UNIVERSITÉ DE NICE-SOPHIA ANTIPOLIS
FACULTÉ DES SCIENCES ET TECHNIQUES

INTÉGRATION RÉFLÉXIVE DE DÉPENDANCES DANS UN MODÈLE À CLASSES

THÈSE DE DOCTORAT
Formation ou école doctorale : Ecole doctorale de Nice-Sophia Antipolis
Discipline : Informatique

présentée
et soutenue publiquement par

DUCASSE Stéphane

le 10 Janvier 1997 à l'ESSI, devant le jury composé de

Rapporteurs	M. COINTE Pierre	Professeur, Ecole des Mines de Nantes.
	M. NIERSTRASZ Oscar	Professeur, Université de Berne.
	M. KICZALES Gregor	Ranx Xerox Parc.
Examinateurs	M. CHAZARAIN Jacques	Professeur, Université de Nice-Sophia Antipolis.
	M. ROUSSEAU Roger	Maitre de Conférence, Université de Nice-Sophia Antipolis.
	M. OUSSALAH Chabane	Professeur, Ecole des Mines de Alès.
Thèse co-encadrée par	M. FRANCHI Paul,	Professeur, Université de Nice-Sophia Antipolis.
	Mme FORNARINO Mireille,	Maître de Conférence, Université de Nice-Sophia Antipolis.

Thèse préparée à l'Université de Nice – Sophia Antipolis
au sein du laboratoire I3S – CNRS URA 1376

A FLOrence,
A mes parents.

Je tiens à remercier sincèrement et avec simplicité Pierre Cointe, Grégor Kiczales et Oscar Nierstrasz d'avoir acceptés d'être mes rapporteurs et j'en suis très honoré. Je sais que leur temps est précieux.
Je remercie

- Paul Franchi de m'avoir accueilli dans son équipe et d'avoir toujours était là pour m'aiguiller lors de choix difficiles.
- Jacques Chazarain, qui m'a fait découvrir les joies de la programmation fonctionnelle et CLOS en Licence et Maîtrise, d'être présent dans mon jury de thèse.
- Roger Rousseau pour ses conseils et ses questions sur l'introduction de la méta-programmation dans une approche génie logiciel.
- Chabane Mourad Oussalah et Martine Magnan pour nos collaborations présentes et futures.
- En tant que béta testeur et premier fan de STklos, Erick Gallésio pour avoir réalisé STklos et avoir répondu à mes nombreuses questions.
- Laurence Nigay pour nos discussions sur PAC, Franck Lebastard et Olivier Jautzy pour leur intérêt pour FLO, Fred Rivard pour les points d'entrée et les métaclasses de NeoClasstalk et nos tardives dicussions, Thomas Ledoux mon compagnon de bureau à Nantes et Philippe Mulet pour nos *méta* discussions.
- Laurent Arditi, mon ami, pour m'avoir supporté durant toutes ces années informatiques et pour l'émulation !
- Florence, ma femme, pour m'avoir supporté durant ces longs mois de rédaction et de tensions.

Enfin, je veux remercier mes deux *mères* informatiques Mireille Blay-Fornarino et Anne-Marie Pinna-Dery de m'avoir épaulé, supporté, écouté, apaisé, encouragé, compris à demi-mots,... et d'avoir toujours trouvé le temps de participer activement à la naissance de FLO...malgré Héloise, Jérémie, Dominique et Virginie et leurs pères. J'ai vraiment passé trois belles années à vos côtés.

Table des matières

1

Problématique et contexte

« While the recent literature recognizes the importance of inter-object behavior, there is surprisingly little language support for its specification and abstraction. This means the existence of behavioral compositions in a system, and in particular behavioral dependencies that they imply, cannot be easily inferred ; they are spread accross many class definition method implementation. This causes subsequent problems in the design, understanding and reuse of object-oriented software. » [HELM 90].

Un grand nombre de travaux ont reconnu que l'abstraction fournie par les classes était trop fine pour le développement de grandes applications. Aussi de nouveaux constructeurs ont été proposés pour grouper des classes : *class dictionaries* [LIEB 88], *Mecanisms* [BOOC 91], *Clusters* [MEYE 90a, MEYE 90b] et *Frameworks* [JOHN 88]. D'un autre côté, de nombreux travaux ont montré les limites du modèle objet [RUMB 87, SHAH 89, FORN 90a, HELM 90, AKSI 92a, BOSC 95a, JUNG 96]. En particulier, le modèle objet traite mal les *dépendances*, *relations* ou *interactions* entre objets [HELM 90]. Alors que des groupes d'objets coopèrent dans le but d'assurer différentes tâches ou de maintenir des invariants entre eux (un ensemble de «radio-boutons» assure qu'un seul des boutons est activé à la fois) les langages à classes traditionnels comme SMALLTALK [GOLD 83], CLOS [STEE 90], EIFFEL [MEYE 90a] ou C++ [STRO 86] n'offrent pas la possibilité d'exprimer facilement de telles tâches.

Le but de cette thèse est de proposer l'intégration de dépendances dans les langages à objets existants, en particulier dans le cadre de langages à classes.

Le plan de ce chapitre est le suivant. Tout d'abord, nous établissons notre vocabulaire en précisant le sens du terme *«dépendance»*. Nous exposons alors les principaux problèmes rencontrés lors de la mise en œuvre de dépendances ou d'interactions dans un langage à classes: absence d'abstraction et de déclarativité des dépendances, problèmes de réutilisation des classes dus au parasitage d'informations relationnelles, absence de contrôle des mécanismes assurant la cohérence des dépendances... Nous proposons notre solution basée sur la séparation claire entre les informations (structure et comportement) intrinsèques de l'objet et les informations relationnelles que sa vie en groupe avec d'autres objets lui confèrent, spécifiées sous forme d'objets relationnels : des dépendances.

1.1 Terminologie

Les termes *relation*, *dépendance* ou *interactions* sont souvent utilisés dans la littérature et ne revêtent pas toujours les mêmes significations. Suivant les auteurs, ces termes prennent tour à tour le sens de références (relation [Rumb 87], associations [Blah 95, Tanz 95]) ou d'interactions (Othelo [Forn 90a], Troll [Jung 93, Jung 96]). Ces mêmes interactions sont modélisées sous différentes formes et désignées diversement selon les méthodes d'analyse et de conception : *object diagrams* [Booc 91], *process model* [de C 91], *message connections* [Coad 91], *data-flow diagrams* [Blah 95] et *collaboration graphs* [Wirf 90].

1.1.1 Relations et dépendances.

J. Rumbaugh définit ainsi le terme relation : *« a relation is an abstraction stating that objects from certain classes are associated in some way »* [Rumb 87]. Pour J. Rumbaugh, une relation associe des objets de n classes. Ces objets définissent l'état de la relation qui peut être changé grâce à des opérations d'addition ou de retrait d'éléments ; cet état est interrogeable par le biais de fonctions de test d'appartenance ; la sélection des éléments d'une relation ainsi qu'une itération sur l'ensemble des éléments de la relation sont possibles.

J. Bosch, quant à lui propose : *« a relation as any connection from one object (or class) Or to another object (or class) Os that influences the behaviour of Or in some way »* [Bosc 95a]. Alors que J. Rumbaugh définit des relations n-aires entre objets qui sont proches de la notion de pointeurs, J. Bosch restreint les relations à des relations binaires et orientées et surtout, insiste sur le fait que l'existence d'une relation peut affecter le comportement des objets ou des classes.

Notre terminologie. Bien que le terme dépendance soit connoté par son utilisation dans la littérature pour désigner un mécanisme de propagation de messages en Smalltalk [Gold 83], nous l'avons préféré au terme *relation* ou *interaction* au vu de sa signification commune[1]. Ainsi nous définissons le terme de dépendance comme suit : *«une dépendance est une entité responsable de l'interconnexion d'un ensemble d'objets, entité qui influence le comportement de ces objets.»*

Cette définition générale proche des précédentes insiste sur l'influence de la dépendance sur le comportement d'un ensemble d'objets. Ainsi, si cette définition recouvre le cas simple des associations [Blah 95], elle intègre également la modélisation des interactions entre les objets. Une remarque s'impose : J. Bosch inclut les relations entre classes (héritage, délégation, parties). Dans cette étude, nous nous limitons aux dépendances entre instances. Les lecteurs intéressés par l'héritage peuvent se référer aux articles suivants : [Card 84, Card 85, Cook 89, Duco 87, Duco 89, Carr 90, Brac 90].

Quelques définitions. Nous définissons les principaux termes que nous utilisons pour parler des dépendances. Ces définitions sont générales et ne sont pas liées à un choix de représentations ; elles s'appliquent aussi bien aux contraintes de ThingLab [Born 86a] qu'aux dépendances de Smalltalk [Gold 83]. Ces définitions seront détaillées lorsque nous décrirons notre modèle.

Une dépendance effective[2] entre objets est caractérisée par :

- une *liste* d'objets participants,
- une *propriété* que les objets participants doivent vérifier. Lorsque la propriété est vérifiée la dépendance est dite *cohérente*. L'expression de cette propriété peut être explicite sous forme d'un prédicat ou implicite.

[1] **dépendance** : n.f. Situation d'une personne qui dépend d'autrui, sujétion, subordination : *être sous la dépendance de quelqu'un* [Larousse85]

[2] Par la suite, nous montrerons que les dépendances à la manière du modèle classe/instances peuvent être décrites puis instanciées de multiples fois. Nous parlons ici d'une «instance» de dépendance qui existe entre des instances particulières.

La dépendance *p1-a-même-abscisse-que-p2* entre les instances `p1` et `p2` qui spécifie que ces deux points ont la même abscisse est cohérente si les `p1` et `p2` qu'elles lient sont effectivement alignés suivant les abscisses. La propriété que doit vérifier la dépendance est l'égalité des valeurs des abscisses des objets liés.

La cohérence d'une dépendance n'est pas forcément vérifiable. Par contre, il est possible de spécifier comment cette propriété est maintenue au vu d'une perturbation connue : c'est le rôle du *comportement dynamique* d'une dépendance.

Ainsi une dépendance *représentation-cohérente* peut spécifier par le biais de règles que la représentation d'un objet est toujours cohérente avec l'état de celui-ci sans que la vérification de cet état soit possible.

- un *comportement dynamique* qui spécifie comment la cohérence de la dépendance est assurée suite à la réception d'un message. Il peut être modélisé sous différentes formes.

  Pour une contrainte en ThingLab [Born 86a], il est défini sous forme de règles réactives permettant de resatisfaire la cohérence de la contrainte. En Othelo [Forn 90a], il est défini par une liste de couples actions/réactions précisant les méthodes à invoquer suite à la réception d'un message. En Smalltalk, il est définit dans les méthodes `update:` des classes des objets participants (voir en annexe A). Dans le cadre de synchroniseurs [Frøl 93], il est défini par des expressions à base de filtrage et d'un mini-langage.

Un ensemble de dépendances induit un *graphe de dépendances* dont les nœuds sont les objets participants aux dépendances et les arêtes étiquetées par les dépendances. Tous les objets ne sont pas obligatoirement liés entre eux par des dépendances. Un graphe de dépendances n'est pas forcément connexe.

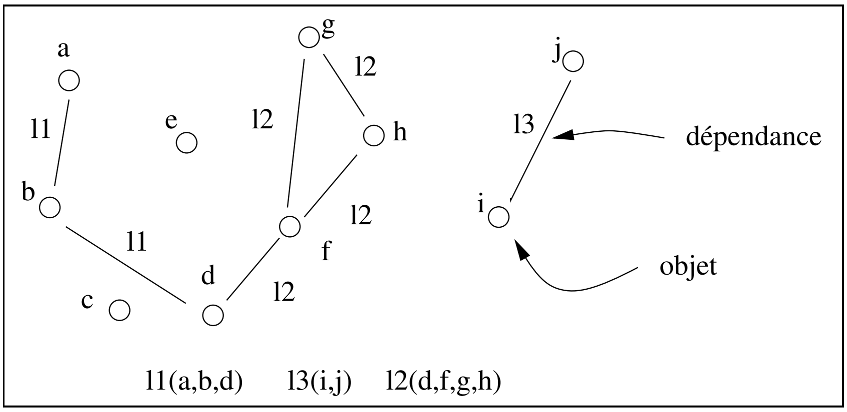


Figure 1.1: Un graphe de dépendance.

Nous appelons *«maintien de la cohérence»* des dépendances l'ensemble des mécanismes qui assurent la cohérence de toutes les dépendances du graphe. Le maintien de la cohérence des dépendances utilise le comportement dynamique des dépendances. Différentes approches sont possibles : certains messages peuvent être autorisés sous certaines conditions (des gardes), certains messages peuvent impliquer des réactions,... Dans ce dernier cas, il est basé sur un rétablissement par propagation [Born 86a, Forn 90a, Chab 93] ; c'est-à-dire qu'une dépendance est dans un état cohérent, puis suite à certaines opérations sur les objets participants la dépendance est rendue incohérente, le maintien de la cohérence a pour but alors d'utiliser le comportement dynamique de la dépendance pour rétablir la cohérence de celle-ci.

Le comportement dynamique de la dépendance *p1-a-même-abscisse-que-p2* peut exprimer que changer l'abscisse d'un des deux points *implique* de changer celle de l'autre. Nous parlons ici de comportement de type *causal*: une cause implique une conséquence [Born 86a, Forn 90a, Chab 93, Jung 96].

Le comportement dynamique peut exprimer qu'un des deux points ne peut bouger que pour avoir la même abscisse que l'autre. Nous parlons ici de comportement *conditionnel* ([Frøl 93, van 91]).

La *«mise en œuvre d'une dépendance»* fait référence à la *définition* d'une dépendance (description du comportement dynamique) et aux mécanismes (ajout de variables d'instances, contrôle des messages, contrôle des affectations...) utilisés pour assurer la cohérence de la dépendance. La mise en dépendance effective d'objets est nommée la *déclaration* de la dépendance entre objets participants.

1.2 Problèmes

Les langages à objets permettent de programmer des dépendances entre objets : il faut pour cela modifier les classes, les méthodes des objets en dépendances pour y introduire les mécanismes de maintien. Cependant, cette façon de faire est très limitée [Rumb 87, Helm 90, Forn 90a, Aksi 92a, Bosc 95a, Jung 96]. En effet, la réutilisation, la maintenabilité, la compréhension et la mise en œuvre des objets et de leurs dépendances sont difficiles, souvent réalisées de manière *ad hoc* et au cas par cas[3]. Pour sa part, la nécessité d'exprimer des dépendances entre objets n'est pas une idée récente, de nombreux travaux ont proposé des mécanismes pour implémenter des dépendances entre objets (dépendances en Smalltalk [Gold 83], valeurs actives [Stef 86a], contraintes [Born 86a]...). Cependant, ni les mécanismes proposés ni le modèle objet traditionnel ne sont satisfaisants par rapport aux problèmes suivants.

Discontinuité entre conception et implémentation.

Le concept de dépendances joue un rôle important dans la modélisation des applications [Booc 91, Blah 95, Wirf 90]. Cependant, la plupart des langages utilisés lors de la phase d'implémentation n'offrent aucun moyen d'exprimer des dépendances au même niveau que dans la phase de conception. Lors de cette phase d'implémentation, le programmeur est contraint d'exprimer les dépendances en utilisant l'héritage, la composition et/ou les mécanismes proposés par certains langages (valeurs actives, méthodes auxiliaires...)[Bosc 95a]. Les dépendances en tant qu'entités conceptuelles sont alors *perdues* ; elles sont dispersées au travers des classes et des méthodes.

Spécification non déclarative.

Les approches traditionnelles restent des approches impératives alors qu'une approche plus déclarative serait souhaitable :

- Que ce soit dans un modèle à classes ou à prototypes, le programmeur doit lors de la définition des objets penser aux dépendances potentielles auxquelles ils pourraient participer. Or la définition d'un objet ne devrait considérer que les informations qui lui sont propres.

- Dans un modèle à classes, le programmeur est obligé de définir de nouvelles classes (ajout d'attributs et démons, modification de méthodes) dans le seul but d'exprimer des dépendances et leur comportement dynamique. Ces classes sont souvent alors dénuées d'une réelle sémantique. Elles ne sont que des extensions dues à une lacune du modèle [Boua 94, Boua 95].

- Le fait d'inclure des aspects liés aux dépendances entre objets à l'intérieur de ceux-ci, pose le problème de la gestion de la complexité du comportement des objets par le programmeur. Ainsi cette manière de faire va à l'encontre de l'abstraction : le programmeur doit être à

[3] Cette remarque est à mettre en parallèle avec les insuffisances des modèles objets traditionnels pour traiter correctement l'introduction de la coordination et de la synchronisation d'objets concurrents. Des solutions *ad hoc* existent mais le modèle ne répond pas à ces exigences [Nier 87, Neus 91, Frøl 92].

même d'introduire des informations relatives aux dépendances et d'appréhender la complexité des objets alors qu'il ne devrait avoir une perception de ces objets qu'au travers de leurs interfaces.

Nous pensons qu'un objet doit définir ses informations intrinsèques et que ses interactions avec d'autres objets et les contraintes qu'elles définissent doivent être spécifiées de manière séparée de l'objet puis appliquées aux objets intéressés. Ainsi un objet doit être perçu suivant un double point de vue : lui et ses interactions avec son entourage.

Mise en oeuvre délicate.

La mise en oeuvre des dépendances (traditionnelle, sous forme d'attributs et de mécanisme réactif comme le déclenchement de démons, de méthodes auxiliaires ou de valeurs actives) s'avére délicate. En effet, outre la prise en compte de la complexité des objets (voir au dessus), certaines dépendances possèdent des comportements dynamiques réactifs ou doivent maintenir des invariants [Aksi 94]. Il est alors de l'entière responsabilité du programmeur de mettre en place les mécanismes contrôlant ces comportements : la gestion des cycles par exemple pose un problème. Ainsi en Loops ou Smalltalk aucun mécanisme n'est proposé pour contrôler la propagation des réactions.

Abstraction inexistante.

L'absence d'abstraction des dépendances ne permet pas de donner aux dépendances une véritable place au sein des concepts objets [Rumb 87]. Or, cette absence d'abstraction des dépendances est préjudiciable au programmeur qui ne peut pas facilement maintenir et faire évoluer son code. A aucun moment, le programmeur n'appréhende une dépendance de la même manière qu'il pourrait le faire pour un objet.

Cette absence d'abstraction est couplée au manque de déclarativité des dépendances. Par exemple, l'utilisation de dépendances en Smalltalk pour le modèle architectural MVC [Kras 88] rend le code difficile à comprendre car il faut suivre le flot de propagation des dépendances pour comprendre les dépendances entre objets [Barr 93] (voir en annexe A.3).

Là encore, il serait souhaitable d'avoir d'un côté la description des objets et de l'autre la description des dépendances entre ces objets clairement exprimées. La compréhension des objets en serait grandement améliorée.

Complexité et manque de réutilisabilité.

Le code d'un objet est parasité par les dépendances auxquelles il est susceptible de participer. Chaque nouvelle dépendance complexifie les objets [Doda 95]. De par cette fusion entre objets et dépendances, les comportements intrinsèques de l'objet ne sont pas facilement identifiables, ce qui nuit à la fois à la réutilisation et à l'évolution des objets et des dépendances.

Perception unique et non adaptable des dépendances.

Certains langages ont proposé des solutions comme les dépendances de Smalltalk [Gold 83, Brif 96], les formules de KR [Gius 92] ou les contraintes en ThingLab [Born 86a] (voir chapitre 2). Cependant, ces approches ne proposent qu'une vision étroite du concept de dépendances. Elles définissent une entité et un mécanisme de maintien unique et figé pour représenter les dépendances. Il n'est pas possible de modifier ou d'étendre la spécification et le comportement des dépendances. Le mécanisme de maintien de la cohérence est figé et non modifiable.

1.3 Notre contribution

Le but de cette thèse est l'étude de l'intégration de dépendances dans les langages à objets existants, en particulier dans le cadre de langages à classes à liaisons dynamiques et à méta-classes comme SMALLTALK et CLOS. Cette étude a donné lieu à :

- la définition d'un modèle de dépendances entre objets,
- l'intégration de ce modèle dans un modèle à classes,
- l'implémentation de ce modèle dans un langage à classes,
- la définition d'un protocole méta-objet pour les dépendances et
- l'illustration de l'utilisation de ces dépendances.

Avant de reprendre plus en détail ces points, nous définissons plus précisément le contexte de notre travail.

1.3.1 Contexte et objectifs

Intégration. Par le terme *intégration* de dépendances dans un modèle à classes, nous signifions que nous voulons modifier au minimum les aspects fondamentaux du modèle objet dans lequel le notre s'intègre. Le modèle obtenu doit être uniforme ; c'est-à-dire proposer un nombre limité de concepts s'intégrant au modèle initial. En particulier, cette intégration doit se faire en utilisant les concepts du modèle objet initial. Nous souhaitons ainsi déterminer les propriétés minimales que doit posséder un langage pour que notre approche puisse y être introduite. Il ne s'agit pas de définir un nouveau langage objet.

Ensuite, par référence à la programmation objet, il ne s'agit pas de proposer un nouveau langage réactif[4] synchrone du type de ESTEREL [BOUS 91] ou LUSTRE [HALB 91] à partir duquel un modèle objet pourrait être défini. Bien qu'une telle approche offre sur le plan sémantique des propriétés comme le déterminisme et la détection de cycles, elle reste à l'opposé de notre approche. Notre approche est de choisir un modèle objet, à savoir principalement celui de SMALLTALK ou d'OBJVLISP, et d'étendre ce langage pour prendre en compte des dépendances.

Ces deux points peuvent être résumés par la phrase : intégration uniforme de dépendances entre objets en utilisant les mécanismes objets existants.

Adaptabilité. Ce second objectif se veut qualitatif par rapport au premier. En effet, les années 1990 ont vu apparaître une nouvelle génération de systèmes, en particulier de langages de programmation dont CLOS reste le chef de file. Ces langages dit *«ouverts» (open languages)* sont adaptables ; ils permettent à un utilisateur de modifier certains de leurs comportements ou d'en définir de nouveaux de l'intérieur même du langage. CLOS permet de modifier la sémantique de l'héritage, de l'application des méthodes en spécialisant le comportement de certaines classes qui décrivent son comportement initial.

Les dépendances que nous modélisons ne nécessitent pas toutes les mêmes mécanismes. Aussi au lieu de définir un langage essayant de proposer *a priori* une solution pour chaque type de dépendances, nous avons choisi de proposer un langage flexible et adaptable. Notre solution doit donc permettre de particulariser et d'avoir le contrôle des mécanismes mis en place pour la gestion des dépendances. Le but est alors de trouver quels sont les mécanismes qui sont à la fois nécessaires à la gestion des dépendances et qui permettent de définir de nouveaux comportements.

1.3.2 Architecture logique de notre approche

Dans cette partie, nous présentons l'architecture logique de notre travail qui est basée sur l'extension de l'envoi de message, le concept de *dépendances* et de *contrôleurs*. Notre travail se décompose en deux niveaux : un modèle de dépendances et son intégration dans un modèle à classes. Tout d'abord, nous distinguons l'expression et le maintien local de dépendances. La nécessité d'un

[4]Le terme «réactif» utilisé ici fait référence à certains comportements réactifs des dépendances que nous proposons.

contrôle global du maintien des dépendances nous a conduit à la notion de contrôleurs. Le second niveau s'intéresse à l'intégration du niveau précédent dans un modèle à classes et à la proposition d'une intégration extensible basée sur la définition d'une méta-architecture.

1.3.3 Un modèle de dépendances

Envoi de message étendu et dépendances.

Dans les langages impératifs du type Pascal ou Modula, le compteur ordinal est la séquence représentée par le point virgule. Dans les langages fonctionnels purs, l'application de fonction représente ce compteur ordinal. Pour les langages à objets purs, le compteur ordinal est l'envoi de message. Notre travail peut être perçu comme une modification du compteur ordinal dans un langage à objets. Ainsi, nous incluons à la signification traditionnel de l'envoi de messages la prise en compte d'un contexte relationnel. De la même manière que Agha dans [FrØl 93], nous considérons que le comportement d'un objet dépend du contexte dans lequel celui-ci se trouve. Dans notre approche, ce contexte est décrit par la donnée de dépendances.

> Ainsi un point `c1` n'a pas le même comportement s'il est isolé que s'il est en relation avec d'autres points pour former un segment horizontal. Un bouton ne réagit pas de la même manière seul ou s'il est en relation avec d'autres boutons pour constituer un groupe de radio-boutons.

Une dépendance décrit de manière déclarative, explicite et locale l'influence de la réception d'un message par un des objets participants sur les participants de la dépendance. Il peut alors s'agir d'interdire le message, de le déléguer ou d'envoyer d'autres messages en réaction à celui-ci.

> Un bouton d'une colonne de radio-boutons ne doit accepter un message que si aucun autre bouton n'est déjà sélectionné.

Dans une approche traditionnelle, le comportement même de l'objet doit prendre en compte ce contexte. Ainsi, pour assurer que lorsqu'un point bouge le segment reste horizontal, la méthode `move` doit être modifiée. Dans notre approche, l'envoi de message étendu prend en compte le contexte donné sous forme de dépendances explicites définies de manière externe aux objets participants. L'envoi de message inclut le maintien de la cohérence des dépendances.

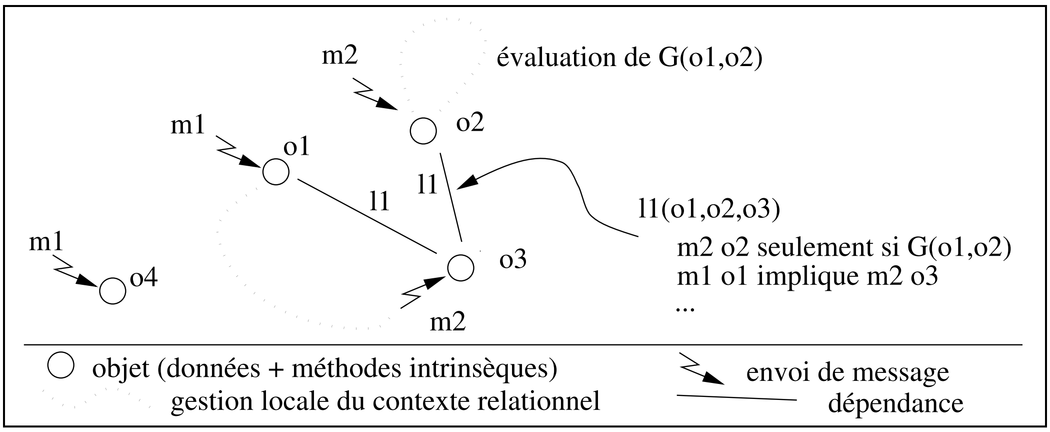


Figure 1.2: Un envoi de message étendu prenant en compte la donnée d'un contexte local d'interprétation d'un message sous forme de dépendance.

Cette situation est illustrée par la figure 1.2. Les objets `o1`, `o2` et `o3` participent à la dépendance `l1`. Lorsqu'un message de sélecteur `m2` est envoyé à l'objet `o2`, la méthode n'est appliquée que si la garde `G` est vérifiée. De même, un message `m1` envoyé à l'objet `o1` implique d'envoyer un message `m2` à `o3`. L'objet `o4` ne participant à aucune dépendance reçoit et traite le message de manière traditionnelle.

Nécessité d'un contrôle.

Les dépendances exprimant une influence locale entre participants par le biais de leur comportement dynamique et pouvant connectées sous forme de graphe, un message perturbant peut entrainer une *propagation de messages*. Un message peut impliquer des messages compensatoires qui à leur tour sont perturbants et impliquent d'autres messages compensatoires. Des cycles peuvent alors apparaître. Contrairement aux systèmes basés sur une propagation *immédiate* comme les valeurs actives ou les dépendances de SMALLTALK, notre approche veut décharger le programmeur de l'implémentation de la gestion des cycles au niveau des dépendances. D'autre part, nous voulons un mécanisme de maintien qui prennent en compte une partie (ou tout le graphe) des dépendances. Finalement, nous ne voulons pas d'un mécanisme de maintien unique, câblé dans l'interprète du langage et non particularisable. C'est pourquoi nous introduisons de nouvelles entités : des contrôleurs. Un contrôleur gère le maintien de la cohérence des dépendances d'un groupe d'objets. Alors qu'une dépendance spécifie l'interaction entre participants, un contrôleur la réalise en utilisant l'ensemble des dépendances définies sur le groupe d'objets qu'ils supervisent.

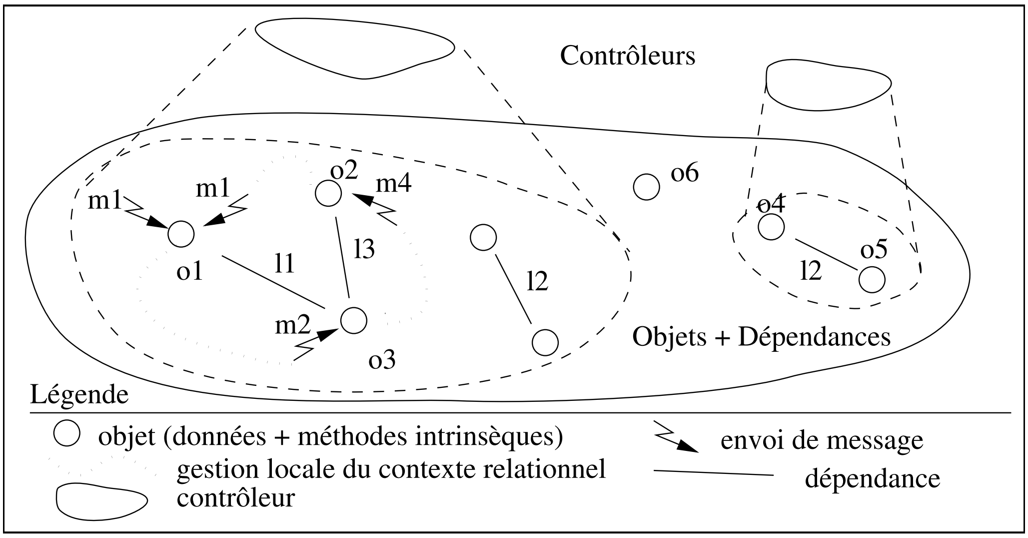


Figure 1.3: Une architecture à deux niveaux pour un contrôle global du maintien des dépendances.

La figure 1.3 illustre le contrôle opéré par les contrôleurs afin d'assurer un maintien global des dépendances. Le contrôleur de gauche gère par exemple un cycle dans le maintien de la cohérence. Ce modèle est présenté au **chapitre 4.**

1.3.4 Une intégration dans un modèle à classes

Jusqu'à présent, nous avons énoncé notre modèle en termes d'objets de manière générale, nous n'avons pas fait d'hypothèse sur l'existence de classes. Dans un modèle à classes, les classes représentent des abstractions de la structure et du comportement des objets. Une classe sert de moule pour générer des instances qui possèdent la même structure et partagent le même comportement. Cette notion de factorisation de structure et de comportement pose un problème par rapport à la notion de dépendances. En effet, nous avons exprimé qu'une dépendance modifie le comportement d'un objet. Il s'agit donc de savoir comment les dépendances se positionnent par rapport aux classes.

L'intégration des entités décrites précédemment dans un modèle à classes nous amène à répondre aux deux questions suivantes :

- Comment les dépendances, et en particulier, la gestion d'un contexte différent pour les objets participants à une dépendance, se définissent-elle par rapport à l'abstraction et la factorisation apportées par les classes? Comment introduire la possibilité de traiter différemment certaines des instances d'une même classe afin de gérer l'existence de dépendances?

- Quels apports l'utilisation de classes offre-t-elle au modèle?

Répondre à la première question en gérant le mécanisme de dépendances au niveau des classes des objets participants ne nous satisfait pas, que ce soit d'un point de vue structurel ou comportemental. En effet, une dépendance décrit une interaction entre objet et n'a pas *a priori* à être associée structurellement à une classe.

Par exemple, il est possible de définir une dépendance spécifiant une situation d'exclusion entre objets et d'appliquer cette dépendance à des instances de différentes classes.

Au niveau comportemental, toutes les instances d'une même classe seraient pénalisées si l'une d'entre elles participe à une dépendance. D'autre part, elles posséderaient toutes le même mécanisme de maintien [Ferb 89].

Notre solution est la suivante : nous ne modifions pas le statut des classes ; elles décrivent la structure et les fonctionnalités intrinsèques de leurs instances. Les dépendances décrivent de manière indépendante et autonome les interactions entre objets. Les envois de messages envoyés à des instances participant à une dépendance sont contrôlés par un contrôleur qui maintient alors la cohérence des dépendances. Cette solution a le mérite de séparer logiquement les fonctionnalités intrinsèques d'un objet de celles liées à la gestion des dépendances.

De la même manière qu'une classe décrit le comportement de toutes ses instances, représenter les dépendances par des classes permet de factoriser le comportement dynamique d'une dépendance et de créer des instances de cette dépendance entre différents groupes de participants. La dépendance *même-abscisse* peut être instanciée plusieurs fois entre différents points.

La description abstraite de dépendances nous a conduit à définir un mécanisme de définition incrémental : un héritage spécifique entre dépendances. Cette intégration des dépendances dans un modèle à classes est présentée aux **chapitres 5 et 6.**

1.3.5 Une méta-architecture pour la gestion des dépendances

L'utilisation de classes pour représenter des dépendances possède d'autres avantages au niveau de l'implémentation du modèle :

- Tout d'abord, le modèle proposé est uniforme : la structure et les fonctionnalités propres aux dépendances sont spécifiés par la définition de classes. Les comportements initiaux sont spécifiés et implémentés à l'aide du même langage. D'autre part, les fonctionnalités propres aux dépendances peuvent être modifiées ou étendues en utilisant la spécialisation offerte par l'héritage entre classes.

- La représentation sous forme d'objets des contrôleurs offrent les mêmes avantages. De plus, elle définit une abstraction du mécanisme même de maintien de la cohérence. Ce mécanisme est alors contrôlable et particularisable : il est possible de spécialiser ou de proposer d'autres mécanismes pour la gestion globale des dépendances.

Afin de proposer une implémentation offrant la possibilité de contrôler la gestion complète des dépendances, nous avons inclus la possibilité de modifier l'héritage entre dépendances ainsi que la structure même de celles-ci. Ces fonctionnalités sont basées sur l'existence de méta-classes responsables de la gestion des dépendances. Cette architecture est présentée aux **chapitres 8 et 9.**

1.3.6 Instanciation du modèle et limites

Bien qu'il soit basé sur un contrôle de l'envoi de messages, le modèle proposé ne pose pas de réelles difficultés pour son implémentation dans de nombreux langages à objets. Diverses techniques existent pour prendre le contrôle de l'envoi de messages. Par contre, les contraintes de certains langages comme C++ amèneraient sûrement à revoir les aspects dynamiques du modèle comme la création dynamique de dépendances.

Il faut remarquer que la méta-architecture est fortement liée à l'existence de méta-classes dans le langage d'implémentation. Elle est facilement définissable dans les langages à méta-classes comme SMALLTALK, NEOCLASSTALK, CLOS ou STKLOS. Les aspects directement liés à l'utilisation des méta-classes (particularisation de l'héritage, changement de représentation de structure des dépendances) sont difficilement adaptables à des langages n'offrant pas de méta-classes.

En conclusion, le modèle, modulo quelques adaptations, ne demande pas de caractéristiques notables pour être introduit dans un langage à objets. Par contre, la méta-architecture nécessite la présence de méta-classes. L'utilisation de langages réflexifs à méta-classes permet une implémentation plus facile et conforme au modèle et à la méta-architecture.

1.4 Plan de la thèse

Cette thèse se compose de quatre parties. La première partie présente une étude de l'existant à partir de laquelle sont formulées les différentes caractéristiques que notre intégration doit posséder. Sur cette base, la deuxième partie présente notre modèle de dépendances : sont précisées alors les différentes entités intervenant dans notre modèle. La troisième partie a un double objectif : d'une part, elle présente une implémentation possible de notre modèle et d'autre part, elle décrit un protocole méta-objet[5] pour la gestion des dépendances. La quatrième partie montre des applications des dépendances.

1.4.1 Un tour d'horizon

Certains travaux proposent une meilleure prise en compte des interactions, relations ou dépendances entre objets. Ces travaux se situent principalement autour de deux axes : la modélisation et la proposition de nouveaux langages de programmation. Ainsi de nouvelles approches de la modélisation des applications ont été proposées (Traverse Activities [KRIS 93, KRIS 94, KRIS 96, ANDE 92], Contracts [HELM 90, HOLL 92]). De nouveaux langages objets sont proposés (TROLL [JUNG 96], LAYOM [BOSC 95a, BOSC 95b], DSM [SHAH 89], ACT [AKSI 94], OTHELO [FORN 90a]). Parfois sans aller jusqu'à proposer un nouveau langage certains travaux proposent de nouveaux outils pour la définition des interactions [DODA 95]. Dans cette thèse, nous avons étudié plus particulièrement les travaux proposant de nouveaux langages. Pour cela, nous avons proposé un ensemble de critères permettant de caractériser chacun des travaux intégrant des dépendances.

Une synthèse. Trois principales approches existent : celles basées sur l'utilisation de mécanismes réactifs, celles modifiant l'interface d'un objet pour y inclure des informations relationnelles et celles proposant le concept de dépendance. Les critères proposés caractérisent : entre quelles entités (champs ou objets) les dépendances sont définies (*granularité*), si les dépendances sont définies en même temps que les classes des objets qu'elles lient (*localisation*), si les dépendances interviennent explicitement du point de vue de l'utilisateur dans le maintien de la cohérence (*transparence*). Ensuite une dépendance peut avoir plusieurs statut : elle peut être *dispersée*, *factorisée*, considérée comme une *entité* ou *prédominante* par rapport aux classes. Les dépendances peuvent alors être sujettes à une réification[6].

Différents modèles de maintien de la cohérence existent. Ils sont basés sur un contrôle du changement de valeurs des variables d'instances ou de l'envoi de messages. Nous distinguons dif-

[5] Un protocole décrit les interactions et les fonctionnalités d'un ensemble de classes. Un protocole méta-objet est un protocole dédié à la particularisation du langage même. Il décrit les interactions et fonctionnalités du méta-langage. C'est par son intermédiaire que le langage peut être adapté ou modifié.

[6] Littéralement le terme «réification» est un terme philosophique synonyme de «transformation en chose». Dans le contexte informatique et plus précisément dans le contexte de la programmation objet, le terme réification désigne l'ensemble des techniques rendant explicite sous forme d'objets certaines parties jusqu'alors implicites ou non accessibles du langage. La réification peut s'appliquer aussi bien aux aspects structurels du langage : classes, méthodes, champs...qu'aux aspects dynamiques : pile d'exécution, contexte d'évaluation...

férentes interprétations du comportement dynamique qui peut être basé sur une réactivité *immédiate* ou *calculée* et la notion de *garde*. La propagation de messages dans un graphe de dépendances peut amener à des cycles. Le déterminisme et la terminaison des programmes sont abordés : à l'exception des systèmes de contraintes aucun des autres langages ne propose une gestion des cycles lors de la phase de propagation dans le graphe de dépendances. De plus, aucun des systèmes n'est déterministe : les états dans lesquels se trouvent les objets suite au maintien de la cohérence dépendent du point de départ dans le graphe et de l'ordre d'évaluation du comportement dynamique des dépendances. Cette étude est présentée au **chapitre 2** et dans [Dery 96a].

Une proposition d'une intégration dans le modèle ObjVlisp. Les critères énoncés précédemment nous servent de base pour définir les caractéristiques que doit posséder une intégration dans le modèle ObjVlisp [Coin 87]. En ObjVlisp, un objet est une entité communiquant exclusivement par envois de messages. Un objet est une boîte noire : il est défini par son comportement et non par sa structure. Nous proposons une intégration basée sur la réification et l'autonomie des dépendances : les dépendances doivent être des entités de même statut que les classes, dissociées des classes des objets participants et exprimées entre objets et non entre leurs variables d'instances. Elles doivent respecter l'encapsulation et donc être basées sur un contrôle de l'envoi de messages. Le maintien de la cohérence doit proposer différentes solutions : de la propagation de réactions suite à une perturbation, des gardes subordonnant l'exécution de méthodes.... Les cycles doivent être gérés. De plus, le mécanisme même de maintien de la cohérence doit être particularisable. Cette proposition est détaillée au **chapitre 3.**

1.4.2 Un modèle

La présentation de notre modèle s'articule en deux parties. Nous présentons au **chapitre 4** notre modèle dans le cadre de langages à objets en général puis dans le contexte d'un modèle à classes aux chapitres **5 et 6**.

Notre modèle définit deux nouvelles entités : des *dépendances* et des *contrôleurs*. Une dépendance spécifie les objets y participant et la manière d'assurer localement le maintien de sa cohérence. Un contrôleur assure effectivement ce maintien de la cohérence du graphe des dépendances en contrôlant les messages envoyés aux participants et en utilisant le comportement dynamique des dépendances.

Les dépendances proposées améliorent les dépendances de Othelo [Forn 90a](l'ancêtre de notre modèle) sur de nombreux points. Nos dépendances sont n-aires et non-orientées. Les groupes d'objets sont gérés. Plus fondamentale est notre perception du comportement dynamique d'une dépendance. En effet, il ne s'agit plus de baser exclusivement le maintien de la cohérence des dépendances sur un mécanisme réactif unique et figé. Le mécanisme du maintien de la cohérence des dépendances proposé est modifiable et extensible. Il est basé sur la notion d'opérateurs qui spécifient l'interprétation des règles qui composent le comportement dynamique. Chaque opérateur possèdent une sémantique propre. De nouveaux opérateurs peuvent être définis de manière indépendante des autres opérateurs : les dépendances peuvent ainsi définir des gardes, de la délégation de messages... Une dépendance spécifie des interactions entre objets et ne se limite pas à une expression réactive du maintien de la cohérence entre objets.

Les contrôleurs sont des entités responsables de l'interprétation des opérateurs et de la cohérence *globale* du graphe des dépendances. Ainsi, ils sont l'abstraction du maintien de la cohérence des dépendances. Par leur intermédiaire notre modèle permet alors de modifier les algorithmes de maintien de la cohérence et de définir de nouveaux opérateurs.

1.4.3 Une implémentation adaptable

Dans cette troisième partie, nous proposons une implémentation basée sur le contrôle de l'envoi de messages [Duca 93, Duca 94a] dans un langage proche du modèle ObjVlisp, c'est-à-dire réflexif et à méta-classes explicites. Cependant, notre discours ne se limite pas à une simple implémentation. En effet, la diversité des dépendances, le besoin de flexibilité pour la définition de ces nouveaux concepts et l'extensibilité du modèle nous ont conduit à proposer une implémentation du modèle extensible et modifiable [Duca 94b, Duca 95a]. Pour cela, nous avons déterminé les fonctionnalités essentielles et suffisantes à la gestion des dépendances à partir desquelles des variations peuvent être introduites. Nous proposons un protocole pour la gestion des dépendances tant au niveau des dépendances que des contrôleurs. Nous montrons quelques extensions possibles. L'architecture et le protocole sont présentés aux chapitres **8 et 9.**

1.4.4 Des applications

Cette dernière partie présente des utilisations des dépendances.

