus généralement, ce partage d'attribut constitue un support pour la transmission de données entre parties fonctionnelles et donc entre contextes. Un contexte peut ainsi déterminer la valeur d'un attribut que des autres contextes sont amenés à exploiter. Par exemple, le lien d'une équipotentielle à une broche d'entrée rangé dans l'attribut `dest` de la description de base est déterminé par le contexte d'édition et consommé par les autres contextes.

---

1. Ces attributs de la description de base sont les seuls qui soient accessibles à l'ensemble des parties fonctionnelles de la classe.
2. ou également par les méthodes de la description de base

Signalons que ce partage d'attributs peut poser des problèmes. Ainsi, les problèmes d'interférence pouvant intervenir entre des méthodes d'une classe manipulant une même attribut se retrouvent également ici entre des méthodes de parties fonctionnelles différentes. Le maintien de la cohérence d'une partie fonctionnelle dépendante d'attributs partagés peut également s'avèrer problématique. En effet, pour garantir sa cohérence, une telle partie fonctionnelle requiert d'être informée de tout changement de valeur de ces attributs effectué par les autres parties fonctionnelles.

En l'état actuel, CROME ne propose pas de solution dédiée à ces deux types de problèmes. Ceux-ci sont réglés de manière ad-hoc. Pour le premier problème, une méthode est introduite dans la description de base qui est la seule à déterminer la valeur, les parties fonctionnelles desirant modifier cette valeur y faisant obligatoirement appel. Pour le second problème, la possibilité d'appels de méthodes inter-contextes est employé (cf section suivante). Des solutions plus adaptées peuvent être envisagées pour chacun de ces problèmes: utilisation d'invariant exprimé dans la description de base sur les attributs partagés pour le premier problème, déclenchement de traitements compensatoires pour le second. L'intégration de ces solutions dans CROME reste un thème à approfondir.

3.9.2 Appel de méthodes inter-contextes

Cet appel de méthode au sein de l'objet permet de répondre à des besoins d'articulation entre contextes portant sur des aspects spécifiques[1] tout en assurant leur encapsulation. Ce moyen d'articulation est complémentaire du partage décrit précédemment qui permet seulement l'articulation sur des caractéristiques de la description de base.

Dans les prochaines sections, nous examinons les différentes caractéristiques liées à cet appel de méthodes.

3.9.2.1 Principes

L'appel de méthodes inter-contextes est basé sur les parties fonctionnelles définies pour une même classe dans des plans différents. Il consiste dans une méthode d'une partie fonctionnelle à appeler une méthode d'une autre partie fonctionnelle, ceci au sein de l'objet.

Nous partons du principe que cette capacité existe systématiquement pour l'ensemble des parties fonctionnelles de la classe. La figure suivante met en évidence par des flèches les possibilités d'appels entre trois des parties fonctionnelles associées à la classe Circuit.

1. C'est à dire détenus par les parties fonctionnelles correspondantes.

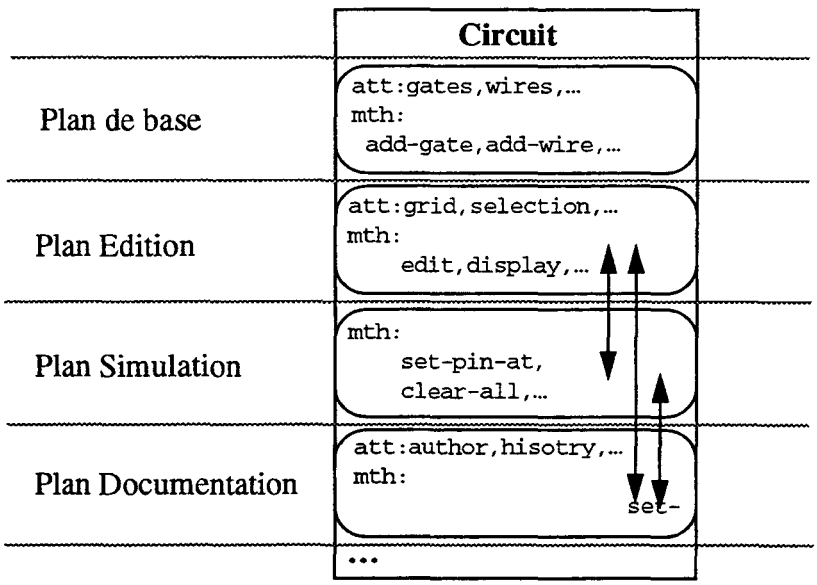


figure 3.35 Possibilités d'appels entre parties fonctionnelles d'une classe

Ces flêches montrent d'une part qu'une partie fonctionnelle a la possibilité d'appeler des méthodes appartenant à des parties fonctionnelles distinctes. Elles montrent d'autre part que des parties fonctionnelles distinctes peuvent faire appel aux méthodes d'une même partie fonctionnelle.

Dans une partie fonctionnelle, nous n'imposons pas de restriction sur l'accès aux méthodes des autres parties fonctionnelles de la même classe. Tout méthode d'une partie fonctionnelle, qu'elle soit publique ou privée, est immédiatement visible aux autres parties fonctionnelles de la classe. L'objet, décomposé en parties fonctionnelles reste l'unité d'encapsulation du système. Ce choix permet à travers des méthodes privées d'offrir un accès interne et protégé à une partie de l'état maintenu par un autre contexte ou un accès à des opérations spécialement conçues pour être appelées depuis un autre contexte (cf l'exemple de la méthode `record-add-gate` du prochain paragraphe). Sans cet accès privilégié, l'objet considéré dans un contexte n'aurait pas plus d'accès à ses propres caractéristiques que les objets participants au contexte pour lequel elles sont introduites, ce qui nous semble trop restrictif. Signalons que, selon ce choix, il n'y a pas de possiblité pour interdire l'accès de méthodes aux autres parties fonctionnelles[1]. Il convient par conséquent de prendre des précautions particulières de façon à ne pas provoquer d'effets de bord entre contextes au sein de l'objet.

Dans notre exemple, l'appel de méthode intra-objet inter-contextes est mis à profit au sein des objets représentant les circuits pour faire coopérer le contexte d'édition avec le contexte documentation. Cet appel est destiné à maintenir automatiquement un historique des

1. Une amélioration possible serait d'introduire une nouvelle catégorie de méthode au sein d'une partie fonctionnelle dédié à l'intéraction inter-contexte, un peu à la manière de la modalité "protected" proposé en C++ et JAVA pour spécifier les accès possibles aux sous-classes. Les méthodes faisant partie de cette catégorie seraient dans ce cas visibles aux autres parties fonctionnelles tandis que celles déclarées avec la modalité private seraient seulement visibles au sein d'une partie fonctionnelle (et celle des sous-classes). On réserverait cette dernière modalité pour des besoins d'implantation de la partie fonctionnelle (factorisation d'une partie du code commun à plusieurs méthodes, décomposition de méthode complexe) ou pour toute méthode de la partie fonctionnelle dont l'appel depuis une autre partie fonctionnelle risquerait d' entrainer des effets de bord non souhaitées dans le contexte appelé.

modifications structurelles apportées au circuit lors de sa conception (ajout, suppression de composant), le but de cet historique étant d'enrichir la documentation fournie par l'utilisateur. La gestion de cet historique par l'objet circuit repose sur des appels de méthodes entre les deux contextes qui dépendent des manipulations effectuées. Les méthodes en question sont spécialement conçues pour enregister les différents types de manipulations dans l'historique. La figure 3.36 montre les appels de méthodes entre les deux parties fonctionnelles correspondantes de la classe circuit.

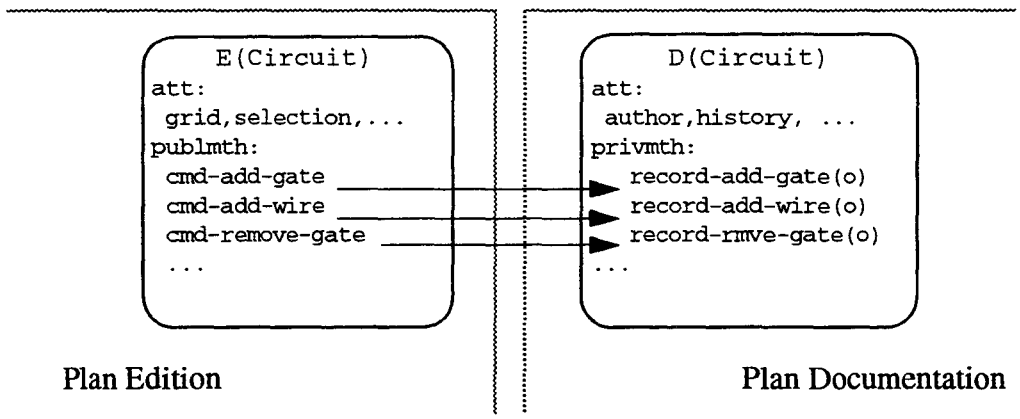


figure 3.36 : Appels de méthode entre les parties fonctionnelles de la classe Circuit

Remarquons ici que les méthodes appelées ne sont pas rendues publiques. De ce fait, elles sont seulement destinées à être invoquées en interne par l'objet. Il faut préciser cependant que ces méthodes ne sont pas exclusivement réservées à l'objet dans son contexte d'édition. Les autres parties fonctionnelles associées à la classe Circuit y ont accès également. En particulier, l'activation de ces méthodes peut être envisagée dans les parties fonctionnelles du circuit dédiées à l'optimisation et à la normalisation afin d'obtenir une trace du processus correspondant.

3.9.2.2 Conséquence du changement de contexte

Dans une partie fonctionnelle, tout appel à une méthode d'une partie fonctionnelle appartenant à un autre contexte provoque un changement de contexte. Pour l'application de la méthode et pour tout appel de méthode qui s'ensuit, l'objet se comporte selon le contexte appelé.

La figure suivante illustre le changement de contexte pour l'exemple discuté précédemment. Pour bien montrer la conséquence d'un tel changement sur les relations inter-objet, nous y avons également figuré la relation de ce circuit avec une de ses portes.

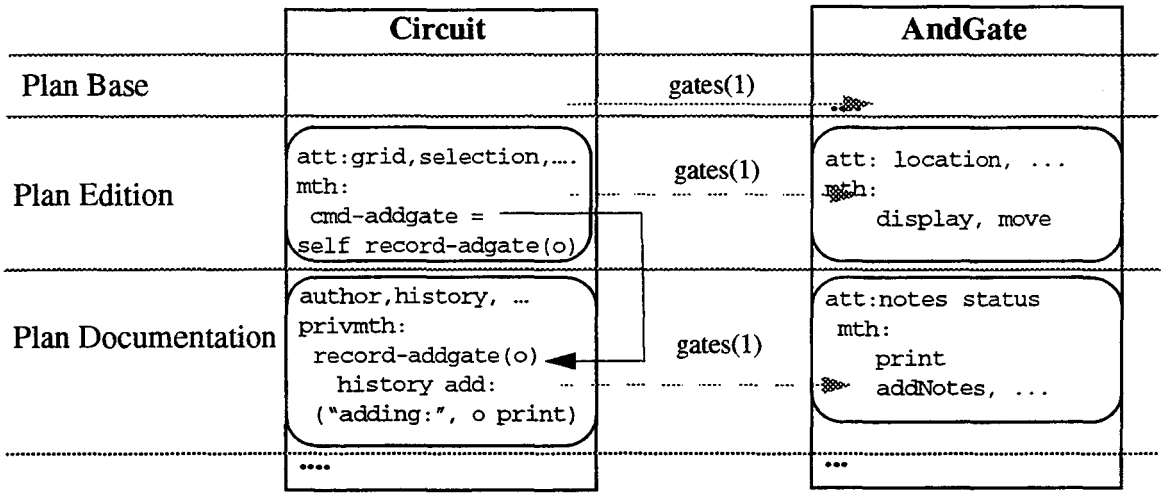


figure 3.37 : Changement de contexte au sein d'un objet Circuit

L'objet circuit se trouve dans le contexte d'édition au moment de l'activation de sa méthode cmd-addgate qui contient un appel à une méthode record-addgate introduite pour le contexte de documentation. Lorsque cette seconde méthode est appliquée, un changement de contexte en résulte. L'objet circuit adopte alors le comportement associé au nouveau contexte et il est en de même pour tout objet activé à partir de cette méthode. Dans l'exemple ci-dessus, le changement de contexte amène l'objet circuit à communiquer avec sa porte dans le contexte de documentation.

Ce qui vient d'être expliqué au niveau de la relation détenue par un objet vaut également pour les références d'objets passées en paramètres lors de l'appel de la méthode provoquant le changement de contexte. Autrement dit, tout objet transmis dans l'appel se comporte selon le nouveau contexte. Un comportement similaire se produit dans le cas d'une référence à un objet retourné comme résultat de l'appel ayant engendré un changement de contexte. Dans ce cas précis, l'objet retourné est amené à se comporter selon le contexte d'appel.

Dans l'exemple décrit à la section précédente, une telle transmission d'objet peut être localisée dans les appels de méthodes inter-contextes réalisées entre le contexte d'édition et de documentation au sein d'un objet circuit. En analysant attentivement la figure, on peut en effet observer que la méthode `record-addGate` prend en argument l'objet concerné par l'action à enregistrer dans l'historique. Dans le contexte de documentation, l'objet circuit fait participer les objets qu'il a ainsi reçus en sollicitant leur fonctionnalité spécifique à ce contexte .

Dans cette partie, nous avons présenté les conséquences liées à une combinaison de méthode inter-contextes. Ceci nous montre finalement que l'appel dans une méthode d'une méthode située dans un autre plan est fondamentalement différente d'un appel à une méthode située dans le même plan ou appartenant au plan de base. Dans ce dernier cas, la méthode appelée est en effet appliquée sans qu'il se produise de changement de contexte. Dès lors, l'objet continue de se comporter dans le même contexte. En outre, les objets dont les références sont passés en argument ou obtenus comme résulat de l'appel reste également dans le même contexte.

3.9.2.3 Appel de méthodes inter-contextes et héritage local

Lors des paragraphes précédents, la combinaison de méthode intra-objet inter-contextes a seulement été examinée dans le cadre d'une seule classe. Cette sous-section précise cette combinaison quand on fait intervenir l'héritage entre classes selon les deux points suivant.

Premièrement, une partie fonctionnelle associée à une classe peut faire appel aux méthodes des autres parties fonctionnelles de la classe mais aussi à celles des parties fonctionnelles associées aux sur-classes du fait de l'héritage local à chaque plan. Ce point est illustré par la figure 3.38 qui montre par des flêches les possibilités d'appel du plan `F1` vers le plan `F2` entre parties fonctionnelles de la même classe `C`.

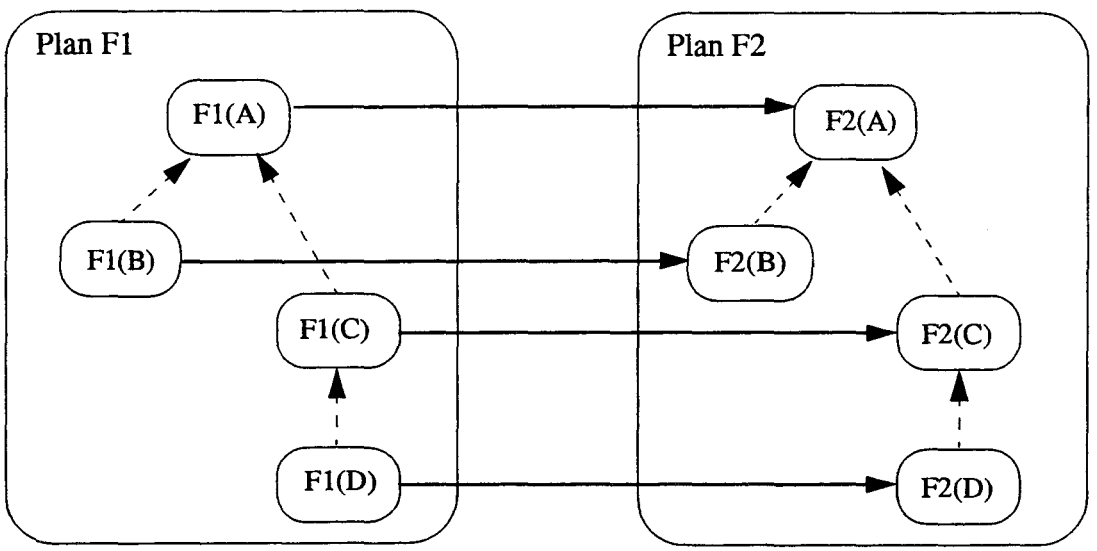


figure 3.38 : Appel de méthode inter-contextes et héritage local

D'après cette figure, les méthodes introduites dans `F2(A)` sont accessibles à ses autres parties fonctionnelles, soit à `F1(A)` dans le cas présent. Elle l'est également par héritage pour les parties fonctionnelles des sous-classes, en l'occurence pour `F1(B)`, `F1(C)` et `F1(D)`. Par contre, en respect de la conception incrémentale des classes[1], une partie fonctionnelle d'une classe ne peut faire référence aux méthodes introduites pour les sous-classes dans un autre plan. Ici, par exemple, `F1(C)` ne peut exploiter les méthodes de `F2(D)`.

Deuxièmement, l'affinement d'une méthode au sein d'un plan fonctionnel intervient aussi pour la combinaison de méthodes inter-contextes. Autrement dit, toute méthode redéfinie dans un plan fonctionnel cause un affinement implicite des méthodes y faisant appel dans les autres plans fonctionnels. La figure suivante va nous servir à illustrer ce point.

1. selon laquelle chaque classe n'a connaissance que d'elle-même et de ses surclasses et décrit à son niveau un modèle complet (éventuellement partiellement abstrait) d'objets.

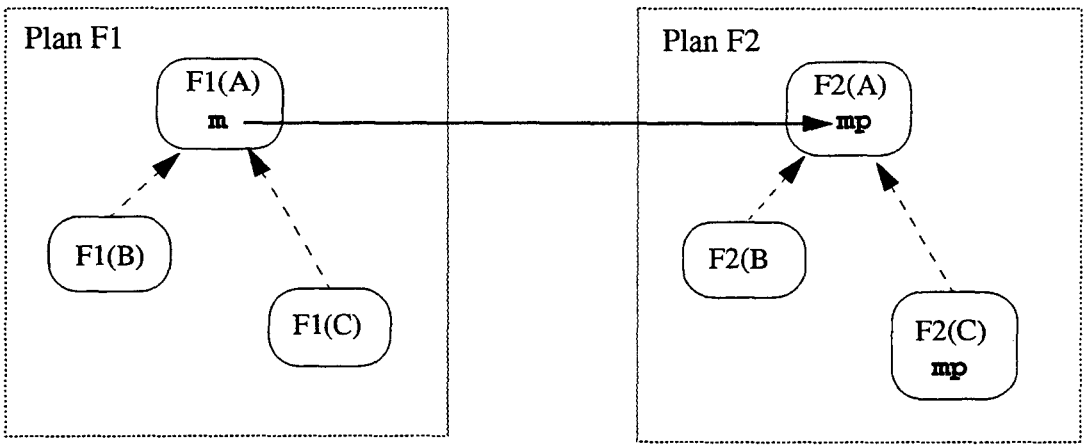


figure 3.39 : Appel de méthode inter-contextes avec redéfinition

Cette figure fait apparaître un appel de méthode intra-objet inter-contextes au niveau de la partie fonctionnelle associée à la sur-classe `A` dans le plan `F1`. Dans le cas présent, la méthode `m` de cette partie fonctionnelle est définie en faisant appel à la méthode mp introduite dans la partie fonctionnelle de la même classe pour le plan `F2`. Or cette méthode est redéfinie localement pour la sous-classe `C` dans ce plan. L'existence de cette redéfinition a alors pour conséquence d'affiner implicitement la méthode m appelante lorsque celle-ci est héritée localement pour la sous-classe `C` dans le plan `F1`. A l'appel de la méthode mp par la méthode m, c'est la définition la plus affinée pour la classe C dans le plan F2, soit celle de `F2(C)`, qui sera appliquée.

En définitive, cet affinement implicite donne lieu à une forme de généricité intra-objet inter-contextes. Il devient possible ainsi d'écrire dans un plan fonctionnel des méthodes génériques par rapport à des méthodes situées dans d'autres plans fonctionnels, sans tenir compte de la façon dont ces méthodes seront redéfinies au sein de ces derniers. Notons que ceci constitue une généralisation à travers les plans des capacités de référence générique au plan de base.

3.9.2.4 Communication inter-objets inter-contextes

CROME ne propose pas directement de possibilités d'envoi de message entre contextes de façon à réduire le couplage entre plans et à limiter les risques d'effets de bord entre contextes. Cependant, comme nous allons le voir sur un exemple, certaines situations peuvent nécessiter l'envoi de tels messages. L'objectif de cette section est de présenter comment prendre en compte de tels besoins en combinant les possibilités d'appel de méthodes intra-objet inter-contextes avec des envois de message au sein du contexte. Ceci va nous amener à présenter deux solutions duales que nous comparons en fin de section.

Prenons l'exemple de la fonction de normalisation où chaque type de porte s'occupe de sa normalisation en créant les portes NonEt et les connexions correspondantes dans le nouveau circuit. Si le circuit normalisé ainsi généré peut directement être simulé, sa visualisation graphique s'avère en revanche plus problématique puisque la position graphique des portes NonEt créées est indéterminée à ce niveau. Or, ce problème peut facilement être résolu en ajoutant un traitement supplémentaire à la normalisation qui consiste à placer automatiquement les portes NonEt du circuit généré en fonction de la position courante des portes à normaliser.

A priori, un tel traitement requiert de la part des objets portes fonctionnant dans le contexte de normalisation d'exploiter les méthodes de placement des objets NonEt prévues pour le contexte d'édition.

Schématiquement, ce problème peut être représenté comme à la figure 3.40. Cette figure illustre à partir d'un exemple le besoin d'intéraction inter-objets inter-contextes entre une porte Ou et une des portes NonEt faisant partie de sa forme normalisée. Elle montre qu'après avoir récupérer sa position courante par une appel de méthode inter-contextes (`get-location`), la porte Ou utilise celle-ci pour déterminer la position de la porte NonET en invoquant sa méthode (`move-to`).

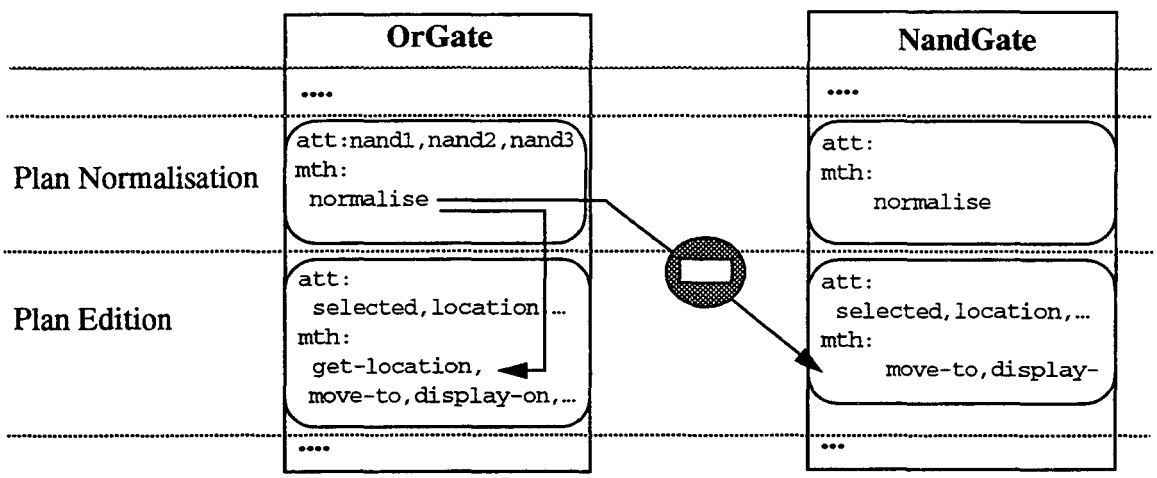


figure 3.40 : Besoin d'interaction inter-objets inter-contextex

L'envoi de messages entre objets de différents contextes n'étant pas autorisé en CROME, nous proposons de résoudre ce problème d'une autre manière en combinant les possibilités d'intéraction intra-objet inter-contextes avec les interactions entre objets au sein du même contexte. Deux solutions peuvent alors être envisagées suivant que l'on privilégie un type d'intéraction ou l'autre. Les prochains paragraphes sont consacrés à l'examen séparé de chaque solution.

Le premier schéma de solution est présenté à la figure 3.41. Dans cette solution, on commence par effectuer un envoi de message au sein du contexte pour ensuite procéder à un changement de contexte. Appliqué à l'exemple, cela conduit l'objet porte Ou à intéragir dans le contexte de normalisation avec un objet porte NonEt de façon à lui communiquer sa position. Ce dernier peut ensuite facilement mettre à jour sa position en invoquant sa méthode `move-to` associée au contexte d'édition. Dans le cas présent, la mise en place de cette solution requiert d'ajouter une nouvelle méthode `move-near` dans la partie fonctionnelle associée à la classe `NandGate` pour le contexte de normalisation.

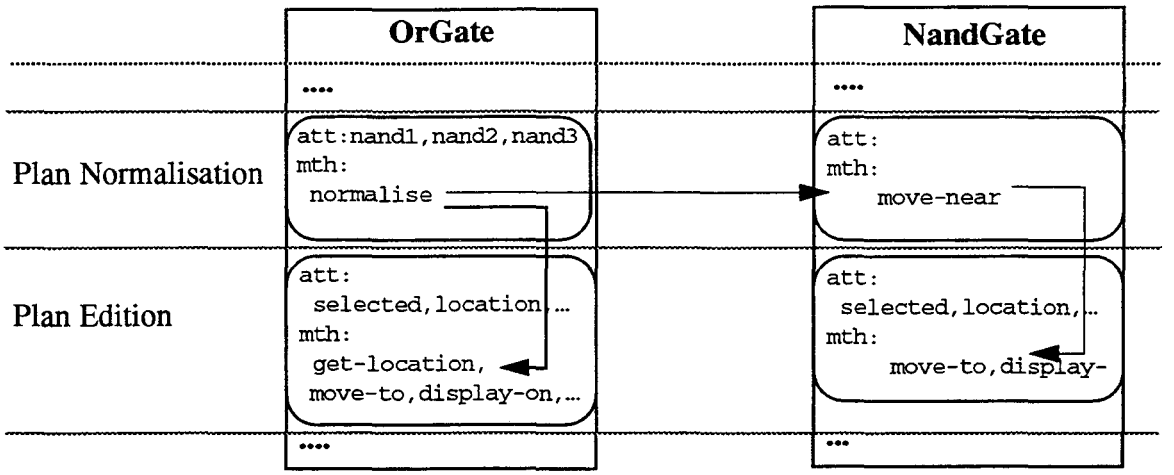


figure 3.41 : Solution en utilisant l'inter-objet entre premier

Le second schéma de solution est dual du premier dans le sens où on accomplit en premier un changement de contexte avant de réaliser un envoi de message dans ce nouveau contexte. L'application de ce schéma à l'exemple est représentée par la figure 3.42. Ici, la porte Or invoque depuis le contexte de normalisation une méthode du contexte d'édition conçue pour déterminer le placement de la porte Nand à positionner. La méthode en question peut alors facilement déterminer la position de la porte Nand passée en paramètre en lui envoyant le message correspondant dans le contexte. Comparée à la solution précédente, nous pouvons constater que la mise en oeuvre de cette solution ne nécessite pas la même modification. Dans le cas présent, on est amené à introduire une nouvelle méthode au niveau de la partie fonctionnelle associée à la classe Or dans le contexte d'édition. Par ailleurs, cette solution nécessite un envoi de message de moins puisque le rapatriement des coordonnées n'est pas nécessaire.

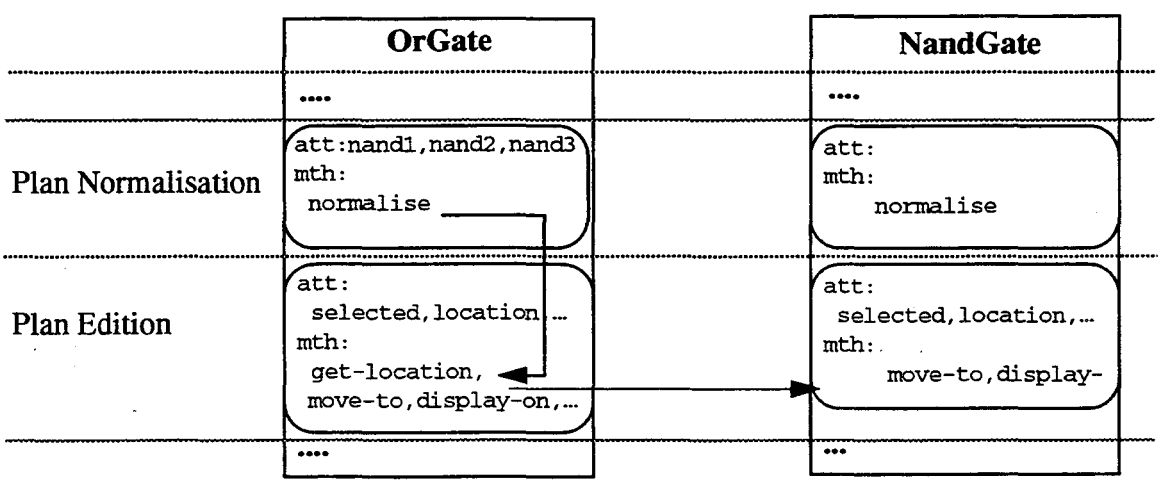


figure 3.42 : Solution en utilisant l'intra-objet en premier

Comme on peut le constater sur les deux figures précédentes, chaque solution permet d'aboutir de façon indirecte au resultat escompté. Quelle solution est la mieux adaptée ? Sous l'angle de la modularité, il est évident que la première solution est plus modulaire puisque la modification requise reste localisée dans le contexte client. Du point de vue conceptuel en revanche, la seconde solution semble plus appropriée car le traitement de placement automatique relève davantage du contexte d'édition que du contexte de normalisation. De plus,

il est facile d'imaginer que ce même traitement pourra aussi servir dans un autre contexte, celui d'optimisation par exemple pour réaliser le placement automatique des portes générées.

De manière générale, nous retiendrons que le besoin d'envoi de message entre contextes peut se ramener à l'un des deux schémas de solutions présentés ici. Même si ces solutions passent par des ajouts au code existant, elles ont le mérite de limiter le couplage entre les plans et de réduire les risques d'effets de bord entre les contextes.

3.10 Propriétés d'une conception en CROME

Dans cette partie, nous étudions les propriétés qui découlent des notions et principes proposés par CROME. Nous commençons par expliciter un gain de modularité pour les objets et une modularité de contextes. Nous montrons ensuite que la démarche tend à favoriser la réutilisation des classes ainsi que leur structuration pour la conception de chaque fonction. Nous terminons cette partie en mettant en évidence une forme d'extensibilité liée aux fonctions que permet le cadre. Dans chaque section, nous insistons également sur la préservation des propriétés de l'approche orientée objet.

3.10.1 Modularité

La démarche de description par plans des objets permet un gain important en modularité par rapport à l'approche objet classique. Ceci va nous amener à présenter un augmentation de modularité au sein des objets eux-mêmes à partir de leur description structurée en parties fonctionnelles. Cette modularité fine se trouve synthétisée au niveau supérieur du système entier par sa structuration en contextes. En dernier lieu, nous montrons que ce gain de modularité est obtenu tout en maintenant celle plus classique liée aux objets et aux classes. Cette préservation permet de bénéficier conjointement de la modularité liée aux objets et de celle liée aux contextes.

3.10.1.1 Modularité intra-objet

La description par plans conduit à un découpage de la description des objets d'une classe en plusieurs parties fonctionnelles qui sont conçues séparemment. Ce découpage donne lieu à une modularité interne aux objets et aux classes. En effet, les parties fonctionnelles peuvent être perçues comme des modules au sein des objets selon les caractéristiques suivantes:

Premièrement, une partie fonctionnelle regroupe la définition des attributs et des méthodes de l'objet qui sont liés à la même fonction. Les facilités qui découlent de ce regroupement logique sont caractéristiques d'une approche modulaire. Chaque partie fonctionnelle formant ainsi un tout cohérent, elle peut être comprise isolément des autres parties fonctionelles.

Deuxièmement, comme tout module, une partie fonctionnelle est caractérisée par un aspect interne et un aspect externe. Dans le cas présent, l'aspect interne est formé des attributs et de l'implantation des méthodes. Cet aspect est strictement encapsulé vis à vis des autres parties fonctionnelles de l'objet. Quant à l'aspect externe constitué ici de l'ensemble des méthodes, il est accessible aux autres parties fonctionnelles. Cette encapsulation des parties fonctionnelles procure au sein de l'objet des propriétés classiques. Elle permet ainsi de rendre une partie fonctionnelle cliente d'une autre indépendamment de son implantation. Elle permet de plus à

une partie fonctionnelle de rester seule responsable de sa cohérence interne. Enfin, une partie fonctionnelle peut être conçue sans tenir compte des autres parties fonctionnelles en particulier vis à vis de l'homonymie.

La modularité qui découle classiquement de l'héritage est également maintenue pour les parties fonctionnelles. Ainsi, la modification d'une partie fonctionnelle associée à une classe possédant des sous-classes n'a d'incidence que sur les parties fonctionnelles de ces dernières situées dans le même plan. Par exemple, compléter la partie fonctionnelle de la classe `Gate` dans le plan édition avec une nouvelle méthode fait automatiquement bénéficier les parties fonctionnelles des sous-classes de ces nouvelles caractéristiques. Elle n'a en revanche aucune incidence sur les autres classes, `Pad`, `PadIn` et `PadOut`, qui sont indépendantes de `Gate`.

3.10.1.2 Modularité de contexte

Les plans apportent une modularité de contextes. Cette modularité provient des caractéristiques suivantes.

Un plan permet en effet de circonscrire toutes les descriptions des objets spécifiques à un contexte, séparemment de ceux destinés aux autres contextes:

- Une modification dans l'implantation d'un contexte n'affecte pas le fonctionnement des objets dans les contextes qui en sont indépendants.

- L'ajout d'un nouveau contexte ne remet pas en cause ceux qu'il n'utilise pas. Ceci reste vérifié tant que le nouveau contexte respecte les règles d'utilisation des attributs partagés de la description de base.

- Chaque contexte peut être conçu sans tenir compte des choix d'implantation retenus pour les autres. Cette caractéristique provient de l'indépendance de conception obtenue pour les parties fonctionnelles.

Les contextes derminent autant de lieux de communications entre les objets, programmés isolément dans les plans correspondant. La localité des communications entre les objets accroit également cette modularité. Selon cette localité, un objet peut seulement être client d'autres objets participant au même contexte. Dans ces conditions, une modification apportée à ces derniers pour le contexte, un changement de spécification notamment, n'a pas d'incidence sur les descriptions de ce même objet pour les autres contextes.

Enfin, nous avons montré à la section 9 comment les problèmes d'articulation entre contextes sont résolus localement au sein même des objets du référentiel. Ce qui structure le couplage entre contextes.

3.10.2 Réutilisabilité

Au fil des sections précédentes, nous avons vu que la démarche adoptée par CROME conduit à rapporter chaque fonction au même ensemble de classes déterminées préalablement. Cette démarche encourage la réutilisation de la structuration en classes du référentiel pour l'élaboration de chaque fonction. Cette forme de réutilisation est différente de celle qui est pratiquée classiquement où les classes sont surtout réutilisées pour en construire de nouvelles par composition et par héritage[1] [Johnson 88].

Nous attirons l'attention sur le fait que cette réutilisation n'est pas l'apanage de CROME. Celle-ci peut être obtenue en fusionnant les descriptions relatives à chaque fonction dans le même ensemble de classes. Cependant, la complexité des classes qui en résulte et l'absence de moyens pour maitriser cette complexité ne favorise pas en général ce genre de réutilisation.

Dans les prochaines sous-sections, nous établissons la généricité du plan de base par rapport aux fonctions. Ceci va nous amener à expliciter successivement le caractère générique des différents éléments déterminés par ce plan.

Réutilisation des classes du référentiel

Dans CROME, la conception de chaque fonction est centrée sur le partitionnement en classes du référentiel. Le fait de réutiliser ce partitionnement pour chaque fonction présente les intérêts suivants:

Tout d'abord, cela limite les efforts de conception lors de l'élaboration d'une fonction puisqu'il n'y a pas lieu dans ces conditions de trouver de nouvelles classes, ni de gérer une organisation liée à ces classes. Le nombre de décisions à prendre est de cette façon réduite. Il suffit de se concentrer uniquement sur l'enrichissement des classes fournies par le plan de base dont le partitionnement est identifié une seule fois.

En second lieu, l'ajout d'une nouvelle fonction complique au minimum la structure du système car cela n'entraine pas l'ajout de nouvelles classes et donc de nouveaux liens d'héritage. Le nombre de classes et de liens d'héritage existant dans la description du système est de cette manière limité.

Enfin, cela facilite aussi la maintenance et la compréhension à un niveau global du système car il n'y a qu'un seul ensemble de classes à connaitre pour l'ensemble des fonctions (moyennant des classes locales spécifiques à chaque fonction). Les conséquences d'une modification de l'ensemble des classes sont plus faciles à localiser parmi les fonctions. L'articulation entre les fonctions devient également plus facile à appréhender.

En contrepartie, ce centrage sur les mêmes classes se fait au détriment d'une certaine liberté de conception au niveau de chaque fonction. Cela oblige en effet à adhérer à un découpage en classes du domaine, sans avoir la possibilité de remettre en cause certaines décisions préalables, voire d'enrichir ce découpage localement à une fonction. De ce point de vue, CROME fait l'hypothèse et le pari que la structuration en classe convient pour toutes les fonctions. Cependant, dans la mesure où le type de système envisagé présente une organisation comprenant plusieurs fonctions centrées sur un même référentiel d'entités, cette hypothèse nous parait fondée. En revanche, il est indéniable que CROME n'est pas adapté à la conception de système pour lequel les fonctions ne partagent aucun référentiel d'entités.

Réutilisation de hiérarchie d'héritage

Les plans fonctionnels permettent la réutilisation de la hiérarchie d'héritage du plan de base pour décrire le code spécifique à chaque fonction. Au sein d'un plan fonctionnel, cette

1. Signalons malgré tout que cette forme de réutilisation plus classique reste possible comme nous allons le voir par la suite.

réutilisation est obtenue en associant des parties fonctionnelles aux classes de la hiérarchie, les relations d'héritage étant implicitement préservées entre ces descriptions spécifiques.

La capacité à réutiliser la hiérarchie d'héritage pour chaque fonction permet de prendre en compte des besoins de factorisation, de généricité et d'abstraction propres à chacune d'elle. Nous en avons montré un aperçu dans plusieurs sections. Soulignons toutefois que les possibilités offertes à cet égard dans un plan fonctionnel sont fortement liées à la hiérarchie d'héritage du plan de base[1]. Par ailleurs, cette réutilisation fournit aussi un support pour le polymorphisme classique au sein de chaque fonction.

Dans notre approche, la réutilisation de la même hiérarchie d'héritage pour plusieurs contextes est facilitée par la localité de l'héritage et l'existence des parties fonctionnelles. La localité de l'héritage assure que l'exploitation n'a pas d'incidence sur les descriptions des autres plans. Quant aux parties fonctionnelles, elles présentent comme intérêt de rendre explicites les caractéristiques héritées localement.

Enfin, l'enrichissement implicite des sous-classes non référencées dans un contexte (cf 3.4.3.2) est une autre caractéristique de l'approche amplifiant cette réutilisation. Cette caractéristique permet une réutilisation partielle de la hiérarchie d'héritage pour les besoins d'une fonction.

Réutilisation de graphes d'objets

Les plans fonctionnels permettent également la réutilisation de graphes d'objets du plan de base afin de supporter des graphes de collaboration spécifiques à chaque fonction. Au sein d'un plan fonctionnel, cette réutilisation est obtenue en associant des parties fonctionnelles aux classes du graphe et en exploitant par héritage ses relations pour traduire ses traitements spécifiques.

Dans notre approche, la réutilisation multiple du même graphe d'objets pour plusieurs contextes est facilitée par la localité de communications et l'existence des parties fonctionnelles. La localité des communications permet une exploitation modulaire du graphe. La façon dont le graphe supporte un contexte n'a aucune incidence sur son exploitation par les autres contextes. De leur coté, les parties fonctionnelles rendent explicites les caractéristiques qui sont accessibles par envoi de message aux autres objets du contexte.

L'intérêt de pouvoir réutiliser un même graphe en le fonctionnalisant de manière multiple est de réduire la complexité de conception. Il n'y a alors qu'un seul graphe d'objets à gérer à travers les différentes fonctions. Dans notre exemple où le graphe d'objets du plan de base représente la structure interne du circuit, le partage de ce graphe permet de prendre en compte de façon immédiate les changements apportés lors de l'édition du circuit au niveau de la simulation, de la normalisation et de l'optimisation.

---

1. Pour une fonction, il n'est pas possible d'obtenir une factorisation entre des classes ne possédant aucune sur-classe commune. Ce qui nécessiterait la possibilité d'établir des relations d'héritage entre parties fonctionnelles. Cette limitation peut apparaitre à première vue comme une restriction du cadre. Parmi les multiples expériences de développements menées en CROME, nous n'avons jamais été confrontés à des factorisations de ce genre.

Enfin, nous avons montré en section 3.8.1 et 3.8.2 comment CROME facilite la réutilisation du graphe de collaboration factorisé dans le plan de base.

3.10.3 Extensibilité

CROME sait prendre en compte deux formes d'extensibilité bien distinctes qui sont:

- l'ajout de nouveaux types d'objets au référentiel dans l'objectif de les faire intervenir dans des contextes existants (enrichissement du référentiel).

- l'ajout de nouveaux contextes dans l'objectif d'y faire participer les objets existants du réferentiel (enrichissement fonctionnel de ces objets).

Pour chaque forme d'extensibilité, CROME fournit une notion appropriée.

Ainsi, l'introduction d'un nouveau type d'objets s'accomplit par enrichissement de la hiérarchie de classes du plan de base, en y insérant une nouvelle classe. Deux aspects sont à prendre en considération. En premier lieu, il s'agit d'étudier si cette classe peut être obtenue par sous-classement d'une classe existante. En second lieu, le plan de base ne servant qu'à déterminer la description de base des objets, il s'agit d'enrichir cette classe dans les plans fonctionnels correspondant aux contextes où ses objets prennent part. Pour chaque plan fonctionnel concerné, cela conduit à y ajouter une partie fonctionnelle associée à cette nouvelle classe. Lors de la définition de cette partie fonctionnelle, l'héritage local au plan peut être mis à profit de façon à exprimer uniquement ce qui lui est spécifique dans le contexte.

Il est à noter que, grâce à l'enrichissement implicite, la nouvelle classe n'a pas forcément besoin d'être enrichie explicitement dans tous les plans fonctionnels correspondant aux contextes où ces objets interviennent. En effet, dans certains situations, la partie fonctionnelle de la sur-classe héritée localement par la nouvelle classe peut directement convenir pour le contexte correspondant, éliminant ainsi le besoin de l'enrichir explicitement.

En plus de l'enrichissement implicite, on dispose d'une autre facilité pour intégrer les nouveaux objets aux contextes existants. Il s'agit du polymorphisme classique qui est préservé pour les contextes. Selon cette caractéristique, le code d'une partie fonctionnelle peut contenir des envois de message correspondant à des méthodes introduites par un classe du plan de base possédant des sous-classes. Ce code reste alors applicable pour toute nouvelle sous-classe de cette classe. Par exemple, la partie fonctionnelle associée à la classe Circuit pour le contexte d'édition contient des méthodes faisant appel aux méthodes introduites par la classe Gate (méthode d'affichage du circuit par exemple). La prise en compte au niveau de ces méthodes de nouveaux types de portes (c-a-d instance d'une sous-classe de `Gate`) est alors obtenue sans nécessiter leur modification.

La prise en compte d'un nouveau contexte se traduit par l'introduction d'un nouveau plan fonctionnel qui vient se juxtaposer aux plans fonctionnels existants. Pour élaborer ce nouveau plan, le plan de base est le seul à connaitre (sauf si le nouveau contexte dépend d'éléménts fournis par d'autres contextes). Cette connaissance est requise afin de déterminer quelles sont les classes à enrichir et les relations pouvant être mises à profit pour supporter le nouveau contexte.

4

Application de CROME au langage ROME

4.1 Introduction

Au cours du chapitre précédent, nous avons vu les principes de CROME comme cadre programmatique. Dans ce chapitre, nous présentons leur application à ROME pour aboutir à un langage de programmation par contexte d'objets. ROME dispose à la base de capacités de représentations multiples par point de vue d'objets, présenté dans le premier chapitre. Nous proposons ici d'exploiter et d'enrichir ces capacités pour passer de la notion de point de vue à la notion de contexte. Dans un premier temps, nous allons systématiser cette représentation au travers de la notion de plan ce qui donnera lieu à l'élaboration de graphes d'héritage multiple complexe. Dans un deuxième temps, nous allons utiliser les possibilités de sélection de points de vue sur un tel graphe pour obtenir l'héritage modulaire. Enfin, la notion de contexte sera rendue compte en appliquant les principes de sélection retenus à l'étape précédente à un ensemble d'objets. La mise en oeuvre de ces différents principes est rendue possible en mettant à profit les capacités de méta-programmation disponibles en ROME ce qui va nous conduire à spécialiser certains mécanismes de base pour les contextes.

Le reste de ce chapitre est structuré comme suit. Dans une première partie, nous rappelons les notions de base du langage ROME en nous focalisant sur les caractéristiques qui vont servir à cette application. La seconde partie expose les principes pour traduire les contextes à partir des différentes notions proposées. La contextualisation d'objets qui donne corps à la notion de contexte est ensuite présentée. La dernière partie détaille l'implantation méta-programmée des contextes.

4.2 Présentation générale de ROME

ROME [Carré 89] est un langage orienté objet basé sur le paradigme classe/instance. Il a été développé pour expérimenter des nouvelles notions comme la représentation multiple et évolutive des objets avec point de vue (cf 2.3.3.2). ROME est défini par un noyau métacirculaire inspiré de ObjVLisp [Cointe 87]. Implanté à l'origine en LeLisp, il a été transposé récemment en Scheme ce qui a permis d'améliorer l'expression des points de vue et d'ajouter de nouvelles capacités réflexives[1], principalement opératoires, au noyau métacirculaire. C'est cette version qui nous servira de référence tout au long de ce chapitre.

4.2.1 Notions de base

Structure des objets

Chaque objet ROME est caractérisé par ses attributs, ses méthodes. Les attributs sont des caractéristiques privées de l'objet qui permettent de contenir n'importe quel type de données (type de base, référence à d'autres objets ROME). Les attributs en ROME sont traités comme des variables d'instances "à la Smalltalk". L'accès en lecteur et en écriture des attributs s'effectue respectivement à l'aide des opérations `(? <nom-attribut>)` et `(<- <nom-`

1. Un aperçu des capacités réflexives du noyau méta-circulaire est donné dans la section 4.2.2.

attribut>).

Les méthodes sont des caractéristiques de l'objet qui définissent son comportement. Une méthode est définie par un couple (<nom-methode> <lamba-expression>). Il est possible de définir une méthode abstraite en utilisant le mot clé #abstract pour le corps de la lambda-expression. Les méthodes sont appliquées par envoi de message qui s'exprime syntaxiquement par: (<objet> <nom-methode> <arguments>). Deux types de méthodes sont distingués: les méthodes publiques et privées[1]. Les méthodes publiques déterminent le protocole externe de l'objet accessible aux autres objets par envoi de message. Les méthodes privées sont seulement applicables par l'objet lui-même. L'objet peut appliquer ses propres méthodes, publiques et privées, en utilisant un self-message: (self <nom-methode> <arguments>). Ici, self est une pseudo-variable qui designe l'objet courant. Celle-ci ne peut apparaitre que dans le corps des méthodes.

Classes

Les objets sont décrits par des classes. Les classes sont elles-mêmes des objets et sont donc décrites par d'autres classes (les méta-classes). ROME distingue à la base deux types de classes: les classes de représentation (dont les classes abstraites) et les classes d'instanciation. Une classe de représentation décrit seulement des modèles d'objets; une classe d'instanciation peut générer de nouveaux objets (leurs instances) correspondant au modèle qu'elle décrit. Toutes les instances d'une même classe ont les mêmes caractéristiques (attributs, méthodes). Une instance est créée en envoyant le message new à cette dernière. De manière uniforme, une classe est définie en envoyant le message new à une métaclasse. Il existe deux métaclasses standard appelées R-CLASS et I-CLASS. R-CLASS est la métaclasse standard des classes de représentation dont les classes abstraites, I-CLASS étant celle des classes d'instanciation.

Voici la définition d'une classe Figure (abstraite) et d'une sous-classe Rectangle (instanciable) en ROME illustrant tous les points introduits précédemment:

```
(define Figure
  (R-CLASS 'new
    'supercl '(OBJECT)                    ; sur-classe
    'attributes '(origin color)           ; attributs
    'privmth '()                          ; méthodes privées (aucune ici)
    'publmth '(                           ; méthodes publiques
      display (lambda () #abstract)       ; méthode abstraite
      erase (lambda ()
              (let ((oc (? 'color))
                (self 'color= 'black)
                (<- 'color oc))
      color= (lambda (ncol) (<- 'color ncol) (self display)
      move (lambda (loc)
              (self 'erase)
              (<- 'origin loc)
              (self 'display))))

(define Rectangle
  (I-CLASS 'new
    'supercl '(Figure)
```

1. qui remplace la distinction sélecteurs/méthodes de ROME dans sa version initiale.

```
'attributes '(corner)
'privmth '(
  resize (lambda (ncorn)
               (self 'erase)
               (<- 'corner ncorn)
               (self display)))
'publmth '(
  display (lambda ()
            (display-rect (? 'origin) (? 'corner) (? 'color))
  grow (lambda (n) (self 'resize ((? 'corner) 'plus n))
  shrink (lambda (n) (self 'resize ((? 'corner) 'minus n))))
```

La classe `Figure` étant abstraite (elle comporte notamment une méthode abstraite `display`), elle est créée à partir de la méta-classe `R-CLASS` en lui envoyant le message `new`. La sur-classe, les attributs et les méthodes de cette classe sont spécifiées comme des arguments de ce message (sous forme de P-List). La classe `Rectangle` étant génératrice d'instance, elle est créée en envoyant le message new à `I-CLASS`. Un objet représentant un rectangle peut être instancié en envoyant le message `new` à cette classe. Par exemple:

```
(define r1 (Rectangle 'new))
```

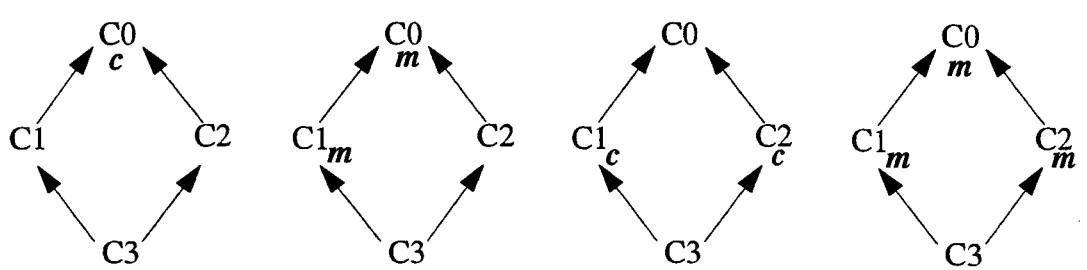
On peut alors communiquer avec cet objet à travers la variable r1 de la façon suivante:

```
(r1 'grow 10)
```

Héritage

Les classes sont organisées dans un graphe d'héritage multiple. Chaque classe est au moins une sousclasse d'une autre classe dont elle hérite les descriptions moyennant les redéfinitions. Toute classe est au minimum sous-classe de la classe la plus générale `OBJECT` qui est abstraite et définit les caractéristiques communes à tous les objets ROME. Par rapport à ses sur-classes, une classe peut ajouter de nouveaux attributs et méthodes mais aussi masquer les méthodes héritées quelles soient publiques ou privées (il n'y a pas de redéfinition possible pour les attributs). Lorsqu'une méthode est redéfinie, l'ancienne définition reste accessible en incluant l'opération `(super)` dans la nouvelle définition. Les arguments passés à la super-méthode sont ceux fournis à l'appel de la méthode contenant le super.

Dans le cas de l'héritage simple, le choix d'une méthode correspondant à un envoi de message est résolu en retenant celle la plus affinée à partir de la classe d'instanciation et en remontant le long des sur-classes. Dans le cas de l'héritage multiple, la stratégie adoptée par ROME peut être classée parmi les stratégies graphiques par opposition aux stratégies linéaires (cf 2.3.3.2). Celle-ci est basée sur les principes illustrés par les graphes d'héritage suivant[1]:



(1) (2) (3) (4)

1. Pour les attributs, les principes d'héritage se réduisent à (1) et (3) .

figure 4.1 : Principes d'héritage multiple en ROME

(1) Les caractéristiques (attributs, méthodes) accessibles suivant plusieurs chemins d'héritage sont héritées une seule fois. Les instances de `C3` disposent donc d'une seule occurence de la caractéristique `c`.

(2) Une classe qui redéfinit une méthode masque celle héritée (**principe d'affinement**). Les instances de `C3` ont une seule méthode `m` dont la définition est celle de `C1`.

(3) Il n'y a pas de conflit d'héritage entre caractéristiques homonymes héritées de classes indépendantes (**principe d'indépendance**). Des classes sont indépendantes si elles ne sont pas sous-classe l'une de l'autre. Les instances de `C3` ont donc deux caractéristiques `c`. La sélection de point de vue intervient pour distinguer entre la caractéristique `c` définie par `C1` et celle définie par `C2`.

(4) Affinement multiple d'une même méthode héritée d'une sur-classe commune. Par (1) et (2), les instances de C3 ont une seule méthode `m` avec plusieurs définitions associés. Le choix d'un affinement particulier passe par une sélection de point de vue.

Points de vue

Les points de vue en ROME permettent de privilégier une partie de la description d'un objet relativement à une classe. Un point de vue définit une sous-ensemble des classes intervenant dans la description de l'objet de la façon suivante:

- soit l'ensemble de représentation d'un objet `O`, `Rep(O)`, l'ensemble de ses classes de représentation directe[1] et leurs sur-classes;

- soit l'ensemble des classes dépendantes d'une classe `C`, `Dep(C)`, l'union de ses superclasses, de ses sousclasses et d'elle-même;

alors le point de vue d'une classe `C` sur un objet `O` se définit par

$$Pdv(O, C) = Rep(O) \cap Dep(C)$$

La figure 4.2 montre sur le même graphe d'héritage les sous-ensembles de classes (zone grisée) correspondant respectivement aux point de vue de la classe C1 (a) et celui de la classe C2 (b) exprimée pour une instance `unC3` de la classe C3.

1. cf présentation de la représentation multiple de ROME en 2.3.3.2

(a)

Pdv(unC3, C1)
= Rep(unC3) ∩ Dep(C1)
= {C3, C1, C2, C0} ∩ {C3, C1, C0}
= {C3, C1, C0}

(b)

Pdv(unC3, C2)
= Rep(unC3) ∩ Dep(C2)
= {C3, C1, C2, C0} ∩ {C3, C2, C0}
= {C3, C2, C0}

figure 4.2 : Points de vue sur un graphe d'héritage

De manière générale, chaque classe du graphe de représentation d'un objet peut servir à spécifier un point de vue sur celui-ci.

Au niveau du langage, le programmeur a la possibilité d'exprimer des points de vue à l'aide de l'opérateur `as`. Cet opérateur peut être associé à un envoi de message et de façon interne à un appel de méthode (self-message), à un appel à la super-méthode (super), et aussi à un accès d'attributs. Par exemple, pour envoyer le message `m` à un instance de `C3` sous le point de vue de `C2`, on écrira l'expression suivante :

```
(unC3 'as C2 'm)
```

L'association d'un point de vue pour l'accès à une caractéristique d'un objet permet de choisir les caractéristiques définies dans une classe faisant partie du point de vue, tout en respectant le principe d'affinement pour les méthodes. Cela permet en particulier:

- de faire un choix entre des caractéristiques de même nom définies sous des points de vue différents. Dans le cas (3) de la figure 4.1, l'expression précédente (en remplacant `m` par `c`) conduit à l'activation de la méthode `c` telle que définie par `C2`.

- de choisir un définition de méthode parmi plusieurs. Dans le cas (4) de la figure 4.1, l'expression précédente conduit à l'activation de la méthode `m` telle que définie par `C2`.

Il convient de préciser que:

- l'expression d'un point de vue ne permet jamais de contourner le principe d'affinement. Ainsi, dans le cas (2) de la figure 4.1, l'envoi du message `m` à une instance de `C3` sous le point de vue `C2` résulte en une sélection de la version affinée de `m` définie dans `C1` et non celle de `C0`. Un point de vue n'est donc pas une restriction du graphe d'héritage.

- une sélection de point de vue trop générale (resp. spécifique) peut maintenir l'ambiguïté. Pour les cas (3) et (4) de la figure 4.1, le choix du point de vue de `C0` (resp. `C3`) pour un instance de C3 ne permet pas de désambiguiser l'accès aux caractéristiques `c` et `m`. Il faut alors préciser un point de vue plus spécifique (resp. plus général) dans ce cas.

En plus des points de vue explicites donnés par le programmeur, des points de vue implicites sont automatiquement appliqués par ROME. Ces points de vue implicites viennent compléter les points de vue explicites pour résoudre les ambiguïtés. Ils sont établis dans deux cas de figure:

- pour un envoi de message à un objet o sans as explicite. Ce cas mène implicitement à appliquer le point de vue de sa classe d'instanciation[1].

- pour chaque application de méthode m interne à o (self-message). Dans ce cas, le point de vue la classe où est trouvée m[2], est retenu.

Les points de vue explicites et implicites associés à un objet au cours de son exécution sont exploités par le lookup pour résoudre les références à des caractéristiques ambigües. Cette exploitation se fait en considérant les points de vue (explicites et implicites) les plus récemment spécifiés[3] en priorité et ceci jusqu'à résolution de l'ambiguïté. Pour plus de détails sur l'exploitation des points de vue, voir [Carré 89].

Montrons à travers un exemple l'établissement des points de vue implicites et la prise en compte des points de vue associés à un objet à partir de deux exemples basés sur le graphe d'héritage précédent dans lesquels les méthodes m1 et m2 sont ajoutées comme indiqué par la figure 4.3.

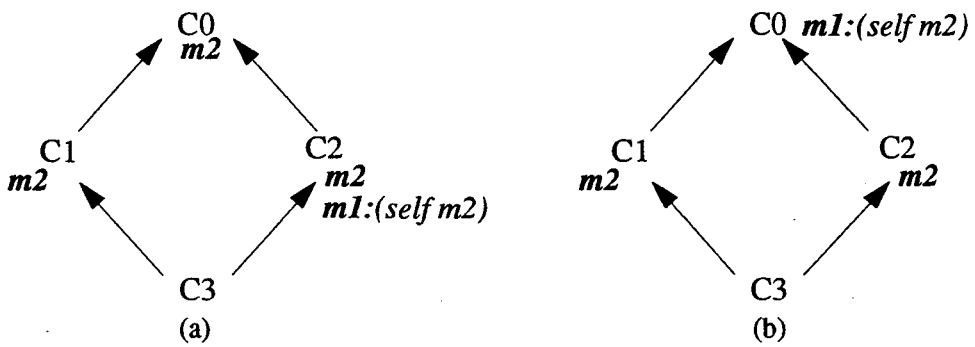


figure 4.3 :Exemple de points de vue implicites

Pour le premier exemple représenté par le graphe (a) de la figure 4.3, considérons l'envoi du message m1 à une instance de C3 sans spécifier de point de vue particulier. Selon la première règle, le point de vue implicite de la classe d'instanciation, celui de C3, est appliqué pour résoudre ce message. La définition de m1 correspondante est trouvée sans ambiguité dans C2. Selon la seconde règle, un second point de vue implicite correspondant à C2 vient s'ajouter au précédent pour l'application de la méthode m1. Ces deux points de vues sont alors exploités pour résoudre la référence à m2 exprimée dans m1, ce qui mène ici à choisir la définition fournie par C2. Ce premier exemple montre que les points de vue implicite permettent de résoudre les ambiguités en cas d'absence de point de vue explicite.

Pour le second exemple représenté par le graphe (b) de la figure, considérons l'envoi du message m1 à une instance de C3 sous le point de vue de C1. La définition de m1 est trouvée sans ambiguité dans C0. Pour l'application de cette méthode, le point de vue explicite précédent est

1. ou de Object.
2. sans ambiguité sinon la référence à m doit être résolue par point de vue explicite.
3. relativement à l'application des méthodes.

complété selon la seconde règle par un point de vue implicite correspondant à C0. La référence à m2 exprimée dans cette méthode est alors résolue en considérant ces deux points de vue, l'ambiguité étant ici levée par le point de vue explicite C1. Cet exemple montre que la portée d'un point de vue ne se limite pas simplement à la résolution de l'appel de méthode pour lequel il est spécifié mais s'applique également pour les enchainements des méthodes appelés ultérieurement.

Représentation multiple et évolutive d'objets

En plus de l'héritage multiple, ROME offre la représentation multiple qui permet de lier les objets, instances d'une classe unique, à plusieurs classes de représentation qui constituent autant de descriptions différentes de celui-ci. Ce mécanisme est controlé par le principe de représentation minimale: Tout classe de représentation d'un objet est sous-classe stricte[1] de sa classe d'instanciation. Ce principe garantit qu'un objet reste toujours au minimum représentant de sa classe d'instanciation. Il assure également la cohérence d'objet en interdisant son rattachement à des classes représentant d'autres objets.

La structure et le comportement d'un objet sont déterminés par ses classes de représentations selon les principes d'héritage énoncés précédemment qui s'applique de manière uniforme pour la représentation multiple. En particulier, la recherche d'une méthode pour un objet est analogue sur le plan sémantique à celle qui serait accomplie si celui-ci avait été défini comme instance d'une sous-classe de ses classes de représention directe. Pour ces objets, le mécanisme de sélection de points de vue reste applicable aux classes de représentation et conserve la même sémantique que précédemment.

Au niveau du langage, les objets à représentation multiple et évolutive sont définis par la classe R-OBJECT, sous-classe abstraite de OBJECT. Cette classe introduit notamment les deux opérateurs permettant d'ajouter ou d'enlever des classes de représentation de l'objet conformément au principe de représentation minimale.

Rattacher un objet à une classe se fait en lui envoyant le message:

```
(<objet> 'r+ <classe>)
```

Implicitement, l'`<objet>` devient aussi représentant de toutes les superclasses de cette `<classe>`.

Pour détacher un objet d'une classe, le message à lui envoyer est:

```
(<objet> 'r- <classe>)
```

Ce message supprime les liens de representation dependant de `<class>` mais l'`<objet>` reste représentant de toutes les sur-classes de celle-ci.

4.2.2 Aspects réflexifs

4.2.2.1 Réflexivité et langages ouverts

La réflexivité est une méthodologie générale visant à doter un système informatique quelconque (langage, système d'exploitation, base de données, boite à outils graphiques, ...) d'une représentation d'un certain nombre de caractéristiques de sa propre architecture et de son propre comportement sur lesquels il peut raisonner et aussi apporter des modifications. Dans les

1. C1 est sous-classe stricte de C2 si aucune sur-classe de C1 n'est indépendante de C2.

langages de programmation, la réflexivité consiste à donner une représentation[1] des programmes et de son exécution dans le langage lui-même (réification) d'une part et d'autre part à rendre accessible (introspection) et modifiable de façon causale (intercession) cette représentation depuis ces mêmes programmes.

Une utilisation importante de la réflexivité dans les langages de programmation est de permettre leur ouverture [Wegner 88][Kiczales 92]. Un langage ouvert selon une approche réflexive donne la possibilité d'adapter certains de ses aspects (abstractions et choix de mise en oeuvre) au sein du langage lui-même. Le besoin d'ouvrir les langages provient du constat que ceux-ci doivent supporter de plus en plus d'applications et donc intégrer de plus en plus de fonctionnalités. Dans le cadre des langages traditionels, qui n'ont pas cette capacité d'ouverture, la prise en compte de ces fonctionnalités pose à la fois des problèmes à leurs concepteurs et à leurs utilisateurs. Ainsi, le concepteur est confronté à un dilemne car il doit anticiper le plus de fonctionnalités et besoins applicatifs possibles. De son coté, le programmeur n'a aucun moyen d'adapter ou d'étendre le langage pour mettre en oeuvre ses besoins spécifiques. Celui-ci doit alors réimplanter les fonctionnalités manquantes en utilisant souvent des contorsions et sans pouvoir véritablement les intégrer avec le reste du langage. Pour éviter ces problèmes, les langages ouverts autorisent (de façon contrôlée) leurs utilisateurs à en particulariser certains aspects et à les enrichir.

De nombreux travaux ont étudié la réflexivité dans les langages de programmation (logique, fonctionnel, objet). La réflexivité s'exprime particulièrement bien dans les langages à objets qui, grâce à leurs techniques de base (objets, message, classe, héritage, polymorphisme, liaison dynamique, généricité) assurent une bonne encapsulation des niveaux [Maes 87] et facilite l'adaptation. Dans ces langages, le principe est de représenter diverses caractéristiques de représentation, d'implantation et d'exécution par un ensemble d'objets, appelé méta-objets. Ces objets et leurs intéractions forme un protocole de méta-objets (Méta-Object Protocol). La spécialisation de ces méta-objets permet alors de particulariser les aspects de représentation et d'exécution de plusieurs objets du programme. Deux exemples de langages à objets intégrant un protocole de méta-objets particulièrement riche sont Smalltalk [Rivard 97] et CLOS [Kiczales 91].

Dans les deux prochaines sections, nous présentons quelques uns des principaux ingrédients de réflexivité structurelle et de réflexivité opératoire intégrés à ROME[2]. La section suivante illustre l'utilisation de ces ingrédient pour ajouter les capacités de représentation multiple et évolutive d'objets au langage. Ces ingrédients vont également servir par la suite pour implanter les contextes.

4.2.2.2 Réflexivité structurelle

Dans les langages à objets à base de classes/instances[3], la réflexivité de structure se ramène à décrire les entités du langage que sont les classes (et éventuellement les attributs, les

1. Lorsqu'on parle de réflexivité dans les langages de programmation, il est courant de distinguer deux types de réflexivité: structurelle et opératoire. La réflexivité structurelle s'intéresse à la représentation des programmes de l'utilisateur dans le langage lui-même, à l'aide de ses structures de données propres. La réflexivité opératoire s'occupe de la représentation des mécanismes contrôlant l'exécution du programme, représentation qui est utilisée par l'interprète lui-même.

2. dont les capacités réflexives sont relativement restreintes comparées à celles des deux langages précités.

méthodes, ...) à l'aide de ces mêmes entités. Le modèle de méta-classe, classes et instances d'ObjvLisp [Cointe 87] propose une solution homogène pour réaliser cette réflexivité structurelle. Dans ce modèle, les classes sont considérées comme des objets à part entière, capables de recevoir et d'envoyer des messages et de communiquer avec d'autres objets. Comme tout objet, les classes sont des instances d'une classe (méta-classe) qui précisent leur structure et les messages auxquelles elles peuvent répondre. Ces méta-classes pouvant à leur tour être considérées de façon identique comme des instances d'une autre classe une méta-méta-classe. ObjVLisp synthétise ce schéma d'instanciation en remarquant qu'une méta-classe est une classe de classe et que sa structure est à la base identique à celle d'une classe. Le résultat est l'introduction dans le modèle d'une méta-classe qui est instance d'elle-même.

Ce modèle a été repris dans plusieurs langages à objet basés sur le modèle classe/instance dont entre autre CLOS[1]. Smalltalk repose également sur un modèle méta-circulaire, dont les limites sont évoquées dans [Briot 89]. Toutefois, l'architecture de Smalltalk reste assez ouverte pour y intégrer les principes d'ObjVlisp [Cointe 93][Rivard 97].

En ROME, la réflexivité structurelle est réalisée selon l'inspiration d'ObjVLisp dont les principes sont repris pour définir les classes et les méta-classe de façon méta-circulaire. Par rapport au noyau métacirculaire d'ObjVlisp, celui de ROME dissocie cependant les deux fonctionnalités fondamentale d'une classe: la fonctionnalité de description (abstraction) et la fonctionnalité de création d'instance. Cette dissociation mène à un noyau méta-circulaire plus explicite que nous présentons.

Le noyau métacirculaire de base

Le noyau de base se compose de trois classes:

- `OBJECT` est la classe la plus générale, racine du graphe des classes, dont tout objet est représentant. Elle définit notamment le modèle commun à tous les objets dont l'attribut `iclass` qui lie l'objet à sa classe lors de sa création.

- `R-CLASS`, sous-classe de `OBJECT`, est la (méta)-classe de toutes les classes sachant seulement décrire des objets (classe abstraite ou classe de représentation) dont elle définit le modèle notamment: les attributs `attributes`, `publicmeths`, `privatemeths`, `supercld`.

- `I-CLASS`, sous-classe de `R-CLASS`, est la (méta)-classe de toutes les classes qui savent instancier et gérer l'exécution des objets selon le modèle qu'elles définissent. C'est elle qui définit notamment la méthode d'instanciation `new` et la méthode de résolution des messages `handle-msg` (cf partie suivante).

Schématiquement, le noyau métacirculaire se présente de la façon suivante :

---

3. La réflexivité structurelle dans les langages à prototypes est moins courante. Elle est étudiée dans [Cointe 92][Mulet93].
1. CLOS représente aussi les méthodes par des objets.

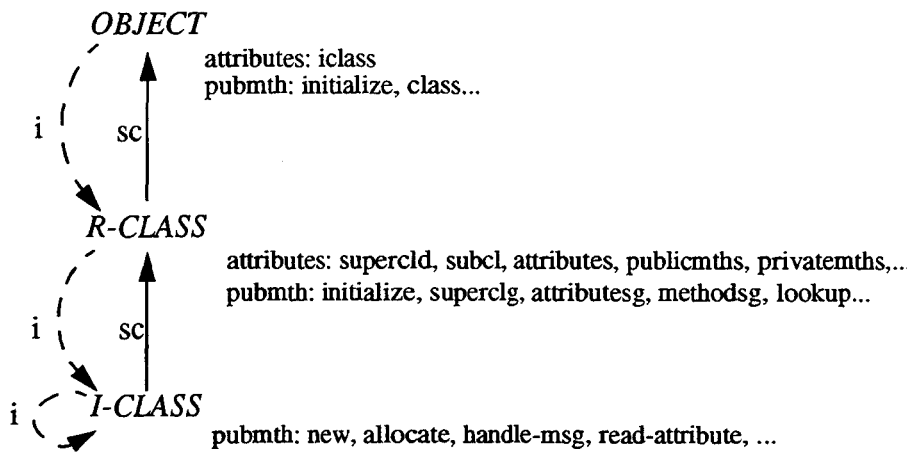


figure 4.4 : Noyau méta-circulaire de ROME

Ces trois classes ne peuvent se définir l'une sans l'autre[1]. OBJECT qui est la classe la plus générale est abstraite et donc instance de R-CLASS. R-CLASS définit les classes abstraites qui sont des objets particuliers; elle est donc sous-classe de OBJECT. Par ailleurs, elle est génératrice de ces classes et doit donc disposer des fonctionnalités définies par I-CLASS; elle en est donc instance. I-CLASS définit les classes concrètes qui sont des classes plus spécifiques que les classes abstraites; elle est donc sous-classe de R-CLASS. Par ailleurs, elle est génératrice de ces classes donc, similairement à R-CLASS, instance de I-CLASS, c'est à dire d'elle-même.

A partir de ce noyau, une nouvelle classe est créée comme une instance de R-CLASS, I-CLASS ou d'une de leurs sous-classes. Elle est de plus sous-classe directe ou indirecte d'OBJECT. Une nouvelle méta-classe est obtenue en sous-classant R-CLASS ou I-CLASS (suivant le besoin de décrire des modèles de classes abstraites ou concrètes) ou une de leurs sous-classes. Puisqu'elle est généralement amenée à créer des classes, une méta-classe est la plupart du temps instance de I-CLASS[2].

4.2.2.3 Réflexivité opératoire

Dans les langages à objets, la réflexivité opératoire concerne la création et l'exécution des objets (accès aux attributs, l'interprétation des messages, ..) qu'il s'agit de rendre explicites et adaptables dans le langage. Pour introduire cette réflexivité, la plupart des travaux [Watanabe 88][Chiba 93][Mulet 93] se sont inspirés de l'approche basée sur la notion de méta-objet introduite dans 3-KRS [Maes 87] qui est un langage sans classe. Selon cette approche, chaque objet dispose d'un méta-objet qui gère les informations et réalise les traitements relatifs à sa représentation (structure d'implantation, ...) et à son exécution (l'accès à ses attributs, la façon de traiter les messages, ...). Ces méta-objets sont traités comme des objets à part entière et peuvent être définis et manipulés avec les mêmes constructions (envoi de message par exemple). En particulier, de nouveaux méta-objets peuvent être définies et associés aux objets. Ces méta-objets particuliers sont alors pris en compte par l'interprète du langage pour particulariser tel ou tel aspect de représentation et d'exécution des objets.

---

1. et sont donc construites par bootstrap (à partir d'un embryon de I-CLASS)
2. mais il est possible de décrire une méta-classe abstraite qui pourrait servir à factoriser des caractéristiques entre plusieurs méta-classes. Une telle méta-classe est à la fois instance et sous-classe de R-CLASS.

L'adaptation de cette approche par méta-objets pour les langages fondés sur le modèle classes/instances a été étudiée dans [Ferber 89b]. Deux solutions pour y intégrer les méta-objets ont ainsi pu être déterminées. Une première solution consiste à prendre la classe comme le méta-objet de toutes ses instances. Une seconde solution consiste à définir un méta-objet par instance, distinct de leur classe et instance d'une classe MetaObject (ou de l'une des sous-classes). Pour une comparaison détaillé de ces deux approches, voir [Ferber 89b].

En ROME, nous avons choisi de réaliser la réflexivité opératoire selon la première approche qui est certes moins flexible mais en revanche :

- s'avère moins complexe;
- respecte mieux la philosophie des langages à classes où les objets d'une même classe partagent toujours le même mode d'interprétation des messages et d'accès à leur structure [Cointe 90];
- s'intègre mieux avec le modèle de réflexivité structurelle dans lequel la classe détient déja toutes les informations structurelles (attributs/méthodes) de ses instances.

L'intégration de cette approche avec le noyau méta-circulaire de ROME a les conséquences suivantes:

- Tous les objets, y compris les classes et les métaclasses, étant instance d'une classe, ceux-ci possèdent implicitement et de manière uniforme un méta-objet. Les métaclasses deviennent notamment les méta-objets des classes et disposent à leur tour de méta-objets. En particulier la méta-classe `R-CLASS` a pour méta-objet `I-CLASS`.

- L'auto-instanciation de `I-CLASS`, règle le problème de la régression infinie de méta-objets. En tant qu'objet, cette classe s'admet comme son propre méta-objet. Elle constitue la frontière du schéma reflexif.

Ajoutons que le traitement des classes comme méta-objet nécessite de modifier l'interprète de base. Celui-ci est modifié de façon à faire intervenir les méta-objets dans la réalisation des mécanismes implicites (résolution de message, accès aux attributs, ...). Le recours aux méta-objets est systématique pour tous les objets et pour l'ensemble des mécanismes implicites réifiés[1].

Les deux prochaines sections présentent deux des mécanismes qui sont réifiés: l'interprétation des messages et la création d'objets. Nous avons choisi de détailler ces mécanismes car nous allons être amenés à les spécialiser par la suite pour mettre en oeuvre les contextes.

Réification du mécanisme d'interprétation des messages

Dans l'approche par méta-objets, l'interprétation d'un message reçu par un objet est délégué

1. On pourrait envisager une solution plus souple dans laquelle un contrôle plus fin sur le passage au niveau méta serait proposé. Il serait alors possible de paramétrer les conditions d'intervention du méta-objet dans la réalisation des mécanismes implicites (ces conditions étant décrites au niveau du méta-objet lui-même). Ceci permettrait par exemple de faire intervenir le méta-objet uniquement pour certaines instances ou certains aspects.

au niveau méta. Cette délégation consiste plus précisemment à envoyer le message `handle-msg` au méta-objet de l'objet receveur suivi des arguments nécessaires pour interpreter le message. La figure 4.5 illustre cette délégation dans le cas où le méta-objet correspond à la classe.

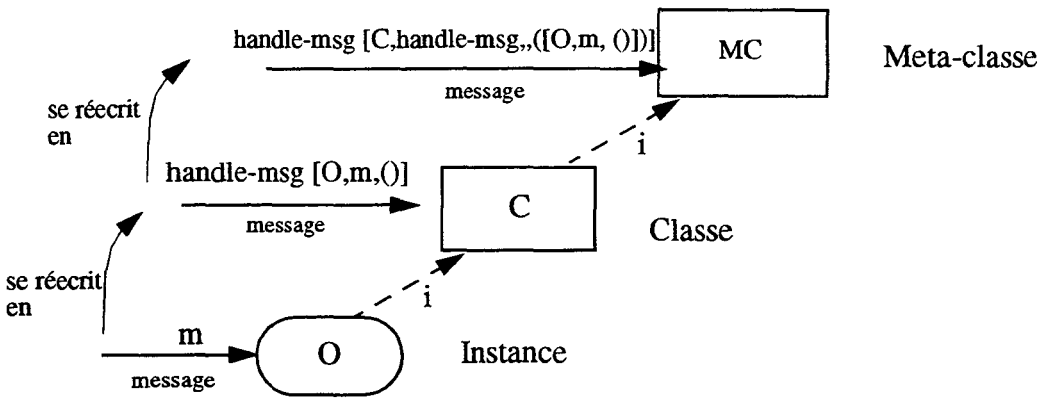


figure 4.5 : Réification de la résolution d'un message avec la classe comme méta-objet

Selon cette figure, le message `m` envoyé à l'objet `o` est transposé en un envoi de message `handle-message` à sa classe `c` avec en argument l'objet receveur `o`, le sélecteur du message et une liste d'arguments. Ce processus de délégation se poursuit ensuite dans les méta-niveaux, l'ascension s'arrêtant quand un cas de base est identifié.

Afin de mettre en place ce schéma, la primitive d'envoi de message est codée au niveau de l'interprète de la façon suivante

```
1       (define (send obj sel args)
2         (if (and (eq? sel 'handle-msg) (eq? obj (car args)))
3             (#handle-msg obj sel args)
4             ((class-of obj) 'handle-msg obj self sel args))))
```

Une analyse de ce code montre que l'envoi de message à un objet est transformé en un envoi de message à sa classe qui constitue son méta-objet (ligne 4). L'obtention de la classe de l'objet est accomplie à l'aide d'une primitive `class-of` et non d'un envoi de message pour éviter un régression infinie. Contrairement à la primitive d'envoi de message proposée dans [Ferber 89], nous ne reifions pas le message. Par ailleurs, ce code met également en évidence le cas de base stoppant l'ascension dans les méta-niveaux. (ligne 2 et 3). Ce cas correspondant à l'envoi du message `handle-msg` à un objet qui est son propre méta-objet. Dans ce cas, l'interprétation du message est traitée par la primitive `#handle-msg` de l'interprète. Le seul objet menant à ce cas dans notre noyau est la classe `I-CLASS` puisqu'elle est instance d'elle-même. Notons qu'ainsi le traitement de la résolution implantée par `I-CLASS` est appliquée à elle-même, guarantissant son auto-description[1].

La définition de la méthode `handle-msg` se trouve toujours dans la métaclasse associée à la classe du receveur. La méthode définissant le comportement standard est situé dans la méta-classe I-CLASS puisque seules les classes possédant des instances sont amenées à recevoir ce

1. En particulier, l'envoi d'un message m à I-CLASS provoque l'envoi du message handle-msg à I-CLASS avec lui-même comme receveur. Ce nouvel envoi de message satisfait la condition du test d'arrêt dans les méta-niveaux et est donc résolu par la primitive de l'interprète.

message[1]. Ce comportement standard consiste à résoudre les messages selon une stratégie d'héritage multiple par point de vue.

Techniquement, les points de vue explicites et implicites sont placés sur une pile[2] qui sert comme contrainte pour l'algorithme de lookup. Ce dernier fait intervenir la pile uniquement en cas d'ambiguités pour déterminer la définition la plus affinée sous le dernier point de vue spécifié, ou l'avant dernier si l'ambiguité persiste, ou l'avant-avant dernier et ainsi de suite jusqu'à éventuellement atteindre le fond de pile.

Dans I-CLASS, cette méthode est définie de la façon suivante[3]:

```
1    handle-msg
2       (lambda (rcvr sndr sel args)
3         (let ((cdef '())
4               (selas (eq? sel 'as))
5               (slfmsg (eq? rcvr sndr)) ;; self-message ?
6               (res '())
7
8           (if selas
9             (begin
10              (#push-pov (car args))
11              (set! sel (caddr args))
12              (set! args (cadddr args)))
13            (if (not slfmsg) (#push-pov (class-of obj))
14
15          (set! cdef (self 'lookup-mth sel slfmsg)
16          (cond ((null? cdef)
17                 (rcvr 'error "message " sel " not understood"))
18                ((list? cdef)
19                 (rcvr 'error "message " sel " ambiguous.
20                      Point of views:" cdef)
21                (else (set! res (self 'applymeth rcvr cdef sel args))
22                      (if selas (#pop-pov))
23                      res)))
```

Cette méthode prend en paramètre l'objet receveur `rcvr`, l'objet à l'origine du message `sndr`, le sélecteur de la méthode `sel` et une liste d'arguments `args`. La première partie de la méthode (ligne 8 à 13) détermine si un point de vue explicite accompagne le message. Cela revient à tester si le sélecteur correspond à l'opérateur `as`. Si c'est le cas, le point de vue rangé dans le premier argument est placé en sommet de la pile. Dans le cas contraire et si le message vient de l'extérieur, le point de vue implicite correspondant à la classe d'instanciation de l'objet est empilé.

La seconde partie (ligne 10 à 21) s'occupe de la résolution du message. Dans la lignée des propositions faites dans [Cointe90], cette interprétation est décomposée en deux phases

1. De fait, les classes instance de R-CLASS ne savent pas répondre au message handle-msg ce qui est cohérent avec leur statut de classe de représentation (de classes abstraites).
2. qui est rangé dans une variable globale de l'interpréteur. Cette pile est manipulée à travers l'opération d'empilement #push-pov et de dépilement #pop-pov.
3. En réalité, cette méthode n'est tout à fait définie ainsi de façon à éviter le bouclage infini de la résolution. En particulier, les self-messages aux méthodes lookup-mth et apply-meth sont effectués en dehors du schéma réflexif.

distinctes.

1. La recherche de la classe la plus spécialisée possédant une définition de méthode correspondant au selecteur. Cette phase est déclenchée par l'appel à la méthode `lookup-mth` correspondante (ligne 15) .

2. L'application de la méthode appartenant à la classe trouvée. Cette phase est déclenchée par l'appel à la méthode `apply-meth` (ligne 21).

La méthode `lookup-mth` standard est déterminée dans la classe `R-CLASS` et est donc héritée au niveau de `I-CLASS`. Cette méthode prend en entrée le sélecteur de méthode (`sel`) à rechercher et un booléan (`allmth`) indiquant une recherche sur toutes les méthodes (self-message) ou seulement celles qui sont publiques (message de l'extérieur). Le résultat de cette méthode est une liste contenant une référence sur la ou les classes (forcément indépendantes) possédant une définition pour le sélecteur spécifié. Elle est définie de la façon suivante:

```
1    lookup-mth
2       (lambda (sel allmth)
3         (let ((cdefs (every sel (? 'publicmth-pairs))
4           (if (and (null? cdefs) allmth)
5             (set! cdefs (every sel (? 'privatemth-pairs))))
6           (if (< 1 (length cdefs))
7             (#project cdefs)
8             cdefs)))
```

Cette méthode commence à rechercher dans une liste de couples (selecteur publique, classe) rangée dans l'attribut `publmth-pairs`[1], la ou les classes déterminant une définition pour le sélecteur passé en paramètre. Si aucune classe n'est trouvée pour le sélecteur (e.g la variable `cdefs` est une liste vide) et le paramètre `allmth` est vrai, une recherche analogue est réalisée dans une liste de couple (selecteur privée, classe) rangé dans l'attribut `privatemth-pairs`. Lorsqu'une ambiguité est détectée (e.g plusieurs classes possèdent une définition de méthode pour le sélecteur et donc la variable `cdefs` contient un liste de taille supérieur à 1), la primitive `#project` est appelée (ligne 6 et 7). Cette primitive confronte les classes passées en paramètre au contenu de la pile des points de vue et retourne une liste contenant la ou les classes qui sont incluses dans les points de vue spécifiés par la pile.

Nous donnons également la définition de la méthode `apply-meth` introduite dans `I-CLASS`. Cette méthode prend en entrée l'objet (`rcvr`) pour lequel la méthode est à exécuter, la classe contenant la définition de la méthode (`cdef`), le sélecteur (`sel`) et les arguments de l'appel (`args`).

```
1    apply-meth
2       (lambda (rcvr cls sel args)
3         (let ((res '())
4               (lmbda (cls 'method-def sel))
5               (prev-sup super)
6               (prev-self self)))
```

1. Les attribut publmth-pairs et privmth-pairs sont calculés lors de la finalisation de l'héritage de chaque classe (instance de R-CLASS et I-CLASS) à partir de ceux des sur-classes et des définitions locales de méthodes. Ils permettent de déterminer directement la classe de définition d'un sélecteur sans avoir à parcourir le graphe à l'exécution.

```
7           (if (null? lmbda)
8             (rcvr 'error "method")
9             (begin
10              (#push-pov cls)
11              (set! super
12                   (lambda supas
13                     (if (not (null? supas)) ;;;; SUPER-AS
14                       (self 'handle-supercall
15                             rcvr cdef 'as (list (cadr supas) sel args))
16                       (self 'handle-supercall rcvr cdef sel args)
17              (set! self rcvr)
18              (set! res (apply (cdef 'method-def sel) (list args))
19              (#pop-pov)
20              (set! super prev-sup)
21              (set! self prev-self)
22              res)))))
```

Cette méthode commence par vérifier que la méthode à appliquer n'est pas abstraite (c-a-d la définition de la méthode n'est pas la liste vide, ligne 7). Si c'est le cas, l'environnement pour la méthode à exécuter est configuré. Cette configuration consiste:

- à empiler un nouveau point de vue (ligne 10) correspondant à la classe où est trouvée la définition de la méthode. Ce point de vue implicite va éventuellement servir à resoudre des ambiguités lors de la prochaine recherche de méthode pour l'objet (cf `lookup-mth`).

- à lier la variable globale `super` à une nouvelle lambda (ligne 11) qui est définie de façon à traduire un appel à la super-methode par un message `handle-supercall` envoyé au méta-objet[1] (`self` en l'occurence). Cette méthode prend un paramètre supplémentaire correspondant à la classe où a été trouvée la définition courante et sert à débuter la recherche à partir des sur-classes de celle-ci[2].

- à lier la variable globale self à l'objet ayant reçu le message (ligne 12).

Ensuite, la définition de la méthode, représentée sous la forme d'une lambda, est exécutée au moyen de la primitive `apply` de Scheme (ligne 13). Après cette exécution, l'état de l'environnement précédent est restitué (ligne 15 à 17) puis le résultat de l'exécution est retourné.

Réification du mécanisme d'instanciation

En ROME, ce mécanisme est uniforme comme en ObjVLisp. Cela signifie que nous utilisons le même traitement pour créer une instance, une classe ou une métaclasse. Ce processus est basé

---

1. La résolution des appels aux super-méthodes est donc également réifié au niveau du méta-objet.
2. Cette méthode est définie dans I-CLASS en réutilisant les méthodes lookup et apply-meth comme suit (pour des questions de place, nous n'indiquons pas le traitement du as sur le super) :

```
handle-supercall
   (lambda (rcvr cls sel args)
      (let ((cdef '())
         (set! cdef (mappend (lambda (c) (c 'lookup-mth sel #f)) (? 'supercld))
         (cond ((null? cdef) (rcvr 'error "incorrect super"))
               ((list? cdef) (rcvr 'error "super call " sel "is ambiguous.
                                    Point of views:" cdef)
      (else (self 'applymeth rcvr cdef sel args)))))
```

sur la combinaison d'une méthode d'allocation (`allocate`) avec une méthode d'initialisation (`initialize`). Cette combinaison est réalisée par la méthode d'instanciation (`new`). Cette méthode prend en argument une liste de variables d'instances et leurs valeurs d'initialisation (P-list). Son code est le suivant:

```
1       new
2          (lambda l-init
3            (let ((newobj (self 'allocate)))
4              (newobj 'initialize l-att)
5              newobj)))
```

`I-CLASS` détient la méthode d'allocation standard[1] ainsi que la méthode d'instanciation standard. Il existe deux méthode d'initialisation. La première est détenue par la classe `OBJECT` et définit l'initialisation standard des objets terminaux. La seconde, détenue par `R-CLASS` et donc hérité au niveau de `I-CLASS`, définit l'initialisation standard des classes. Cette seconde méthode d'initialisation est une spécialisation de la première.

Autres mécanismes réifiés

La résolution des messages et la création des objets ne sont pas les seuls mécanismes réifés. Il existe également une réification de l'accès aux attributs des objets mise en oeuvre sur un principe analogue aux envois de messages. L'accès en lecture (resp. en écriture) à un attribut d'un objet est ainsi transformé par l'interprète en un envoi de messsage `read-attribute` (resp. `write-attribute`) à sa classe[2]. Les méthodes standards correspondantes sont définies dans `I-CLASS`. Voici à titre d'exemple, la définition de la méthode `read-attribute` de cette classe:

```
1  read-attribute
2     (lambda (o cls attname attas)
3       (let ((cdef '())
4             ((res '())))
5
6         (if (not (null? attas)) (#push-pov attas))
7
8         (set! cdef (cls 'lookup-att attname))
9         (cond ((null? cdef)
10                (o 'error "attribute " attname " does not exist"))
11              ((list? cdef)
12                (o 'error "attribute " attname " is ambiguous.
13                           Point of views:" cdef)
14              (else (set! res (#get-value o cdef (cdef 'iv-position attname)))
15                    (if attas (#pop-pov)
16                    res)))
```

1. Dans l'implantation actuelle en Scheme, nous avons fait le choix de représenter les objets par des fermetures à la manière de ce qui est fait [Mulet94]. La méthode allocate standard retourne donc une lambda liée à un environnement. La valeurs des attributs de l'objet sont stocké dans une variable de la fermeture sous forme d'une liste d'associations (A-List). Chaque association de cette liste est constituée d'une référence à une classe et d'un vecteur dont les éléments contiennent les valeurs des attributs correspondant à cette classe.

2. Cette transformation est réalisée selon un principe similaire à la transformation du super dans la méthode apply-mth. Autrement dit, des lambdas définis pour faire appel à read-attribute et write-attributes avec les bons paramètres sont liés au symbole ? et <- dans apply-mth.

Cette méthode prend en paramètre l'objet `o` sur lequel porte l'accès, la classe `cls` où est réalisé cet accès, le nom de l'attribut `attname` et un point de vue `attas` associé à l'attribut. En premier lieu, la méthode détermine (ligne 6) détermine si un point de vue explicite (ici différent de la liste vide) accompagne l'accès à l'attribut. Si c'est le cas, le point de vue est placé en sommet de la pile. Ensuite, la méthode effectue la recherche de la classe introduisant l'attribut. Cette recherche est réalisée en appelant la méthode `lookup-att` (introduite dans R-CLASS) de la classe `cls`. Cette méthode procède selon un principe similaire à `lookpup-mth` pour résoudre les ambiguïtés. Si aucune erreur n'est détectée lors de cette recherche alors la valeur de l'attribut est récupérée dans la liste d'association de l'objet à l'aide de la primitive #get-value ce qui nécessite notamment de préciser la classe et une position dans le vecteur associé.

4.2.2.4 Application: la représentation multiple et évolutive d'objets

L'intérêt pour un langage à objet de disposer des méta-classes est, outre de pouvoir s'auto-définir, de pouvoir l'étendre en concevant de nouvelles méta-classes par spécialisation. Le programmeur peut alors bénéficier des extensions définies par les méta-classes spécialisées en définissant les classes de son programme comme des instances de celles-ci.

Nous illustrons ici cette démarche d'extension, appliquée par la suite pour les contextes, à travers l'ajout de la représentation multiple et évolutive à ROME[1] au noyau méta-circulaire présenté précédemment.

Pour la représentation multiple et évolutive d'objets, deux aspects particuliers sont notamment à prendre en compte par rapport aux objets de base:

- Le premier aspect concerne la gestion d'une structure physique "malléable" pour ces objets, celle-ci étant déterminée suivant les liens de représentation.

- Le second aspect concerne l'exécution des objets (interprétation des messages, accès à la structure) qui doit se faire en prenant en compte leurs liens multiples de représentation à la place de leur unique lien d'instanciation.

L'extension pour des objets à représentation multiple et évolutive selon les principes vus en 2.3.3.2 et en 4.2.1 se résume à deux classes prenant en considération les deux aspects précédents:

- `RME-CLASS` est la (méta-)classe des classes instanciant de tels objets. Elle introduit les fonctionnalités nécessaires à la gestion de leur structure physique "malléable" ainsi qu'à leur exécution selon leurs liens de représentation.

- `R-OBJECT` est la sur-classes des classes de tels objets, introduisant en particulier les méthodes de base `r+` et `r-` pour manipuler les liens de représentation.

---

1. La décision d'ajouter la représentation multiple et évolutive par méta-extension plutôt que de l'intégrer directement dans le noyau méta-circulaire se justifie pour des raisons de simplicité et de clarté. Une telle intégration reviendrait en effet à poser la question de la représentation multiple et évolutive des classes et des méta-classes ce qui mériterait tout une étude en soi à la fois pour en déterminer les applications et pour en maitriser les conséquences sémantiques.

La figure suivante montre les liens d'instanciation et d'héritage qu'entretiennent ces deux classes avec celles du noyau de base. Une partie de l'exemple des produits logiciels introduit en section 2.3.3.2 pour présenter la représentation multiple et évolutive est également montrée afin d'indiquer les liens des classes correspondantes avec les deux classes précédentes. On peut observer que la classe d'instanciation de ces objets (`ProduitLogiciel`) est définie comme une instances de `RME-CLASS` et comme sous-classe directe (ou indirecte) de `R-OBJET`. Les sous-classes de la classe d'instanciation qui sont des classes de représentations sont définies comme des instances de `R-CLASS`.

Par rapport à ce schéma, insistons sur le fait que la classe d'instanciation est le méta-objet des objets à représentation multiple et évolutive.

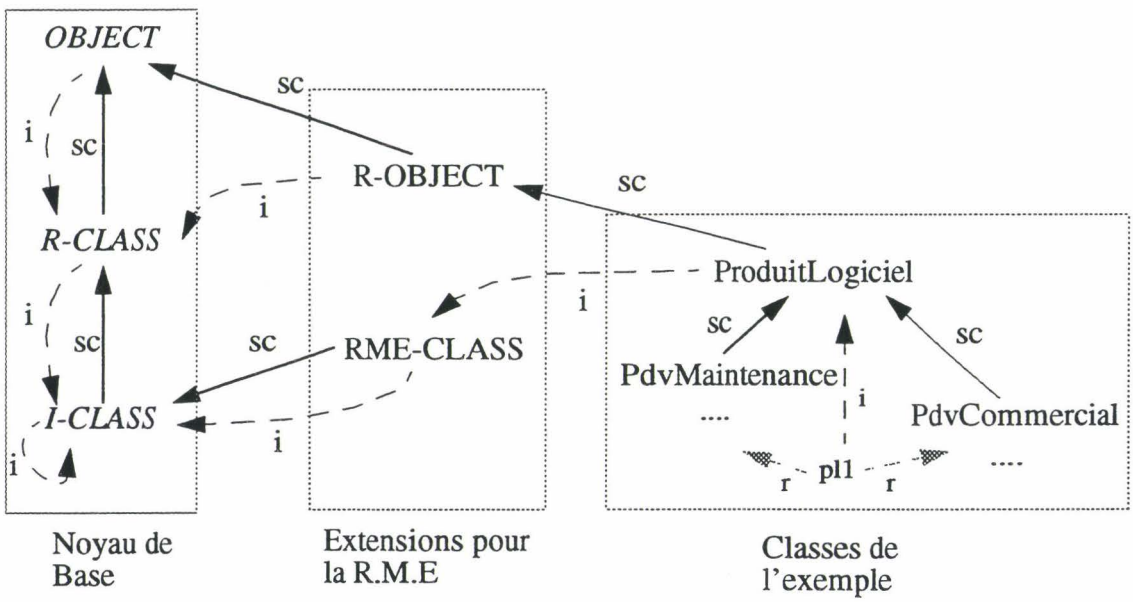


figure 4.6 : Noyau de base avec les extensions pour la représentation multiple et évolutive

`R-OBJECT` est instance de `R-CLASS` et sous-classe de `OBJECT`[1]. Elle définit le modèle commun à tous les objets à représentation multiple et évolutive qui est constitué de :

- l'attribut `rclasses` qui contient la liste des classes de représentation directe propre à un objet
- la méthode `r+` de rattachement d'un objet à une classe fournie en paramètre.
- la méthode `r-` permettant de détacher l'objet d'une de ses classes de représentation fournie en paramètre.