Supports pour l'implémentation de schémas de conception. Les *schémas de conception, (design patterns)*, sont de plus en plus utilisés lors des phases de conception des applications. Cependant, les langages d'implémentations ne proposent pas de mécanismes spécifiques pour leur prise en compte. Notre modèle en exprimant et contrôlant les interactions entre objets par le biais des dépendances et des contrôleurs offrent un outil adapté pour l'implémentation de certains schémas de conception. Cette utilisation des dépendances est illustrée au chapitre **10.**

Communications entre composants de modèles architecturaux. Dans le domaine de la réalisation et de la décomposition logicielle des interfaces Hommes-Machines, il est courant d'utiliser des modèles architecturaux pour modéliser la séparation entre les parties fonctionnelles de leur représentations. Notre langage permet une implémentation immédiate d'interfaces graphiques sur le modèle ALV [Hill 92]. Nous montrons comment notre modèle permet d'implémenter également le modèle PAC [Cout 89] dont nous proposons une relecture. Ce travail est décrit dans [Dery 94, Duca 95b, Dery 96b, Dery 96c] et abordé au chapitre **11.**

Réification de l'héritage. Dans le dernier chapitre, nous réifions l'héritage du modèle ObjVlisp au moyen de dépendances entre classes. Une telle approche pose alors le problème du bootstrap et de la stabilité du système obtenu [Duca 95c, Duca 96a]. Ce travail est traité au chapitre **12.**

1.4.5 Des références

Les travaux présentés dans cette thèse ont fait l'objet de différentes publications (articles ou rapports de recherche) :

- une première intégration du modèle dans CLOS [Duca 93, Duca 94a] ;
- une implémentation à base de méta-objets et contrôle du maintien de la cohérence [Duca 94b, Duca 95a] ;
- une présentation générale du langage [Duca 96b] ;
- la réification de l'héritage et *bootstrap* dans le modèle ObjVlisp [Duca 95c, Duca 96a] ;
- application à la construction d'interfaces [Dery 94, Duca 95b, Dery 96b, Dery 96c] et
- un état de l'art en matière du traitement des dépendances [Dery 96a].

Une version du langage FLO implémentée en STklos est disponible à l'adresse `http://www.essi.fr/~ducasse/FLO/`.

Une version en NeoClasstalk est en cours de développement. Une version en OpenC++ est en cours de réalisation par O. Jautzy de l'équipe BD du CERMICS à Sophia-Antipolis. L'objectif est d'utiliser le langage FLO pour offrir des contraintes d'intégrités réactives dans le cadre de bases de données objets.

Partie I

État de l'art

2

Objets et Dépendances

« A relation represents an inherent constraint between objects of two or more classes. This constraint is not something to be hidden, but rather to be specified abstractly, without imposing an implementation. » [RUMB 87].

Le but de ce chapitre est de faire le point sur la modélisation des dépendances dans les langages à objets ou les bases de données orientées objets actives. Nous avons choisi de présenter les différentes intégrations des dépendances suivant trois axes. Nous étudions les impacts de l'adjonction des dépendances au modèle objet, puis nous analysons les apports d'une approche objet pour la modélisation des dépendances et finalement nous nous intéressons plus précisément aux différents mécanismes de maintien de la cohérence mis en jeu dans les langages à objets intégrant des dépendances. Ce travail de synthèse nous permet alors de proposer au **chapitre 3** une intégration idéale des dépendances dans un modèle à classes de type OBJVLISP ou SMALLTALK. Une partie de cette synthèse constitue un des chapitres du livre [OUSS 96]. Une étude plus détaillée de certains des langages présentés dans cette synthèse est proposée en annexe.

2.1 Introduction

Le concept de *dépendance*, *relation* ou *interaction* dans les langages à objets a été et reste le sujet de nombreuses recherches. Ce bouillonnement de travaux provient principalement du rôle central de ce concept.

Or, les langages à objets «traditionnels» ne permettent pas l'expression de telles dépendances et offrent tout au plus quelques mécanismes pour les implémenter. Pour pallier cette faiblesse, des extensions du modèle objet ont été définies qui facilitent le travail du programmeur en lui permettant d'exprimer les dépendances sans avoir à implémenter les mécanismes qui les gèrent. Ainsi, en fonction des applications visées, ont été intégrés des outils pour exprimer : des contraintes géométriques dans les interfaces (THINGLAB [BORN 86b], GARNET [MYER 90], PLUS [BOUA 94]), des communications entre les différents composants des interfaces graphiques (RENDEZVOUS [HILL 94]), de l'animation d'algorithmes (ANIMUS [BORN 86b]), des contraintes d'intégrité ou des triggers dans les bases de données orientées objets actives (OODB) [WIDO 96] (ODE [GEHA 92], HIPAC [DAYA 88, DAYA 96],Sentinel [CHAK 92], SAMOS [GATZ 92], AMOS [RISC 92], ACOOD [BERN 92] O_2 [MEDE 91]), de la coordination d'objets dans les langages concurrents ([FRØL 93, FRØL 94], PROCOL [VAN 91]), des contraintes (TROPES [GENS 93], PROSE [BERL 92], SOLVER [PUGE 93], CSPOO [KOKE 93]), du maintien de cohérence dans des hiérarchies de composition (COBRAS

[Vaut 96]), de la composition de composants logiciels (Gluon [Pint 93]) et des interactions entre objets (Troll [Jung 93, Jung 96], LayOM [Bosc 95a], ACT [Aksi 94]).

Parce que guidées par des impératifs d'applications, ces intégrations présentent des caractères très différents tant d'un point de vue modèle objet qu'elles proposent, que par la diversité des dépendances qu'elles permettent d'exprimer. Cependant, tous ces travaux ont comme buts communs l'abstraction, la composition et la réutilisation des dépendances. Pour les bases de données, nous limitons notre étude aux bases de données orientées objets. En effet, bien que les bases de données (relationnelles ou non) comme SQL3, Ingres, Sybase, DIPS ou DATEX offrent des mécanismes de maintien de la cohérence des données sous formes de règles, elles n'intègrent pas le concept d'objet et les règles sont souvent limités aux actions d'insertion ou de mise a jour des données [Kim 90, Wido 96]. Les bases de données orientées objets actives offrent pour leur part la possiblité, par le biais d'événements d'exprimer des règles de maintien au niveau du comportement des objets.

2.2 Concept d'objet et dépendances

Nous étudions, dans cette partie, quelles entités du langage à objets peuvent être liées, quand et où les dépendances doivent être déclarées, quels nouveaux comportements peuvent être induits par l'intégration des dépendances et quels sont ses impacts sur l'encapsulation.

2.2.1 Différents types de couplage objet / dépendance

L'introduction de dépendances dans les langages à objets ou les OODB a pris différentes formes. Nous les avons regroupées en trois familles : celle qui propose à l'utilisateur des mécanismes réactifs pour implémenter lui-même ses dépendances, celle qui enrichit l'interface des objets et celle qui introduit effectivement le concept de dépendances.

Mécanismes réactifs.

Entre autres mécanismes offerts pour implémenter des dépendances, notons les valeurs actives [Stef 86b], les démons [Bout 93], les attachements procéduraux des langages de frames comme KRL [Bobr 77], FRL ou Klone [Brac 83, Brac 85] ou encore les méthodes auxiliaires (Flavors [Moon 86], CLOS [Stee 90, Keen 89]) (voir en Annexe A).

Ces mécanismes permettent le déclenchement automatique des mises à jour. Ils offrent un premier niveau de séparation entre les fonctionnalités de l'objet et l'implémentation des dépendances. Mais ils ne permettent qu'une mise en oeuvre partielle des dépendances. Les valeurs actives ne permettent pas de lier un objet dans son identité globale à un autre, mais de lier seulement une ou plusieurs variables à un objet. La dépendance ne s'exprime que par réaction à des modifications de variables d'instances, or certaines dépendances pour être maintenues nécessitent de contrôler des messages (autres que des accesseurs).

Interface relationnelle.

Des travaux proposent d'enrichir l'interface des objets par la spécification de l'aspect dynamique du comportement des objets et de leur coopération [Arap 92, Yell 94, Bast 95]. Dans la suite nous nommerons ces extensions de l'interface de l'objet : *interface relationnelle*.

Yellin et Strom proposent dans [Yell 94] d'enrichir les interfaces des objets grâce à des spécifications qui incluent des contraintes de séquencement des méthodes. Ces contraintes de séquencement sont définies au moyen de grammaire d'états finis et de règles de transitions, elles spécifient les messages que l'objet peut recevoir ou envoyer suivant l'état dans lequel il se trouve. Ces contraintes rendent explicites certaines dépendances entre messages dans une application, dépendances qui sont souvent données implicitement dans les commentaires ou dans le code. Pour sa part, Arapis [Arap 92] enrichit l'interface des objets pour prendre en compte des contraintes d'ordonnancement des messages reçus et émis par une approche formelle basée sur la logique temporelle de la composition d'objets. Dans le formalisme des objets coopératifs (OC) [Bast 95], cet aspect est décrit par des réseaux de Petri. Lors d'une invocation entre objets coopératifs, la sémantique de la communication entre objets est également décrite par réseaux de Petri.

Cette approche conduit à centraliser les communications. Elle présente alors l'avantage de pouvoir assurer des propriétés telles que la présence ou l'absence de cycles, la terminaison, etc.

Concept de dépendance.

Une autre forme de couplage consiste à considérer les dépendances comme des éléments du langage. Deux approches peuvent être distinguées : une qui consiste à fusionner les modèles d'objets et de contraintes telle que ThingLab [Born 86b] (voir en Annexe B), une autre qui consiste à étendre le modèle objet par l'introduction de dépendances entre objets. Ces deux approches permettent d'apporter un soin particulier à la définition de dépendances qui ne sont plus dispersées dans les classes.

Smalltalk fut le premier langage à introduire le concept de dépendance et à l'utiliser de manière quasi systématique, en particulier pour la gestion du modèle architectural MVC [Gold 83, Kras 88, Brif 96]. Initialement, les dépendances auraient dues être classées comme un mécanisme réactif de propagation de messages. Aujourd'hui, les dernières versions de Smalltalk, comme VisualWorks [Par 94] intègrent une réification des dépendances : les `DependencyTransformer` [Brif 96]. Certes ces dernières n'offrent pas une expressivité aussi poussée que les relations de Troll [Jung 96] mais elles pallient certains désavantages des dépendances initiales de Smalltalk (voir annexe A.3).

Les bases de données actives introduisent la notion de contraintes (Ode [Geha 92] et de triggers (Ode,SAMOS [Gatz 92]). Les triggers déclenchent des procédures lorsque certains événements se produisent. Certaines approches utilisent les triggers afin d'exprimer les contraintes [Mede 91], d'autres par contre dissocient complétement ces deux fonctionnalités. Les contraintes assurent l'intégrité des objets et les triggers servent à réagir à certains événements de niveau plus abstrait (Ode).

Exemple de relation en Troll. L'exemple suivant, tiré de [Jung 93], illustre les relations entre objets proposées en Troll. Cet exemple définit clairement comment un distributeur de billet *Automatic Teller Machine* et une banque communiquent. La relation établit des *relations d'appels (calling relationships)* entre une instance de la classe `Bank` et une instance de la classe `ATM`. La relation d'appel `e1` » `e2` entre deux événements `e1` et `e2` établit que lorsque l'événement `e1` a lieu, l'événement `e2` doit aussi avoir lieu. Cependant, `e2` peut avoir lieu sans que `e1` ait lieu.

```
relationship RemoteTransaction Between Bank, ATM
  interaction
     variables atm:|ATM|; n,p:nat; m:money;
     /* vérification de la carte */
     ATM(atm).check_card_w_bank(n,p) >> Bank.verify_card(n,p,atm);
     Bank.no_such_account(atm) >> ATM(atm).bad_account_msg;
     Bank.bad_PIN(atm) >> ATM(atm).bad_PIN-msg;
     Bank.card_ok(atm) >> ATM(atm).card_accepted;
     /* transaction */
     ATM(atm).issue_TA(n,m) >> Bank.process_withdrawal(n,m,atm);
     Bank.TA_failed(atm) >> ATM(atm).TA_failed_msg;
     Bank.TA_OK(atm,m) >> ATM(atm).dispense_cash(m);
end relationships RemoteTransaction;
```

Discussion.

Parmi ces trois formes de couplage, l'approche qui associe à l'objet une interface relationnelle et celle qui introduit le concept de dépendance sont les plus intéressantes. L'une centralise l'expression de plusieurs dépendances dans un objet, tandis que la seconde localise l'expression de chaque dépendance dans une entité spécifique. Cette dernière approche peut conduire à étendre les fonctionnalités du système comme on l'a montré avec Smalltalk, mais elle ne modifie pas le concept d'objet, tout au plus son implémentation. Il nous semble bien plus facile de programmer des dépendances dans des langages de ce type en permettant de les modéliser localement. A l'inverse en associant une interface relationnelle à l'objet, on met en évidence l'automate qui dans les approches traditionnelles était caché. La robustesse du système semble mieux assurée. Cependant, la description de l'ensemble des dépendances de façon centralisée est plus difficile.

2.2.2 Granularité d'une dépendance

La granularité précise entre quelles entités du langage une dépendance peut être définie : objets dans leur intégralité ou champs. Suivant les langages, les dépendances sont énoncées :

- soit entre les valeurs des champs à l'intérieur d'un même objet (ThingLab [Born 86b]), Tropes [Gens 93], Solver [Puge 93] Ode,
- soit entre les valeurs des champs d'objets différents ou variables (KR [Gius 92], Socle [Harr 86], Tropes, Solver, Prose [Berl 92], Rendezvous [Hill 92, Hill 94]),
- soit entre objets ([Mede 91], PluS [Boua 95], Procol [van 91], Synchroniseur [Frøl 93], LayOM [Bosc 95a], Adaptor[Yell 94], Mediator [Sull 92], Gluon [Pint 93], Sentinel [Chak 92]).

Deux catégories de contraintes dans les bases de données actives existent entre objets (inter-objets) ou sur un seul objet (intra-objet). Dans certains cas, les premières sont spécifiées en utilisant des contraintes intra-objet pour chaque objet impliqué (Ode).

Exemple de granularité entre champs : les formules de KR. Dans KR [Gius 92] qui est un langage à prototypes sur lequel est basé le système Garnet [Myer 90], les dépendances sont exprimées par des formules associées aux champs des objets. Ainsi dans l'exemple ci-après, on définit un objet `Rectangle2` dont le champ `x` a la valeur `34` (ligne 1'). Le champ `y` est défini à l'aide d'une formule : elle spécifie que la valeur du champ `y` sera celle du champ `y` de l'objet dénoté par le champ `left-obj` à laquelle on aura ajoutée `15` (ligne 3).

```
1 (create-instance 'Rectangle1 objet-boîte
2    (:x 10)
3    (:y 20))

1' (create-instance 'Rectangle2 objet-boîte
2'    (:x 34)
3'    (:y (o-formula (+ (gvl :left-obj :y) 15)))
4'    (:left-obj Rectangle1))
```

La valeur du champ `y` de `Rectangle2` est donc dépendante de celle du champ `y` de l'objet `Rectangle1`. Lorsque la valeur de `y` de `Rectangle1` prend une nouvelle valeur, le champ `y` de `Rectangle2` est invalidé. Sa valeur n'est en effet calculée en fonction de cette nouvelle valeur que lors d'un accès en lecture.

2.2.3 Localisation de la définition et de la déclaration d'une dépendance

Les dépendances sont soit définies et *associées* directement (aux classes) des objets sur lesquelles elles agissent (ThingLab, Animus, Procol, Tropes, LayOM, Solver, Ode, SAMOS, HiPAC) soit définies de manière *autonome* en tant qu'entités distinctes (Othelo, Rendezvous, Interactors, Gluons, DSM, Troll, PowerClasses, Scripts[Nier 91], Sophtalk [Clem 91], SAMOS).

SAMOS permet de définir des règles internes ou externes aux classes [Gatz 92]. Les règles internes sont strictement associées à la classe de l'objet. Par contre, il est possible de spécifier des règles externes aux classes. Ces dernières ne sont alors exprimées qu'en utilisant des événements abstraits.

Il existe principalement deux attitudes par rapport à la définition et la déclaration de dépendances : *a priori* ou *a posteriori*.

a priori : les dépendances sont définies lors de la définition de la classe, que le langage propose un mécanisme rudimentaire (valeurs actives) ou élaboré (contraintes de ThingLab, Tropes, Socle, propagateur de contraintes de Procol ou relations de LayOM) pour représenter la dépendance. Elles sont déclarées lors de l'instanciation de ces classes.

a posteriori : les classes sont définies pour elles-mêmes sans tenir compte de possibles dépendances entre leurs instances. Les dépendances sont définies *a posteriori* [Frøl 93, Doda 95, Hill 92, Gens 93, Clem 91, Yell 94, Pint 93] et déclarées ultérieurement entre des instances.

Certaines nuances peuvent cependant être apportées. Par exemple, les relations proposées par [Aksi 94], les ACT pour Abstract Communication Type, sont définies en tant qu'entités autonomes mais dont l'autonomie est amoindrie par la nécessité de spécifier au niveau des classes d'objets la dépendance à laquelle ils participent. Il s'agit là d'une localisation autonome mais nécessitant une attitude *a priori*.

Exemples de définition associée à une classe. ThingLab permet de spécifier au niveau d'une classe des contraintes entre les différentes parties de l'objet représenté par la classe. Une contrainte est décrite par un prédicat et un ensemble de procédures qui peuvent être invoquées pour satisfaire la contrainte. Ainsi les relations de ThingLab sont fortement liées à la structure des objets ; le fait qu'un point soit le milieu de deux autres s'exprime au travers d'une classe définissant une nouvelle notion de segment et non d'une contrainte sur des points.

```
1 Class MidPointLine
2   Superclasses
3     GeometricObject
4   Part Descriptions
5     line: aLine
6     midPoint: aPoint
7   Constraints
8     midPoint = (line point1 + line point2) / 2
9     midpoint <- (line point1 + line point2) / 2
10    line point1 <- midpoint * 2 - line point2
11    line point2 <- midpoint * 2 - line point1
```

Une instance de la classe MidPointLine est composée d'une ligne et d'un point (lignes 5 et 6). La valeur du point est sujette à une contrainte définie par le prédicat (ligne 8) qui indique si les coordonnées de ce point sont en accord avec celles de la ligne. L'ordre de définition des procédures (lignes 9, 10 et 11) indique un ordre décroissant quant à la procédure à appliquer en cas de déclenchement de la contrainte. Il faut noter que la définition même du prédicat et des procédures implique que les composants de l'objet sur lequel porte la contrainte soient directement accessibles au travers d'accesseurs (voir en Annexe B.6).

Ode permet la définition de contraintes associées à la classe des objets. Le fragment de code suivant spécifie qu'un employé ne peut pas gagner plus que son patron. Il s'agit d'une contrainte entre objets : le patron et un employé. Cette contrainte est convertie en deux contraintes intra-objets associées à chacune des classes.

```
class manager;
class employee {  ...
   persistent manager *mgr;        float sal;
public:   ...   float salary() const;
constraint: sal <mgr->salary();   };

class manager: public employee {
   persistent employee *emp<MAX>;
   int sal_greater_than_all_employees();  ...
public: ...
constraint:  sal_great_than_all_employee(); };

int manager::sal_greater_than_all_employees()
{  persistent employee *e;
   for (e in emp) if (e->salary() > salary()) return 0;
   return 1; }
```

2.2.4 Transparence des dépendances

Dans la grande majorité des langages, la présence d'une dépendance est transparente du point de vue des objets : un objet s'adresse à un autre sans se soucier de la présence de dépendances qui seront maintenues automatiquement soit par contrôle des envois de messages (LayOM, Interactors, Othelo [Forn 90a]), soit par contrôle des accès aux variables (Rendezvous [Hill 94]). Cependant, il existe des langages dans lesquels la dépendance doit être explicitement invoquée pour être maintenue. La notion de groupe de STklos [Gall 96], qui permet de propager automatiquement un message à tous les objets d'un groupe, implique que les messages soient envoyés au groupe. Un objet du groupe peut recevoir directement un message mais dans ce cas celui-ci n'est pas propagé aux autres objets du groupe. De même, la présence d'un *gluon* [Pint 93] qui sert de médiateur entre un objet fournisseur et des clients n'est pas transparente pour les clients. Un client doit s'adresser au gluon pour obtenir les services offerts par la présence d'une relation entre le client et ses fournisseurs.

2.2.5 Respect de l'encapsulation des langages à objets

Une dépendance respecte l'encapsulation des objets qu'elle lie si elle est exprimée dans les termes de l'interface de ces objets. C'est le cas des ACTs de [Aksi 94], des propagateurs de contraintes de Procol [van 91], des Interactors [Doda 95], des relations de Troll ou des Synchroniseurs [Frøl 93] qui sont uniquement exprimés en termes des interfaces des objets qu'ils contraignent. Les dépendances qui font intervenir directement les champs d'autres objets en n'utilisant pas d'accesseurs violent quant à elles l'encapsulation. Notons que ce critère ne s'applique que pour les langages à objets qui encapsulent les données. En particulier, il ne s'applique pas à la plupart des langages de frames. SAMOS porte une attention particuliere au respect de l'encapsulation. Ainsi il est possible de définir des règles internes aux objets[Gatz 92]. Ces règles peuvent manipuler l'état de l'objet et sont strictement internes à celui-ci. Par contre, il est possible de définir des événements abstraits que des régles externes peuvent utiliser. Cette solution est réellement bien adaptée à l'utilisation des règles réactives dans des bases de données orientées objets.

Un exemple de propagateur de contraintes en Procol. Les auteurs de Procol ont souhaité une intégration de contraintes qui n'implique pas une modification du code des objets contraints et respecte le principe de l'encapsulation. *« Propagators can be used to maintain relationships between objects, or to visualize non-visual algorithms, or to simply monitor all accesses to a particular object. »* [van 91]. Ils proposent des contraintes mono-directionnelles exprimées, non pas en termes de changement de valeurs des variables, mais en termes de méthodes. A chaque

fois qu'un objet reçoit un message sur lequel porte une contrainte, le code associé à cette contrainte est exécuté.
Ainsi la contrainte suivante : `constraint Point1.Move(x, y) -> Point2.Move(x + 5, y );` exprime qu'à chaque fois que l'objet Point1 reçoit le message `Move`, l'objet `Point2` reçoit le message `Move` de telle sorte qu'il soit aligné à la droite de `Point1` à une distance de `5` unités. Les contraintes de Procol permettent d'animer des algorithmes à la manière de la visualisation d'algorithmes de Animus [Born 86b].

L'exemple suivant permet de visualiser un tableau d'entiers, nommé `array`, par un objet de visualisation nommé `bargraph`.

```
Obj      DEMO (INTARRAY array)
Declare  BARGRAPH bargraph;
         int i, val;  Init  new bargraph;
         constraint
         array.SetValue(i,val) -> bargraph.SetValue(i,val);
EndObj   DEMO.
```

2.2.6 Concept d'objet et dépendances : évaluation

L'intégration de dépendances dans le modèle objet n'est pas anecdotique. Selon les approches, le modèle objet est modifié pour intégrer une dimension *relationnelle*. Il se pose alors différentes questions quant au respect des grands principes objets : encapsulation et modularité. L'encapsulation des données est violée par l'expression directe de contraintes entre les champs d'objets différents. Se pose aussi le problème de l'atomicité des classes lorsque les dépendances sont exprimées *a priori* (au niveau des classes). L'ensemble des éventuelles futures dépendances entre les objets apparaît alors dès la définition des classes qui se trouvent ainsi polluées par des informations extrinsèques. Par contre, le modèle objet est peu modifié par l'introduction de mécanismes tels que les valeurs actives. Ceux-ci permettent de modéliser les dépendances champs à champs et ont prouvé leur efficacité dans la réalisation d'un grand nombre d'applications en particulier celui des interfaces graphiques. Cependant, la programmation et le maintien des systèmes qui les utilisent pour maintenir des dépendances complexes entre objets ne sont pas simples et la réutilisabilité des objets en est affectée.

2.3 Modèles de dépendances

Nous avons vu au paragraphe précédent la grande diversité des sens que l'on accorde aux termes «relations» ou «dépendances». Nous retrouvons cette diversité dans les modèles de dépendances qui ont été définis. Après avoir situé les dépendances par rapport aux objets et aux classes, nous abordons dans cette partie une description plus précise des dépendances elles-mêmes. Selon le statut octroyé à la dépendance, les propriétés et la sémantique qu'elle véhicule, nous montrons que sa modélisation est très différente.

2.3.1 Statut des dépendances

Nous distinguons quatre statuts pour les dépendances allant d'une absence totale d'abstraction vers un statut réel des dépendances. Les dépendances sont : dispersées, factorisées, conceptuellement perçues comme des entités à part entière ou prédominantes.

Dépendances dispersées.

La définition d'une dépendance peut être dispersée dans les classes : sa structure est alors mêlée à celle des objets, son comportement dynamique est mis en oeuvre dans les méthodes de différentes classes et sa sémantique est parfois seulement écrite dans les commentaires. Lore [Beno 89] fait partie de cette catégorie, les relations en Lore étant soit des accesseurs pour les relations unaires, soit des méthodes pour les relations n-aires. Cette vision des méthodes est d'ailleurs à rapprocher des fonctions génériques de CLOS [Stee 90] qui pourraient elles aussi être vues comme des relations entre plusieurs classes. Nous considérons que les formules proposées dans KR [Gius 92] ou les valeurs actives [Stef 86b, Ilo 91] ou les démons démons [Bout 93] appartiennent à cette catégorie. Ces langages proposent un mécanisme réactif associé aux classes similaire aux méthodes auxiliaires `:after` de CLOS.

Dépendances factorisées.

Certains langages, sans définir d'entités représentant la dépendance, offrent des mécanismes de séparation entre les dépendances (structures et fonctionnalités) et les objets mis en relation. C'est le cas des contraintes événementielles *(trigger constraints)* de Animus, propagateurs de contraintes de Procol.

Exemple de dépendance factorisée. Dans Animus [Born 86b], les contraintes événementielles sont utilisées pour spécifier comment les animations évoluent en réponse aux événements reçus par les objets qu'elles représentent. De telles contraintes sont basées sur la définition d'actions à effectuer en réponse à la réception par l'objet contraint d'un message dont le sélecteur est spécifié dans la contrainte de l'objet. L'exemple ci-après illustre ces contraintes.

Il s'agit d'animer une liste lors du tri de ses éléments. Pour cela, le système fait clignoter les éléments d'un tableau lorsqu'ils sont sujets à une comparaison et visualise leur permutation. Ainsi la définition de la contrainte à la ligne 7 exprime le fait que lorsqu'une instance de `SortQueue` reçoit un message `compare:with:`, la contrainte fait clignoter les éléments comparés de cette instance. De même, la ligne 10 exprime que la réception d'un message `swap:with:` nécessite la mise en oeuvre d'une animation montrant la permutation des deux éléments.

```
1  Class SortQueue
2    Superclasses
3      AnimusObject
4    Part Descriptions
5      list: anOrderedCollection
6    Constraints
7      list triggersOn: #compare:with:
8           causing: (Flasher at: a1
9                     Flasher at: a2)
10     list triggersOn: #swap:with:
11          causing: (Trajectory from: a1 to: a2
12                    Trajectory from: a2 to: a1)
```

Dépendances considérées comme des entités distinctes.

Les dépendances entre objets peuvent avoir un rôle aussi important que celui des classes, ceci que les relations soient réifiées (Rendezvous, Tropes, LayOM, ACT, PluS, Solver, Prose, IlogPowerClasses, Gluon SAMOS, AMOS, ACOOD [Bern 92], Sentinel [Chak 92], Cobras [Vaut 96], HiPAC [Daya 88]) ou non (Synchroniseur, Mediator [Sull 92], Adaptor, Contract). Le fait de considérer les relations comme des entités spécifiques a plusieurs avantages [Rumb 87, Shah 89, Blah 95, Bosc 95a, Frøl 93] : les objets liés ne représentent plus que le concept qu'ils modélisent ; leur code est donc plus facilement modifiable et réutilisable. De plus, cette abstraction permet de définir clairement une dépendance, de composer et de réutiliser ces abstractions.

Il faut noter, dans le cas des dépendances de Smalltalk, une tentative de réification des dépendances. VisualWorks [Brif 96, Par 94] propose d'émettre un message sur un objet cible lors de la réception d'un message par un objet source grâce aux `DependencyTransformer`.

Exemple : les synchroniseurs. Agha et Frølund proposent des synchroniseurs dans [Frøl 93] (voir en Annexe D). Un distributeur automatique est composé de plusieurs parties : un monnayeur qui accepte de l'argent et des compartiments contenant, par exemple, des fruits. Ces différents composants doivent être contraints afin d'offrir le comportement attendu : il faut mettre assez d'argent pour pouvoir avoir un fruit ; une fois le fruit pris l'argent inséré n'est plus comptabilisé pour un choix ultérieur ; si l'on appuie sur un bouton spécial du monnayeur, l'argent inséré est rendu. Si l'on modélise chacune des parties de cet appareil par un objet distinct la contrainte de fonctionnement est énoncée par un synchroniseur.

```
1 VendingMachine(accepter,apples,bananas,apple_price,banana_price)
2   init amount := 0
3    amount < apple_price    disables apples.open,
4    amount < banana_price disables bananas.open,
5    accepter.insert(v) updates amount := amount + v,
6    (accepter.refund or apples.open or bananas.open)
7                   upates  amount := 0
```

Le synchroniseur nommé `VendingMachine` permet de contraindre le monnayeur : l'objet `accep -ter` et les compartiments : les objets `bananas` et `apples`. Des variables supplémentaires sont passées comme arguments du synchroniseur : le prix associé à chaque compartiment (ligne 1). Le synchroniseur `VendingMachine` définit une variable locale `amount` qui représente l'argent que l'utilisateur a mis dans le monnayeur (ligne 2). Les lignes 3 et 4 spécifient que la méthode `open` d'un compartiment ne peut être activée que lorsque la somme insérée dans la machine est supérieure ou égale au prix associé au compartiment. La ligne 5 spécifie que la valeur de la variable `amount` reflète constamment la somme insérée dans le monnayeur. Les lignes 6 et 7 sont complémentaires par rapport à la ligne 5, elles assurent que la valeur de la variable `amount` reflète bien la somme d'argent insérée depuis le dernier choix ou la dernière demande de remboursement.

Dépendances prédominantes.

Finalement les dépendances peuvent être prédominantes par rapport aux objets qu'elles mettent en relation. Cette situation apparaît exclusivement dans Contract [Helm 90].

Les auteurs de Contract [Helm 90] proposent de mettre l'accent sur la définition des interactions entre objets, les contrats, puis de définir les classes par leur conformité aux rôles spécifiés par les contrats. La spécification d'une dépendance à l'aide d'un contrat nécessite la donnée des objets participants à la composition, les obligations contractuelles *(contractual obligations)* de chacun d'entre eux, les invariants et les méthodes qui instancient le contrat. Les obligations contractuelles se composent d'une part d'*obligations de type* de la part des participants : un participant doit avoir certaines variables et méthodes et d'autre part d'*obligations causales* : à la réception de certains messages, un participant doit effectuer une séquence d'actions, en particulier des envois de messages aux autres participants, pour vérifier les invariants (voir en Annexe C). Nous illustrons les différents aspects d'un contrat en commentant la définition du contrat `SubjectView` tiré de [Helm 90].

`SubjectView` spécifie les différentes actions que les objets participants doivent satisfaire afin de maintenir la cohérence entre un objet, `Subject`, et les différentes vues, `Views`, qui sont susceptibles de le représenter.

Dans le contrat `SubjectView`, pour les obligations de types, l'objet `Subject` doit posséder une variable d'instance `value` de type non spécifié `Value` et par exemple une méthode `Notify()` (lignes 2 et 5). Ainsi l'expression `Draw()` $\mapsto$ `Subject()` $\rightarrow$`GetValue()` à la ligne 11 de la figure 2.1 spécifie que chaque vue doit à la réception du message Draw() se comporter d'une *certaine manière* qui mène à l' envoi de message `GetValue()` à l'objet `Subject`.

Le mot clé **`invariant`** permet la définition de l'invariant du contrat. Ainsi pour le contrat `SubjectView` (ligne 14), l'expression *Subject.SetValue(val)* est l'action qui conduit à la satisfaction de l'invariant. Pour être complet un contrat doit spécifier, à l'aide du mot clé **instantiation**, des *préconditions* sur les différents *participants* ainsi que les actions le mettant en oeuvre. Ainsi

```
    contract SubjectView
1       Subject supports [
2           value : Value
3           SetValue(val:Value)↦ Δvalue{value = val};Notify()
4           GetValue() : Value ↦ return value
5           Notify() ↦ ⟨‖ v : v ∈ Views : v→ Update()⟩
6           AttachView(v:View) ↦ {v ∈ Views}
7           DetachView(v:View) ↦ {v ∉ Views}
8           ]
9       Views : Set (View) where each View supports [
10          Update() ↦ Draw()
11          Draw() ↦ Subject() → GetValue() {View reflects Subject.value}
11          SetSubject(s:Subject) ↦ {Subject = s}
12          ]
13      invariant
14          Subject.SetValue(val) ↦ ⟨∀v : v∈ Views : v reflects Subject.value⟩
15      instantiation
16          ⟨ ‖ v : v ∈ Views :
17          ⟨Subject → AttachView(v) ‖ v → SetSubject(Subject)⟩⟩
    end contract
```

Figure 2.1: Définition du contrat SubjectView

un contrat `SubjectView` est opérationnel lorsque les méthodes `AttachView` et `SetSubject` sont exécutées avec des arguments appropriés (lignes 16 et 17).

2.3.2 Expressivité de la dépendance

L'expressivité d'une dépendance décrit son comportement dynamique. Selon les langages, l'expression de ce comportement est plus ou moins riche.

- Pour les langages ayant une expressivité que nous qualifions de «faible» [Gius 92, Boua 95, Stef 86b, Bout 93], le comportement dynamique d'une dépendance se limite à une règle : un message ou un changement de valeur d'un champ implique d'envoyer d'autres messages ou de calculer d'autres valeurs.
- Dans d'autres langages, une dépendance peut regrouper plusieurs règles (Rendezvous, synchroniseur). Ces groupements permettent de définir des dépendances dont la sémantique est non seulement plus riche mais plus claire qu'un ensemble de règles prises une à une. Pour le cas particulier des contraintes exprimées sous forme de prédicat (ThingLab, Tropes, Prose...), le groupement de plusieurs contraintes est généralement une conjonction de contraintes.

Classifier les bases de données orientées objets actives est délicat [Geha 92, Daya 88, Daya 96, Mede 91, Gatz 92, Bern 92]. Les contraintes d'intégrité sont définies à l'aide de règles ECA ou transformées en celles-ci. Une règle ECA (Evénement, Condition, Action) spécifie l'action A à exécuter lorsque l'événement E s'est produit et que la condition C est vérifiée. Nous avon sdonc différents niveaux d'abstraction. D'autre part, les règles ECA peuvent avoir des sémantiques très riches (incluant comme dans HiPAC diverses données indiquant leur type, le moment de leur exécution ...). On retrouve cette problématique dans Cobras qui utilise des règles ECA [Vaut 96].

2.3.3 Représentation d'une dépendance par un objet de première classe

« the relation as a semantic construct in an object-oriented language clearly expresses associations and constraints among objets which would otherwise be buried in implementation code »

[Rumb 87].

Les dépendances, en particulier quand elles expriment des interactions entre objets, présentent un *potentiel d'abstraction*. La réification[1] des dépendances est alors non seulement un choix d'implémentation mais aussi une façon d'expliciter ce potentiel. Ainsi, au delà de raisons purement pragmatiques, le programmeur modifie sa façon de penser en s'adaptant aux moyens d'expression que le langage lui offre. La citation suivante résume cela parfaitement : *« making relations a first-class semantic construct affects a programmer's way of thinking about a problem from the design stage all the way through to the coding »* [Rumb 87]

Du point de vue de l'implémentation, la réification des dépendances offre des avantages du même ordre que ceux issus de la réification d'autres aspects des langages à objets (variables d'instances, méthodes en CLOS) : manipulation explicite des données, factorisation de l'information, séparation claire par rapport aux classes, description de la sémantique du comportement, ...(Troll Rendezvous [Hill 94]). Ainsi, les bases de données orientées objets actives comme [Mede 91], SAMOS [Gatz 92], ACOOD [Bern 92] ou HiPAC[Daya 88] réifient les contraintes d'intégrité ou les règles de production (Event Condition Action) : ils associent des informations propres à ces entités (priorité, couplage, status, ...). De même, en génie logiciel, les dépendances entre classes peuvent être réifiées pour faciliter leur vérifications, comme les assertions d'Eiffel [Coll 96a, Coll 96b] ou les deltas d'évolutions[Bris 95]. Pour J. Rumbaugh, la description d'une association inclut son degré, sa cardinalité et la liste de ses champs. Par exemple, si l'on considère qu'une personne ne peut pas cumuler plusieurs emplois, l'association `Travail_pour` est une association de cardinalité *plusieurs-à-un* de personnes vers société : plusieurs personnes travaillent pour une seule société. Lorsque les dépendances sont réifiées, des propriétés telles que la cardinalité peuvent être vérifiées ce qui est plus difficile dans les autres modèles de représentation.

Un exemple de réifications des dépendances. Illustrons les dépendances réifiées intégrant un comportement dynamique du langage Rendezvous basé sur le modèle architectural ALV (Abstraction/Link/View) [Hill 92, Hill 94] (voir au chapitre 11). Un jeu de Tic-Tac-Toe est composé d'une partie applicative (partie Abstraction) complètement autonome, de deux représentations personnalisées (partie View) et de liens gérant la cohérence entre ces différentes parties. Dans l'exemple ci-après, la dépendance, définie aux lignes 4 et 5, assure que la valeur de la variable d'instance `state` de l'objet pointé par `view` est égale à celle de l'objet pointé par `abstraction`, et inversement. La dépendance `TTTCellPositionLink`, définie aux lignes 8 à 13, gère la cohérence entre une cellule et sa représentation. Elle calcule les coordonnées de la représentation en fonction de la colonne et de la rangée de la cellule.

```
1 (defRVClass TTTCellLink (Link)
2    (:add-dependencies
3        TTTCellPositionLink
4        (TTTCellStateLink = (state (view self))
5                            (state (abstraction self)))))
6
7 (defDependency TTTCellPositionLink (TTTCellLink)
8     (setf (xOffsetFromParent (view self))
9           (+ 2.0 (* 20.0
10                (float (:source (column (abstration self)))))))
11    (setf (yOffsetFromParent (view self))
12          (+ 2.0 (* 20.0  13
13                 (float (:source (row (abstraction self))))))))
```

[1] Littéralement le terme «réification» est un terme philosophique synonyme de «transformation en chose». Dans notre contexte informatique et plus précisément dans le contexte de la programmation objet, le terme réification désigne l'ensemble des techniques rendant explicite sous forme d'objets certaines parties jusqu'alors implicites ou non accessibles du langage. La réification peut s'appliquer aussi bien aux aspects structurels du langage : classes, méthodes, champs...qu'aux aspects dynamiques : pile d'exécution, contexte d'évaluation...

La mise en oeuvre d'un tel lien se fait par l'instanciation de la classe `TTTCellLink` : supposons que nous disposions d'un objet cellule `AbstractCell-151` et de sa représentation `GraphicCell-24`, le code suivant illustre la création d'un lien.

```
(create 'TTTCellLink :abstraction AbstractCell-151 :view GraphicCell-24)
```

2.3.4 Héritage de dépendances

Le fait de considérer les dépendances comme des entités renforce la question de l'incrémentalité : quels mécanismes utiliser pour définir de nouvelles dépendances à partir de dépendances existantes ? Lorsque les dépendances sont réifiées, il est naturel d'utiliser l'héritage pour définir de nouvelles relations par spécialisation ou par composition. Cependant, peu de langages offrent cette possibilité. L'exemple précédent avec RENDEZVOUS présente un cas d'héritage par addition de comportements réactifs. CONTRACT offre deux opérations permettant l'expression de contrats complexes en termes de contrats plus simples : l'inclusion et l'affinage *(refinement)*. L'inclusion définit des contracts par la composition de sous-contrats. L'affinage permet une spécialisation des obligations contractuelles et des invariants. Lorsque certaines dépendances sont associées aux classes comme c'est le cas des contraintes ou des règles des bases de données orientées objets [MEDE 91], l'héritage entre classes est étendu pour prendre en compte ces nouvelles entités.

2.3.5 Modèle de dépendances : évaluation

Il existe plusieurs modèles de dépendances. Ceux qui réifient ce concept présentent de nombreux avantages dont l'essentiel est de donner un support sémantique à la dépendance. Propriétés statiques et dynamiques sont alors factorisées dans une même entité.

En donnant un même statut aux objets et aux dépendances, il n'en demeure pas moins difficile dans certains cas de distinguer ce qui est de la connaissance relationnelle et ce qui est de la connaissance intrinsèque. Ainsi HARRISON et OSSHER dans [HARR 93] montrent l'importance du contexte dans lequel un objet est défini pour la définition de son interface. Nous ne saurions répondre à des questions aussi difficiles. L'importance grandissante donnée à l'expression des dépendances dans les langages à objets devrait permettre d'apporter des solutions à la question du statut de l'objet dans un monde dans lequel les relations entre objets sont explicites.

2.4 Modèles de maintien de cohérence

L'intégration des dépendances dans les langages à objets nécessite de prendre en considération les problèmes de maintien de la cohérence des objets en relation. Ce maintien est géré par le système lorsque celui-ci offre au programmeur les moyens (réactions aux messages, gardes, formules, etc) d'exprimer comment conserver cette cohérence. Cette partie développe différentes techniques de maintien de cohérence des dépendances utilisées.

2.4.1 Points de contrôle

Nous avons distingué deux situations :

Contrôle des changements de valeurs : lorsque le changement de valeur d'un champ remet en cause la cohérence d'une dépendance, le système propage de proche en proche ce changement de valeurs aux champs dont la valeur dépend du précédent (KR, SOCLE, TROPES, THINGLAB, SOLVER). Bien que les contraintes de THINGLAB soient énoncées en termes d'envoi de messages, nous l'associons à cette catégorie car ses contraintes sont des expressions toujours écrites par rapport aux valeurs des champs : une contrainte spécifiant comment remettre à jour les champs avec une valeur consistante par rapport à la contrainte.

Contrôle des messages : lorsqu'un objet reçoit un message qui remet en cause la cohérence d'une dépendance, le système réagit par l'émission d'un ou de plusieurs autres messages. Les messages ici ne sont pas limités aux accesseurs des champs des objets : il s'agit de n'importe quelle méthode d'un objet (ANIMUS, PROCOL, PLUS, Interactor, ACT, TROLL). Cette propagation est souvent décrite sous forme de règles (message → réactions). Les liens de RENDEZVOUS sont activés par changement de valeurs et la plupart du temps propagent une nouvelle valeur dans les champs dépendants, cependant ils permettent également d'émettre des messages tels que la création d'objets.

2.4.2 Maintien de cohérence : sémantique du comportement dynamique des dépendances

Suite à une modification d'objets (modification de variables ou envoi de messages), la cohérence des relations se trouve perturbée. Selon les systèmes, différentes sémantiques de rétablissement de la cohérence sont alors proposées.

Réactivité immédiate.

Les valeurs actives et les démons sont une première approche de réactivité immédiate. La propagation est alors dirigée par les données et en profondeur d'abord. En SMALLTALK, un protocole gère le comportement dynamique des dépendances. Il s'agit principalement des méthodes : `changed` et `update:`. Le mécanisme de propagation automatique de SMALLTALK est simple ; pour informer les dépendants d'un changement d'état, l'objet modifié s'envoie le message `changed`, ce qui a pour conséquence d'appliquer la méthode `udpate:` sur ses dépendants. Une dépendance est alors une relation orientée de l'objet influant vers les objets dépendants. Ces différentes phases doivent être gérées par le programmeur (voir en Annexe A).

Réactivité calculée.