Voici la définition de cette classe:

```
(define R-OBJECT
  (R-CLASS 'new
    'supercl '(OBJECT)
    'attributes '(rclasses)
    'privmth '()
```

1. et non confondue avec cette dernière. Ainsi, les classes elles-mêmes ne sont pas des objets à représentation multiple et évolutive.

```
    'publmth '(
      r+ (lambda (cls) ((? 'iclass) 'add-rlink self cls))
      r- (lambda (cls) ((? 'iclass) 'remove-rlink self cls))
))
```

La manipulation des liens de représentations ayant un impact sur la structure des objets, la réalisation des traitements associées relèvent du comportement des classes. Les méthodes `r+` et `r-` sont donc définies en déléguant cette responsabilité à la classe d'instanciation (une RME-CLASS) dont l'objet reste toujours représentant (principe de représentation minimale). Cette délégation consiste à lui envoyer le message `add-rlink` (resp. `remove-rlink`) pour `r+` (resp. `r-`) muni de l'objet concerné (self) et de la classe à rattacher (resp.à détacher) en arguments.

RME-CLASS est instance de I-CLASS car elle est génératrice des classes instanciant des objets à représentation multiple et évolutive. Elle est sous-classe de I-CLASS pour en hériter la fonctionnalité d'instanciation ainsi que celle liée à l'exécution des objets pour ces classes. Par rapport à cette dernière, cette nouvelle méta-classe :

- ajoute un attribut `rcvr-rcls` destiné à contenir la liste des classes de représentation directe de l'objet pour lequel la classe résoud un message. Cet attribut est utilisé dans la redéfinition de la méthode `lookup-mth` (cf ci-dessous) et varie à chaque résolution suivant l'objet destinaire du message.

- ajoute les méthodes `add-rlink et remove-rlink` prenant en argument un objet et une classe. Ces deux méthodes apportent à la structure physique de l'objet les modifications liées au rattachement /détachement de la classe. Elles mettent aussi à jour son attribut `rclasses` et vérifie le respect du principe de représentation minimal.

- redéfinit la méthode `allocate` pour prendre en compte le besoin d'une structure physique "malléable".

- redéfinit les méthodes liées à l'exécution des objets et plus précisemment `handle-msg`, `read-attribute`, `write-attribute`, `lookup-mth` pour tenir compte des liens de représentation. La méthode `handle-msg` est ainsi redéfinie de la façon suivante dans cette classe:

```
1   handle-msg
2     (lambda (rcvr sndr sel args)
3       (let ((res '()))
4         (<- 'rcvr-rcls (self 'read-attribute rcvr 'rclasses))
5         (super)))
```

Dans cette méthode, on commence par récuperer la liste des classes de représentations directe de l'objet pour lequel un message est à résoudre puis celle-ci est rangée dans l'attribut `rcvr-rcls`. (ligne 4). Un appel à la super-méthode (`handle-msg` de I-CLASS) est ensuite réalisé (ligne 5) pour appliquer la résolution standard.

Cette résolution est alors implicitement redéfinie dans la phase de recherche par la définition suivante de `lookup-mth` à ce niveau.

```
1   lookup-mth
2     (lambda (sel allmth)
```

```
3        (mappend '(lambda (cls) (cls 'lookup-mth sel allmth) (? 'rcvr-rcls))
```

Cette redéfinition complète de `lookup-mth` consiste à distribuer la recherche de la méthode parmi les classes de représentation directe de l'objet rangées préalablement dans l'attribut `rcvr-rcls`. Le résultat retourné est une liste fusionnant le résultat de l'envoi du message `lookup-mth` sur chacune de ces classes (ligne 3). Notons que la méthode standard `apply-mth` d'application des méthodes n'a pas besoin d'être redéfinie pour cette résolution de message. La gestion des points de vue implicite reste identique dans le cas d'objets à représentation multiple et évolutive.

Cette application montre les possiblités d'extension du langage. L'extension obtenue peut à son tour être étendue. C'est ce que nous allons montrer en section 5 lorsqu'il s'agira d'implanter les contextes.

4.3 Des points de vue aux contextes

L'objet de cette partie est la présentation des principes et techniques pour mettre en oeuvre les contextes en ROME. Cette mise en oeuvre réutilise avantageusement les facilités offertes par ce dernier. La description multiple par sous-classement (cf chapitre 1 section 3.3.2) est exploitée pour décrire modulairement les parties fonctionnelles ce qui mène à des graphes d'héritage multiple complexes dont la structuration réflète les plans. Les descriptions issues de ces graphes sont ensuite rattachées aux objets par manipulation de liens objet-classes que rend possible la représentation multiple de ROME. La contextualisation d'objets est alors vue à travers des opérations de sélection de parties de graphes qui font appel à la notion de point de vue.

4.3.1 Représentation des plans

4.3.1.1 Plan de base

Pour rendre compte du plan de base en ROME, nous introduisons un nouveau type de classe que nous appelons *classe de base*. Ces classes de base introduisent les identités des objets intervenant dans plusieurs contextes et déterminent leur description de base destinée à être partagée par tous les contextes. Comparées aux classes ROME standard, celles-ci se distinguent par le fait qu'elles peuvent être enrichies selon plusieurs contextes.

Pour définir une classe de base, on procède de manière analogue aux classes ROME, en envoyant le message `new` à la métaclasse concernée avec en paramètres les caractéristiques qui la définissent. Les méta-classes permettant de définir des classes de base sont `CA-CLASS` et `CI-CLASS`. Ces deux méta-classes traduisent la distinction entre classe abstraite (instance de `CA-CLASS`) et classe concrète (instance de `CI-CLASS`) pour ce nouveau type de classe. Par ailleurs, toutes les classes de base sont au moins sousclasse de `CR-OBJECT` qui est la classe racine du plan de base. Cette classe spécifie le modèle commun à tous les objets fonctionnant dans des contextes multiples.

Voici, en guise d'illustration, une partie du plan de base de notre exemple exprimée en ROME correspondant à la définition des classes `Circuit`, `Wire`, `Pin`, `InputPin`.

```
(define Wire
  (CI-CLASS 'new
   'name 'Wire
```

```
    'supercl '(CADObject)
    'attributs '(output input)
    'privmth '(
         initialize (lambda (l-init) ...)
    'publmth '(
         output (lambda () ...)
         input (lambda () ...)
         connected-to? (lambda (gate) ...)
         disconnect-all (lambda ())
   ))


(define Pin
   (CA-CLASS
    'supercl '(CADObject)
    'attributs '(gate)
    'private '(
         initialize (lambda (l-init) ...))
    'public '(
         gate (lambda () ...)
         gate=? (lambda (gate) ...)
         connected? (lambda () #abstract)
         connect-as-output? (lambda () ...)
         connect-as-input? (lambda () ...)
         connect-to-wire (lambda (wire) #abstract)
         disconnect-wire (lambda (wire) #abstract)
         disconnect-all (lambda () #abstract)
   ))

(define OutputPin
   (CI-CLASS
    'supercl '(Pin)
    'attributs '(wires)
    'private '()
    'public '(
         connected? (lambda () ...)
         connect-as-output? (lambda () ...)
         connect-to-wire (lambda (wire) ...)
         disconnect-wire (lambda (wire) ...)
         disconnect-all (lambda () ...))

(define Circuit
   (CI-CLASS
    'supercl '(CADObject)
    'attributs '(input-pins output-pins gates wires)
    'private '(
      all-pins (lambda () ...)
      add-gate (lambda (gate) ...)
      add-wire (lambda (wire) ...)
      add-input (lambda (name) ...)
      add-output (lambda (name) ...)
      valid? (lambda ()...))
    'public '(
      gate-named (lambda (symb) ...)
      wire-named (lambda (symb) ...)
      pin-at (lambda (i) ...)
      pin-named (lambda (symb) ...))
   ))
```

Nous pouvons constater que la définition d'une classe de base s'apparente à celle d'une classe ROME. Elle se compose de la spécification d'une sur-classe (qui doit nécessairement être une classe de base), d'un ensemble d'attributs et d'un ensemble de méthodes publiques et

privées. Seul l'héritage simple est permis entre les classes de base. Pour exprimer la description de base des objets, les possibilités offertes par l'héritage entre deux classes de base sont les mêmes que pour deux classes ROME. Dans la définition de la sousclasse, on peut faire référence aux attributs et méthodes de la sur-classe mais aussi redéfinir les méthodes héritées. Lorsqu'une méthode est redéfinie, l'ancienne définition reste accessible en incluant l'opération (super) dans la nouvelle définition.

4.3.1.2 Plans fonctionnels

A partir de la représentation du plan de base décrite à la section précédente, il s'agit ensuite de prendre en compte les plans fonctionnels. Pour représenter les parties fonctionnelles constituant de tels plans, nous utilisons la description multiple des objets fournie en ROME. La classe de base est ainsi identifiée à la classe qui introduit les identités des objets. Les parties fonctionnelles jouant un rôle purement descriptif et donc n'introduisant pas de nouveaux objets, elles sont représentées en interne par des classes de représentation que nous appelons *classe de partie fonctionnelle*. Ces classes détiennent les caractéristiques des parties fonctionnelles et sont placées comme sous-classe directe de la classe concernée pour hériter sa description de base. Cette approche forme un fil directeur tout au long de cette partie.

Partie fonctionnelle = Classe de représentation

Les classes de parties fonctionnelles sont créées à partir de la définition d'une partie fonctionelle qui s'effectue en invoquant la méthode `extend-in`[1] des classes de base (abstraite et concrète). L'appel à cette méthode se présente syntaxiquement comme suit:

```
(<class-base>
     'extend-in <name>
     'attributes (<att1> ...<attN>)
     'privmth (<mth1> ... <mthN>)
     'publmth (<mthM>... <mthP>))
```

La méthode `extend-in` prend en entrée quatre paramètres qui sont:

- un symbole `<name>` désignant un plan fonctionnel auquel la partie fonctionnelle est destinée. Une classe ne peut être enrichie qu'une seule fois dans un plan fonctionnel.

- une liste définissant les attributs de la partie fonctionnelle selon le même format que pour une classe. Les noms de ces attributs doivent être différents de ceux introduits et hérités au niveau de la description de base de la classe d'une part et de ceux hérités localement d'autre part.

- une liste définissant les méthodes privées de la partie fonctionnelle selon le même format que pour une classe. Cette liste peut contenir des redéfinitions de méthodes privées héritées localement ou issues du plan de base.

- une liste définissant les méthodes publiques de la partie fonctionnelle selon le même format que pour une classe et les mêmes modalités de redéfinitions que pour les méthodes privées.

---

1. Cette opération existe au même titre par exemple que la méthode new de création des objets disponibles pour les classes concrètes.

La méthode `extend-in` ne retourne pas de résultat mais des erreurs sont déclenchées si les arguments ne respectent pas les conditions énoncées ci-dessus.

En guise d'illustration, donnons comme exemple la définition des parties fonctionnelles associée aux classes de base `Wire`, `InputPin` et `AndGate2` pour le contexte d'édition schématique. Cette opération revient à envoyer le message `extend-in` à chacune de ces classes en spécifiant le même plan. Ce qui nous donne le code suivant:

```
(Wire
   'extend-in 'Schematic
   'attributes (selected)
   'privmth ()
   'publmth (
      select (lambda () ...)
      unselect (lambda () ...)
      display-on (lambda (canvas) ...)
      cross-over? (lambda (g) ...)
      selected-at?  (lambda (location) ...)

(Circuit
   'extend-in 'Schematic
   'attributes (grid selection window)
   'privmth (
      find-pin-at (lambda (loc) ...)
      find-wire-at (lambda (loc) ...)
      find-gate-at (lambda (loc) ...)
      invalidate (lambda () ...)

   publmth (
      display-on (lambda (canvas) ...)
      edit (lambda () ...)
      cmd-close (lambda () (<- 'window '()))
      cmd-addGate (lambda () ...)
      cmd-addWire (lambda () ...)
      cmd-removeWire (lambda ()...
      cmd-removeGate (lambda () ...)
      cmd-move (lambda () ...)
      cmd-lock (lambda () ...)
      cmd-unlock (lambda ()...)
      cmd-setGrid (lambda () ...)
      set-selection (lambda (loc) ...)
      process-menu (lambda (loc) ...)))
```

Les classes générées à partir des parties fonctionnelles définies dans l'exemple précédent sont les suivantes:

```
   (Wire
      'extend-in 'Schematic ...)

=> (R-CLASS 'new
      'supercl '(Wire)
      'attributes (selected)
      'privmth ()
      'publmth (
         select (lambda () ...)
         unselect (lambda () ...)
         display-on (lambda (canvas) ...)
         cross-over? (lambda (g) ...)
         selected-at?  (lambda (location) ...)
```

```
   (Circuit
      'extend-in 'Schematic ...)
=> (R-CLASS 'new
      'supercl '(Circuit)
      'attributes (grid selection window)
      'privmth (
         find-pin-at (lambda (loc) ...)
         ....)
      'publmth (
         display-on (lambda (canvas) ...)
         edit (lambda () ...)
         cmd-close (lambda () (<- 'window '()))
         cmd-addGate (lambda () ...)
         ....)
```

Nous pouvons remarquer que les classes générées sont des classes de représentation (instance de R-CLASS) et sont obtenues de façon quasi-immédiate à partir de la définition des parties fonctionnelles. Les attributs et les méthodes (publiques ou privées) fournies par celle-ci sont reportés au niveau de la nouvelle classe en restant inchangés. Nous pouvons également noter que ces classes sont placées en tant que sous-classe directe de leur classe de base respective.

Le processus de sous-classement décrit ci-dessus s'applique de façon systématique pour toute partie fonctionnelle. L'ensemble des classes obtenues quand plusieurs parties fonctionnelles sont définies pour la même classe de base est illustrée à la figure 4.7 pour l'exemple de Circuit.

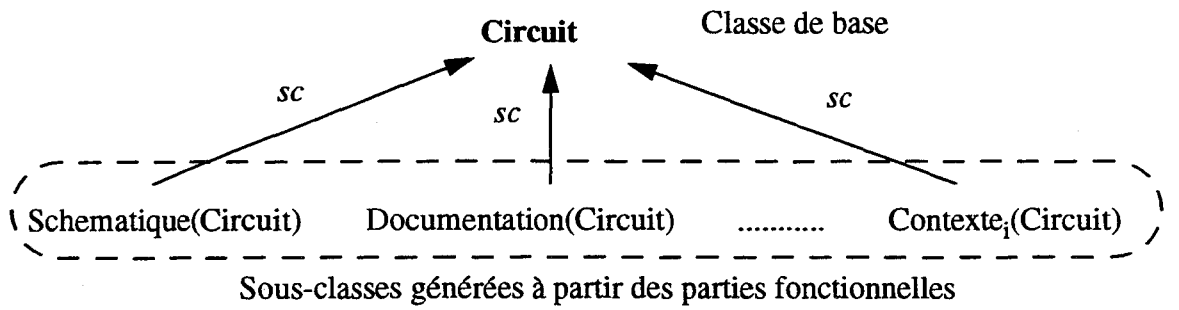


figure 4.7 : Ensemble de classes déterminant la description des objets Circuit

Le résultat est un ensemble de classes qui rend compte des enrichissement définis pour les contextes. Au sein de cette structuration, la sur-classe (une instance de CI-CLASS) est la classe de base. Celle-ci détient l'identité des objets ainsi que leur description de base. Quant aux sous-classes (des instances de R-CLASS), elles définisssent les parties fonctionnelles pour ces objets. Il n'y a toujours qu'une seule classe de partie fonctionnelle pour chaque contexte.

Cette structuration amène deux remarques importantes.

- La première concerne la modularité des descriptions obtenues grâce au sous-classement. D'une part, chaque classe ne pouvant avoir connaissance que d'elle-même et de ses sur-classe, une classe de partie fonctionnelle n'est pas autorisée à faire référence aux attributs introduits par les autres classes de parties fonctionnelles. Ceci assure la restriction d'accès aux attributs entre parties fonctionnelles d'une même classe. D'autre part, les caractéris-

tiques propres à chaque partie fonctionnelle sont localisées dans des classes distinctes et indépendantes. Ceci rend possible l'existence de caractéristiques de même nom dans plusieurs parties fonctionnelles ainsi que des redéfinitions multiples d'une même méthode de la description de base

- La seconed remarque concerne le statut des sous-classes de parties fonctionnelles générées. Contrairement à une classe de base, de telles classes ne sont pas destinées à être hérité explicitement pour élaborer de nouvelles classes d'objet, ni même à être instanciée. Ces classes existent uniquement pour l'implantation. Pour éviter au programmeur toute confusion entre celles-ci et ses classes mais aussi toute utilisation incohérente de sa part, nous faisons le choix de ne pas rendre explicites l'existence de ces classes.

Le principe décrit ci-dessus est systématiquement appliquée pour chaque classe de base comme illustré à la figure 4.8.

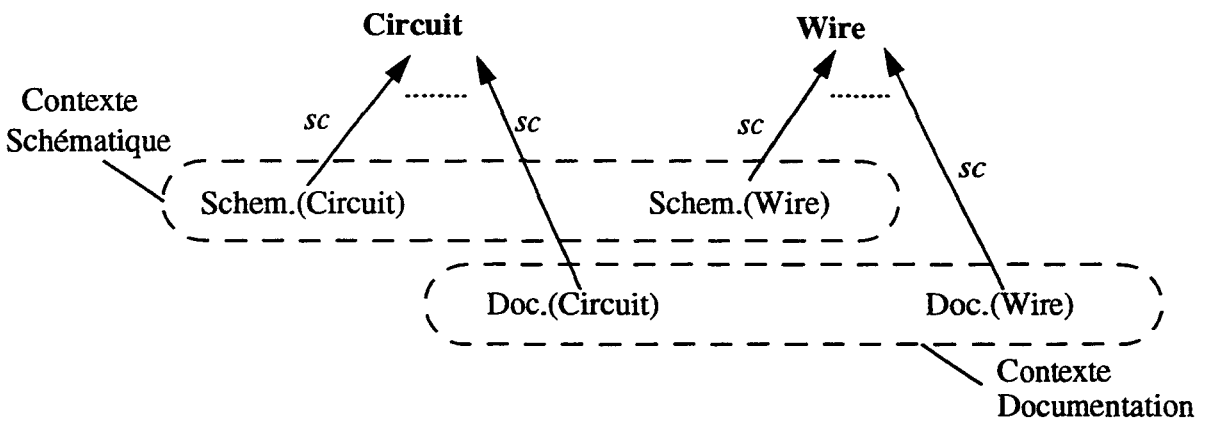


figure 4.8 : Structuration systématisée à plusieurs classes

L'aspect intéressant mis en évidence par cette figure est le sous-classement par rapport à un même contexte ce qui est indiqué en pointillé. L'intervention des objets `Circuit` et `Wire` dans le même contexte d'édition schématique se traduit conjointement par l'introduction d'une classe de partie fonctionnelle destinée à ce contexte. Ceci met en évidence une systématisation de la description multiple transversalement à plusieurs objets. Par la suite, nous donnerons un statut à cettre transversalité.

A ce stade, nous savons prendre en compte l'enrichissement de classes de base sans relation d'héritage entre elles. Dans ce qui suit nous nous intéressons au cas d'une hiérarchie de classes.

Classes de parties fonctionnelles et héritage local au plan

Le principe décrit précédemment s'applique également pour une hiérarchie de classes de base. Ceci entraine le sous-classement de chaque classe de la hiérarchie avec des classes de parties fonctionnelles associées. Ce sous-classement n'est toutefois pas suffisant: il faut également prendre en compte l'héritage local au plan fonctionnel.

Pour prendre en compte cette caractéristique, nous mettons systématiquement en relation d'héritage les classes de parties fonctionnelles d'un même plan selon la hiérarchie des classes du plan de base. De cette manière, une classe de partie fonctionelle associée à une classe de base

devient une sur-classe directe pour les classes de parties fonctionnelles associées aux sous-classes directes de cette dernière. Au niveau d'une classe de partie fonctionnelle, cela entraine la gestion d'un nouveau lien d'héritage en plus de celui qui le lie à sa classe de base.

Pour illustrer ce principe, considérons les parties fonctionnelles définies dans le plan d'édition schématique pour quelques classes de la hiérarchie des portes à deux entrées.

```
(Gate2
   'extend-in 'Schematic
   'attributes '()
   'privmth '()
      display-formOn (lambda (canvas)
                       (let ((loc (? 'location)))
                          ((? 'input1) 'display-on canvas loc)
                          ((? 'input2) 'display-on canvas loc)
                          ((? 'output) 'display-on canvas loc)))
      bounds (lambda () (Rectangle 'new 'origin (? 'location) 'with 10 'height 20))

   'publmth '(
      pin-at-location
         (lambda (loc)
            (cond ((? 'input1) ' loc) (? 'input1)
                  ((? 'input1) ' loc) (? 'input2))
                  ((? 'output) ' loc) (? 'output))
                  (else '()))

(AndGate
   'extend-in 'Schematic
   'attributes '()
   'privmth (
      display-formOn (lambda (canvas)
                          (canvas 'display-form AndForm (? 'location))
                          (super)))
   'publmth '())

(OrGate
   'extend-in 'Schematic
   'attributes '()
   'privmth (
      display-formOn (lambda (canvas)
                          (canvas 'display-form OrForm (? 'location))
                          (super)))
   'publmth '())
```

Ici, le fait d'exprimer des parties fonctionnelles pour les classes `Gate2`, `AndGate` et `OrGate` relativement au même plan suffit à établir l'héritage local entre celles-ci. Cet héritage est exploité à plusieurs endroits. Il est employé dans la méthode `display-formOn` des parties fonctionnelles associées à `AndGate` et `OrGate` pour appeler la super-méthode (super). L'accès à l'attribut `location` (attribut introduit dans la partie fonctionnelle de `CircuitEntiy`) dans cette même méthode rend également compte de cet héritage local.

Les classes de parties fonctionnelles correspondantes ainsi que leurs relations d'héritage sont représentées par la figure 4.9.

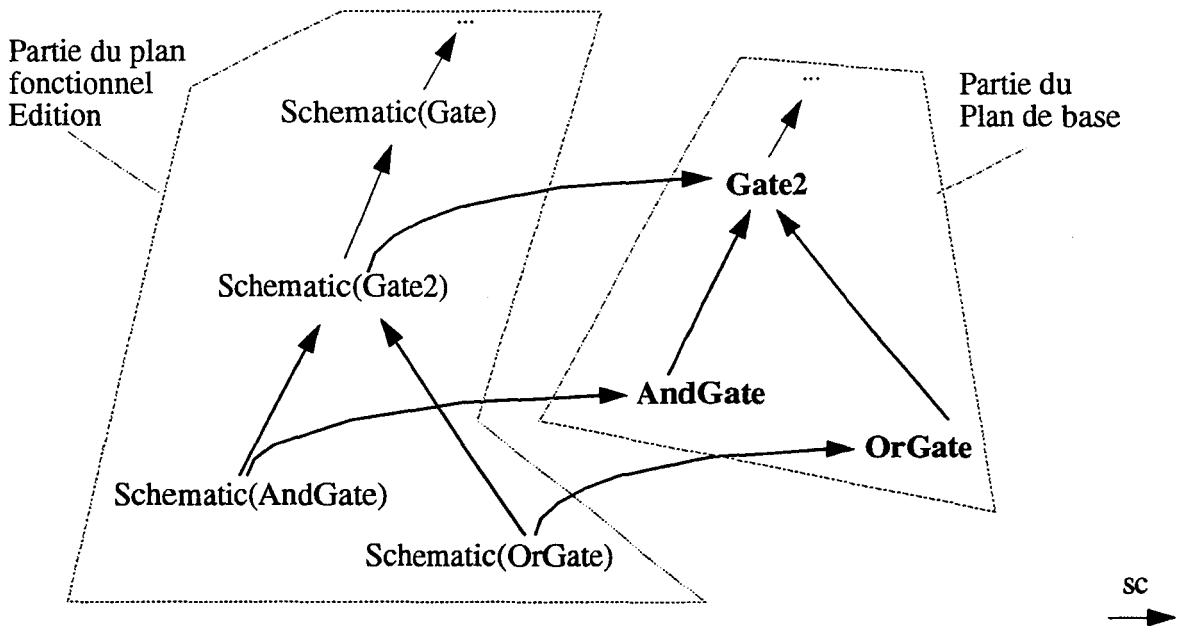


figure 4.9 : Application des principes à une hiérarchie de classes

Cette figure met en évidence une sous-hiérarchie supplémentaire de classes de parties fonctionnelles induite par l'approche. Celle-ci est parallèle à la hiérarchie des classes de base avec laquelle elle est mise en relation par héritage pour l'accès à la description de base.

De façon général, la règle de sousclassement entre classes de parties fonctionnelles relativement à un contexte peut être formulée ainsi:

Soit un contexte C_0
et deux classes de base A et B enrichies pour ce contexte,
avec B sous-classe de A
alors la classe $C_0(B)$ représentant la partie fonctionnelle de B associée au contexte C_0
doit être sous classe de $C_0(A)$ représentant celle de A pour le même contexte

La prise en compte de cette règle a des répercussions sur la création des classes de parties fonctionnelles. Examinons à ce sujet, les créations de classes de parties fonctionnelles pour l'exemple précédent. Cela nous donne:

```
(Gate2 'extend-in 'Schematic ....)
=> (R-CLASS 'new 'supercl '(Gate2 Schematic(Gate)) ....)

(AndGate 'extend-in 'Schematic ....)
=> (R-CLASS 'new 'supercl '(AndGate Schematic(Gate2)) ....)

(OrGate 'extend-in 'Schematic ....)
=> (R-CLASS 'new 'supercl '(OrGate Schematic(Gate2))....)
```

Par rapport à la génération indiquée à la sous-section précédente, la différence essentielle se situe au niveau de la liste des sur-classes générées. Celle-ci est augmentée avec la classe de partie fonctionnelle générée pour la sur-classe relativement au même plan fonctionnel.

Cette règle s'applique pour l'ensemble des contextes, ce qui conduit à des graphes d'héritage multiples complexes dotés cependant d'une structure régulière qui réflète les plans. La figure suivante schématise cette structure pour des plans fonctionnels associés aux contextes C1 et C2.

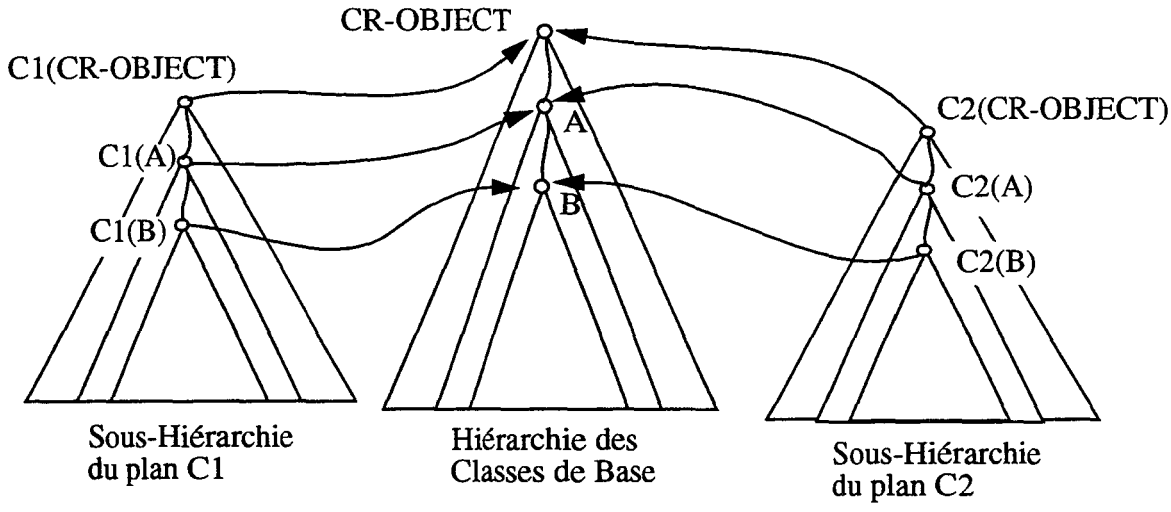


figure 4.10 : Graphe d'héritage structuré selon les plans

Cette figure montre que la règle produit des sous-hiérarchies pour chaque plan fonctionnel. Ces sous-hiérarchies sont indépendantes et regroupent les classes de parties fonctionnelles issues d'un même plan. Cette modularité des sous-hiérarchies est induite par la modularité au niveau de chaque classe (cf sous-section précédente).

Chaque sous-hiérarchie a pour racine une classe de représentation qui est sous-classe directe de `CR-OBJECT` pour le contexte. Ces classes introduisent chacune un plan fonctionnel. Elles existent toujours[1] et sont obtenues comme les autres parties fonctionnelles au moyen de l'opération `extend-in` appliquée sur la classe de base la plus générale. Ces classes ne définissent aucune caractéristique supplémentaire. Nous reviendrons plus en détail sur l'importance de ces classes et leur emploi pour simplifier le calcul des contextes par points de vue dans la section 4.3.3.2.

Jusqu'ici, nous avons un peu simplifié la règle précédente en négligeant les cas d'enrichissement implicite où une classe de base n'a pas de partie fonctionnelle associée dans un plan . Lorsque ce cas survient, les enrichissements associés à ses sur-classes pour le contexte doivent être hérités pour ceux de ses sousclasses. La solution retenue consiste à relier par héritage les classes de parties fonctionnelles liées aux sous-classes à la classe de partie fonctionnelle la plus spécialisée parmi celle des sur-classes[2]. Cette solution peut être illustrée à travers l'exemple de la classe `Gate2` dans le plan de normalisation. La figure suivante montre les classes de représentation issues des enrichissements explicites définies pour ce plan et les relations d'héritage entre celles-ci. L'absence de partie fonctionnelle pour la classe `Gate2` conduit à faire hériter directement `Normalization(AndGate)` et `Normalization(OrGate)` de `Normalization(Gate)`.

1. Ces classes sont générés systématiquement par le système.
2. Une autre solution aurait été de définir une partie fonctionnelle sans caractéristique pour la sur-classe.

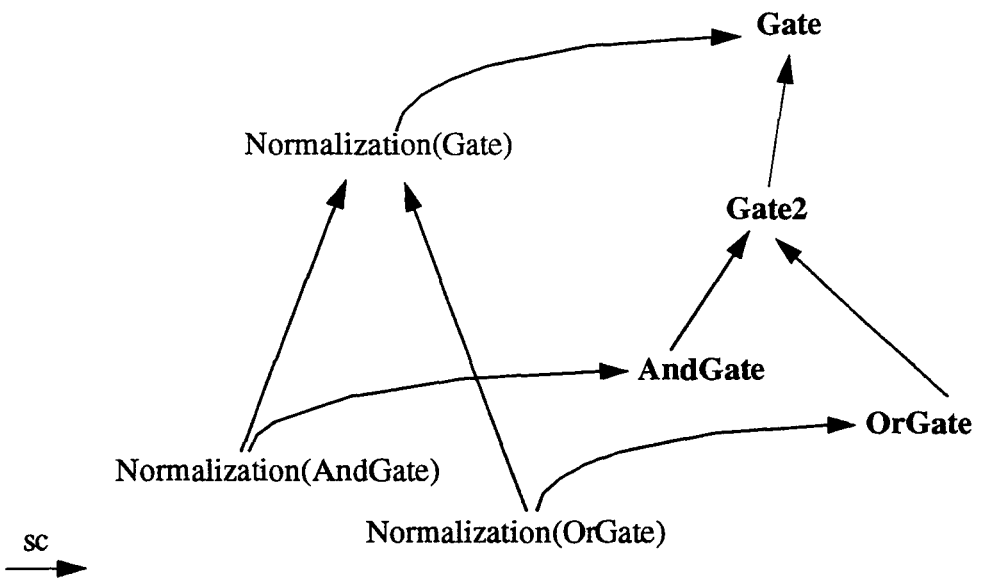


figure 4.11 : Enrichissement implicite des sous-classes de Gate2 dans le plan Normalisation

Au cours de cette partie, nous avons vu les principes de construction des graphes d'héritage multiple à partir des définitions fournies par le programmeur. Dans la prochaine section, nous nous intéressons à la représentation des objets ainsi décrits.

4.3.2 Représentation des objets

Lors de la section précédente, nous avons vu que la définition de plusieurs parties fonctionnelles pour une même classe de base mène à générer des classes de représentation correspondantes. Les objets identifiés par la classe de base sont alors complètement décrits par celle-ci et l'ensemble de ses classes de parties fonctionnelles. Pour obtenir une représentation des objets qui rend compte de cette description, nous exploitons les possibilités de représentation multiple offertes en ROME.

Le principe consiste dans un premier temps à créer les objets en tant qu'instances de la classe de base (une `CI-CLASS`) ce qui rend compte de leur identité. Dans un second temps, on procède, par manipulation des liens objet-classe, à leur rattachement à l'ensemble des sousclasses représentant les parties fonctionnelles. La figure suivante représente le principe pour l'exemple d'un objet représentant une équipotentielle :

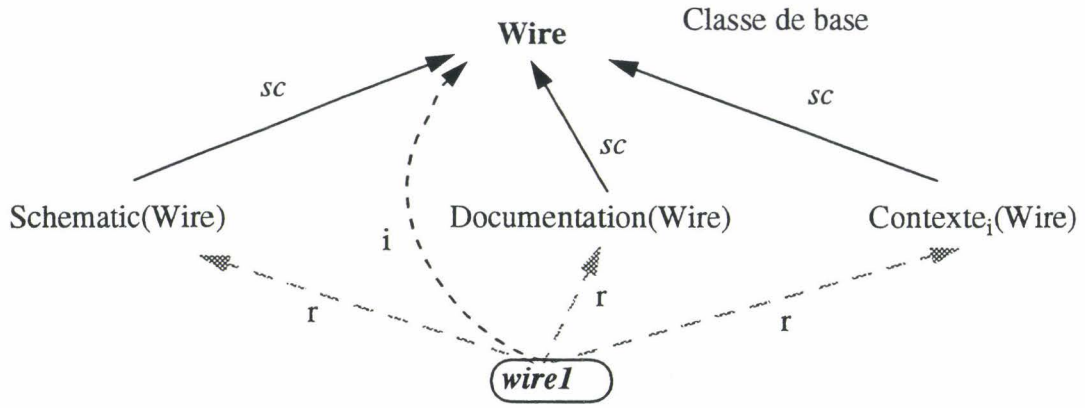


figure 4.12 : Représentation multiple appliquée pour les contextes

On voit sur cette figure que l'objet `wire1` est instance de la classe de base correspondante et représentant de toutes les sous-classes de parties fonctionnelles, lui procurant ainsi un seul lien de représentation par contexte. Contrairement à la représentation multiple libre en ROME, ceci amène à systématiser le rattachement de tous les objets issus de la classe de base aux sous-classes de parties fonctionnelles.

Il en est de même dans le cas d'objets décrits par une hiérarchie de classes de base. Par exemple, pour une instance de `AndGate`, la figure 4.13 présente une partie de son graphe de représentation (les pointillés indiquent l'existence de sous-classes non prises en compte dans le graphe de représentation des objets).

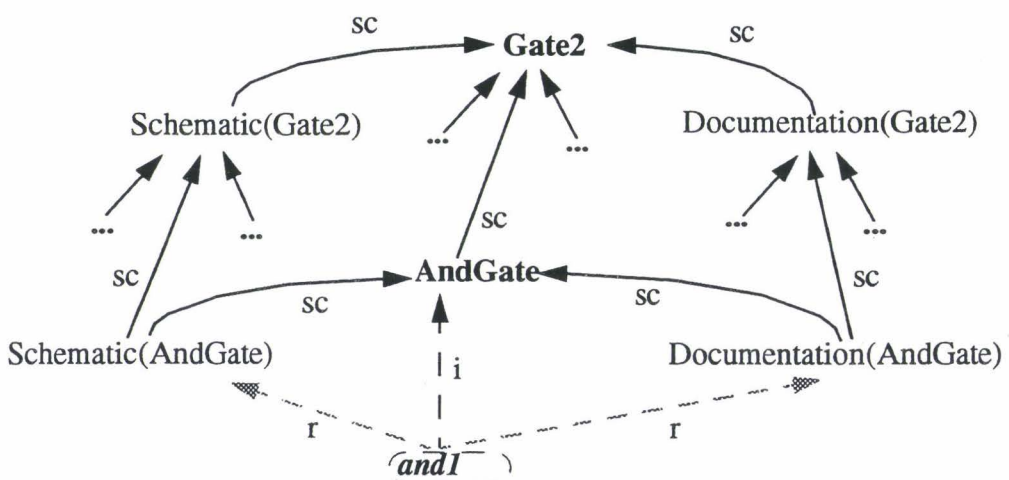


figure 4.13 : Représentation multiple et hiérarchie de classes

De manière générale, le graphe de représentation des objets se compose pour une part de la classe d'instanciation et de ses sur-classes, pour une autre part, de toutes les classes de parties fonctionnelles obtenues par enrichissement de ce premier ensemble.

Les attributs dont disposent les objets après rattachement sont déterminés suivant les principes de l'héritage multiple de ROME formulés par la première et la troisième règle (cf section 4.2.1). Selon le premier principe, les caractéristiques introduites par les classes de base

sont héritées une seule fois. Dans l'exemple, l'objet `and1` possède donc une seule occurence des attributs de la description de base. Selon le troisième principe, les objets disposent de tous les attributs définis dans des classes de parties fonctionnelles indépendantes (par exemple dans les classes `Schematique(Gate2)` et `Documentation(Gate2)` même si ils ont le même nom.

Pour les rattachements aux sous-classes de la classe de base , une précision doit être apportée relativement aux principes de la représentation multiple de ROME. Il a été précisé dans la section 4.2.1 que cette représentation obéit à un principe de représentation minimale qui ne permet pas de rattacher les objets à une sous-classe possédant une sur-classe indépendante de la classe d'instanciation. Rapporté à l'exemple précédente, ce principe s'oppose donc au rattachement des instances de AndGate aux sous-classes `Schematic(AndGate)` et `Documentation(Gate2)`. Ces classes ont en effet une de leurs sur-classes respectives `Schematic(Gate)` et `Documentation(Gate)` qui se situe en dehors du domaine d'affinement délimité par la classe d'instanciation (ici `AndGate`). Ce problème trouve son explication dans le fait que la représentation multiple a été initialement conçue pour traiter des points de vue sur une seule classe d'objet et non pas pour traiter des points de vue sur un graphe d'héritage comme c'est le cas ici. De façon à inclure les classes de parties fonctionnelles dans le graphe de représentation des objets, nous relachons donc cette contrainte de la façon suivante: toute classe de représentation directe d'un objet est sous-classe (et non plus sous-classe stricte) de sa classe d'instanciation. Ce qui permet de réaliser l'héritage local des parties fonctionnelles, parallèle à l'héritage des classes correspondantes du plan de base[1].

Enfin, un dernier problème est lié à l'enrichissement implicite des classes de base concrète. Le problème vient du fait qu'il n'est pas possible en respect du principe de représentation minimal de rattacher directement les instances à la classe de partie fonctionnelle dont il faut hériter implicitement. La solution adoptée dans le cas présent consiste à définir des parties fonctionnelles sans caractéristiques pour toutes les classes de base concrètes concernée par l'enrichissement implicite. Les classes de parties fonctionnelles correspondantes sont alors générées selon les principes indiqués à la section précédente. Pour l'exemple d'enrichissement implicite des classes instanciables InputPin et OutputPin dans le contexte de documentation donné dans le chapitre précédent, la solution est représentée à la figure 4.14. Ici, les classes de parties fonctionnelles générées de façon implicite sont `Documentation(OutputPin)` et `Documentation(InputPin)`. Chacune de ces classes est directement reliée à `Documentation(Pin)` pour garantir l'héritage local.

1. Notons, bien que la représentation évolutive libre ne nous concerne pas ici que ce relachement reste compatible avec ses règles en ROME. En effet, tout sur-classe qui n'est pas sous-classe de la classe d'instanciation ne peut être classe de représentation directe de l'objet. La représentation évolutive (par r-) vers une telle classe reste donc impossible.

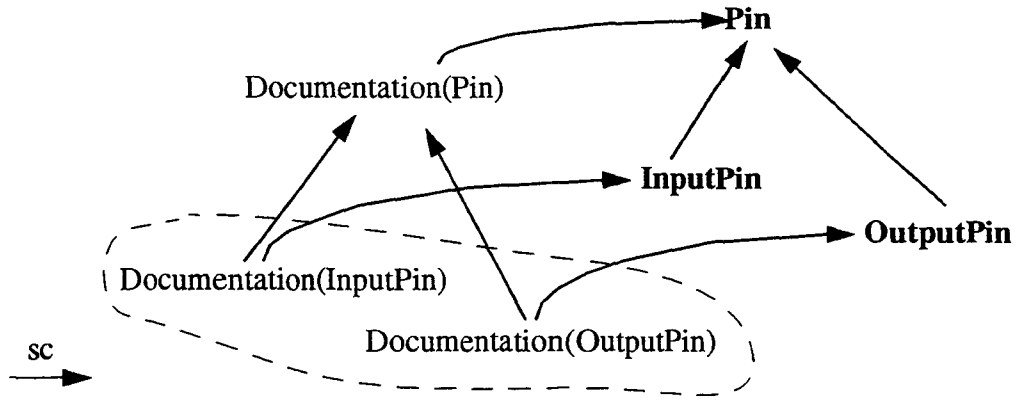


figure 4.14 : Enrichissement implicite des classes de base concrètes

La représentation des objets étant établie, nous allons maintenant nous pencher sur la recherche de leurs caractéristiques par contextes.

4.3.3 Recherche par contextes des caractéristiques

Dans cette partie, nous précisons comment obtenir la recherche par contextes de caractéristiques pour les objets représentés selon les principes précédents. Cette recherche se fait sur une partie de la description de l'objet qui dépend du contexte, cette partie étant considérée selon la stratégie modulaire introduite au chapitre précédent. Nous commençons par montrer que cette recherche consiste à parcourir un sous-graphe du graphe d'héritage selon des modules. Le besoin de considérer ainsi des partie du graphes nous conduit alors à examiner la notion de point de vue comme moyen de réalisation de cette recherche. Nous mettons alors en évidence les problèmes que pose l'utilisation des points de vue. Pour remédier à ces problèmes, nous proposons finalement une solution qui consiste à gérer les points de vue de façon spécifique. Cette solution va nous permettre de traduire aisemment la recherche par contextes à l'aide de sélections de point de vue.

4.3.3.1 Exploitation modulaire du graphe de représentation des objets

Selon les principes de mise en oeuvre décrits précédemment, la recherche liée à un contexte consiste à faire le choix d'une partie du graphe de représentation des objets. La structure modulaire du graphe permet d'identifier cette partie, quelque soit le contexte considéré. Elle se réduit au sous-graphe comprenant les classes de base et les classes de parties fonctionnelles relatives au contexte. La figure 4.15 montre en encadré grisé les deux sous-ensembles des classes qui déterminent la description des objets `NotGate` correspondant respectivement au contexte d'édition schématique et de documentation.

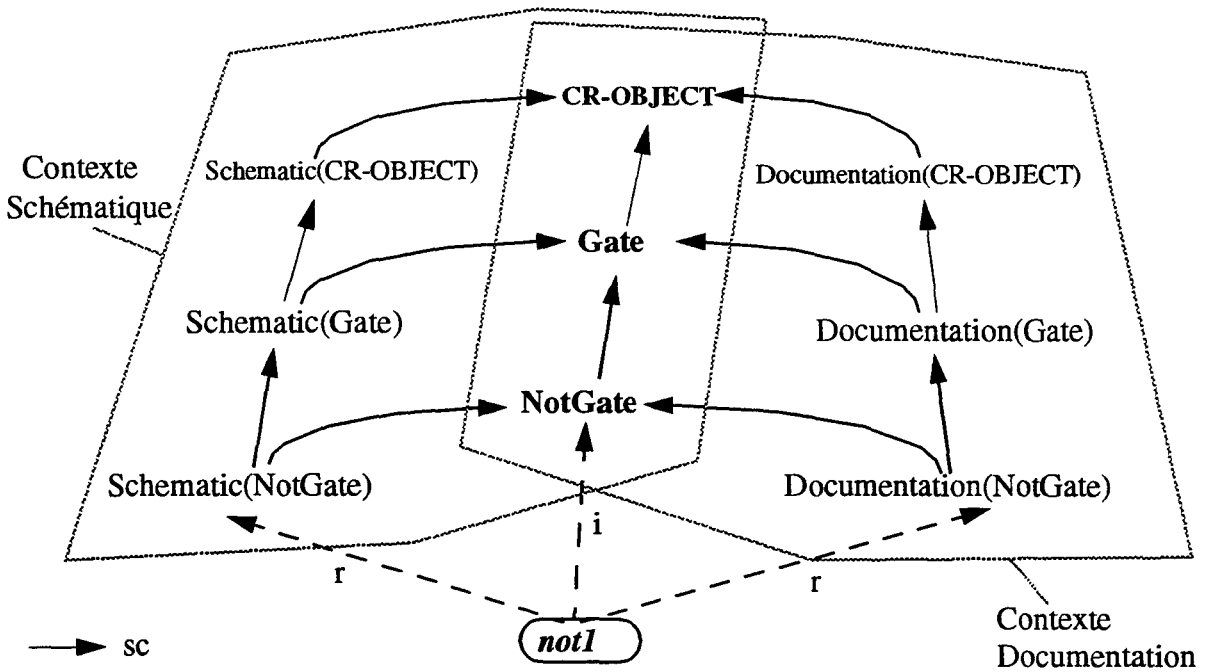


figure 4.15 : Sous-graphes relatifs aux contextes

On obtient ainsi une décomposition du graphe de représentation en sous-graphes relatifs aux différents contextes. Ces sous-graphes ne sont pas disjoints car ils ont en commun l'ensemble des classes de base. Rechercher une caractéristique de l'objet pour un contexte revient alors à parcourir le sous-graphe correspondant. Pour un contexte donné, ce sous-graphe est toujours le même. Par ailleurs, un changement de contexte se traduit par un changement de sous-graphe à parcourir.

Pour un sous-graphe relatif à un contexte, il s'agit de l'exploiter selon notre stratégie d'héritage modulaire. Les deux modules nécessaires à l'application de cette stratégie pour un graphe de représentation d'un objet sont:

- le *module de base* incluant les classes de base;
- le *module spécifique* incluant les classes de parties fonctionnelles.

Par exemple, le module de base et le module spécifique obtenus pour le sous-graphe du contexte d'édition schématique sont représentés à la figure 4.16.

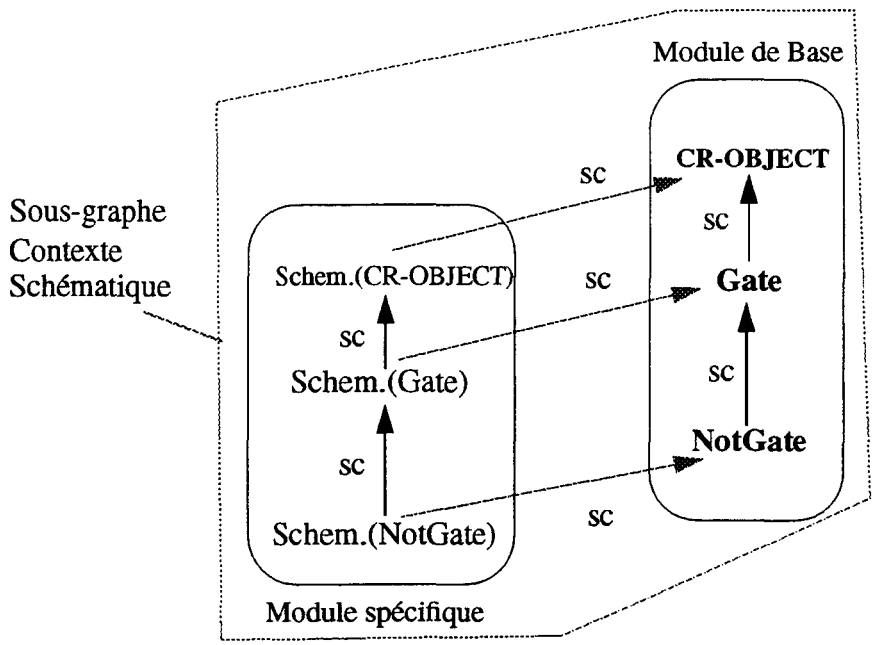


figure 4.16 : Structuration en modules du sous-graphe

La structuration en modules présentée ci-dessus pour le sous-graphe relatif à un contexte est applicable aux sous-graphes des autres contextes. Cela conduit à interpréter le graphe de représentation des objets sous la forme d'une hiérarchie structurée de modules. Pour l'exemple des portes Non donné à la figure 4.15, la hiérarchie de modules obtenue se présente de la façon suivante:

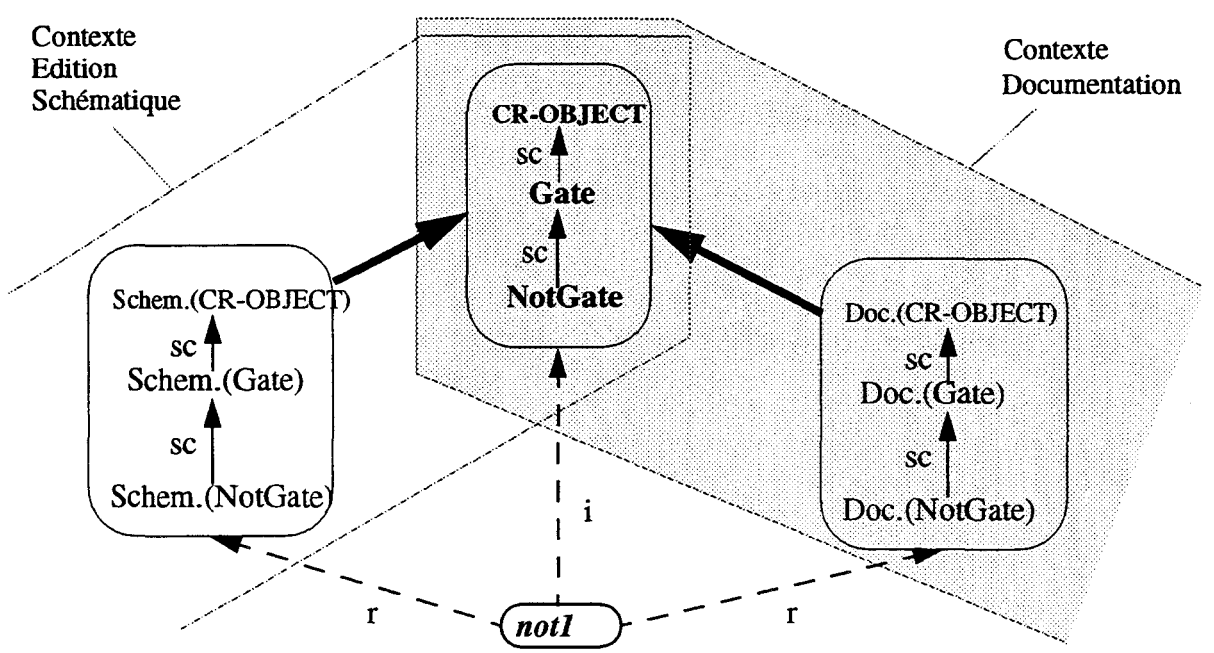


figure 4.17 : Structuration modulaire du graphe de représentation d'un objet

La description des objets selon un contexte correspond alors au résultat de l'héritage sur la partie restreinte de la hiérarchie des modules composée du module spécifique et du module de

base. Par exemple, la partie de la hiérarchie donnant la description de l'objet `not1` pour le contexte Documentation est délimitée par la zone grisée de la figure précédente.

4.3.3.2 Modules et points de vue

Dans cette section, nous cherchons à obtenir la recherche basée sur les modules par application de la stratégie d'héritage par point de vue de ROME. Notre intention dans ce but est d'utiliser les possibilités offertes par la selection de points de vue sur un graphe d'héritage.

En prenant comme principe d'obtenir les modules par des expressions de points de vue, deux problèmes sont à résoudre. Toutes les classes du graphe de représentation d'un objet pouvant déterminer un point de vue, le premier problème est d'identifier ceux qui coïncident avec les modules. Le second problème concerne la réalisation de l'héritage entre les modules, ce qui se ramène à un problème d'articulation de points de vue. Dans cette section, nous allons nous concentrer sur le premier problème.

Identification des modules spécifiques

L'examen des multiples points de vue possibles sur un graphe de représentation des objets met en évidence que les modules spécifiques à chaque contexte peuvent être obtenue en choisisant les classes de partie fonctionnelle respective de `CR-OBJECT`. Pour l'exemple précédent, les points de vue déterminant les modules spécifiques aux contextes schématique et documentation sont respectivement ceux des `Schematique(CR-OBJECT)` et `Documentation(CR-OBJECT)`. Comme le montre la figure 4.18 pour le contexte schematique, ce choix de point de vue inclut bien toutes les classes du module correspondant, ce qui peut également se vérifier en développant la définition d'un point de vue donnée en 4.2.1.

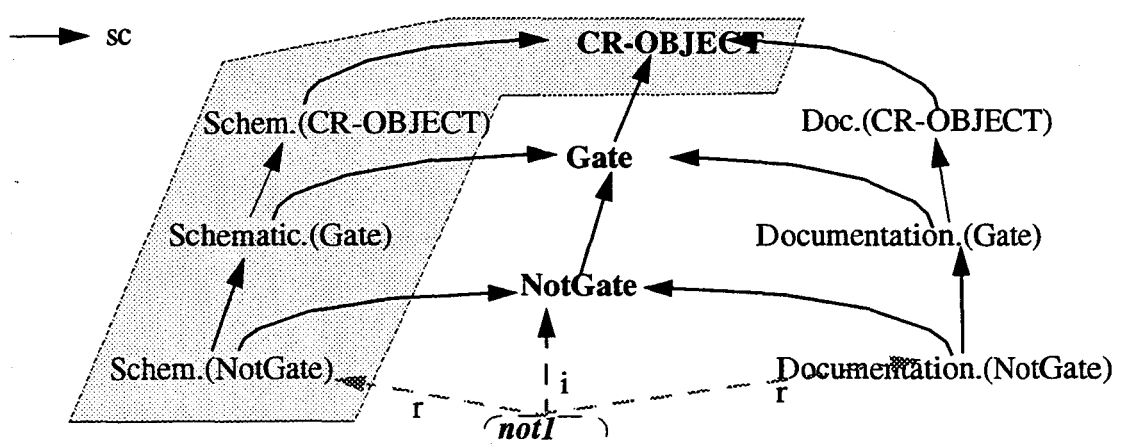


figure 4.18 : Point de vue pour le module spécifique

```
Pdv(not1, Schematic(CR-OBJECT))
     = Rep(not1) ∩ Dep(Schematic(CR-OBJECT))
     = {NotGate, Gate, CR-OBJECT, Schematic(NotGate), Schematic(Gate),
       Schematique(CR-OBJECT),     Documentation(NotGate),     Documenta-
       tion(Gate), Documentation(CR-OBJECT)} ∩ {Schematic(NotGate), Sche-
       matic(Gate), Schematique(CR-OBJECT), CR-OBJECT}
     = {Schematique(NotGate), Schematique(Gate), Schematique(CR-OBJECT),
       CR-OBJECT}
```

Une caractéristique particulièrement intéressante provient des classes utilisées pour exprimer les points de vue relatifs au module spécifique. Celles-ci demeurent identique pour tout objet considéré dans le même contexte, indépendamment de leur classe. Cette caractéristique est obtenue grâce à la généricité de la composante `Dep(C)` de la définition d'un point de vue. Ainsi, pour un objet de la classe `Wire` considéré dans le même contexte d'édition schématique qu'un objet de la classe `NotGate`, la même classe `Schematique(CR-OBJECT)` sera utilisée pour déterminer les points de vues correspondant aux modules spécifiques aux classes de ces deux objets. La figure suivante montre graphiquement cette caractéristique pour ces deux objets.

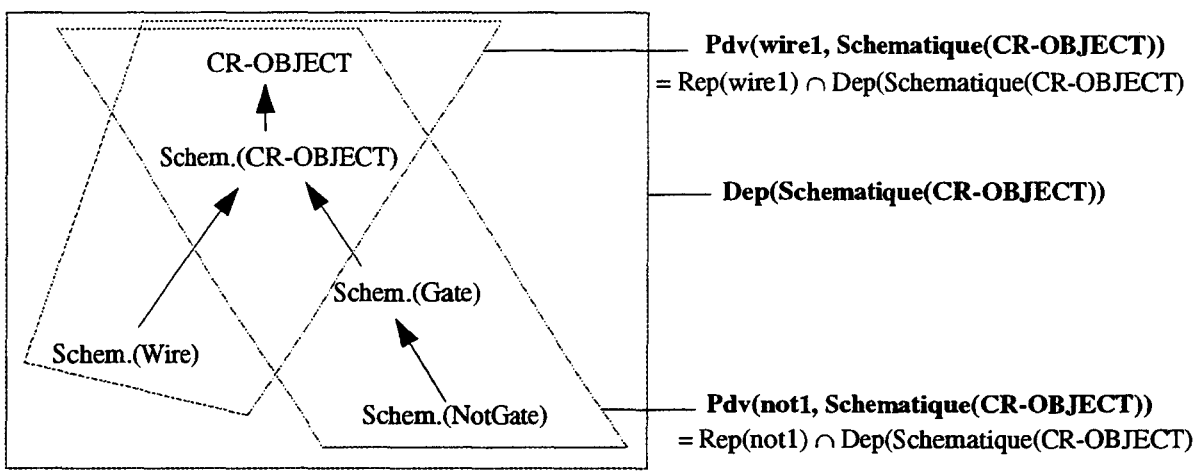


figure 4.19 : Modules spécifiques obtenues à partir de la même classe

Ici, le point de vue de la classe `Schematique(CR-OBJECT)` permet d'obtenir le module spécifique du contexte à la fois pour les objets des classes `NotGate` et `Wire`. De façon générale, la classe `X(CR-OBJECT)` suffit pour déterminer tous les modules spécifiques relatifs à un contexte `X`. Nous reviendrons sur l'intérêt de cette caractéristique au moment d'aborder la partie traitant de l'inter-objet au sein d'un contexte.

Problème d'obtention du module de base

De la même manière que pour les modules spécifiques, on veut pouvoir associer un point de vue correspondant au module de base. Une analyse minutieuse du graphe de représentation fait apparaitre qu'aucun point de vue ne permet d'obtenir ce module. Le problème vient du fait que, tout point de vue d'une classe de base inclut systématiquement toutes les classes de parties fonctionelles situés directement sous celle-ci. Ce problème est illustré ci-dessous lorsqu'on fait le choix de la classe d'instanciation comme point de vue, soit Pdv(not1, NotGate).

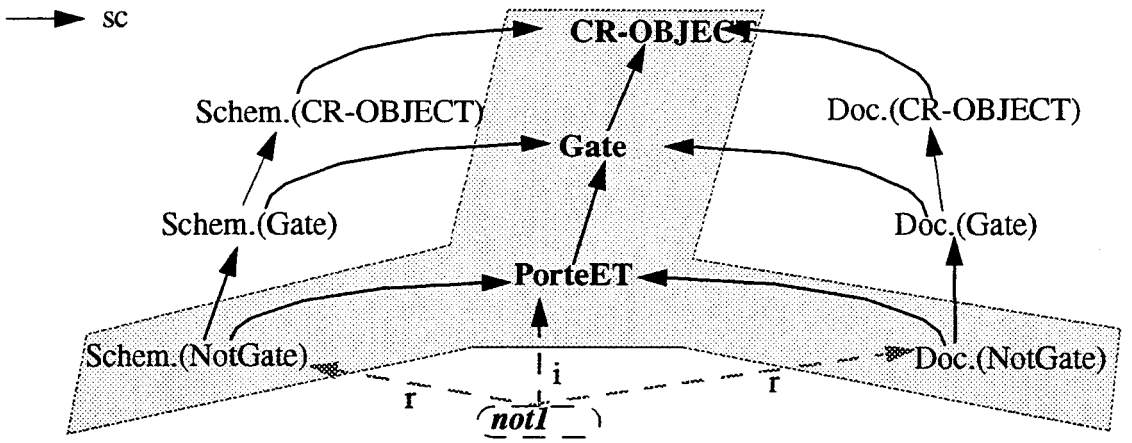


figure 4.20 : Point de vue de la classe d'instanciation

On peut voir que ce point de vue comprend ici les deux sous-classes `Schematic(NotGate)` et `Documentation(NotGate)`. De façon évidente, choisir les points de vue des sur-classes ne permet pas non plus d'obtenir ce module.

L'impossibilité d'obtenir le module de base pose des problèmes lorsqu'il s'agit de contraindre la recherche dans celui-ci ce qui s'avère nécessaire dans deux cas précis:

- celui où il faut hériter pour un contexte les définitions des méthodes issues du module de base, des redéfinitions locales de celles-ci existant dans d'autres contextes.

- celui où il s'agit d'appliquer une méthode du module de base[1], celle-ci étant masquée dans le module spécifique.

Pour remédier au problème, la solution que nous avons choisie pour permettre le calcul du module de base par sélection de point de vue est une réorganisation des classes de base de façon à mettre à plat le graphe des modules.

Plus précisemment, l'idée est de sous-classer les classes de base de façon analogue à ce qui est fait pour les classes de parties fonctionnelles[2]. Pour chaque classe de base, nous introduisons une sous-classe dans laquelle ses méthodes y sont recopiés. Illustrons l'idée sur la classe de ba se NotGate. Cela nous donne le schéma de transformation suivant.

1. par l'opération (base)
2. Le point de départ de cette idée provient du constat, que, contrairement au module de base, les modules spécifiques sont quant à eux calculable par des points vue.

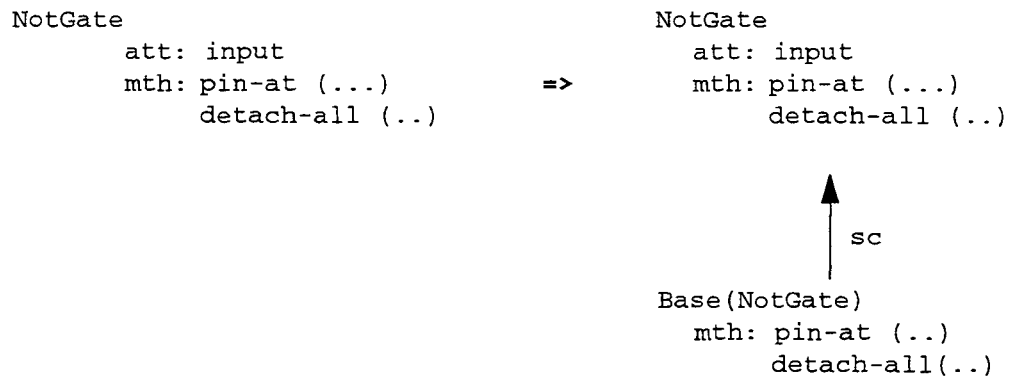


figure 4.21 : Sous-classement systématique de la classe de base

Nous observons que toutes les implantations des méthodes de la classes de base sont ainsi localisées au niveau de la nouvelle sousclase Base(NotGate).

Ce principe s'applique de manière itérée pour chaque classe de base. Pour celles qui sont en relation d'héritage, les sousclasses respectives ainsi obtenues sont mise en relation d'héritage selon le même principe que pour les classes de parties fonctionnelles. Cette héritage est nécessaire pour préserver correctement l'héritage des descriptions de base. Appliquée au graphe des objets NotGate, la nouvelle structuration des classes qui en résulte devient :

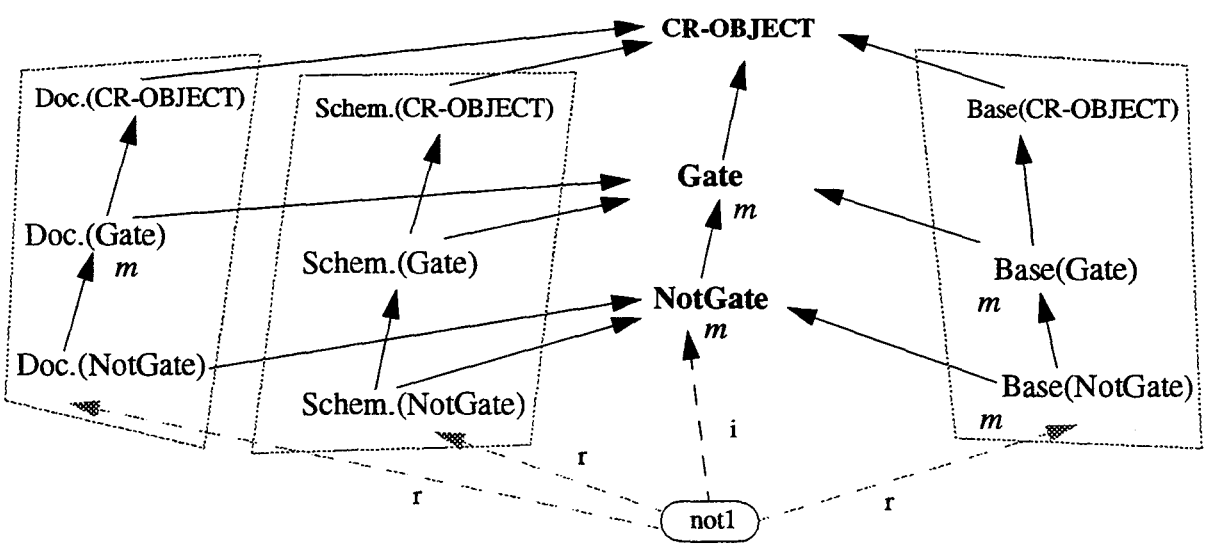


figure 4.22 : Application du sous-classement à plusieurs classes de base en relation d'héritage

L'examen de cette structure fait apparaître un sous-graphe supplémentaire. Chaque classe de ce sous-graphe héritant de la classe de base, elle en détermine une description complète. Ce sous-graphe peut donc être assimilé au module de base qui devient ainsi calculable par point de vue. Le point de vue qu'il convient de prendre dans ce cas est, similairement au module spécifique, celui de la classe racine qui est `Base(CR-OBJECT)`. Il est à remarquer que le caractére générique lié au choix du point de vue de la classe racine et qui a été mis en avant à la section

précédente, vaut aussi à présent pour le module de base.

En considérant le graphe représenté ci-dessus, le choix du point de vue de `Base(CA-OBJET)` permet de considérer prioritairement les caractéristiques définies sous ce point de vue qui sont ici les implantation des méthodes appartenant au module de base.

Le problème du calcul du module de base étant résolu, nous sommes en mesure d'expliciter l'ensemble des modules par des points de vue. Il reste à expliquer comment réaliser l'héritage entre modules sur lequel est fondé la recherche liée à un contexte. C'est ce que nous proposons d'examiner dans ce qui suit.

4.3.3.3 Réalisation de l'héritage de modules

Pour réaliser cet héritage, il faut composer et articuler les deux points de vue correspondant de sorte que la recherche d'une caractéristique se fasse prioritairement dans le module spécifique puis dans le module de base si nécessaire .

Pour prendre en compte ce besoin, ROME ne permet que l'expression d'un point de vue simple au moyen de l'opérateur as (celui d'une classe), notamment pas de point de vue composé explicitement. Cependant, nous avons vu en 4.2.1 que les points de vue sont composés dans leur exploitation qui se fait du plus récemment spécifié au plus ancien, en s'arrêtant dès que la définition de la caractéristique recherchée est trouvée sous l'un des points de vue ou il n'y a plus de points de vue à exploiter. Ainsi, deux points de vue sélectionnés l'un à suite de l'autre sont utilisés dans l'ordre inverse de leur référence.

A partir de cette information sur l'exploitation des points de vue, nous pouvons en déduire les points de vue à établir afin d'obtenir la recherche telle qu'elle a été décrite précédemment. Cela mène à appliquer une double sélection de point de vue déterminée de la façon suivante:

Soit `o` un objet d'une classe de base et `c` un contexte valide pour cette objet, alors les points de vue établis pour rechercher une caractéristique de l'objet dans ce contexte sont dans l'ordre :

1. Le point de vue associé au module de base: `Pdv(o, Base(CR-OBJECT))`

2. Le point de vue associé au module spécifique: `Pdv(o, c(CR-OBJECT))`

Le premier point de vue correspondant au choix du module de base est constant quelque soit le contexte tandis que le second correspondant au choix du module spécifique varie pour chaque contexte. Ces deux points de vue sont les seuls sélectionnés pour résoudre les références aux caractéristiques de l'objet dans un contexte.

Montrons pour l'exemple des objets `PorteNON` représenté à la figure 4.22 que les deux points de vue choisis produisent l'effet souhaité.

Commençons par examiner la recherche de la méthode `m` selon le contexte Documentation pour un tel objet. Pour ce contexte, les deux points de vue sélectionnés sont dans l'ordre celui de la classe `Base(CR-OBJECT)` et celui de `Documentation(CR-OBJECT)` .La méthode m est recherchée sur l'ensemble du graphe ce qui conduit à une ambiguité entre les classes `Documentation(Gate)` et `Base(CR-OBJECT)`. Le point de vue prioritaire étant `Documentation(Gate)` la définition de `m` qui est choisie est par conséquent celle de la classe

`Documentation(Gate)`, redéfinition locale à ce contexte, ce qui est correct.

Considérons à présent la même méthode recherchée suivant le contexte schématique ce qui mène à l'expression du point de vue de la classe `Base(CR-OBJECT)` puis celui de `Schematic(CR-OBJECT)`. La recherche de la méthode `m` donne lieu à une ambiguïté entre les classes `Documentation(Gate)` et `Base(CR-OBJECT)`. Pour résoudre cette ambiguïté, le point de vue `Schematic(CR-OBJECT)` est considéré en premier. Aucune des classes en conflit faisant partie de ce point de vue, ceci amène à exploiter le point de vue de `Base(CR-OBJECT)` et par conséquent à retenir la définition de `m` fourni dans la classe `Base(NotGate)`. C'est bien la définition de la méthode `m` escomptée en respect de l'héritage du module de base pour le contexte.

A travers ces deux exemples, nous constatons que l'exploitation des deux points de vue spécifiés précédemment produit un comportement correct pour la recherche d'une caractéristique selon un contexte. En particulier, il n'y a jamais de choix d'une caractéristique hors contexte. Par ailleurs, du fait de la prise en compte implicite de l'affinement sous un point de vue, on vérifie facilement que cette évaluation reste aussi valide de manière générale, notamment dans le cas où un module comporte une chaine de classes plus importante.

4.4 Contextualisation d'objets

Dans cette section, nous présentons de quelle façon la notion de contexte prend corps dans le cadre de cette application en insistant sur les différentes propriétés qui en découlent.

4.4.1 Contextualisation explicite par sélection de contexte

Pour pouvoir dialoguer avec les objets dans un contexte particulier, indépendamment des autres contextes où ils interviennent, nous introduisons l'opérateur de sélection de contexte in[1]. Cet opérateur peut être utilisé avec toutes les instances des classes de base. Intuitivement, une sélection de contexte permet d'indiquer dans quel contexte on veut communiquer avec l'objet. Sémantiquement, une telle sélection consiste à ne retenir que la partie de l'objet relative au contexte tout en appliquant l'héritage modulaire à cette partie.

La sélection d'un contexte pour un objet est toujours associée à un envoi de message. Cela permet d'activer une des méthodes de l'objet pour le contexte spécifié. Cette association a la forme syntaxique générale suivante:

```
(<objet> 'in <contexte> <selecteur> <arguments>)
```

Dans cette forme, `<objet>` est une instance d'une classe de base et `<contexte>` est un symbole désignant un contexte[2] valide pour cet objet. Le sélecteur de méthode `<selecteur>` doit obligatoirement être significatif pour l'objet dans le contexte spécifié. Nous considérons en particulier comme invalide l'association d'une sélection de contexte avec un envoi de message correspondant à une méthode définie pour un autre contexte. Par ailleurs, si des objets sont passés en arguments du message, ceux-ci seront activés dans le contexte spécifié.

---

1. qui s'apparente dans la forme à l'opérateur as de sélection de point de vue sur un objet.
2. Ce symbole doit forcément correspondre à un nom de contexte, c'est à dire celui utilisé pour la définition des parties fonctionnelles relatives à un même plan.

Donnons quelques exemples d'expression montrant l'utilisation de l'opérateur in.

L'expression suivante consiste à envoyer le message `edit` à un objet circuit en sélectionnant le contexte d'édition schématique:

```
(circuit1 'in 'Schematic 'edit)
```

La réaction de l'objet `circuit1` à cet envoi de message est d'appliquer sa méthode `edit` définie pour le contexte d'édition schématique. Ici, le message correspond à une méthode introduite dans le plan fonctionnel d'édition schématique. La description de base étant toujours prise en compte dans un contexte, nous aurions pu tout aussi bien envoyer à cet objet un message correspondant à une méthode définie dans le plan de base. Soit par exemple pour la méthode `gate` donnant un accès à une porte:

```
(circuit1 'in 'Schematic 'gate 'And3)
```

Pour dialoguer avec une instance d'une autre classe que `Circuit` dans le même contexte, il suffit d'utiliser le même symbole pour le paramètre désignant le contexte dans l'in-expression. L'expression suivante illustre ceci pour un objet porte Et:

```
(and1 'in 'Schematic 'display-on view1)
```

Il est également possible de faire intervenir ce même objet dans un autre contexte, celui de simulation par exemple. Ce qui s'exprime ainsi:

```
(and1 'in 'Simulation 'evaluate)
```

Par ailleurs, toutes les portes sachant s'évaluer dans ce contexte, le même message evaluate peut aussi être envoyé à un objet porte Ou.

```
(or1 'in 'Simulation 'evaluate)
```

De façon générale, il est à noter en comparaison avec un envoi de message classique que:

- le même message peut être envoyé au même objet dans des contextes différents. L'action associée dépend toujours du contexte sélectionné.

- le même message peut être envoyé à des objets différents dans le même contexte. L'action associée dépend toujours du receveur.

Ajoutons par ailleurs que le fait de sélectionner un contexte élimine toute ambiguité lors de l'accès à l'objet.

4.4.2 Contextualisation implicite au sein de l'objet

Lorsqu'une méthode d'un contexte est exécutée par un objet, les attributs et les autres méthodes referencés par celle-ci sont implicitement interprétés selon le même contexte. Cette contextualisation implicite des références garantit la cohérence de l'activation d'un objet dans un contexte.

Afin d'illustrer cette interprétation implicite des références, considérons l'envoi du message `optimize-into` à un objet porte Et dans le contexte optimisation:

```
(and1 'in 'Optimization 'optimize-into circuit2)
```

Pour ce message, la méthode associée est celle fournie par la partie fonctionnelle associée à

la classe AndGate pour ce contexte. Cette méthode y est définie de la façon suivante:

```
optimize-into
       (lambda (circuit)
         (when (self 'can-be-optimized)
           (cond ((= (? 'optimization-type) 1)
                  (self 'remove-distribution circuit))
                 ((or (= (? 'optimization-type) 2)
                      (= (? 'optimization-type) 3))
                  (self 'apply-distribution circuit))))
         (output 'optimize-into circuit))
```

On remarque que cette méthode fait référence à d'autres caractéristiques de l'objet (attributs et méthodes) introduites par la même partie fonctionnelle ou héritées. Ces références sont alors résolues implicitement selon le contexte d'exécution de cette méthode. Dans le cas présent, par exemple, la méthode apply-distribution est trouvée dans Optimize(AndGate). C'est en effet la définition la plus affinée pour cet objet dans le contexte.

Cette contextualisation implicite induit une facilité pour le programmeur qui n'a pas à spécifier explicitement le contexte lorsqu'il programme un objet dans un contexte. Sans cette contextualisation implicite, il serait nécessaire d'analyser finement la description des objets dans les autres contextes pour déterminer quelles sont les caractéristiques à desambiguïser. De plus, il faudrait tenir compte du cas où une caractéristique non ambigüe à un moment donné le devient suite à l'ajout ultérieur de nouveau plan. Ce qui obligerait à altérer la définition initiale afin d'éliminer l'ambiguïté.

Le contexte implicite vaut également pour les méthodes du plan de base, partagées par tous les contextes. Dans ce cas, l'interprétation implicite des caractéristiques référencées dans ces méthodes dépend du contexte de référence (qui existe toujours car on n'est jamais hors contexte). Cette interprétation implicite peut être illustrée en envoyant par exemple le message traverse à un objet de la classe AndGate dans le contexte de simulation (cf chapitre 3 section 8.2):

```
(circuit1 'in 'Normalization 'traverse (Circuit new))
```

Pour ce message, la méthode correspondante se trouve dans la description de base de la classe. Sa définition est la suivante:

```
traverse
     (lambda (circuit)
       (self 'doactions circuit)
       (for-each (lambda (pi)
                   (pi 'traverse circuit))
           (? 'padins)))
```

Dans cette définition, l'appel à la méthode doactions est alors implicitement interprétée selon le contexte de reférence qui est en l'occurence celui spécifié avec l'envoi du message traverse. La méthode doactions possédant une redéfinition locale pour ce contexte, c'est cette dernière qui est appliquée.

Cette caractéristique fait apparaitre une généricité des appels de méthodes exprimés dans celles du plan de base. En effet, les redéfinitions locales des méthodes appelées sont implicitement prise en compte selon le contexte choisi. Le programmeur n'a par conséquent pas

besoin de redéfinir localement les méthodes du plan de base pour que les méthodes appelées soit interprétées selon le bon contexte.

4.4.3 Contextualisation implicite inter-objets

La contextualisation implicite mentionnée à la section précédente s'applique également pour les envois de messages aux autres objets. Ainsi, lorsqu'une méthode d'un contexte est exécutée par un objet, tous les autres objets invoqués par envoi de message à l'intérieur de cette dernière sont implicitement considérés selon ce contexte. Ce qui donne corps à la notion de contexte sur un ensemble d'objets.

L'exemple des portes Ou considérées dans le contexte de simulation permet d'illustrer cette caractéristique. Dans ce contexte, le message evaluate peut être envoyé à de tels objets ce qui s'exprime ainsi:

```
(or1 'in 'Simulation 'evaluate)
```

Ce message entraine l'application de la méthode telle que définie dans la partie fonctionnelle associée à la classe `OrGate` pour ce contexte dont voici le code:

```
evaluate (lambda ()
        (let ((iv1 ((? 'input1) 'get)))
              (iv2 ((? 'input2) 'get))))
          ((? 'output) 'set (or iv1 iv2))))
```

Au niveau de code, on peut noter l'envoi du message `get` aux deux broches d'entrée `input1` et `input2` ainsi que l'envoi du message `set` à la broche de sortie `output`. Pour chacun de ces messages, l'objet concerné est implicitement considéré selon le contexte de la méthode. Ici, en l'occurence, le contexte associé avec ces envois de message est celui de simulation. De manière récurrente, ces objets peuvent à leur tour communiquer par envoi de message avec d'autres objets qui seront alors implicitement considérés selon ce dernier contexte.

Cette contextualisation implicite garantit que les intéractions entre objets se font toujours localement à un contexte. Un objet activé dans un contexte communique avec les autres objets selon le même contexte. Cette caractéristique induit une facilité pour le programmeur qui n'a en aucun cas le besoin de gérer cette localité. En particulier, celui-ci n'a pas à sélectionner explicitement un contexte lors d'un envoi de message correspondant à une méthode du plan de base redéfinie localement dans plusieurs contextes.

Cette contextualisation implicite s'applique également pour les envois de message entre objets exprimé dans leur description de base. Comme pour les appels de méthodes internes à l'objet effectué au même niveau, le contexte implicite associé à ces envois de message dépend du contexte de référence. Un exemple illustrant cette caractéristique est donné dans l'exemple de la méthode `traverse` des circuits considéré lors de la section précédent. Elle apparait au niveau de l'envoi du message `traverse` aux objets padin référencés par la variable `pi` dans la boucle.

4.4.4 Appel de méthodes inter-contextes au sein de l'objet

Cet appel est exprimé en associant l'opérateur de sélection de contexte `in` décrit précédememnt avec un self-message. L'objet designé dans la sélection doit dans ce cas être

l'objet courant (c-a-d `self`). La forme syntaxique générale pour appeler une méthode d'un autre contexte se présente alors comme suit:

```
(self 'in <contexte> <selecteur> <arguments>)
```

Cette forme exprime un self-message ayant pour sélecteur `<selecteur>` avec les arguments `<arguments>` en considérant l'objet courant suivant son contexte `<contexte>`. Pour être valide, le sélecteur de méthode doit être significatif pour l'objet courant dans le contexte spécifié, c'est à dire correspondre à une méthode publique ou privée définie pour l'objet dans ce contexte. D'autre part, les objets des classes de base transmis en arguments du message s'exécuteront dans le contexte sélectionné.

Un exemple d'appel de méthode entre contextes au sein de l'objet est donné par la définition de la méthode `remove-gate` défini dans la partie fonctionnelle associée au circuit pour le contexte d'édition schématique. Le code de cette méthode est le suivant:

```
cmd-removeGate
     (lambda ()
       (when (Confirm 'new 'label "Do you really want to delete this gate)
         (self 'remove-gate (? 'selection))
         (self 'in 'Documentation 'record-remove-gate (? 'selection))
         (self 'invalidate)))
```

Cette méthode appelle la méthode privée `record-remove-gate` définie pour la même classe dans le contexte documentation.

4.5 Implantation métaprogrammée des contextes

Les principes de mise en oeuvre des contextes reposant en grande partie sur les notions et mécanismes existants à la base dans ROME, nous cherchons à en hériter tout en ayant la capacité de les adapter ou de les enrichir selon nos besoins. Nous mettons à profit les capacités réflexives de ROME. Nous procédons en dérivant de nouvelles méta-classes à partir du noyau de base, dans la lignée de ce qui a été fait pour intégrer la représentation multiple et évolutive (cf 4.2.2). Le résultat est un triplet de classes spécifiques (dont deux méta-classes) qui fournissent l'interface de programmation par contextes et se chargent automatiquement de réaliser (en les systématisant) les principes sous-jacents décrits au cours des deux sections précédentes.

Dans la prochaine section, nous présentons ces classes et les situons par rapport aux classes du noyau de ROME. Les ajouts et les adaptations introduites par ces classes pour les contextes sont ensuite analysés.

4.5.1 Classes de l'implantation

L'extension pour les contextes se résume à trois classes que nous avons déjà mentionnées lors de la partie 3:

- `CR-OBJECT` est la classe de base la plus générale. Tout objet intervenant dans plusieurs contextes est représentant de cette classe. Cette classe sert également à obtenir la structuration par plans des graphes d'héritage multiple.

- CA-CLASS est la (méta-)classe de toutes les classes de base abstraites. Elle introduit la fonctionnalité d'enrichissement contextualisé pour les classes de base .

- CI-CLASS est la (méta-)classe de toutes les classes de base concrètes. Elle hérite la fonctionnalité d'enrichissement contextualisé de CA-CLASS et adapte la fonctionnalité de représentation multiple et évolutive héritée de RME-CLASS aux contextes.

La figure 4.23 montre les liens d'instanciation et d'héritage qu'entretiennent ces trois classes avec celles du noyau de base. Deux classes de base (Gate2 et AndGate) sont également figurées pour montrer leurs liens avec ces classes.

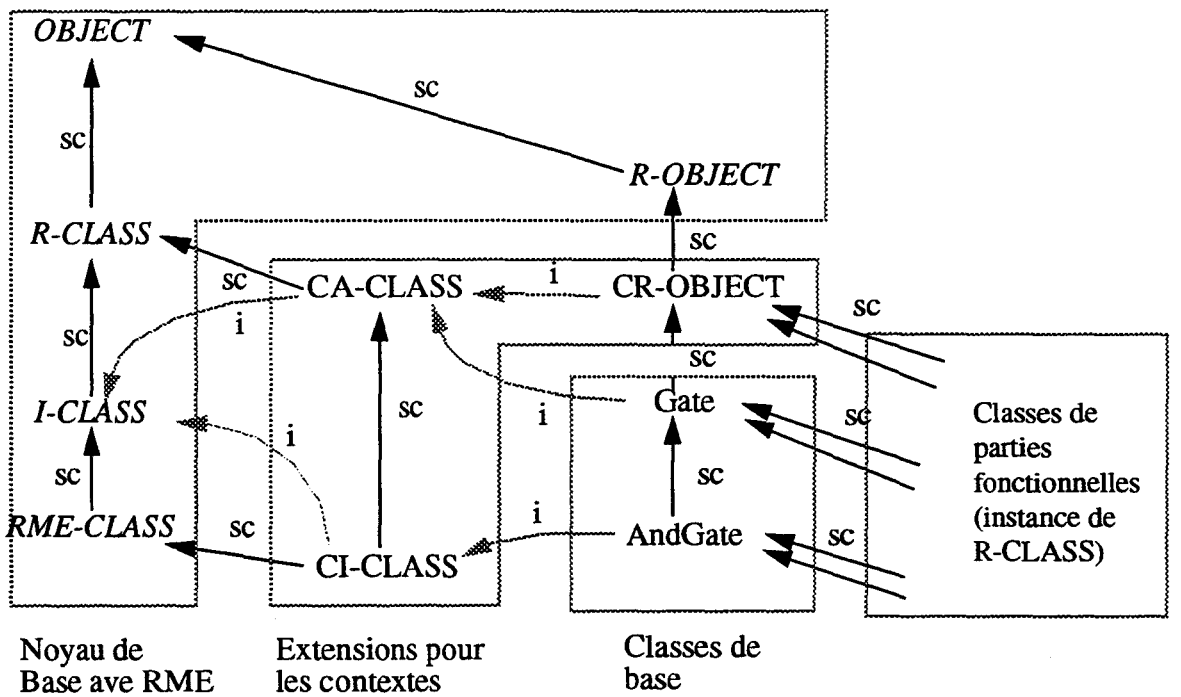


figure 4.23 : Noyau de base avec les extensions pour les contextes

CA-CLASS définit les classes de base abstraites qui sont des classes possédant la fonctionnalité d'abstraction d'objets définie par R-CLASS (attributs, méthodes, surcl,...) mais n'ont pas celle d'instanciation. Cette classe est donc sous-classe de R-CLASS. Par ailleurs, elle est génératrice de ces classes, et doit donc disposer de la fonctionnalité d'instanciation définie par I-CLASS. Elle en est donc instance.

CI-CLASS définit les classes de base concrètes qui sont des classes de base particulières sachant instancier et gérer l'exécution d'objets à représentation multiple selon les contextes. Cette classe est donc sous-classe de CA-CLASS pour en hériter la fonctionnalité d'enrichissement contextualisé et sous-classe de RME-CLASS pour en hériter la fonctionnalité de représentation multiple. Notons qu'il n'y pas de conflit entre les caractéristiques héritées de CA-CLASS et celles héritées de RME-CLASS et I-CLASS. Par ailleurs, de façon similaire à CA-CLASS, CI-CLASS est génératrice de ces classes ce qui conduit à en faire une instance de I-CLASS.

Enfin CR-OBJECT, qui est la classe de base la plus générale, est instance de CA-CLASS puisqu'elle est destinée à être enrichie pour chaque contexte (cf 4.3.1.2) mais n'est pas succeptible d'être instanciée. Elle est également sous-classe de R-OBJECT de façon à doter les objets issus des classes de base de la capacité de représentation multiple.

4.5.2 Génération des classes de parties fonctionnelles

Dans notre cas, nous faisons le choix d'associer cette responsabilité aux classes de base elles-mêmes. Ce choix nous parait pertinent dans la mesure où toute classe de partie fonctionnelle est relative à une classe, pour un contexte donné. L'implantation duale consisterait à introduire des objets contextes regroupant toutes les classes de parties fonctionnelles de tous les objets. Le premier choix, que nous retenons, privilégie une implantation orientée objet en regroupant toutes les classes de parties fonctionnelle d'un objet dans sa classe de base. Celle-ci peut ainsi les gérer, en particulier les créer par le message `extend-in` et maintenir des liens vers ces classes.

En ce qui concerne les liens d'héritage à établir entre les classes de parties fonctionnelles, leur établissement peut également être pris en charge par les classes de base. Chaque classe de base détenant des liens vers ses classes de parties fonctionnelles, il suffit à une telle classe d'interroger sa sur-classe pour connaitre la classe de parties fonctionnelle à spécifier au moment de la génération. Cela suppose l'existence pour les classes de base d'une méthode donnant accès aux classes de parties fonctionnelles en fonction du contexte.

La prise en charge des parties fonctionnelles est définie dans `CA-CLASS`. Par rapport à `R-CLASS`, cette nouvelle méta-classe ajoute notamment:

- l'attribut `cntxt-rcls` qui contient une liste d'associations de la forme (<contexte>,<classe de partie fonctionnelle>) maintenant la correspondance entre les contextes et les classes de représentation générées à partir des parties fonctionnelles;
- la méthode `extend-in` permettant la définition des parties fonctionnelles et effectuant le traitement de génération associé;
- la méthode `rclass-in` qui retourne la classe de partie fonctionnelle utilisée pour le contexte fourni en paramètre. Cette méthode est exploitée par `extend-in` pour mettre en place l'héritage local.

Ces caractéristiques sont héritées par `CI-CLASS`. Les classes de base concrètes ont donc également l'attribut `cntxt-rcls` et savent également repondre au message `extend-in` et `rclass-in`.

La méthode `extend-in` est définie de la façon suivante:

```
1  extend-in
2     (lambda (context . features)
3       (let ((newrcls '())
4             (supcl '())
5             (attrs (crm:getprop 'attributes features))
6             (publmths (crm:getprop 'publmth features)))
7             (privmths (crm:getprop 'privmth features)))
8
9         (if (not (assq context (? 'cntxt-rcls)))
10          (self 'rome-error self "already extended in context " context))
11
12        (set! supcl ((car (? supercld)) 'rclass-in context))
13        (set! newrcls
14                  (R-CLASS 'new
```

```
15                    'supercl (list self supcl)
16                    'attributes attrs
17                    'privmth privmths
18                    'publmth publmths))
19        (<- 'cntxt-rcls
20            (cons (list context newrcls) (? 'cntxt-rcls)))))))))
```

Après récupération des caractéristiques fournies pour la partie fonctionnelle (ligne 5 à 7), cette méthode vérifie qu'il n'existe pas déjà de partie fonctionnelle définie pour la classe dans le contexte (ligne 9 et 10). La sur-classe est ensuite interrogée pour connaitre la classe de partie fonctionnelle à spécifier lors la génération ce qui se fait en lui envoyant le message `rclass-in` paramétré par le contexte (ligne 14). La génération est effectuée en prenant comme sur-classes directes l'objet courant (c'est à dire la classe de base ayant reçu la demande d'une nouvelle partie fonctionnelle) et la classe de partie fonctionnelle obtenue par `rclass-in` (ligne 13 à 18). Les caractéristiques pour la classe à créer sont celles fournies en paramètres. Finalement, la classe de représentation résultante est associée au contexte dans la a-liste `cntxt-rcls` (ligne 19 et 20).

La méthode `rclass-in` employée dans `extend-in` est définie ainsi:

```
1  rclass-in
2    (lambda (context)
3      (let* ((pr (assq context (? 'cntxt-rcls))
4        (if pr
5          (if (eq? self CA-OBJET)
6            ()
7            ((car (? 'supercld)) 'r-class-in context)
8          (car pr)))
```

La définition de cette méthode est récursive. Si la classe de base pour laquelle la méthode est invoquée possède une classe de partie fonctionnelle pour le contexte, celle-ci est retournée (ligne 3, 4 et ligne 8). Dans le cas contraire, le message est renvoyé à la sur-classe (ligne 7) et ainsi de suite recursivement. Le processus s'arrête dès qu'une sur-classe (directe ou indirecte) de la classe de base possède une classe de partie fonctionnelle auquel cas cette dernière est renvoyée ou que la classe de base est `CA-OBJET`. Dans le cas où cette classe n'a pas non plus de classe de partie fonctionnelle correspondante[1], la liste vide est renvoyée.

La figure 4.24 représente l'état de la classe de base `NotGate`, instance de `CI-CLASS`, après les définitions pour celle-ci des parties fonctionnelles relatives aux contextes schématique et Simulation.

1. ce qui s'interprète comme l'absence de déclaration de plan fonctionnel designé par le paramètre context

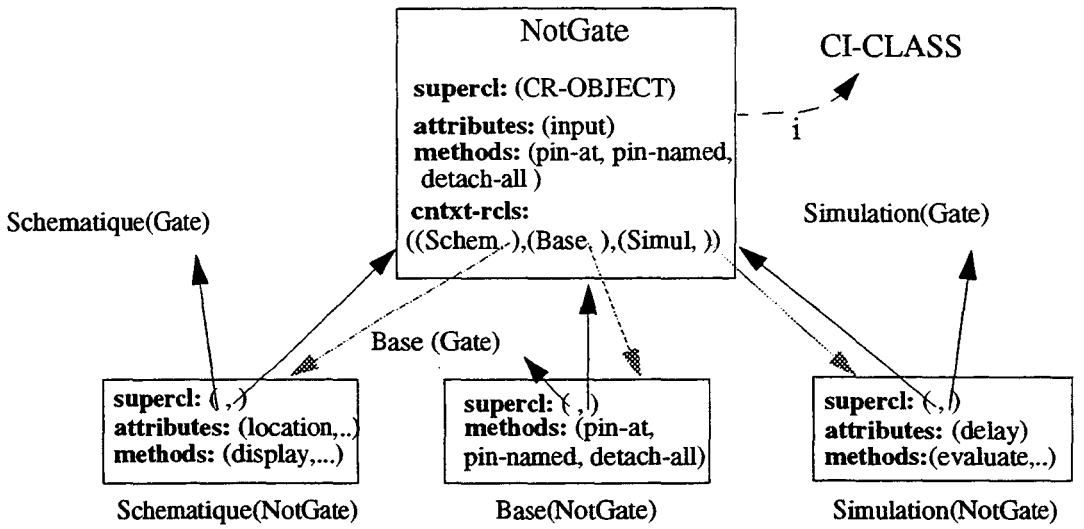


figure 4.24 : Liens entre une classe de base et ses classes de parties fonctionnelles

Sur cette figure, on remarque les liens inverse de la classe de base vers les classes de partie fonctionnelle. De cette façon, chaque classe de base a connaissance des éléments décrivant ses instances et reste aussi seule détentrice de ces éléments.

Enrichissement implicite

Le premier cas d'enrichissement implicite (celui où une classe enrichie explicitement a sa sur-classe directe qui l'est implicitement, cf 4.3.1.2) est directement prise en charge dans la méthode `extend-in`. Cette prise en charge intervient lors de l'appel à la méthode `rclass-in` qui retourne la classe de partie fonctionnelle utilisée pour une classe dans un contexte.

Pour prendre en compte le second cas d'enrichissement implicite (celui où il faut faire hériter localement les classes de base concrètes non enrichies explicitement de leur sur-classe, (cf 4.3.1.2) nous adoptons également une approche répartie. Le principe est de parcourir la hiérarchie des classes de base et de faire en sorte que chaque classe de base décide si elle doit être implicitement enrichie pour le contexte. Dans le cas d'une classe de base abstraite, aucun enrichissement implicite n'est effectué. Dans le cas d'une classe de base concrète, cela dépend en fait de l'existence d'une partie fonctionnelle associée à celle-ci pour le contexte.