Quelques systèmes intégrant objets et dépendances (plus précisément les contraintes entre valeurs de champs) maintiennent la cohérence des contraintes par planification. Dans THINGLAB ou RENDEZVOUS, la satisfaction des contraintes statiques se décompose en deux phases. La première consiste à planifier les modifications de valeurs, c'est-à-dire à déterminer une suite d'actions à exécuter afin de resatisfaire le graphe de contraintes de façon optimale, puis à stocker cette planification en compilant à la volée une méthode qui sera invoquée pour resatisfaire les contraintes. La planification essaye de minimiser le code à exécuter ; pour cela les auteurs de THINGLAB utilisent une technique de propagation d'états connus et de propagation de degrés de liberté. Lorsque la planification est impossible pour cause de cycles, THINGLAB utilise une méthode numérique itérative de relaxation [BORN 86b]. La seconde phase quant à elle utilise simplement cette planification : il s'agit alors d'invoquer la méthode, compilée lors de la précédente phase, sur l'objet dont la contrainte doit être satisfaite. Cette planification s'effectue en THINGLAB ou RENDEZVOUS lors du changement de valeur d'une des variables.

Gardes.

Certains langages gèrent le maintien de cohérence des dépendances en utilisant la notion de garde : une condition devant être satisfaite pour qu'un message soit accepté (PROCOL, Synchroniseurs). Ainsi en SMALLTALK, il est possible de spécifier des gardes avec la méthode `changeRequest`. Lorsqu'un objet influant veut changer, il interroge ses dépendants par le biais de la méthode `updateResquest` pour savoir s'il en a le droit. Les gardes d'un synchroniseur [FRØL 93] sont exclusivement exprimées en fonction de l'état de celui-ci et non des objets qu'il contrôle. Notons cependant que PROCOL et les synchroniseurs se situent dans un monde concurrent et offrent des contraintes de synchronisations pour la coordination des messages plus élaborées que la notion de garde présentée ici.

2.4.3 Terminaison et déterminisme

La terminaison des programmes contenant des dépendances consiste à savoir si les systèmes gèrent les cycles en les détectant soit statiquement, soit dynamiquement. Le déterminisme précise, lorsque plusieurs dépendances aboutissent à un même objet, que l'ordre d'exécution des dépendances n'influe pas sur le résultat final.

A l'exception des langages à base de contraintes comme ThingLab, Rendezvous ou KR les autres langages ne proposent pas de gestion des cycles : ni détection statique, ni dynamique. La détection de cycles est à la charge du programmeur.

ThingLab détecte les cycles statiquement et rejette de telles situations. ThingLabII, basé sur l'algorithme SkyBlue, permet de construire des graphes cycliques. SkyBlue[Sann 93] ne peut satisfaire les contraintes impliquées dans un cycle mais il maintient correctement les contraintes acycliques définies ailleurs dans le graphe.

Rendezvous assure la terminaison des programmes mais n'est pas déterministe. Il ne parcourt les cycles qu'une seule fois et ignore simplement les situations de confluence ; c'est-à-dire les cas dans lesquels une variable est déterminée par de multiples contraintes. En général, la valeur de la variable est celle de la dernière contrainte évaluée ayant cette variable comme cible. Cette valeur dépend donc de l'ordre d'évaluation des variables. Les auteurs de Rendezvous signalent alors que bien que la valeur de la variable soit inconsistante vis à vis des autres contraintes, il plus judicieux dans le cadre de l'implémentation des interfaces de ne pas traiter ces cas comme des erreurs [Hill 93a].

KR gère les cycles mais est non-déterministe. En effet, lorsque le système cherche par propagation locale la valeur d'une variable et qu'il détecte un cycle la valeur sera celle précédemment calculée. Ce faisant, cette valeur dépend de l'endroit à partir duquel le calcul a commencé [Gius 92].

Les langages à base de contraintes de part la nature des objets (des variables) sur lesquels portent ces contraintes et des dépendances, souvent des contraintes arithmétiques, offrent en général la terminaison des programmes. Elle est rendue possible soit en détectant statiquement la présence de cycles et en les rejetant (ThingLab), soit en les gérant dynamiquement par la présence de prédicats établissant si la propagation doit se poursuivre ou pas, soit en les ignorant temporairement (ThingLabII et Rendezvous).

Le problème du déterminisme des programmes reste encore ici un réel problème. Ainsi malgré des algorithmes de planification qui à notre sens essayent de décrire des solutions globales à partir de définitions locales et le fait que les contraintes manipulent des méthodes atomiques (des méthodes dont le comportement est connu et élémentaire), le déterminisme n'est pas assuré. Ainsi ThingLab ne traite pas les cas de confluence et Rendezvous propose des solutions qui dépendent de l'ordre d'évaluation des contraintes. Il faut noter que ThingLabII permet d'associer des priorités aux contraintes qui sont des indications utilisées lors de la planification pour le système afin de trouver la solution globale la plus satisfaisante.

2.4.4 Modèles de maintien de la cohérence : évaluation

L'objectif des différents mécanismes présentés dans cette partie est toujours un maintien de la cohérence des dépendances. Un problème sous-jacent est la cohérence globale d'un graphe de dépendances (voir figure 1.1). En effet, l'expression des dépendances est locale aux objets liés. Lorsque la propagation se produit, il ne s'agit plus de maintenir une dépendance, mais un ensemble de dépendances locales. Cette globalisation du maintien de la cohérence du système est assez bien gérée dans les systèmes où la planification est possible, mais présente une sémantique beaucoup moins claire lorsque la réactivité est immédiate. Le programmeur est alors confronté à différents problèmes tels que : le non déterminisme des solutions obtenues qui dépendent du chemin choisi pour le rétablissement de la cohérence, l'apparition de cycles de causalité, des situations de blocage pour lesquels le système ne sait plus rétablir la cohérence. Bien sûr des règles pour remédier à l'apparition des cycles peuvent être édictées telles que la non réentrance d'une dépendance. Mais celles-ci n'ont de validité que tant que la stabilité ne doit pas être retrouvée par recherche d'un point fixe. Le problème du non-déterminisme est encore plus difficile à appréhender. La solution d'une réactivité contextuelle semble à ce jour n'apporter que des solutions *ad hoc*. Par contre les systèmes de contraintes en limitant le langage des dépendances peuvent assurer des propriétés du réseau des dépendances telles que la terminaison. L'ensemble du graphe est considéré comme une boîte noire dont le programmeur perçoit seulement le comportement extérieur. Toute la difficulté est alors encore une fois de bien comprendre le processus global de résolution afin de savoir exprimer localement les contraintes de telle sorte que le système sache les résoudre globalement. Les travaux sur Esterel [Bous 91] ou Lustre [Halb 91] introduisant le concept d'instant détiennent peut-être une part de la solution.

2.5 Conclusion de la synthèse

Dans ce chapitre, nous avons étudié différentes approches qui introduisent les dépendances dans les langages à objets. Certaines étendent l'interface des objets en leur associant une interface relationnelle qui précise le comportement dynamique de l'objet. Dans ce contexte, l'expression d'une dépendance consiste à encapsuler les objets liés dans un autre objet qui présente à son tour une interface relationnelle. En poussant plus loin cette démarche, on obtient des systèmes hiérarchiques d'objets au sens d'une subordination des uns par les autres ce qui correspond à un contrôle centralisé. Une autre approche s'intéresse davantage à la coopération entre objets en la décrivant dans des entités de dialogue ce qui permet une expression décentralisée des dépendances et peut conduire à une décentralisation du maintien de la cohérence. Le graphe des dépendances peut alors évoluer dynamiquement. Cette approche plus naturelle présente aujourd'hui l'inconvénient d'un maintien de la cohérence globale difficile à contrôler. Seuls les systèmes dont l'algèbre des contraintes est bien connue offre des éléments de preuve. A notre sens ces deux approches sont complémentaires et l'avenir est probablement à une écriture délocalisée des dépendances telle que le propose la seconde approche et à une compilation à terme au sens de la première approche.

En guise de conclusion, nous donnons un tableau récapitulatif dans lequel sont classifiés les principaux langages sur lesquels a porté notre présentation. Les critères examinés dans ce tableau ont été détaillés dans les différents paragraphes de ce chapitre.

Le chapitre suivant discute de la valeur de chacun de ces critères dans le contexte du modèle ObjVlisp. Nous définissons ainsi un premier cadre pour l'intégration de dépendances dans un modèle à classes.

Langage Relation	*Granularité*	*Statut et position*	*Comportement*	*Encapsulation*	*Spécificité*
LOOPS[STEF 86b]	EC	Factorisée: OI AV	PV	oui/non	messages en réaction
KR[GIUS 92]	EC	Dispersée: AV	PV	non	évaluation paresseuse
PluS[BOUA 94]	EO	Entité: OI	PM	oui	
RendezVous[HILL 92]	E0	Entité: OI	PVR	non	messages en réaction
ThingLab[BORN 86b]	ECO	Factorisée: DC	PVR	oui/non	Planification
Animus[BORN 86b]	EO	Factorisée: DC	PMR	oui/non	
ACT[AKSI 94]	EO EC	Entité:OI/AC	PMR	oui	Distribution
Interactor[DODA 95]	EO	Entité: OI	PMR	oui	
Procol[VAN 91]	EO	Factorisée:DC	PM	oui	protocoles explicites
Synchroniseurs[FRØL 93]	EO	Entité: I	PMR	oui	opérateur d'atomicité
Tropes[GENS 93]	EC/ECO	Entité:OI/AC	PVR	non	intégré à un LRC
Solver[PUGE 93]	EC/ECO	Entité:OI/AC	PVR	non	satisfaction
PROSE[BERL 92]	EC	Entité: OI	PVR	non	interfaçable à tout LRC
Socle[HARR 86]	EC/ECO	Entité: I/AC	PVR	non	
Adaptor[YELL 94]	EO	Entité: I	PMR	oui	protocole explicite
Gluon[PINT 93]	EO	Entité: OI	PM	oui	médiateur explicite
OTHELO [FORN 90a]	EO	Entité: OI	PM	oui	
ODE [GEHA 92]	EO/ECO	Factorisée: AC	PM	oui/non	contraintes et triggers
SAMOS [GATZ 92]	EO/ECO	Entité: AC/I	PM	oui	Règles internes ou externes
HIPAC [DAYA 96]	EO/ECO	Entité: AC	PM	oui/non	

Granularité : EO = entre objets ; EC = entre champs ; ECO = entre champs d'un même objet
Dépendance : AV = associée aux variables d'instances ; AC = définie dans la classe ; I = indépendante ; O = Objet ; R = riche
Comportement : PV = propagation de valeurs ; PM = propagation de messages

Tableau 2.1: Tableau de synthèse des principaux langages intégrant des relations

3

Proposition d'une intégration dans le modèle ObjVlisp

Ce chapitre est le prolongement direct de la synthèse présentée au chapitre précédent. Dans la synthèse, des critères nous ont permis de caractériser les dépendances de travaux existants. Les valeurs de ces critères découlent très souvent du modèle objet choisi. Nous présentons maintenant rapidement le modèle objet qui sert de base à notre travail : le modèle ObjVlisp. Au vu de ce modèle, nous étudions les valeurs des critères énoncés lors de la synthèse. Ce faisant, nous énonçons les caractéristiques que doit possèder une intégration de dépendances dans un langage à classes de type Smalltalk ou ObjVlisp. Ces caractéristiques seront prises en compte dans notre modèle aux chapitres 5 et 6.

Le plan suivi pour cette description est le suivant : nous abordons tout d'abord la place des dépendances par rapport aux classes, ensuite les propriétés mêmes des dépendances et nous terminons par le maintien de la cohérence des dépendances.

3.1 Choix d'un modèle objet : le modèle ObjVlisp

« *The only protocol to activate an object is message sending : a message specifies which procedure to apply (denoted by its name, the selector), and its argument.* » [Coin 87].

Un des objectifs de cette thèse est d'introduire des dépendances dans des langages existants et non de définir un nouveau langage, donc le choix d'un modèle objet a des répercussions sur l'intégration de dépendances et définit le contexte de cette thèse. Les modèles objets tels que ObjVlisp [Coin 87] et plus généralement celui de Smalltalk ont fait leur preuve. Leur perception d'un objet comme une entité communiquant exclusivement par envoi de messages et les facilités qu'ils offrent pour contrôler ou étendre le langage nous a amené à les choisir comme base à notre travail.

Le modèle objet est décrit par cinq postulats [Coin 87] :

P1 : Un objet représente des données et des procédures pour manipuler ces données. Un objet se définit par son comportement et non par sa structure [Nier 87]. Cette vision de l'objet s'inscrit complètement dans une vision comportementale de celui-ci : un objet est une boîte noire et n'est manipulable que par l'utilisation exclusive de son interface. L'encapsulation des données est fondamentale.

P2 : L'unique moyen de communication entre objets est l'envoi de message. Un message précise à l'aide d'un sélecteur quel comportement de l'objet doit être activé.

P3 : Tout objet est instance d'une classe qui spécifie sa structure sous forme de *variables d'instances*, appelées aussi champs ou attributs, et son comportement sous forme de *méthodes*. Toutes les instances d'une même classe possèdent la même structure et partagent le même comportement.

P4 : Bien que des problèmes de compatibilité subsistent entre les méta-classes [GRAU 89, DANF 94a, MULE 95a], nous avons choisi un modèle de méta-classes explicites à la manière de CLOS ou d'OBJVLISP, contrairement aux méta-classes implicites de SMALLTALK, pour des raisons de souplesse et de factorisation du comportement des méta-classes [COIN 87]. Une classe est ainsi un objet, instance d'une autre classe appelée sa méta-classe.

P5 : L'héritage permet de définir incrémentalement des classes à partir d'autres classes. Nous n'associons aucune sémantique de typage à l'héritage et le considérons comme un mécanisme de partage d'informations (variables d'instances et méthodes).

3.2 Classes et dépendances

Nous définissons les valeurs des critères que nous avons défini au chapitre précédent.

Pour une réification et une autonomie des dépendances. Les dépendances possèdent un potentiel d'abstraction, aussi considérer une dépendance comme une entité à part entière permet d'en spécifier clairement le comportement et d'apporter le même soin à la dépendance qu'aux autres objets modélisés. La réification des dépendances permet de les intégrer au modèle objet choisi et de leur appliquer les mêmes techniques de factorisation et de définition incrémentale qu'aux autres objets.

> Dans le concept de radio-boutons, la contrainte d'exclusion qui spécifie qu'un seul des boutons n'est activé à la fois, est une donnée à part entière logiquement dissociable du comportement d'un seul bouton.

De plus, un objet dépendance représente des informations intrinsèques à la dépendance comme le type des objets sur lesquels elle est susceptible d'agir, leur nombre, le comportement dynamique. Elle est l'endroit idéal pour définir des informations liés à la communication comme la conversion de données échangées lors de la communication.

> Une dépendance peut adapter ou convertir un message ou ses arguments afin qu'il soit *conforme(s)* aux attentes des objets. Le fait d'associer ce traitement à la dépendance évite de spécialiser les objets dans le seul but d'adapter leurs interfaces.

La réification d'une dépendance augmente l'expressivité de celle-ci ; le comportement et les informations intrinsèques de la dépendance peuvent être prises en compte lors de la définition du comportement dynamique de celle-ci.

Pour une dissociation entre la mise en œuvre d'une dépendance et les objets. La définition d'une dépendance doit être dissociée de celle des classes d'objets sur lesquelles elle porte. En effet, lier une dépendance à une classe limite la possible réutilisation de la dépendance. Les dépendances exprimées sont limitées à des dépendances internes aux objets.

> En THINGLAB, définir qu'un point est au milieu de deux autres passe par la définition d'un objet `MidPointLine`. La dépendance est alors intimement liée à cet objet et ne peut pas être réutilisée pour d'autres objets. La solution qui consiste à définir d'un côté les points et de l'autre la dépendance *est-au-milieu-de* distingue logiquement le comportement et la structure des objets des dépendances.

Après la déclaration d'une dépendance, l'utilisation de son comportement dynamique permet de maintenir sa cohérence. La déclaration ne doit pas être liée à la création des objets. En règle générale, des objets sont créés, une dépendance assujettit temporairement leurs comportements. Lorsque la dépendance est détruite, les objets doivent retrouver leur comportement initial.

> Le troisième point n'était pas au milieu des deux autres : après déclaration de la dépendance *est-au-milieu-de*, il est effectivement au milieu des deux autres points. Lorsqu'un des points changent de coordonnées les autres points doivent changer de telle sorte que les trois points vérifient la dépendance qui les lient. Lorsque la dépendance est détruite, ce point n'est plus contraint.

Dépendance et composition. Une dépendance peut faire partie intégrante d'un objet. Elle représente alors des contraintes entre les différentes parties d'un objet. Nous appelons de telles dépendances des *intra-dépendances*. Bien que de telles dépendances soient liées étroitement à la classe de l'objet composite, nous pensons que la dépendance doit être spécifiée de manière distincte des fonctionnalités de l'objet composite : les méthodes de l'objet composite doivent exclusivement prendre en compte le comportement de celui-ci. La dépendance gère les contraintes entre les composants de celui-ci. Dans ce cas, la dépendance doit avoir la même durée de vie que l'objet.

> Dans le cas, d'un feu de signalisation ou de radio-boutons, la contrainte d'exclusivité entre les lampes ou les boutons doit être exprimée de manière explicite et dissociée autant que possible des fonctionnalités du feu ou de la colonne de radio-boutons de manière à être réutilisable ou plus facilement maintenue.

3.3 Propriétés des dépendances

Granularité entre objets. Au vu des postulats P1 et P2 (voir 3.1), un objet est une entité non pas définie par sa structure mais par son comportement. Les dépendances doivent porter sur des objets et non entre variables d'instances de ceux-ci.

Pour des dépendances transparentes. Une dépendance ne doit pas être explicitement invoquée par le programmeur.

> Dans le cas du point au milieu des deux autres, lorsqu'un des points bougent, le système demande aux autres points de bouger. Le programmeur n'a pas à invoquer directement le rétablissement de la dépendance.

Pour un respect de l'encapsulation. En complet accord avec le premier postulat (voir 3.1), une dépendance doit respecter l'encapsulation des objets sous sa dépendance. Ce respect de l'encapsulation est conforté par le fait que nous avons choisi d'exprimer uniquement des dépendances entre objets (voir granularité). Une dépendance doit donc être exclusivement décrite qu'à l'aide de méthodes.

Pour des dépendances riches. Une dépendance ne doit pas être limitée : ni sur le nombre d'objets y participant, ni sur un éventuel sens d'influence de la dépendance. En particulier, une dépendance doit pouvoir spécifier le comportement d'ensembles d'objets.

Pour un héritage entre dépendances. La spécification des dépendances doit offrir des mécanismes facilitant leur définition incrémentale et la réutilisation. De tels mécanismes doivent tirer partie des spécificités des dépendances comme le comportement dynamique.

> Dans RENDEZVOUS, le comportement dynamique d'une dépendance peut être enrichi par ajout de nouvelles règles. Dans les systèmes à base de contraintes, l'utilisation d'opérateurs logiques pour combiner plusieurs contraintes est nécessaire.

3.4 Modèle de maintien de la cohérence

Nous étudions maintenant comment le système doit prendre en compte le maintien de la cohérence des dépendances.

Contrôle des messages. En accord avec le postulat P2 qui précise que le seul moyen de communication entre objets est l'envoi de messages et avec la granularité choisie (dépendance entre objets, voir en 3.3), le maintien de la cohérence doit être basé sur **le contrôle des messages** reçus par les objets participants à une dépendance (voir 2.4).

Une remarque liée à la fois au contrôle des messages et au fait que nous avons choisi de dissocier les dépendances des objets (voir localisation en 2.2.3) doit être faite : le contrôle des messages doit être sélectif. Le postulat P3 précise que tous les objets appartenant à une classe possèdent le même comportement. Or, le contrôle des messages doit s'appliquer exclusivement aux objets mis en dépendance et ne pas pénaliser l'ensemble des objets d'une classe dont une des instances participe à une dépendance.

Maintien de cohérence : réactivité et gardes. Idéalement, la cohérence d'une dépendance devrait être décrite par un invariant et le système devrait la maintenir. Ainsi d'un état cohérent, la dépendance deviendrait incohérente suite à une perturbation et le système resatisferait cet état de cohérence.

Cependant, la réalisation de cette vision idéale pose des problèmes :

- Tout d'abord, la spécification d'invariants est parfois très difficile à définir. La dépendance entre un objet et une représentation graphique n'est pas facilement décrite par un invariant.
- L'inférence automatique par le système des actions à exécuter pour resatisfaire la cohérence de la dépendance n'est possible que dans des cas très limités comme l'arithmétique linéaire (voir ThingLabII). Ceci provient principalement des faits suivants :
  - Les comportements d'un objet sont *a priori* complexes et inconnus.
  - Les objets ne sont pas limités à des variables. Les méthodes ne se limitent pas à l'affectation de valeur aux variables d'instances.

La solution consiste à faire décrire par le programmeur le maintien local de la cohérence des dépendances par la donnée de règles. Le système utilise ces règles pour assurer le maintien global des dépendances. Une approche naturelle dans le contexte de langages objets consiste à proposer des règles causales de type *«si cause alors conséquence»* (la *cause* perturbe la cohérence, la *conséquence* la rétablit) ou l'interdiction de messages.

> La contrainte d'exclusivité d'une colonne de radio-boutons peut être exprimée de la manière suivante : un des boutons n'est sélectionné que si aucun autre bouton ne l'est déjà.

Cependant, l'utilisation de règles causales n'assure pas la terminaison et le déterminisme des programmes. C'est pourquoi le mécanisme de maintien de la cohérence doit être manipulable afin de permettre de modéliser à la fois un plus riche éventail de dépendances et de mettre en place des algorithmes de maintien différents.

> Pour des raisons d'efficacité, les conséquences peuvent être filtrées afin d'éviter toutes redondances [Kars 93]. Ainsi il n'est plus utile de translater la représentation d'un objet si celui-ci est détruit.

Terminaison. Dans un graphe de dépendances, différentes sortes de cycles peuvent apparaître. En accord avec l'encapsulation des données, seuls les cycles dus à l'unique présence de dépendances explicites[1] doivent être traitées.

[1] Les dépendances implicites écrites à l'intérieur des classes ne nous concernent pas (la méthode M appelant la N et inversement).

Idéalement, une détection de cycles statique doit avertir le programmeur d'éventuels cycles. Cependant, il ne faut pas interdire la définition de telles situations car les cycles peuvent exprimer une symétrie dans la dépendance. Le système doit cependant être capable de gérer dynamiquement ces cycles.

Dans RENDEZVOUS [HILL 93b], de nombreux liens pourraient être syntaxiquement détectés comme menant à des cycles. Cependant, de telles définitions spécifient simplement la symétrie de la dépendance comme le montre l'exemple suivant.

```
(defClass ttt-game-link
   :superclass Link
   :dependencies ((setf (reset-pending view) (reset-pending abstraction))
                  (setf (reset-pending abstraction) (reset-pending view))))
```

Le système doit empêcher dynamiquement que de telles dépendances mènent à un cycle.

Certaines dépendances permettent de spécifier un prédicat indiquant si elle sont cohérentes. Il est possible d'utiliser de telles informations afin de stopper dynamiquement les cycles. La dépendance *est-au-milieu-de* permet facilement d'énoncer un prédicat précisant sa cohérence.

Déterminisme. Idéalement, étant donné un graphe de dépendances, le maintien de la cohérence devrait être déterministe. Cependant, comme nous l'avons souligné en 2.4.3 aucun des langages objets intégrant des dépendances ne l'assure. Même les langages basés sur des algorithmes de planification comme RENDEZVOUS [HILL 93a] n'assurent pas un maintien de la cohérence déterministe.

En réponse, à ce non-déterminisme, il est nécessaire de proposer au programmeur un éventail de solutions et que celui-ci soit quelquefois partie prenante dans les choix mis en œuvre pour tendre vers des solutions satisfaisantes. Ainsi différentes stratégies de maintien de la cohérence des dépendances doivent pouvoir être offertes comme par exemple, de la profondeur, de la largeur, du filtrage ... Cette dernière remarque nous amène naturellement à un dernier aspect qu'une réelle intégration doit réalisée : l'adaptabilité du système.

3.5 Adaptabilité des dépendances

Les travaux étudiés se limitent à la définition d'un seul type de dépendance pour lequel la définition, la propagation, les comportements en cas de cycles sont proposés. Aucun d'entre eux ne proposent de mécanismes de définition des dépendances et de maintien de la cohérence adaptables.

Or une réelle intégration dans un langage à classes ne doit pas se limiter à une seule sorte dépendances. En plus de proposer un mécanisme de base permettant la définition du comportement dynamique, le maintien de la cohérence... cette intégration doit permettre de particulariser les différents aspects d'une dépendance suivant les besoins des utilisateurs. Cette possibilité s'intègre alors parfaitement aux propriétés réflexives du modèle choisi.

Partie II

Un modèle de dépendances pour un modèle à classes

Dans la partie précédente, nous avons énoncé les différents problèmes posés par l'absence de prise en compte des dépendances au sein même des modèles objets (voir en 1.2). Nous avons ensuite analysé les solutions proposées par d'autres langages au **chapitre 2**. Ceci nous a permis de définir les caractéristiques d'une intégration idéale de dépendances dans un modèle à classes de type ObjVLisp**au chapitre 3**.

Cette partie est dédiée à la présentation de notre modèle de dépendances.

Au **chapitre 4**, nous présentons notre modèle de dépendance. Nous décrivons comment les dépendances rendent explicites des informations liées aux interactions entre objets et comment ces informations sont utilisées par des contrôleurs pour assurer la cohérence des dépendances. Cependant, ce modèle général ne présuppose aucun choix de structuration des objets (classes ou prototypes).

Les deux chapitres suivants définissent une intégration de ce modèle dans un modèle à classes. De même, ils peuvent être perçus comme la description du langage FLO. FLO est le langage dont l'architecture est présentée dans la troisième partie de cette thèse (voir aux chapitres 8 et 9).

Au **chapitre 5**, nous abordons donc comment ce modèle s'intègre dans un modèle à classes. En particulier, nous discutons des relations entre les objets, leurs classes et les dépendances. Le maintien de la cohérence des dépendances est abordé. Nous insistons ensuite sur les avantages de la réification des dépendances. Nous montrons que la dépendance est l'endroit dans lequel les informations relationnelles doivent être définies. Nous abordons la possibilité de définir des dépendances entre dépendances. Les limites du modèle sont abordées en montrant ces faiblesses lors de l'expression de dépendances entre classes. Un mécanisme de définition incrémental est proposé : un héritage entre dépendances. Nous terminons en montrant que les dépendances sont particulièrement bien adaptées pour gérer les accès entre composants et le maintien de la cohérence dans des hiérarchies de composition.

Dans le **chapitre 6**, nous abordons les problèmes de globalité induits par la définition locale des dépendances. Nous montrons comment les contrôleurs, des entités responsables du maintien global des dépendances, apportent des solutions.

4

Un modèle de dépendances

« The behaviors of objects depend on the context in which they exist. Synchonizers allows us to express the contextual constraints of a part-whole hierarchy as an aspect of the hierarchy, not an aspect of the parts » [FRØL 93].

Dans le premier chapitre de cette thèse, nous avons fait état de la diversité des termes utilisés pour nommer les dépendances entre objets et nous avons proposé une définition intuitive d'une dépendance. Dans ce chapitre, nous explicitons formellement cette définition dans le contexte général d'un langage à objets ; c'est-à-dire sans considérer l'existence de classes. Dans les chapitres suivants, ce modèle sera étendu pour prendre en compte l'abstraction et la factorisation offertes par les classes.

4.1 Objets et contexte

De la même manière que AGHA dans [FRØL 93], nous considérons que le comportement d'un objet est influencé par le contexte dans lequel il se trouve. Le contexte d'un objet est défini par ses interactions avec d'autres objets et les contraintes impliquées par ces interactions : ce que nous nommons des dépendances.

> Dans le cadre des interfaces hommes-machines, une représentation fidèle des objets est souvent nécessaire. Ainsi à chaque changement d'état d'un objet, ses représentations doivent être mises à jour. Le fait de représenter graphiquement un objet ne modifie pas fondamentalement sa nature ou ses fonctionnalités. Seul le *contexte* de l'objet change : l'objet est en relation avec ses représentations et doit leur indiquer ses changements d'états.

Dans une approche traditionnelle, cette prise en compte du contexte est gérée de manière interne par les objets. Ainsi la structure et les fonctionnalités des objets mêlent les aspects intrinsèques et relationnels de l'objet. Comme nous l'avons énoncé au chapitre 1, cet amalgame pose des problèmes.

4.1.1 Prise en compte d'un contexte à base de dépendances

Notre approche est basée sur une spécification *explicite*, sous forme de dépendances, du contexte dans lequel les objets évoluent. La prise en compte de ce contexte ne se fait plus à l'intérieur du code des objets (structure et comportement) mais lors d'un envoi de message. A ce titre, nous proposons un envoi de message étendu qui inclut la gestion du contexte dans lequel les objets interagissent.

Les objets définissent les données structurelles et comportementales qui leur sont propres. Les dépendances entre objets sont spécifiées de manière indépendante du code de ces objets et prises en compte lors de l'envoi de messages. Cette perception d'un objet et de son contexte est proche des travaux sur la notion de point de vue des objets [Carr 90, Harr 93, Ossh 95, Carr 96] dans le sens où l'objet peut avoir un comportement enrichi, modifié suivant le point de vue (le contexte) que l'on considère.

Une dépendance décrit de manière déclarative, explicite et locale l'influence de la réception d'un message par un des objets participants sur l'ensemble des objets participant à la dépendance[Forn 90a, Chab 93]. Elle décrit par exemple les conditions sous lesquelles une méthode peut être appliquée, la délégation ou l'émission d'autres messages en réaction à la réception d'un message.

4.1.2 Illustrations : points et alignement

Nous illustrons notre approche dans le cas de points géométriques et d'expression de dépendances entre eux. La figure 4.1 illustre la situation.

Des points. Un objet point définit sa structure et son comportement de manière intrinsèque et ceci quelque soit le contexte dans lequel il se trouve. Il définit des coordonnées et des méthodes comme le changement absolu ou la translation : `x`, `y`, `déplacer`, `translater`.

Par contre, l'interprétation du comportement d'un point est influencé par les dépendances qui existent entre ce point et d'autres points. Ainsi si un point est en dépendance d'alignement avec d'autres points, l'interprétation de son comportement prend en compte cette dépendance.

Des dépendances. La dépendance, nommée *aligner-2-sur-1*, qui spécifie qu'un point, *Point*2, est aligné horizontalement sur un autre point, *Point*1, s'énonce déclarativement comme défini dans la figure 4.1.

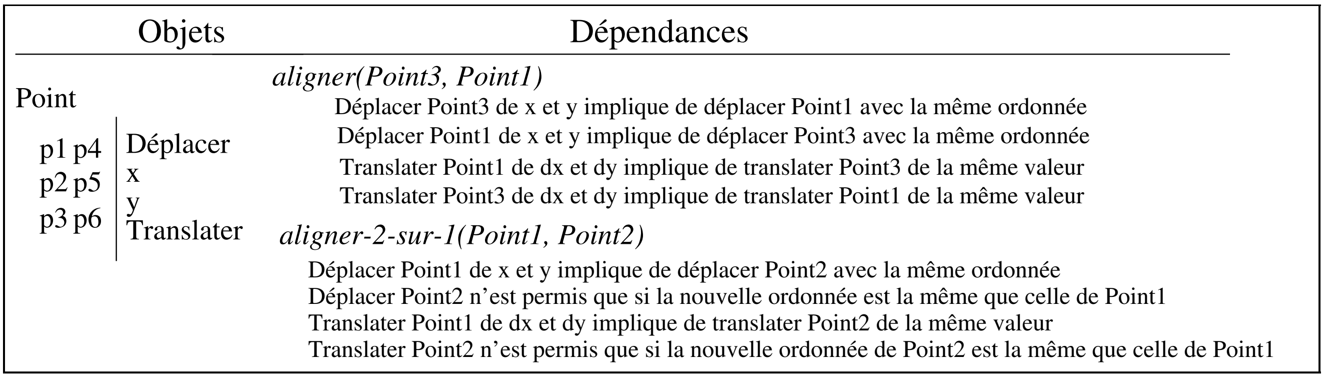


Figure 4.1: Une séparation logique entre objets et dépendances. Les objets points définissent leur structure et comportement intrinsèques. Les dépendances spécifient de manière explicite, déclarative et locale le contexte de certains des points.

Prise en compte du contexte. Lors de la réception d'un message par des participants, la prise en compte du contexte peut être de propager les changements de coordonnées (cas de la dépendance *aligner* entre les points `p1` et `p3`) ou de lui interdire de bouger (cas de la dépendance *aligner-2-sur-1* : le point `p2` ne peut bouger que pour s'aligner avec le point `p1`). Cette variation du comportement d'un point est gérée par le mécanisme d'envoi de messages qui «intègre» le contexte décrit par les dépendances.

La figure 4.2 illustre le maintien de la cohérence des dépendances. Lorsqu'un message `déplacer` est envoyé au point `p3`, un message `déplacer` est envoyé au point `p1` (comme le spécifie la première ligne de la dépendance *aligner*). Le point `p1` étant en dépendance avec le point `p2` par la dépendance *aligner-2-sur-1*, ce message `déplacer` envoyé au point `p1` implique d'envoyer un message `déplacer` au point `p2`. Cependant, l'application de la méthode `déplacer` sur `p2` est soumise à condition (une garde) comme défini à la ligne 2 de la figure 4.1. L'objet `p4`, ne participant à aucune dépendance, reçoit et traite les messages normalement.

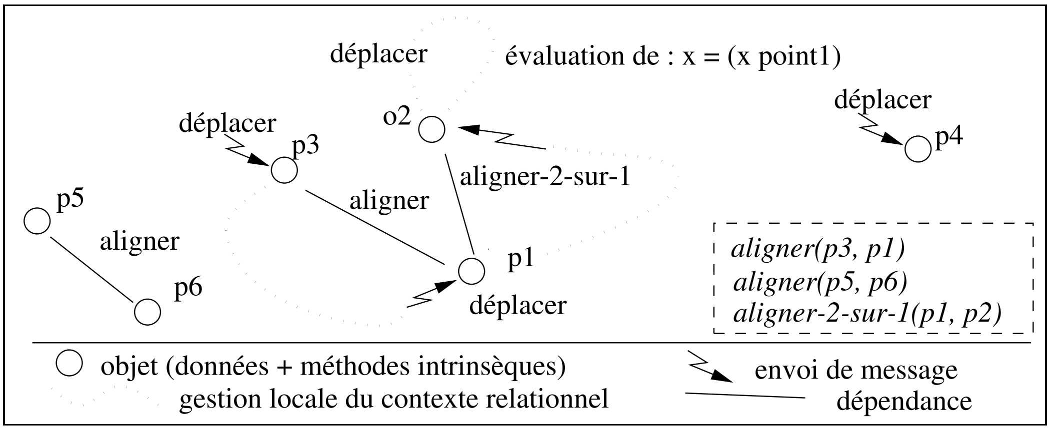


Figure 4.2: Un envoi de message étendu prenant en compte la donnée d'un contexte local d'interprétation d'un message sous forme de dépendance.

4.2 Définitions

Nous proposons maintenant quelques définitions qui nous permettent de définir plus formellement notre modèle.

4.2.1 Objet.

Soit Θ l'ensemble des objets, o un objet quelconque de Θ est défini par les propriétés suivantes :

- un *état*, noté $s(o)$, qui est composé de valeurs de types de base (entiers, chaînes de caractères...) ou d'autres objets.
- un ensemble de *méthodes* $M(o)$ permettant de modifier l'état de l'objet.

Par extension, $S(o^1, o^2, ..., o^n)$ est l'état d'un ensemble d'objets. De plus, on définit Σ comme l'ensemble des états des objets de Θ et Π l'ensemble des méthodes.

Application d'une méthode.

Soit la fonction d'*application*
$\Pi \times \Sigma \to \Sigma$
$a, e \to a(e)$

Nous notons $s_a(o)$ le nouvel état résultant de l'application d'une méthode a sur l'état d'un objet o : $a(s(o)) = s_a(o)$. En particulier, tous les objets possèdent la méthode identité Id qui appliquée à un objet ne modifie pas l'état de l'objet : $id(s(o)) = s(o) = s_{Id}(o)$.

Par extension, nous définissons la fonction d'*application* multiple
$(\Pi \times \Theta)^n \to \Sigma^n$
$((a^1, o^1), (a^2, o^2)...(a^n, o^n)) \to S((a^1, o^1)(a^2, o^2)(a^3, o^3)...(a^n, o^n))$

Notons que cette définition ne précise pas l'ordre d'applications des méthodes. De plus, on a l'équivalence suivante : $(s_{a^1}(o^1), s_{a^2}(o^2), ...) = S((a^1, o^1)(a^2, o^2)...)$

4.2.2 Dépendance.

Soit D l'ensemble des dépendances, $D \subset \Theta$.
Une dépendance, d, est caractérisée par :

- un *nom*,
- $P(d)$, un *ensemble* d'objets participants, $P(d) = \{o, o', o'', ..., o^n\}$
- une *propriété*,$Prop_d$ que les objets participants doivent vérifier.

  $(\Sigma)^* \rightarrow (Vrai, Faux)$
  $(s(o), s(o'), ..., s(o^n)) \rightarrow Prop_d(s(o), s(o'), ..., s(o^n))$

- un *comportement dynamique*, CD, qui spécifie comment la cohérence de la dépendance est assurée suite à la *réception* d'un message. Il rend une liste de méthodes à appliquer sur les participants de la dépendance.

  $D \times \Pi \times \Theta \rightarrow (\Pi \times \Theta)^*$
  $d, a, o \rightarrow CD(d, a, o)$

  Soient $d \in D, o \in P(d), a \in M(o)$,
  $CD(d, a, o)$ *rend* $((a', o')(a'', o'')(a''', o''')...)$
  tel que $Prop_d(S(CD(d, a, o)) \bigcup S(P(d)))) = Vrai$
  où $o', o'' \in$ a $P(d)$ et $a' \in M(o'), a'' \in M(o''), a''' \in M(o''')$

  Notons que o', o''... peuvent être égaux ce qui signifie que plusieurs messages peuvent être adressés à un même objet ; que o', o''... peuvent être égaux à o, ce qui signifie que l'objet ayant reçu le message o peut appliquer la méthode invoquée ou recevoir de nouveaux messages. De même, a' peut être égale à a et $a', a'', ...$ peuvent être égaux. Il faut remarquer que l'ordre d'évaluation de la liste des méthodes rendue par le comportement dynamique n'est pas précisé.

Cohérence.

- Lorsque la propriété d'une dépendance est vérifiée, elle est dite *cohérente*. L'expression de cette propriété peut être explicite sous forme d'un prédicat ou implicite.

  Soit $d \in D$, d est cohérente si $Prop_d(S(P(d))) = Vrai$.

Contexte d'un objet.

- On note $L(o) = \{l_1, l_2, ..., l_n\}$ l'ensemble ordonné des dépendances relatives à un objet tel que $\forall d \in D, d \in L(o)$ *ssi* $o \in P(d)$

Méthode non-perturbante.

- Lorsqu'une méthode a de $M(o)$ n'invalide pas la cohérence d'une dépendance d, elle est dite *non-perturbante* pour d par rapport à o. Le comportement dynamique spécifie simplement que la méthode doit être appliquée.
  $CD(d, a, o)$ rend $((a, o))$

On définit le prédicat *perturbante?* qui précise si une méthode est perturbante :

$D \times \Pi \times \Theta \rightarrow (Vrai, Faux)$
$d, a, o \rightarrow perturbante?(d, a, o)$

Méthode perturbante.

- Lorsqu'une méthode a invalide la cohérence de d, a est dite *perturbante* pour d par rapport à o. Le comportement dynamique gère trois cas qui peuvent être combinés. Ainsi afin d'assurer la cohérence d'une dépendance :

  – une *délégation* de message peut être réalisée. La méthode invoquée n'est pas définie pour l'objet o, une nouvelle méthode est invoquée sur le même ou d'autres objets.

  $CD(d, a, o)$ rend
  si $a \notin M(o)$
  alors $((a', o'))$
  sinon $((a, o))$

  De plus, on définit le prédicat *délégation?* :

  $D \times \Pi \times \Theta \rightarrow (Vrai, Faux)$
  $d, a, o \rightarrow delegation?(d, a, o)$

  Ce prédicat précise si une délégation est spécifiée sur la dépendance d, pour la méthode a et l'objet o.

  – une *garde*, qui est un prédicat associé à la dépendance, à l'objet et relatif à la méthode invoquée, peut être testée. La méthode n'est appliquée que si elle ne remet pas en cause la cohérence de la dépendance.

  Soit $a \in M(o)$,
  $CD(d, a, o)$ rend
  si $valeur - garde(d, a, o) = Vrai$
  alors $((a, o))$
  sinon $((Id, o))$.

  On définit le prédicat *application-interdite?* :

  $D \times \Pi \times \Theta \rightarrow (Vrai, Faux)$
  $d, a, o \rightarrow application - interdite?(d, a, o)$

  Ce prédicat précise si une garde est évaluée à faux sur la dépendance d, pour la méthode a et l'objet o.

  – un *message compensatoire* peut être appliqué en réaction à l'application de la méthode invoquée. La méthode invoquée est donc appliquée, puis des messages sont envoyés aux participants de la dépendance pour rétablir sa cohérence.

  Soit $a \in M(o)$,
  $CD(d, a, o)$ rend
  $((a, o)(a'', o'')(a''', o''')...)$
  oú $o, o', o'''... \in P(d)$ *et* $a \in M(o), a'' \in M(o''), a''' \in M(o''')$

  Une délégation et une garde ou un message compensatoire ne peuvent être définis pour une même méthode. De plus, la combinaison d'une garde et d'un message compensatoire assure que la garde est prioritaire et que la méthode invoquée n'est appliquée qu'une fois. Ainsi pour une garde vraie et un message compensatoire, $CD(d, a, o)$ rend $((a, o)(a'', o'')(a''', o''')...)$ l'application de a sur o n'est réalisée qu'une seule fois. Ceci est expliqué en 4.4.

On définit les deux fonctions d'ordre supérieur, *Some* et *Every*.

$\Pi \times \Theta \times Predicat \rightarrow (Vrai, Faux)$
$a, o, p \rightarrow Some(a, o, application - interdite?)$

Etant donnée un objet, *Some* spécifie qu'au moins une dépendance $L(o)$ vérifie le prédicat donné par rapport à la méthode a et *Every* si toutes les dépendances vérifient ce prédicat.

4.2.3 Illustrations

Reprenons l'exemple des points géométriques. La dépendance d'alignement *aligner-2-sur-1* qui spécifie qu'un point, nommé point2, est toujours aligné suivant les ordonnées par rapport à un point nommé point1, est caractérisée par:

- un nom : *aligner-2-sur-1*,
- une liste de participants, $L(aligner-2-sur-1) = \{point1, point2\}$,
- une propriété à maintenir, $Prop_{aligner-2-sur-1}$: $(y\ point1) = (y\ point2)$ et
- un comportement dynamique, $CD_{aligner-2-sur-1}$:
  CD(d, déplacer x y, point1) = ((déplacer x y, point1) (déplacer x (y point1), point2))
  CD(d, déplacer x y, point2) =
  si x = (x point1)
  alors((déplacer x y, point2))
  sinon((Id, point2))
  CD(d, translater dx dy, point1) = ((translater dx dy, point1) (translater dx dy, point2))
  CD(d, translater dx dy, point2) =
  si (x point2) + dx = (x point1)
  alors((translater dx dy, point2))
  sinon((Id, point2))
  CD(d, changecouleur val, point1) = ((changecouleur val, point1))

Dans cet exemple, le comportement dynamique n'utilise pas la dépendance *d*. Cependant, la dépendance étant un objet, elle est susceptible d'être invoquée et de participer au maintien de sa cohérence.