Pour implanter cette approche répartie, nous associons une nouvelle méthode aux classes de base appelée `implicit-extend-in`. Dans `CA-CLASS`, cette méthode est définie comme suit:

```
1  implicit-extend-in
2     (lambda (context)
3       (letrec ((cral (? 'contxt-rcls))
4                (memv-al (lambda (v al)
5                           (cond ((null? al) #f)
6                                 ((eq? v (cadar al)) #t)
7                                 (else (memv-al v (cdr al)))))))))
8        (for-each (lambda (cl)
9                    (if (not (memv-al cl cral)
10                       (cl 'implicit-extend context)))
```

```
11              (? 'subcld))))
```

L'analyse de ce code montre que le traitement consiste simplement à activer la même méthode sur les sous-classes directes (ligne 8 à 11) qui sont des classes de base. Le test précédent l'appel (ligne 9) sert à filtrer ces classes. Il est requis car les classes de partie fonctionnelle[1] font également partie des sous-classes directes d'une classe de base.

La méthode précédente est redéfinie dans CI-CLASS pour générer éventuellement la partie fonctionnelle requise (cf 4.3.1). Ce qui mène à la définition suivante:

```
1  implicit-extend-in
2     (lambda (context)
3        (if (assq context (? 'cntxt-rcls))
4           (if (self 'inherit-abstract)
5              (self 'rome-error "concrete class " self
6                    " can not be implicitly extended in " context
7                    " with inherited abstract methods")
8              (self 'extend-in context
9                 'attributes '()
10                'privatemethods '()
11                'publicmethods '())))
12       (super))
```

Un premier test détermine si la classe de base concrète (designé ici par self) possède déjà une classe de partie fonctionnelle pour le contexte (ligne 3). Si ce n'est pas le cas, on procède à son enrichissement implicite. Un second test est alors accompli (appel à `inherit-abstract` pour s'assurer que cette classe n'hérite pas implicitement de méthodes abstraites (ligne 4) auquel cas l'erreur correspondante est déclenchée. Dans le cas contraire, une nouvelle partie fonctionnelle sans caractéristique est définie pour la classe en question et pour le contexte fourni en paramètre. Cela se fait en réutilisant la méthode `extend-in` (ligne 7 à 10) ce qui garantit la prise en compte de l'héritage local. La méthode se termine en invoquant la super-méthode (ligne 11) décrite précédemment, permettant ainsi de poursuivre le traitement au niveau des sous-classes (des instances de CI-CLASS) si celles-ci existent.

A partir de ces définitions, tous les enrichissements implicites liés à un plan fonctionnel peuvent être obtenue en activant la méthode `implicit-extend-in` sur la classe racine CA-OBJET avec comme argument le symbole désignant le plan.

Classes de représentation pour le module de base

Les sous-classes nécessaires au calcul du module de base sont créées de façon systématique lors de l'initialisation des classes de base. Pour chaque classe de base, une classes de représentation reprenant l'implantation des méthodes de la classe de base est ainsi générée. La méthode d'initialisation[2] (méthode initialize) de ces classes est donc spécialisée au niveau de CA-CLASS de façon à ajouter le traitement créant la sous-classe.

```
1  initialize
2     (lambda (init-lst)
3        (super)
```

1. Or ces classes ne savent pas répondre au message implicit-extend-in .
2. introduite dans la classe R-CLASS

```
4       (let ((publmths (crm:getprop 'publicmethods features)))
5            (privmths (crm:getprop 'privatemethods features)))
6         (self 'extend-in 'Base
7               'attributes '()
8               'privatemethods privmths
9               'publicmethods publmths)))
```

Techniquement, nous obtenons la sous-classe en réutilisant la méthode `extend-in` munie en paramètres du symbole `Base` pour le contexte et des méthodes de la classe pour les caractéristiques (ligne 6 à 7) . Cette réutilisation permet d'une part une prise en charge immédiate de l'héritage entre les sous-classes ainsi créées. D'autre part, le rattachement des objets de la classe de base à cette sousclasse est ainsi directement accompli lors de l'instanciation (cf prochaine section).

4.5.3 Représentation multiple des objets selon les contextes

Dans la lignée de la proposition précédentes, nous laissons les classes de base prendre en charge la rattachement systématique des objets à toutes les classes de parties fonctionnelles. Ce choix est d'autant plus légitime qu'une classe de base connait à quelles classes de parties fonctionnelles il faut rattacher ses objets. En outre, une classe de base reste selon ce choix seul responsable de la représentation de ses objets. Il est à noter que ce comportement ne concerne que les classes de base concrètes.

Pour mettre en place la systématisation de la représentation multiple, nous spécialisons au niveau de `CI-CLASS` la création d'objet standard héritée de `I-CLASS` de façon à y incorporer le traitement des liens de représentation. Ce traitement ayant trait à la structure des objets, il doit intervenir lors de la phase d'allocation. Ceci nous amène à spécialiser plus précisemment la méthode `allocate` d'allocation des objets pour `CI-CLASS`, et donc implicitement de la méthode `new`, de la façon suivante:

```
1    allocate
2       (lambda ()
3         (let ((newobj (super)))
4           (for-each (lambda (pr)
5                       (newobj 'r+ (cadr pr))
6                     (<- 'cntxt-rcls))
7           newobj))
```

L'analyse de ce code montre que le nouvel objet est relié systématiquement à l'ensemble des classes de parties fonctionnelles au moyen de l'opérateur `r+` (cf 4.2.1). Ces classes sont obtenues en accèdant l'attribut `cntxt-rcls` hérité de `CA-CLASS` qui contient les associations (<contexte>, <classe de partie fonctionnelle>) maintenues localement par chaque classe (cf 4.5.2).

En plus de la méthode allocate, une autre méthode héritée de `RME-CLASS` est redéfinie au niveau de `I-CLASS` pour la représentation des objets. Il s'agit de la méthode `add-rlink` (cf 4.2.2.4) redéfinie pour relacher le principe de représentation minimale par rapport aux sur-classes (cf 4.3.2).

4.5.4 Mise en oeuvre de la recherche

Nouvelle gestion des points de vue pour les contextes

Pour mettre en place les contextualisations présentées en 4.4, nous proposons une gestion particulière de la pile des points de vue, différente de celle appliquée par défaut en ROME. Rappelons que dans ce dernier cas, cette pile est utilisée pour résoudre les ambiguités survenant au cours de l'exécution d'un objet. Elle est mise à jour dans deux cas de figure:

- Elle peut l'être de façon explicite par le programmeur au moyen de l'opérateur `as`.
- Elle l'est aussi de façon implicite à chaque application de méthode en fonction de sa classe de définition.

La gestion des points de vue proposée pour les contextes repose sur les principes suivants :

- Les points de vue empilés correspondent uniquement à des modules;
- Les points de vue sont toujours empilés par couple, jamais individuellement. Ces couples ont toujours la forme (`Base(CR-OBJECT, C(CR-OBJECT)`);
- Une sélection de contexte (in-expression) entraine l'empilement du couple de points de vue correspondant. Ce couple est dépilé après exécution de la méthode associée.
- Le contenu de la pile reste inchangé entre les applications de méthodes (contexte implicite intra-objet)
- Le contenu de la pile reste inchangé lors d'envois de message entre objets (contexte implicite inter-objets).

Soulignons que cette gestion particulière pour les contextes ne nécessite pas de modifier la façon dont la recherche de base exploite le contenu de la pile en cas d'ambiguités: cette exploitation se fait toujours en considérant en priorité les points de vue les plus récents. La recherche de base par points de vue (cf méthode `lookup-mth`) peut donc être conservée pour les contextes.

Montrons l'évolution de la pile de points de vue en appliquant ces principes à l'exemple considéré en 4.4.4. Soit l'envoi du message `cmd-removeGate` à un objet circuit dans le contexte d'édition, le contenu de la pile étant vide. La sélection de contexte associée à ce message a pour effet d'empiler dans l'ordre les points de vue `Base(CR-OBJECT)` et `Edition(CR-OBJECT)` au sommet de la pile. Suite à ce premier empilement, le contenu de la pile se présente ainsi:

```
s => Edition(CR-OBJECT)
     Base(CR-OBJECT)
------------------------
```

La recherche de la méthode `cmd-removeGate` sur tout le graphe de représentation de l'objet ne conduit aucune ambiguïté. La définition de cette méthode est trouvée dans la partie fonctionnelle associé à Circuit dans le contexte edition. Celle-ci est donc appliquée, ce qui laisse inchangé la pile. La méthode `remove-gate` du plan de base à laquelle il est fait référence en premier dans cette méthode est ensuite recherchée. De façon à illustrer l'exploitation de la pile, supposons pour cette méthode introduite dans le plan de base qu'il existe une redéfinition locale dans un autre contexte `C`. La recherche sur tout le graphe de cette méthode détecte par conséquent une ambiguité entre la classe `Base(Circuit)` et la classe `C(Circuit)` ce qui

conduit à faire intervenir la pile de point de vue. Ceci amène à tester l'inclusion des classes en conflit à ce niveau par rapport aux points de vue contenus dans la pile. Pour le premier point de vue, `Edition(CR-OBJECT)`, l'ambiguité persiste car les deux classes en question n'y sont pas incluses. Celles-ci sont donc confrontées au point de vue suivant, en l'occurence `Base(CA-OBJET)`. De façon évidente, seule la classe `Base(Circuit)` fait partie de ce point de vue. C'est donc la définition de `removeGate` détenue par cette classe qui est choisie.

Considérons à présent le changement de contexte qui suit l'appel à la méthode `remove-gate`. Ce changement de contexte provoque l'empilement des point de vue `Base(CR-OBJECT)` et `Documentation(CA-OBJET)` correspondant au module de base et au module spécifique pour le contexte documentation. A cet instant, la pile possède le contenu suivant:

```
s => Documentation(CR-OBJECT)
     Base(CR-OBJECT)
     Edition(CR-OBJECT)
     Base(CR-OBJECT)
-----------------------------
```

Ces deux derniers points de vue deviennent les points de vue prioritaires en cas d'ambiguïtés. La méthode `record-remove-gate` est donc recherchée en priorité sous le point de vue `Documentation(CR-OBJECT)`. Il en est de même pour les caractéristiques de l'objet référencées à l'intérieur de cette méthode et de façon générale pour tous les accès aux caractéristiques (internes et externes) des objets qui s'ensuivent dans le contexte.

Par rapport à ce dernier point, il est important de rappeler que toute ambiguïté est forcément résolue sous les deux points de vue situés au sommet. Dans le cas présent, les points de vue `Edition(CR-OBJECT)` et `Base(CR-OBJECT)` ne sont donc jamais considérés pour résoudre les références aux caractéristiques des objets dans ce contexte.

Après exécution de la méthode ayant provoqué le changement de contexte, les deux points de vue `Documentation(CR-OBJECT)` et `Base(CR-OBJECT)` sont dépilés, plaçant les points `Base(CR-OBJECT)` en sommet de pile. Le contexte initial est ainsi restitué. L'appel à la méthode `invalidate` de l'objet et les références aux caractéristiques des objets qui s'ensuivent sont alors résolues en considérant ces deux points de vue.

Au travers de cet exemple, nous pouvons constater des différences importantes entre la gestion des points de vue pour les contextes et celle appliquée par défaut en ROME:

- La pile de points de vue est ici gérée comme une pile de contextes;
- Les deux points de vue situés au sommet déterminent le contexte d'exécution pour plusieurs objets. Cette factorisation des points de vue entre les objets n'existe pas en ROME, chaque exécution d'objet donnant lieu à des empilements particuliers;
- La pile de point de vue n'est mise à jour qu'une seule fois pour chaque contexte. Au contraire, la pile de points de vue est modifiée à chaque application de méthode en ROME.

Il en résulte une gestion différente des points de vue dans le cadre des contextes.

Sélection de contexte

L'opérateur de sélection de contexte (`in`) est traité de façon similaire à l'opérateur de sélection de point de vue (`as`). Son traitement se situe dans la méthode `handle-msg` qui est redéfinie au niveau de `CI-CLASS` dans ce but. L'implantation cette méthode se décline comme suit:

```
1    handle-msg
2      (lambda (rcvr sender sel args)
3        (let ((cdef '())
4              (selin (eq? sel 'in))
5              (res '())
6
7          (if selin
8            (begin
9              (#push-pov (CR-OBJECT 'rclass-in 'Base))
10             (#push-pov (CR-OBJECT 'rclass-in (car args))
11             (set! sel (caddr args))
12             (set! args (cadddr args)))
13           (if (eq? sel 'as)
14             (rcvr 'rome-error "incorrect use of AS !"))
15
16         (set! res (super))
17         (if selin
18           (begin
19             (#pop-pov)
20             (#pop-pov)))
21         res)
```

La méthode commence par tester si le sélecteur du message correspond à l'opérateur de sélection de contexte (ligne 4 et 7). Dans la négative, un test supplémentaire est effectué pour vérifier que l'opérateur de sélection de point de vue n'est pas appliqué à l'objet receveur (ligne 13 et 14). Dans l'affirmative, les deux points de vue correspondant au contexte passé en paramètre sont déterminés par l'envoi du message `rclass-in` à `CR-OBJECT` et empilés sur la pile des points de vue (ligne 9 et 10) à l'aide de la primitive `#push-pov`. Ces empilements sont suivis de la mise à jour du sélecteur et des arguments .

La super-méthode, c'est à dire celle définie dans `RME-CLASS` (cf 4.2.2), est ensuite invoquée (ligne 16) conduisant à appliquer le mécanisme de résolution de message pour les objets à représentation multiple. Avant de rendre le résultat de cette résolution, un dernier test est effectué pour déterminer si il y a eu sélection de contexte auquel cas les points de vue correspondant ont besoin d'être dépilés (ligne 17 à 20).

Gestion des points de vue pour les contextes

Nous avons vu en 4.2.2 que le traitement standard de l'application des méthodes est défini par la méthode `apply-meth` associée à la méta-classe circulaire `I-CLASS`. Pour prendre en compte la préservation du contexte, cette méthode est redéfinie au niveau de `CI-CLASS` de la façon suivante:

```
1 apply-meth
2    (lambda (rcvr cls sel args)
3      (let ((res ()) (old-sup super) (old-base base))
```

```
4         (set! super (lambda () (self 'handle-supercall rcvr cdef sel args)))
5         (set! base (if (memq (CR-OBJECT 'rclass-in 'Base) (cls 'superclg))
6                        (lambda () (rec 'rome-error "incorrect use of (base)"))
7                        (lambda ()
8                          (self 'handle-basecall rec sel args)))
9
10        (set! self rcvr)
11
12        (set! res (apply (cdef 'get-method sel) (list rcvr args))
13        (set! super old-sup)
14        (set! base old-base)
15        res)))))
```

Comparé au traitement standard (cf 4.2.2.3) hérité ici de I-CLASS, une première différence porte sur l'inférence implicite des points de vue. Cette inférence est supprimée au niveau de cette redéfinition, laissant l'état de la pile inchangé entre les applications de méthodes. Une second différence avec le traitement standard concerne la mise en place de l'opération `(base)` selon un procédé analogue à `(super)` (ligne 5 à 8). Le corps de la lambda-expression associée est élaboré de façon à invoquer le message `handle-basecall` sur l'objet courant (c-a-d le méta-objet). Le cas où l'opération `(base)` est utilisé incorrectement dans une méthode de la description de base, est résolu en associant une lambda-expression chargée de déclencher l'erreur à l'exécution.

Traitement de l'opérateur base

Le traitement de l'opération `(base)` requiert des actions spécifiques lors de la résolution de message. Il s'agit notamment de contraindre le lookup au module de base sans changer le contexte d'exécution. Ceci nous amène à définir la méthode `handle-basecall` au niveau de `CI-CLASS` avec le code suivant:

```
1  handle-basecall
2     (lambda (rec sel args)
3      (let ((cdef '()))
4        (#push-pov (CR-OBJECT 'class-for-context 'Base))
5        (set! cdef (self 'lookup-mth sel))
6        (#pop-pov)
7
8        (if (null? cdef)
9         (rec 'rome-error "there is no base method corresponding to " sel)
10        (self 'applymeth rec cdef sel args))))
11
```

Pour contraindre la recherche au module de base, nous empilons (par la primitive `#push-pov`) un point de vue avec la classe `Base(CR-OBJECT)` au dessus du couple déterminé pour le contexte. Ceci permet de rendre prioritaire la recherche sous ce point de vue. La méthode recherchée est forcément trouvée sous ce point de vue. Une fois le lookup terminé, cette classe est aussitôt dépilée (par la primitive `#pop-pov`) de façon à rétablir l'état de la pile pour l'application de la méthode ainsi trouvée. Il est à remarquer que le lookup de base (méthode `lookup-mth`) hérité de `RME-CLASS` est conservé dans tous les cas de figure.

Remarque sur l'accès aux attributs

Les méthodes `read-attribute` et `write-attribute` réifiant l'accès aux attributs ne sont

pas redéfinies pour les contextes. C'est donc le traitement standard de ces méthodes défini dans I-CLASS (cf 4.2.2.3) et basé sur le graphe d'héritage de la classe dans laquelle est exprimé l'accès qui est utilisé. Dans le cas présent, ce traitement s'avère suffisant grâce à la structure modulaire du graphe d'héritage construit pour représenter les parties fonctionnelles et la description de base. Cette structure modulaire garantit en effet que les attributs accèdés dans une partie fonctionnelle sont toujours choisi parmi ceux défini dans les classes du contexte (c'est à dire les classes de parties fonctionnelles et les classes de base).

4.6 Conclusion

Dans ce chapitre, nous avons étudié l'application de CROME à ROME en tant que langage de programmation. Cette réalisation a été obtenue en prenant pour point de départ les capacités de représentation multiple par point de vue des objets offertes à la base dans ROME. Nous en avons alors proposé une systématisation à travers les notions de plan, d'héritage modulaire et de contexte. Ceci nous a amené à prendre en compte les dimensions relatives au graphe d'héritage et à un ensemble d'objets qui ne sont pas traitées en ROME du fait de la granularité fine des points de vue.

Le langage obtenu est un langage de programmation par contexte d'objets en ROME. Ce langage rend compte de toutes les notions et reprend l'intégralité des principes précisés par le cadre. Il préserve également l'ensemble des propriétés explicitées à la fin du chapitre précédent. Nous avons également donné dans ce chapitre tous les détails pour une implantation méta-programmée de ce langage.

5

Application de CROME à l'environnement Smalltalk

5.1 Introduction

Contrairement à l'application de CROME à ROME en tant que langage de programmation présentée dans le chapitre précédent, il s'agit ici d'évaluer la généralité des principes en montrant son application à un environnement de développement. L'environnement choisi est celui de Smalltalk [Goldberg 83] pour ses capacités d'ouverture. Nous nous imposons comme contrainte d'obtenir des classes et des objets Smalltalk standard ce qui nous amène à intervenir uniquement au niveau de l'environnement de développement. Ainsi, nous partons des techniques des catégories de classes et de méthodes qui offrent les bases pour une structuration fonctionnelle des applications. Nous proposons ici de les étendre et de les systématiser au travers de la notion de plan. Le résultat est un environnement de conception d'objets Smalltalk orientée contexte offrant en particulier un flâneur adapté. Nous insistons sur le fait que cette extension (tout comme la technique des catégories) se situe uniquement au niveau de l'environnement de développement et n'a aucune conséquence opérationnelle sur le langage lui-même.

Le reste de ce chapitre est structuré comme suit. Nous commençons par expliciter les difficultés posées pour obtenir une description structurée par contextes des classes Smalltalk. Nous proposons alors une solution qui repose sur l'extension de la technique des catégories. Cette solution est exploitée pour offrir un flâneur orienté contexte que nous présentons. Des expériences de développement réalisées avec cet environnement sont présentées. Nous terminons en précisant les principaux éléments d'implantation ainsi que les aspects non supportés par cette mise en œuvre [1].

5.2 Prise en compte des contextes en Smalltalk

Nous cherchons à expliciter les contextes fonctionnels des objets en Smalltalk. Dans un premier temps, nous introduisons l'exemple de CAO de circuits en Smalltalk. Cet exemple est un peu différent dans sa conception de celui utilisé pour présenter le modèle car il a été conçu pour prendre en compte des circuits hiérarchiques[2]. Nous examinons ensuite la technique des catégories, présente en standard dans la plupart des systèmes Smalltalk, qui offre les bases pour une structuration par contextes des classes.

5.2.1 Exemple

Smalltalk relève de l'approche objet classique telle qu'elle a été développée à la section 3.2 du premier chapitre. Les objets Smalltalk sont en effet décrits de manière monolithique par leur classe et il n'y pas de moyens structurants pour rendre compte des fonctions. Nous rappelons brièvement les difficultés que cela pose.

---

1. encapsulation interne relatives aux fonctions, héritage modulaire, envoi de message paramétré par contextes.
2. Ceci se traduit au niveau de l'exemple par l'apparition de nouvelles classes comme MacroGate, LogicGate qui remplace les classes Circuit et Gate. Seuls trois contextes ont été développés en Smalltalk: Edition Schématique, Simulation, Inspection hiérarchique.

Pour décrire notre exemple en Smalltalk, la solution consiste à introduire des classes correspondant aux objets qui sont issues de la conception: `Logicate, AndGate, NotGate, OrGate, MacroGate, Pad, PadIn, PadOut, Wire, Pin, InputPin, OutputPin, ....` Faute de moyens structurants appropriés, on est obligé de fusionner les multiples parties fonctionnelles de ces objets en une seule description non structurée. Dans le cas de notre exemple, ce regroupement des caractéristiques produit des classes qui sont schématisées à la figure 5.1.

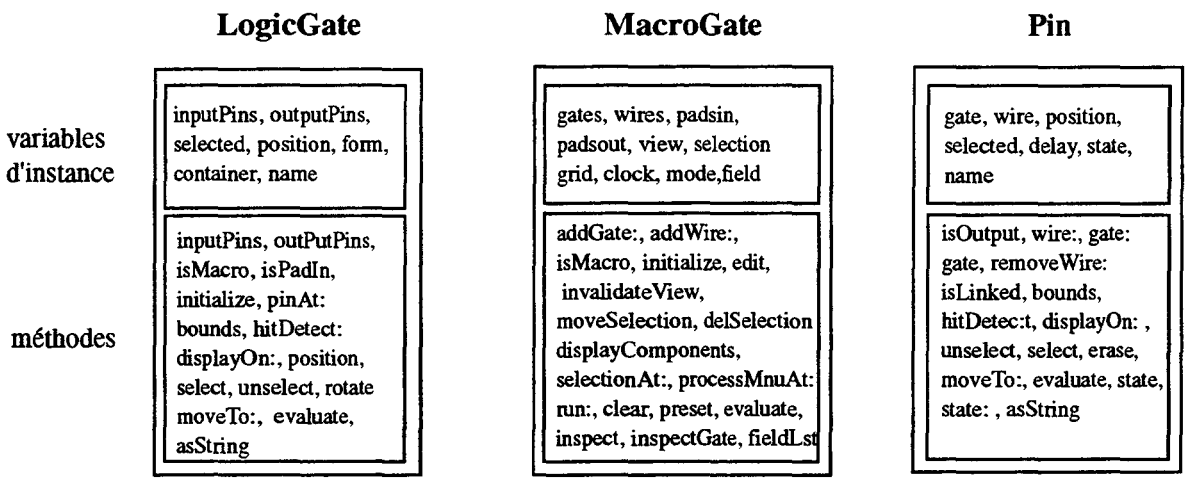


figure 5.1 : Quelques classes de l'exemple en Smalltalk

Comme nous l'avons analysé dans la section 2.3.2.1 du premier chapitre, la figure montre la difficulté de discerner précisément pour chaque classe le sous-ensemble des variables d'instance et des méthodes qui sont spécifiques à un contexte fonctionnel particulier. Des moyens de structurations supplémentaires apparaissent nécessaires pour maîtriser la complexité interne de tels objets et le caractère transversal des fonctions du système.

En Smalltalk, les classes sont systématiquement organisées de façon hiérarchique par la relation d'héritage. Une telle organisation met en avant la structuration en classes/sous-classes mais ne permet pas d'expliciter les contextes d'intervention des objets. Le fait que la relation d'héritage prime pour structurer les classes a en effet deux conséquences vis à vis des contextes. Premièrement, cela conduit à mélanger au sein de la même hiérarchie l'ensemble des classes issues de tous les contextes, supprimant ainsi la distinction entre ces derniers. Deuxièmement, des classes relatives à un même contexte peuvent, en raison de leur lien d'héritage, se retrouver à des endroits éloignés dans la hiérarchie, brisant ainsi leur relativité. La figure 5.2 illustre ces problèmes avec une partie de la hiérarchie des classes de notre exemple et deux des contextes

qui sont ici mis en évidence.

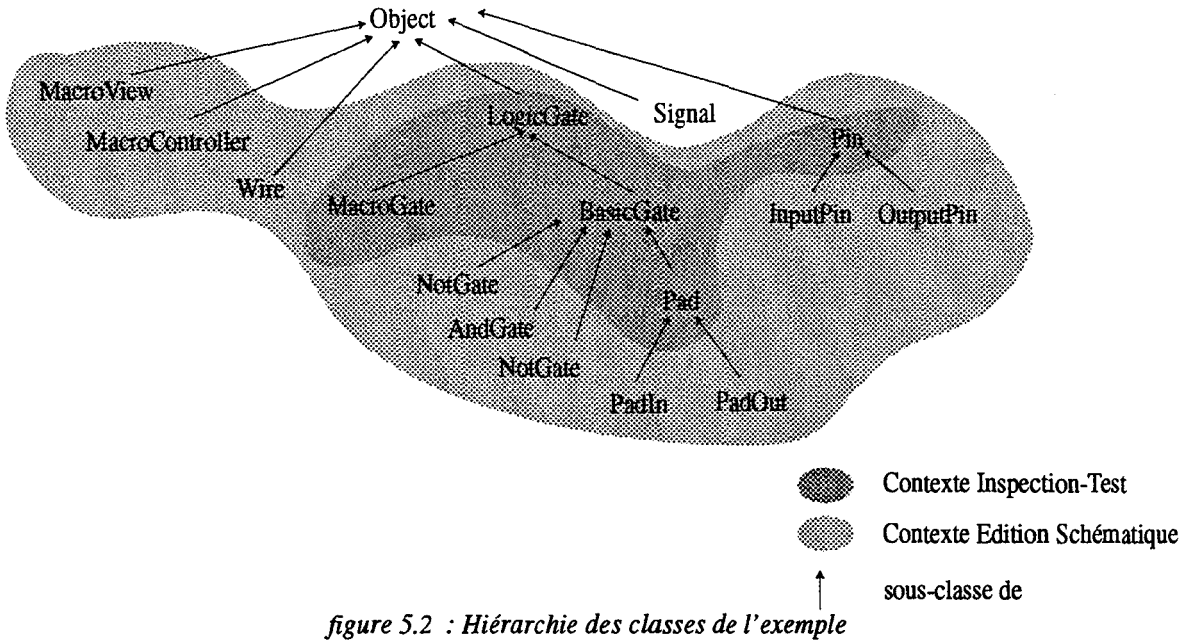


figure 5.2 : Hiérarchie des classes de l'exemple

Il faut noter qu'à partir d'une telle hiérarchie de classes, retrouver les contextes et les objets y intervenant reste toujours possible. Toutefois, cela requiert un examen approfondi du code ce qui oblige à considérer la plupart des objets dans leur intégralité et donc à considérer des aspects non pertinents.

L'analyse précédente a montré que l'explicitation des contextes s'avère difficile en se limitant à la description monolithique des classes Smalltalk. Ceci nous amène à étudier les moyens supplémentaires fournis par l'environnement pour augmenter la structuration de ces classes.

5.2.2 Les catégories comme point de départ pour expliciter les contextes

Les catégories se présentent comme un outil général de structuration des classes situé au niveau de l'environnement. Elles ne font pas partie du langage et n'ont aucune conséquence opérationnelle sur celui-ci. Dans les environnements Smalltalk incluant cette technique, on trouve généralement deux types de catégories, gérées de façon indépendante et situés à des niveaux de granularité différents: les catégories de méthodes et les catégories de classes. Nous présentons et examinons ces catégories pour obtenir la structuration selon les contextes.

5.2.3 Catégories de méthodes

Les catégories de méthodes apportent des capacités de structuration au sein de la classe. Elles ont pour objectif de partitionner les méthodes de la classe suivant des relations logiques. Une classe possède en général plusieurs de ces catégories. Les catégories d'une classe sont traitées indépendamment de celles des autres classes. Pour prendre un exemple de l'environnement, l'examen de la classe Date révèle les catégories de méthodes `arithmetic`, `comparing` et `printing` correspondant à ses différentes fonctionnalités.

Ce début de structuration offert par ces catégories peut être utilisé afin de structurer les méthodes relativement aux contextes. Appliqué aux classes considérés précédemment, cela leur

donne une structuration interne représentée à la figure 5.3.

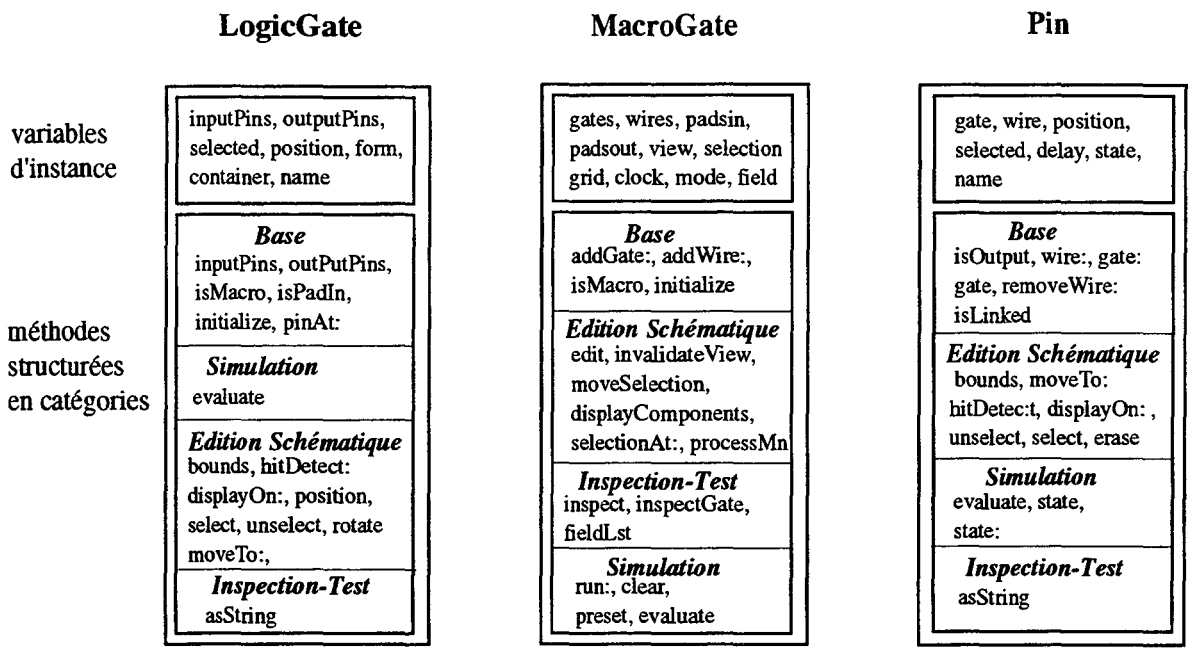


figure 5.3 Structuration interne des classes par les catégories de méthodes

La structuration des classes ainsi obtenue constitue une amélioration comparée à la situation précédente mais elle met aussi en évidence deux problèmes.

Le premier concerne le problème de structuration par contextes des variables d'instance qui n'est pas réglé. En l'absence d'une telle structuration, le programmeur n'a aucune aide pour connaître facilement les variables d'instances qu'il peut utiliser dans les méthodes d'un même contexte. Le problème se pose également en sens inverse pour déterminer le contexte d'une variable d'instance. Obtenir cette information nécessite d'inspecter le code des méthodes. De façon évidente, le traitement de ce problème réclame pour les variables d'instances des capacités de structuration analogues aux catégories de méthodes.

Le second problème concerne l'application de cette structuration pour plusieurs classes ce qui est illustré par la figure 5.3. Cette figure fait apparaître la structuration de plusieurs classes par une même catégorie. Ainsi, les classes `LogicGate`, `MacroGate` et `Pin` possèdent chacune une catégorie de méthodes qui réflète leur participation au même contexte simulation. Sous l'angle des contextes, nous sommes donc plus intéressés par cette structuration transversale que la structuration en catégories d'une seule classe. De tels besoins de structuration entraînent cependant une complexité importante de l'utilisation des catégories qui doit se faire relativement à plusieurs classes. Gérer transversalement ces catégories de façon cohérente requiert donc une systématisation.

5.2.4 Catégories de classes

Les catégories de classes procurent un niveau d'organisation qui vient se superposer à la hiérarchie d'héritage, permettant le regroupement des classes suivant des relations logiques diverses. Le programmeur peut ainsi définir plusieurs catégories de classes et déclarer explicitement pour chaque classe de la hiérarchie la catégorie à laquelle elle appartient. De cette façon, il est possible de réunir au sein de la même catégorie, des classes provenant de branches in-

dépendantes de la hiérarchie d'héritage. Signalons que cette facilité de regroupement est exploitée pour partitionner les classes outils du système Smalltalk lui-même selon leurs axes fonctionnels. Ainsi, à titre d'exemple, on peut citer la catégorie `Collection-Unordered` et la catégorie `Graphics-Geometry` de l'environnement qui regroupe respectivement les classes d'objets correspondant aux collections non ordonnées (`Bag`, `Set`, `Dictionary`, ...) et aux figures géométriques (`Point`, `PolyLine`, `Circle`, ...).

Leur capacité à ranger les classes orthogonalement à la hiérarchie d'héritage suggère d'exploiter ces catégories pour rendre compte des contextes. L'idée consiste alors à associer un contexte à une catégorie de classe. Cependant, pour les classes d'objets intervenant dans plusieurs contextes, l'application de ce principe pose un problème. En effet, cela requiert qu'une classe puisse appartenir simultanément à plusieurs catégories. Or, à notre connaissance, aucun environnement Smalltalk gérant cette technique n'offre une telle possibilité.

Il est important de préciser que la seule utilisation des catégories de classes permet d'expliciter les contextes mais ne résoud pas le problème de la structuration interne des classes. Cette observation va nous conduire par la suite à rapprocher les catégories de classes et catégories de méthodes.

Au cours de cette section, nous avons étudié la technique des catégories dans l'optique d'une structuration par contextes des objets Smalltalk. De ce point de vue, nous avons constaté que cette technique laisse entrevoir des possibilités intéressantes. Les problèmes relevés plus haut montrent néanmoins qu'il est nécessaire de systématiser les catégories. C'est ce que nous allons faire dans la prochaine section en proposant de les étendre et de les exploiter pour la structuration en contextes.

5.3 Des catégories aux contextes

L'exploitation des catégories pour réaliser les contextes passe par les extensions suivantes. Dans une premier temps, il s'agit d'introduire un nouveau type de catégorie pour les variables d'instance, couplé avec les catégories de méthodes. La structuration interne ainsi obtenue est, dans un deuxième temps, corrélée avec les catégories de classes transversales au système.

5.3.1 Catégories internes et parties fonctionnelles

Au sein d'une classe, le but est d'obtenir la structuration de ces multiples parties fonctionnelles. Rappelons qu'une partie fonctionnelle est un sous-ensemble de caractéristiques, variables d'instance et méthodes dans la terminologie Smalltalk.

Nous avons vu précédemment que les catégories de méthodes apportent un début de structuration. Cependant, pour obtenir les parties fonctionnelles, des capacités de structuration analogues sont également requises pour les variables d'instance. L'environnement Smalltalk ne disposant pas à la base de telles capacités, nous avons été amenés dans un premier temps, à introduire des catégories pour les variables d'instance. Ce type de catégorie rend possible le partitionnement de ces dernières et facilite également leur manipulation (ajout, retrait) par catégorie à la différence de Smalltalk où les variables sont introduites de façon monolithique via le message `subclass:instanceVariableNames:...`

Munis de ce type de catégorie supplémentaire, nous obtenons la structuration d'une classe représentée par le schéma (a) de la figure 5.4.

L'étape suivante consiste à associer catégories de variables d'instance et catégories de méthodes relativement à une même partie fonctionnelle. Ce couplage conduit à une structuration de la classe illustrée au schéma (b). De cette façon, les variables et les méthodes d'un même contexte sont directement associées ce qui supprime les problèmes évoqués en 5.2.2.

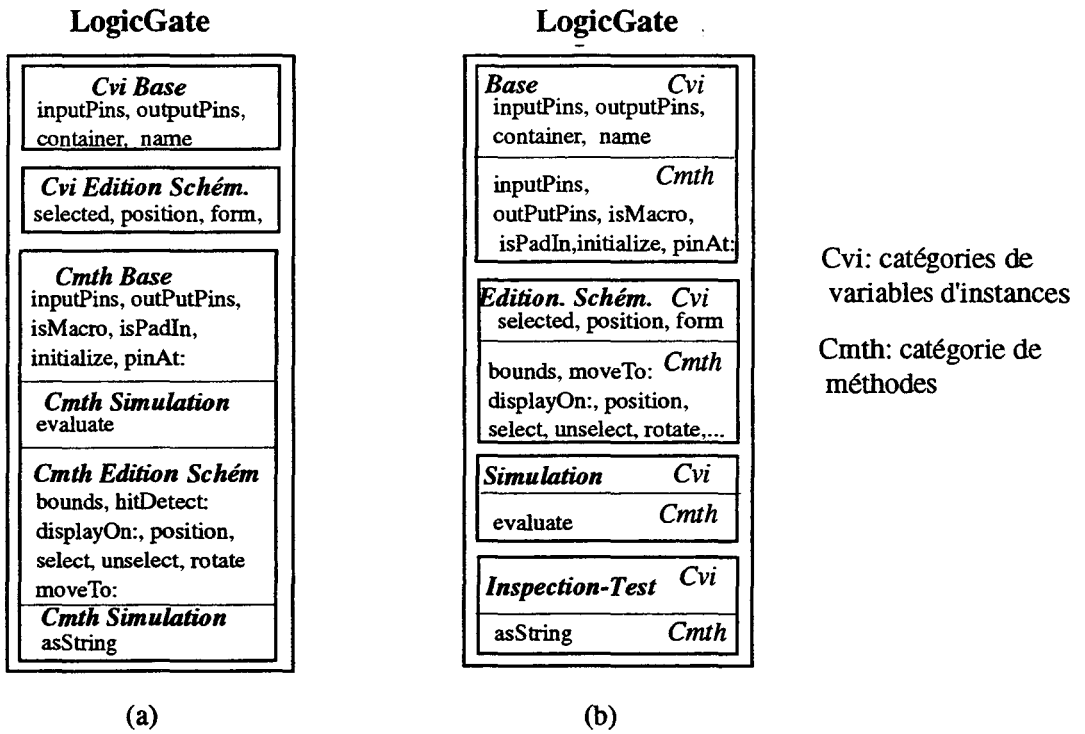


figure 5.4 Structuration d'une classe en catégories devariables d'instance/méthodes

A ce stade, nous savons rendre compte des multiples parties fonctionnelles au sein d'une classe. Il s'agit maintenant de systématiser transversalement cette structuration interne des classes par les contextes, en exploitant la notion de catégorie de classes.

5.3.2 Catégories de classes et plans fonctionnels

Le plan de base est pris en compte en l'associant à une catégorie de classe appelée "Base". Toute classe appartenant à cette catégorie est considérée selon le mode de conception de CROME. Précisons que l'héritage entre classes déterminé par le plan de base se traduit dans le cas présent par un héritage entre classes Smalltalk.

Pour obtenir les plans fonctionnels transversaux du système, l'idée consiste à leur associer systématiquement une catégorie de classes, comme suggéré en 3.2.2. La déclaration d'appartenance d'une classe à une catégorie est alors interprétée comme sa participation au contexte associé, ce qui induit une partie fonctionnelle pour celle-ci. De cette façon, une catégorie détient les descriptions des classes relatives au contexte correspondant. La contextualisation multiple de ces classes requiert cependant le relâchement de la contrainte de mono-appartenance aux catégories.

La structuration résultante pour les classes `Logicate` et `Pin` de notre exemple est donnée par la figure 5.5. Cette figure explicite le couplage entre la structuration interne en catégories de la classe et son appartenance aux différentes catégories de classes associées: une classe a autant

de parties fonctionnelles que de catégories à laquelle elle appartient. Elle montre par ailleurs que des classes appartenant à la même catégorie de classes possèdent une partie fonctionnelle relative.

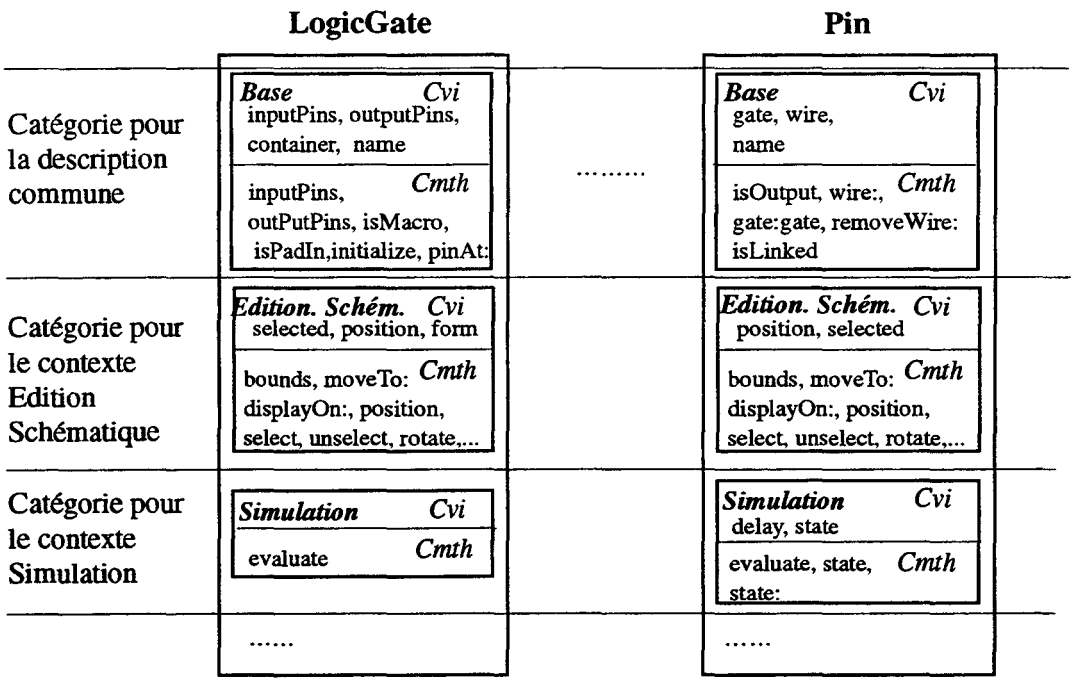


figure 5.5 Structuration des classes Smalltalk selon CROME

On peut constater d'après cette figure que ces classes possèdent une description relative à la catégorie "Base". Celle-ci correspond à la description commune à l'ensemble des contextes. De ce fait, elle est systématiquement ajoutée aux classes Smalltalk lors de leur création. Notons qu'ainsi, la visibilité plan de base/plan fonctionnel se ramène à de l'accès entre caractéristiques de la même classe.

5.3.3 Héritage et structuration interne des classes

La systématisation de la structuration interne exposée précédemment s'applique aussi dans le cas de classes en relation d'héritage. La déclaration d'appartenance d'une classe et d'une sous-classe à une même catégorie entraine pour celle-ci une partie fonctionnelle relative au même contexte. Cette combinaison de l'héritage avec la systématisation est représentée à la figure 5.6 pour les classes `Gate`, `AndGate` et `MacroGate` de notre exemple.

La structuration interne des classes ainsi obtenue n'a aucune conséquence sur leurs relations d'héritage[1]. La hiérarchie des classes déterminée dans la catégorie de base est de cette manière préservée conformément à l'approche CROME.

1. et sur leurs relations de composition également.

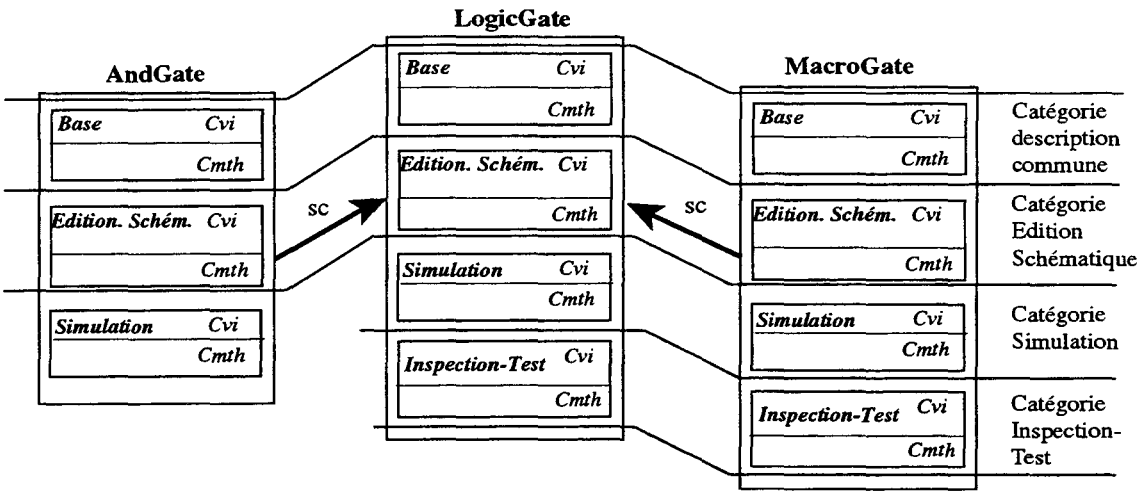


figure 5.6 : Héritage et structuration interne des classes

Dans le cas présent, cette structuration interne des classes par les catégories sert également pour rendre compte de l'héritage local à un contexte. Ainsi une redéfinition dans une partie fonctionnelle d'une classe est autorisée à condition que la méthode soit définie dans une partie fonctionnelle d'une sur-classe relative au même contexte. Ce traitement est requis pour éviter qu'une partie fonctionnelle d'une sous-classe masque accidentellement par héritage la définition d'une méthode issue d'une partie fonctionnelle liée à un autre contexte. Il n'est pas permis par exemple dans la partie fonctionnelle de la classe `MacroGate` pour le contexte d'édition de redéfinir la méthode `evaluate` définie au niveau de la classe `Gate` pour le contexte simulation.

Signalons que ce comportement est obtenu sans intervenir au niveau opérationnel. A l'ajout d'une redéfinition, il suffit simplement de vérifier que sa catégorie de méthode correspond avec celle de la définition masquée[1] détenue par une sur-classe.

Précisons pour terminer que la contextualisation implicite des classes est directement supportée par l'héritage de Smalltalk. Ainsi, sur la figure précédente, on peut remarquer que la classe `AndGate` ne possède pas de description associée au contexte d'inspection. Elle hérite néanmoins pour ce contexte de la description correspondante associée à sa surclasse `LogicGate`.

5.3.4 Classes locales aux contextes

Nous proposons ici une extension du cadre qui permet d'augmenter le pouvoir structurant des contextes pour la conception de systèmes dans leur intégralité.

Au sein d'un contexte, les objets du référentiel co-existent en général avec des objets purement locaux, en particulier les objets techniques liés à la fonction (surface d'affichage, dispositifs d'impression et de sauvegarde, ...). Dans notre exemple, les objets `MacroView` sont seulement locaux au contexte d'édition schématique.

1. En fait, il faut aussi réaliser cette vérification par rapport aux sous-classes pour éviter inversement lors de l'ajout d'une méthode à une sur-classe pour un contexte que celle-ci soit masquée par l'existence dans une sous-classe d'une méthode de même sélecteur définie pour un autre contexte.

Pour prendre en charge ces objets de manière homogène avec ceux du référentiel, nous donnons la possibilité de les décrire par des classes locales à un contexte. Cette possibilité est obtenue en déclarant leur appartenance à la catégorie de classes correspondante.

Dans cette partie, nous avons expliqué comment obtenir une conception stratifiée des classes Smalltalk guidée par la structuration en contextes du système. Dans la section suivante, nous présentons un outil qui supporte cette organisation et présente celle-ci de façon appropriée au programmeur.

5.4 Programmation par contextes en Smalltalk

Nous illustrons la programmation par contexte des classes de façon pragmatique en donnant un aperçu de la démarche facilitée par le flâneur orienté contexte.