4.2.4 Remarques.

Il n'est pas toujours simple de définir la cohérence d'une dépendance à l'aide d'un prédicat entre ses participants. Il faut remarquer que nous ne voulons pas modifier les objets pour définir de tels prédicats et les seules informations que nous connaissions de l'objet sont celles données par son interface (un ensemble de symboles représentant ses méthodes). Par contre, il est possible de spécifier comment cette propriété est maintenue suite à la réception d'un message par un des objets participants : c'est le rôle du *comportement dynamique*.

Ainsi, bien qu'une dépendance *représente-historique-valeurs* spécifie par le biais de règles la cohérence entre un graphe représentant l'historique des valeurs d'un objet et celui-ci, la cohérence d'une telle dépendance est difficilement décrite sous forme d'un prédicat entre l'objet et le graphe.

Prise en compte de l'intention de perturbation. Nous insistons sur le fait que le comportement dynamique spécifie comment la cohérence d'une dépendance effective est maintenue suite à la réception d'un message et non à l'application d'une méthode. En effet, le maintien de la cohérence n'est pas uniquement basé sur un rétablissement après perturbation. Il prend en compte l'*intention* de perturbation. Ainsi il est possible de spécifier qu'une méthode invoquée n'est pas obligatoirement appliquée. Nous précisons ce point en 4.4.

4.3 Graphe de dépendances et nécessité d'un contrôle.

Un ensemble de dépendances induit un *graphe de dépendances* dont les nœuds sont les objets participants aux dépendances et les arêtes étiquetées par les dépendances. Le graphe n'est pas forcément connexe (voir la figure 4.2).

Les dépendances exprimant une influence locale entre participants par le biais de leur comportement dynamique et pouvant être connectées sous forme de graphe, un message perturbant peut entraîner une *propagation de messages*. Ainsi l'envoi d'un message à un objet pour rétablir la cohérence d'une dépendance peut nécessiter de rétablir la cohérence d'autres dépendances connexes. Lors d'une propagation de messages, des cycles ou des problèmes liés au non-déterminisme des réactions peuvent apparaître. Ainsi lorsqu'un objet reçoit plusieurs messages compensatoires de différents chemins du graphe de dépendances, l'état de l'objet peut dépendre de l'ordre de réception des messages ou du chemin suivi dans le graphe (voir figure 4.3).

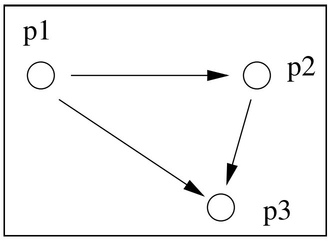


Figure 4.3: Problèmes du déterminisme des réactions : le chemin suivi par le flot de propagation pour atteindre `p3` change l'état de ce dernier.

Contrairement aux systèmes basés sur une propagation *immédiate* (valeurs actives [STEF 86a], dépendances [GOLD 83],..), notre modèle introduit des entités gérant le maintien global de la cohérence : des contrôleurs. Alors qu'une dépendance spécifie l'interaction entre participants, un contrôleur la réalise en utilisant l'ensemble des dépendances définies sur le groupe d'objets qu'il supervise. Les contrôleurs permettent de définir différents algorithmes de maintien de la cohérence. Ils sont une abstraction du maintien de la cohérence qui dans de nombreux travaux est non explicite et câblé dans l'interprète du langage.

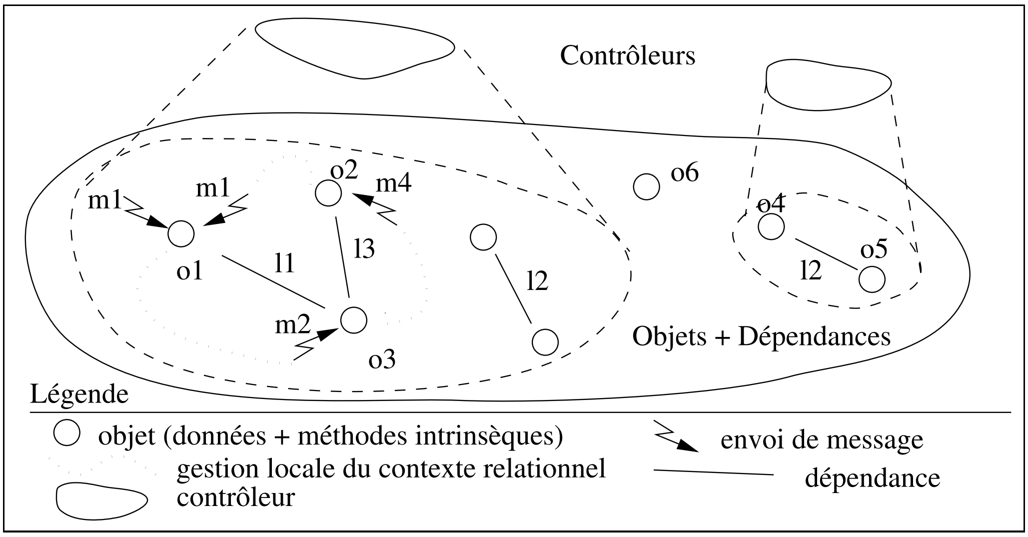


Figure 4.4: Une architecture à deux niveaux pour un contrôle global du maintien des dépendances.

La figure 4.4 illustre le contrôle opéré par les contrôleurs afin d'assurer un maintien global des dépendances. Le contrôleur de gauche gère par exemple un cycle dans le maintien de la cohérence.

4.3.1 Définitions.

Soit C l'ensemble des contrôleurs, c un contrôleur appartenant à C est caractérisé par :

- un *ensemble d'objets contrôlés*, $OC(c) = \{o_1, o_2, ..., o_n\}$
- un *contexte* : un ensemble ordonné de dépendances, $EDE(c) = \{l_1, l_2, l_3, ...l_n\}$

- une *fonction de maintien* F qui assure que les dépendances $EDE(c)$ sont cohérentes.

$$\Pi \times \Theta \rightarrow \Sigma$$
$$m, o \rightarrow F(c, a, o)$$
$$\text{tel que } \forall d \in EDE(c), Prop_d(F(OC(c))) \;=\; Vrai.$$

Compléments.

- Tout objet o est associé à un contrôleur et un seul c tel que $control(o) = c$ et $o \in OC(c)$.

- Pour une méthode *non-perturbante* a sur une dépendance, on a
$\forall o \in \Theta, S(s(o^1), s(o^2), ..., s_a(o), ..s(o^n)) = F(c, a, o)$

Nous explicitons dans la section suivante comment la fonction de maintien utilise le comportement dynamique des dépendances pour assurer la cohérence globale du systéme.

4.4 Syntaxe et interprétation sémantique

4.4.1 Syntaxe abstraite

La donnée du comportement dynamique est verbeuse. Nous introduisons des opérateurs afin de simplifier la définition des dépendances. Le comportement dynamique possède la syntaxe abstraite suivante :

comportement_ dynamique	::= {*règle_ d'interaction*}+
règle_ d'interaction	::= *sélecteur receveur args opérateur action*
action	::= {*sélecteur cible args*}+
receveur	::= symbole
cible	::= symbole
opérateur	::= **implies** | **permitted-if** | **corresponds**
args	::= symbole + | *action*

La sémantique des opérateurs est celle décrite plus avant pour le comportement dynamique :

- L'opérateur `implies` spécifie un message compensatoire : suite à la réception et à l'application d'un message à un objet dit *receveur*, un message compensatoire est émis.

- L'opérateur `permitted-if` une garde spécifie une garde devant être vérifiée pour appliquer la méthode invoquée.

- L'opérateur `corresponds` spécifie une délégation de message : un objet reçoit un message, la méthode n'est pas appliquée, un nouveau message est émis.

Illustrations. Le comportement dynamique de la dépendance *aligner-2-sur-1* s'écrit alors :

déplace point1 x y	implies	déplacer point2 x y
déplace point2 x y	permitted-if	(= x (x point1))
translater point1 dx dy	implies	translater point2 dx dy
translater point2 dx dy	permitted-if	(= (+ (x point2) dx) (x point1)))

On note que les méthodes non-perturbantes sont omises.

4.4.2 Maintien de la cohérence et opérateurs.

F, la fonction de maintien, peut être définie par des règles de réécriture R telle que :

Soit l'état des objets $\xi = (s(o^1), s(o^2), ..., s(o^n)) = S(OC(c))$
$F(c, a, o) = \xi'$
où $< a, \xi, o, \xi_i, () > \Rightarrow_{R*} < Id, \xi', \_, \xi_i, () >$

Nous nommons le tuple $< a, \xi, o, \xi_i, LA >$, une *configuration partielle.* Il est composé d'une méthode a, de l'état courant ξ de tous les objets contrôlés par c, d'un objet o, de l'état initial ξ_i de tous les objets contrôlés par c et d'une liste d'application de méthodes qu'il reste à effectuer LA.

Une règle de réécriture conditionnelle spécifie sous quelle condition une configuration partielle peut être réécrite dans une autre configuration partielle. Soit la règle ci-dessous, la configuration B est réécrite en C, si les conditions A sont satisfaites.

$$\frac{A}{B \Rightarrow_R C}$$

Non perturbation. Le cas d'une méthode non-perturbante spécifie simplement que l'état global prend en compte la modification due à l'application de la méthode invoquée sur l'objet. Ainsi s'il reste des applications à traitées, le système les applique (règles 1).

$$\frac{a \in M(o)\&(Not(Some(a, o, Perturbante?)))}{< a, \xi, o, \xi_i, ((a', o')(a'', o'')...) > \Rightarrow_R < a', \xi' = (s(o_1), s(o_2), ...s_a(o), ...s(o_n)), o', \xi_i, ((a'', o'')...) >} \tag{4.1}$$

$$\frac{a \in M(o)\&(Not(Some(a, o, Perturbante?)))}{< a, \xi, o, \xi_i, () > \Rightarrow_R < Id, \xi' = (s(o_1), s(o_2), ...s_a(o), ...s(o_n)), \_, \xi_i, () >} \tag{4.2}$$

Garde. Lorsqu'une garde est vraie, le message contrôlé est appliqué normalement. Par contre, une dépendance peut interdire l'application d'une méthode afin de préserver la cohérence d'une dépendance.

$$\frac{Some(o, a, application - interdite?)}{< a, \xi_i, o, \xi_i, () > \Rightarrow_R < Id, \xi_i, \_, \xi_i, () >} \tag{4.3}$$

$$\frac{Some(o, a, application - interdite?)\&\xi_i \neq \xi}{< a, \xi, o, \xi_i, ((a', o')(a'', o''..)) > \Rightarrow_R < Id, \xi_i, \_, \xi_i, () >} \tag{4.4}$$

$$\frac{Some(o, a, application - interdite?)\&\xi_i \neq \xi}{< a, \xi, o, \xi_i, () > \Rightarrow_R < Id, \xi_i, \_, \xi_i, () >} \tag{4.5}$$

La règle 4.3 spécifie qu'une interdiction est valide lorsqu'il n'y a plus d'applications à traiter et que le système n'a pas appliqué de messages compensatoires. Dans ce cas, l'état global du système ne change pas. Les deux dernières règles précisent que si une interdiction est spécifiée pour assurer la cohérence d'une dépendance et qu'il reste des applications de méthodes à traiter pour assurer la cohérence (règle 4.4) ou que des applications ont déjà été effectuées (règle 4.5), il y a un conflit. Dans ces deux cas, l'état global du système retourne à l'état initial. Ceci est illustré en 4.5.1.

Délégation. Le cas d'un message délégué spécifie que si des dépendances définissent une délégation de message, alors une seule délégation est traitée (règle 4.6). En effet, le cas d'une méthode retournant un résultat nous a amené à considérer qu'il doit y avoir uniformité entre toutes les délégations traitant du même message délégué. De plus, lorsque le message invoqué n'appartient pas aux méthodes de l'objet receveur et qu'aucune délégation n'est définie, il y a un conflit. L'état global retourne à l'état initial (règle 4.7).

$$\frac{a \notin M(o) \& \exists li \in L(o), CD(li, o, a) = ((a'_i, o'_i))}{< a, \xi, o, \xi_i, LA > \Rightarrow_R < a'_i, \xi, o'_i, \xi_i, LA >} \tag{4.6}$$

$$\frac{a \notin M(o) \& Not(Some(a, o, delegation?))}{< a, \xi, o, \xi_i, LA > \Rightarrow_R < Id, \xi_i, \_, \xi_i, () >} \tag{4.7}$$

Message compensatoire. Lorsqu'il ne s'agit ni d'une délégation de message, ni d'une garde et que des messages compensatoires sont définis, la méthode invoquée est appliquée puis les messages compensatoires sont traités en profondeur comme le montre la seconde règle. Les nouveaux messages compensatoires sont traités en premier.

On pose :

$$\begin{aligned} \mathrm{A} = {} & a \in M(o) \& \ \nexists li \in L(o), CD(li, o, a) = ((Id, o)) \& \\ & \forall li \in L(o), CD(li, o, a) = ((a, o)(a_{11}, o_{11})...(a_{1p}, o_{1p})...(a_{i1}, o_{i1})....(a_{ip}, o_{ip})...(a_{n1}, a_{n1})...(a_{np}, o_{np})) \end{aligned}$$

$$\frac{A}{< a, \xi, o, \xi_i, () > \Rightarrow_R < a_{11}, \xi', o_{11}, \xi_i, ((a_{12}, o_{12})....(a_{ip}, o_{ip})...(a_{np}, o_{np})) >} \tag{4.8}$$

$$\frac{A}{< a, \xi, o, \xi_i, ((a''_1, o''_1)(a''_2, o''_2)...) > \Rightarrow_R < a_{11}, \xi', o_{11}, \xi_i, ((a_{12}, o_{12})..(a_{n1}, a_{n1})..(a_{np}, o_{np})...(a''_1, o''_1)(a''_2, o''_2)..) >} \tag{4.9}$$

Arrêt. Au vu des toutes ces règles, on termine lorsqu'il ne reste plus de méthodes à traiter.

4.5 Problèmes fondamentaux

Le comportement dynamique spécifie les conséquences sur les participants d'une dépendance de manière purement locale. Alors que cette spécification est adaptée pour la définition du maintien de la cohérence d'une dépendance, elle ne prend pas en compte les problèmes dus aux interactions entre plusieurs dépendances. Principalement trois problèmes existent : les conflits entre gardes et messages compensatoires, l'ordre et le déterminisme lors du maintien de la cohérence et les cycles.

4.5.1 Conflits

La conjugaison entre gardes et messages compensatoires n'est pas simple. En effet, une garde et un message compensatoire peuvent être en conflit : l'un interdisant un message et l'autre nécessitant ce message pour que des dépendances soient cohérentes.

Nous illustrons rapidement la solution que nous proposons. Soient des objets colorés `A` et `B` et deux dépendances `l1` et `l2` dont les comportements dynamiques sont :

```
l1 : (change-couleur A couleur) implies (change-couleur B couleur)
l2 : (change-couleur B couleur) permitted-if FAUX
```

Dans l'état initial, `A` et `B` sont de couleur bleue. Le message `A change-couleur rouge` mènerait à un conflit car la garde de la dépendance `l2` étant fausse, la méthode `change-couleur` sur l'objet B ne serait pas appliquée.

Les règles que nous avons définies garantissent que le système se trouve finalement dans un état cohérent.

- Dans l'état initial, la règle 4.3 définit que l'interdiction d'appliquer la méthode `change-couleur` à l'objet `B` ne mène pas à un conflit.

- Les règles 4.4 et 4.5 spécifient que si l'état de l'ensemble des objets contrôlés n'est plus à l'état initial, alors pour éviter un conflit, une interdiction provoque un retour à l'état initial. Ici si la méthode `change-couleur` a été appliquée sur l'objet `A`, le traitement de l'interdiction de cette même méthode sur l'objet `B` provoque un retour en arrière à l'état initial.

La solution que nous avons choisi dans ce modèle est brutale et elle mériterait d'être affinée. Pour le moment lorsqu'une interdiction est traitée et que des réactions restent à traiter le système considère que l'état sera incohérent et donc revient dans l'état initial. Or cette façon de faire ne devrait se produire que si les réactions restant à traiter sont liées à la méthode ayant conduit à l'interdiction. De même, ce retour à un état cohérent lors du traitement d'une garde pourrait être évité dans certains cas, si l'on avait la possibilité de savoir si la dépendance dont provient la garde est cohérente.

4.5.2 Ordre et déterminisme

L'interprétation sémantique de la gestion globale de la cohérence des dépendance est fortement liée d'une part à l'ordre des dépendances par rapport à un objet donné ($L(o)$) lors du calcul de l'ensemble des messages compensatoires et d'autre part à l'ordre de traitement des messages compensatoires (en profondeur). Cette situation mène à un modèle d'interprétation non-déterministe. Ainsi suivant les différents ordres de traitement des messages compensatoires ou des délégations, l'état final sera cohérent mais pourra être différent.

Une solution à ce problème pourrait être de savoir s'il est possible de définir un ordre local des dépendances pour chaque objet compatible avec un ordre global ainsi qu'un ordre global du traitement des messages compensatoires. Cependant, ce type solution semble difficile à trouver.

4.5.3 Gestion des cycles

La fonction de maintien que nous avons défini au travers des règles précédentes ne gère pas les cycles. Ainsi une ou plusieurs dépendances peuvent définir des messages compensatoires menant à des appels mutuels infinis. Or, nous avons justifié l'existence de contrôleurs comme des entités permettant de spécifier le maintien global d'un groupe de dépendances et d'objets. Pour spécifier un maintien global de la cohérence d'un ensemble de dépendances gérant des problèmes de cycles, il suffit de définir une nouvelle fonction de maintien F_{cycles} associée à un nouveau contrôleur.

Pour gérer les cycles, la configuration partielle est enrichie pour prendre en compte la liste des messages compensatoires déjà exécutés. Le tuple $< a, \xi, o, \xi_i, LA >$ devient $< a, \xi, o, \xi_i, LA, LD >$ dans lequel $LD = ((l, a, o)(l', a', o')...)$ représente la liste des messages compensatoires précédemment appliqués. F_{cycles}, la fonction de maintien, peut alors être définie par des règles de réécriture R telles que :

Soit l'état des objets $\xi = (s(o^1), s(o^2), ..., s(o^n)) = S(OC(c))$
$F_{cycles}(c, a, o) = \xi'$
où $< a, \xi, o, \xi_i, (), () > \Rightarrow_{R*} < Id, \xi', \_ , \xi_i, (), ((l, a, o)(l', a', o')....) >$

Rendre la liste des méthodes appliquées offre des informations qui peuvent être utilisées pour analyser le comportement du système (trace, explications, débogage). Les règles suivantes décrivent le fait qu'une méthode n'est pas appliquée si une application identique a déjà eu lieu pour la même dépendance. Les règles précédentes autres que celles gérant les messages compensatoires sont étendues simplement pour prendre en compte l'information ajoutée à la configuration partielle, aussi nous ne les écrivons pas ici. On pose :

A $= a \in M(o) \& \; \nexists li \in L(o), application - interdite?(li, o, a) \&$
$\forall li \in L(o), CD(li, o, a) = ((a, o)(a_{11}, o_{11})...(a_{1p}, o_{1p})_{l1}...(a_{n1}, a_{n1})...(a_{np}, o_{np})_{ln})$

$$\frac{A\&(a,o,l1) \notin LD}{< a,\xi,o,\xi_i,(),LD > \Rightarrow_R < a_{11},\xi',o_{11},\xi_i,((a_{12},o_{12})...(a_{1p},o_{1p})_{l1}...(a_{n1},a_{n1})...(a_{np},o_{np})_{ln}),((a,o,l1)LD) >} \tag{4.10}$$

$$\frac{(a,o,l1) \in LD}{< a,\xi,o,\xi_i,((a_{11},o_{11})...(a_{1n},a_{1n})_{l1}...(a_{np},o_{np})_{ln}),LD > \Rightarrow_R < a_{11},\xi,o_{11},\xi_i,B,LD >} \tag{4.11}$$

Avec : $B = ((a_{12},o_{12})..(a_{1p},o_{1p})_{l1}..(a_{n1},a_{n1})...(a_{np},o_{np})_{ln})$

Ces deux règles définissent qu'une méthode n'est appliquée que si elle ne fait pas partie des méthodes déjà appliquées pour la même dépendance.

4.6 Instanciations du modèle

Envoi de message étendu. Notre modèle est basé sur le contrôle de l'envoi de messages. Nous montrons au chapitre 9 différents mécanismes pour prendre le contrôle de l'envoi de messages lorsque les langages n'offrent pas cette possibilité. Nous pensons *a priori* que ce modèle peut donc être implémenté dans différents langages à objets.

Modèles à prototypes. Nous montrons dans la suite de cette thèse une intégration dans un modèle à classes. Nous avons étudié l'intégration de ce modèle dans le langage à prototypes [Lieb 86a, Unga 87] réflexif Moostrap [Mule 93a, Mule 95a]. Bien que les langages à prototypes ne proposent pas de mécanisme d'abstraction et de factorisation comme les classes du modèle à classes [Male 95], le fait de proposer des entités conceptuellement autonomes pour représenter les dépendances reste un avantage. Ainsi un prototype définit les caractéristiques de l'objet «physique» qu'il représente sans prendre en compte les données relationnelles au sein même de ses champs[1] du prototype. Ces données sont spécifiées extérieurement par le biais de prototypes représentant les dépendances.

En Moostrap, la question du contrôle des messages est simple, le modèle proposé est basé sur la décomposition de l'envoi de messages. Tout objet possède un méta-objet responsable de la recherche des méthodes. Il est ainsi possible de particulariser à souhait l'envoi de message. De plus, contrairement à l'approche de P. Maes [Maes 87a, Maes 88] un même méta-objet en Moostrap peut être associé à plusieurs objets. Cette notion est importante dans notre modèle pour gérer la globalité du graphe de dépendances.

Notre modèle est fortement basé sur une perception comportementale d'un objet. Aussi, ce modèle pourrait s'appliquer à des langages d'acteurs.

Contraintes d'implémentation et limites. Le modèle que nous avons présenté assure la consistance globale du graphe de dépendances. Cependant, certains aspects comme un retour à un état cohérent lors de conflits de gardes, sont difficilement réalisables lors de l'implémentation. De même, il est difficile de s'assurer par une spécification qu'un nouveau contrôleur est correct et maintient l'ensemble des dépendances qu'il contrôle. Ce modèle décrit de manière idéale d'un mécanisme de dépendances, sert de base à de nouveaux travaux permettant de savoir quelles parties du graphe peuvent être maintenue indépendamment les unes des autres [?]. Cependant, nous sommes conscients que les limitations dues à l'implémentation doivent être prises en compte afin de savoir qu'elles propriétés du modèle restent valides lors de l'implémentation.

[1] Certains langages uniformisent les champs et méthodes en ne proposant que des champs parmi lesquelles les méthodes sont définies.

5

Intégration dans un modèle à classes : le langage FLO

« *Relationships between dynamic objects are, however, not as simple as relations in e.g. the Entity-Relationship model - here, we must be able to specify interactions between objects and dependencies between objects. Other authors have pointed out the need to have relationships in high-level object-oriented modeling, but those relationships describe only static relationships between object references.* » [JUNG 93].

Dans le chapitre précédent, le modèle proposé ne présupposait pas l'existence de classes. Ce chapitre présente comment un tel modèle peut être introduit dans un modèle à classes. Notre modèle définit les dépendances comme des entités distinctes des objets participant aux dépendances. Dans notre proposition d'une intégration de dépendances dans un modèle à classes au chapitre 3, nous avons justifié la distinction entre classes et dépendances. Nous présentons donc la place des dépendances par rapport aux classes et aux instances. Ensuite, en nous appuyant sur deux exemples, nous précisons le vocabulaire et la définition des dépendances. Ces bases étant posées, nous illustrons l'utilisation du comportement dynamique. Ensuite, nous montrons comment la réification des dépendances renforce leur expressivité et leur importance. Cette réification nous permet d'aborder l'écriture de dépendances entre dépendances. De plus, les classes étant des objets dans le modèle OBJVLISP, nous définissons des dépendances entre classes et montrons les limites de notre modèle lors de l'héritage de classes liées par des dépendances. Nous répondons ensuite aux problèmes de l'incrémentalité de la définition des dépendances en proposant un héritage adapté aux dépendances. Pour finir, nous montrons comment notre modèle offre des solutions aux problèmes de l'encapsulation et de maintien de cohérence dans les hiérarchies de composition.

Ce faisant nous décrivons le langage FLO qui implémente ce modèle et dont l'architecture est étudiée en détail au chapitre 8

5.1 Définition de dépendances

Nous présentons la place de dépendances par rapport aux classes puis à l'aide d'exemples nous montrons comment une dépendance est définie puis mise en œuvre entre objets. Ces exemples nous servent à définir notre vocabulaire.

5.1.1 Séparation classes / dépendances

Dans notre modèle, les classes définissent la structure et les comportements intrinsèques des objets, les dépendances les différentes contraintes entre ces objets (voir figure 5.1). Le programmeur définit les classes, crée des instances, *définit* des dépendances et *déclare* les objets impliqués dans ces dépendances (sans modifier les classes). Le système, par le biais de contrôleurs, maintient automatiquement la cohérence du graphe des dépendances déclarées, en contrôlant les messages envoyés aux objets liés.

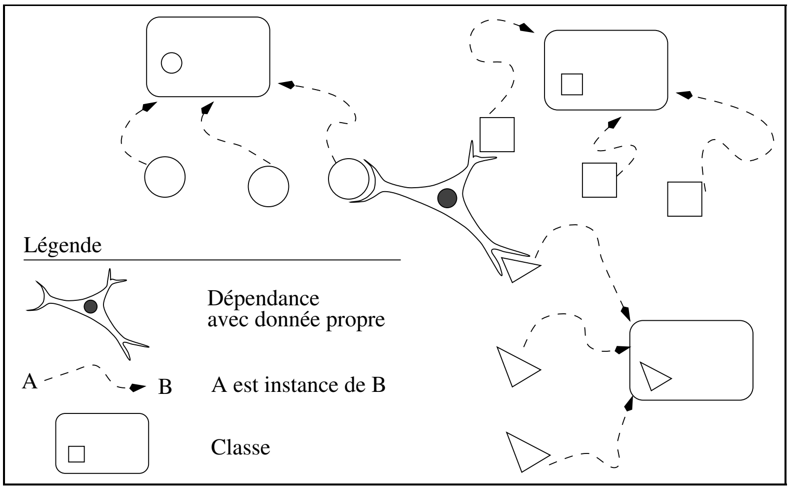


Figure 5.1: Situation entre classes, instances et dépendances.

Les dépendances sont dissociées des classes : la *définition* d'une dépendance n'est pas liée à celle d'une classe et la dépendance peut être mise en œuvre sur des objets bien après leur création : il s'agit de la *déclaration* d'une dépendance. De plus, une même dépendance peut être déclarée entre différentes instances.

5.1.2 Illustrations

Les exemples suivants nous permettent de présenter la façon dont les dépendances sont définies puis déclarées.

Pile et mémoire.

Supposons qu'un programmeur ait défini deux classes indépendantes : la classe `Stack` représentant une pile et la classe `Memory` représentant une mémoire. La classe `Stack` définit quatre méthodes : `pop`, `push`, `empty?` et `empty`. La classe `Memory` définit les méthodes `store`, `unstore`, `total-elements`, `occurrence` et `not-full?`. Il utilise des instances de ces classes normalement. Puis, pour deux instances particulières, `p1` une pile et `m1` une mémoire, il souhaite exprimer une dépendance, appelée *mémorisée-par*, entre `p1` et `m1`, de telle sorte que toutes les valeurs dépilées de la pile soient enregistrées dans la mémoire tant que celle-ci n'est pas pleine (voir figure 5.2).

Définition. Pour ce faire, il définit une dépendance avec l'opérateur `deflink`. Il spécifie le *comportement dynamique* de cette dépendance à l'aide des méthodes définies par les classes `Stack` et `Memory` et des opérateurs `implies` et `permitted-if`, comme défini ci-après.

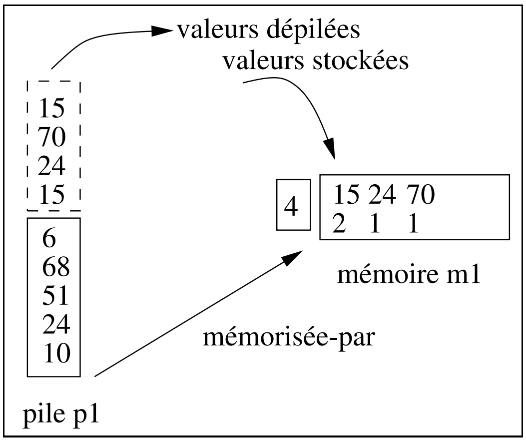


Figure 5.2: Pile et mémoire

```
1  (deflink  mémorisée-par  (:stack :memory)
2     :behavior
3     (((pop :stack) implies (store :memory result))
4        ;; une valeur dépilée est enregistrée dans la mémoire
5      ((pop :stack) permitted-if (not-full? :memory))))
6        ;; la pile n'est dépilée que si la mémoire n'est pas pleine
```

Sans entrer dans les détails, la ligne 3 spécifie que lorsqu'une pile, reçoit un message `pop`, le message `store` est envoyé à la mémoire, avec comme valeur le résultat du message `pop`. La ligne 5 spécifie que la pile ne pourra exécuter la méthode `pop` que si la mémoire n'est pas pleine.

Déclaration. Ensuite, le programmeur indique entre quelles instances, ici `p1` et `m1`, la dépendance est mise en œuvre : c'est la *déclaration* de la dépendance. Les instances `p1` et `m1` sont alors dépendantes.

```
(define p1 (make stack))         ;; création d'une pile
(define m1 (make memory))        ;; création d'une mémoire
(define p1-mp-m1 (make  mémorisée-par:stack p1 :memory:m1))
 ;; déclaration d'une dépendance  mémorisée-par entre p1 et m1
```

Exclusion.

De la même manière que précédemment, supposons qu'un programmeur ait défini une classe de boutons. Un bouton est un objet possédant une valeur et pouvant être sélectionné ou désélectionné à l'aide des méthodes `select` ou `deselect`. Le programmeur veut exprimer une contrainte d'exclusion mutuelle entre un ensemble de boutons : un seul bouton doit être sélectionné à la fois.

Définition. Plusieurs solutions existent pour assurer une telle contrainte ; nous avons choisi la solution suivante pour sa simplicité. Cette définition, qui est inspirée de la définition d'un Synchronizer de [Frøl 94], est basée sur une précondition forte du comportement des boutons : un bouton ne reçoit les messages `select` et `deselect` que de manière alternée.

```
1  (deflink exclusion-mutuelle (:buttons)
2   :var ((active?:initform #f :accessor active?))
3   :behavior
4     (((deselect :buttons-receiver) implies (set! active? link #f))
5      ((select :buttons-receiver)   implies (set! active? link #t))
6      ((select :buttons-receiver)   permitted-if  (not (active? link)))))
```

Cette dépendance spécifie qu'un bouton n'est sélectionné que si aucun autre bouton ne l'est déjà (ligne 6). La variable `active?` représente le fait qu'un bouton est, ou non, sélectionné. Elle est mise à jour à chaque fois qu'un bouton reçoit un message (ligne 4 et 5).

Cependant, il faut pouvoir s'assurer que les participants sont dans un état cohérent avant la déclaration. Par exemple, aucun des boutons ne doit déjà être sélectionné. Nous exprimons cela comme suit :

```
(define-method action-before-effective ((lk exclusion-mutuelle) initargs)
       (for-each deselect (give lk :buttons)))
```

Déclaration. Une fois définie, la dépendance est déclarée entre plusieurs groupes de boutons. `gp1` et `gp2` sont deux groupes de boutons distincts entre lesquels existent une dépendance d'exclusion.

```
(define b1  (make button))  (define b2 (make button))
(define b3  (make button))  (define b4 (make button))
(define b5  (make bouton))  (define b6 (make bouton))
(define gp1 (make exclusion-mutuelle :buttons (list b1 b2 b3)))
(define gp2 (make exclusion-mutuelle :buttons (list b4 b5 b6)))
```

5.1.3 Vocabulaire

Définition et déclaration.

Les deux exemples précédents mettent en avant deux phases distinctes : la *définition* et la *déclaration* d'une dépendance. Nous utilisons le terme de *«dépendance effective»* lorsque nous parlons d'une dépendance entre des instances. Le terme *«dépendance»* fait toujours référence à la définition d'une dépendance. De plus, nous faisons une distinction typographique entre les dépendances et les dépendances effectives : les dépendances sont toujours écrites en italique alors que les dépendances effectives avec la typographie de `machine à écrire`.

> `p1-mp-m1` est une dépendance effective de la dépendance *mémorisée-par*.
> `gp1` et `gp2` sont des dépendances effectives de la dépendance *exclusion-mutuelle*.

A la manière du schéma classe/instances, une dépendance représente la structure et le comportement dynamique des dépendances effectives. Ainsi les dépendances effectives `gp1` et `gp2` possèdent une valeur propre pour la variable `active?` et partagent le même comportment dynamique.

Anatomie d'une dépendance.

Dans le modèle défini au chapitre 4, une dépendance est caractérisée par un nom, un ensemble de participants, une propriété que les participants de celles-ci devaient vérifiée et un comportement dynamique. De plus, une dépendance peut en tant qu'entité définir ses propres variables et comportements. Ceci est symbolisé[1] par la figure 5.3.

Comme nous l'avons expliqué au chapitre précédent, la propriété que doivent vérifier les participants d'une dépendance peut être parfois très compliquée à définir. Elle peut parfois nécessiter de

[1] Ce schéma ne tient pas compte du niveau d'abstraction fournie par la distinction classe/instances et dépendance/dépendance effective.

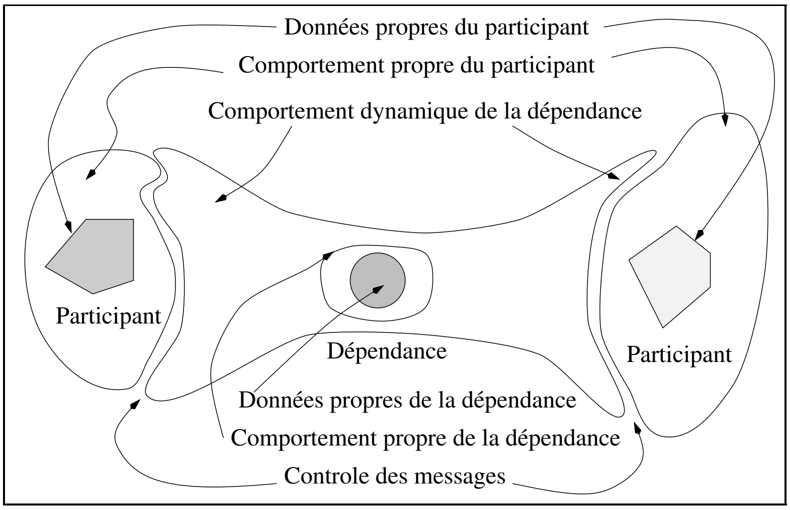


Figure 5.3: Modélisation de la séparation entre objets et dépendances.

modifier les objets pour y introduire des données supplémentaires ou d'être définie en terme de la structure des objets. . Cette façon de faire va à l'encontre de notre perception d'un objet comme d'une boîte noire dont nous ne connaissons que l'interface (un ensemble de symboles représentant des méthodes). Aussi la définition d'une dépendance n'impose pas la spécification d'une telle propriété. Nous montrons en 8.3.3 comment cette spécification peut être ajoutée.

Comportement dynamique. Le *comportement dynamique* est défini, à l'aide de `:behavior`, par la donnée des *conséquences*[2] sur les participants à la dépendance lorsque l'un d'eux reçoit un message perturbant. Comme nous l'expliquons en 4.4, l'opérateur `implies` définit un message compensatoire (ligne 1 ou 2) et l'opérateur `permitted-if` définit une condition devant être vérifiée pour autoriser l'application de la méthode concernée.

> Le comportement dynamique de la dépendance *exclusion-mutuelle* spécifie qu'un seul des boutons n'est activé à la fois. Il est spécifié comme suit :

```
1   :behavior  (((deselect :buttons-receiver) implies (set! active? link #f))
2               ((select :buttons-receiver)   implies (set! active? link #t))
3               ((select :buttons-receiver)   permitted-if  (not (active? link))))
```

Variables actives. Une dépendance définit des variables représentant les objets participants à la dépendance. Elles sont définies après le nom de la dépendance (ligne 1) :

```
1 (deflink mémorisée-par (:stack :memory) ...)
2 (make mémorisée-par :stack p1 :memory m1)
```

Ces variables ressemblent aux mots-clés LISP utilisés lors de la déclaration d'une dépendance (ligne 2). En effet, nous voulons que l'utilisateur garde à l'esprit que de telles variables font référence aux objets associés aux mots-clés lors de l'instanciation. Pour chaque dépendance effective, les variables actives représentent les objets participant à la dépendance. Un même objet peut être associé à plusieurs variables actives appartenant à diverses dépendances. L'affectation de ces variables a lieu lors de la déclaration d'une dépendance.

[2]Le terme «conséquence» doit être considéré au sens large. En effet, les gardes introduisent la notion de précondition pour l'acceptation de messages et la délégation de messages ne correspond pas exactement à une conséquence (voir en 5.2.1 et en 5.5.1).

> La dépendance *mémorisée-par* définit deux variables `:stack` et `:memory`. Pour la dépendance effective `p1-mp-m1`, la variable `:stack` réfère l'objet `p1`.
> La dépendance *exclusion-mutuelle* définit une variable active `:boutons`. Cette variable aura pour valeur la liste des boutons `i1`, `i2`, `i3` pour la dépendance effective `gp1` et `i4,i5,i6` pour `gp2`.

Variables et comportements propres. Une dépendance définit des données et des comportements propres. Nous nommons *variables* d'une dépendance, les variables décrivant les données propres de la dépendance. `:var` permet de les définir. De manière similaire aux variables d'instances en CLOS, il est possible de définir une valeur par défaut et des accesseurs pour ces variables.

> La dépendance *exclusion-mutuelle* définit une variable `active?` initialisée à faux. Celle-ci représente le fait qu'un des boutons soit activé ou non.
>
> ```
> :var ((active? :initform #f :accessor active?))
> ```

Les comportements propres d'une dépendance représentent l'ensemble des opérations qu'une dépendance effective est susceptible d'accomplir. Transposées pour des objets, ces opérations correspondent à des méthodes. Nous décrirons plus précisément de telles informations en 5.3.

5.1.4 Définition et déclaration dynamique de dépendances

Les exemples précédents ont montré que la *déclaration* dynamique de dépendances entre objets est possible. D'autre part, la *définition* dynamique de dépendances[3] est indépendante à la fois des classes des objets et des autres dépendances.

Exemple. La représentation graphique des piles n'a pas été prévue dans les fonctionnalités offertes par la classe `Stack`. Supposons que l'utilisateur veuille avoir une représentation graphique de la pile `p1` et qu'il ait à sa disposition une classe qui puisse remplir cette fonction : la classe `Single-Descriptor`. Cette classe compte parmi ses fonctionnalités les méthodes `remove-first`, `add` et `reset`.

Pour lier une instance de la classe `Stack`, `p1`, et de la classe `Single-Descriptor`, `gr-p1`, le programmeur définit la nouvelle dépendance, *représentée-par*, dont la définition est donnée ci-dessous (voir figure 5.4. Elle spécifie qu'un message `pop` sur la pile implique le retrait de la valeur graphique correspondante, qu'un message `push` sur la pile implique l'ajout de la nouvelle valeur (lignes 1 à 6). Puis, il déclare la dépendance entre les objets `p1` et `gr-p1` (ligne 8).

```
1  (deflink représentée-par (:pile :graphique)
2   :behavior
3    (((pop :pile)          implies   (remove-first :graphique))
4     ((push :pile valeur)  implies   (add :graphique ( int-to-pixels valeur)))
6     ((empty :pile)        implies   (reset :graphique))))

7  (define gr-p1 (make single-descriptor)) ;; on créé la représentation graphique
8  (define p1-graph1 (make représentée-par :pile p1 :graphique gr-p1))
```

Cette dernière définition est indépendante de celle de *mémorisée-par* alors même qu'elles concernent les mêmes objets.

5.1.5 Commentaires sur les exemples

Par la confrontation des exemples donnés ci-dessus, nous mettons en avant des caractéristiques des dépendances proposées.

[3] Au niveau du modèle, nous considérons que la définition dynamique de dépendances possible. Cependant, cette possibilité dépend essentiellement si le langage implémentant le modèle permet de créer dynamiquement les structures qui représentent les dépendances.

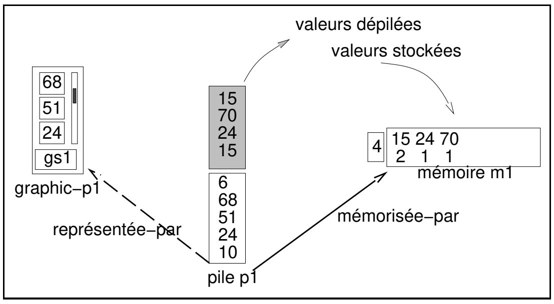


Figure 5.4: Pile, mémoire et représentation graphique

- Une dépendance est spécifiée en utilisant différents messages envoyés aux différents participants de la dépendance. Tous les messages de tous les objets participants peuvent être utilisés pour la spécification de la dépendance.
- Les dépendances sont n-aires : un groupe d'objets peut être sous influence de la même dépendance.
- Les dépendances sont non-orientées : les dépendances ne distinguent pas d'objet prédominant par rapport aux autres. Il n'y a pas de relation de *maître-esclave* entre objets comme c'est le cas dans OTHELO [FORN 90a, FORN 90b, FORN 90c, DUBO 91] ou de LOGTALK [MOUR 94].
- Contrairement aux Synchronizers[FRØL 93], le comportement dynamique ne se limite pas à changer l'état de la dépendance ou à prendre en compte cet état dans la définition des conditions d'acceptation d'un message. En effet, une dépendance décrit de manière déclarative le maintien de sa cohérence basée sur l'envoi de messages aux participants de la dépendance.
- Les dépendances factorisent le comportement et la structure des dépendances effectives.

5.2 Maintien de la cohérence

Après avoir défini une dépendance puis déclaré les objets y participant, nous montrons maintenant le maintien de la cohérence des dépendances. Le comportement dynamique d'une dépendance spécifie comment les envois de message doivent être traités afin d'assurer sa cohérence. Il est décrit à l'aide d'opérateurs dont nous rappelons le sens.

5.2.1 Opérateurs et comportement dynamique

Le maintien de la cohérence des dépendances dépend de la manière dont les dépendances sont modélisées. Un modèle simple et naturel repose sur la causalité ou la subordination des messages : une cause implique une conséquence, une action n'est possible que sous certaines conditions. Afin d'introduire ces notions, nous avons introduit deux opérateurs. Nous présentons la sémantique des opérateurs, ce qui nous conduit ensuite à présenter la syntaxe du comportement dynamique et le mécanisme de filtrage utilisé.

Opérateurs.

Notre modèle est conçu pour permettre la définition de nouveaux opérateurs (voir en 5.5.1 et en 9.4). Nous nous limitons ici à la présentation des opérateurs `implies` et `permitted-if`.

Implies. L'opérateur `Implies` associe un *message compensatoire*[4] à une méthode[5]. Cette méthode est appelée *déclenchante*. De la même manière que les contraintes de PROCOL [VAN 91] ou de ANIMUS [BORN 86b], un message compensatoire permet de définir une action qui est effectuée après l'exécution de la méthode auquel le message compensatoire est associé.

Un message n'implique de message compensatoire que si le message est envoyé à un objet participant à une dépendance pour laquelle un tel message est déclenchant.

La ligne 3 de la définition de la dépendance *mémorisée-par* donnée en 5.1.2 : `((pop :stack) implies (store :memory :result))` associe donc le message compensatoire `(store :memory :result)` à la méthode déclenchante de sélecteur `pop`. Lorsqu'un message de sélecteur `pop` est envoyé à l'objet désigné par la variable active `:stack`, le message `store` est envoyé à l'objet désigné par la variable active `:memory` avec comme argument le résultat du message `pop`.

Permitted-if. L'opérateur `permitted-if` associe une *garde* à un message. Une garde est une expression booléenne qui spécifie sous quelle condition le message auquel elle est associée pourra être autorisé. Les gardes sont supposées ne faire aucun effet de bord.

La ligne 4 de la définition de la dépendance *mémorisée-par* en 5.1.2 : `((pop :stack) permitted-if (not-full? :memory))` spécifie qu'un message de sélecteur `pop` n'est exécuté sur `:stack` que si `:memory` n'est pas pleine.

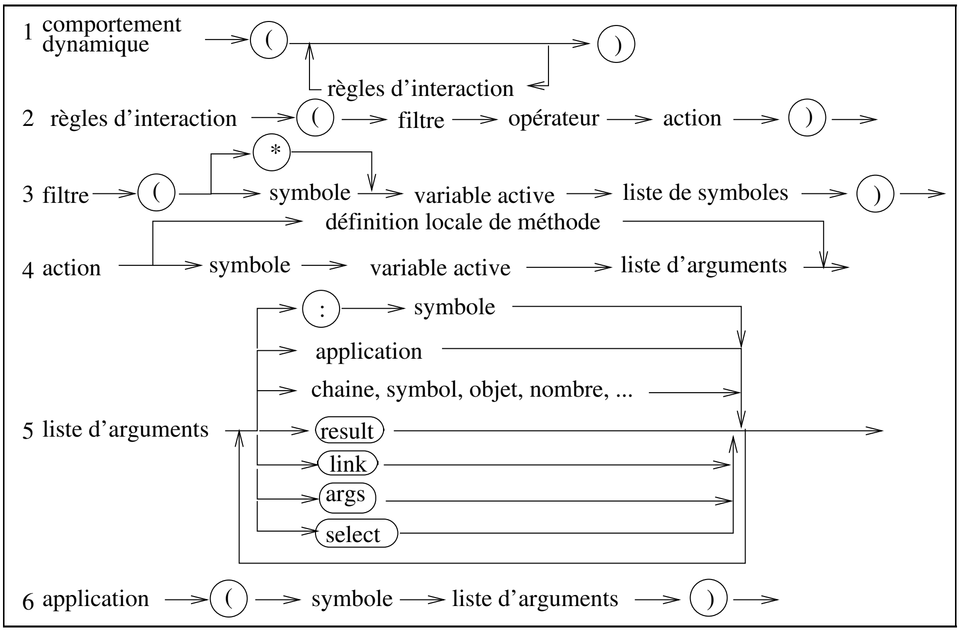


Figure 5.5: Syntaxe du comportement dynamique.

Syntaxe et filtrage.

Les diagrammes syntaxiques de la figure 5.5 décrivent la grammaire du comportement dynamique.

[4]Nous avons choisi ce terme par analogie aux *compensating messages* de [MEDE 91] qui sont des messages permettant d'assurer la cohérence de contraintes d'intégrité entre éléments de bases de données.

[5]Plus précisément, il associe un message compensatoire à la réception par un objet d'un message dont le sélecteur est associé à une méthode.

Le comportement dynamique est composé de *règles d'interaction* spécifiées en termes des interfaces des objets. Une règle d'interaction est l'association d'un message au travers d'un filtre, d'un opérateur et d'une action (règle 2). Un filtre est composé d'un sélecteur de méthode (un symbole ou un joker `*` qui spécifie n'importe quelle méthode), d'une variable active et si nécessaire d'une liste de symboles. La communication dans le modèle objet choisi est l'envoi de message, donc la donnée d'une variable active est obligatoire[6] (règle 3).

Une action définit l'appel à une méthode (règle 4). Cette méthode peut être définie localement. Dans ce cas, la syntaxe de définition du langage hôte est utilisée. Sinon il s'agit d'invoquer une méthode en précisant le sélecteur, le receveur et les arguments d'appels.

Filtrage. Les symboles d'un filtre (règle 3) sont liés par filtrage aux arguments d'appel du message (voir Figure 5.6). Par ce biais, les arguments d'appels sont utilisés dans la définition de la liste d'arguments de l'action associée au filtre (règles 2 et 5). La liste d'arguments (règle 5) peut être composée de nombres, chaînes, mots-clés ou d'applications de nouveaux messages sur une nouvelle liste d'arguments.

Dans la règle d'interaction de la dépendance *mémorisée-par*: `((push:pile valeur) implies (add:graphique (`*int-to-pixels*` valeur)))` le symbole `valeur` de la liste d'arguments de l'action a pour valeur la valeur du second argument lors de l'invocation de la méthode `push` sur l'objet `:pile`. Pour le message `(push p1 12) valeur` vaut 12.

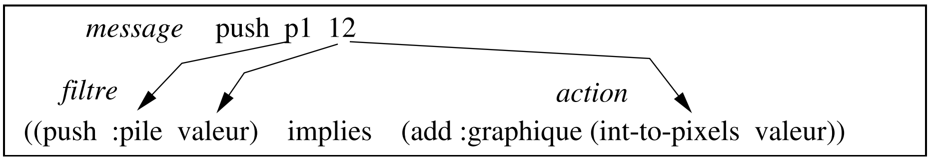


Figure 5.6: Un exemple de filtrage.

Le filtrage des arguments d'appels d'une méthode permet au concepteur de la dépendance de manipuler ces arguments et d'appliquer des méthodes à ces arguments. Ceci dans plusieurs buts :

– à la même manière d'un gluon [Pint 93], la dépendance peut convertir les arguments ou en générer de nouveaux. Les conversions de types (chaîne vers tableau, entier vers pixels) sont facilement exprimables.

  L'action dont le code est `(add :graphique (`*int-to-pixels*` valeur))` permet de convertir la valeur d'un argument avant de l'afficher.

– Une dépendance peut aussi adapter l'ordre et le nombre des arguments pour permettre la communication entre objets à la manière des Adaptors [Yell 94]. Cette information (la conversion ou l'adaptation) est alors liée à la dépendance[7]. La conversion peut être définie comme une méthode propre d'une dépendance (voir en 5.3.1).

Mot-clés.

Le statut des dépendances et leur spécificité rend nécessaire l'introduction de mots-clés.

link permet de faire référence à la dépendance effective elle-même, comme self en Smalltalk. La perception d'une dépendance comme un objet ayant un comportement et une structure propres a rendu l'introduction de cette variable fondamentale.

[6] Par contre, pour être uniforme avec la syntaxe du modèle ObjVlisp, il aurait été possible de définir un filtre comme : une variable active, un sélecteur et de possibles arguments.

[7] Elle n'est plus traitée dans un des objets participants comme cela est le cas avec les langages qui ne proposent pas d'entités de liaison entre objets. Dans ces langages, la solution qui consiste à définir de nouvelles couches de méthodes n'est guère satisfaisante car l'objet perd de son autonomie et sa réutilisabilité.

result représente le résultat de l'application de la méthode du filtre d'une règle d'interaction. Cette variable n'apparaît que dans l'action d'une règle d'interaction d'opérateur `implies`.

La variable `result` dans la règle d'interaction donnée ci-dessous :
`((pop:stack) implies (store:memory result))` représente le résultat rendu par l'appel de la méthode `pop` sur la variable `:pile`; c'est-à-dire la valeur dépilée.

:xxx-receiver représente l'objet receveur du message. Les dépendances gérant un groupe d'objets ont besoin de cette variable afin de préciser l'objet du groupe ayant reçu un message déclenchant. Le programmeur remplace `xxx` par la variable active qu'il utilise pour introduire un ensemble d'objet.

Dans la dépendance *exclusion-mutuelle*, les boutons sont spécifiés sous forme d'une liste de boutons : `(make exclusion-mutuelle :boutons (list b1 b2 b3))`
Par contre, il est nécessaire de pouvoir faire référence à un objet distinct du groupe : `((select :boutons-receiver) implies ....)`

***, selector, args.** La variable `*` agit comme un joker pour les sélecteurs de méthodes signifiant que la règle d'interaction est définie pour toutes les méthodes de l'objet receveur (ceci permet de définir très facilement des dépendances espionnant les objets). Les variables `selector` et `args` représentent le sélecteur de la méthode et la liste des arguments d'appel.

Il était inutile d'introduire une variable `receveur` car une telle variable aurait été redondante avec les variables actives représentant les objets participants.

5.2.2 D'un état cohérent à un autre

Le maintien de la cohérence présuppose que les objets participants sont cohérents par rapport à la dépendance avant la déclaration de celle-ci. La dépendance est cohérente, une modification l'invalide ou veut l'invalider et le système, en utilisant le comportement dynamique, la rétablit ou l'empêche. Grâce à la méthode de dépendance `action-before-effective`, notre modèle permet la spécification d'actions sur les objets qui participent à une dépendance avant la déclaration de celle-ci. Cet aspect est illustré par la dépendance *exclusion-mutuelle* définie en 5.1.2.

D'autre part, lorsque la cohérence d'une dépendance peut être spécifiée de manière explicite sous forme d'un prédicat, la donnée d'une procédure assurant la cohérence de la dépendance est nécessaire. Le système s'assure que les objets participants sont dans un état cohérent. Ces aspects sont illustrés par la dépendance *aligner* entre des points. Outre la donnée du comportement dynamique, on spécifie le prédicat `coherent?` qui vérifie si deux points sont alignés et la procédure `do-coherence` qui positionne les points de manière à vérifier le prédicat. Ces deux méthodes sont invoquées durant la déclaration de la dépendance (voir en 8.3.3).

```
(deflink  aligner (:point1 :point2)
 :access ((point1 :point1) (point2 :point2))
 :behavior
  (((move :point1 x y)        implies  (move :point2 x (y :point2)))
   ((move :point2 x y)        implies  (move :point1 x (y :point1)))
   ((translate :point2 dx dy) implies  (translate :point1 dx dy))
   ((translate :point1 dx dy) implies  (translate :point2 dx dy))))

(define-method coherent? ((lk aligner))
   (= (x (point1 lk)) (x (point2 lk))))

(define-method do-coherence ((lk aligner))
   (move (point2 lk) (x (point1 lk)) (y (point2))))
```

5.2.3 Maintien de la cohérence

Jusqu'à présent, nous n'avons pas abordé la façon dont le contrôle induit par les dépendances s'intègre au flot de messages entre les objets. Nous montrons maintenant comment le comportement dynamique permet de maintenir la cohérence d'une dépendance en contrôlant les messages entre ses différents participants.

Avant de devenir l'un des *«participants»* [Helm 90] d'une dépendance, un objet est dit *«libre»* : les messages qui lui sont adressés sont traités normalement.

> Pour p2, une instance de la classe `stack` dans la figure 5.7 qui est libre, le message `pop` conduit normalement à l'exécution de la méthode.

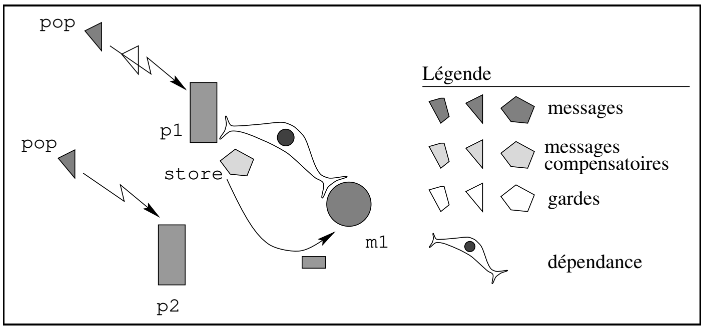


Figure 5.7: Influence d'une dépendance.

Par contre, dès qu'un objet participe à une dépendance, le système contrôle les messages qui lui sont envoyés afin d'assurer la cohérence des dépendances.

> Dans la figure 5.7, le message `pop` envoyé à l'instance `p1` est contrôlé car il existe une garde et un message compensatoire pour l'objet `p1` et le sélecteur `pop`. Donc la garde doit être évaluée, si elle est vraie la méthode associée au sélecteur du message est exécutée et ensuite le message compensatoire est envoyé : il s'agit du message `store` à l'instance `m1` avec la valeur retirée de `p1`.

L'ensemble des dépendances effectives constitue un graphe. Nous considérons deux types de contrôle : un local et un global. Lorsqu'un message est envoyé à un objet participant à plusieurs dépendances effectives, le contrôle local assure la consistance de celles-ci. Le contrôle global quant à lui assure la consistance de l'ensemble du graphe.

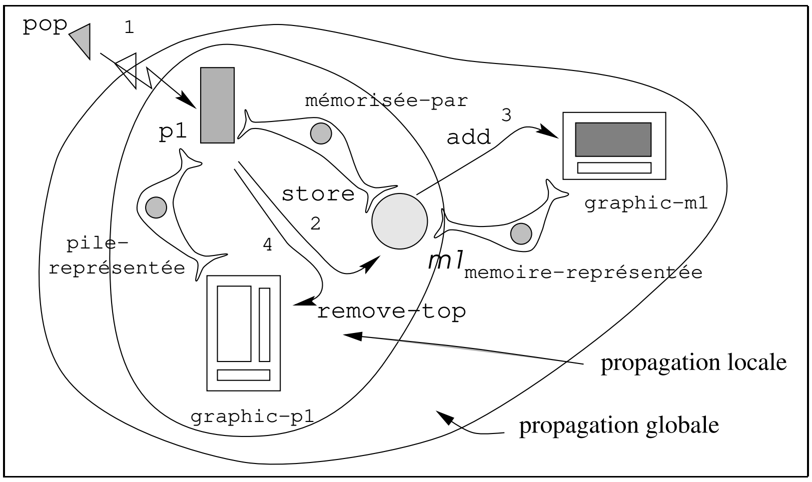


Figure 5.8: Deux étapes dans la propagation

Contrôle local. Nous présentons ici le contrôle local standard. Cependant, comme nous le montrons au chapitre 9, notre modèle permet de modifier ce contrôle. Ainsi lorsqu'un objet reçoit un message, *toutes* les gardes définies pour ce message dans toutes les dépendances concernant cet objet doivent être vérifiées pour que la méthode invoquée soit appliquée. Après l'application de cette méthode, tous les messages compensatoires sont invoqués.

> Dans la figure 5.2 une pile `p1` est liée à une mémoire `m1` et à sa représentation graphique `graphic-p1`; aussi l'envoi d'un message dont le sélecteur est `pop` à `p1` implique : la vérification de la garde concernant la mémoire et si la garde est vraie, l'application de la méthode `pop` puis l'appel de la méthode `store` sur `m1` et de `remove-top` sur `graphic-p1`.

Contrôle global. Les messages compensatoires peuvent eux-mêmes être contrôlés. Ce dernier point induit la nécessité d'un contrôle global du flot de propagation. Le contrôle global doit assurer que l'ensemble des dépendances est cohérent. Cependant, un ordre d'exécution doit être choisi. Le comportement du contrôle global est de répéter le comportement local pour tous les messages compensatoires. Ainsi la propagation des messages se réalise en profondeur comme illustré par la séquence des messages (1) et (2) dans la figure 5.8. Par contre, nous ne définissons pas au niveau du modèle d'ordre pour l'exécution de deux messages compensatoires *non consécutifs* comme les messages (2) et (4).

> La figure 5.8 illustre la propagation de messages dans le graphe de dépendances suivant : `p1` est *mémorisée-par* `m1`, `p1` est *représentée-par* `graphic-p1` et la mémoire `m1` est aussi représentée par un objet graphique `graphic-m1`. Un message `pop` envoyé à `p1` (1) implique l'envoi de `store` à `m1` (2). Ce message implique la mise à jour de `graphic-m1` (3). Le message `pop` sur `p1` implique la mise à jour de `graphic-p1` (4). Seule l'exécution du message (3) après le message (2) est spécifié, ainsi le message (2) aurait pu intervenir avant le séquencement des messages (2) et (3).

5.3 Dépendances = objets

Jusqu'à présent, nous avons porté notre attention sur la définition du comportement dynamique des dépendances en laissant de côté ce que nous avons appelé les *informations propres d'une dépendance*. Nous comblons ici ce retard en présentant comment une dépendance définit à la fois des données et des comportements propres. La réification des dépendances offre une solution à la fois simple, puissante et uniforme pour représenter les données et les comportements de telles dépendances. Elle permet d'exprimer des dépendances entre dépendances. D'autre part, le choix d'un modèle objet dans lequel les classes sont des objets offrent la possibilité d'exprimer des dépendances dont les participants sont des classes.

5.3.1 Variables et méthodes de dépendances

Nous reprenons ici l'exemple d'exclusion entre boutons présenté en 5.1.2. Par cet exemple, nous montrons qu'une dépendance reste le lieu privilégié pour la définition d'informations propres à la coordination ou aux contraintes entre objets.

Boutons et exclusion.

Les boutons que nous modélisons peuvent être activés par la méthode `select` et désactivés par la méthode `deselect`. Pour pouvoir comparer, les dépendances aux Synchroniseurs, nous appliquons la même précondition que FRØLUND : un bouton ne reçoit pas deux fois de suite le même message `select` ou `deselect`.

Dans une colonne de radio-boutons, un seul bouton du groupe doit être actif. Deux approches peuvent assurer ce principe d'exclusion : soit un bouton n'est activé que si aucun autre bouton ne l'est déjà, soit l'activation d'un bouton nécessite de désactiver le bouton précédemment activé (comme illustré par la figure 5.9).

Ces deux possibilités illustrent bien la place donnée à la réactivité. La première solution assure la coordination en empêchant le cas échéant un bouton d'être sélectionné et en agissant

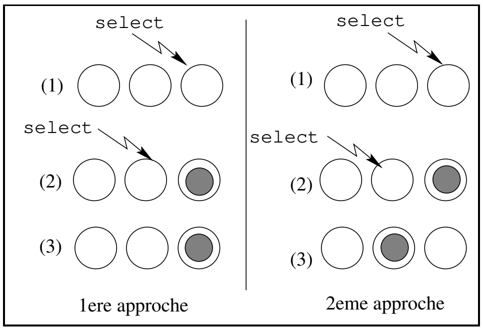


Figure 5.9: Deux stratégies d'exclusion.

exclusivement sur l'état de la dépendance. La seconde maintient par réaction l'état d'exclusion mutuelle qui existe entre les boutons ; les actions agissent à la fois sur la dépendance effective et sur les boutons. Contrairement au Synchronizers, notre modèle permet d'exprimer les deux types de solutions. La première solution est celle qui a été définie en 5.1.2.

Une solution réactive entre objets. La seconde solution est exprimée à l'aide de la dépendance *exclusion-mutuelle-reactive*. Celle-ci définit deux variables locales `active?` et `last-button-selected`, ce qui permet de savoir : si un bouton est sélectionné ou non (ligne 2) et, dans le cas d'une sélection, le dernier bouton sélectionné (ligne 3).

```
1  (deflink  exclusion-mutuelle-réactive  (:buttons)
2    :var ((active?:initform #f  :accessor active?)
3           (last-button-selected :accessor selected))
4    :access ((group-of-buttons :buttons))
5    :behavior
6      (((deselect :buttons-receiver) implies (set! active? link #f))
7       ((select :buttons-receiver)
8        implies-before (when (active? link) (deselect (selected link))))
9       ((select :buttons-receiver)
10       implies (set! active? link #t)
11                (set! selected link :buttons-receiver))))
12  (define-method action-before-effective ((lexclusion-mutuelle-réactive) args)
13       (for-each deselect (grouf-of-buttons l)))
```

Le comportement dynamique[8] définit que : lors de la sélection d'un bouton, les variables de la dépendance sont affectées pour représenter cet état (lignes 9, 10 et 11). Les lignes 7 et 8 précisent que la sélection d'un bouton implique de désélectionner, si besoin, le bouton précédemment sélectionné. Lors d'une désélection[9], les variables décrivent ce nouvel état (ligne 6). La déclaration d'une telle dépendance est semblable à celle de la dépendance *exclusion-mutuelle* (voir en 5.1.2). Cette définition présuppose qu'aucun des boutons n'est sélectionné avant que la dépendance ne soit déclarée, ce que l'on exprime aux lignes 12 et 13.

Une digression : l'opérateur implies-before. Nous avons présenté deux opérateurs en 5.2.1 et précisé que notre modèle prend en compte la définition de nouveaux opérateurs. L'opérateur `implies-before` (ligne 8) est une variante de l'opérateur `implies`, il spécifie qu'un message compensatoire s'effectue avant, et non après l'exécution de la méthode contrôlée comme c'est le cas avec

[8] Définir les dépendances *exclusion-mutuelle* et *exclusion-mutuelle-reactive* sans la précondition d'alternance des messages `select` et `deselect` ne pose pas de problèmes : il suffit de faire attention au fait que désélectionner un bouton non sélectionné n'implique pas obligatoirement de changer l'état de la variable `active?`.

[9] Il s'agit du bouton sélectionné car, d'après la précondition, aucun autre bouton n'est désélectionné. En effet, pour recevoir un message de désélection, un bouton doit avoir été au préalable sélectionné.

l'opérateur `implies`. L'utilisation de l'opérateur `implies-before` est nécessaire si l'on souhaite avoir une réelle exclusion. L'opérateur `implies` n'assure pas une telle exclusion car deux indicateurs sont sélectionnés en même temps durant l'exécution de l'action de rétablissement.

Informations propres.

Une dépendance est l'endroit privilégié pour définir des informations propres à l'interaction entre les objets.

Variables d'instances de dépendance. Pour la dépendance *exclusion-mutuelle-réactive*, les informations permettant de savoir si un des boutons est sélectionné ne sont pas stockées dans les boutons mais bien dans la dépendance effective. Les variables `active?` et `last-button-selected` sont des propriétés propres à la dépendance. Ainsi chaque bouton n'a pas à posséder de variables représentant le fait qu'un bouton du groupe soit sélectionné.

Les dépendances permettent de faire une distinction logique entre les informations représentant les objets et les informations représentant les contraintes entre ces objets. En faisant explicitement référence à la dépendance et à son comportement propre, le comportement dynamique est plus riche. Le comportement dynamique de la dépendance peut modéliser des situations plus complexes (comme des automates) que des règles causales simples.

Accès. Une dépendance définit des méthodes d'accès aux objets y participant. Ces dernières enrichissent l'interface de la dépendance et permettant une utilisation uniforme des dépendances. Cette possibilité va dans le même sens que les «noms de rôle» de la méthode OMT [Blah 92, Blah 95].

Ainsi, étant donnée une dépendance effective et une variable active, un accès rend l'objet associé. Un accès est associé à une et une seule variable de la dépendance. La syntaxe est la suivante :

```
:access '((nom-de-l'acces variable-associée) ...)
```

La dépendance *exclusion-mutuelle-réactive* définit la méthode d'accès `group-of-buttons` à la ligne 4.

```
(define gp1 (make exclusion-mutuelle-réactive:buttons (list i1 i2 i3))
(group-of-buttons gp1) -> (i1 i2 i3)
```

De la même manière, pour la dépendance *mémorisée-par*, l'accès à la mémoire et à la pile se définit comme suit :

```
(deflink mémorisée-par
   :access '((stack-of :stack) (memory-of :memory))
  ...)
```

Méthodes de dépendances. Une méthode de dépendance définit le comportement propre d'une dépendance qui a deux origines possibles : soit il est lié à la gestion des dépendances (création, destruction, maintien de cohérence...), soit il définit le comportement spécifique d'une dépendance.

- La réification des dépendances permet de définir le comportement de toutes les dépendances exprimées. Ce comportement spécifie les aspects communs de la *vie* des dépendances : la création, la déclaration, le maintien de la cohérence des dépendances, l'ajout, le retrait d'objets participants. L'ensemble de ces fonctionnalités et leurs interactions constitue un protocole pour la gestion des dépendances. La troisième partie de cette thèse y est entièrement consacrée (voir les chapitres 8 et 9).

  L'ajout d'un bouton à l'ensemble des boutons en exclusion est possible à l'aide de la méthode `add-object-in-link` comme le montre l'exemple suivant :

```
(define b1 (make button)) (define b2 (make button))
(define b3 (make button))
(define gp3 (make exclusion-mutuelle-reactive :buttons (list b1 b2 b3)))
(define i4 (make button))
(group gp3) -> (b1 b2 b3)
(add-object-in-link gp3 b4 :buttons)
(group gp3) ->  (b1 b2 b3 b4)
```

Bien d'autres actions peuvent être nécessaires, comme par exemple, savoir si la dépendance est cohérente, pouvoir définir une action lors de la déclaration d'une dépendance (voir 5.1.2). Ces méthodes font partie du protocole de gestion des dépendances.

Si l'on veut exprimer que lors de la déclaration de la dépendance effective d'exclusion, un des boutons doit être activé, il suffit de définir la méthode `action-after-creation`. La méthode `give` rend l'objet participant associé à une variable active. Ces deux méthodes font partie des méthodes permettant de spécialiser le comportement des dépendances.

```
(define-method action-after-creation ((lk  exclusion-mutuelle-réactive))
    (select (car (give lk :buttons))))
```

- Outre la spécialisation de comportements prédéfinis, il est possible de définir des comportements spécifiques à une dépendance, de la même manière que les méthodes sont définies pour les objets.

La conversion de types peut être associée à une dépendance. Ainsi, la dépendance *représentée-par* peut définir une règle d'interaction de la forme : `((push:pile valeur) implies (add:graphique (`*`int-to-pixels`*` link valeur)))`
`int-to-pixels` est une méthode définie sur la dépendance *représentée-par*. Cette méthode peut être modifiée ou spécialisée indépendamment des objets mis en dépendance.

La méthode `select-next` définit pour la dépendance d'exclusion mutuelle permet de sélectionner le prochain bouton.

```
(define-method select-next ((lk exclusion-mutuelle-réactive))
   (select (if (active? lk)
               (next (give lk:boutons) (selected lk))
               (car (give lk:boutons)))))
```

5.3.2 Des dépendances entre dépendances

Les dépendances expriment des contraintes entre objets. Or, elles sont elles-mêmes représentées dans notre modèle par des objets, donc il est possible d'exprimer des dépendances dont les participants sont des dépendances.

De telles dépendances permettent d'exprimer de manière puissante des propriétés entre des dépendances, comme l'exclusion, l'inverse... Les dépendances entre dépendances sont très utiles pour la modélisation de connaissances. Les auteurs de Othelo ont introduit ce concept et l'ont utilisé dans le domaine de la représentation des connaissances [Forn 90a]. Nous considérons de telles dépendances comme un outil de manipulation des dépendances s'intégrant à notre modèle de manière totalement uniforme. De plus, elles permettent d'appliquer les mêmes règles relatives à l'indépendance et à la réutilisabilité des objets aux dépendances.

Nous montrons maintenant à titre d'exemple comment la dépendance *inverse* peut être exprimée. Cette dépendance nous permet alors d'exprimer les relations que l'on trouve dans Merise [Nanc 92, Dery 96b].

La dépendance inverse. La dépendance *inverse* spécifie que si une dépendance r existe entre deux objets x et y, alors la dépendance r^{-1} existe entre y et x. La dépendance inverse est une dépendance binaire. Elle nécessite que ces participants soient des dépendances elles aussi binaires et orientées.

```
1 (deflink  inverse   (:masterlink :slavelink)
3 :behavior
4   (((create :masterlink obj1 obj2)
5      implies (create :slavelink obj2 obj1))
6    ((create :slavelink  obj2 obj1)
7      permitted-if (if-not-exist :slavelink obj2 obj1))))
```

Supposons que nous ayons défini les dépendances binaires *est-père-de* et *est-fils-de* et deux objets sur lesquels de telles dépendances peuvent s'appliquer : ici des humains `césar` et `brutus`. Déclarer que *est-père-de* est inverse de *est-fils-de* (ligne 10) assure qu'à la création d'une dépendance effective *est-père-de* entre `césar` et `brutus` (ligne 11), une dépendance *est-fils-de* entre `brutus` et `césar` sera automatiquement créée. L'utilisation de la méthode `create` qui fixe l'ordre et le nombre des arguments nous évite d'associer explicitement les participants aux variables actives.

```
8   (define brutus (make Human:name "brutus"))
9   (define césar  (make Human:name "césar"))
10  (create inverse est-père-de est-fils-de)
11  (create est-père-de césar brutus)   ;; crée automatiquement
12  ;; la dépendance effective est-fils-de brutus césar
```

Cependant, cette définition n'assure pas qu'à la création de la dépendance effective *est-fils-de* entre `brutus` et `césar`, la dépendance *est-père-de* entre `césar` et `brutus` soit créée.

Pour ce faire, une première solution est de définir explicitement la dépendance *inverse* entre la dépendance *est-fils-de* et *est-père-de*. Une seconde solution beaucoup plus élégante et puissante consiste à utiliser la réflexivité naturelle de la dépendance *inverse* : la dépendance *inverse* est inverse d'elle-même (ligne 13). De ce fait, la création de la dépendance *inverse* entre *est-père-de* et *est-fils-de* (ligne 14) entraine la création automatique de la dépendance *inverse* entre *est-fils-de* *est-père-de*. Ainsi lorsqu'une dépendance effective est créée par exemple *est-fils-de* entre `brutus` `césar`, la dépendance effective *est-père-de* entre `césar` `brutus` l'est aussi.

```
13 (create inverse inverse inverse)
14 (create inverse est-père-de est-fils-de)
15 ;; crée automatiquement  est-fils-de inverse est-père-de
16 (create est-fils-de brutus césar)
17 ;; crée automatiquement césar est-pere-de  brutus
```

5.3.3 Dépendances entre classes : aux limites du modèle

Supposons que nous souhaitions disposer d'une classe `Memorised-Stack` dont chaque instance est associée de manière automatique à une instance de la classe mémoire par une dépendance effective de la dépendance *mémorisée-par*. En plus, nous voulons toujours pouvoir créer des instances de la classe `Stack`.

Approche traditionnelle par redéfinition de la méthode make. La solution illustrée par la Figure 5.10 consiste à créer une sous-classe de la classe `Stack` et à particulariser la méthode de création `make` de telle sorte que la création d'une instance de `Memorised-Stack` implique la création d'une dépendance effective entre chaque instance de cette classe et des instances de la classe `Memory`.

Cette solution peut nécessiter d'introduire des variables d'instances pour référencer les classes des objets devant être liés et la dépendance.

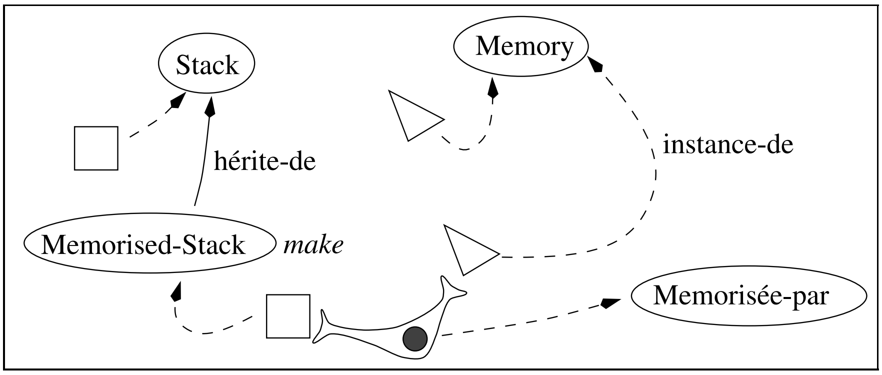


Figure 5.10: Mise en œuvre automatique de dépendances par redéfinition de `make` qui gère la création des instances et de leur mise en dépendance.

En définissant des dépendances entre classes.

Dans le modèle à classes que nous avons choisi, les classes sont des objets. Cela nous amène à considérer l'expression de dépendances entre classes. Deux cas se présentent : une dépendance peut être exprimée (a) uniquement entre classes ou (b) entre des classes et des instances.

> Une dépendance peut être exprimée entre une classe et la classe de la représentation graphique de ses instances. Ainsi chaque instance est liée par une dépendance à sa représentation graphique comme le montre la figure 5.11. D'autre part, une classe peut être liée par une dépendance à une instance la représentant.

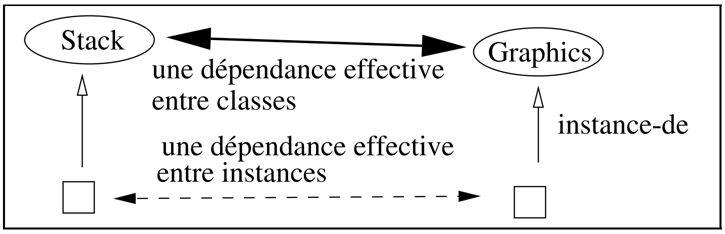


Figure 5.11: Une dépendance entre classes pour définir automatiquement des dépendances entre leurs instances. instances.

Une autre solution à notre problème consiste alors à définir une dépendance au niveau des classes des objets. La figure 5.12 illustre cette situation. La dépendance *BetweenMS-M* est définie entre la classe `Memorised-Stack`, la classe `Memory` et la dépendance `mémorisée-par`.

```
(deflink betweenMS-S  (:msclass :mclass :depclass)
 :behavior
  '(((make :msclass . args)
    implies
    (make :depclass :memory (make :mclass) :stack result))))

(make betweenMS-S :msclass Memorised-Stack :mclass Memory :depclass mémorisée-par)
```

La dépendance *betweenMS-S* précise qu'à la création d'une instance de la classe `:msclass`, une instance de la classe `:mclass` est créée pour être liée par une dépendance effective instance de la classe `:depclass` Il est possible d'écrire directement le nom de la classe `Memory` et la dépendance *mémorisée-par* au lieu d'utiliser les variables actives `:mclass` et `:depclass`.

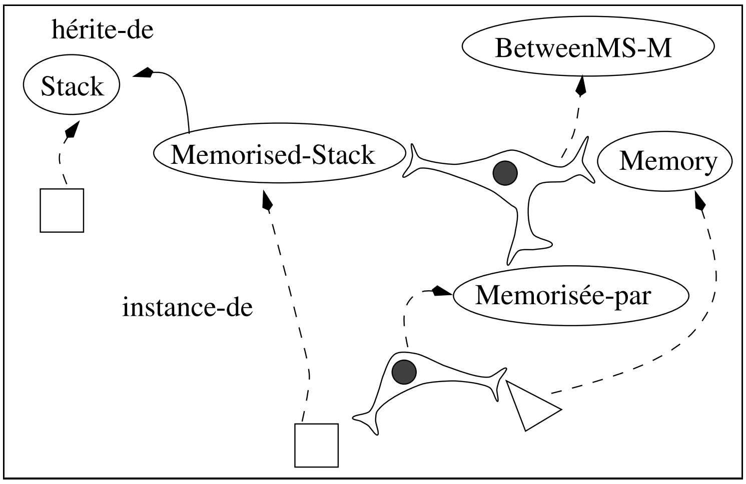


Figure 5.12: Mise en œuvre automatique de dépendances à l'aide de dépendances entre classes. Le comportment dynamique gère la mise en dépendance et la création des instances.

Discussions. La seconde solution offre le résultat attendu. Cependant, des questions se posent quant à l'utilisation des dépendances entre classes. Nous pensons que la définition de dépendances entre classes est justifiée lorsque le comportement de celles-ci ne se limite pas à contrôler exclusivement la création d'instances. En effet, lorsque la définition d'une dépendance entre classes ne spécifie qu'un comportement lié à la création d'instances, il est préférable de spécialiser la méthode `make` prévue à cet effet par le modèle objet utilisé.

Nous considérons que la première solution spécialise bien le comportement de la classe par l'utilisation de la méthode `make` et que dans ce cas précis il est préférable de l'utiliser. Dans la première solution la création d'une sous-classe a plus de sens que dans la seconde, le comportement de création est spécialisé. Cependant, la seconde solution est adaptée lorsque les dépendances exprimées ne se limitent plus à un comportement similaire à une méthode auxiliaire `:after` de CLOS; c'est-à-dire lorsqu'il s'agit de spécifier une *«interaction»* entre classes.

Limites. La définition de dépendances entre classes pose un problème lors de la spécialisation de ces classes. Ce problème a deux causes : la première est que les dépendances ne sont pas définies et liées aux classes et la seconde que les dépendances peuvent liées indifféremment des classes entre elles, des instances entre elles ou des classes et des instances.

Aucune des deux approches précédentes n'offre de solutions satisfaisantes et généralisables. Ainsi une classe peut évoluer par spécialisation et cette évolution peut soit amener à définir une nouvelle dépendance, soit à enrichir la dépendance.

Comme le notre la figure 5.13, plusieurs problèmes existent : tout d'abord, nous ne pouvons pas savoir si lors de la spécialisation d'une classe, toutes les classes participant à la dépendance sont sous-classées. D'autre part, nous ne savons pas si la dépendance reste valide par rapport à la spécialisation. Et finalement, nous ne savons pas si la nouvelle dépendance doit être réellement héritée de la dépendance qui existe entre les classes ancêtres. Des problèmes similaires se posent pour les dépendances portant sur des classes et des instances.

Nous n'avons pas introduit de changement dans l'héritage classique proposé par le modèle ObjVlisp pour qu'il prenne en compte l'existence de dépendances au niveau des classes. Nous avons donc choisi de laisser cet aspect à la charge du programmeur : il doit clairement expliciter le comportement qu'il souhaite obtenir.

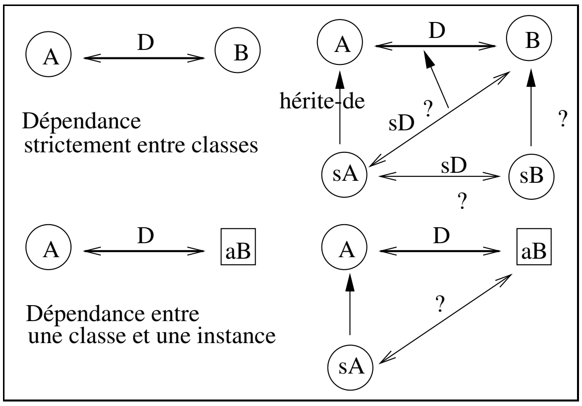


Figure 5.13: Problèmes lors de la spécialisation de classes participant à des dépendances entre classes .

5.4 Héritage entre dépendances

Les dépendances en tant qu'objets à part entière sont définies de manière incrémentale afin de permettre une plus grande réutilisation. L'héritage entre dépendances s'applique aux composants d'une dépendance : la structure et le comportement propre, les accès aux participants et le comportement dynamique de la dépendance (voir en 5.1.3). L'héritage proposé est à la fois classique pour l'héritage de la structure, du comportement propre et des accès et spécifique pour le comportement dynamique.

Dans la suite, nous appelons une *sous-dépendance*, une dépendance qui hérite d'autres dépendances dites ses *sur-dépendances*. Nous appelons les règles d'interactions définies dans une sous-dépendance des règles d'*interaction locales*, par opposition à celles héritées.

5.4.1 Structure et méthodes

L'héritage des variables et du comportement propre s'apparente à un héritage classique entre classes. Il est alors possible d'ajouter des variables, de spécialiser les méthodes des dépendances. Le modèle proposé utilise l'héritage proposé par les langages sur lesquels est basé notre modèle.

Dans l'exemple, ci-après, la dépendance *mémorisée-par* hérite de la dépendance *no-reenter-link* et s'enrichit de tous les comportements et variables de dépendances qui assurent la gestion des cycles.

```
(deflink  mémorisée-par
  :inherit '((no-reenter-link))
   ...)
```

5.4.2 Accès

Les accès permettent d'accéder aux objets participant à une dépendance. Ainsi étant données une dépendance effective et une variable active, un accès rend l'objet associé à la variable. L'héritage des accès spécifie qu'un accès défini sur une sur-dépendance est hérité par la sous-dépendance.

5.4.3 Comportement dynamique

Le comportement dynamique d'une dépendance nécessite la définition d'un héritage spécifique. L'héritage proposé est un héritage multiple permettant une fusion des règles d'interaction, basée sur le renommage des variables des dépendances.

Héritage simple.

L'héritage simple entre deux dépendances est défini de la façon suivante : lorsque les règles d'interaction héritées sont *différentes* de celles locales, les interactions héritées s'ajoutent aux interactions locales pour définir le nouveau comportement de la sous-dépendance. Lorsqu'elles sont *compatibles*, elles sont fusionnées. Il faut donc préciser les critères qui nous permettent de dire si deux règles d'interactions sont *différentes* ou *compatibles*.

Compatibilités de règles d'interactions et renommage. Soient deux interactions `I1` et `I2` définies comme suit :

```
I1 = sélecteur1 signature1 opérateur1 action1
I2 = sélecteur2 signature2 opérateur2 action2
```

`I1` et `I2` sont *différentes* si :

- `sélecteur1` et `sélecteur2` sont différents ou si,
- `opérateur1` et `opérateur2` sont différents ou si,
- les éléments de `signature1` et de `signature2` ne permettent pas d'établir un filtrage basé sur le nom des symboles contenus dans les deux signatures. Le filtrage doit être capable de proposer une liste d'équivalence des éléments d'une des signatures dans l'autre. Cette liste est utilisée pour renommer[10] les éléments de la signature dans l'action lors de la fusion des règles.

L'incompatibilité de deux règles d'interactions peut être due à une différence de variables actives. Aussi lors de la définition d'une sous-dépendance une *clause de renommage* peut être associée à chaque sur-dépendance dont cette sous-dépendance hérite (voir ligne 9). Ainsi deux règles d'interactions peuvent être différentes avant renommage mais compatibles à la suite de celui-ci.

Exemple. Nous illustrons ici l'héritage du comportement dynamique. Soient deux dépendances *R1* et *R2* telles que *R2* hérite de *R1* dont les définitions sont :

```
1 (deflink  R1   (:x :y :z)
2   :behavior
3    (((m1 :x) implies (A1))
4     ((m2 :y) permitted-if (B1?))
5     ((m3 :z) implies (C1))
6     ((m1 :y) implies (D1))))
7 (deflink  R2   (:x :y :z)
8   :behavior
9   :inherit (( R1 (rename :z as :x)))
10   (((m2 :x) implies (F2))
11    ((m1 :x) implies (A2))
12    ((m5 :w) implies (C2))
13    ((m3 :x) implies (G2))
14    ((m2 :y) permitted-if (G2?))))
```

La dépendance `R2` possède le comportement dynamique suivant :

```
15 (((m1 :x) implies (A1 A2))        ;; fusion : une séquence d'exécutions
16  ((m2 :y) permitted-if (B1? G2?)) ;; fusion : une conjonction de gardes
17  ((m2 :x) implies (F2))            ;; définition locale
18  ((m3 :x) implies (C1 G2))         ;; renommage + fusion
19  ((m5 :w) implies (C2))            ;; nouveau participant défiintion locale
20  ((m1 :y) implies (D1)))           ;; ajout d'une règle héritée
```

[10] Il faut aussi renommer les variables dites *libres* qui pourraient être *capturées* lors du renommage.

Dans la dépendance `R2` à la ligne 12, un nouveau participant est ajouté lors de la définition locale d'une nouvelle interaction.