5.4.1 Description par contextes des objets: présentation par le flâneur spécialisé

En Smalltalk, le flâneur est l'outil le plus commode pour explorer et éditer l'espace des classes. Dans le but d'offrir des commodités similaires guidées par les contextes, nous avons élaboré un flâneur dédié. Les principales caractéristiques de ce flâneur sont à la fois de rendre explicites les contextes et de faciliter l'édition et l'examen des classes de façon systématique par contexte[1]. Avec ce flâneur, la manière de programmer au sein d'un contexte n'est pas modifiée et reste de la programmation Smalltalk classique.

La figure 5.7 donne un aperçu de ce flâneur appliqué à notre exemple. Celle-ci montre la partie fonctionnelle de la classe `Gate` dans le contexte d'édition schématique. La fenêtre du flâneur se divise en quatre sous-vues. La sous-vue en haut à gauche contient la liste des contextes. La sous-vue centrale contient la liste des classes concernées par le contexte sélectionné. Suivant la valeur du bouton d'échange "common/local", cette liste correspond soit aux parties fonctionnelles, soit aux classes locales du contexte. La sous-vue en haut à droite fournit la liste des méthodes associées. Enfin, la dernière sous-vue sert à éditer le code des classes. Initialement, celle-ci contient les patrons destinés aux classes. Le choix d'une méthode permet l'édition de son code dans cette sous-vue.

1. Ce flâneur rend aussi l'organisation en catégories totalement transparente pour le programmeur en la gérant de manière automatique.

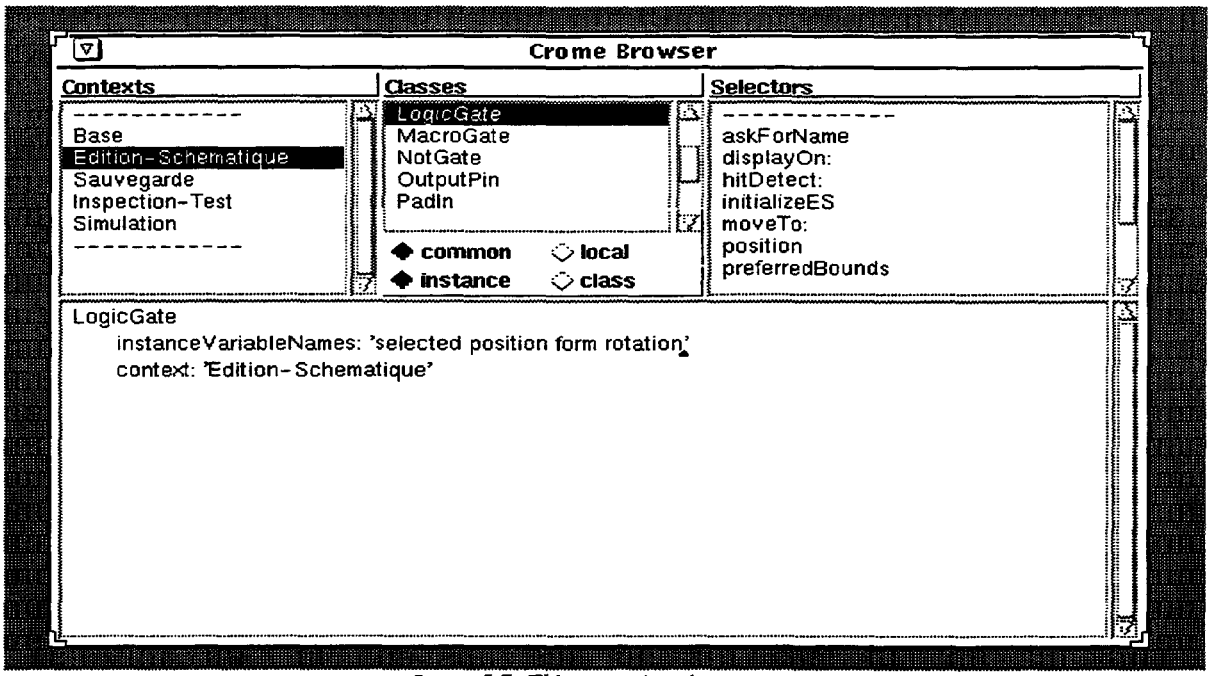


figure 5.7 Flâneur orienté contexte

Définir les classes d'objets multi-contextes

A la première ouverture du flâneur, la sous-vue des contextes ne contient qu'un seul élément libellé "Base" qui joue un rôle particulier. Celui-ci sert en effet à introduire les classes d'objets multi-contextes ainsi que leur description commune à tous les contextes. Dans le cas de notre exemple, la sélection de cet élément provoque l'apparition des classes `LogicGate, AndGate, NotGate, OrGate, MacroGate, Pad, PadIn, PadOut, Wire, Pin, InputPin, Output-Pin, ...` dans la sous-vue des classes. Pour créer une classe d'objets multi-contextes, on procède de manière classique par édition de son entête (message `subclass:instanceVariableNames:...`) dans laquelle sont précisées la surclasse ainsi que les variables d'instances communes. Une fois la classe créée, l'ajout de ses méthodes communes se réalise comme dans le flâneur standard.

Nous savons à présent comment introduire des classes d'objets multi-contextes avec le flâneur, considérons maintenant l'ajout de contextes à l'aide de celui-ci.

Création d'un contexte

La création d'un contexte nécessite de le nommer pour permettre son identification. Cette opération donne lieu à l'ajout du contexte dans la sous-vue des contextes et fait aussi apparaitre une liste vide dans la sous-vue des classes indiquant l'absence de classes pour l'instant dans le contexte. A ce stade deux possibilités sont offertes à l'utilisateur pour compléter ce dernier. On peut soit fournir une partie fonctionnelle pour une classe d'objets multi-contextes existantes, soit introduire des classes locales. Concernant l'ajout d'une classe locale au contexte, cela se fait classiquement à partir de son entête. L'ajout d'une partie fonctionnelle au contexte s'obtient en éditant un patron approprié (message `instanceVariableNames:context:` envoyé à la classe)[1]. Un exemple d'un tel patron est présenté à la figure précédente pour la classe `LogicGate` dans le plan `Edition-Schematique`. Comme indiqué par l'exemple, ce patron sert également à déterminer les variables d'instances de la classe spécifiques au contexte. Pour ce qui est de l'ajout à celle-ci de méthodes relatives au contexte, il faut au préalable avoir validé son patron correspondant. L'ajout de méthodes reste analogue au flâneur standard, ces dernières étant sys-

tématiquement rangées dans la partie fonctionnelle correspondant au contexte sélectionné[1].

Examen d'un contexte

Pour visualiser tous les éléments contenus dans un contexte, il suffit de le sélectionner dans la sous-vue correspondante[2]. Ceci donne alors accès aux parties fonctionnelles ainsi qu'aux classes locales. Ainsi, sur la figure précédente, on peut observer que les classes `AndGate`, `OutputPin`, `PadIn`, ... ont une partie fonctionnelle pour le contexte Edition-Schematique. La sélection de l'une de ces classes donne le contenu de sa partie fonctionnelle dans les sous-vues correspondantes: patron associé (et donc ses variables d'instances) dans la vue d'édition ainsi que ses méthodes dans la sous-vue appropriée. On peut ainsi, en procédant par sélection successive des classes, naviguer dans le contexte transversalement à ces dernières. De façon plus générale, un contexte étant sélectionné, le flâneur présente toujours la description des objets seulement pertinentes pour celui-ci, supprimant tous les détails liés aux autres contextes.

La figure 5.8 résume les possibilités de navigation offertes par le flâneur qui s'accomplit suivant une double dimension classe/contexte.

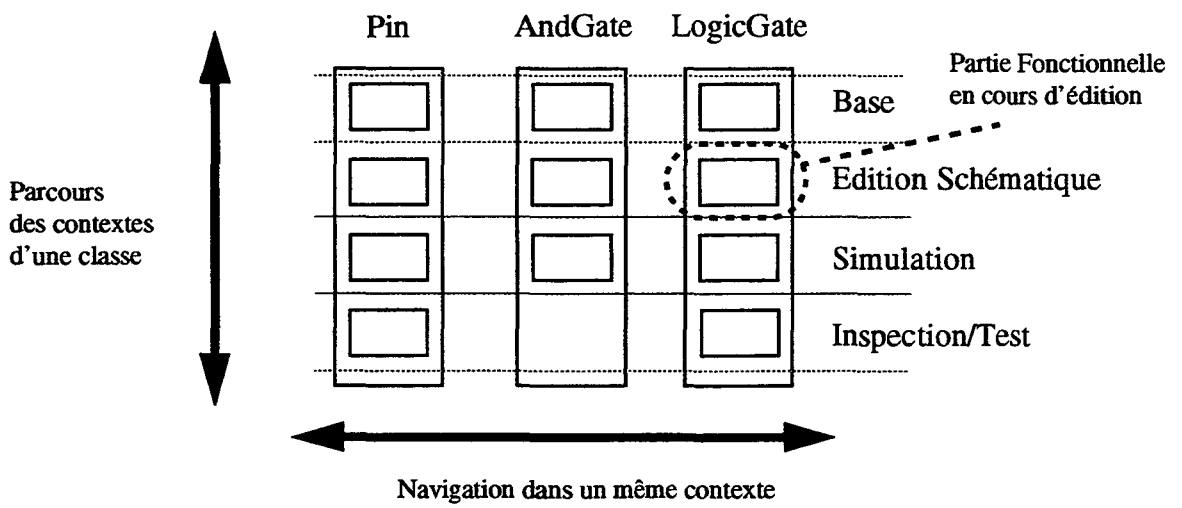


figure 5.8 : Possibilités de navigation offertes par le flâneur

Comme le montre cette présentation succinte, l'examen et la manipulation des classes par ce flâneur orienté contexte sont différentes de celles accomplies avec le flâneur standard. Ce flâneur encourage une description stratifiée par contextes des classes au lieu d'une description des classes prises une à une dans leur intégralité comme c'est le cas classiquement. On obtient ainsi une description des classes proche d'une conception dans laquelle les contextes sont considérés isolemment.

---

1. Ce patron est déterminé en fonction du contexte sélectionné. Si le contexte choisi est par exemple celui de simulation alors le patron pour définir une partie fonctionnelle est: `NameOfClass instanceVariableNames:'instVarName1 instVarName2' context: 'Simulation.` L'utilisateur doit alors éditer le nom de la classe concernée et éventuellement les variables d'instances associé à la partie fonctionnelle.
1. Il est tout à fait possible de créer textuellement les parties fonctionnelles de chaque classe en ayant recours aux méthodes `instanceVariableNames:context:` et compile:classified: offertes par celle-ci.
2. Avec ce flâneur, il faut nécessairement sélectionner un contexte pour accéder à une classe. De ce fait, on examine toujours une classe dans un contexte particulier.

5.4.2 Fonctionnalités du flâneur

Dans son ensemble, le flâneur implémente la plupart des fonctionnalités liées aux différentes activités de conception. Ces fonctionnalités sont similaires à celles rencontrées dans le flâneur standard mais sont adaptées à l'examen et la manipulation des contextes. Elles se répartissent en quatre sous-ensembles:

- Édition : Ce sous-ensemble concerne tout ce qui a trait à la définition et l'organisation des contextes. Il comprend des opérations de création/modification/suppression des contextes et des descriptions des objets qu'ils contiennent. C'est au niveau de ces opérations que sont effectués des contrôles destinés à assurer l'intégrité des contextes. Précisons par ailleurs que la suppression d'un contexte pour les classes d'objets multi-contextes entraine uniquement le retrait des caractéristiques correspondantes au sein de celles-ci, préservant ainsi leurs descriptions dans les autres contextes.

- Navigation : Ce sous-ensemble donne le moyen d'explorer les descriptions contextualisées selon deux modes. Un premier mode orienté objet permet d'examiner les descriptions contextualisées d'une même classe (par sélection d'une classe puis sélection successive des contextes). Le second mode orienté contexte, dual du premier, permet d'explorer les descriptions qui sont relatives à un même contexte (par sélection d'un contexte puis sélection successive des classes).

- Recherche : Ce sous-ensemble sert à retrouver différents types d'informations sur les objets décrits par contexte. Par exemple, il est possible d'extraire la liste des parties fonctionnelles qui implantent un message particulier dans un contexte ou encore de connaitre les contextes dans lesquels une classe possède une partie fonctionnelle associée.

- Stockage : Ce sous-ensemble offre la possibilité de stocker les descriptions des objets sous la forme de fichier source en format standard et réinterprétable. Deux formes de stockage sont proposées. On peut stocker soit un contexte dans son intégralité, soit une classe dans tous ses contextes. A titre d'exemple, voici une portion du code produite pour le stockage du contexte édition schématique.

```
Gate
   instanceVariableNames: 'selected position form'
   context: 'Edition-Schematique'!

!Gate methodsContext: 'Edition-Schematique'!
displayOn: aGraphicContext...!
hitDetect: aPoint
   | pin |
   pin := self allPins select:[:p | p hitDetect: aPoint]
   pin isNil ifFalse:[^self bounds containsPoint: aPoint]
   ^false!
select...!
moveTo: aPoint...! !

MacroGate
   instanceVariableNames: 'view selection grid'
   context: 'Edition-Schematique'!

!MacroGate methodsContext: 'Edition-Schematique'!
edit
   | wndw |
   wndw := ScheduledWindow label: self name.
   view := self buildMacroView. wndw component: view.
   wndw open...!
displayComponentsOn: aGraphicContext...!
moveSelection ...!!
selectionAt: aPoint..! !

Pin
   instanceVariableNames: 'selected position form'
   context: 'Edition-Schematique'!

!Pin methodsContext: 'Edition-Schematique'!
displayOn: aGraphicContext...!
hitDetect: aPoint
   ^self bounds containsPoint: aPoint !
moveTo: aPoint...!
select
   selected ifFalse:[ selected := true]...! !

View subclass: #MacroView
   instanceVariableNames:''
   classVariableNames:''
   poolDictionary:''
   context:'Edition-Schematique'! !
!MacroView methodsContext: 'Edition-Schematique'!
```

La lecture de ce code fait apparaître la définition des parties fonctionnelles de quelques classes participant au contexte d'édition schématique ainsi que celle d'une classes locale (`MacroView`). Lors du rechargement d'un tel contexte, les parties fonctionnelles sont réinterprétées et rangées directement dans les classes concernées tout en préservant la structure interne des classes liées aux autres contextes.

Certaines conditions doivent être respectées pour qu'un contexte puisse être rechargé sans complications. Les parties fonctionnelles étant destinées à un ensemble de classes, il faut que celles-ci existent au préalable dans l'environnement d'une part. Signalons que les classes d'objets multi-contextes et leur description commune peuvent également être stockées dans un fichier particulier au même titre que les contextes. Il faut d'autre part que les méthodes d'autres

contextes référencées dans une partie fonctionnelle soient déjà présentes dans la classe associée. Le rechargement des contextes doit donc se faire en respectant un ordre. Actuellement, les prérequis d'un contexte vis à vis d'autres contextes pour qu'il fonctionne correctement n'est pas rendu explicite.

En dehors de ce flâneur, il existe également un inspecteur d'objet adapté. A la différence de l'inspecteur standard, la liste des variables de l'objet est structurée selon ses contextes. La figure suivante montre l'utilisation de cet inspecteur sur plusieurs objets de notre exemple.

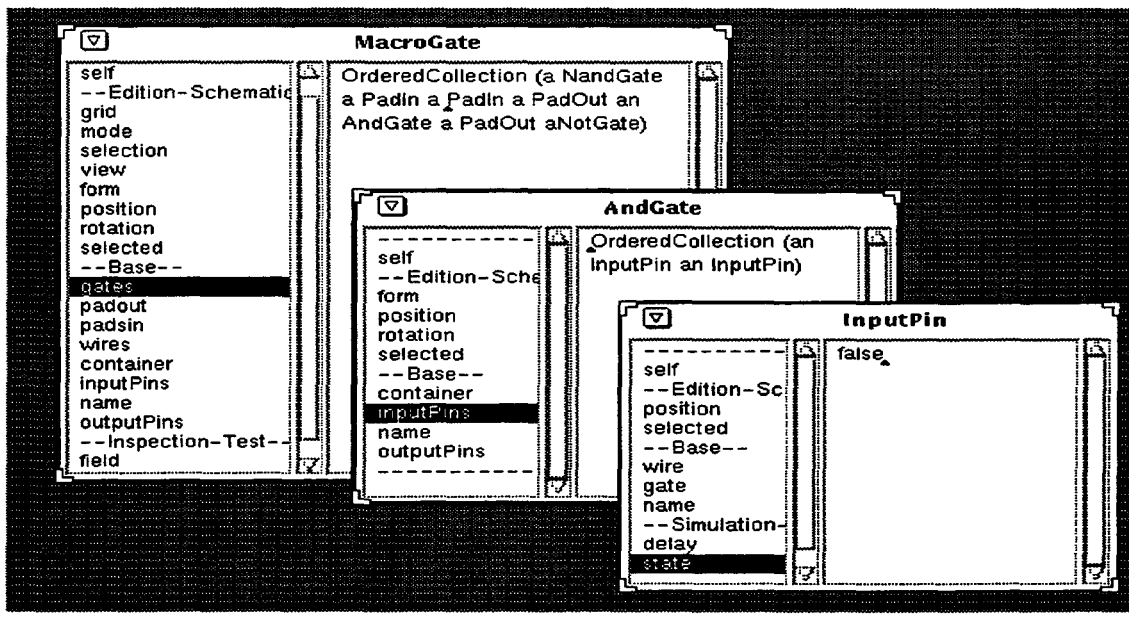


figure 5.9 : Inspecteur d'objets basé sur la structuration par contextes des classes

Il est bon de noter que, grâce au choix des catégories, les autres outils de l'environnement (flâneur standard, inspecteur, debugger, ...) restent disponibles pour les objets conçus par contextes. Il faut cependant souligner que l'utilisation conjointe du flâneur spécialisé et du flanêur standard sur les mêmes classes est permise à condition de limiter ce dernier à la consultation et non à l'édition pour ne pas corrompre les structures mises en place pour la gestion des contextes.

Une précision supplémentaire est à apporter au sujet de la coexistence des objets décrits par contextes avec les autres objets Smalltalk décrits classiquement. Les catégories ne modifiant pas la nature des objets, ces deux sortes d'objets sont de même nature et peuvent donc interagir entre eux librement. Au niveau des classes, la coexistence est également permise: une classe multi-contexte peut hériter d'une classe Smalltalk.

Signalons pour terminer que toutes les facilités offertes par Smalltalk concernant le stockage et l'évolution des instances suite à un changement de structure de leur classe restent applicables à ces objets décrits par contextes. Cette dernière facilité présente la propriété intéressante d'autoriser une contextualisation des classes même si celles-ci possèdent déjà des instances.

5.4.3 Expérimentations avec l'environnement

La démarche de conception induite par ce flâneur a été expérimentée pour programmer en Smalltalk trois études de cas présentées. Chacune de ces études de cas a permis de mettre à l'épreuve le flâneur pour des situations de conception différentes.

5.4.3.1 Edition structurée de Document

Un système dédié à l'élaboration de documents structurés a été développé [Vanwormhoudt 97]. Ce système est structuré en quatre contextes autour du référentiel présenté en figure 5.10. Pour simplifier, nous avons pris des documents composés de parties textes et graphiques. Une copie d'écran de l'application est également donnée à la figure 5.11.

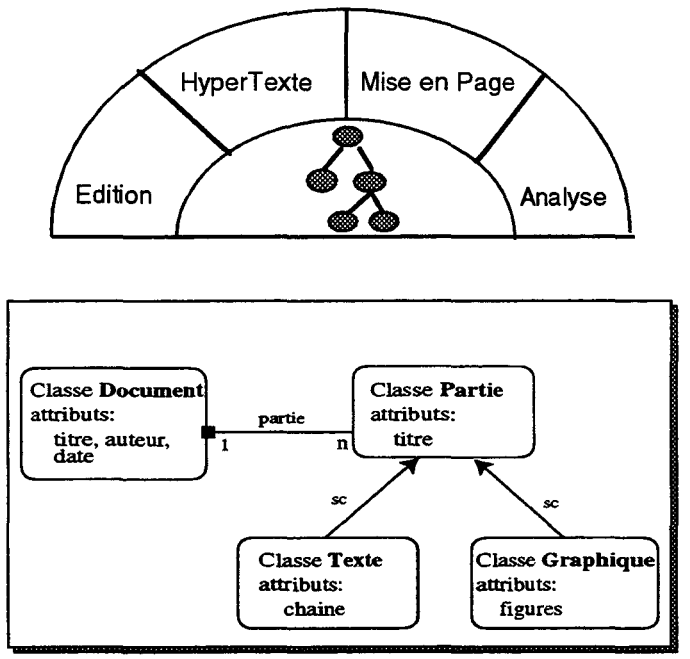


figure 5.10 : Contextes et référentiel de l'application d'édition structurée

Cet exemple a permis de tester l'intérêt de la démarche et l'efficacité du flâneur dans le cas où le nombre de contextes à développer est important. Il en ressort d'une part que l'environnement permet de maîtriser la complexité d'un tel système et d'autre part que celui procure une grande flexibilité pour l'ajout de nouveaux contextes. Dans la démarche, deux contextes ont été en effet apposés a postériori: génération d'impression "postscript" et le stockage de document.

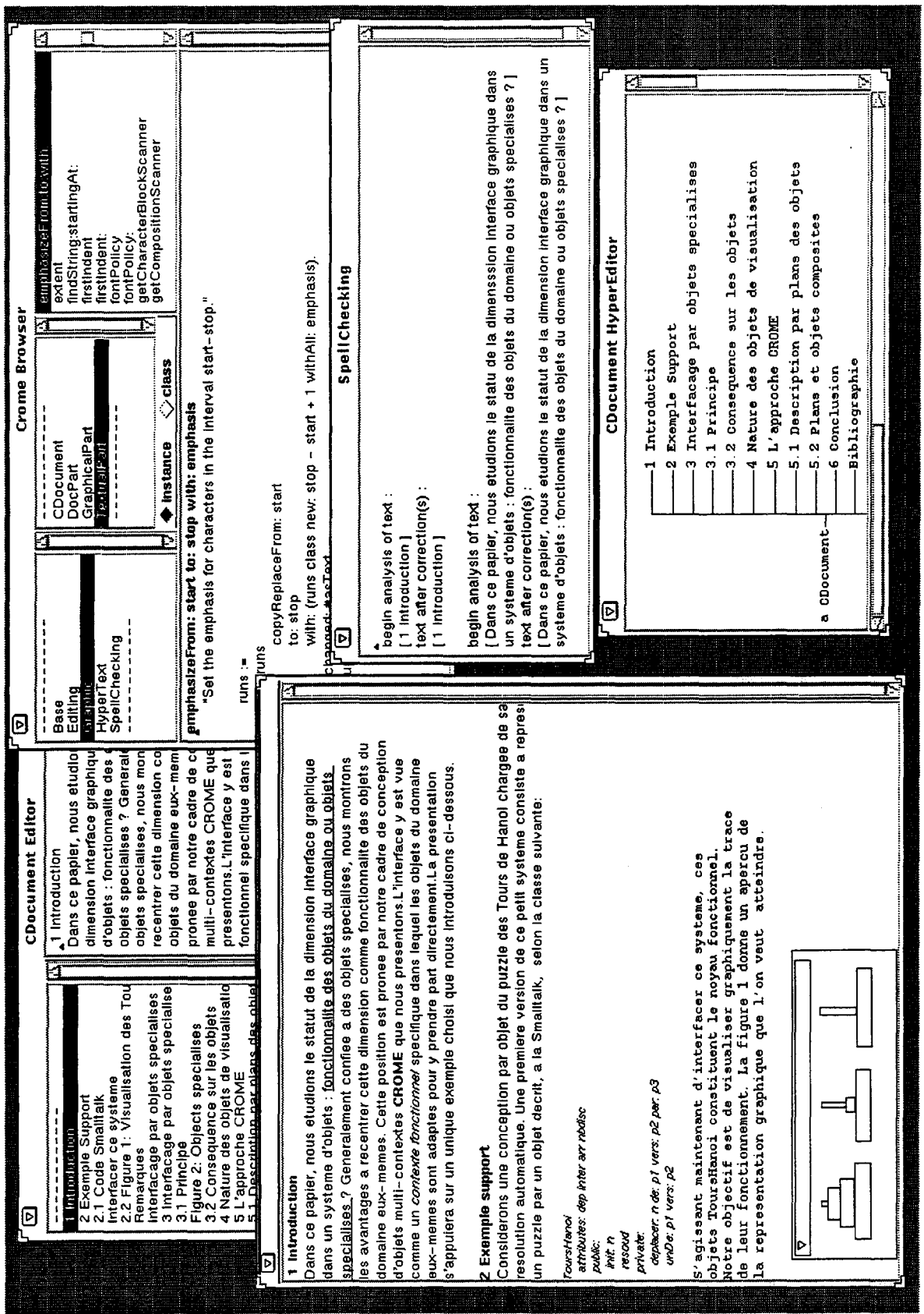


figure 5.11 : Copie d'écran de l'application

5.4.3.2 Organisation d'une conférence [Kristensen 93]

L'exemple de l'organisation d'une conférence [Kristensen 93] utilisé pour introduire la technique des "transverse activities" a aussi été développé dans notre environnement [Vanwormhoudt 98]. La figure 5.12 présente le référentiel d'objets.

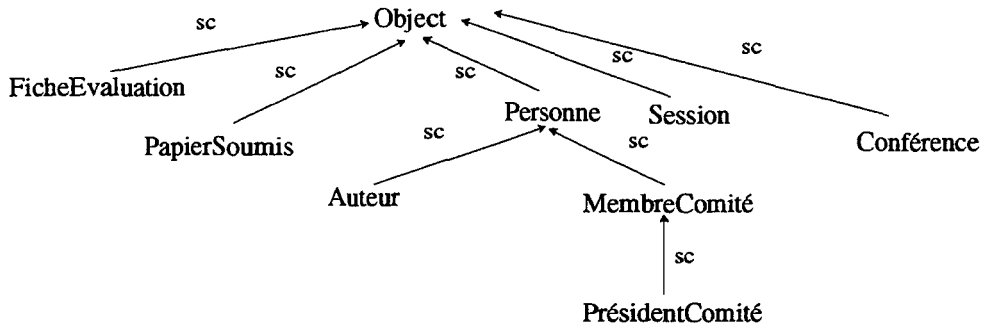
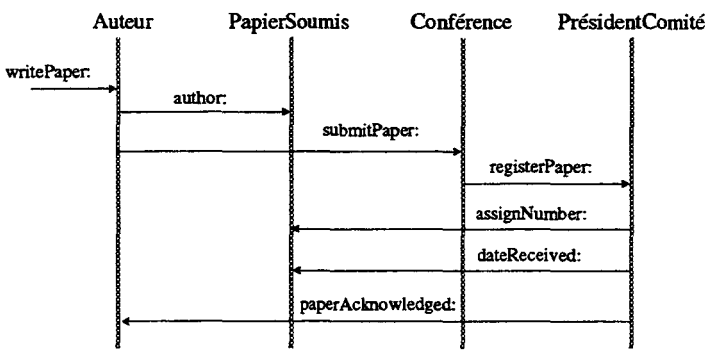


figure 5.12 : Les classes de l'exemple d'organisation d'une conférence
figure 5.13

Cet exemple se caractérise par des schémas complexes de communication inter-objets dans chaque contexte: Préparation de l'appel à communications, Soumission des papiers, Evaluation des papiers, Elaboration du programme. La figure 5.14 présente deux de ces schémas correspondant au contexte de soumission des papiers et à celui de leur évaluation.

Intéractions inter-objets pour la soumission des papiers



Intéractions inter-objets pour l'évaluation

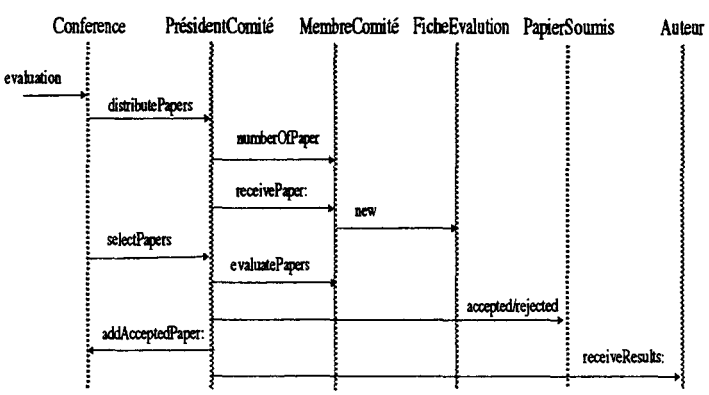


figure 5.14 : Intéractions inter-objet dans deux contextes

La description de cet exemple nécessite par conséquent des facilités pour examiner et décrire les méthodes de plusieurs classes relativement au même contexte. En isolant chaque contexte et en facilitant la navigation au sein de ceux-ci, CROME fournit une aide précieuse pour la conception inter-objets au sein d'un contexte.

5.4.3.3 CAO de circuits

Le système destiné à la conception de circuits logiques hiérarchique a également fait l'objet d'une implantation en Smalltalk. Il a été développé dans le cadre d'un projet de fin d'étude de deux élèves ingénieurs [Charlet 97] possédant une première expérience de Smalltalk. Cette expérimentation a permis de valider le flâneur sur une application de taille respectable comportant des objets de types variés et fortement structurés. Elle a également montré que les contextes peuvent servir au développement décentralisé [Ossher 94] d'un même système. Dans le cas présent, cela s'est fait en donnant aux programmeurs le même ensemble de classes de base déterminé préalablement, chacun ayant pour tâche de réaliser des contextes différents. A partir de cet ensemble de classes, ils ont pu élaboré séparément leur partie du système décrite par des contextes.

5.4.3.4 Conclusion des expérimentations

La présence dans le même environnement du flâneur standard et celui dédié aux contextes est riche d'enseignements. Il nous permet de comparer leurs qualités pour la conception d'un même ensemble de classes. Cette comparaison montre en particulier que le flâneur standard oblige en permanence lors du passage d'une classe à l'autre à retrouver la partie fonctionnelle correspondant au contexte en cours d'élaboration. Avec ce même flâneur, la gestion des catégories transversalement à plusieurs classes reste entièrement manuelle ce qui réclame des efforts supplémentaires de la part du programmeur. Celui-ci doit alors itérer les opérations ((re)nommage, suppression, stockage) sur chaque classe du contexte. Toutes ces manipulations sont par ailleurs complexifiées par le fait que les classes sont regroupées dans une seul catégorie en raison de la contrainte de mono-appartenance. Face à ces difficultés, le flâneur orienté contexte devient rapidement indispensable dès lors que le nombre de classes et le nombre de contextes à concevoir augmentent.

Pour ce qui est de l'utilisation du flâneur, nous n'avons pas rencontré le besoin de revenir au flâneur standard y compris lorsqu'il s'agissait de travailler sur une classe dans sa globalité (inter-contextes). Les deux modes de navigation offerts se sont révélés à l'usage tout à fait suffisants. La possibilité d'éditer et de visualiser des classes locales au contexte avec ce même flâneur contribue également à rendre le flanêur standard superflu.

L'utilisation de ce flâneur suggère deux améliorations possibles. La première concerne l'édition des méta-classes privées à chaque classe. Ce nouveau flâneur ne sait pas les structurer fonctionnellement de façon analogue aux classes. L'expérience montre néanmoins que de tels besoins de structuration existent parfois. Actuellement, le seul moyen pour le programmeur de structurer une métaclasse relativement à des contextes est de passer par le flâneur standard et saisir manuellement les catégories de méthodes correspondantes. L'intégration de cette fonctionnalité pourra faire l'objet d'un développement ultérieur.

Une seconde limitation se situe au sein d'une partie fonctionnelle. Puisque les catégories de méthodes sont utilisées pour rendre compte des parties fonctionnelles, il n'existe pas de moyens pour une structuration supplémentaire des méthodes. Une telle structuration s'avère pourtant utile dans certains cas, par exemple, pour exprimer le caractère privé de certaines méthodes appartenant à une partie fonctionnelle. Une solution à ce problème passe par la gestion d'un niveau supplémentaire de catégorie de méthodes[1].

5.5 Eléments d'implantation

L'obtention de la structuration en contexte des classes s'est faite en apportant trois extensions à l'environnement de base. Une première extension a consisté à introduire les catégories sur les variables d'instances et à les mettre en relation avec les catégories de méthodes de façon à obtenir les parties fonctionnelles. Une seconde extension a été nécessaire afin de relâcher la contrainte de mono-appartenance des classes aux catégories. Enfin, une troisième extension nous a conduit à mettre en place le couplage entre catégories de classes et descriptions contextualisées.

Ces extensions ont été implantées dans l'environnement VisualWorks 2.0 [Parcplace 94], un descendant direct de la version originale de Smalltalk-80. Dans cet environnement, la gestion des catégories est la responsabilité de deux classes indépendantes.

La classe `ClassOrganizer` prend en charge la catégorisation des méthodes associées à une classe. Dans l'environnement, chaque classe détient une référence vers une instance de cette classe (appelée son organisation, qui lui est propre) avec laquelle elle collabore lors des mises à jour de son dictionnaire de méthodes. L'organisation d'une classe est ensuite accédée et manipulée par le browser qui se charge de sa présentation à l'utilisateur.

Concernant la catégorisation des classes, c'est la classe `SystemOrganizer` qui en a la responsabilité. Par défaut, l'environnement ne gère qu'une seule instance de cette classe rangée dans la variable globale `SystemOrganisation`. Cette instance contient l'ensemble des catégories de classes existantes dans l'environnement. Elle est modifiée par la classe `ClassBuilder` qui traite les changements structurels d'une classe et en particulier le changement de catégorie.

L'implantation des extensions est obtenue par spécialisation de ces deux classes, ce qui est schématisé à la figure 5.15.

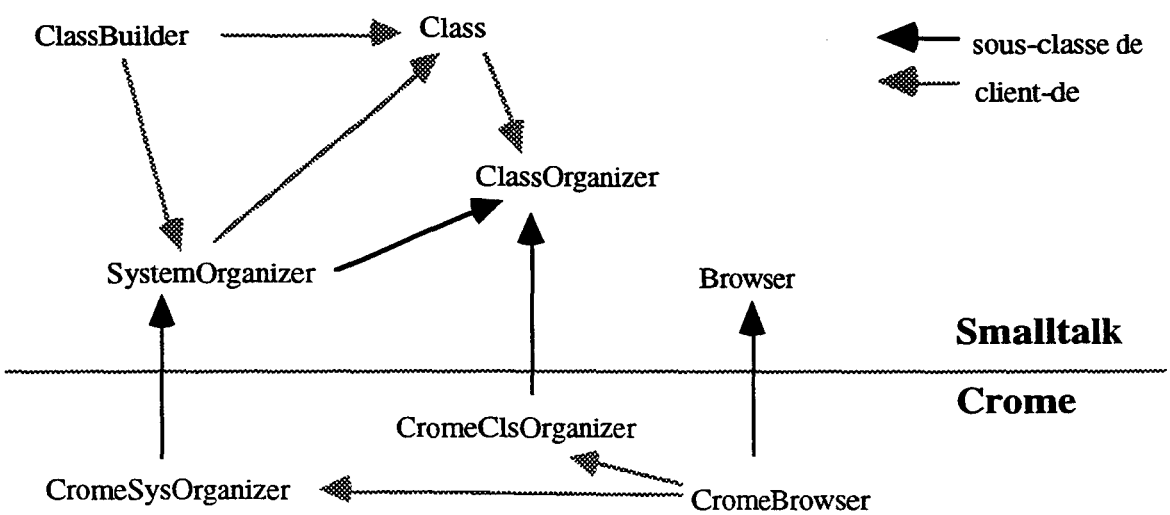


figure 5.15 Extension des classes de l'environnement pour les contextes

Une première classe `CromeClsOrganizer` est introduite pour ajouter la catégorisation des variables d'instance et coupler ces catégories avec celles des méthodes. Le couplage est réalisé

1. en utilisant par exemple un même préfixe pour les sous-catégories d'une même partie fonctionnelle.

en donnant un nom identique aux deux catégories d'une même partie fonctionnelle. Les instances de cette classe sont destinées à servir d'organisation pour les classes CROME. Une particularité de cette classe `CromeClsOrganizer` est de préserver la sémantique de sa surclasse `ClassOrganizer` ce qui rend possible l'examen des classes d'objets multi-contextes par le flaneur standard.

La seconde classe `CromeSysOrganizer` gère un ensemble de contextes selon les principes décrits auparavant. Elle lève la contrainte de mono-appartenance des classes aux catégories et met en place le couplage des catégories de classes et de la structuration interne des classes par les catégories de variables d'instance/méthodes. Ce couplage est réalisé en prenant pour convention le même nom pour la catégorie de classes et les catégories des parties fonctionnelles associées. Notons que pour rester compatible avec les outils standards, nous faisons le choix de placer systématiquement les classes de tous les contextes dans une même catégorie de l'environnement de base que nous avons spécialement nommée `Crome`.

A elles seules, ces deux nouvelles classes ne sont toutefois pas suffisantes pour obtenir une structuration interne des classes comme indiqué précédemment. Il faut également enrichir et modifier le comportement standard des classes selon les points suivants:

- Pour pouvoir être structurée selon les contextes, une classe doit posséder une organisation instance de `CromeClsOrganizer`. Elle doit par ailleurs être gérée par l'organisateur des classes CROME, une instance de `CromeSysOrganizer`, en plus de celui de Smalltalk. Pour obtenir ce paramètrage des classes, il est nécessaire de redéfinir certaines méthodes standards telle que `subclass:instanceVariableNames:.....`

- Pour permettre l'ajout de parties fonctionnelles, une classe doit fournir des opérations spécifiques. La méthode `instanceVariableNames:context:` est introduite à cet effet. Cette méthode permet de déclarer une nouvelle partie fonctionnelle et déterminer les variables d'instances associées. En interne, cette méthode se charge de la création des catégories de variables d'instance et de méthodes associées si la partie fonctionnelle n'existe pas déjà. Elle répercute aussi les changements structurels dus à la définition des variables d'instances (ajout, suppression) et les enregistre dans leur catégorie. Pour la définition des méthodes d'une partie fonctionnelle, nous retenons et adaptons la méthode standard `compile:classified` qui compile une méthode donnée sous forme de chaîne de caractéres pour la classe en la rangeant dans la catégorie spécifiée. Ajoutons que ces deux méthodes prennent soin de vérifier que les caractéristiques (attributs et méthodes) définies pour la partie fonctionnelle n'interfèrent pas avec celles de même nom pouvant exister dans d'autres parties fonctionnelles de la classe. La définition de la caractéristique est dans ce cas rejetée en informant l'utilisateur de l'origine du problème.

- Certains traitements de base des classes comme la suppression, le renommage, le stockage de catégories, ont besoin d'être spécialisés pour tenir compte de la nouvelle organisation proposée. Il faut en particulier y intégrer le traitement des catégories de variables d'instance et leur couplage avec les catégories de méthodes.

En raison des spécificités précédentes, une classe CROME apparait comme une extension des classes standards, ce qui demande de les définir par une nouvelle métaclasse. Cependant, l'absence de métaclasses explicites en Smalltalk rend peu aisée une telle opération[1,2]. Nous dev-

1. Une discussion détaillée sur ce sujet est donnée dans [Briot 89].

ons nous résoudre à l'une des deux approches suivantes: soit introduire une nouvelle classe en rangeant la fonctionnalité dans la métaclasse associée et en forçant tout classe CROME à hériter de cette classe, soit incorporer directement la fonctionnalité dans la méta-classe du noyau . Aucune de ces deux approches n'est entièrement satisfaisante. La première approche ne permet pas d'hériter d'une classe Smalltalk quelconque. La seconde approche résoud ce problème mais vient en contre-partie "polluer" une classe du noyau. C'est néanmoins cette seconde approche qui est opérationnelle dans l'implantation actuelle.

Pour terminer, nous avons spécialisé la classe `Browser` du flâneur standard en introduisant la classe `CromeBrowser` de façon à prendre en compte toutes les extensions exposées auparavant. Ce nouveau flâneur sait en particulier retrouver les parties fonctionnelles à partir du contexte et de la classe sélectionnée en utilisant la convention de nommage évoqué précédemment. Contrairement au flâneur standard, une instance de `CromeBrowser` est couplé avec l'organisateur des classes CROME (une instance de `CromeSysOrganiser`).

5.6 Conclusion

Dans ce chapitre, nous avons montré comment il est possible d'appliquer de façon relativement simple et immédiate l'approche CROME à Smalltalk, attestant de cette manière de la généralité de ses principes. Nous avons pris comme hypothèse de départ de ne pas intervenir au niveau du langage. Ceci nous a conduit à examiner la technique des catégories située au niveau de l'environnement puis de façon originale à l'étendre et la systématiser pour une structuration en contexte des classes Smalltalk. Cette extension a ensuite été exploité pour construire un flâneur dédié aidant le programmeur à concevoir systématiquement par contextes. Le résultat est une amélioration de la notion des catégories et une augmentation des capacités de structuration de l'environnement Smalltalk.

Les idées présentées ici sont applicables à tout langage orienté objet en général. En effet, la structuration interne des classes proposée ne dépend pas de la sémantique des classes et de l'héritage entre classes. Des éléments de solution pour appliquer cette approche dans le cadre de langages compilés comme Java, C++ ou Eiffel pourraient s'inspirer de [Ossher 92] qui présente une approche syntaxique pour combiner des ensembles d'"extensions" (correspondant à un paquet de variables d'instances et de méthodes pour une classe) avec une hiérarchie de classes.

La structuration des classes obtenue par les catégories a aussi ses limites. Celle-ci n'intervenant qu'au niveau de l'environnement, elle n'a aucune répercussions sur la structure des classes au niveau du langage. En particulier, les règles de nommage et de visibilité au sein des classes n'en sont pas modifiés. Il en résulte que certains aspects du modèle CROME ne sont pas supportés par cette réalisation. Ainsi, les possibilités d'adaptation du comportement de base dans un contexte ne sont pas disponible pour les objets Smalltalk décrit selon les contextes. De même, la présence de caractéristiques de même nom (attributs et méthodes) dans des parties fonctionnelles distinctes n'est pas autorisée. Il n'est pas possible par conséquent de décrire chaque partie fonctionnelle indépendamment des autres. Enfin, l'encapsulation entre parties fonctionnelles n'est pas supportée non plus. A l'intérieur d'une partie fonctionnelle, l'accès aux variables d'instances des autres contextes demeure sans restriction. Le respect de l'encapsulation reste alors du ressort du programmeur.

---

2. Une solution à ces difficultés serait d'utiliser NéoClassTalk [Rivard 97], un environnement Smalltalk où les méta-classes sont manipulables explicitement.

6

Conclusion générale

Bilan

Dans le premier chapitre, nous avons examiné les besoins de structuration liés à la multiplicité des fonctions dans un système d'objets. On trouve en général de tels besoins dans les systèmes logiciels de conception et d'ingénierie. En étudiant l'organisation de ces systèmes, nous avons constaté l'orthogonalité entre les objets et les fonctions qui les composent. Le problème central est alors de rendre compte de ces deux axes de structuration et de leur articulation dans la description d'un système d'objets afin d'en améliorer la modularité. Plus précisément, il s'agit de structurer fonctionnellement les descriptions des objets et orthogonalement de circonscrire les fonctions transversales du système. Pour prendre en compte ce besoin de double structuration objets/fonctions, nous avons analysé l'existant sous les deux angles.

Sous l'angle des objets, nous avons rappelé les problèmes posés à l'approche objet classique qui offre peu de moyens pour rendre compte de la structuration multi-fonctionnelle des objets. Ceci nous a amené à montrer comment aborder ce problème de structuration avec les constructions de base telles que l'héritage multiple et la composition. Puis nous avons présenté des techniques spécifiques relevant principalement de l'approche multi-points de vue. L'analyse de ces techniques a permis de constater qu'elles apportent des éléments de solution pour structurer les descriptions des objets relativement aux fonctions. Cette analyse a aussi mis en avant la granularité fine de ces techniques et leurs limites pour rendre compte de la transversalité des fonctions.

Sous l'angle des fonctions, les techniques permettant d'en rendre compte existent mais sont peu nombreuses. Il s'agit pourtant d'une difficulté et d'une préoccupation importante comme l'attestent les travaux sur la représentation de "transverse activities" [Kristensen 93], la programmation par sujets [Harrison 93] ou encore la conception de systèmes par modèles de rôles [Andersen 92]. Nous avons constaté que ces approches ne font pas directement le lien entre la structuration en fonctions du système, ou du moins de sa spécification, et la description structurée des objets prenant part aux différents contextes qui s'en déduisent.

Ces constats au niveau de chaque axe de structuration nous ont conduit à proposer dans le chapitre 3 notre approche CROME qui vise à les étudier conjointement. Cette approche permet une conception en termes d'objets et de contextes fonctionnels tout en tirant profit des moyens structurants offerts par ces deux axes. Dans ce chapitre, nous avons présenté:

- la structuration par plans de la description des objets qui rend compte des deux axes de structuration et systématise leur articulation;
- la stratégie d'héritage modulaire basée sur la notion de plan qui détermine la description des objets dans un contexte;
- la communication entre objets circonscrite au contexte qui assure une programmation modulaire des multiples traitements fonctionnels assurés par les objets;
- l'application systématisée des principes précédents et leurs bénéfices pour la contextualisation multiple de graphes d'objets et la factorisation de code sur un ensemble de fonctions;

- la résolution des problèmes d'articulation entre contextes au sein même des objets, ce qui structure leur couplage

Tous ces points ont été illustrés de manière concrète au travers d'un exemple de CAO électronique.

Dans les deux derniers chapitre du mémoire, nous avons abordé les questions liées à la mise en oeuvre des propositions de CROME.

Au chapitre 4, nous avons étudié son application à ROME en tant que langage de programmation. Cette réalisation a été obtenue en prenant pour point de départ les capacités de représentation multiple par points de vue des objets offertes à la base dans ROME. Nous en avons alors proposé une systématisation à travers les notions de plan, d'héritage modulaire et de contexte. Ceci nous a amené à prendre en compte des dimensions qui ne sont pas traitées en ROME du fait de la granularité fine des points de vue. Le langage obtenu permet une programmation par plans et une exécution par contexte des objets respectant les principes du cadre dont il préserve l'ensemble des propriétés. Nous avons également donné dans ce chapitre tous les détails pour une implantation méta-programmée de ce langage.

Dans le dernier chapitre, nous avons étudié l'application de CROME à Smalltalk comme environnement de conception. Cette réalisation a mis à profit le caractère extensible de l'environnement Smalltalk pour en augmenter ses capacités de structuration relativement aux contextes. La solution retenue est basée de façon originale sur l'extension et la systématisation de la technique des catégories qui offrent les bases d'une structuration des classes généralement peu développée. Le résultat est un environnement de conception, offrant en particulier un flâneur adapté, qui permet de décrire systématiquement par contextes les classes des objets Smalltalk.

CROME apporte des réponses aux problèmes posés par les techniques examinées selon chaque angle. Comparé aux techniques de point de vue, CROME se différencie par une systématisation des descriptions sur un ensemble d'objets et une gestion au niveau de granularité supérieur des fonctions, ce qui se traduit par de nombreuses propriétés. Comparé aux techniques conçues pour rendre compte de la structuration en fonctions du système, CROME se différencie en rapportant directement celle-ci à la structuration interne des objets. D'un point de vue structurel, ceci permet de circonscrire les parts de descriptions d'objets contribuant à une fonction. Mais, plus important encore, cela permet de structurer par fonctions les multiples traitements transversaux assurés par les objets.

Finalement, l'approche permet de bénéficer de qualités logicielles examinées en section 10 du chapitre 2. Nous avons ainsi déterminé un gain de modularité au sein des objets eux-mêmes, qui conduit à l'existence d'une modularité de contexte. Nous avons aussi montré que CROME favorise la réutilisation de la structuration en classes du référentiel à travers les contextes ce qui permet d'obtenir une généricité du graphe d'héritage et des graphes d'objets relativement aux fonctions.