Choix de l'addition ou de la fusion des règles d'interactions. Soient `sous-dépendance` une sous-dépendance et `sur-dépendance` la dépendance dont elle hérite. Le comportement dynamique après héritage de la sous-dépendance est spécifié par la fonction `Hérite` définie ci-après.

```
Hérite(sous-dépendance, sur-dépendance)
   SD := Renomme(sur-dépendance,
                 Clause-de-renommage(sous-dépendance, sur-dépendance))
   Fusion-ou-Addition(sous-dépendance, SD)
```

Les règles d'interactions de la sur-dépendance sont renommées (fonction `Renomme`) en suivant les informations données dans la sous-dépendance (fonction `Clause-de-renommage`). Ensuite l'héritage entre la sous-dépendance et la sur-dépendance renommée est effectué.

```
Fusion-ou-Addition(sous-dépendance, sur-dépendance)
     H := Interactions(sur-dépendance)
     Pour tous I de Interactions(sous-dépendance):
         si non Compatible(I, H)
            alors H := Ajoute(I, H)
            sinon H: = Fusionne(I, H)
     rend H
```

La compatibilité entre les règles héritées et définies localement est vérifiée. Lorsqu'une règle est différente, elle est ajoutée aux règles locales ou précédemment héritées. Sinon elle est fusionnée avec les règles compatibles. Lors de la fusion, les symboles utilisés dans les différentes actions sont renommées afin d'éviter des captures de variables.

Fusion. Nous avons choisi des politiques différentes pour cette fusion suivant les opérateurs traités :

- pour l'opérateur `implies`, les messages compensatoires sont concaténés. L'ordre choisi pour la concaténation des messages compensatoires est le même que celui des méthodes auxiliaires `:after` de CLOS. C'est-à-dire, les messages compensatoires locaux sont exécutés après les messages compensatoires hérités, ceci dans l'ordre spécifié par la donnée des sur-dépendances pour l'héritage multiple (voir les règles définies aux lignes 3, 11 et 15 dans les définitions de *R1* et *R2* ci avant.).

- Pour l'opérateur `permitted-if`, la garde résultante de fusion de plusieurs interactions est la conjonction des gardes de celles-ci. Ce choix est similaire à celui fait pour les gardes des contraintes de synchronisation [FRØL 92] et les préconditions de EIFFEL. Ceci est illustré par la fusion des comportements définis lignes 4 et 14 dans les définitions de *R1* et *R2* de l'illustration.

Héritage multiple.

L'héritage multiple est la généralisation de l'héritage simple, les différentes interactions provenant des sur-dépendances sont traitées de la même manière que lors de l'héritage simple.

```
HériteMultiple(sous-dépendance, sur-dépendances)
    HM := sous-dépendance
    Pour tous R de sur-dépendances
         HM := Hérite(HM, R)
    rend HM
```

5.4.4 Limites de l'héritage proposé et extensions possibles

- Nous n'avons pas introduit la possibilité d'enlever une règle d'interaction d'une sur-dépendance dans la définition d'une sous-dépendance par analogie avec l'héritage entre classes. Une sous-dépendance spécialise une sur-dépendance donc elle met en œuvre au minimum le même comportement que la sur-dépendance.

- Nous n'avons pas proposé un renommage des sélecteurs de filtre de règles d'interaction. Cependant, bien qu'un tel mécanisme ne soit pas simple de mise en œuvre, il pourrait être utile pour décrire des dépendances ***abstraites***, c'est-à-dire non directement instanciables, décrivant des dépendances en terme de méthodes abstraites. De telles dépendances seraient ensuite *«instanciées»* en substituant ces méthodes abstraites par des méthodes réelles.

  La dépendance *exclusion-mutuelle* sur les boutons pourrait être vue comme une dépendance abstraite *«instanciée»* avec renommage en terme de méthodes de boutons.

  Un tel renommage pose cependant des problèmes en termes de signatures des méthodes et d'expressivité des actions abstraites.

- Bien que ce dernier point déborde sur des considérations d'implémentation, il faut noter que l'héritage proposé est multiple lorsque le langage hôte le permet. Cette restriction provient principalement du fait qu'une action peut faire référence à une variable d'instance d'une dépendance qui, elle, est gérée par l'héritage du langage hôte.

5.5 Dépendances et composition

« We would like the parts remain visible, while at the same time access is mediated by the owner. » [Blak 87].

Notre travail n'est pas directement connecté aux travaux sur la relation de composition [Blak 87, Duge 87, Mura 89, Woli 91, Cive 93, Magn 94, Vaut 96]. Cependant, nous abordons maintenant comment les dépendances permettent d'exprimer différemment certaines relations de composition. Nous traitons en particulier le problème de l'encapsulation de composants dans une hiérarchie de composition. Des travaux, parmi lesquels le modèle Cobras [Vaut 96], ont une approche du maintien de la cohérence entre composants strictement basée sur la structure des objets composites. Notre modèle de dépendances n'assujetit ni le maintien à la connaissance structurelle des objets, ni à la hiérarhie de composition.

5.5.1 Composition et encapsulation

Dans cette partie, nous présentons le problème de la composition d'objets par rapport à l'encapsulation. Nous étudions les solutions proposées puis nous donnons une solution à l'aide de dépendances.

Le problème.

Le traitement traditionnel de la composition d'objets consiste à utiliser des variables d'instances pour représenter les relations de composition. L'encapsulation de ces variables d'instances permet de définir des agrégations d'objets strictement internes à l'objet composite. Ainsi l'objet composite garantit sa cohérence interne en se réservant l'accès à ses composants. Cette façon de procéder rend alors les composants invisibles et inaccessibles et nécessite de redéfinir les méthodes des composants au niveau de l'objet composite comme le dit E. Blake: *« The whole protocol which a part understands [...] will have to be re-implemented as the protocol of the whole. The net result is that the part hierarchy is replaced by a single monolithic whole as far as the external world* *is concerned »* [Blak 87]. En effet, le comportement de chaque composant n'est atteint qu'au travers de l'objet composite (voir A.2).

La solution qui consiste à stocker des informations relatives aux parties au niveau de l'objet composite va à l'encontre du principe qui veut que des informations ne soient stockées qu'au niveau de composition auxquelles elles appartiennent. *« Information about the whole is not stored in the parts, information about the parts which is not modified by the whole remains with the parts »* [Blak 87].

Dans le cas contraire, si la hiérarchie de composition est visible (l'objet composite laissant des accès à ses composants), il est alors possible d'accéder et de modifier les composants librement. Ceci entraîne des problèmes de maintien de la cohérence de l'objet composite. La solution idéale veut alors que les parties restent visibles, alors que dans le même temps leurs accès soit gérés par l'objet composite.

Des solutions.

- E. Blake propose une solution basée sur la délégation des messages adressés à l'objet composite et sur la notion d'*accès censuré*: l'objet composite délègue des messages (qui peuvent être censurés[11]) aux parties et les réponses (qui peuvent être aussi censurées) proviennent des parties.

  Les accès aux composants sont basés sur l'introduction de sélecteurs de messages composés. Un sélecteur de messages composés s'écrit: <chemin>+"."+<sélecteur>. Le premier élément du chemin est nommé *préfixe*. Le chemin décrit alors un parcours de délégation de messages dans la hiérarchie de composition à la manière des chemins de composition de ThingLab [Born 86b].

  Accéder au pouce du pied de la jambe gauche de `joe` s'écrit: `joe leftLeg.foot.bigToe.`

  Si le sélecteur composé fait partie des méthodes de l'objet composite, la méthode est appliquée (ce faisant l'objet composite interdit l'accès à ses composants). Sinon le préfixe est supprimé du chemin du message qui constitue alors un nouveau message composé délégué à la variable d'instance représentée par le préfixe.

- F. Civello propose une solution basée sur la définition de propriétés sémantiques associées à la relation de composition. Parmi ces propriétés se trouvent: la *visibilité* qui indique si l'objet composite peut envoyer des messages aux composants, l'*encapsulation* qui permet de rendre la structure interne de l'objet composite invisible avec différentes nuances [Cive 93].

- M. Magnan propose pour sa part un modèle d'objets composites dans lequel chaque relation de composition peut être exclusive ou partagée, prédominante forte, faible ou non prédominante, à dépendant unique, multiple ou indépendant. Nous renvoyons le lecteur à [Magn 94] pour plus de détails.

Notre solution: un opérateur de délégation pour dépendances.

Notre solution s'apparente à la solution de E. Blake. Elle est basée sur la délégation[12] des messages adressés à l'objet composite. Cependant, cette délégation est spécifiée à l'aide d'une dépendance liant l'objet composite et ses composants. L'utilisation des dépendances permet de regrouper les informations relatives à la relation de composition dans une entité manipulable.

[11] L'objet composite en définissant les méthodes correspondant aux messages délégués peut censurer l'accès à ses composants.

[12] Nous sommes conscient des divers sens que le terme «délégation» possède, en particulier par analogie aux prototypes. Ici, il signifie plus précisément un «renvoi de message».

L'opérateur corresponds. Il permet de déclarer qu'un message reçu et non compris par un objet doit être renvoyé à un autre objet. Avec cet opérateur, l'ensemble des messages des parties n'a plus à être ré-implémenté au niveau de l'objet composite. Son utilisation est illustrée comme suit: `(method1 :object1 argn) corresponds (method2 :object2 (fct argn))`
signifie que lorsqu'un message de sélecteur `method1` non défini sur la classe de l'objet receveur du message (`:object1`), le message de sélecteur `method2` est envoyé, en utilisant ou non les arguments de l'appel, à un autre objet ou au même objet. Il faut noter que le message `corresponds` permet non seulement de déléguer les messages mais il permet d'adapter ceux-ci.

> Si un utilisateur veut préciser dans un exemple de composition que la couleur d'une voiture est celle de sa carrosserie, il écrira : `(color :whole) corresponds (color :part)`. Ainsi le message `color` envoyé à la voiture (l'ensemble) sera redirigé à la carrosserie (une des ses parties) qui elle sera capable de répondre à ce message.

Un exemple : composition et accès dans un feu tricolore. Nous reprenons ici l'exemple du feu tricolore proposé dans [Gold 83] pour illustrer l'utilisation des dépendances Smalltalk (voir en A.3). Ce même exemple est utilisé par Murata et Kusumoto dans [Mura 89] (voir en A.2) pour présenter leur solution au problème de l'encapsulation dans les hiérarchies de composition. Le problème correspond exactement à l'énoncé fait plus haut : comment accèder aux lampes d'un feu tricolore sans avoir à réimplémenter chacunes des méthodes des lampes au niveau du feu.

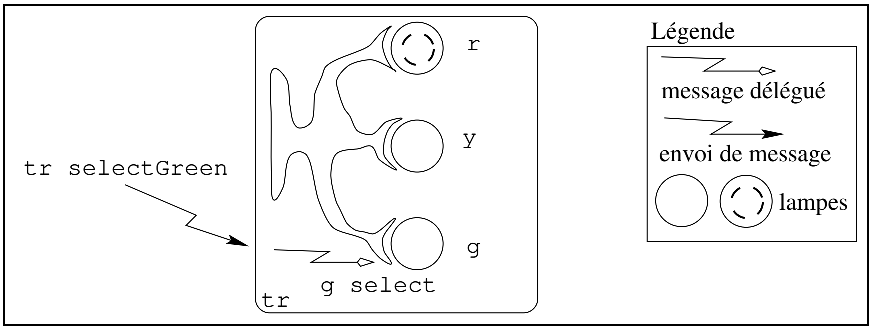


Figure 5.14: Une dépendance gérant l'accès aux composants dans une hiérarchie de composition.

Nous définissons une dépendance nommée *between-trafficl-lights*. Elle spécifie que les messages pour éteindre ou allumer les lampes, `(de)selectRed`, `(de)selectYellow` et `(de)selectGreen`, adressés au feu sont délégués aux lampes. Ce comportement est illustré par la figure 5.14.

```
(deflink between-trafficl-lights (:trafficl :red :yellow :green)
 :behavior
   (((deselectRed :trafficl)    corresponds (deselect :red))
    ((deselectYellow :trafficl) corresponds (deselect :yellow))
    ((deselectGreen :trafficl)  corresponds (deselect :green))
    ((selectRed :trafficl)      corresponds (select   :red))
    ((selectYellow :trafficl)   corresponds (select   :yellow))
    ((selectGreen :trafficl)    corresponds (select   :green))))

(define tr (make traffic-light))
(define r  (make light :color 'red)) (define g  (make light :color 'green))
(define y  (make light :color 'yellow))
(define btl (make between-trafficl-lights :trafficl tr
                                          :red r :green g :yellow y))
```

La création de l'objet composite peut impliquer la création de ses composants. La dépendance effective est alors déclarée au moment de la création (dans la méthode de création).

5.5.2 Composition et maintien de cohérence

En plus de répondre au problème de l'encapsulation, les dépendances peuvent être utilisées pour maintenir la cohérence entre les composants d'un objet composite. De la même manière que certaines dépendances structurelles dans le cadre des objets composites peuvent propager ou maintenir des propriétés entre les composites et leurs composants ou entre composants eux-mêmes [Magn 94], les dépendances peuvent assurer des contraintes comportementales entre ces objets. A la manière de Cobras [Vaut 96] dans lequel les dépendances comportementales sont décrites au niveau de la classe des objets composites, de telles dépendances peuvent être associées à la classe de l'objet composite et être instanciées automatiquement pour toutes les instances de cette classe.

> La définition de la dépendance d'exclusion peut être liée à la définition d'un objet composite `feu-tricolore`. Ainsi lors de la création d'un feu, la dépendance d'exclusion est déclarée entre ses lampes.

Exemple de composition et maintien. Pour reprendre l'exemple du feu tricolore, la définition d'une dépendance gérant à la fois les accès aux composants par délégation et une contrainte d'exclusion entre les lampes est facilement définissable. Il est possible soit de déclarer la dépendance *exclusion-mutuelle-réactive* entre les lampes du feu et de déclarer la dépendance *between-trafficl-lights*, soit de définir une nouvelle dépendance par héritage comme nous le montrons ci-après. La figure 5.15 illustre à la fois la délégation de messages et le maintien de la contrainte d'exclusion entre les lampes.

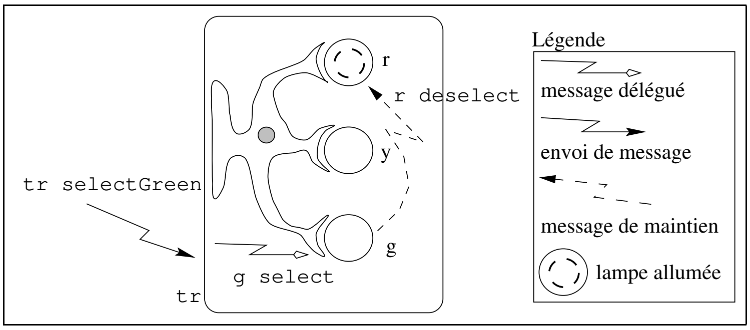


Figure 5.15: Une dépendance assurant à la fois la délégation et le maintien de cohérence dans une hiérarchie de composition.

```
(deflink  traffic  (:lights :traffic :red :yellow :green)
   :inherit '(( exclusion-mutuelle-reactive (rename :buttons as :lights))
              ( between-trafficl-lights)))

(make  traffic :lights (list r g y) :traffic tr :red r :yellow y :green g)
```

Remarque. Contrairement à Cobras [Vaut 96], dans notre modèle, la définition d'un objet composite n'est pas toujours nécessaire.

> La dépendance d'exclusion peut être associée à une colonne de radio-boutons, à un feu tricolore ou plus simplement à un groupe d'objets ne nécessitant pas la définition d'un objet les englobant structurellement.

Ceci offre la possibilité de pouvoir abstraire une dépendance des objets y participant afin qu'elle soit réutilisée.

> La dépendance d'exclusion peut être réutilisée pour lier des objets composites ou non. Ce comportement n'est pas lier à la structure des objets.

Contrairement aux travaux de M. MAGNAN [MAGN 94], nous n'abordons qu'un aspect des problèmes de composition dans les langages objets. Cependant, nous pensons que notre modèle est assez souple pour offrir des solutions à d'autres problèmes comme, par exemple, la propagation de valeurs dans les hiérarchies de composition. D'autre part, nous avons étudié les solutions proposées par le système COBRAS [VAUT 96] en cours d'ébauche par l'équipe de C. OUSSALAH. Le but de COBRAS est d'introduire un maintien de cohérence entre les composants d'un objet composite. Après étude avec les concepteurs de COBRAS nous pensons qu'une implémentation du moteur de maintien de cohérence de COBRAS est réalisable avec notre modèle.

5.6 Conclusion

Dans ce chapitre, nous avons présenté

- la place des dépendances par rapport aux classes : les dépendances sont distinctes et autonomes des classes des objets mis en dépendances.
- comment, à la manière du schéma classe/instances, les dépendances sont définies puis déclarées entre les objets qu'elles contrôlent ;
- illustré comment le comportement dynamique est utilisé pour garantir la cohérence de la dépendance.
- les avantages de la réification des dépendances : la dépendance est alors l'endroit idéal pour définir des données liées à la communication entre objets (conversion, adaptation d'arguments de messages) ou des données propres à la dépendance.
- les limites de notre modèle lors de la définition de dépendances entre classes. Ayant choisi de ne pas modifier le mécanisme traditionnel d'héritage entre classes, les dépendances entre classes ne sont pas héritées automatiquement.
- un mécanisme de définition incrémental adapté aux dépendances.

Nous avons conclu en montrant que les dépendances s'intégrent aux modèle à classes en particulier dans les hiérarchies de composition. Les dépendances permettent de gérer des problèmes d'accès et de maintien de cohérence dans les objets composites.

6

Contrôleurs et globalité

Notre modèle en se concentrant sur la notion de dépendances comportementales, exacerbe les problèmes souvent enfouis et gérés de manière *ad hoc* par les langages objets. En effet, les solutions traditionnelles utilisées pour représenter les dépendances (voir au chapitre 1 et en annexe A) ou les nombreux travaux introduisant des dépendances (voir au chapitre 2) ne gèrent que rarement les cycles induits par les dépendances. De plus, l'expression locale de dépendances pose le problème de la difficulté de synthétiser un comportement global déterministe.

Dans ce chapitre, nous motivons l'introduction d'entités responsables du contrôle des messages : des *contrôleurs* [Mins 87, Mins 89]. Nous montrons comment ces contrôleurs offrent une plus grande abstraction au programmeur et proposent des solutions aux problèmes des cycles. Le but de notre modèle étant de permettre une description d'un grand nombre de dépendances, nous montrons comment les contrôleurs apportent une flexibilité dans la modélisation de la gestion globale des dépendances. Cependant, la description des contrôleurs restent dans ce chapitre au niveau de la modèlisation, l'implémentation de telles entités est traitée au chapitre 9.

6.1 Contrôleurs

Dans les systèmes pour lesquels aucun contrôle ou modification du comportement du maintien de la cohérence des dépendances n'est prévu, la sémantique du maintien est implicite, unique et souvent immuable. Or nous avons spécifié au chapitre 2 qu'un contrôle du mécanisme même de maintien de la cohérence des dépendances doit être offert au programmeur. Un tel contrôle implique l'introduction dans le modèle de nouvelles entités permettant de s'abstraire de la localité des dépendances. Nous définissons donc des *contrôleurs*: des entités responsables du contrôle du comportement des instances et de la gestion des dépendances. Les contrôleurs représentent des abstractions du mécanisme de gestion des dépendances, elles offrent ainsi au programmeur la possibilité d'adapter le comportement du modèle ; en particulier en permettant de spécifier le comportement global de la gestion des dépendances.

Avant de situer ces nouvelles entités par rapport aux dépendances, classes et objets, nous précisons leurs rôles. Nous présentons le concept du contrôleur.

6.1.1 Rôle et statut des contrôleurs

Rôle.

Un contrôleur est une entité responsable de la cohérence des dépendances qui existent entre les objets sous sa responsabilité (voir 6.2). Pour cela, il contrôle les messages envoyés à de tels objets et utilise le comportement dynamique des dépendances. Ce contrôle est *«implicite»* du point de vue de la communication entre objets, c'est-à-dire que les messages continuent à être adressés aux objets et non à leur contrôleur.

Dans le cas d'une dépendance effective *mémorisée-par* entre `p1` et `m1`, lorsqu'un message de sélecteur `pop` est envoyé à l'objet `p1`, son contrôleur contrôle ce message : sous la condition que la garde définie par la dépendance soit vraie, il exécute la méthode `pop` et déclenche le message compensatoire : un message `store` est envoyé à l'objet `m1` (voir la figure 6.1).

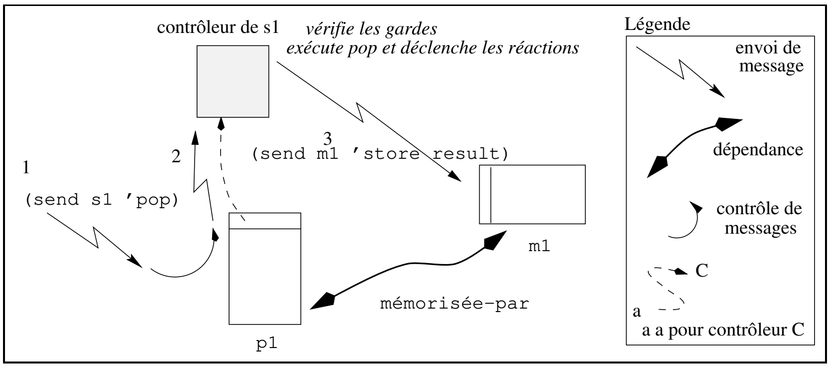


Figure 6.1: Maintien de la cohérence des dépendance à l'aide de contrôleurs.

Avant d'aborder les conséquences de cette modélisation du maintien de la cohérence, nous justifions le statut des contrôleurs comme des objets de plein droit.

Entités à part entière.

Les contrôleurs sont des entités distinctes des classes des objets ou des dépendances. Plusieurs raisons justifient ce choix et nous poussent vers leur réification :

Un contrôle distinct des dépendances. La principale raison réside dans notre volonté d'offrir un contrôle global du mécanisme de maintien de la cohérence des dépendances. Celui-ci doit permettre de définir différents comportements, comme une propagation en largeur, l'élimination de messages compensatoires redondants, le debuggage ou une gestion de la distributivité. De tels comportements sont dissociés de la dépendance. Celle-ci définit de manière locale des actions par rapport à la réception d'un message, elle ne définit pas de traitement global. Ainsi un ensemble de dépendances peut être maintenue de différentes manières (filtrage des messages compensatoires...), ce qui implique qu'un tel contrôle ne soit pas lié aux dépendances.

A la manière de la dépendance *représentée-par*, des dépendances sont utilisées pour assurer la cohérence entre un objet de l'application et sa représentation graphique. Lors d'applications distribuées [Kars 93] ou de resynchronisation entre les objets et leurs interfaces après déconnexion temporaire [Dery 96b], il est nécessaire d'éliminer les messages redondants : par exemple, il est inutile de changer la couleur d'un objet deux fois de suite, seul le dernier changement est significatif. Ce traitement alors n'est absolument pas lié au comportement dynamique de la dépendance. Celle-ci spécifie que si un objet reçoit le message `x` alors sa représentation doit changer de couleur et elle ne prend pas en compte un possible traitement global.

Une approche objet du contrôle. Le contrôle du mécanisme de maintien définit des données et des comportements qui lui sont propres. La réification du contrôle permet de le particulariser facilement.

Un contrôle distinct des classes. Les dépendances lient des instances particulières. Nous souhaitons donc un contrôle qui se situe au niveau des instances. Or, définir le mécanisme de maintien dans les classes des objets liés signifierait que toutes les instances d'une même classe soient obligées d'avoir ce même contrôle [Ferb 89]. Donc les classes ne peuvent pas représenter les contrôleurs.

Objets, dépendances et contrôleurs.

Contrairement aux travaux de [Maes 87a, Ferb 89, Chib 93a] dans lesquels un méta-objet est associé à un objet et un seul, un contrôleur peut être associé à un groupe d'objets. Par contre, un objet ne possède qu'un seul contrôleur. Un contrôleur peut contrôler plusieurs dépendances. La figure 6.2 illustre cette situation : deux contrôleurs contrôlent de manières différentes deux groupes d'objets et les dépendances existant entre ces objets.

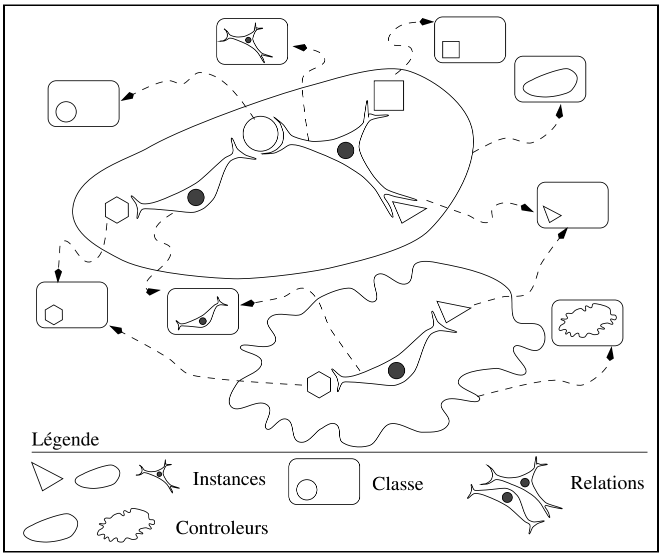


Figure 6.2: Contrôleurs, relations, classes et instances

6.1.2 Le contrôle

Lors de l'envoi d'un message à un objet, deux situations peuvent se produire : soit le message est *non-perturbant* par rapport à cet objet (la cohérence d'aucune dépendance n'est remise en cause). Dans ce cas, le message est normalement résolu, la méthode est cherchée puis appliquée normalement. Soit le message est perturbant, dans ce cas le message est contrôlé.

> L'envoi d'un message `mes1` à un objet `o1` provoque l'envoi du message `contrôle` à `Co1` le contrôleur de `o1` comme illustré par la figure 6.3.

Le contrôle des messages est défini par la méthode `contrôle` associée au contrôleur de l'objet receveur : un contrôleur doit savoir s'il existe des dépendances sur l'objet receveur du message et si la cohérence de telles dépendances risque d'être altérée par ce message. Ensuite, il doit invoquer les actions associées suivant la sémantique décrite par les opérateurs. La méthode `contrôle` combine ces différents aspects qui seront présentés en détail au chapitre 9. Nous introduisons une pseudo définition de cette méthode afin de pouvoir présenter certaines modifications par la suite.

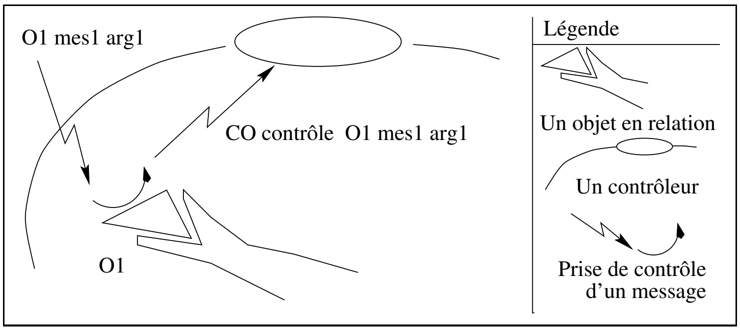


Figure 6.3: La réception d'un message par un objet.

La méthode des `contrôle` du contrôleur ne gérant que les messages compensatoires est définie comme suit :

```
1 contrôle contr m-contrôlée receveur arguments
2   (appliquer m-contrôlée receveur arguments)
3   si  (existe (dépendances contr receveur m-contrôlée implies))
4   alors (déclencher-mesg-compensatoires
                (dépendances contr receveur m-contrôlée implies))
5   rendre-résultat-application
```

La méthode est appliquée (ligne 2) puis si des messages compensatoires sont nécessaires (lignes 3), ils sont déclenchés (ligne 4), le résultat de l'application est ensuite rendu (ligne 5). Il faut noter que ce contrôleur propage les réactions de manière implicite en profondeur.

6.1.3 Contrôleurs et opérateurs

Nous précisons maintenant quelles exigences le contrôle des messages doit prendre en compte. Le contrôle doit être :

Spécifique. Un contrôle est spécifique à chaque opérateur.

> L'opérateur `permitted-if` impose de subordonner l'exécution de la méthode invoquée à la vérification des gardes. L'opérateur `implies` impose de déclencher des messages compensatoires.

Combinaison. Un même contrôleur doit pouvoir combiner différents opérateurs. Lorsque des dépendances définissent des gardes et des messages compensatoires, le contrôleur doit combiner ces deux aspects des dépendances ; c'est-à-dire subordonner le déclenchement des messages compensatoires à la vérification des gardes.

Optimisation et nouveaux opérateurs. Il est inutile pour un contrôleur de gérer des opérateurs si aucune des dépendances qu'il maintient n'en utilise. D'autre part, de nouveaux opérateurs doivent pouvoir être définis.

Ces exigences nous amènent donc à privilégier une solution dans laquelle le contrôle est à la fois minimal et adaptable.

Une association contrôleurs / opérateurs. Notre solution est basée sur une association entre contrôleurs et opérateurs. La sémantique d'un opérateur est définie par un ou plusieurs contrôleurs associés à l'opérateur. L'introduction d'un nouvel opérateur est liée à la définition d'un contrôleur.

> Pour l'opérateur `implies`, le contrôleur spécifie le contrôle des messages que nous avons montré dans l'exemple précédent.

Cependant, cette liaison ne se limite pas à une simple association entre un opérateur et un contrôleur. Un opérateur est lié à plusieurs contrôleurs susceptibles d'assurer son interprétation. En effet, le contrôleur d'un objet doit gérer tous les opérateurs de toutes les dépendances auxquelles cet objet participe. Ainsi chaque dépendance, au travers de ces opérateurs, impose la nature des contrôleurs associés aux objets qu'elles lient.

La figure 6.4 illustre l'influence des opérateurs sur la nature des contrôleurs des objets. Lorsqu'une dépendance n'est définie qu'à l'aide d'un seul opérateur (cas 1 et 2), le contrôleur des objets y participant doit gérer exactement cet opérateur (cas 1 et 2). Lorsqu'une dépendance utilise différents opérateurs, le contrôleur doit gérer ces différents opérateurs (cas 3).

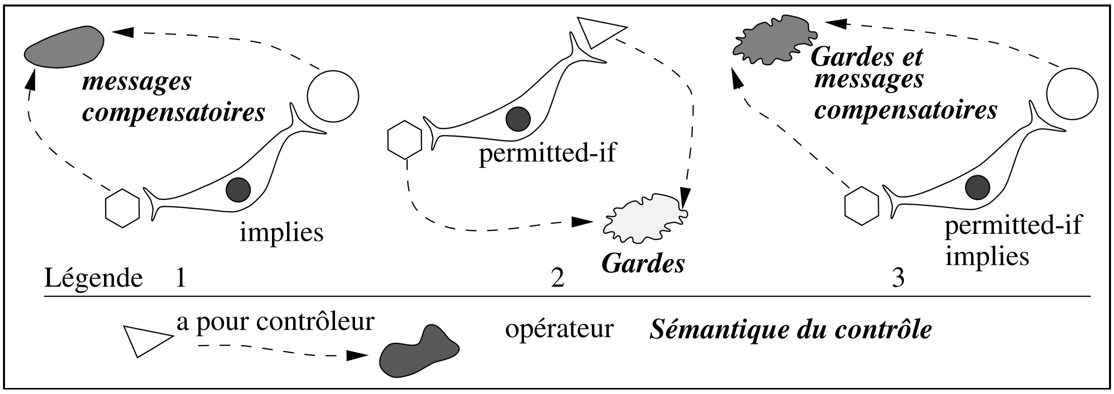


Figure 6.4: Des contrôleurs en accord avec les dépendances.

Choix et combinaison de contrôleurs. Pour assurer un contrôle à la fois minimal et adaptable, notre solution consiste à associer le meilleur contrôleur par rapport aux exigences des dépendances. Nous choisissons le type de contrôle en fonction des opérateurs utilisés pour décrire le comportement dynamique de la dépendance grâce à un *graphe de compatibilité* entre contrôleurs dont les nœuds sont des classes de contrôleur et les arêtes étiquetées par des opérateurs (voir la figure 6.5). Ce choix est effectué automatiquement par le système lors de la déclaration d'une dépendance. Ainsi, les contrôleurs sont changés dynamiquement.

Si un objet possède un contrôleur gérant l'opérateur `implies` et qu'une nouvelle dépendance utilisant l'opérateur `permitted-if` est déclarée sur cet objet, cet objet aura un nouvel contrôleur gérant ces deux opérateurs.

6.1.4 Globalité et contrôleurs

Le comportement des contrôleurs que nous avons présenté ne représente que des comportements de maintien purement locaux. Cependant, un contrôleur permet aussi de définir un comportement global du maintien des dépendances. Alors que la propagation des messages compensatoires est initialement en profondeur, ce comportement peut être modifié pour définir des traitements sur les messages compensatoires comme une propagation en largeur, un filtrage ou une gestion des cycles.

La solution que nous avons proposé utilise les opérateurs pour déterminer le bon contrôleur. Or comme nous l'avons déjà dit en 6.1.1, les comportements *globaux* ne sont pas directement liés à la définition même des dépendances. Les contrôleurs permettant la mise en place de tels comportements doivent alors tenir compte de la sémantique des opérateurs mais ne peuvent pas être déduits simplement à partir de comportements de base des opérateurs.

Face à ce problème, notre solution considère qu'il est de la responsabilité du programmeur d'associer des contrôleurs cohérents avec le comportement global souhaité. De plus, lors d'un

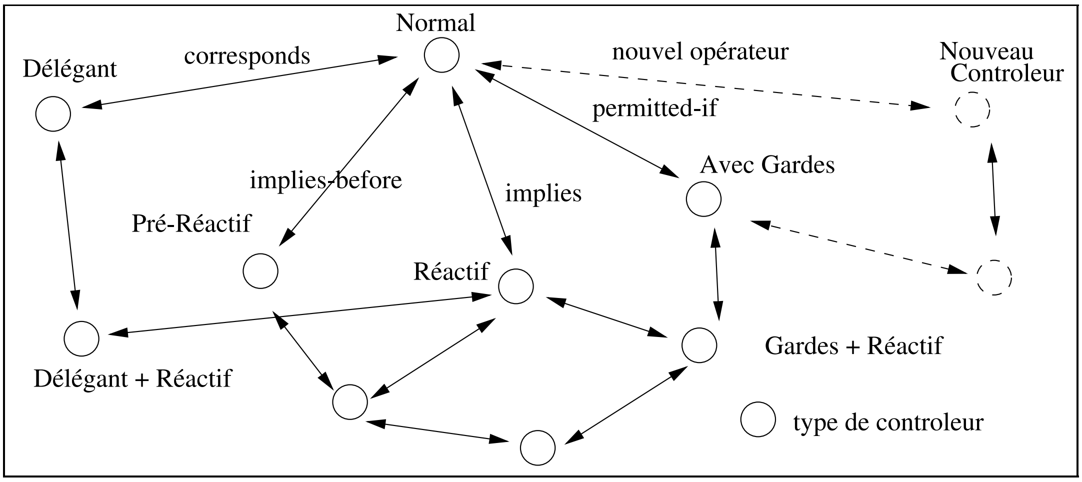


Figure 6.5: Graphe de compatibilité et d'évolution des contrôleurs.

maintien global de la cohérence, il est préférable d'associer un contrôleur à un groupe d'objets. En effet, il est plus difficile de réaliser un maintien global avec plusieurs contrôleurs.

Cette solution s'inspire de la solution proposée par CLOS pour garantir les problèmes de compatibilité de méta-classes [Grau 89, Danf 94b]. Ainsi en CLOS, initialement toutes les sous-classes d'une classe doivent avoir la même méta-classe que celle de la classe. Cette contrainte est levée en redéfinissant la méthode `validate-superclass` (p.240 de [Kicz 91] et p.84 de [Pae 93]). Ce faisant le programmeur prend la responsabilité de définir de nouvelles méta-classes compatibles avec les méta-classes déjà existantes.

La suite de ce chapitre illustre comment les contrôleurs introduisent une vision globale de la propagation. Nous montrons diverses solutions au problème des cycles puis comment un algorithme de planification est introduit pour le maintien de la cohérence.

6.2 Variations sur les cycles

Lors du maintien de la cohérence d'une dépendance, les messages compensatoires peuvent impliquer d'autres messages compensatoires et provoquer des cycles. Il y a un cycle quand un objet reçoit le même message durant la même[1] phase de propagation.

Nous traitons exclusivement les cycles liés à l'existence de dépendances *explicites* entre objets. En effet, nous ne traitons pas des cycles dus aux relations *implicites* entre méthodes car d'après le modèle objet que nous avons choisi (voir en 3.1), nous considérons que nous n'avons pas accès au code des objets[2].

Nous montrons maintenant comment les contrôleurs définissent une gestion des cycles. Notre solution précise les limites d'une détection statique des cycles et met en avant une détection dynamique.

6.2.1 Une relation, des variations

Les dépendances étant n-aires, il est possible d'exprimer une même situation de différentes façons. Cette diversité d'expression devant être prise en compte, nous proposons différentes stratégies afin d'assurer la terminaison des programmes.

[1] Certains cycles peuvent être utilisés pour raffiner un résultat, notre proposition les traite également.

[2] Même si nous avions accès au code, ce poserait le problème de la détection de cycles dans un langage à objet ce qui reste à notre sens du ressort de la preuve de programmes.

Exemple. En CENTAUR [BORR 87], une même portion de programme peut être visualisée dans plusieurs fenêtres sous différentes formes. Ces différentes représentations doivent être cohérentes entre elles. Ainsi la sélection d'une section dans une des représentations peut avoir des répercussions dans les autres représentations. Limitons nous ici à la cohérence entre deux représentations d'un même programme et montrons comment une dépendance peut maintenir la cohérence entre les deux arbres représentant ce même programme (cet exemple est inspiré de [DISS 96]). Nous supposons données les fonctions de transformations d'une représentation dans l'autre.

```
(deflink  traduit-1-vers-2 (:arbre1 :arbre2)
 :behavior
  (((selectionne :arbre1 part)
    implies (selectionne :arbre2 (de1vers2 part)))))

(deflink  traduit-2-vers-1 (:arbre1 :arbre2)
 :behavior
  (((selectionne :arbre2 part)
    implies (selectionne :arbre2 (de2vers1 part)))))

(make  traduit-1-vers-2:arbre1 a1:arbre2 a2)
(make  traduit-2-vers-1:arbre1 a2:arbre2 a1)
```

Deux définitions sont possibles selon la finesse de séparation des dépendances que l'on veut exprimer. Ainsi il peut être nécessaire d'avoir deux dépendances *traduit-1-vers-2* et *traduit-2-vers-1*, ou bien au contraire, on peut souhaiter exprimer de manière forte et structurelle la symétrie de la dépendance comme c'est le cas dans la dépendance *traduit*[3].

Nous distinguons deux sortes de cycles : des cycles *«inter»* dépendances dus à la présence de plusieurs dépendances, comme le montre les définitions et déclarations ci-dessus et des cycles *«intra»* définition comme le montre la dépendance *traduit*.

```
(deflink  traduit  (:arbre1 :arbre2)
 :behavior
  (((selectionne :arbre1 part)
    implies (selectionne :arbre2 (de1vers2 part)))
   ((selectionne :arbre2 part)
    implies (selectionne :arbre2 (de2vers1 part)))))

(make  traduit:arbre1 a2:arbre2 a1)
```

La lecture de ces trois dépendances amène deux remarques. Tout d'abord, le contrôle des cycles est nécessaire. En second lieu, la définition de ces dépendances décrit bien un besoin clair de symétrie : quelque soit l'arbre sélectionné, cette sélection doit apparaître dans l'autre représentation.

6.2.2 Limites d'une détection statique

La détection statique des cycles peut apparaître comme possédant de bonnes propriétés dans certains langages comme ESTEREL [DISS 96]. Dans notre cas, nous pensons qu'elle ne doit servir qu'à avertir le programmeur d'éventuels cycles au niveau des dépendances et non à proscrire la définition de cycles car l'écriture de telles dépendances est souvent nécessaires [HILL 94]. Par contre, le système doit éviter que dynamiquement de tels cas ne fassent boucler le programme.

Le système doit proposer différents moyens de contrôler le comportement dynamique d'une dépendance : ne pas déclencher plusieurs fois la même dépendance durant une propagation, tester, lorsque cela est possible, si la dépendance est cohérente avant de déclencher les messages compensatoires...

[3] Cette dépendance peut être aussi définie par héritage des deux dépendances précédentes.

6.2.3 Propagation si nécessaire

Une première solution pour gérer dynamiquement les cycles est de ne propager les messages compensatoires que lorsque les dépendances ne sont pas cohérentes. Il est nécessaire que la définition de la dépendance inclut la définition d'un prédicat.

Si l'on suppose que l'on sait écrire le prédicat `arbres-cohérent?`, la propagation ne boucle pas comme le montre la figure 6.6. Cette approche s'applique aux différentes dépendances *traduit* ou *traduit-1-vers-2* et *traduit-2-vers-1*. Ainsi lorsqu'un message de sélecteur `selectionne` envoyé à un des deux arbres (1), la méthode est appliquée (2), ensuite, avant de réagir à ce message du fait de la présence d'une dépendance, la cohérence de la dépendance est testée (3). Lorsque la cohérence n'est pas vérifiée, la propagation est déclenchée (4) ce qui conduit à appliquer la méthode `selectionne` sur l'arbre `a2`. Ce message est de nouveau déclenchant pour la dépendance mais avant de déclencher une réaction, la cohérence de la dépendance est testée et la propagation s'arrête.

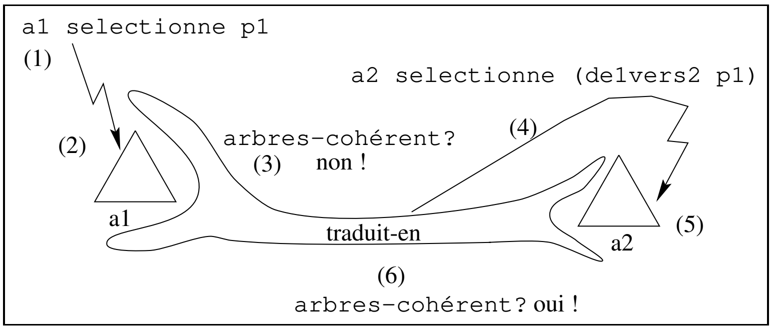


Figure 6.6: Détection dynamique à l'aide d'un prédicat.

La mise en œuvre d'un tel mécanisme reste assez simple. Elle consiste à modifier le déclenchement des messages compensatoires afin d'intégrer le test du prédicat. Au niveau de la définition de la dépendance, la définition d'un prédicat est nécessaire. D'autre part, la donnée du prédicat permet de s'assurer que lors de la déclaration de la dépendance les participants sont dans un état cohérent.

Limites. Cette approche possède les limites suivantes :

- La donnée d'un prédicat permettant de connaître l'état de cohérence d'une dépendance n'est pas toujours possible.

  Comment définir un prédicat pour les dépendances *mémorisée-par* ou *représentée-par*?

- Le test d'un prédicat peut s'avérer coûteux.

- La définition d'un tel prédicat est souvent redondante par rapport à la définition du comportement dynamique des dépendances. Bien que ThingLabII [Free 89] puisse déduire les réactions à partir d'une équation mathématique linéaire, une telle déduction n'est pas possible dans le cas général.

6.2.4 Détection basée sur le marquage des dépendances

Une autre solution consiste à «marquer» la dépendance lors de son déclenchement et à tester qu'elle n'est pas déjà marquée avant de déclencher les messages compensatoires. Les exemples proposés sont basés sur un «marquage» booléen[4] ; c'est-à-dire qu'une dépendance peut être ou non en cours

[4] Un marquage booléen reste avant tout la forme la plus simple afin de s'assurer que l'on ne passe pas deux fois par la même dépendance. Cependant, il est facile de définir des marqueurs plus élaborés comme par exemple, le sélecteur ou le message déclenchant la phase de réaction.

de propagation. Nous montrons deux approches : contrairement à la seconde approche, la première est limitée et n'utilise pas les contrôleurs.

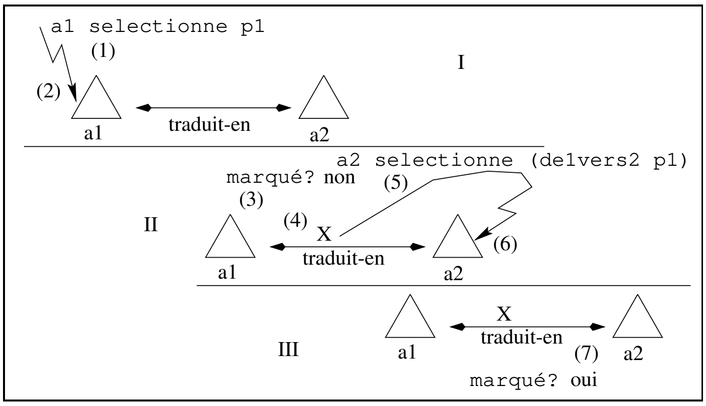


Figure 6.7: Détection de cycles basée sur la non-réentrance dans le cas de cycles intra définition.

Absence de globalité. Une première solution consiste à redéfinir le déclenchement des messages compensatoires afin qu'ils ne soient émis que lorsque la dépendance n'a pas déjà été déclenchée. Pour cela, lorsque la dépendance n'est pas marquée, il suffit de la marquer, de déclencher les messages compensatoires et de la démarquer.

Cette première solution fonctionne très bien pour les cycles *intra* définition comme le montre la figure 6.7. Un message est envoyé à `a1` (1), la méthode invoquée est appliquée (2), la phase de déclenchement des messages compensatoires teste si la dépendance n'a pas déjà été utilisée (3). Si c'est le cas, elle est marquée (4), puis les messages compensatoires sont émis (5) et appliqués (6). De nouveau, la phase de déclenchement teste si l'on se trouve dans une phase de déclenchement. Comme c'est le cas la propagation s'arrête (7).

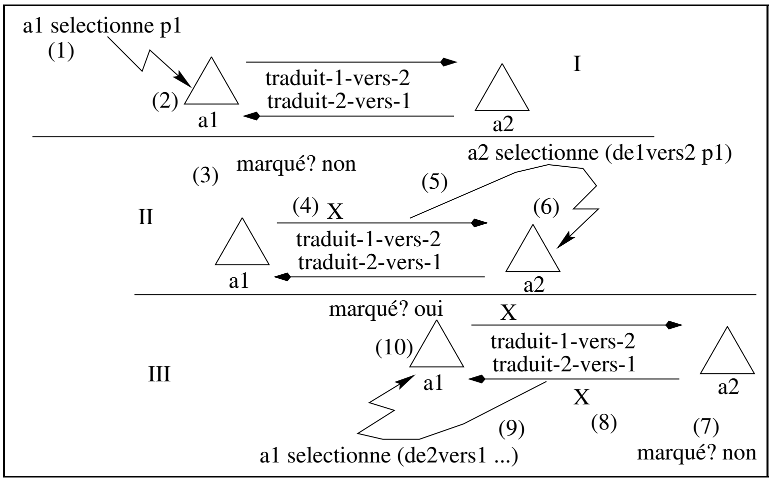


Figure 6.8: Limites d'une détection non-réentrante dans le cas de cycles entre dépendances.

Par contre, lors de cycles provoqués par l'existence de plusieurs dépendances, cette solution assure correctement la détection de cycles mais implique une répétition du message déclenchant comme le montre la figure 6.8. Le déroulement est le même jusqu'au point (7). La phase de déclenchement des messages compensatoires teste si la dépendance est marquée (7). Or ici la dépendance pour laquelle le message `a2 selectionne` est déclenchant est la dépendance effective *traduit-2-vers-1* qui elle n'est pas marquée. Donc la phase de déclenchement continue : la dépendance est marquée (8), les messages sont émis (9) et appliqués à l'objet `a1` qui a finalement reçu deux fois le même message. Cette solution ne prend pas en compte l'aspect global de la gestion des cycles.

Une solution globale. Cette solution est basée sur la même idée que la précédente mais elle prend en compte l'ensemble des dépendances. De manière similaire, le déclenchement des messages compensatoires doit être enrichi afin de marquer la dépendance, d'émettre les messages et de démarquer la dépendance. De plus, il est nécessaire de modifier le contrôle des messages afin de subordonner l'application des méthodes et le déclenchement des messages au fait que les dépendances ne sont pas déjà en phase de déclenchement.

Ce nouveau comportement du contrôle des messages peut être décrit de la manière suivante :

```
Contrôle-des-messages contr m-contrôlée receveur arguments
 si (réaction-possible? (dépendances contr receveur m-contrôlée implies))
 alors (appliquer m-contrôlée receveur arguments)
       (déclencher-msg-compensatoires
                  (dépendances contr receveur m-contrôlée implies))
       rendre-résultat-application
 sinon rien
```

La figure 6.9 illustre cette nouvelle gestion des cycles avec le même exemple. Ainsi le message `selectionne ...` est envoyé à l'objet `a1` (1). Avant d'appliquer la méthode associée au sélecteur `selectionne`, le contrôleur teste si les réactions sont possibles (2) et dans l'affirmative applique la méthode (3) et déclenche la phase de déclenchement des messages compensatoires : marquage de la dépendance (4), émission de la réaction (5). Cet envoi de message est à nouveau contrôlé (6), la dépendance n'étant pas marquée, la méthode est appliquée (7), le déclenchement s'effectue : marquage de la dépendance (8) et déclenchement des messages compensatoires (9). Cette dernière émission de message sur l'objet `a1` est de nouveau contrôlée mais cette fois-ci la méthode n'est pas invoquée (10).

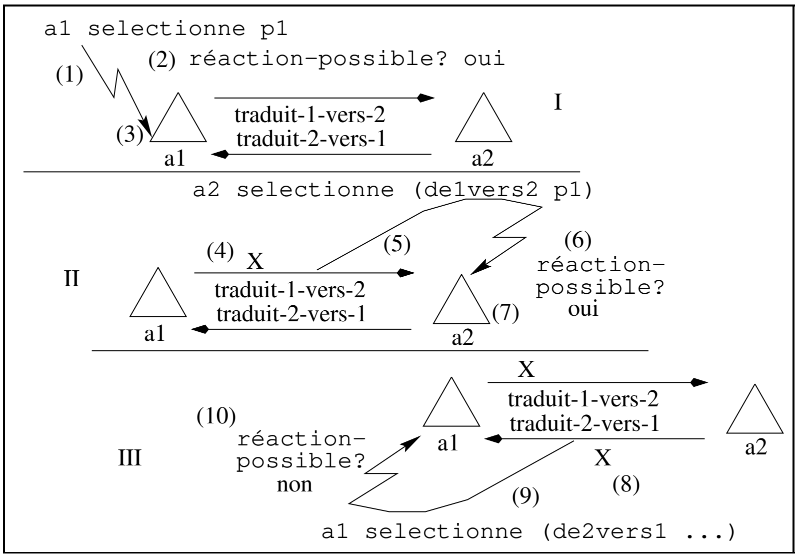


Figure 6.9: Détection globale de cycles.

6.3 Planification

Lorsque plusieurs dépendances contrôlent un même objet, il peut arriver qu'il y ait une redondance des réactions ; c'est-à-dire qu'un même objet reçoive plusieurs messages modifiant les mêmes aspects. Par exemple, les coordonnées d'un point peuvent être modifiées plusieurs successivement durant une même phase de propagation. Les systèmes à base de contraintes proposent des algorithmes de planification afin d'ordonnancer et d'optimiser cette propagation. La planification même si elle ne résout pas les problèmes de non-déterminisme [Hill 93a], prend en compte la globalité du graphe et évite ainsi des réactions redondantes.

Nous illustrons donc les faiblesses du comportement initial des dépendances puis nous montrons comment notre modèle permet la mise en place d'une planification. Notre propos n'est pas de proposer de nouvelles techniques de planification mais de montrer que notre modèle permet d'utiliser, lorsque les conditions sont réunies (méthodes à comportement simple et connu, vision d'un objet comme une variable, arithmétique linéaire), les algorithmes utilisés par les systèmes de contraintes. L'extension proposée ici se veut simple et n'a pas la prétention d'intégrer les dernières techniques des systèmes de contraintes. A ce sujet G. Trombettoni propose dans [Trom 97] un panorama des différents algorithmes de maintien de cohérence par propagation locale.

6.3.1 Un exemple : des contraintes géométriques entre points

La classe `Point` définit les méthodes : `x` pour accéder à l'abscisse d'un point, `x:` pour y affecter une nouvelle valeur. Nous définissons deux dépendances : la dépendance `au-milieu-de` et la dépendance `à-côté-de` qui spécifie qu'un point est toujours à 60 unités d'un autre point.

```
(deflink au-milieu-de (:extrem1 :extrem2 :pmilieu)
:behavior
 (((x: :extrem1 val) implies (x: :pmilieu (/ (+ val (x :extrem2)) 2)))
  ((x: :extrem2 val) implies (x: :pmilieu (/ (+ val (x :extrem1)) 2)))
  ((x: :pmilieu val) implies (x: :extrem1 (- (* 2 val) (x :extrem2))))))

(deflink à-côté-de (:un :deux)
 :behavior
 (((x: :un val) implies (x: :deux (+ 60 val)))
  ((x: :deux val) implies (x: :un (- val 60)))))
```

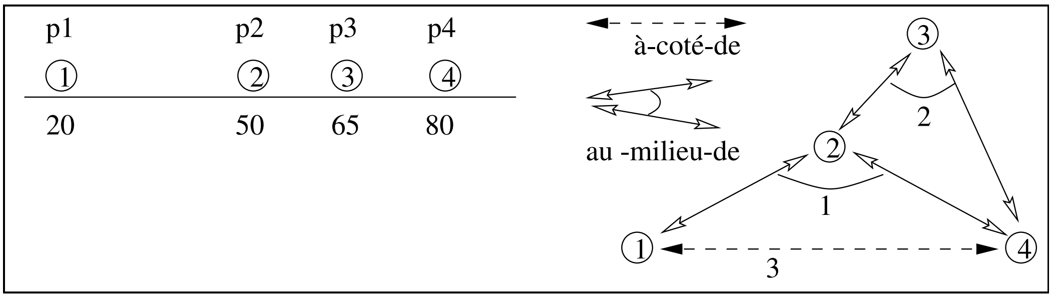


Figure 6.10: Trois dépendances.

Puis nous créons quatre points[5] (`p1`, `p2`, `p3` et `p4`) et déclarons trois dépendances effectives entre ces points comme le montre la figure 6.10.

```
(define p1 (make point :x 20))      (define p2 (make point :x 50))
(define p3 (make point :x 65))      (define p4 (make point :x 80))
```

L'ordre de déclaration des dépendances effectives est important dans notre exemple car il met clairement en avant la nécessité de planifier la propagation. Nous déclarons d'abord, deux dépendances de type *au-milieu-de* puis une dépendance *à-côté-de* :

[5]Nous avons choisi de créer des points ayant des valeurs satisfaisant les dépendances, pour ne pas compliquer l'exemple.

```
(define aumil1 (make au-milieu-de:extrem1 p1:pmilieu p2:extrem2 p4))
(define aumil2 (make au-milieu-de:extrem1 p2:pmilieu p3:extrem3 p4))
(define acote3 (make à-côté-de:un p1:deux p4))
```

Redondance de messages. Si le point `p1` change d'abscisse suite à la réception du message `(p1 x: 0)`, les messages compensatoires sont envoyés en déclenchant les dépendances suivant l'ordre de déclaration. Nous obtenons la situation suivante illustrée par la figure 6.11:

1. `aumil1` implique d'affecter l'abscisse du point `p2` à la valeur `40` ((80 - 0) / 2) ligne 3, il y a donc envoi du message `(p2 x: 40)`. `aumil2` implique, suite à ce message, d'affecter l'abscisse du point `p3` à la valeur `60` (40 + (80 - 40) / 2) ligne 3.

2. Suite au changement de valeur de l'abscisse de `p1`, la dépendance *à-côté-de* implique d'affecter la valeur de l'abscisse de `p4` à `60` (0 + 60) (ligne 9). Cette modification nécessite de changer à nouveau la valeur de l'abscisse du point `p2` et donc par propagation du point `p3` (ligne 4).

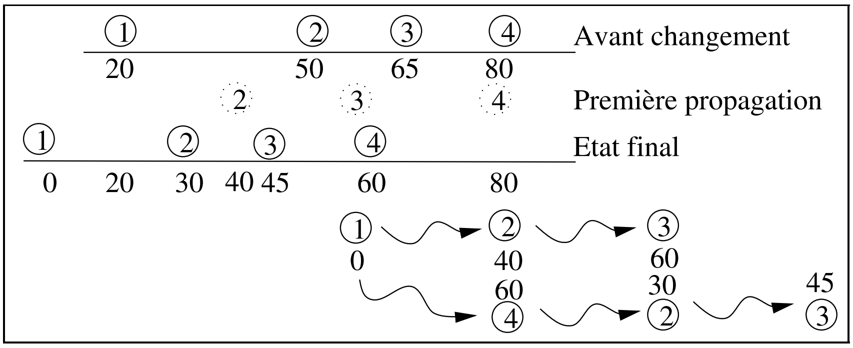


Figure 6.11: Flot de propagation avec redondance : les points `p2` et `p3` changent d'abscisse deux fois, bien qu'une fois puisse suffir.

La planification.

La redondance de messages est due au fait que les changements de valeur ne sont pas correctement ordonnés. Comme les points `p2` et `p4` dépendent du point `p4` par les dépendances effectives `au-mil1` et `au-mil2`, les changements d'abscisse de ces points doivent être effectués seulement après que celui-ci soit modifié.

Les algorithmes de planification proposent des solutions à de telles situations. Ils sont basés sur la construction de plans [Born 86b, Born 87, Born 92, Hill 93a, Sann 93, Ouss 96]. R. Hill définit un plan comme une liste ordonnée de contraintes telle que : pour chaque contrainte, C de la liste, toutes les contraintes pouvant modifier des variables *sources*[6] de C précédent C dans la liste. Ces algorithmes peuvent échouer dans la construction des plans du fait de cycles ou de dépendances mutuelles. Avec la qualification des contraintes, les auteurs de ThingLabII ont introduit un déterminisme lié à l'algorithme de planification.

Adaptations.

L'utilisation de nos dépendances pour une planification nécessite de modifier légèrement la définition du comportement dynamique.

[6] Ce sont les variables à partir desquelles sont calculées les valeurs des variables cibles. Soit k = f(x,y,z), x, y et z sont des variables sources et k une variable cible.

En effet, notre modèle ne permet pas initialement de spécifier **deux** règles d'interactions ayant pour un même sélecteur une même variable active. Aucune disjonction entre règles de même message déclenchant n'est gérée. En corollaire, l'ordre de définition des règles d'interactions n'a pas d'importance.

L'écriture de la dépendance *au-milieu-de* définie précédemment tient compte de cette limitation. Ainsi nous avions dû choisir une seule règle pour le changement d'abscisse de chaque point. Or l'algorithme de planification a besoin de connaître toutes les possibilités pour construire un plan. L'extrait ci-dessous présente la définition d'une contrainte similaire à la dépendance *au-milieu-de* en ThingLab. La ligne 4 définit clairement que si les points `midpoint` **ou** `point2` changent alors la valeur de `point1` doit être recalculée.

```
1   Constraints
2      midPoint = (line point1 + line point2) / 2
3      midPoint ← (line point1 + line point2) / 2
4      line point1 ← midPoint * 2 - line point2
5      line point2 ← midPoint * 2 - line point1
```

Dans la définition suivante (équivalente à celle de ThingLab à l'aide d'une *dépendance adaptée*), les règles d'interactions aux lignes 4 et 9, 5 et 7 et 6 et 8 illustrent bien l'introduction d'un choix lors de la propagation. Ainsi quand un point, que nous supposons associé à la variable active `extrem1`, reçoit un message `x:` deux messages compensatoires peuvent rétablir la cohérence de la dépendance. Cette disjonction de règles d'interactions est nécessaire et gérée par l'algorithme de planification.

```
1 (deflink au-milieu-de (:extrem1 :extrem2 :pmilieu)
2 :is Meta-Link-For-Plan
3 :behavior
4  (((x: :extrem1 val) implies (x: :pmilieu (/ (+ val (x :extrem2)) 2)))
5   ((x: :extrem2 val) implies (x: :pmilieu (/ (+ val (x :extrem1)) 2)))
      ;; correspond à ligne 3 de la définition en ThingLab
6   ((x: :pmilieu val) implies (x: :extrem1 (- (* 2 val) (x :extrem2))))
7   ((x: :extrem2 val) implies (x: :extrem1 (- (* 2 (pmilieu x)) val)))
      ;; correspond à ligne 4 de la définition en ThingLab
8   ((x: :pmilieu val) implies (x: :extrem2 (- (* 2 val) (x :extrem1))))
9   ((x: :extrem1 val) implies (x: :extrem2 (- (* 2 (pmilieu x)) val)))))
```

Il faut noter qu'un mécanisme de traduction peut générer automatiquement de telles dépendances à partir de définitions comme celles de ThingLab.

Contrôleurs pour la planification. Dans notre modèle, le rôle d'un contrôleur est de maintenir la cohérence des dépendances en utilisant les informations spécifiées par le comportement dynamique. Pour la planification, le rôle du contrôleur reste le même, il utilise un plan pour maintenir la cohérence de l'ensemble des dépendances.

La solution proposée ici est basée sur le fait que tous les objets intervenant dans la planification possèdent le même contrôleur. Associer des contrôleurs à chacun des objets pose des problèmes pour la synthèse du plan et la gestion du contrôle (lignes 3 et 5). Lorsqu'un objet reçoit un message perturbant, le contrôleur détermine le plan à exécuter. Différentes techniques peuvent être mises en place, le plan peut être incrémentalement déduit d'un précédent plan ou reconstruit [Trom 97]. Ensuite le contrôle temporaire des messages est inhibé, le plan exécuté puis le contrôle rétabli.

```
1 contrôle-des-messages contr m-contrôlée receveur arguments
2   (détermine-plan contr m-contrôlée receveur arguments)
3   (stop-control contr)
4   (applique-plan contr)
5   (start-control contr)
```

Conclusion et extensions futures.

Notre modèle est extensible et peut utiliser des algorithmes de planification. Nous pensons qu'il reste du travail dans cette direction. Il serait en effet intéressant d'approfondir cette approche en précisant parmi les nombreux algorithmes de planification quels sont ceux qui sont les plus appropriés et de savoir s'il est possible de faire communiquer un solveur de contraintes et notre approche.

Où l'on reparle du déterminisme des programmes. Les langages réactifs synchrones [Benv 91] comme ESTEREL [Bous 91] ou LUSTRE [Halb 91] proposent une vision globale même s'ils possèdent qu'une notion de module. Ces langages sont basés sur l'hypothèse d'instructions de temps nul selon laquelle la réaction à un événement est terminée avant qu'un nouvel événement arrive. Cette hypothèse permet de faire abstraction du temps, de considérer les communications par signaux comme instantanées et de prouver le déterminisme des programmes. Cependant, ces langages sont loin d'être considérés comme des langages à objets. Certains travaux tentent d'apporter à ces langages une dynamicité qu'ils n'ont pas. Ainsi S. Dissoubray introduit un opérateur de reconfiguration dynamique pour ESTEREL [Diss 96]. D'autres travaux comme ceux de F. Boulanger choisissent une autre approche et intègrent des modules synchrones dans un cycle de développement d'objet [Boul 94].

Dans ce dernier travail, l'intégration d'un module synchrone consiste à transformer les automates fournis par les compilateurs de code synchrone intermédiaire (oc) de LUSTRE ou ESTEREL en classes C++. On obtient une classe par module. Cependant, cette intégration ne répond pas à nos objectifs. Nous ne souhaitons pas utiliser une hypothèse synchrone mais permettre l'expression de dépendances entre objets. De plus, notre modèle est homogène. Le programmeur utilise un même modèle. Il n'a pas à programmer dans plusieurs langages (la syntaxe et la sémantique des dépendances restent très proches de la programmation objet). Mais surtout, le déterminisme proposé par F. Boulanger est assuré à l'intérieur d'un module par la sémantique de l'automate généré par ESTEREL ou LUSTRE mais la communication entre des modules synchrones et des objets normaux peut détruire le déterminisme global du programme. L'approche prônée n'introduit qu'un déterminisme local aux modules et non au programme.

6.4 Globalité et gardes

L'introduction de l'opérateur `permitted-if` qui définit des gardes (c-à-d subordonne l'application d'une méthode à la valeur d'une expression) permet un maintien différent de la cohérence d'une dépendance. Celui-ci n'est plus garanti par réaction à une perturbation mais empêche la perturbation. Lors de la définition du modèle au chapitre 4, nous avons montré en 4.5 comment notre modélisation traite les problèmes liés aux possibles conflits entre gardes et messages compensatoires. La solution présentée alors était basée sur un retour brutal à un état cohérent. Dans la réalité, un tel retour n'est pas possible. Nous présentons les problèmes rencontrés et les solutions que nous proposons.

6.4.1 Le problème

Une garde, en interdisant l'application d'une méthode suite à la réception d'un message, contribue à assurer la cohérence de la dépendance. Cependant, en empêchant cette application, elle peut mettre en péril la cohérence d'une autre dépendance dont la cohérence doit être maintenue par l'application de cette même méthode. Il y a alors incohérence dans la définition même du graphe de dépendances. Une dépendance pour être maintenue interdit une action que l'autre exige !

Illustration. L'exemple qui suit est basé sur l'expression de contraintes géométriques très simples entre points. Définissons deux dépendances : la dépendance *plus50* qui spécifie qu'un point est à 50 pixels d'un autre point.

```
(deflink plus50 (:point1 :point2)
  :behavior
  (((move :point1 x y) implies (move :point2 (+ 50 x) y))
   ((move :point2 x y) implies (move :point2 (- x 50) y))))
```

La seconde dépendance qui stipule qu'un point doit toujours resté à gauche d'un autre.

```
(deflink agauche (:point1 :point2)
  :behavior
  (((move :point1 x y) permitted-if (> (x :point2) x))))
```

Ensuite, comme l'illustre la figure 6.12 nous déclarons les dépendances suivantes :

```
(define d1 (make plus50  :point1 p1 :point2 p2))
(define d2 (make plus50  :point1 p2 :point2 p3))
(define d3 (make agauche :point1 p3 :point2 p4))
```

Si l'on déplace ensuite le point `p1` à une abscisse de `40`, alors l'abscisse du point `p2` prend la valeur `90`. Ce dernier changement implique de déplacer le point `p3` à l'abscisse `140`. Or ce n'est pas possible car la dépendance entre `p3` et `p4` n'autorise le message `move` sur `p3` que s'il reste à gauche de `p4`. La dépendance `d2` est donc incohérente.

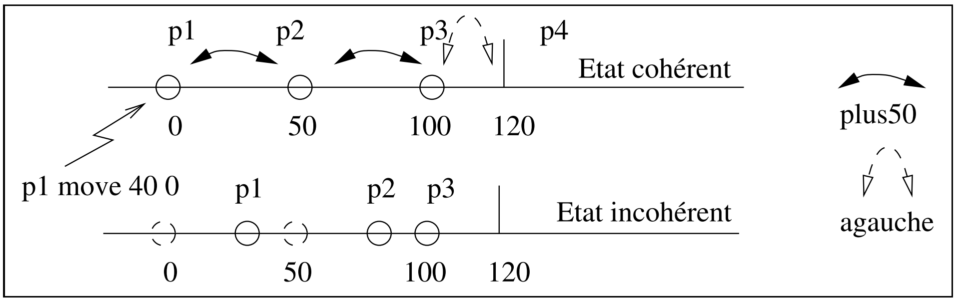


Figure 6.12: Illustration du problème de l'introduction de gardes : après propagation dans une partie du graphe certaines dépendances sont incohérentes. Ici les points P2 et P3 ne satisfont plus la dépendance PLUS50.

Remarque. Ce simple exemple exprime clairement le cœur du problème liée à l'introduction de gardes. Ce problème peut se produire avec des objets et des dépendances ayant des comportements complexes.

6.4.2 Des solutions

Ce problème ne semble pas pris en compte par les langages que nous avons étudiés (voir au chapitre 2). En effet, soit les langages ne proposent qu'un comportement réactif, soit ils proposent des gardes mais ne traitent pas ce problème. Chacune des solutions que nous présentons maintenant peut apporter un élément de solution dans un cas particulier. Mais il n'existe pas de solution idéale.

Détection statique. Une analyse statique du graphe des dépendances peut avertir le programmeur de l'éventualité de tels problèmes. Cette solution est très limitée car certaines dépendances peuvent former un graphe contenant de potentiels problèmes sans pour autant que ces problèmes arrivent lors de l'exécution du programme.

A l'aide de fantômes. Une solution à ce problème est d'opérer la phase de propagation sur des copies des objets liés puis si la cohérence globale est satisfaite, soit de repropager sur les vrais objets, soit de substituer les copies aux vrais objets. Cette solution demande une mise en œuvre lourde : copier un objet n'est pas toujours évident, la gestion des cycles entre objets alourdit le mécanisme.

Par contre, dans certains domaines, il est possible de définir clairement quel doit être le degré de copie des objets. C'est le cas du dessin ou du positionnement interactif de figures géométriques. En effet, il est possible d'utiliser à la place de copies complètes des objets leurs contours : des *fantômes*. La propagation s'effectue sur les contours qui détiennent l'essence des objets graphiques (leurs limites et coordonnées) et lorsque la cohérence globale est assurée le système substitue les objets réels à leurs contours. Une telle solution est utilisée comme modèle de maintien de la cohérence par le système SEMDRAW [CHAB 93, CHAB 94].

Accepter des incohérences temporaires. On peut imaginer une solution encore une fois réalisable pour le positionnement ou le dessin vectoriel.

L'utilisateur définit les dépendances qu'il souhaite pour ces objets puis déclare de telles dépendances entre ces objets graphiques. Lors de la manipulation de ces objets, certaines dépendances peuvent être laissées incohérences. Dans une phase de finition du placement des objets et sur demande de l'utilisateur, le système se charge de détecter les dépendances non cohérentes[7] et de les signaler à l'utilisateur qui en dernier ressort ajuste le placement des objets.

Synthèse automatique de gardes impossible. Dans l'absolu, c'est-à-dire pour des méthodes dont on ne connaît pas la sémantique, la synthèse automatique de gardes prenant en compte la globalité du graphe est impossible. Par contre, il semble que dans le cas particulier de contraintes mathématiques linéaires, le système puisse définir automatiquement des gardes.

> Dans l'exemple, une garde qui interdit au premier point de se déplacer au-delà de l'abscisse 19 et au second au-delà de 69 est envisageable.

Cependant, cette solution est impossible à mettre en œuvre dans le cas d'objets et de dépendances quelconques. En effet, les dépendances expriment des contraintes entre objets dont on ne connaît pas le sens. Le système maintient les dépendances d'après leurs comportements dynamiques mais ne peut déduire des informations supplémentaires sur le sens des méthodes contrôlées. De plus, l'idée d'une migration automatique des gardes pose plus de problèmes qu'elle n'en résout.

Limitation aux dépendances. Parmi tous les travaux que nous avons étudié (voir au chapitre 2), deux seulement permettent d'interdire l'exécution de méthodes par le biais de dépendances. Les auteurs de PROCOL [VAN 91] offrent la possibilité de définir des gardes associées cette fois-ci à l'objet et des réactions sous forme de propagateurs de contraintes entre objets (voir en annexe D.2) mais ils n'abordent pas ce type de problèmes. AGHA et FRØLUND dans [FRØL 93] introduisent des gardes (définies par l'opérateur `disables`). Cependant, la sémantique de cet opérateur liée à celle de l'opérateur `udpates` ne pose pas ces problèmes. En effet, l'opérateur `updates` ne permet que de modifier l'état d'un Synchroniseur et non d'envoyer des messages aux objets participant à d'une dépendance. De plus, les gardes ne sont exprimées qu'en termes de l'état d'un Synchroniseur. De cette manière, ils limitent de manière forte à la fois les problèmes et les possibilités d'expressions (voir en D.1). Les auteurs des Synchroniseurs sont conscients de la nécessité de prendre en compte les objets dans les réactions et gardes comme l'illustre la citation suivante : *« It might be desirable for synchronizers to be able to receive messages as well as trigger new activities. »* [FRØL 93]. Par cette volonté, ils se rapprochent de notre modèle et de ces problèmes.

Pour notre part, si nous limitons l'utilisation des messages compensatoires et des gardes de la même manière que les Synchroniseurs, nous éliminons les problèmes liés aux gardes ou aux problèmes plus généralement liés à la propagation de messages. L'exemple d'exclusion que nous

[7] Ce qui implique la donnée de prédicats vérifiant la cohérence des dépendances.

avons présenté au chapitre 5 illustre alors parfaitement la différence d'expression qu'une telle limite apporte. Notre modèle offre la possibilité de définir des Synchroniseurs à l'aide de dépendances [8].

Remarques générales. Il faut noter que les problèmes que nous mentionnons ici existent de manière naturelle dans les langages à objets ou dans les bases de données actives [Hans 93, Wido 96]. Il est facile de spécialiser une méthode en contraignant son exécution suivant la valeur d'une expression. Une telle subordination peut être en contradiction avec la bonne exécution du programme lors de l'appel de cette méthode par un autre objet. Ce faisant les objets peuvent être dans un état incohérent. Ce type de situation n'apparaît pas comme un problème. Il est de la responsabilité du programmeur de bien appréhender la globalité de la situation et de bien spécifier de tels informations dans la description de cet objet. Notre modèle en se préoccupant essentiellement des dépendances entre objets met en avant ces problèmes.

6.5 Conclusion

En guise de conclusion nous montrons les principales avancées de notre modèle par rapport au modèle de dépendances proposé par son précurseur Othelo [Forn 90a].

Quelques propriétés du modèle.

Les dépendances proposées répondent aux problèmes énoncés au chapitre 1. Les dépendances sont des entités de même statut et importance que les classes. Elles sont *indépendantes* des classes des objets sur lesquels elles portent. Elles constituent des *entités autonomes*. Cette autonomie permet en particulier d'exprimer des dépendances au niveau des instances sans pour autant impliquer que toutes les instances d'une classe soient concernées.

La possibilité de définir des informations propres (structure et fonctionnalités) et le comportement dynamique d'une dépendance permet d'établir clairement la sémantique de la dépendance. L'abstraction ainsi proposée permet une meilleure lisibilité et définition des dépendances comme des classes d'objets participant. Cette séparation logique entre les objets et les dépendances permet de réduire la complexité des objets liés et par la même offre de plus grande chance de réutilisation des objets. Notre modèle préserve le principe d'encapsulation. Le comportement dynamique des dépendances est exclusivement exprimé à l'aide des interfaces des objets participants.

Spécificités de notre modèle de dépendances.

Le modèle proposé ici améliore le modèle de dépendances proposé dans Othelo sur de nombreux points.

Dépendances générales. Les dépendances d'Othelo étaient binaires et orientées. Un objet jouait le rôle de maître et l'autre d'esclave. Ce faisant les dépendances entre plusieurs objets et impliquant des dépendances multi-directionnelles étaient de définitions et d'utilisation délicates.

Notre modèle uniformise et généralise ces dépendances. Les dépendances sont n-aires et non-orientées : plusieurs objets peuvent nécessiter différents messages compensatoires. Les dépendances d'Othelo sont simplement un cas particulier. Notre modèle permet la gestion de groupes d'objets. Un ensemble d'objets participants est précisé de manière implicite (sans avoir à nommer explicitement chacun des éléments de l'ensemble). De plus, un objet peut être retiré ou ajouté dynamiquement d'un groupe.

[8] Si l'on ne considère pas l'opérateur d'atomicité `atomic` qui permet de considérer la réception de plusieurs messages comme atomique.

Opérateurs. Au niveau des dépendances, la principale différence réside dans l'adaptabilité de notre modèle par rapport à l'expressivité du comportement dynamique. En effet, en OTHELO, le comportement dynamique se limitait à un comportement réactif. Le modèle ne proposait aucune notion d'opérateurs, seule la donnée de points d'activation des couples d'actions/réactions (des actions perturbant la cohérence de la dépendance et des réactions la rétablissant) permettait de définir le comportement dynamique. Ce comportement était alors difficilement modifiable et extensible.

Dans notre modèle, les règles d'interactions définissent le comportement dynamique. L'interprétation de ces règles est spécifiée par des opérateurs. Chaque opérateur définit le sens des actions et du contrôle des messages. Ainsi le comportement dynamique ne se limite pas à l'expression de conséquences pour rétablir la cohérence d'une dépendance. Différents opérateurs peuvent être définis. La notion de gardes ou de délégation est facilement introduite. L'adaptabilité du comportement dynamique permet l'évolution de l'expression même de la notion de dépendance. Ainsi il est possible de spécifier des interactions entre objets et non simplement des dépendances réactives.

Une approche globale du maintien. Dans ce chapitre, nous avons montré comment l'introduction de nouvelles entités, des contrôleurs, permet une abstraction du mécanisme même de maintien de la cohérence des dépendances. A notre connaissance, il n'existe pas de travaux faisant état de telles entités. Hormis les systèmes de contraintes qui proposent un mécanisme global de résolution non adaptable, les travaux que nous avons étudiés introduisent la notion de dépendance mais ne gèrent pas la dépendance en tant qu'entité connectée à d'autres dépendances.

Nous avons montré comment le modèle propose un contrôle optimal ; c'est-à-dire minimal et adapté à chaque opérateur. De plus, l'abstraction fournie par les contrôleurs offre la possibilité de définir de nouveaux comportements. Le modèle est alors extensible : de nouveaux opérateurs peuvent être définis, et adaptable : la gestion des dépendances peut être modifiée. Nous avons montré en particulier comment différentes stratégies de détection de cycles ou de planification étaient possibles.

Partie III

Architecture et MOP du langage FLO

Dans cette partie, nous abordons le troisième objectif de cette thèse : l'extensibilité et l'adaptabilité du langage implémentant le modèle que nous avons décrit dans la partie précédente. Cette partie a pour but de décrire l'architecture et le protocole[9] que nous proposons pour la gestion des dépendances. Ainsi nous montrons comment les dépendances se fondent dans le modèle objet et surtout comment une telle intégration présentent les différents niveaux d'extensions possibles.

Dans cette troisième partie, certaines extensions illustrent la possibilité d'adaptation de notre modèle. Cependant un programmeur n'a pas à savoir mettre en œuvre de telles extensions : c'est le rôle d'un méta-programmeur que de mettre à la disposition des autres programmeurs les extensions souhaitées. Cette partie doit donc être lues avec l'œil d'un *futur méta-programmeur de* FLO.

Un langage extensible pour un modèle extensible.

Le modèle présenté précédemment est extensible : la gestion des dépendances peut être enrichie pour prendre en compte de nouveaux opérateurs, de nouveaux comportements de la propagation sont possibles. C'est pourquoi nous avons choisi un langage extensible comme langage d'implémentation. Ainsi notre choix s'est porté sur des langages comme CLOS [Keen 89, Stee 90, Kicz 91, Pae 93], NeoClasstalk [Brio 89a, Coin 92, Riva 96a] ou STklos [Gall 94, Gall 96] qui permettent aux programmeurs de particulariser leurs aspects fondamentaux : création de classes, envoi de messages, création d'instances... Ce faisant, le langage résultant, nommé FLO, est un langage extensible intégrant notre modèle de dépendance.

Plan de cette partie.

Cette partie est structurée en trois chapitres de la manière suivante :

Programmation et systèmes extensibles. Tout d'abord, nous présentons le contexte général des langages extensibles. En effet, l'apparition de ces langages est une réponse à un besoin des utilisateurs de langages. Pour cela, nous définissons le vocabulaire propre à ces langages ou aux techniques employées (MOP, méta-objets...). Ensuite, nous présentons les difficultés spécifiques rencontrées lors de la conception de ces langages.

Architecture et MOP des dépendances. Ensuite, nous nous concentrons sur l'architecture minimale du modèle au niveau des dépendances et nous montrons comment nous l'étendons afin d'avoir de nouvelles fonctionnalités.

Contrôle et globalité. Le modèle proposé étant basé sur un contrôle de l'envoi de messages, nous étudions les différentes possibilités pour le mettre en œuvre et nous présentons les fonctionnalités des contrôleurs. Pour finir, nous montrons comment les contrôleurs offrent un nouveau niveau d'extension au langage.

Remarques. Différentes implémentations de ce modèle ont été effectuées (CLOS, STklos, ObjVlisp) ou sont en cours d'implémentation (NeoClasstalk, OpenC++). Nous présentons dans cette partie l'architecture globale du langage. Le code utilisé pour illustrer notre discours est en STklos [Gall 94, Gall 96], un langage objet basé sur CLOS et le langage Scheme et Tk/Tcl. Cependant, le protocole proposé est adaptable à différents langages car nous n'avons pas utilisé la multi-discrimination offerte par les fonctions génériques de CLOS lors du choix des méthodes effectives. Cette restriction de l'utilisation des fonctions génériques nous conduit naturellement à ne pas utiliser ce terme. Nous ne parlons pas d'applications de fonctions mais d'envoi de messages et non pas de fonctions génériques mais de méthodes : celles-ci étant alors sélectionnables exclusivement par rapport à la classe de l'objet receveur du message.

[9] Un protocole est un ensemble de méthodes, de classes et leurs interactions.

7

Réflexivité et systèmes ouverts

« The metaobject protocol approach, [...], is based on the idea that one can and should "open languages up", allowing users to adjust the design and implementation to suit their particular needs. In other words users are encouraged to participate in the language design process. » [Kicz 91].

.

De nombreux langages réflexifs sont apparus ces dernières années : langages objets [Maes 87a, Bobr 86, Coin 87, Brio 89a, Chib 93a, Danf 94a, Chib 95, Bran 96, Riva 96b], langages d'acteurs [Ferb 84, Ferb 88, Carl 93, Brio 94, Brio 96], programmation concurrente [Wata 88, Ishi 91, Masu 92, Mats 92], langages à prototypes [Mule 93b, Mule 93a, Mule 95a]. L'utilisation de protocoles méta-objet a permis de mettre en place des mécanismes variés comme des types de données atomiques [Stro 95], l'introduction d'objets persistants [Paep 90], des boîtes à outils graphiques [Rao 91, Gall 96] et la construction de systèmes d'exploitation. Après avoir posé le vocabulaire, nous analysons les besoins qui ont favorisé l'émergence des systèmes dit *ouverts*. Nous abordons ensuite les difficultés spécifiques à la conception de ces systèmes. Nous présentons rapidement deux protocoles méta-objet (MOP) en présentant celui de CLOS [Kicz 91] et de CodA [McJa 95]. Nous avons choisi CLOS car son MOP constitue la référence en la matière, mais aussi le MOP nommé CodA qui choisit une factorisation des méta-comportements originale et plus générale centrée autour du concept d'objet.

7.1 Concepts et définitions

« La réflexion désigne l'ensemble des préoccupations visant à rendre visible dans le langage certains mécanismes internes. » [Mule 95a].

La réflexion est à la base des langages ouverts. Ce concept qui ne se limite pas exclusivement aux langages à objets existe aussi en programmation logique ou fonctionnelle [Deme 95]. B. Smith fut le premier à introduire un tel concept dans un langage de programmation : 3-Lisp, son langage réflexif, était un langage fonctionnel Lisp [Smit 84]. Ainsi B. Smith définit la réflexion comme : *« An entity's integral ability to represent, operate on, and otherwise deal with itself in the same way that it represents, operates on and deals with its primary subject matter ».*

Dans le cadre des langages de programmation, cette définition peut s'énoncer comme : *« Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation :* introspection *and* intercession *[...] Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called* reification. » [Bobr 93]

P. Maes a pour sa part proposé, dans le premier chapitre de sa thèse [Maes 87b], des définitions précises permettant de caractériser clairement la programmation réflexive. Nous reprenons ici ces définitions :

- A **computational system** is something that **reasons** about and **acts** upon some part of the world, called the **domain** of the system (p 13).
- A computational system may also be **causally connected** to its domain. This means that the system and its domain are linked in such a way that if one of the two changes, this leads to an effect upon the other (p 15).
- A **meta-system** is a computational system that has as its domain another computational system, called its **object-system**. [...] A meta-system has a representation of its object-system in its data. Its program specifies **meta-computation** about the object-system and is therefore called a **meta-program** (p 17).
- **Reflection** is the process of reasoning about and/or acting upon oneself (p 19).
- A **reflective system** is a causally connected meta-system that has as object-system itself. The data of a reflective system contain, besides the representation of some part of the external world, also a causally connected representation of itself, called **self-representation** of the system. [...] When a system is reasoning or acting upon itself, we speak of **reflective computation** (p 19).
- A language with a **reflective architecture** is a language in which all systems have access to a causally connected representation of themselves.
- A programming environment has a **meta-level architecture** if it has an architecture which supports meta-computation, without supporting reflective computation (p 34).
- The **meta-object** of an object X represents the explicit information about X (e.g. about its behavior and its implementation). The object X itself groups the information about the entity of domain it represents (p 120).

Quelques commentaires. La principale différence entre une architecture à méta-niveaux et une architecture réflexive réside dans le fait qu'une architecture à méta-niveaux ne propose qu'un accès statique à la représentation du système, alors qu'une architecture réflexive offre un accès dynamique, ce qui permet au langage lui-même, durant son exécution, de modifier son propre fonctionnement.

Lorsqu'on parle de réflexion, il est courant de distinguer deux types de réflexion : *structurelle* et *comportementale*. La réflexion structurelle nécessite de la part du langage d'offrir une réification complète à la fois du programme et des types de données abstraits. La réflexion comportementale implique la capacité du langage à offrir à la fois une réification de sa propre sémantique et des données qu'il utilise pour exécuter le programme. Dans le cadre de la programmation objet, la réflexion structurelle s'exprime, par exemple, par la réification des classes [Coin 87]. La réflexion comportementale s'exprime en particulier au travers du contrôle des envois de messages (recherche de la méthode associée et application de la méthode)[Mule 93a]. De plus amples informations peuvent être trouvées dans [Kicz 91, Pae 93, Deme 95, Mule 95a, Riva 96a].

7.2 Une nouvelle approche pour la conception de langages

Il arrive que les approches traditionnelles pour la conception des langages sous-estiment les besoins des programmeurs. Ce faisant ceux-ci sont obligés de passer outre l'abstraction offerte par ces langages afin de satisfaire leurs problèmes. Après avoir présenté, les limites des approches traditionnelles, nous présentons comment la nouvelle approche de conception des systèmes dits *ouverts* apporte une réponse à ces problèmes. Nous montrons les principes émergeant de cette nouvelle approche ainsi que ses difficultés de mise en œuvre.

7.2.1 Limitations des approches traditionnelles

« The view of abstraction on which software engineering is based does not support the reality of practice: it suggests that abstractions hide their implementation, whereas the evidence is that this is not generally possible. » [Kicz 92a].

L'approche classique de la conception des langages de programmation est de proposer des langages possédant une sémantique ou un comportement figé. Ces langages offrent alors une abstraction de leur comportement. Les utilisateurs de ces langages considèrent ces langages comme des boîtes noires sur lesquelles ils vont construire leurs applications. L'abstraction ainsi proposée est utile et nécessaire car elle permet aux utilisateurs de ces programmes de ne pas se soucier des aspects liés à l'implémentation du langage. Cette vision de la conception des langages trouve une justification dans la nécessité de proposer des implémentations plus facilement portables d'une machine à une autre, des techniques de compilation plus efficace, une uniformité et cohérence au langage ...