Perspectives

Hiérarchie de plans

Initialement, nous avons fait l'hypothèse que toutes les fonctions à décrire sont disjointes

entre elles. Or, il existe parfois des situations où les fonctions présentent des objectifs comparables [Naja 97]. Dans l'exemple de CAO, de telles fonctions pourraient être la simulation graphique et la simulation temporelle des circuits ou encore la normalisation à l'aide de porte Nand et celle à l'aide de porte Nor. Il pourrait maintenant être intéressant d'envisager ce cas de figure et ainsi généraliser l'approche. Ceci pourrait conduire à la constitution d'une hiérarchie de plans où les plans intermédiaires servirait à factoriser des descriptions entre différentes variantes ou sous-fonctions d'une même fonction. Par exemple, le plan simulation serait affiné par deux "sous-plans" apposant respectivement les enrichissements spécifiques à l'animation graphique et à la prise en compte du temps. L'exploitation de cette hiérarchie de plans pourrait alors s'appuyer sur la stratégie d'héritage modulaire proposée entre un plan fonctionnel et le plan de base qu'il serait nécessaire d'étendre à l'ensemble tout en conservant les mêmes principes. Notons par ailleurs que ceci donnerait lieu à des hiérarchies de parties fonctionnelles au sein des objets.

Articulation entre contextes

Un autre approfondissement possible de l'approche concerne les moyens d'articulation entre contextes au sein des objets. Ceux-ci sont limités pour l'instant à du partage et à de l'appel de méthodes. Un problème se pose cependant lorsqu'au sein d'un objet un contexte dépend pour sa cohérence d'un changement d'état ou de l'exécution d'un traitement se produisant dans un autre contexte. Actuellement, la seule solution pour avoir connaissance de ces évènements depuis le contexte dépendant consiste à modifier la définition du contexte pour incorporer des éléments supportant une notification. C'est typiquement la solution adoptée dans l'exemple de coopération entre le contexte d'édition et celui de documentation réalisée au sein du circuit (cf section du chapitre 3). Cette solution est toutefois peu modulaire puisque le contexte fournissant l'information devient finalement dépendant du contexte client. Ce problème situé au sein de l'objet est comparable à celui qui existe dans les implantations classiques de dépendances ou d'interactions entre objets [Ducasse 95][Pintado 95]. Ceci suggère l'étude de nouveaux moyens d'articulation explicites entre contextes alors plus déconnectés.

Une première solution qui reste toutefois à approfondir peut être envisagée. L'idée serait précisement d'autoriser un ou plusieurs contextes à enrichir modulairement une méthode fournie par un autre contexte avec un incrément de traitements spécifiques. Ces traitements seraient alors déclenchés à l'activation de la méthode uniquement, chaque incrément provoquant un changement de contexte au sein de l'objet. Ainsi dans l'exemple précédent, le contexte documentation aurait seulement à définir un incrément de traitement pour chaque méthode du contexte d'édition qui l'intéresse. Comparée à l'appel de méthode, cette solution présente au moins les deux avantages suivants. D'une part, la méthode ne serait plus parasitée avec du code destiné aux autres contextes. D'autre part, l'ajout de nouveaux enrichissements ne nécessiterait plus d'altérer la définition première de la méthode.

La mise en place d'une telle solution pose des questions intéressantes sur l'expression de ces enrichissements qui ne doivent pas influencer la définition initiale. Des pistes sont fournies par le "inner" de BETA [Madsen 93], l'héritage par "mixins" décrit dans [Bracha 90] ou encore les possibilités de combinaison (before/after/around) offertes en CLOS [Paepcke 93].

Evolution dynamique

Une autre perspective de ce travail serait d'étudier une évolution dynamique des objets relativement aux contextes. Rappelons que dans cette étude, nous avons pris l'hypothèse que les

objets ont une représentation figée tout au long de leur vie, celle-ci comprenant toutes les caractéristiques nécessaires à leur fonctionnement dans tous les contextes où ils interviennent.

L'idée serait à présent d'introduire des capacités de contextualisation dynamique d'objets et plus généralement de graphes d'objets. La contextualisation dynamique d'un objet se traduirait par une évolution de sa représentation un peu à la manière de la représentation évolutive de ROME mais relativement à des contextes et non des classes. Pour un graphe d'objets, sa contextualisation consisterait en l'application systématisée sur chacun de ces objets. Une fois contextualisés, les objets seraient alors capables de s'exécuter dans leur nouveau contexte. Grâce à de telles facilités, on pourrait par exemple procéder à la contextualisation d'un circuit dans son ensemble pour pouvoir le manipuler dans un nouveau contexte non prévu lors de la conception du système. De même, des contextualisations alternatives du circuit deviendraient alors possible, le choix se faisant suivant ces caractéristiques ou les résultats des étapes précédentes de son élaboration. Concernant ce dernier point, il serait intéressant d'étudier cette possibilité avec les propositions de hiérarchie de plans discutées précédemment.

De telles capacités supposent l'existence d'un mécanisme de base pour la contextualisation d'un objet dont la sémantique reste à établir. Ce mécanisme devra notamment prendre en compte l'absence d'enrichissement possible dans un contexte mais aussi les dépendances entre les contextes. Différentes stratégies pour gérer ce dernier point sont possibles: refus de la contextualisation tant que l'objet n'est pas contextualisé dans ceux dont il dépend, contextualisation implicite de l'objet dans les autres contextes,... Ce qui mérite tout une étude en soi.

Statut des contextes

A travers ses notions et principes, CROME concourt à donner un statut aux contextes. Il fournit à cet effet le support et les mécanismes nécessaires à leur description et leur articulation. Nous pensons que des améliorations sont possibles pour donner davantage de consistance aux contextes aussi bien pour la description du système que pour son exécution.

Une première amélioration est la possibilité d'avoir plusieurs occurences d'un même contexte à l'exécution[1] ce qui peut être intéressant dans le cadre d'environnements de conception pour supporter par exemple différentes alternatives du produit à élaborer. Cela se traduirait au sein des objets par l'existence de plusieurs occurences de variables d'instances, une pour chaque occurence de contexte. Une source d'inspiration intéressante à ce sujet est [Shilling 89] qui propose une extension du modèle à classes supportant de telles occurences (cf 2.4.2). Signalons qu'une telle caractéristique soulève de nombreux points à étudier: création de ces occurences, référence à ces occurences, articulation entre des occurences de contextes différents, ...

Une seconde amélioration consiste à donner un statut d'environnement d'objets aux contextes. Ceci permettrait par exemple le partage implicite d'objets entre tous les participants à un contexte. Une telle caractéristique passe par la possiblité d'associer des propriétés aux contextes seulement accessible à leurs participants.

CROME et Programmation par Aspects

1. Une telle possiblité est notamment suggérée dans [Harrison 93] où il est question de plusieurs activations d'un même sujet.

La programmation par aspects ("Aspect-Oriented Programming" ou AOP) est un nouveau courant de recherche, initié dans [Kiczales 97] dont le but est la séparation des composants du domaine fonctionnel des multiples dimensions techniques (synchronisation, persistence, répartition, distribution, gestion mémoire, ...) qui s'entrecroise ("cross-cut") généralement dans la descripton d'un système. Le principal problème vient de ce que la modularité des constructions fournies par les langages existants (procédures, modules, objets, méthodes) ne permet pas facilement de circonscrire et d'exprimer clairement de tels aspects.

Pour résoudre ce problème, la programmation par aspects vise à fournir les méthodes et techniques pour arriver à expliciter chaque dimension technique et à les localiser séparemment des composants fonctionnels. Ceci amène à étudier dans un premier temps des langages d'aspects spécifiques qui viennent en complément des langages de composants. Dans un second temps, il s'agit d'étudier des techniques dites de "tissage" ("weaving") qui vise à reconstituer l'entrecroisement des composants et des aspects ainsi séparés de façon à obtenir le code devant être finalement exécuté.

CROME est différent dans ses objectifs de la programmation par aspects. En effet, même si il étudie également des problèmes de multiplicité et de transversalité dans un système, ceux-ci sont d'un autre caractère tout comme la programmation par sujets (SOP pour Subject Oriented Programming) : ils se situent au sein du niveau fonctionnel. Ce qui rend orthogonales les deux approches (cf figure 6.1) .

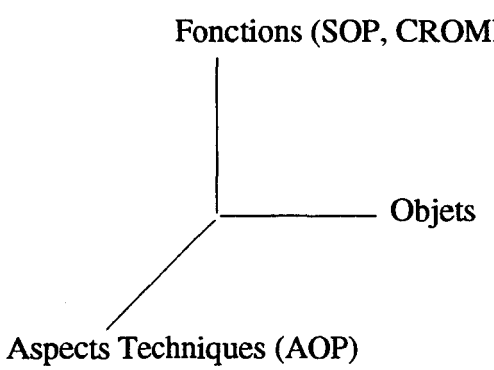


figure 6.1

Cependant, malgré ces différences au niveau de objectifs, ces deux approches se comparent quand aux difficultés posées (entrecroisement, transversalité) et au moyen utilisé pour les resoudre (langage d'aspects, plan). Il nous semble intéressant d'étudier un rapprochement entre ces deux approches à travers leurs solutions techniques pour finalement retenir un seul moyen d'explicitation des multiples dimensions transversales (fonctionnelles ou techniques) d'un système. Ceci permettrait de rejoindre la tendance actuelle de généraliser le courant AOP à la SOP [Czarnecki 99][Mens 97] et par là à notre approche.

CROME et Conception concourante (Concurrent engineering)

Dans ce cadre, deux voies peuvent être explorées suivant que l'on considère CROME comme outil de conception concourante d'un référentiel d'objets ou comme outil pour le développement d'environnements de conception concourante.

De par ses principes de structuration, CROME fournit un support pour le développement

concourant d'un même référentiel d'objets. En effet, le découpage en plan fonctionnels donne la possiblité de confier en même temps le développement du même ensemble de classe à différents programmeurs. Il faut savoir que cette conception concourante est difficile avec les techniques classiques du fait de l'incapacité de pouvoir définir séparement des parties d'une même classe [Ossher 94]. Ce qui nécessite en général de recourir à des environnements spécialisés.

Une première expérimentation de CROME en tant qu'outil de conception concourante a été menée avec l'environnement Smalltalk pour l'exemple de la conception de circuit. Dans cette expérimentation, le développement de l'application s'est fait en distribuant à chaque programmeur une même définition du plan de base que celui-ci a pu enrichir et testé dans son environnement. L'intégration de l'ensemble a alors été obtenue facilement en ramenant les plans élaborés séparement dans le même environnement.

Cette expérimentation nous a permis de valider CROME comme outil de conception concourante. Il serait à présent nécessaire de le comparer plus finement à d'autres environnements existants pour identifier ses apports originaux afin de les développer mais aussi déterminer ce qui lui manque pour en faire un véritable environnement. Dans cette perspective, on peut envisager la prise en compte de versions ou encore une gestion de la concurrence lors du développement des mêmes classes.

L'étude de CROME s'est faite dans le cadre d'un langage séquentiel ce qui a conduit à l'hypothèse qu'un objet s'exécute toujours dans un seul contexte à la fois. Son application pour la construction d'environnements de conception concourante [Decouchant94][Gruber 92] ouvre des perspectives pour son approfondissement dans un cadre de langage concurrent à objets. Ces environnement doivent en effet être capables de supporter simultanément plusieurs activités distinctes d'élaboration d'un même produit. L'extension de CROME à la concurrence pose des questions intéressantes sur l'association des flots d'exécution. Faut-il les associer aux objets, aux contextes ou aux deux ? Même si cela reste à étudier, nous pensons que l'association la plus intéressante consiste à gérer des flots associés à chaque contexte, l'exécution restant séquentiel à l'intérieur du contexte. Ceci conduirait les objets du référentiel à être actifs dans plusieurs contextes à la fois. Par exemple, un même circuit pourrait ainsi être simultanément documenté et simulé par des utilisateurs différents. D'autres éléments à étudier pour une prise en compte de la concurrence sont des moyens de synchronisation (sémaphore, rendez-vous) au sein des objets de façon à gérer de manière cohérente l'articulation entre contextes concurrents.

Cette étude sur CROME pourrait se faire conjointement avec son étude dans un environnement distribué. On pourrait alors s'inspirer des travaux en cours menés par Olivier Caron sur l'implantation de CROME orienté système d'information [Debrauwer 98] dans un environnement Corba.

Annexe

Définition de l'exemple de conception de circuits

Nous avons choisi d'illustrer les principes de notre approche en développant un exemple simplifié de système d'aide à la conception de circuits logiques combinatoire[1] et non hiérarchique[2] inspiré de systèmes existants développés selon une démarche de conception par objets [VanderMeulen 87][Hollant 90].

Dans les prochains paragraphes, nous décrivons les divers éléments intervenant dans la constitution des circuits. Cette présentation va servir à établir un modèle de représentation de tels circuits. Elle sera suivie d'une description des différentes fonctions globales envisagées dans le cadre de cet exemple.

Un circuit logique est constitué de portes logique élémentaires. Chaque porte possède des broches qui sont soit des entrées, soit des sorties. Par l'intermédiaires de ces entrées (resp. sorties), un porte recoit (resp. émet) un signal logique qui prend seulement deux valeurs possibles, 0 et 1. Une porte réalise une fonction logique préféfinie qui calcule la valeur des sorties en fonction de la valeur des entrées. Pour limiter la taille de l'exemple, nous distinguerons par la suite les quatre types de portes élémentaires suivants[3]: les portes Non (appélé aussi inverseur), les portes Et, les portes Ou et les portes NonEt.

Il est possible de connecter ces portes élémentaires pour former des fonctions logiques plus complexes (celle des circuits). La connexion est réalisée par une équipotentielle. Une équipotentielle relie une sortie à plusieurs entrées et permet de transporter le signal logique entre les deux portes. La figure A.1 illustre sur un exemple simple la structure et les constituants d'un circuit logique correspondant à un demi-additionneur. Ce circuit consiste en l'assemblage d'une porte Ou, deux portes Et et une porte Non (représenté ici avec leur symboles graphiques usuels).

1. Fondamentalement deux catégories de circuits coexistent dans les systèmes logiques: ceux dont la fonction de sortie s'exprime selon une expression logique des seules variables d'entrée les circuits combinatoires et ceux dont la sortie dépend non seulement de l'état des variables d'entrée mais également de l'état antérieur de certaiens variables de sortie, les circuits séquentiels ou à mémoire, qui présente des propriétés de mémorisation.

2. Pour simplifier la présentation, nous laissons de coté certains problèmes traités dans les systèmes existants comme la conception hiérarchique des circuits, le traitement de boucle dans le circuit et l'élaboration de modèles de circuit.

3. En réalité, seules les portes NonEt sont réellement nécessaires puisqu'elles permettent de réaliser par combinaison toutes les autres portes de façon équivalente. Dans la pratique, cependant, on n'élabore jamais un circuit uniquement à partir de telles portes car cela rendrait sa conception et son interprétation trop complexe.

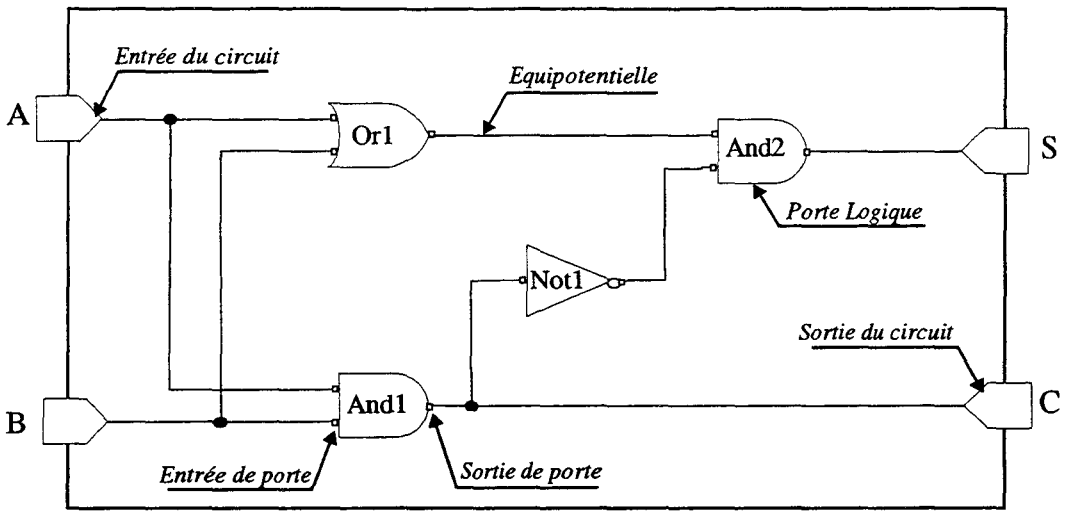


figure A.1 Circuit demi-additionneur

Au même titre que les portes élémentaires qui le constituent, un circuit logique possède également des broches d'entrées et de sorties, appelées usuellement des ports, qui sont directement connectés aux portes internes. Ces ports servent à recevoir (resp. à transmettre) les signaux provenant de (resp. vers) l'extérieur. Ils permettent notamment de réutiliser le circuit conçu dans un circuit plus complexe.

En ce qui concerne les fonctions envisagées pour notre système, elles sont au nombre de cinq. Celle-ci sont toutes complémentaires et ne peuvent se substituer l'une à l'autre. Ces fonctions sont les suivantes:

- Edition schématique: Cette fonction permet de définir intéractivement et visuellement la structure d'un circuit logique évitant ainsi au concepteur d'utiliser un langage de description de circuit. Les différents constituants, portes et connexions, sont présentés graphiquement avec leurs symboles usuels et sont configurés par manipulation directe. Outre les contraintes graphiques (ex: pas de chevauchement entre symboles de portes), cette fonction doit également prendre en compte les contraintes structurelles liées à la nature des entités manipulées (ex: interdiction de lier plusieurs sorties à une même entrée)

- Simulation Logique: Cette fonction fournit une simulation logique des circuits. La simulation envisagée à la base est fort simple et se caractérise par les propriétés suivantes: évaluation synchrone, régime permanent, prise en compte du temps de traversée des portes, pas de prise en compte de la fréquence. Des facilités pour espionner les changements d'états de certains composants lors de la simulation sont également proposées par cette fonction.

- Documentation: Cette fonction est destinée à associer des informations diverses au circuit (nom du concepteur, date de création , projets l'utilisant, fonction du circuit, ...) et à ses éléments (texte explicatif indiquant des choix de conception). Cette fonction gère ausi automatiquement une historique des manipulations effectuées sur le circuit au cours de son édition. Enfin, elle permet également le tirage de la nomenclature du circuit enrichie des informations précédentes.

- Optimisation: Une fois le circuit testé et validé, des optimisations sont souvent possibles.

Cette fonction a pour but de créer automatiquement un circuit équivalent[1] au circuit conçu dans lequel le nombre de portes et de connexions a été minimisé. L'obtention d'un circuit équivalent est rendu possible par l'utilisation des lois classiques (distribution, absorption) de l'algèbre de Boole.

- Normalisation: Cette dernière fonction vise également à créer un circuit équivalent au circuit conçu dans une forme normalisée, c'est à dire constitué essentiellement d'un seul type de porte, en général des portes NonEt. Ce circuit est obtenu en remplaçant chaque porte d'un type particulier par un combinaison de portes NonEt correspondante.

1. Un circuit est équivalent à un autre si et seulement si, les valeurs de sorties sont les mêmes pour toutes les configurations identiques de leurs entrées.

Plan de base

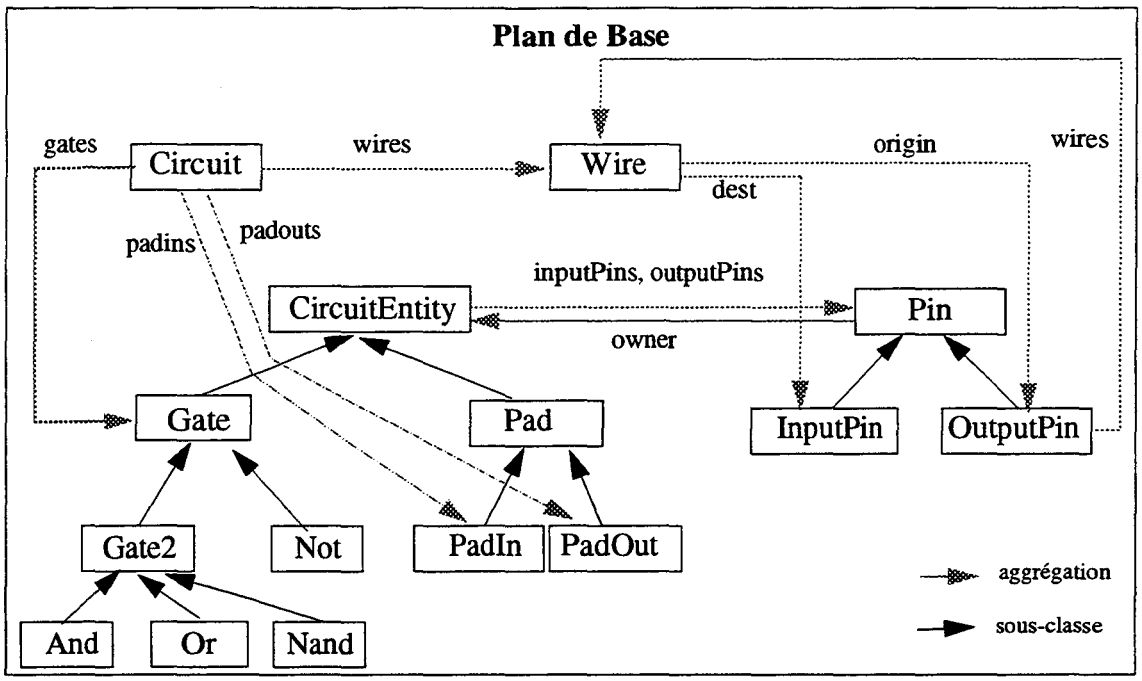


figure A.2

```
(define CADObject
   (CR-CLASS
      'supercl '(CR-OBJECT)
      'attributs '(name)
      'privmth  '(
        initialize (lambda (l-init)
                        (<- 'name (assq 'name l-init)))
      'publmth  '(
        name<- (lambda (symb) (<- 'name symb))
        name (lambda () (? 'name)))
))

(define Wire
   (CI-CLASS
      'supercl '(CADObject)
      'attributs '(origin dest)
      'privmth  '(
           ;; cette methode initialise une equipotentielle avec une broche d'entrée et
           ;; une broche de sortie en vérifiant leur validité

           initialize
              (lambda (l-init)
                 (let ((o (assq 'origin l-init))
                       (i (assq 'dest l-init)))
                 (super)
                 (if (eq? o i)
                     (self 'error "output and input pins can not be the same")
                     (begin
                       (cond ((not (o 'connect-as-output))
                              (self 'error "wrong output"))
                              ((not (i 'connect-as-input))
```

```
                              (self 'error "wrong input"))
                              (else (<- 'origin o) (o 'connect-to-wire self)
                                   (<- 'dest i) (i 'connnect-to-wire self))))))
      'publmth  '(
            output (lambda () (? output))
            input (lambda () (? input))
            connected-to? (lambda (gate)
                              (or (input 'gate=? gate)
                                  (output 'gate=? gate)))
            disconnect-all
               (lambda ()
                  ((? 'input) 'disconnect-wire self)
                  ((? 'output) 'disconnect-wire self)
                  (<- 'input #nil)
                  (<- 'output #nil))
   ))

(define Pin
   (CR-CLASS
      'supercl '(CADObject)
      'attributs '(gate)
      'privmth  '(
            initialize (lambda (l-init)
                           (<- 'gate (assq 'gate l-init)))
      'publmth  '(
            gate (lambda () (? 'gate)
            gate=? (lambda (gate) (eq? gate (? 'gate)))
            connected? (lambda () abstract)
            connect-as-output? (lambda () #false)
            connect-as-input? (lambda () #false)
            connect-to-wire (lammbda () #abstract)
            disconnect-wire (lammbda () #abstract)
            disconnect-all (lammbda () #abstract)

   ))

(define OutputPin
   (CI-CLASS
      'supercl '(Pin)
      'attributs '(wires)
      'privmth  '()
      'publmth  '(
            connected? (lambda () (eq? (? 'wires) #nil)
            connect-as-output? (lambda () #true)
            connect-to-wire (lambda (wire)
                           (when (not (member? wire (? 'wires))
                              (<- 'wires (cons wire (? 'wires)))))
            disconnect-wire (lambda (wire)
                                 (when (member? wire (? 'wires))
                                    (<- 'wires (remove wire (? 'wires)))))
            disconnect-all (lambda ()
                              (when (self 'connected?)
                                 (for-each (lambda (w)
                                                (w 'disconnect-all))
                                           (? 'wires))
                                 (<- 'wires #nil))))
   ))

(define InputPin
   (CI-CLASS
      'supercl '(Pin)
      'attributs '(wire)
      'privmth  '()
```

```
      'publmth  '(
            connected? (lambda () (eq? (? 'wire) #nil)
            connect-as-input? (lambda () (self 'connected?))
            connect-to-wire (lambda (wire)
                                    (if (self 'connected?)
                                       (self 'error "input already connected")
                                       (<- 'wire wire)))
            disconnect-wire (lambda (wire)
                                    (when (eq? wire (? 'wire))
                                       (<- 'wire #nil)))
            disconnect-all (lambda ()
                                  (wire 'disconnect-all))
   ))

(define CircuitEntity
   (CR-CLASS
      'supercl '(CADObject)
      'attributs '()
      'publmth  '(
        detach-all (lambda () #abstract)
   ))

(define Gate
   (CR-CLASS
      'supercl '(CircuitEntity)
      'attributs '(output)
      'publmth  '(
        pin-at (lambda (n) (lambda #abstract))
        pin-named (lambda (n) (lambda (#abstract))
   ))


(define NotGate
   (CI-CLASS
      'supercl '(Gate)
      'attributs '(input)
      'privmth  '(
        initialize (lambda (l-init)
                        (super)
                        (<- 'input (InputPin 'new 'gate self 'name 'input))
                        (<- 'output (OutputPin 'new 'gate self 'name 'output)))
      'publmth  '(
            pin-at (lambda (i)
                       (cond ((= i 1) input)
                             ((= i 2) output)
                             (else (self 'error "wrong pin number"))))
            pin-named (lambda (symb)
                        (cond ((input 'name=? symb) input)
                              ((output 'name=? symb) output)
                              (else (self 'error "wrong pin name")))))
            detach-all (lambda ()
                             (input 'disconnect-all)
                             (output 'disconnect-all)))
   ))



(define Gate2
   (CR-CLASS
      'supercl '(Gate)
      'attributs '(input1 input2 output)
      'privmth  '(
            initialize
```

```
            (lambda (l-init)
                  (super)
                  (<- 'input1 (InputPin 'new 'gate self 'name 'input1))
                  (<-'input2) (InputPin 'new 'gate self 'name 'input2))
                  (<-'output (OutputPin 'new 'gate self 'name 'output))))
      'publmth  '(
            pin-at (lambda (i)
                        (cond ((= i 1) input1)
                              ((= i 2) input2)
                              ((= i 3) output)
                              (else (self 'error "wrong pin number"))))
            pin-named (lambda (symb)
                           (cond ((input1 'name=? symb) input1)
                                 ((input2 'name=? symb) input2)
                                 ((output 'name=? symb) output)
                                 (else (self 'error "wrong pin name")))))
            detach-all (lambda ()
                              (input1 'disconnect-all)
                              (input2 'disconnect-all)
                              (output 'disconnect-all)))
   ))


(define AndGate
   (CI-CLASS
      'supercl '(Gate2)))

(define OrGate
   (CI-CLASS
      'supercl '(Gate2)))

(define NandGate
   (CI-CLASS
      'supercl '(Gate2)))

(define Pad
   (CR-CLASS
      'supercl '(CircuitEntity)
      'attributs '(circuit))

(define PadIn
   (CI-CLASS
      'supercl '(Pad)
      'attributs '(output)
      'privmth  '(
        initialize
           (lambda (l-init)
              (super)
              (<-'output (OutputPin 'new 'gate self 'name 'output))))
      'publmth  '(
        detach-all
           (lambda ()
              (output 'disconnect-all))))
   ))

(define PadOut
   (CI-CLASS
      'supercl '(Pad)
      'attributs '(input)
      ''privmth  '(
        initialize
           (lambda (l-init)
              (super)
```

```
            (<-'input (InputPin 'new 'gate self 'name 'input))))
      'publmth  '(
        detach-all
           (lambda ()
             (input 'disconnect-all))))
   ))


(define Circuit
   (CI-CLASS
      'supercl '(CADObject)
      'attributs '(padins padouts gates wires)
      'privmth  '(
        add-gate
             (lambda (gate)
                (when (not (memq gate (? 'gates)) ;; detection nom
                   (<- gates (cons gate (? 'gates)))))



        add-wire (lambda (wire)
                        (when (not (memq wire (? 'wires)))
                           (<- 'wires (cons wire (? 'wires))))


        add-input (lambda (name)
                      (when (null? (self 'input-named name))
                           (<- 'padins (cons (PadIn 'new 'name name) (? 'padins)))))

        add-output (lambda (name)
                        (when (null? (self 'output-named name))
                           (<- 'padouts (cons (PadOut 'new 'name name) (? 'padouts)))))

        remove-input (lambda (name)
                       (let ((inp (self 'input-named name)))
                          (when inp
                             (<- 'padins (remove inp (? padins)))
                             (inp 'disconnect-all)))


        remove-output (lambda (name)
                          (let ((outp (self 'output-named name)))
                             (when inp
                                (<- 'padouts (remove outp (? padouts)))
                                (outp 'disconnect-all)))


        remove-wire (lambda (wire)
                        (<- 'wires (remove wire (? wires)))
                        (wire 'disconnect-all)))


        remove-gate (lambda (gate)
                        (when (memq gate (? 'gates))
                           (gate 'detach-all)
                           (<- 'gates (remove gate (? 'gates)))))
                           (for-each (lambda (w)
                                       (when (w 'connected-to? gate)
                                          (w 'disconnect-all)))
                                     (? gates))


        valid? (lambda ()...))
```

```
      'publmth  '(
        gate-named (lambda (symb)
                     (detect (lambda (gate)
                                (eq? (gate 'name) symb))
                             (? 'gates)))
        wire-named (lambda (symb)
                     (detect (lambda (wire)
                                (eq? (gate 'name) symb))
                             (? 'wires)))
)
   ))
```

Plan fonctionnel Simulation

Explications sur la mise en oeuvre

Chaque unité de la simulation est réalisée par la propagation du message `propagate-tick` à travers les objets du circuts. Cette propagation permet de mettre à jour les compteurs associés localement aux objets Porte et Equipotentielle pour prendre en compte leur délai spécifique dans la propagation des signaux.

```
(Wire
   'extend-in 'Simulation
   'attributes  (counter)
   'privmth  (delay (lambda () 2))
   'publmth  (
      reset-counter (lambda () (<- 'counter (self 'delay)))
      propagate-tick
         (lambda ()
            (<- 'counter (- (? 'counter) 1))
            (if (= (? 'counter) 0)
               (begin
                  (dest 'sets (? 'origin gets))
                  (self 'reset-counter)))
            (dest 'propagate-tick))))

(Pin
   'extend-in 'Simulation
   'attributes  (state probed)
   'privmth  (
      print-probe (lambda ()
                     (display "Pin:" (self 'name) "of gate:" (gate 'name))
                     (display "new value =" (? 'state))))
   'publmth  (
      probe (lambda () (<- 'probed #true))
      unprobe (lambda () (<- 'probed #false))
      propagate-tick (lambda () #abstract))
      sets (lambda (signal)
              (when (eq? signal state)
                 (<- 'state signal)
                 (if (? 'probed) (self 'print-probe))))
      gets (lambda () (? 'state))))

(InputPin
   'extend-in 'Simulation
   'attributes '(visited)
   'publmth  (
      visited (lambda () (? 'visited))
      unmark (lambda () (<- 'visited #f)
      propagate-tick (lambda () ((? 'gate) 'propagate-tick))))

(OutputPin
   'extend-in 'Simulation
   'publmth  (
      propagate-tick
         (lambda ()
            (for-each (lambda (w) (w 'propagate-tick) (? 'wires))))

(CircuitEntity
   'extend-in 'Simulation
   'publmth  (
      propagate-tick (lambda () #abstract))))
```

```
(Gate
   'extend-in 'Simulation
   'attributes  (counter)
   'privmth  (
      evaluate (lambda () #abstract)
      delay (lambda () #abstract)
   'publmth  (
      propagate-tick
         (lambda ()
            (<- 'counter (- (? 'counter) 1))
            (if (= (? 'counter) 0)
               (begin
                  (self 'evaluate)
                  (<- 'counter (self 'delay))))
            (output 'propagate-tick))

      unprobe (lambda () #abstract)
      probe (lambda () #abstract)))


(NotGate
   'extend-in 'Simulation
   'privmth '(
      evaluate (lambda () ((? output) 'sets (not ((? 'input) 'gets)

   'publmth  (
      propagate-tick (lambda () ((? input) 'unmark) (super))
      probe (lambda ()
               ((? 'input) 'probe)
               ((? 'output) 'probe))
      unprobe (lambda ()
                  ((? 'input) 'unprobe)
                  ((? 'output) 'unprobe))


(Gate2
   'extend-in 'Simulation
   'privmth  (
      delay (lambda () 10)
   'publmth  (
      propagate-tick (lambda ()
                        (if (and ((? input1) 'visited) ((? 'input2) 'visited))
                           (begin ((? 'input1) 'unmark)
                                  ((? 'input2) 'unmark)
                                  (super))))
      probe (lambda ()
               ((? 'input1) 'probe)
               ((? 'input2) 'probe)
               ((? 'output) 'probe))
      unprobe (lambda ()
                  ((? 'input1) 'unprobe)
                  ((? 'input2) 'unprobe)
                  ((? 'output) 'unprobe))))

(AndGate
   'extend-in 'Simulation
   'privmth  (
      evaluate (lambda ()
                  (let ((iv1 ((? 'input1) 'gets)))
                        (iv2 ((? 'input2) 'gets))))
                   ((? 'output) 'sets (and iv1 iv2)))
   ))
```

```
(OrGate
   'extend-in 'Simulation
   'privmth  (
      evaluate (lambda ()
                 (let ((iv1 ((? 'input1) 'gets)))
                      (iv2 ((? 'input2) 'gets))))
                  ((? 'output) 'sets (or iv1 iv2)))
   ))

(NandGate
   'extend-in 'Simulation
   'privmth  (
      delay (lambda () 6)
      evaluate (lambda ()
                 (let ((iv1 ((? 'input1) 'gets)))
                      (iv2 ((? 'input2) 'gets))))
                  ((? 'output) 'sets (not (and iv1 iv2))))
   ))

(Pad
   'extend-in 'Simulation
   'publmth  (gets (lambda () #abstract))

(PadIn
   'extend-in 'Simulation
   'publmth  (
      gets (lambda () ((? 'output) 'gets))
      sets (lambda (signal) ((? 'output) 'sets signal))))

(PadOut
   'extend-in 'Simulation
   'publmth  (
      gets (lambda () ((? 'input) 'get))))

(Circuit
   'extend-in 'Simulation
   'publmth  (
      run (lambda (t)
            (when (self 'check)
               (do 1 t  (for-each (lambda (ip) (ip 'propagate-tick)) (? 'padins)))))
      step (lambda () (self 'run 1)
      reset (lambda ()
               (for-each (lambda (w) (w 'reset-counter)) (? 'wires)))
               (for-each (lambda (g) (g 'reset-counter)) (? 'gates))))
      unprobe-gate (lambda (name) ((self 'gate-named name) 'unprobe)
      probe-gate (lambda (name) ((self 'gate-named name) 'probe)
      set-input-at (lambda (index signal) (self 'pin-at index) 'sets signal))
      get-input-at (lambda (index) ((self 'pin-at index) 'gets))
      set-input-named (lambda (name signal) ((self 'pin-named name) 'sets signal)
      get-output-named (lambda (name) ((self 'pin-named name) 'gets))))
```

Lancement d'une simulation

```
(with (dm1 'in 'Simulation)
   (dm1 'reset)                          ; on reset le circuit
   (dm1 'probe-gate 'Or1)                ; on veut espionner la porte Or1
   (dm1 'probe-gate 'Not1)               ; on veut espionner la porte Not1
   (dm1 'set-input-named 'A #true)       ; les deux entrées sont configurées
   (dm1 'set-input-named 'B #false))
   (dm1 'run 10)                         ; simulation pendant 10 unités de temps
   (display (dm1 'get-output-named 'S))  ; lecture du resultat au niveau des sorties S et C
   (display (dm1 'get-output-named 'C)))
```

Plan fonctionnel Edition Schématique

```
(Wire
   'extend-in 'Schematic
   'attributes  (selected)
   'publmth  (
      select (lambda () (<- 'selected #true))
      unselect (lambda () (<- 'selected #false))
      display-on (lambda (canvas)
                    (let ((p1 (input 'location))
                          (p2 (output 'location)))
                       (canvas 'draw-line ...)
                       (canvas 'draw-line ...)
                       (canvas 'draw-line ...)
      cross-over? (lambda (g) ...)
      selected-at?  (lambda (location) ...)

(Pin
   'extend-in 'Schematic
   'attributes  (relative-loc))
   'publmth  (
      location (lambda () ((gate 'location) 'plus (? 'relative-loc))
      display-on (lambda (canvas) #abstract)
      selected-at? (lambda (loc) ((Rectangle new (self 'location) 4 4) 'contains? loc)

(InputPin
   'extend-in 'Schematic
   'publmth  (
      display-formOn (lambda (canvas)
                        (canvas 'display-form InputPinForm (self 'location)))

(OutputPin
   'extend-in 'Schematic
   'publmth  (
      display-formOn (lambda (canvas)
                        (canvas 'display-form OutputPinForm (self 'location)))



(CircuitEntity
   'extend-in 'Schematic
   'attributes '(location)
   'privmth  (
      display-frameOn (lambda (canvas) (canvas 'display-rectangle (self 'bounds)))
      display-formOn (lambda (canvas) #abstract)
      display-nameOn (lambda (canvas) (canvas 'display-text (? 'name) (? 'location))))
      bounds (lambda () #abstract))
   'publmth  (

      locked? (lambda () (? 'locked))
      lock (lambda () (<- locked #true))
      unlock (lambda () (<- locked #false))
      select (lambda () (<- 'selected #true))
      unselect (lambda () (<- 'selected #false))
      display-on
            (lambda (canvas)
                  (self 'display-formOn canvas))
                  (when (not (null? (? 'name))
                     (self 'display-nameOn canvas))
                  (when (? selected)
                        (self 'display-frameOn canvas)))

      move-to (lambda (newloc) (when (not (? 'locked)) (<- 'location newloc)))
```

```
      selected-at? (lambda (loc) ((self 'bounds) 'contains? loc))))
))

(Gate
   'extend-in 'Schematic
   'publmth  (
      pin-at-location (lambda () #abstract))

))


(NotGate
   'extend-in 'Schematic
   'privmth  '(
      display-formOn (lambda (canvas)
                       (let ((loc (? 'location)))
                          ((? 'input) 'display-on canvas)
                          ((? 'output) 'display-on canvas)))

      bounds (lambda () (Rectangle 'new 'origin (? 'location) 'with 10 'height 15))
   'publmth  '(
      pin-at-location
         (lambda (loc)
            (cond ((? 'input) 'selected-at? loc) (? 'input))
                  ((? 'output) 'selected-at? loc) (? 'output))
                  (else '())))
   ))


(Gate2
   'extend-in 'Schematic
   'privmth  '(
      display-formOn (lambda (canvas)
                       (let ((loc (? 'location)))
                          ((? 'input1) 'display-on canvas)
                          ((? 'input2) 'display-on canvas)
                          ((? 'output) 'display-on canvas)))

      bounds (lambda () (Rectangle 'new 'origin (? 'location) 'with 10 'height 20))
   'publmth  '(
      pin-at-location
         (lambda (loc)
            (cond ((? 'input1) 'select loc) (? 'input1)
                  ((? 'input1) ' loc) (? 'input2))
                  ((? 'output) ' loc) (? 'output))
                  (else '()))))
   ))


(AndGate
   'extend-in 'Schematic
   ''privmth  (
      display-formOn (lambda (canvas)
                          (canvas 'display-form AndForm (? 'location))
                          (super)))
   ))


(OrGate
   'extend-in 'Schematic
   'privmth  (
      display-formOn (lambda (canvas)
                          (canvas 'display-form OrForm (? 'location))
```

```
                        (super)))
   ))

(NandGate
   'extend-in 'Schematic
   'privmth  (
     display-formOn (lambda (canvas)
                        (canvas 'display-form NandForm (? 'location))
                        (super)))
   ))


(PadIn
   'extend-in 'Schematic
   'privmth  (
     display-formOn (lambda (canvas)
                      (canvas 'display-form PadInForm (? 'location))
                      (super))
))

(PadOut
   'extend-in 'Schematic
   'privmth  (
     display-formOn (lambda (canvas)
                      (canvas 'display-form PadOutForm (? 'location))
                      (super))
   ))


(Circuit
   'extend-in 'Schematic
   'attributes  (grid selection window)
   'privmth  (


     find-pin-at (lambda (loc)
                   (let ((g (detect (lambda (gate)
                                      (gate 'pin-at-location loc))
                                    (? 'gates)))
                      (if (null? g) '() g)))

     find-wire-at (lambda (loc) (detect (lambda (wire)
                                          (wire 'selected-at? loc)
                                        (? 'wires)))
     find-gate-at (lambda (loc) (detect (lambda (gate)
                                          (gate 'selected-at? loc)
                                        (? 'gates)))

     invalidate (lambda () ((? 'window) redisplay-model))

   'publmth  (
     display-on (lambda (canvas)
                  (grid 'display-on canvas)
                  (for-each (lambda (gate) (gate 'display-on canvas)) (? 'gates))
                  (for-each (lambda (wire) (wire 'display-on canvas)) (? 'wires))))

     view (lambda ()
            (if (? 'window)
               (? window 'place-at-top)
               (begin
                  (<- window (GraphicWindow 'new
                                'model self
                                'display-selector 'display-on)
```

```
            (window 'open))))

edit (lambda ()
        (if (not (null? (? 'window)))
           ((? window) 'place-at-top)
           (begin
              (<- window (GraphicWindow 'new
                            'model self
                            'display-selector 'display
                            'change-selector 'set-selection)
                            'menu-selector 'process-menu
              (window 'open))

cmd-close (lambda () (<- 'window '()))

cmd-addGate
     (lambda ()
        (let* ((collide '())
               (res (PopUpMenu 'new 'labels '("And" "Or" "Not" "PadIn" "PadOut")
                            'location (Cursor 'current-location))))
               (newgate (case res
                         (("And") (AndGate 'new))
                         (("Or") (OrGate 'new))
                         ...
                         (else '()))))
           (when (not (null? newgate))
              (newgate 'move-to (grid 'nearest-to (Cursor 'select-location))
              (set! collide (detect (lambda (g)
                                       (gate 'intersect-with g))
                                    (? 'gates)))
              (when (not collide)
                 (set! collide (detect (lambda (w)
                                       (w 'cross-over g))
                                    (? 'wires)))
                 (when (not collide)
                    (self 'addGate newgate)
                    (self 'invalidate))))

;; ajout
cmd-addWire
     (lambda ()
        (let* ((input '()) (output '()) (collide '()) (new-wire '()))
           (set! output (self 'find-pin-at (Cursor 'select-location))
           (when output
              (set! input (self 'find-pin-at (Cursor 'select-location))
              (when input
                 (set! new-wire (Wire 'new 'output output 'input input))
                 (set! collide (detect (lambda (g)
                                          (new-wire 'cross-over g))
                                       (? 'gates)))
                 (when (not collide)
                    (self 'add-Wire new-wire)
                    (self 'invalidate)))))

;; suppression d'une equipotentielle

cmd-removeWire
     (lambda ()
        (when (Confirm 'new 'label "Do you really want to delete this wire)
           (self 'remove-wire (? 'selection))
           (self 'invalidate)))

cmd-removeGate
```

```
      (lambda ()
         (when (Confirm 'new 'label "Do you really want to delete this gate)
               (self 'remove-gate (? 'selection))
               (self 'invalidate)))

;;; pemet de déplacer la selection courante

cmd-move
      (lambda ()
         (let ((loc (Cursor 'select-location))
            ;; On verifie de ne deplacer la selection sur un element du circuit
            (for-each (lambda ()

            ((? 'selection) 'move-to (grid 'nearest-to loc))
            (self 'invalidate))))

;;; permet de determiner si l'objet peut être deplace ou non

cmd-lock
      (lambda () ((? 'selection) 'lock))

cmd-unlock
      (lambda () ((? 'selection) 'unlock))

;;; permet de modifier la grille du circuit

cmd-setGrid
      (lambda ()
            (grid 'set-value (ValueDialog 'new 'label "Grid new value"
                                    'default (grid 'value)))
            (for-each (lambda (gate)
                        (gate 'move-to (grid 'nearest-to (gate 'location))))
                  (? 'gates))
            (self 'invalidate))

;; selectionne l'entité du circuit se trouvant à la position loc

set-selection
      (lambda (loc)
         (let* ((selobj (self 'find-gate-at loc)))
            (when (null? selobj)
               (set! 'selobj (self 'find-wire-at loc)))
            (when (not (null? selobj))
               ((? 'selection) 'unselect)
               (selobj 'select)
               (<- 'selection selobj)
               (self 'invalidate))))


;; affiche le menu principal en fonction de la selection en cours

process-menu
      (lambda (loc)
         (let ((location (Cursor 'current-location))
                (label '())
                (selectors '())
         (cond ((null? (? 'selection))
                (set! 'labels '("Add Wire" "Add Gate" "Set Grid"))
                (set! 'selectors '(cmd-addWire cmd-addGate cmd-setGrid)))
               (((? selection) 'isWire)
                (set! 'labels '("Delete" "Rename"))
                (set! 'selectors '(cmd-deleteWire cmd-rename)))
               (((? selection) 'isGate)
```

```
        (set! 'labels '("Move" "Lock" "Unlock" "Delete" "Rename"))
        (set! 'selectors '(cmd-moveGate cmd-lock cmd-unlock
            cmd-deleteGate cmd-rename))))
    (PopUpMenu 'new 'receiver self 'labels labels
            'selectors selectors 'location loc)))
```

Lancement d'une édition

```
((dm1 in 'Schematic) 'edit)
```

Plan fonctionnel Documentation

```
(Wire
   'extend-in 'Documentation
   'publmth  (
      list-infos (lambda (stream)
                     (stream 'put (? 'name))
                     (stream 'put
                        "connect: input ((? 'origin) 'name) "of "
                           (((? 'origin) 'gate) 'as-string) "to output: "
                           ((? 'dest) 'name) "of (((? 'origin) 'gate) 'as-string))))


(Gate
   'extend-in 'Documentation
   'attributes '(status notes)
   'privmth '(
        type-string (lambda () #abstract))
        list-structure (lambda () #abstract))
   'publmth  (
      as-string (lambda ()
                  (string-append (self 'type-string) (? 'name))

      list-infos (lambda (stream)
                     (stream 'put (g 'as-string))
                     (self 'list-structure stream)
                     (stream 'put (? status))
                     (for-each (lambda (n) (stream 'put 'n) (? 'notes))))
      add-notes (lambda (str) (notes 'add-last str)
      status (lambda () (? 'status))
      set-status (lambda (st) (if (> st (? status)) (<- 'status st))))

(Gate2
   'extend-in 'Documentation
   'privmth '(
      list-structure (lambda (stream)
                        (stream 'put "input1:" ((? input1) 'name))
                        (stream 'put "input2:" ((? input2) 'name))
                        (stream 'put "out:" ((? output) 'name)))))

(AndGate
   'extend-in 'Documentation
   'privmth '(
        type-string (lambda () "a And: ")))

(NotGate
   'extend-in 'Documentation
   'privmth '(
      list-structure (lambda (stream)
                        (stream 'put "input:" ((? input) 'name))
                        (stream 'put "out:" ((? input) 'name)))))
      type-string (lambda () "a Not:")))

(NandGate
   'extend-in 'Documentation
   'privmth '(
        type-string (lambda () "a Nand: ")))

(OrGate
   'extend-in 'Documentation
   'privmth '(
        type-string (lambda () "a Or: ")))
```

```
(Pad
   'extend-in 'Documentation
   'publmth  (
      list-infos (lambda (stream) (stream 'put (? name))))


(Circuit
   'extend-in 'Documentation
   'attributes  (creation-date history author project version status)
   'privmth (
         record-add-gate
            (lambda (g) (history 'add-last (Time now)
                  (string-append "add-gate "(g 'as-string))
         record-add-wire
            (lambda (g) (history 'add-last (Time now)
                  (string-append "add-wire: " (w 'asString)))
         record-remove-gate
            (lambda (g) (history 'add-last (Time now)
                  (string-append "remove-gate: " (g 'asString))
         record-remove-wire
               (lambda (w) (history 'add-last (Time now)
                  (string-append "remove-wire: " (w 'asString)))))

   'publmth  (
      set-project (lambda (str) (<- project str)
      set-version (lamba (n)
                     (if (> n (? version)) (<- version n))
      add-notes (lambda (str) (history 'add (Time now) "notes :" str))
      notes-before (time) (mappend (lambda (cpl)
                                          (if (< (car cpl) time) (cadr cpl))
                                    (? 'history))
      notes-after (time) (mappend (lambda (cpl)
                                          (if (> (car cpl) time) (cadr cpl))
                                    (? 'history))

      reset-history (lambda () (<- 'history '())
      generate-netlist
            (lambda (stream)
               (self 'print-infos)
               (self 'print-history)
               (stream 'put "PadIns list:")
               (for-each (lambda (g) (g 'list-infos stream) (? 'padins))
               (stream 'put "PadOuts list:")
               (for-each (lambda (g) (g 'list-infos stream) (? 'padouts))
               (stream 'put "Gates list:")
               (for-each (lambda (g) (g 'list-infos stream) (? 'gates))
               (stream 'put "Wires list:")
               (for-each (lambda (w) (w 'list-infos stream) (? 'wires)))

      print-infos
            (lambda (stream)
               (stream 'put (? 'project))
               (stream 'put (? 'version))
               ...)

      print-history
            (lambda (stream)
               (for-each (lambda (cpl)
                              (stream 'put (car cpl) " : " (cadr cpl))
                        (? 'history)))
```

```
      add-gate-notes (lambda (sym str) ((self 'gate-named sym) 'add-notes str))
      revise-gate-named (lambda (sym) ((self 'gate-named sym) 'set-status 'revise)
      replace-gate-named (lambda (sym) ((self 'gate-named sym) 'set-status 'replace)
      status (lambda (sym) ((self 'gate-named sym) 'status)
))
```

Plan fonctionnel Normalisation

Réalisation des portes Non, Et, Ou à partir de portes NonEt.

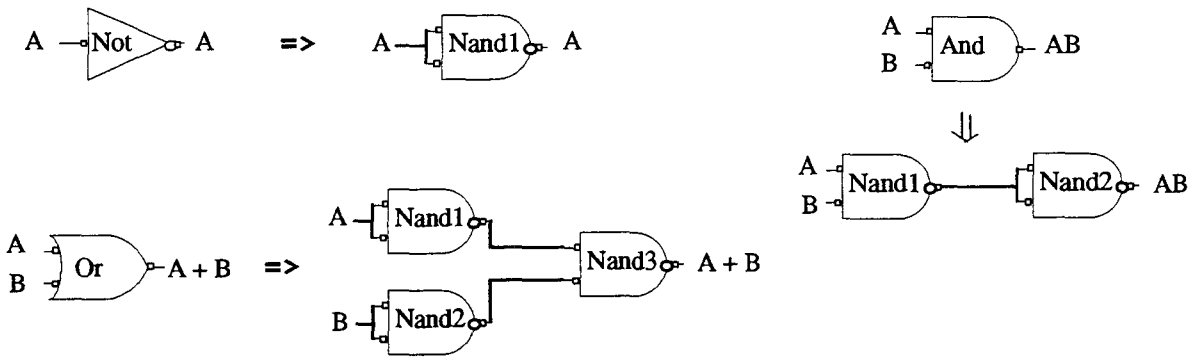


figure A.3

Explications sur la mise en oeuvre de la normalisation:

Le principe de mise en oeuvre repose sur une distribution du travail parmi les constituants du circuit (Pin, Wire, Pad, Gate). Initialement, un nouveau circuit est construit puis est transmis à l'ensemble des constituants, chacun s'occupant d'apporter sa contribution au circuit. En particulier, chaque porte du circuit construit sa forme normalisée puis vient l'ajouter dans le circuit résultat.

```
(Wire
   'extend-in 'Normalization
   'attributes  ()
   'privmth  ()
   'publmth  (
     normalize
        (lambda (circuit)
           ((? 'input) 'normalise circuit)
           (circuit add-wire (Wire 'new
                                   'output ((? 'output) 'get-mirror)
                                   'input ((? 'input) 'get-mirror)))
           (let ((dup ((? 'input) 'get-duplicate)))
              (when (not (null? dup))
                 (circuit add-wire (Wire 'new
                                         'output ((? 'output) 'get-mirror)
                                         'input dup)))))))

(Pin
   'extend-in 'Normalization
   'attributes  (mirror-pin)
   'privmth  ()
   'publmth  (
     normalize
        (lambda (circuit)
           (<- 'mirror-pin '()))
     is-normalised (lambda () (not (null? (? 'mirror-pin)))
     set-mirror (lambda (pin) (<- 'mirror-pin pin)
     get-mirror (lambda (pin) (? 'mirror-pin))))

(InputPin
```

```
    'extend-in 'Normalization

    'attributes  (duplicated-pin)
    'publmth  (
      normalize
         (lambda (circuit)
            (super)
            (<- 'duplicated-pin '())
            ((? 'gate) 'normalise circuit))
      set-duplicate (lambda (input) (<- 'duplicated-pin input)
      get-duplicate (lambda () (? 'duplicated-pin))))

(OutputPin
    'extend-in 'Normalization
    'publmth  (
      normalize
         (lambda (circuit)
            (super)
            (for-each (lambda (w) (w 'normalize circuit)) (? 'wires)))))

(CircuitEntity
    'extend-in 'Normalization
    'publmth  (
      normalize (lambda (circuit) #abstract)

 (AndGate
    'extend-in 'Normalization
    'attributes  (nand1 nand2)
    'publmth  (
      normalize
       (lambda (circuit)
         (when (or (null? nand1) (null? nand2))
         ;; creation des portes Nand équivalentes dans le nouveau circuit

            (<- 'nand1 (Nand 'new 'name (string (? 'name) "_nand1")))
            (<- 'nand2 (Nand 'new 'name (string (? 'name) "_nand2")))
            (circuit 'add-gate (? 'nand1))
            (circuit 'add-gate (? 'nand2))

         ;; duplication des équipotentielles pour un And
            (circuit 'add-wire (Wire 'new
                                  'output ((? 'nand1) 'pin-named 'output)
                                  'input ((? 'nand2) 'pin-named 'input1)))
            (circuit 'add-wire (Wire 'new
                                  'output ((? 'nand1) 'pin-named 'output)
                                  'input ((? 'nand2) 'pin-named 'input2)))

         ;; etablissement des liens entre les broches du And et de sa forme normalisé
            ((? 'input1) 'set-mirror ((? 'nand1) 'pin-named 'input1))
            ((? 'input2) 'set-mirror ((? 'nand1) 'pin-named 'input2))
            ((? 'output) 'set-mirror ((? 'nand2) 'pin-named 'output))

            ;; propagation de la normalisation
            (output 'normalise)))))

  (OrGate
    'extend-in 'Normalization
    'attributes  (nand1 nand2 nand3)
    'publmth  (
      normalise
         (lambda (circuit)
            (when (null? (? 'nand1))
            ;; creation des portes Nand equivalentes dans le nouveau circuit
```

```
            (<- 'nand1 (Nand 'new 'name (string (? 'name) "_nand1")))
            (<- 'nand2 (Nand 'new 'name (string (? 'name) "_nand2")))
            (<- 'nand3 (Nand 'new 'name (string (? 'name) "_nand3")))
            (circuit 'add-gate (? 'nand1))
            (circuit 'add-gate (? 'nand2))
            (circuit 'add-gate (? 'nand3))

         ;; creation des équipotentielles pour la forme normalisée

            (circuit 'add-wire (Wire 'new
                                     'output ((? 'nand1) 'pin-named 'output)
                                     'input ((? 'nand3) 'pin-named 'input1)))
            (circuit 'add-wire (Wire 'new
                                     'output ((? 'nand1) 'pin-named 'output)
                                     'input ((? 'nand3) 'pin-named 'input2)))

         ;; établissement des liens entre les broches du Or et de sa forme normalisée

            ((? 'input1) 'set-mirror ((? 'nand1) 'pin-named 'input1))
            ((? 'input1) 'set-duplicate ((? 'nand1) 'pin-named 'input2))
            ((? 'input2) 'set-mirror ((? 'nand2) 'pin-named 'input1))
            ((? 'input2) 'set-duplicate ((? 'nand2) 'pin-named 'input2))
            ((? 'output) 'set-mirror ((? 'nand3) 'pin-named 'output))

         ;; propagation de la normalisation au reste du circuit

            (output 'normalise)))))


(NorGate
  'extend-in 'Normalization
  'attributes  (nand1)
  'privmth  ()
  'publmth  (
    normalise
       (lambda (circuit)
          (when (null? nand1)
             ;; creation de la porte Nand equivalente dans le nouveau circuit

             (<- 'nand1 (Nand 'new 'name (string (? 'name) "_nand1")))
             (circuit 'add-gate (? 'nand1))

          ;; établissement des liens entre les broches du And et de sa forme normalisée

             ((? 'input) 'set-mirror ((? 'nand1) 'pin-named 'input1))
             ((? 'input) 'set-duplicate ((? 'nand1) 'pin-named 'input2))
             ((? 'output) 'set-mirror ((? 'nand1) 'pin-named 'output))

          ;; propagation de la normalisation
          (output 'normalise)))))

(NandGate
   'extend-in 'Normalization
   'attributes  (nand1)
   'privmth  ()
   'publmth  (
     normalise
        (lambda (circuit)
           (when (null? nand1)
        ;; creation de la porte Nand equivalente (simple clonage) dans le nouveau circuit

           (<- 'nand1 (Nand 'new 'name (string (? 'name) "_nand1")))
```

```
            (circuit 'add-gate (? 'nand1))

          ;; établissement des liens entre les broches du Nand et de sa forme normalisée

            ((? 'input1) 'set-mirror ((? 'nand1) 'pin-named 'input1))
            ((? 'input2) 'set-mirror ((? 'nand1) 'pin-named 'input2))
            ((? 'output) 'set-mirror ((? 'nand3) 'pin-named 'output))

            ;; propagation de la normalisation
            (output 'normalise)))))


    (PadIn
       'extend-in 'Normalization
       'publmth  (
         normalize
            (lambda (circuit)
               ;; clonage du PadIn dans le nouveau circuit

               (let ((pdi (PadIn 'new 'name (? 'name))))
                  ((? 'output) 'set-mirror (pdi 'pin-named 'output))
                  (circuit 'add-gate pdi)
                  ((? 'output) 'normalise)))))

    (PadOut
       'extend-in 'Normalization
       'publmth  (
         normalize
            (lambda (circuit)
               ;; clonage du PadIn dans le nouveau circuit

               (let ((pdo (PadOut 'new 'name (? 'name))))
                  ((? 'input) 'set-mirror (pdo 'pin-named 'input))
                  (circuit 'add-gate pdo)))))

    (Circuit
       'extend-in 'Normalization
       'publmth  (
         normalize
            (lambda ()
               (let ((nc (Circuit 'new 'name (string (? 'name) "_norm"))))
                  (for-each (lambda (ip) (ip 'normalise nc)) (? 'input-pins))))))
```

Lancement d'une normalisation

```
((dm1 in 'Normalization) 'normalize)
```

Plan fonctionnel Optimisation

L'un des soucis majeurs des concepteurs de circuits logiques consiste à en réduire le nombre de portes nécessaires à la réalisation, afin de minimiser le coût en nombre de portes et de connexions, la surface d'implantation, la consommation électrique, etc.

Minimiser la complexité d'un circuit revient en fait à en créer un autre équivalent, qui réalise la même fonction dans les mêmes conditions. Cette recherche d'équivalence est menée à bien en utilisant les lois de l'algèbre de Boole. Les lois considérées dans le cas de notre exemple sont données par le tableau suivant.

Table des optimisations prises en compte (lois de l'algèbre de boole)

Nom de la loi	Forme ET	Forme Ou
loi distributive	(A+B) (A+C) = A + BC	AB + AC = A (B+C)
loi d'absorption	A (A+B) = A	A + AB = A

Montrons sur un exemple l'application de la loi distributive pour une forme Ou. La figure suivante montre à droite le résultat obtenu. Ce résultat est fonctionnellement équivalent au schéma de gauche (ceci se vérifie aisémment en examinant les tables de vérités respectives) mais il contient un nombre de portes inférieur. A cet égard, on peut dire qu'il est meilleur.

figure A.4 : Exemple d'une optimisation

Explications sur la mise en oeuvre:

Une analyse du problème fait apparaitre que le processus d'optimisation procède en examinant des groupes de portes et que le choix de l'optimisation dépend en général du type et des entrées des portes contenu dans ces groupes.

Comme pour la normalisation, nous distribuons la réalisation de l'optimisation entre les différentes portes du circuits. Le principe de base est de laisser chaque porte du circuit déterminer si une optimisation est possible. Pour cela, chaque porte connait un certain nombre de règle correspondant aux différents cas d'optimisation possibles à partir de son type.

Le processus se décompose en deux phases.

La première phase consiste à parcourir la structure du circuit de façon à configurer les liens

de précédences entre les portes et analyser les optimisations possibles. Cette configuration est réalisée par les envois de messages `previous-gate` entre les constituants du circuit. Ces envois de messages servent notamment à transmettre à chaque porte les références des portes reliées à ces entrées.

Lorsqu'à un moment du parcours une porte connait toutes les portes qui sont réliées à ces entrées, celle-ci teste si des optimisations sont possibles. A cet fin, elle interroge ces dernières à la fois sur leur type, sur leurs entrées et leurs optimisations. Si une éventuelle optimisation est détectée par la porte concernée, celle-ci mémorise son type pour la seconde phase puis fait en sorte de poursuivre le parcours du reste de la structure.

La seconde phase correspond à la génération du circuit optimisé. Le circuit initial commence par créer le circuit résultat sans lui associer de composants. Le choix de ces derniers est délégué aux différents constituants. Ceci conduit à réaliser un second parcours de la structure au cours duquel chaque élément s'occupe d'ajouter ce qu'il faut au nouveau circuit. Ainsi, dans le cas où un groupe de porte peut être simplifié, c'est la porte ayant identifiée une optimisation possible lors de la première phase qui se charge de créer et relier l'ensemble des portes équivalentes à ce groupe au sein du circuit optimisé.

Pour simplifier, nous considérons uniquement les optimisations possibles pour la porte Et (AndGate).

```
(Wire
   'extend-in 'Optimization
   'publmth  (
     optimize-into (lambda (circuit) (dest 'optimize-into circuit))
     new-wire (lambda (circuit op)
              (circuit add-wire (Wire 'new
                                      'output op
                                      'input ((? 'input) 'get-mirror))))
     previous-gate (lambda (pg) (dest 'previous-gate pg))

(Pin
   'extend-in 'Optimization
   'publmth  (
     previous-gate (lambda (pg) #abstract)))

(InputPin
   'extend-in 'Optimization
   'publmth  (
     previous-gate (lambda (pg) ((? 'gate) 'previous-gate pg))))

 (OutputPin
   'extend-in 'Optimization
   'attributes  ()
   'privmth  ()
   'publmth  (
     optimize-into (lambda (circuit)
                     (for-each (lambda (w)
                        (w 'optimize-into circuit)) (? 'wires)))
     new-wire (lambda (circuit input)
              (for-each (lambda (w) (w 'new-wire circuit input)) (? 'wires)))
     previous-gate
        (lambda (pg)
           (for-each (lambda (w) (w 'previous-gate pg)) (? 'wires)))))

(CircuitEntity
```

```
   'extend-in 'Optimization
   'publmth  (
      isOrGate (lambda () #false)
      isAndGate (lambda () #false)
      previous-gate (lambda (gate) #abstract)
      optimize-into (lambda (circuit) #abstract)
      can-be-optimized (lambda () #false)

(Gate
   'extend-in 'Optimization
   'attributes  ()
   'privmth  ()
   'publmth  (
      is-linked-to (lambda (gate) #abstract)
      has-same-input (lambda (gate) #abstract)
      get-same-input (lambda (gate) #abstract)
      get-diff-input (lambda (gate) #abstract)))


(And
   'extend-in 'Optimization
   'attributes  (previous1 previous2 optimization-type)
   'privmth  (
      check-optimization
         (lambda ()
            (cond ((self 'check-distribution) (<- 'optimization-type 1))
                  ((self 'check-absorption1) (<- 'optimization-type 2))
                  ((self 'check-absorption2) (<- 'optimization-type 3))
                  (else (<- 'optimization-type 0))))

      check-distribution
         (lambda () (and (not ((? 'previous1) 'can-be-optimized)
                         (not ((? 'previous2) 'can-be-optimized)
                         ((? 'previous1) 'isOrGate) ((? 'previous2) 'isOrGate)
                         ((? 'previous1) 'has-same-input (? 'previous2)))
      check-absorption1
         (lambda () (and (not ((? 'previous1) 'can-be-optimized)
                         ((? 'previous1) 'isOrGate)
                         (self 'has-same-input (? 'previous1))))
      check-absorption2
         (lambda () (and (not ((? 'previous2) 'can-be-optimized)
                         ((? 'previous2) 'isOrGate)
                         (self 'has-same-input (? 'previous2))))

      apply-absorption
         (lambda (circuit)
          ;; creation des equipotentielles
          (let ((si (if (= (? 'optimization-type) 2)
                        (self 'get-same-input (? 'previous1))
                        (self 'get-same-input (? 'previous2)))))

            ;; creations des équipotentielle en deleguant a la broche de sortie
            (output 'new-wire circuit si)))

      remove-distribution
         (lambda (circuit)
         ;; creation et ajout de la porte Or et And
          (let ((or1 (Or 'new 'name (string (? 'name) "_dist"))
                (and1 (And 'new 'name (string (? 'name) "_dist")))

            (circuit 'add-gate (? 'or1))
            (circuit 'add-gate (? 'and1))
```

```
      ;; creation de l'équipotentielle entre le and et le or
         (circuit 'add-wire (Wire 'new
                                  'output (and1 'pin-named 'output)
                                  'input (or1 'pin-named 'input2)))

      ;; ajout d'une equipotentielle lié avec input1 du Or
         (circuit 'add-wire (Wire 'new
                                 'output ((? 'previous1) 'get-same-input (? 'previous2))
                                 'input (or 'pin-named 'input1)))

      ;; ajout des equipotentielle lié avec input1 et input2 du And
         (circuit 'add-wire (Wire 'new
                                 'output ((? 'previous1) 'get-diff-input (? 'previous2))
                                 'input (and1 'pin-named 'input2)))
         (circuit 'add-wire (Wire 'new
                                 'output ((? 'previous2) 'get-diff-input (? 'previous1))
                                 'input (and1 'pin-named 'input2)))


   'publmth  (
      is-linked-to (lambda (gate) (or (eq? (? 'previous1) gate) ((eq? 'previous2) gate))
      has-same-input (lambda (gate) (null? (gate 'get-same-input self)))
      get-same-input
            (lambda (gate)
               (cond ((gate 'is-linked-to (? 'previous1))
                      (? 'previous1))
                     ((gate 'is-linked-to (? 'previous2))
                      (? 'previous2))
                     (else nil))
      get-diff-input (lambda (gate)
                       (let ((si (gate 'get-same-input))
                             (cond ((null? si) nil)
                                   ((not (eq? si (? 'previous1))
                                    (? 'previous1)
                                   (else (? 'previous2)))
      isAndGate (lambda () #true)
      can-be-optimized (lambda () (= (? 'optimization-type) 0))
      previous-gate
         (lambda (gate)
            (cond ((null? (? 'previous1))
                   (<- 'previous1 gate))
                  ((null? (? 'previous2))
                   (<- 'previous2 gate)
                   (self 'check-optimization)
                   (output 'previous-gate self))))

      optimize-into
         (lambda (circuit)
            (when (self 'can-be-optimized)
               (cond ((= (? 'optimization-type) 1)
                      (self 'remove-distribution circuit))
                     ((or (= (? 'optimization-type) 2) (= (? 'optimization-type) 3))
                      (self 'apply-distribution circuit))))
            (output 'optimize-into circuit))


(PadIn
   'extend-in 'Optimization
   'publmth  (
      previous-gate (lambda (gate) (output 'previous-gate self)
      optimize-into
         (lambda (circuit)
            (let ((pdi (PadIn 'new 'name (? 'name))))
```

```
            (circuit 'add-gate pdi)
            ((? 'output) 'optimize-into circuit)))))

(PadOut
   'extend-in 'Optimization
   'privmth  (previous)
   'publmth  (
     previous-gate (lambda (gate) (<- 'previous gate))
     optimize-into
        (lambda (circuit)
          (let ((pdo (PadOut 'new 'name (? 'name))))
            (circuit 'add-gate pdo)
            (circuit 'add-wire (Wire 'new
                                'output ((? 'previous-gate) 'pin-named 'output)
                                'input (self 'pin-named 'input)))))))

(Circuit
   'extend-in 'Optimization
   'privmth (
     prepare-optimization
        (lambda ()
          (for-each (lambda (pi) (pi 'previous-gate nil)) (? 'padins))))
     optimize-into
        (lambda (circuit)
          (for-each (lambda (ip) (pi 'optimize-into circuit) (? 'padins)))))

   'publmth  (
     optimize
        (lambda ()
         ;; l'optimization est decomposée en deux phases:
         ;; la première pour établir les liaisons appropriées entre les portes du circuit
         ;; la seconde pour réaliser l'optimisation dans le nouveau circuit

           (self 'prepare-optimization)
           (if (null? (detect (lambda () (gate 'can-be-optimized) (? gates))))
              self       ;; self si aucune optimization possible

         ;; Creation du circuit resultat puis lancement de l'optimisation
              (let ((nc (Circuit 'new 'name (string (? 'name) "_optmz"))))
                 (self 'optimize-into nc)
                 (nc 'optimize))) ;; on relance une optimisation sur le circuit obtenu
                                  ;; de façon à optimiser le plus possible
```

Lancement d'une optimisation

```
((dm1 in 'Optimization) 'optimize)
```

8

Bibliographie

[Albano 93] Albano A., Begamini R., Ghelli G. and Orsini R., *An Object Data Model with Roles*, in VLDB Conference, Dublin, Ireland, pp. 39-51, August 1993

[Amiel 95] Amiel E. et Dujardin E., *Un outil d'aide à la résolution explicite des ambiguïtés des multi-méthodes*, in Langages et Modèles à Objets 95, Nancy , pp. 131-152, Octobre 1995

[Andersen 92] Andersen E.P and Reenskaug T., *System Design by Composing Structures of Interacting Objects*, in Proceedings of ECOOP'92 , LCNS 615, Springer Verlag, Utrecht, The Netherlands, pp. 133-152, June 1992

[Bardou 96] Bardou D. and Dony C., *Split Objects: a Disciplined use of Delegation within Objects*, in Proceedings of OOPSLA'96, ACM SIGPLAN, San Jose, USA, pp. 122-137, October 1996

[Bardou 98] Bardou D, *Étude de langages à prototypes, du mécanisme de délégation et de son rapport à la notion de point de vue*, Thèse de Doctorat en Informatique, LIRMM, Université de Montpellier 2, Avril 1998

[Bertino 95] Bertino E. and Guerrini G., *Objects with Multiple Most Specific Classes*, in Proceedings of ECOOP'95, Springer-Verlag, Aarhus, Denmark, pp. 102-126, June 1995

[Blake 87] Blake E. and Cook S., *On Including Part Hierarchies in Object-Oriented Languages*, with an Implementation in Smalltalk, in Proceedings of ECOOP'87, Paris, France, July 1987

[Bobrow 77] Bobrow D.and Winograd T., *An Overview of KRL, a Knowledge Representation Language*, Cognitive Science, Vol. 1, 1997

[Booch 91] Booch G., *Object-Oriented Design with Applications*, New York NY: Benjamin/ Cummings, 1991

[Borning 82] Borning A. and Ingalls D., *Multiple Inheritance in Smalltalk-80*, in Proceedings of AAAI'82, Pittsburgh, Pennsylvania, pp. 233-237, August 1982

[Bracha 90] Bracha G. and Cook W., *Mixin-based Inheritance*, in Proceedings of ECOOP/ OOPSLA'90 Joint Conference, ACM SIGPLAN, Ottawa, Canada, pp. 303-311, November 1990

[Briot 89] Briot J.P. and Cointe P., *Programming with Explicit Metaclasses in Smalltalk-80*, in Proceedings of OOPSLA'89, New-Orleans, USA, pp. 419-431, November 1989

[Carn 92] Carn N., *Représentation orientée objet de système opérationnel avec application au domaine spatial*, in Thèse de doctorat en informatique, I.N.P., Toulouse, 1992

[Carré 89] Carré B., *Méthodologie orientée objet pour la représentation des connaissances,*

concept de point de vue, de représentation multiple et évolutive d'objet, Thèse de Doctorat en Informatique, LIFL, Université de Lille, Janvier 1989

[Carré 90a] Carré B. and Geib J.M., *The Point of View notion for Multiple Inheritance*, in Proceedings of ECOOP/OOPSLA'90 Joint Conference, ACM SIGPLAN, Ottawa, Canada pp. 312-321, October 1990

[Carré 90b] Carré B., Dekker L. and Geib J.M., *Multiple and Evolutive Representation in the ROME Language*, in Proc. of the Second International Conference TOOLS, Paris, 1990

[Carré 91] Carré B. and Dekker L., *Inheriting Object-Oriented Features through Meta-Programming, A Frame Extension to ROME*, in Proc. of the East European Conference on Object-Oriented Programming, Bratislava, Czecho-Slovakia, 1991

[Carré 96] Carré B. et Vanwormhoudt G. , *Conception modulaire en CROME. Le cas de l'interfaçage graphique d'objets*, in Interface-Homme-Machine'96, http://www-ihm96.imag.fr, Octobre 1996

[Chambers 91] Chambers C., Ungar D., Chang B-W. and Hölzle U., *Parents are shared parts of objects: inheritance and encapsulation in Self*, in Lisp and Symbolic Computation, Vol 4. No 3., Kluwer Academic Publishers, pp. 21-36, June 1991

[Chambers 93] Chambers G., *Predicate Classes*, in Proceedings of ECOOP'93, pp. 268-296, July 1993

[Chang 95] Chang B.W., Ungar D. and Smith R.B., *Getting Close to Objects: Object-Focused Programming Environments*, in Visual Object Oriented Programming , Burnett M., Goldberg A. and Lewis T. (Eds), Prentice-Hall, pp. 185-198, 1995

[Charlet 98] Charlet V. et Tarrade V., *Conception en CROME d'un atelier de CAO*, Rapport de Stage, Ecole Universitaire d'Ingénieurs de Lille, Mars 98

[Chiba 93] Chiba S. and Masuda T., *Designing an Extensible Distributed Language with a Meta-Level Architecture*, in Proceedings of ECOOP'95, pp. 482-501, July 93

[Cointe 87] Cointe P., *Metaclasses are First Class: the ObjVlisp Model*, in Proceedings of OOPSPLA'87 , Orlando, Florida, pp. 156-167, 1987

[Cointe 90] Cointe P., *The ClassTalk System: A Laboratory to study reflection in Smalltalk*, Reflection and Metalevel Architectures in Object-Oriented Programming, ECOOP/OOPSLA'90 Workshop, October 1990

[Cointe 92] Cointe P., Malenfant J, Dony C. et Mulet P., *Etude de la réflexion de comportement dans le langage SELF*, dans Actes de la 1ère Conférence sur la Représentation par Objets , La Grande Motte, EC2 Ed, Juin 1992

[Czarnecki 99] Czarnecki K. and Eisenecker U., *Generative Programming: Methods, Techniques and Applications*, Addison-Wesley, 1999.

[Debrauwer 97] Debrauwer L., Vanwormhoudt G. et Carré B., *Un cadre de conception par con-*

textes fonctionnels de systèmes d'information à objets, dans XVème Congrès INFORSID, Toulouse, Juin 1997

[Debrauwer 98] Debrauwer L., *Des vues aux contextes pour la structuration fonctionnelle de bases de données à objets en CROME*, Thèse de Doctorat en Informatique, LIFL, Université de Lille, Decembre 1998

[Deconninck 92] Deconninck A. S., *Modélisation par objets de systèmes complexes dans le cadre d'applications scientifiques spatiales. Introduction de la notion de version dans un modèle objet multi-vue*, in Thèse de doctorat en informatique, Toulouse, I.N.P, Octobre 1992

[Decouchant94] D. Decouchant. *RétroAction de Groupe et Édition coopérative de Documents Structurés*, IHM'94, Lille, Décembre 1994, pp 145-150

[Dekker 92] Dekker L. et Carré B., *Multiple and Dynamic Representation of frames with Points of View in FROME*, dans Actes de la 1ère Conférence sur la Représentation par Objets , La Grande Motte, EC2 (Eds), Juin 1992

[Dekker 94] Dekker L., *FROME: Représentation multiple et classification d'objets avec points de vue*, Thèse de Doctorat en Informatique, LIFL, Université de Lille, Juin 94

[Dony 92] Dony C. , Malenfant J. and Cointe P. , *Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation*, in Proceedings of OOPSLA'92 , ACM SIGPLAN, Vancouver, Canada, pp. 201-217, November 1992

[Ducasse 95] Ducasse S, Blay-Fornarino M, and Pinna-Dery A.M, *A Reflective Model for First Class Dependencies*, in Proceedings of OOPSLA'95, ACM SIGPLAN, Austin, pp. 265-280, October 95.

[Ducourneau 89] Ducourneau R. et Habib M., *La multiplicité de l'héritage dans les langages à objets*, in Techniques et Sciences Informatiques, Vol.8, No. 1, AFCET-Bordas, 1989

[Ducournau 95] Ducournau R., Habib M., Huchard M., Mugnier M.M., Napoli A., *Le point sur l'héritage multiple*, in Techniques et Sciences Informatique 14(3), 1995

[Ferber 88] Ferber J., *Coreferentiality: the Key to an Intentional Theory of Object-Oriented Knowledge Representation*, in Artificial Intelligence and Cognitive Sciences, Manchester University Press, 1988

[Ferber 89a] Ferber J., *Objets et agents: une étude de représentation et de communications en Intelligence Artificielle*, Thése d'Etat, Paris VI, 1989

[Ferber 89b] Ferber J., *Computational Reflection in Class based Object Oriented Languages* , in in Proceedings of OOPSLA'89, New-Orleans, USA , pp. 317-326, 1989

[Ferber 91] Ferber J., *Conception et Programmation par objets*, Hermes, 1991

[Gamma 94] Gamma E., Helm R., Johnson R. and Vlissides J., *Design Patterns: Catalogue de modèles de Conception Réutilisables*, International Thomson Publishing, 1994

[Gibbs 90] Gibbs S., Tsichritzis D, Casais E., Nierstrasz O and Pintado X., *Class management for software communities*, in Communications of the ACM, Vol 33. No 9., pp. 90-103, September 1990

[Golberg 83] Golberg A. and Robson D., *Smalltalk-80, the Language and its Implementation*, Massachusets, Addison-Wesley, 1983

[Goldstein 80] Goldstein I.P. and Bobrow D. G., *Extending Object Oriented Programming in Smalltalk*, in Proceeding of the Lisp Conference, Stanford University, 1980

[Gottlob 96] Gottlob G., Schrefl M. and Roeck B., *Extending object-oriented systems with roles*, in ACM Transactions on Information Systems, vol. 14 n. 3, pp. 268-296, 1996

[Graube 89] Graube N., *Metaclass Compatibility*, in Proceedings of OOPSLA'89, ACM Press SIGPLAN, New-Orleans, USA, pp. 305-315, 1989

[Gruber92] T.R. Gruber, J.M Tenenbaum and J.C. Weber. *Toward a Knowledge Medium for Collaborative Product Development.*, Artificial Intelligence in Design'92, Proccedings of the Second International Conference on Artificial Intelligence in Design; Pittsburgh, Kluwer Academic Publishers, USA, June 1992, pages 413-432

[Hailpern 90] Hailpern B. and Ossher H., *Extending Object to Support Multiple Interface and Access Control*, in IEEE Transactions on Software Engineering, Vol.16 , No.11, pp. 1247-1257, November 1990

[Halbert 87] Halbert D.C. and O'Brien P.D. , *Using Types and Inheritance in Object-Oriented Languages*, in Proceedings of ECOOP'87, Paris, France, pp. 23-34, 1987

[Harrison 92] Harrison W., Kavianpour M. and Ossher H., *Integrating Coarse-grained and fine-grained Tool Integration,* in Proceedingss of 5th International Workshop on Computer-Aided Software Engineering, 1992

[Harrison 93] Harrison W. and Ossher H., *Subject-Oriented Programming (A Critique of Pure Objects)*, in Proceedings of OOPSLA'93, ACM Press SIGPLAN, Washington, USA, pp. 411-428, October 1993

[Hauck 93] Hauck F., *Inheritance Modeled with Explicit Bindings: An Approach to Typed inheritance*, in Proceedings of OOPSLA'93, ACM Press SIGPLAN, Washington, USA, pp. 231-239, 1993

[Hedin 88] Hedin G. and Magnusson B., *The Mjolner environment: direct interaction with abstractions*, in Proceedings of ECOOP'88, pp. 41-54, 1988

[Helm 90] Helm R., Holland I. and Gangopadhyay D., *Contracts: Specifying Behavioural Composition in Object-Oriented Systems*, in Proceedings of ECOOP/OOPSLA'90 Joint Conference, ACM Sigplan, Ottawa, Canada, pp. 168-180, 1990

[Hendler 86] Hendler J., *Enhancement for multiple-inheritance*, in Proceedings of OOPSLA'86, ACM Press SIGPLAN, Portland, USA, pp. 98-106, 1986

[Hollant 90] Hollant T., *Etude d'une approche objet pour la CAO Electronique: Application à la Simulation Logique*, Thèse de Doctorat en Informatique, LIFL, Université de Lille, Mars 1990

[Jalote 89] Jalote P., *Functional Refinement and Nested Objects for Object-Oriented Design*, in IEEE Transactions on Software Engineering, Vol. 15. No 3., pp. 264-270, Mars 1989

[Johnson 88] Johnson R.E and Foote B., *Designing reusable classes*, in Journal of Object-Oriented Programming, Vol 1. No 2., pp. 22-35, June 1988

[Kent 91] Kent W., *A rigorous model of object reference, identity, and existence*, in Journal of Object-Oriented Programming, Vol 4. No. 3, pp. 28-36, June 1991

[Khoshafian 86] Khoshafian S.N. and Copeland G.P., *Object Identity*, in Proceedings of OOPSLA'86, ACM Press SIGPLAN, Portland, USA, pp. 406-416, 1986

[Kiczales 91] Kiczales G., Des Rivières J. and Bobrow D.G , *The Art of the Metaobject Protocol*, MIT Press, 1991

[Kiczales 92] Kiczales G., *Towards a New Model of Abstraction in the Engineering of Software*, in Proceedings of IMSA'92 Conference, 1992

[Kiczales 97] Kiczales G., Lamping J., Mendhekar A., Maeda C, Lopes C.V., Loingthier J.M. and Irwin J., *Aspect-Oriented Programming*, in Proceedings of ECOOP'97, LNCS 1241, Springer-Verlag, Jyvaskyla, Finland, pp. 279-296, 1997

[Kim 88] *Object-Oriented concepts, applications and databases*, Kim W. and Lochowsky F. (eds), Addison-Wesley, 1988

[Krief 91] Krief P., *Utilisation des langages à objets pour le prototypage*, Masson, 1991

[Kristensen 93] Kristensen B.B., *Transverse Activities: Abstractions in Object-Oriented Programming*, in Proceedings of International Symposium on Object Technologies for Advanced Software (ISOTAS'93), Springer-Verlag, Japan, 1993

[Kristensen 96] Kristensen B.B and Osterbye K., *Roles: Conceptual Abstraction. Theory and Practical Lanaguage Issues*, in a Special Issue of Theory and Practice of Object Sytstems (TAPOS) on Subjectivity in Object-Oriented Systems, 1996

[Larver 94] Larvet P., *Analyse des systèmes: de l'approche fonctionnelle à l'approche objet*, Inter-Editions, 1994

[Lieberman 86] Lieberman H., *Using Prototypical Objets to Implement Shared Behavior in Object Oriented Systems*, in Proceedings of OOPSLA'86, ACM Press SIGPLAN, Portland, USA, pp. 214-223, 1986

[MacAffer 95] MacAffer J., *MetaLevel Programming with CodA*, in Proceedings of ECOOP'95, Springer-Verlag, Aarhus, Denmark, 1995

[Madsen 92] Madsen O.L. and Moller-Pedersen B., *Part Objects and their Location, in Tech-*

nology of Object-Oriented Languages and Systems, in Proceedings of TOOLS 7, Dortmund, Englewood Cliffs NJ: Prentice-Hall, 1992

[Madsen 89] Madsen O.L. and Moller-Pedersen B., *Virtual Classes: a powerful mechanism in object-oriented programming*, in Proceedings of OOPSLA'89, ACM Press SIGPLAN, New-Orleans, USA, pp. 397-406, 1989

[Maes 87] Maes P., *Computational Reflection*, in VUB AI-Lab TR 87-2, Ph. D. thesis, V.U, Brussels, 1987

[Malenfant 96] Malenfant J., *Abstraction et encapsulation en programmation par prototypes*, in Technique et Science Informatiques, Vol. 15 No. 6, Hermes, pp. 709-733, 1996

[Marcaillou 93] Marcaillou S., Coulette B. and Vo D.P., *An Approach to View Point Modelling*, in Proc. of TOOLS Europe'93, pp. 151-164, 1993

[Marino 93] Marino O., *Raisonnement classificatoire dans une représentation à objets multi-points de vue*, Thèse de Doctorat en Informatique, Université Joseph Fourier, Grenoble, Octobre 1993.

[Masini 89] Masini G., Napoli A., Colnet D., Léonard D., et Tombre K., *Les langages à objets*, InterEditions, 1989

[Meyer 88] Meyer B. , *Object-Oriented Software Construction*, Englewood Cliffs NJ: Prentice Hall, 1988

[Mens 97] Mens K, Lopes C.V., Tekinerdogan B. and Kiczales G., *Aspect-Oriented Programming Workshop Report*, in ECOOP'97 Workshops Reader, LNCS 1357, Springer Verlag, pp. 481-494, June 1997.

[Meyer 87] Meyer B., *Reusability: The Case for Object-Oriented Design*, in IEEE Software 4(2), pp. 50-64, Mars 1987

[Meyer 92] Meyer B., *EIFFEL - The Language*, Prentice Hall, Object-Oriented Series, 1992

[Mezini 97] Mezini M., *Dynamic Object Evolution without Name Collisions*, in Proceedings of ECOOP'97, LNCS 1241, Springer-Verlag, Jyvaskyla, Finland, pp. 190-219, 1997

[Mulet 93] Mulet Ph. et Cointe P., *Définition d'un noyau réflexif pour un language à prototypes*, dans Actes de la 2ème Conférence sur la Représentation par Objets, La Grande Motte, pp. 101-116, Juin 1993

[Mulet 94] Mulet Ph. et Marco J., *De la parenté entre les environnements de MitScheme et les prototypes de Self*, dans Langage et Modèle à Objets 94, Grenoble, France, Octobre 1994

[Naja 95] Naja H., Lahlou Y. et Mouaddib N., *Un modèle pour la représentation multiple dans les bases de données orientés objet*, dans Langage et Modèle à Objets 95, Nancy, France, 1995.

[Naja 97] Naja H., *La représentation multiple d'objets pour l'ingénierie*, L'Objet, Vol. 4 No. 2, Hermes, 1997

[Nguyen 92] Nguyen G.T, Rieu D. and Escamilla J., *An Object Model for Engineering Design*, in Proceedings of ECOOP'92 , LCNS 615, Springer Verlag, Utrecht, The Netherlands, LCNS 615, Springer Verlag, Utrecht, The Netherlands

[Nierstraz 88] Nierstraz O.M., *A survey of object-oriented concepts*, in *Object-Oriented concepts, applications and databases*, Kim W. and Lochowsky F. (eds), Addison-Wesley, , pp. 3-21, 1988

[Ossher 92] Ossher H. and Harrison W., *Combination of Inheritance Hierarchies*, in Proceedings of OOPSLA'92 , ACM Sigplan, Vancouver, Canada, pp. 25-40, August 1992

[Ossher 94] Ossher H., Harrison W., Budinski F. and Simmonds I., *Subject-Oriented Programming : Supporting Decentralized Development of Objects*, in Proceedings of the 7th IBM Conference on O-O- Technology, Santa-Clara, pp. 337-349, July 1992

[Ossher 95] Ossher H., Kaplan M. Harrison W., Katz A. and Kruskal V., *Subject-Oriented Composition Rules*, in Proceedings of OOPSLA'95, ACM Sigplan, Austin, pp 235-250, 1995

[Ossher 98] Ossher H. and Tarr P., *Using Subject-Oriented Programming to Overcome Common Problems in O-O Software Development and Evolution*, ECOOP'98 Tutorial, Brussels, Belgium, July 1998

[Osterbye 95] Osterbye K. and Kristensen B.B., *Roles*, Technical Report R-95-2006, Department of Mathematics and Conputer Science, Aalborg University, Denmark, March 1995

[Paepcke 93] Paepcke A., *Object-Oriented Programming: the CLOS Perspective*, MIT Press, Cambridge, Massachusetts, 1993

[Parcplace 94] ParcPlace Systems, *VisualWorks Release 2.0,* Sunnyvale, California, USA, 1994

[Pernici 90] Pernici B., *Objects with Roles*, in Proc. of the IEEE/ACM Conference on Office Information Systems, Cambridge, Mass., USA, pp. 205-215, 1990

[Perrot 92] Perrot J.F. et Wolinski F., *Modélisation par objets en robotique*, in Technique et Science Informatiques, Vol. 11 No. 1, pp. 97-115, 1992

[Pintado 95] Pintado X., *Gluons and the Cooperation between Software Components*, in Object-Oriented Software Composition, Nierstraz O. and Tsichritzis (Eds), Prentice-Hall, 1995

[Prieto 97] Prieto M. and Victory P., *Subject Object Behaviour*, in Object Expert, SIGS Publication, pp. 24-25, Mars 1997

[Richardson 91] Richardson J. and Schwarz P., *Aspects: Extending Objects to Support Multiple, Independant Roles*, in ACM-SIGMOD International Conference on Management of Data, Denver, Colorado, ACM. Sigmod Record, Vol. 20., pp. 298-307, May 1991

[Riel 94] Riel A.J., *Object Oriented Design Heuristics*, Addison-Wesley, 1994

[Rivard 97] Rivard F., *Évolution du comportement des objets dans les langages à classes ré-*

flexifs, Thèse de Doctorat en Informatique, Université de Nantes, Juin 1997

[Sakkinen 89] Sakkinen M., *Disciplined inheritance*, in Proceedings of ECOOP'89, Cambridge, England, pp. 39-56, 1989

[Sciore 89] Sciore E., *Object Specialization*, in ACM Transactions on Information Systems, Vol. 7, No. 2, pp. 103-122, April 1989

[Seiter 96] Seiter L., Palsberg J. and Lieberherr K., *Evolution of object behavior using context relations*, in Proceedings of the 4th ACM SIGSOFT Symposium on Fondation of Software Engineering, Software Engineering Notes, Vol. 21 No. 6, ACM Press, pp. 45-56, 1996

[Shilling 89] Shilling S. and Sweeney P.F, *Three steps to views: extending the object-oriented paradigm*, in Proceedings of OOPSLA'89, New-Orleans, USA, pp. 353-361, 1989

[Smith 96] Smith R.B. and Ungar D., *A Simple and Unifying Approach to Subjective Objects*, in a Special Issue of Theory and Practice of Object Sytstems (TAPOS) on Subjectivity in Object-Oriented Systems, 1996

[Smaragdakis 98] Smaragdaki Y. and Batory D., *Implementing Layered Designs with Mixin-Layers*, in Proceedings of ECOOP'98, Springer-Verlag, Brussels, Belgium, July 1998

[Snyder 87] Snyder A., *Inheritance and the Development of Encapsulated Software Components*, in Research and Direction in Object-Oriented Languages, MIT Press, pp. 165-188, 1987

[Snyder 93] Snyder A., *The essence of objects: concepts and terms*, in IEEE Software, Vol 10. No.1, pp. 31-42, January 1993

[Snyder 91] Snyder A., *Inheritance in Object-Oriented Programming Languages, in Inheritance Hierarchies*, in Knowledge Representation and Programming Languages, Lenzerini M., Nardi D. and Simi M. (Eds), John Wiley & Sons, pp. 153-171, 1991

[Sommerville 91] Sommerville I., *Le Génie Logiciel*, Addison-Wesley, France

[Stata 95] Stata R. and Guttag J. V., *Modular Reasoning in the Presence of Subclassing*, in Proceedings of OOPSLA'95, ACM Sigplan, Austin, USA, pp. 200-214, October 1995

[Stata 97] Stata R., *Modularity in the Presence of Subclassing*, PhD Thesis, Technical Report 711, Laboratory for Computer Science, Massachusetts Institute of Technology, April 1997

[Stefik 86] Stefik M. and Bobrow D., *Object-oriented programming: themes and variations, in AI Magazine*, Vol 6. No 4., pp. 40-62, 1986

[Stein 88] Stein L., Lieberman H. and Ungar D., *A Shared View of Sharing: The Treaty of Orlando*, in Object-Oriented concepts, applications and databases, Kim W. and Lochowsky F. (eds), Addison-Wesley , pp. 31-48, 1988

[Taivalsaari 93] Taivalsaari A., *Object-Oriented programming with modes*, in Journal of Object-Oriented Programming, Vol 6. No 3., pp. 25-32, 1993

[Trousse 97] Trousse B., *Objets et CAO*, dans Ingénierie Objet - Concepts et techniques, Oussalah Ch. (Eds), InterEditions, pp. 153-171, 1997

[Ungar 87] Ungar D. and Smith R.B., *Self: The Power of Simplicity*, in Proceedings of OOPSLA'87 , Orlando, Florida, pp. 227-242, 1987

[VanderMeulen 87] VanderMeulen P.S., *INSIST : Interactive Simulation in Smalltalk*, in Proceedings of OOPSLA'87 , Orlando, Florida, pp. 366-376, 1987

[Van Limberghen 96] Van Limberghen M. and Mens T., *Encapsulation and composition as orthogonal operations on mixins: A solution to multiple inheritance problem*, in Object-Oriented Systems, Vol. 3 No.1, Chapman&Hall, 1996

[VanHilst 96] VanHilst M. and Notkin D., *Using Role Components to Implement Collaboration-Based Designs*, in Proceedings of OOPSLA'96, ACM Sigplan, San Jose, USA, October 1996

[Vanwormhoudt 97a] Vanwormhoudt G., Carré B. et Debrauwer L., *Programmation par objets et contextes fonctionnels. Application de CROME à Smalltalk*, dans Langages et Modèles à Objets 97, Roscoff, France, Octobre 97.

[Vanwormhoudt 97b] Vanwormhoudt G., *Programmation par contextes en Smalltalk.*, L'Objet, Hermes, Vol 3. N.4, Decembre 97.

[Wegner 90] Wegner P., *Concepts and Paradigms of Object-Oriented Programming*, in OOPS Messenger, Vol 1. No 1, ACM Press, August 1990

[Wierenga 94] Wierenga R., de Jonge W. and Spruit P., *Roles and dynamic subclasses: a modal logic approach*, in Proceedings of ECOOP'94, pp. 32-59, July 1994

[Wirfs-Brock 88] Wirfs-Brock R.J and Wilkerson B., *An overview of modular Smalltalk*, in Proceedings of OOPSLA'89, ACM Press SIGPLAN, New-Orleans, USA, pp. 123-134, 1988

[Wirfs-Brock 89] Wirfs-Brock R.J. and Wilkerson B., *Object-oriented design: a responsibility-driven approach*, in Proceedings of OOPSLA'89, ACM Press SIGPLAN, New-Orleans, USA, 1989

[Wolinski 90] Wolinski F., *Etude des capacités de modélisation systèmique des langages à objets appliquées à la représentation de robots*, in Thèse de Doctorat en Informatique, Université Paris VI, 1990

UNIVERSITÉ DES SC. ET TECHN. BIBLIOTHÈQUE UNIVERSITAIRE LILLE