Cependant, comme le présente fort justement G. Kiczales dans [Kicz 92a], cette vision basée sur une trop stricte utilisation de l'abstraction pose des problèmes. G. Kiczales remet en cause une forme d'abstraction qui couperait complètement le programmeur de l'implémentation. En effet, alors même que l'abstraction correspond à une nécessité par rapport au raisonnement humain et qu'elle est importante dans de nombreuses activités intellectuelles, lors de la conception de programmes, des problèmes liés à l'implémentation (inefficacité, surcharge, ...) oblige le programmeur à prendre en compte l'implémentation du langage allant ainsi à l'encontre de l'abstraction.

Les solutions qui s'offrent alors au programmeur sont de deux ordres : soit il se voit contraint de réimplémenter les fonctionnalités souhaitées pour une application donnée, ce qui complique considérablement le code ; soit il peut utiliser certains aspects de l'implémentation à contre emploi et *«coder entre les lignes»*. Par exemple, le programmeur peut être forcer d'utiliser la pagination de la mémoire et le groupement des objets de façon à accélérer les accès en mémoire. Cependant, ce faisant l'abstraction n'est plus utilisée correctement et ces choix sont préjudiciables lors de l'évolution de l'application (changement de système, portage sur une autre machine,...).

Les langages de programmation, et plus généralement les systèmes, dits *ouverts* proposent une nouvelle approche.

7.2.2 Une alternative : la conception de langages ouverts

La vision proposée par les auteurs de CLOS est basée sur l'idée que les langages doivent être *ouverts*, laissant aux utilisateurs la possibilité d'en particulariser certains aspects. Cependant cette ouverture est délicate car elle ne doit pas compromettre la portabilité ou l'efficacité du langage. Elle consiste à définir des protocoles méta-objets, *(Metaobject Procotol ou MOP)*. Un MOP est une interface du langage qui donne aux utilisateurs la possibilité de modifier incrémentalement le comportement du langage et son implémentation. Un MOP est défini par un ensemble de méta-objets et leurs interactions et représente certains aspects de l'implémentation du langage pour lequel il est défini.

Les langages offrant des protocoles méta-objets gomment la distinction entre les utilisateurs des langages et les concepteurs car ils permettent l'utilisation d'une double interface (voir figure 7.1). Ainsi un programmeur doit éventuellement écrire deux sortes de programmes : un programme à l'aide du langage dit *de base* dans lequel il utilise les abstractions mises à sa disposition sans se soucier des détails de l'implémentation ; l'approche reste alors traditionnelle. Par contre, lorsque le programmeur est confronté à des situations pour lesquelles l'implémentation n'est pas adéquate, il a recours au MOP afin de particulariser l'implémentation pour qu'elle corresponde plus exactement à ses besoins.

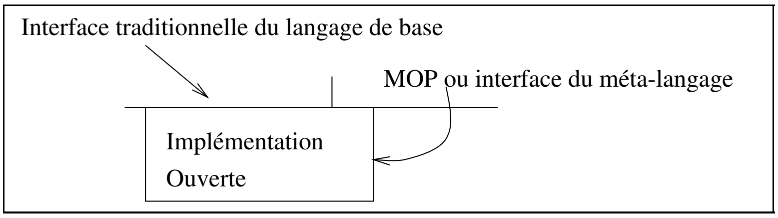


Figure 7.1: Une double interface : une interface traditionnelle du langage et une interface du méta-langage.

Remarques. Le fait d'ajouter un MOP à un langage permet à celui-ci d'être la base commune à une *région* de langages flexibles et extensibles (voir figure 7.2). Une région de langages évoque des langages relativement proches dans la mesure où seulement quelques aspects ont été adaptés ou modifiés.

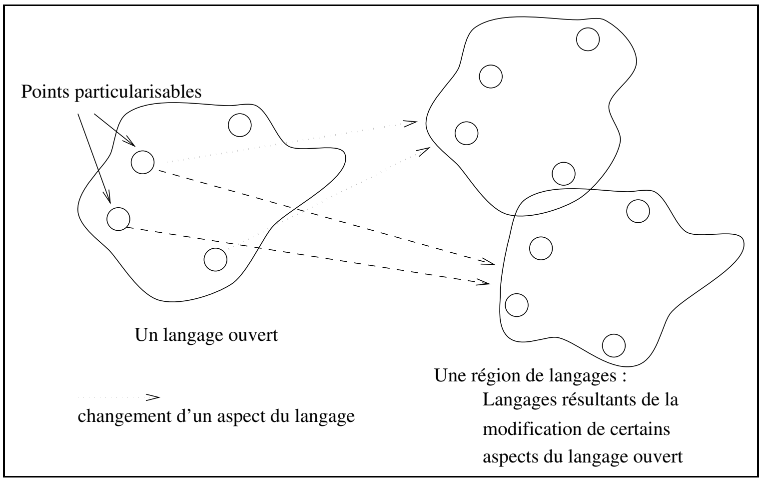


Figure 7.2: Un langage ouvert : un générateur d'une région de langages.

Apport de l'approche objet. L'utilisation de la technologie objet pour implémenter de tels langages et leur protocole méta-objet permet d'utiliser l'héritage et la spécialisation afin de particulariser finement les aspects choisis et de définir le comportement initial du langage. Deux types de protocole peuvent être mis en place :

– les protocoles d'*introspection*, c'est-à-dire les protocoles permettant d'avoir accès, de manière contemplative[1], à des données généralement cachées. De tels protocoles sont utiles pour la construction d'aide à la programmation (*browsers, debuggeurs...*).
– Les protocoles dit *d'interventions*, *(intercessory protocols)* qui permettent de modifier le comportement même du programme[2].

Ces deux protocoles sont basés sur la réification de certaines données significatives du langage (classes, méthodes, variables d'instances ...).

Dans le cadre de CLOS, le langage d'implémentation sur lequel est construit le MOP est CLOS lui-même, ce qui permet une auto-définition du langage. Cependant, alors que le fait d'avoir un MOP dépend essentiellement d'une technologie objet, le langage de base (pour lequel on définit un MOP) n'a pas nécessairement à être objet. Ainsi PLOY [VAHD 92] qui est un interprète SCHEME

[1]Il s'agit de regarder à l'intérieur des boîtes noires.
[2]Il s'agit là d'agir sur les données et les comportements des boîtes noires.

offre un MOP définit en CLOS permettant d'expérimenter différentes sémantiques pour le langage SCHEME [R4R 90].

7.2.3 Principes de conception

« ...the designer of a dual-interface works : iteratively. They start with traditional abstraction and gradually add a meta-level interface as it becomes clear what kinds of ways a close implementation can cause problems for the users. » [KICZ 92a].

La conception de systèmes ouverts est délicate, l'expérience acquise par les auteurs de CLOS a permis de mettre en avant quelques principes étroitement corrélés [KICZ 92a] :

Étendue du contrôle: lorsqu'un programmeur utilise l'interface du méta-langage pour particulariser un des aspects de l'implémentation, il doit avoir la possibilité de choisir l'étendue de la modification aussi bien en termes d'actions (par exemple, l'accès aux variables d'instances) que d'objets (par exemple, la classe `Point`).

Séparation conceptuelle: il doit être possible de spécialiser un aspect particulier à l'aide de l'interface du méta-langage sans pour autant avoir à *comprendre l'ensemble* de cette interface. Ce point est étroitement dépendant de la finesse accordée à cette interface. Par contre, comme certains aspects sont liés, il s'agit alors de comprendre un comportement sous ensemble local du MOP.

Incrémentalité: le programmeur qui choisit de particulariser un aspect du langage, ne doit pas avoir la responsabilité de l'ensemble de l'implémentation mais seulement de sa modification. Il s'agit de définir seulement les parties nécessaires au nouveau comportement et de réutiliser le reste de l'implémentation. L'utilisation de langages objets pour la définition de méta-langages facilitent grandement cette tâche.

Robustesse: lorsqu'une extension définie par un utilisateur contient une erreur, celle-ci doit avoir des effets limités et ne pas pénaliser d'autres extensions.

7.2.4 Difficultés de conception

*« Building an open system like this [*CLOS *] is therefore generally much more time consuming and frustrating than the construction of a more traditional design »* [PAEP 93].

Une des principales difficultés mise en évidence lors de la conception de protocoles méta-objet est le choix des aspects du langage qui doivent être ouverts tout en tenant compte des principes énoncés précédemment. Ces choix sont lourds de conséquences car ils induisent des familles potentiellement différentes de langages.

L'autre difficulté réside dans le choix du niveau de détails à offrir lors de la définition d'un protocole : lorsque trop de détails sont spécifiés, les *implémenteurs* du MOP n'ont pas assez de liberté pour introduire les optimisations nécessaires. Dans le cas contraire, il devient difficile pour les utilisateurs de savoir où et comment les modifications doivent être effectuées [KICZ 92b]. La meilleure solution afin de trouver le bon niveau de détail reste l'expérience apportée par la conception d'applications. En effet, une des caractéristiques primordiale et inhérente à la conception de systèmes ouverts est liée au processus itératif mis en jeu. Ainsi le concepteur d'un MOP n'a pas à déterminer *a priori* quels sont les aspects de l'implémentation du langage qui doivent être ouverts, mais bien d'ajuster au fil de applications les possibilités de particularisation offertes.

7.3 Deux exemples de MOP

Nous avons choisi de présenter le protocole méta-objet de CLOS [ATTA 89, KICZ 91, Pae 93] car, de part son utilisation et l'étendue des concepts mis en avant, il est considéré comme la quintessence des protocoles méta-objets [Mule 95b]. Le langage CLOS a été le sujet de nombreux articles et livres [Keen 89, Stee 90]. Comme second exemple, nous avons choisi CodA pour sa manière différente d'aborder la répartition des aspects particularisables.

7.3.1 Le MOP de Clos

La conception d'un langage ouvert comme CLOS et son MOP repose principalement sur l'utilisation de la réflexivité du langage et de la technologie objet. La réflexivité rend possible l'accès à l'implémentation en offrant un niveau d'abstraction suffisant pour qu'à la fois les implémenteurs utilisent les spécificités de machines cibles et les utilisateurs ne soient pas engloutis par les détails inutiles. La réflexivité permet une mise en œuvre effective des changements de comportement du système [Maes 88]. La technologie objet permet de définir : les aspects du langage méritant des particularisations, sous forme d'objets et les opérations associées, le comportement par défaut de tels objets et la possibilité d'offrir différents niveaux de particularisation du langage.

Le MOP de CLOS considère cinq types de méta-objets différents implémentés sous forme de classes : `class`, `slot-definition`, `generic-funtion`, `method` et `method-combination`. C'est à l'aide de ces cinq méta-objets que le comportement initial du langage est défini. Le MOP de CLOS est lui-même défini en CLOS, ainsi les extensions du langage peuvent être réalisées à partir du langage même par spécialisation des comportements des cinq méta-objets initiaux. Cette auto-description, permise par la réflexivité du langage, offre ainsi une plus grande portabilité des extensions.

CLOS permet de particulariser aussi bien ses aspects structurels que ses aspects comportementaux : la création d'instance, l'accès aux variables d'instances, la combinaison des méthodes, l'application des fonctions génériques, l'héritage sont décrit en CLOS et particularisables. Différents niveaux de particularisation sont possibles, il peut s'agir d'une classe, d'une variable d'instance, aussi bien que de l'application même des fonctions génériques ou méthodes. Les points d'entrée du MOP sont très vastes et offrent la possibilité de modifier des parties précises du langage sans pour autant le modifier globalement. Nous renvoyons le lecteur à [Kicz 91, Pae 93] pour une description complète du MOP.

Un exemple : le contrôle de l'application des fonctions génériques. Bien que nous ayons précisé que notre modèle est basé sur un mécanisme d'envoi de messages et non d'applications de fonctions génériques, nous illustrons la possibilité offerte en CLOS de particulariser l'application de celles-ci. En effet, nous considérons l'envoi de message ou l'application des fonctions génériques comme des mécanismes fondamentaux d'un langage à objets que peu de langages offrent la possibilité de modifier.

Supposons que pour des raisons d'analyse d'un programme, on souhaite savoir combien de fois certaines fonctions génériques ont été appliquées. Pour cela, il suffit de définir la classe `counting-gf` : une nouvelle classe de fonction générique possédant la variable d'instance `howmany`?

```
(defclass counting-gf (STANDARD-GENERIC-FUNCTION)
  ((howmany :initform 0 :accessor howmany))
  (:metaclass FUNCALLABLE-STANDARD-CLASS))
```

Ensuite nous redéfinissons la fonction `compute-discriminating-function` qui est responsable de l'application[3] des fonctions génériques. Cette redéfinition précise qu'avant d'appliquer la fonction générique (ligne 3), le nombre d'application de la fonction générique traitée est incrémenté (ligne 4). L'application de fonction générique est effectuée en invoquant le comportement standard à l'aide de la fonction `call-next-method`.

[3] Cette fonction rend une lambda-expression dans un but d'optimisation. Il s'agit d'un technique dite de *mémoization* : le MOP dissocie les aspects stockables; c'est-à-dire calculables une seule fois, de ceux devant être calculés à chaque exécution. Ceux de la première catégorie sont alors stockés afin d'être réutilisés sans recalcul.

```
(defmethod compute-discriminating-function ((gf counting-gf))
  (let ((normal-dfun (call-next-method)))
    #'(lambda (&rest args)
        (incf (howmany gf))
        (apply normal-dfun args))))
```

Ensuite il suffit de déclarer une fonction générique comme instance de la classe `counting-gf` pour que ce comportement soit exécuté.

```
(defgeneric open (l)
  (:generic-function-class counting-gf))
```

7.3.2 CodA : un méta-protocole composite

J. Mc Affer dans [McJa 95] propose un protocole méta-objet baptisé CodA. L'approche choisie est différente de celle de CLOS. En effet, CodA n'est pas un langage réflexif, mais plutôt un protocole qui n'est pas lié à un langage particulier. Outre cette première distinction, CodA applique une décomposition logicielle au méta-niveau. De plus, CodA n'est un méta-protocole lié aux caractéristiques du langage (classes, méthodes...) comme c'est le cas du MOP de CLOS. Au contraire, CodA met l'accent sur les fonctionnalités de l'objet. Les différents comportements d'un objet (envoi de message, réception, état, ...) sont les éléments de base du MOP de CodA.

Un méta-niveau est composé de méta-composants, des objets représentant des comportements spécifiques de la gestion des objets de base. Chaque objet possède alors un méta-niveau conceptuel qui regroupe plusieurs méta-composants spécialisés. CodA définit sept méta-composants qui peuvent être à leur tour spécialisés ou étendus. Nous les présentons rapidement :

- Le composant `send` gère les interactions entre l'expéditeur et le receveur d'un message. Il achemine un message jusqu'au composant `accept` du receveur.
- Le composant `accept` précise si le message est accepté, ce dernier pouvant alors être stocké par le composant `queue` pour traitement ultérieur.
- Le composant `queue` gère les messages en attente de traitement.
- Le composant `receive` correspond à la phase finale de réception d'un message qui est déclenchée lorsqu'un objet recherche un message à exécuter.
- Le composant `protocol` a en charge la recherche de la méthode à exécuter.
- Le composant `execution` a en charge l'application de la méthode trouvée.
- Le composant `state` décrit la gestion de l'état interne d'un objet et les modes d'accès.

J. Mc Affer centre son MOP sur les activités des objets et non sur le langage l'implémentant. La décomposition (en méta-composants) d'un méta-niveau offre la possibilité de particulariser différemment le comportement des objets en offrant une vision plus fine du dit comportement. Contrairement à la décomposition de messages à l'aide du protocole **lookup** $\circ$ **apply** utilisé dans Moostrap [Mule 95a], CodA décompose l'envoi de message à la fois du point de vue de l'objet émetteur et de l'objet receveur. CodA offre une description du comportement des objets au travers de méta-composants distincts (émission, réception des messages...) contrairement à CLOS ou à Classtalk dans lesquels la description du comportement des objets passent par une description des éléments du langage (classes, méthodes, variables, ...). La différence réside dans le choix de factorisation des comportements : en CLOS les méta-comportements sont plus ou moins répartis dans les cinq méta-objets de base, dans CodA un méta-comportement est regroupé dans un même méta-objet : un méta-composant.

7.4 Conclusion

La conception de langages ouverts n'assurent pas seulement une durée de vie plus grande aux langages mêmes mais aussi à leurs applications. Nous seulement les applications peuvent évoluer sous la pression des utilisateurs, mais le langage même peut évoluer afin d'offrir de nouvelles possibilités qui à leur tour seront utilisées pour faire évoluer les applications. Il y a ainsi une forte interaction entre l'évolutivité d'un langage et celles de ses applications. Le manque de flexibilité des langages traditionnels conduit à une évolution des applications non continue ou limitée par le langage. L'utilisation de langages ouverts permet une évolution continue et adaptée a de nouvelles exigences. Cette situation n'est pas fictive. Nous sommes en contact avec la société Nichimen Graphics qui développe depuis une dizaine d'années un produit industriel hautement interactif pour la conception d'images de synthèse. En dix ans, beaucoup de nouvelles techniques et de machines sont apparues. Leur produit a évolué et évolue encore conjointement au langage utilisé pour l'implémenter. Ainsi pour proposer un produit toujours plus adapté aux besoins des utilisateurs, CLOS, le langage utilisé, continue d'être enrichi. Cet enrichissement du langage permet alors une évolutivité continue et souple des applications.

C'est dans cet esprit que nous avons voulu proposer une intégration de dépendances dans un modèle à classes. Nous ne nous sommes pas limités à proposer une implémentation stricte et immuable d'un mécanisme de dépendance. Bien au contraire, nous nous sommes efforcés de déterminer les mécanismes qui permettent la meilleure évolution et adaptation pour des utilisations futures des dépendances. D'autre part, comme nous utilisons un langage adaptable notre implémentation se devait de se fondre dans le langage en étant à son tour adaptable.

8

Architecture et dépendances

« A living language is one that is able to adapt and evolve as the needs of the people who use it change. A language which is not monolithic, but that is instead built of dynamic first-class objects can provide this kind of flexibility. » [Foot 93].

En exagérant à l'extrême, nous pouvons dire que le langage FLO, présenté aux chapitres précédents, se limite à deux fonctions : `deflink` pour la définition et `make` pour la déclaration de dépendances. Nous présentons dans cette partie comment l'architecture du langage et le méta-objet protocole proposé pour la gestion des dépendances.

Dans un premier temps, nous présentons comment les dépendances sont représentées dans FLO. Ce choix nous amène ensuite à définir les différentes classes intervenant dans FLO et leurs interactions. Nous présentons ensuite le protocole de FLO. Parmi les différentes possibilités de présentation, nous avons choisi de présenter le MOP au travers de la *vie* d'une dépendance. Nous abordons donc la définition, la déclaration et le maintien de la cohérence d'une dépendance. Pour compléter ce choix, nous abordons les fonctionnalités du MOP liées à l'introspection. Pour finir, nous illustrons son utilisation par quelques extensions. Dans ce chapitre, nous nous concentrons sur les dépendances, le chapitre suivant s'intéressera en particulier aux contrôleurs.

8.1 Architecture de FLO

Le fait d'intégrer notre modèle de dépendances dans un langage à classes en utilisant un langage comme CLOS, ObjVlisp ou NeoClasstalk nous amène à choisir la même solution que celle choisie par ces langages pour proposer un langage extensible (voir au chapitre 7) : nous définissons tout d'abord les méta-objets sur lesquels notre langage et son MOP sont bâtis. A la manière du MOP de CLOS, les méta-objets sont des classes à partir desquelles les comportements initiaux sont spécifiés et peuvent être spécialisés. Ces méta-objets définissent l'architecture de notre langage et son MOP. Celle-ci est fortement influencée par le choix de représentation des dépendances que nous précisons maintenant.

8.1.1 Une dépendance implémentée par une classe

Une dépendance représente une abstraction d'un comportement collectif qui peut se répéter entre différents groupes d'objets. Une dépendance effective représente une instanciation de ce comportement pour un groupe d'objets donnés.

La dépendance *exclusion-mutuelle* représente un même comportement collectif qui est défini entre plusieurs groupes de boutons.

Nous représentons une dépendance par une classe instanciable de multiple fois entre différents objets. Une variable de dépendance est alors une variable d'instance et les méthodes propres de la dépendance sont les méthodes associées à la classe.

La dépendance *exclusion-mutuelle* est une classe qui possède en particulier une variable d'instance `active?` correspondant à la variable de la dépendance. Des instances de cette classe représentent des dépendances effectives.

Graphe d'instanciation. Une dépendance possède des caractéristiques qui lui sont propres (comportement dynamique, variables, accès...). Chaque classe représentant une dépendance possède alors des valeurs spécifiques. Dans un langage à méta-classes, ce raisonnement nous amène à considérer les classes représentant les dépendances comme instances d'une méta-classe spécifique que nous nommons **`Meta-Link`**. En effet, le comportement dynamique,l'héritage entre dépendances et les accès sont des données de dépendances donc de classes. Ce choix nous conduit donc à l'architecture à trois niveaux illustrée par la figure 8.1.

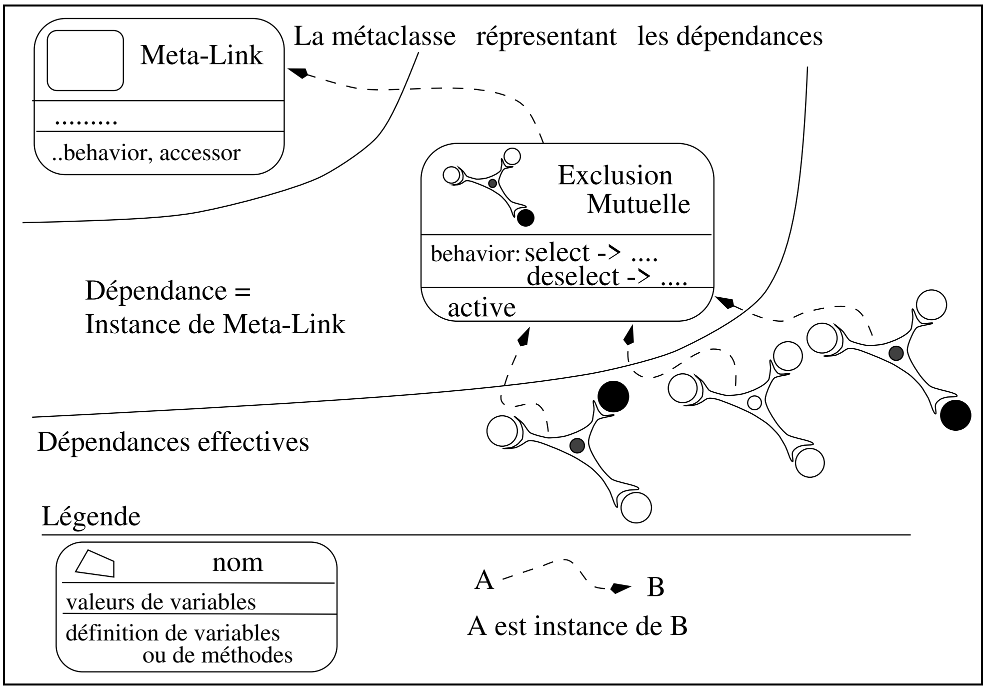


Figure 8.1: Une architecture à trois niveaux : des dépendances effectives instances (de classes) de dépendances, elles-mêmes instances de méta-classes.

Dans la figure 8.1, dans la méta-classe **`Meta-Link`** nous avons fait apparaître une variable **`behavior`** parmi les variables d'instances décrivant les classes. Cette variable représente le comportement dynamique de chaque dépendance. La valeur de cette variable pour la classe représentant la dépendance *exclusion-mutuelle* est `select implies... deselect implies ....` Les variables d'instance des dépendances sont représentées par des variables d'instances de classes. La variable `active?` définie sur *exclusion-mutuelle* est une variable d'instance.

8.1.2 Trois nouveaux méta-objets

La figure 8.2 présente le noyau minimal de FLO. On retrouve les classes `Object` et `Class` du noyau de ObjVlisp ou de CLOS. Nous nous sommes limités aux classes que nous utilisons ainsi les méta-objets responsables des méthodes, fonctions génériques... ne font pas directement parties de notre MOP. L'architecture de FLO définit trois nouveaux méta-objets : `Meta-Link`, `Link` et `Controller` sur lesquels repose l'implémentation de FLO et de son MOP :

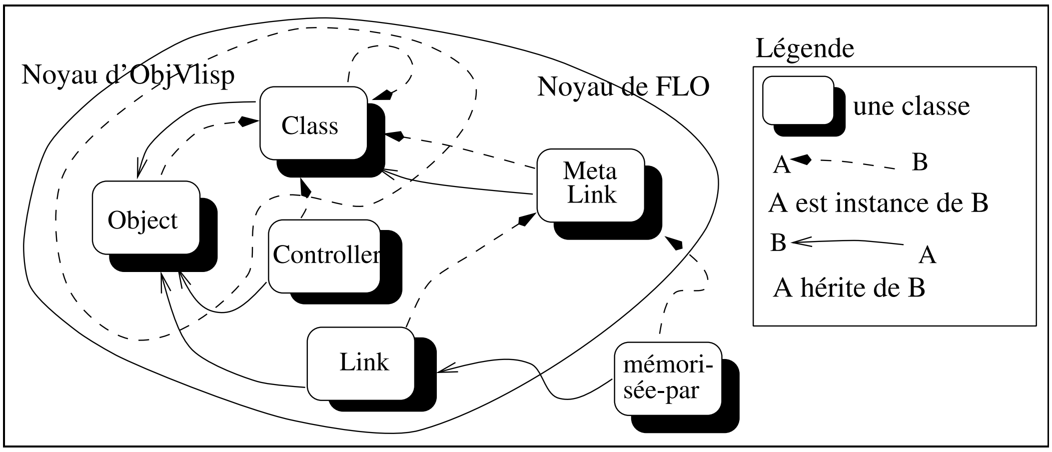


Figure 8.2: Noyau minimal de FLO

`Meta-Link` : elle est la classe de toutes les classes représentant les dépendances. C'est la racine du graphe d'instanciation des dépendances.

`Link` : une dépendance effective peut ajouter, retirer de nouveaux participants... Les fonctionnalités communes à toutes les dépendances que nous abordons plus en détail dans la section suivante sont définies dans la classe `Link`. Ainsi lorsqu'une dépendance est définie, la classe la représentant hérite directement ou indirectement de la classe `Link`.

`Controller` : elle définit le comportement commun à tous les contrôleurs. Nous étudions ces fonctionnalités de manière détaillée dans le prochain chapitre.

Les classes `Link` et `Controller` sont des classes abstraites.

8.2 De la définition au comportement dynamique

Après avoir présenté, une description structurelle de FLO, nous abordons ici une description comportementale de la gestion des dépendances au travers des différentes étapes que sont la définition, la déclaration et le maintien de la cohérence d'une dépendance. Ce faisant, nous définissons les méthodes qui constituent le MOP de FLO.

8.2.1 Définition d'une dépendance

Lors de la définition d'une dépendance, une classe la représentant est créée. Cette classe est une instance indirecte de la méta-classe `Meta-Link`. Nous n'avons pas énoncé encore assez d'informations pour présenter le mécanisme complet mis en œuvre, nous reviendrons en détail sur ce mécanisme en 8.2.4. Cependant, nous montrons maintenant comment la méta-classe `Meta-Link` est responsable de la structure des dépendances et de l'héritage entre dépendances.

Gestion de l'héritage. La classe `Meta-Link` définit l'héritage entre dépendances qui a lieu lors de la définition d'une dépendance. Ainsi la méthode `initialize` de la classe `Meta-Link` invoque ses méthodes `rename-active-and-inherit-access` et `inherit-behavior`.

La méthode `rename-active-and-inherit-access` réalise à la fois le renommage des variables actives et l'héritage des accesseurs. La méthode `inherit-behavior` s'occupe plus particulièrement de l'héritage des règles d'interactions (voir en 8.2.4). Ces comportements de la classe `Meta-Link` permettent de définir différentes approches de l'héritage entre dépendances.

La hiérarchie des appels pour la définition d'une dépendance est la suivante :

```
initialize ⟨Meta − Link⟩
    inherit-behavior ⟨Meta − Link⟩
        rename-active-and-inherit-access ⟨Meta − Link⟩
```

L'indentation représente le fait que la méthode la plus à gauche invoque la méthode indentée.

- (INHERIT-BEHAVIOR $\langle meta - link \rangle$) : cette méthode gère l'héritage du comportement dynamique. Elle invoque tout d'abord la méthode `rename-active-and-inherit-access`. Puis pour chaque type d'opérateur du comportement dynamique, elle invoque les méthodes associées `inherit-implies`, `inherit-permitted-if`, `inherit-`*operator* définies sur les sous-classes de `Meta-Link` (voir en 8.2.4).
- (RENAME-ACTIVE-AND-INHERIT-ACCESS $\langle meta - link \rangle$) : cette méthode gère le renommage des variables actives ainsi que la définition des accesseurs. Elle utilise la clause de renommage spécifiée lors de la définition de la dépendance. Cette méthode gère les conflits dus au renommage et crée lorsque cela est nécessaire les accès.

8.2.2 Déclaration d'une dépendance

Une fois une dépendance définie, celle-ci est déclarée entre différents objets. Le modèle objet que nous utilisons définit la méthode `make` comme le moyen de spécialiser la création d'instances. Cette méthode, spécialisée sur la classe `Meta-Link` ou ses sous-classes, est responsable de la création et de la *mise en service* la dépendance effective.

Initialisation. La création de dépendances effectives revêt un caractère particulier lié aux informations supplémentaires nécessaires pour gérer les variables actives. En effet, la possibilité d'exprimer plusieurs règles d'interactions ayant pour une même méthode des receveurs différents nécessite de nouvelles informations. Ainsi étant donné un sélecteur et un objet, nous devons être capables de choisir la règle d'interaction correspondante. De plus, les variables définissant des groupes d'objets doivent être gérées.

Soit la dépendance *UneDep* déclarée entre les objets `o1` et `o2`. Lorsque l'objet `o1` reçoit le message `m1`, il faut être en mesure de choisir la première règle d'interaction. Il faut savoir que l'objet `o1` est lié à la variable active `:x`.

```
(deflink UneDep (:x :y)
   :behavior (((m1 :x ...) implies B)
              ((m1 :y ...) implies C)))

(make UneDep :x o1 :y o2)
```

La méthode `initialize` de la classe `Link` est spécialisée : une table est définie pour chaque dépendance effective dans laquelle chaque participant est lié à la variable active qui le représente et inversement. Cette représentation nous permet de gérer les dépendances n-aires. Cette solution est justifiée par le fait que la solution qui consiste à ajouter des variables d'instance à la classe représentant la dépendance n'offre pas assez d'informations et n'éviterait pas la présence d'une table. Pour nous, les variables actives ne représentent pas des données structurelles de la dépendance car une dépendance ne nécessite pas obligatoirement une relation de composition entre ses participants.

Une fois l'initialisation effectuée, trois étapes se succèdent : une de vérification, une d'initialisation des participants et finalement de déclaration à proprement parler. La hiérarchie d'appels pour la déclaration d'une dépendance est la suivante :

make $\langle Meta - Link \rangle$
    check-and-error $\langle Link \rangle$
    action-before-effective $\langle Link \rangle$
    establish-link $\langle Link \rangle$
        register-link $\langle Controller \rangle$

Vérification et initialisation. La méthode `check-and-error` définie sur la classe `Link` s'assure de la cohérence entre les contrôleurs et les opérateurs définis, nous expliquons ce point en détail en 9.4. Ensuite, la méthode `action-before-effective` définie sur la classe `Link` est invoquée. Par défaut, cette méthode ne fait rien. Son but est d'offrir un moyen d'exprimer une action sur les objets participants avant que la dépendance effective ne soit mise en place. Par contre, la création de la dépendance a déjà eu lieu donc il est possible d'utiliser les variables actives pour faire références aux participants.

Déclaration. Au delà de la création physique d'une dépendance effective à l'aide de la décomposition traditionnelle création(x) = initialisation(allocation(x)), il est nécessaire de pouvoir distinguer clairement le moment exact au cours duquel elle est prise en compte par les contrôleurs ; c'est-à-dire le moment exact à partir duquel le comportement des objets est modifié du fait de cette dépendance effective.

Cette phase est réalisée par la méthode `establish-link` (de la classe `Link`) qui invoque la méthode `register-link` (définie sur la classe `Controller`) sur chaque contrôleur des objets participants (voir en 9.3.1). Une fois cette méthode invoquée, la dépendance existe réellement et est prise en compte lors au contrôle du comportement des objets.

Destruction d'une dépendance effective. Une dépendance effective peut être détruite ; c'est-à-dire qu'elle n'influence plus les objets participants. La méthode `remove-link` (définie sur la classe `Link`) est invoquée pour une destruction définitive[1] de la dépendance. Pour cela, cette méthode invoque la méthode `unregister-link` (définie sur la classe `Controller`) sur les contrôleurs des objets participants.

remove-link $\langle Link \rangle$
    unregister-link $\langle Controller \rangle$

Gestion de groupe. L'ajout ou le retrait dynamique d'objets dans une dépendance sont réalisés par les méthodes `add-object-in-link` et `remove-object-in-link` définies sur la classe `Link`. Ces méthodes vérifient que les dépendances offrent la possibilité de gérer les groupes ; c'est-à-dire si la dépendance possède une variable active référençant une liste d'objets. Elles appellent respectivement les méthodes `register-link` et `unregister-link`.

L'exemple suivant ajoute dynamiquement un bouton au groupe de boutons participant à la dépendance d'exclusion.

```
(define b1 (make bouton))
(define b2 (make bouton))
(define b3 (make bouton))
(define gp3 (make exclusion-mutuelle-reactive :boutons (list b1 b2 b3)))
(define b4 (make bouton))
(group gp3) -> (b1 b2 b3)
(add-object-in-link gp3 b4 :boutons)
(group gp3) ->  (b1 b2 b3 b4)
```

[1] Une dépendance peut être temporairement inhibée (voir en 8.3.1).

En résumé. Ces méthodes peuvent être particularisées afin de proposer des comportements différents de ceux proposés par défaut. Les fonctions `register-link` et `unregister-link` sont définies au chapitre suivant. Les fonctions spécialisables suivantes sont définies sur la classe `Link` :

- (ADD-OBJECT-IN-LINK ⟨*link*⟩ ⟨*object*⟩ ⟨*keyword*⟩) : cette méthode ajoute l'objet *object* à la dépendance *link* en lui associant la variable de comportement dynamique *keyword* qui doit être définie pour une liste d'objets. Cette méthode appelle la méthode `register-link`.
- (CHECK-AND-ERROR ⟨*link*⟩) : cette méthode vérifie la cohérence entre le comportement dynamique de la dépendance et les contrôleurs associés aux objets. Son comportement est détaillé en 9.3.1.
- (ACTION-BEFORE-EFFECTIVE ⟨*link*⟩ ⟨*initargs*⟩) : cette méthode permet de spécifier une action à réaliser sur les objets participant à la dépendance avant que le comportement effectif de celle-ci n'influe sur le comportement de ces objets. Cette méthode est invoquée après que les informations relatives aux participants ait été traitées. L'utilisateur peut donc faire référence aux participants en utilisant les variables actives qui les représentent.
- (ESTABLISH-LINK ⟨*link*⟩) : cette méthode déclare la dépendance auprès des contrôleurs. Elle invoque la méthode `register-link` de chaque contrôleur des objets participants.
- (REMOVE-LINK ⟨*link*⟩) : cette méthode détruit la dépendance *link*. Pour cela, elle fait appel à la méthode `unregister-link` du contrôleur de chaque objet impliqué dans la dépendance.
- (REMOVE-OBJECT-IN-LINK ⟨*link*⟩ ⟨*object*⟩ ⟨*keyword*⟩) : cette méthode retire l'objet *object* associé à la variable de comportement dynamique *keyword* de la dépendance *link*. Cette méthode invoque la méthode `unregister-link`.

Réflexions sur la vérification de la validité des dépendances.

Le comportement dynamique d'une dépendance est défini en terme de méthodes. Cependant, ces méthodes peuvent ne pas correspondre aux interfaces des objets liés. Par analogie à la philosophie des langages hôtes comme CLOS et SMALLTALK qui ne testent pas lors de la définition de méthodes si les méthodes utilisées existent ; FLO ne vérifie pas que les méthodes contrôlées ou invoquées existent.

Des propositions non implémentées. Nous avons réfléchi à une éventuelle vérification[2] de la validité des dépendances. La mise en place d'une telle vérification doit distinguer deux phases distinctes en vérifiant : (1) que les messages perturbants sont bien définis pour les classes des objets liés et (2) que les actions (messages compensatoires...) existent sur les objets.

La première vérification ne pose pas de problèmes. Par contre, s'assurer que les méthodes appelées en réaction (lors des messages compensatoires ou des gardes) existent bien est plus délicate. En effet dans le cas d'une définition locale de méthodes, cette vérification s'apparente à détecter dynamiquement le type des objets.

Cette vérification devrait être particularisable afin de permettre la définition d'opérateur de délégation (qui présupposent alors que les méthodes contrôlées n'existent pas voir en 9.5.1). Pour les méthodes contrôlées (cas 1), une méthode nommons-la `interface-coherent`, devrait être invoquée avant de créer une dépendance effective. Cette méthode rendrait la valeur vraie (signifiant que la dépendance peut être créée) si tous les objets définissent les méthodes, sinon elle rendrait la liste des règles d'interactions dont les méthodes ne sont pas définies. Cette solution permet de spécialiser aisément cette vérification afin de permettre la définition d'opérateurs de définition locale de méthodes ou de délégation (comme l'opérateur `corresponds`). Pour le test des méthodes invoquées par les règles d'interactions (cas 2), la solution est moins immédiate car les méthodes non définies peuvent être des méthodes définies localement par d'autres dépendances.

[2]Dans FLO, on ne spécifie pas les classes des variables actives lors de la définition d'une dépendance, donc la vérification doit avoir lieu lors de la déclaration de la dépendance.

8.2.3 Maintien de cohérence

Structure et méta-classes. Dans le modèle objet que nous utilisons, la structure des classes est régie par les méta-classes. Donc dans FLO, la structure d'une dépendance est régie par sa méta-classe au travers de deux méthodes **`find-action`** et **`give-args`** définies sur la classe **`Meta-Link`**. **`find-action`** rend la règle d'interaction associée au message perturbant. **`give-args`** rend les arguments effectifs sur lesquels l'action doit porter. Le filtrage présenté en 5.2.1 est assuré par cette méthode.

- (FIND-ACTION $\langle meta-link \rangle$ $\langle selector \rangle$ $\langle receiver \rangle$ $\langle operator \rangle$): cette méthode rend l'action associée au sélecteur *selector* pour l'objet *receiver* et pour l'opérateur *operator*.

  Avec la dépendance *mémorisée-par* donnée en 5.1.2, `(find-action` *`mémorisée-par`* `pop p1 implies)`
  rend l'interaction `(pop :stack) implies (store :memory :result)`

- (GIVE-ARGS $\langle meta-link \rangle$ $\langle link \rangle$ $\langle sign-cont \rangle$ $\langle list-of-args \rangle$ $\langle sign-act \rangle$ & OPTIONAL $\langle opt \rangle$) : cette méthode réalise le filtrage entre les listes *sign-cont* représentant la signature de la méthode contrôlée, la liste *list-of-args* des arguments effectifs de l'appel *list-of-args*. La liste *sign-act* décrit les arguments d'appels rendus. Le paramètre optionnel *opt* sert à passer des arguments supplémentaires comme le résultat de l'application pour l'opérateur **`implies`**.

  `(give-args` *`mémorisée-par`* `p1-mp-m1 (:stack) (p1) (:memory :result) 12)`
  `rend (m1 12)`

La particularisation de ces deux méthodes nous permet de modifier les choix de représentation des dépendances. Nous avons ainsi tester différentes représentations d'une dépendance pour optimiser la place lorsqu'un grand nombre de dépendances sont créées, soit la vitesse d'accès aux informations.

Maintien et opérateurs. Le maintien de la cohérence d'une dépendance nécessite, lors de l'envoi de messages perturbants sur les objets participants, différentes opérations : évaluer une expression, envoyer un message compensatoire, déléguer un message...voir en 5.2.1.

La nature même de ces différentes opérations, qui sont invoquées par le contrôleur responsable de l'objet, dépend de l'opérateur de la règle d'interaction définie pour le message perturbant.

Ainsi dans la dépendance *exclusion-mutuelle-reactive*, lorsqu'un bouton reçoit le message perturbant `select`, l'opérateur `implies-before` à la ligne 8 précise la nature de l'*action*: ici un message compensatoire doit avoir lieu avant l'exécution de la méthode contrôlée.

```
1  (deflink  exclusion-mutuelle-reactive  (:buttons)
    ...
5   :behavior
6     (((deselect :buttons-receiver) implies (set! active? link #f))
7      ((select :buttons-receiver)
8       implies-before (when (active? link) (deselect (selected link))))
       ...))
```

L'interprétation de chaque opérateur est spécifiée par une méthode que le contrôleur à en charge d'invoquer. Cependant, le fait de permettre la définition de nouveaux opérateurs nous a poussé à définir la solution suivante (voir en 9.3.1 et en 8.2.4) : au niveau des classes **`Meta-Link`** et **`Link`** nous n'avons défini aucun comportement directement lié à un opérateur spécifique. Ce n'est qu'au niveau des sous-classes de ces classes que la gestion réelle est définie.

Ainsi le déclenchement des messages compensatoires de l'opérateur de causalité **`implies`** est assuré par la méthode **`firing`** définie sur la classe **`Reactive-Link`** sous-classe de la classe **`Link`**. L'évaluation d'une garde définie par l'opérateur **`permitted-if`** est assurée par la méthode **`verify-guard`** définie sur la classe **`Guarded-Link`**.

- (FIRING $\langle reactive - link\rangle$ $\langle selector\rangle$ $\langle args\rangle$ $\langle res\rangle$) : cette méthode réalise le déclenchement des messages compensatoires. Par défaut, cette méthode envoie le message compensatoire[3] de la dépendance associée au sélecteur *selector* avec les arguments d'appels du message déclenchant *args*. *res* représente le résultat de l'application de la méthode contrôlée.
- (VERIFY-GUARD $\langle guarded - link\rangle$ $\langle selector\rangle$ $\langle args\rangle$ & OPTIONAL) : cette méthode évalue des gardes. Par défaut, cette méthode évalue la garde de la dépendance associée au sélecteur *selector* avec les arguments d'appel du message déclenchant *args*.

8.2.4 Opérateurs et architecture

La prise en compte de nouveaux opérateurs (lors de l'héritage) et la distinction d'interprétation de chacun des opérateurs amènent à enrichir l'architecture présentée au début de ce chapitre.

Un sous-protocole pour l'héritage des opérateurs. Chaque opérateur définit un héritage qui lui est propre. Pour gérer cette situation ainsi que l'ajout de nouveaux opérateurs, nous proposons la solution suivante : la méthode `inherit-behavior` définie sur la classe `Meta-Link` appelle pour chaque type d'opérateur du comportement dynamique une méthode pour réaliser un héritage différencié. Par convention, le nom de ces méthodes est la concaténation de «`inherit-`» et de l'opérateur traité. L'ordre d'appel de ces méthodes n'est pas spécifié.

```
inherit-behavior ⟨Meta − Link⟩
    rename-active-and-inherit-access ⟨Meta − Link⟩
    inherit-permitted-if ⟨Guarded − Meta − Link⟩
    inherit-implies⟨Reactive − Meta − Link⟩
    inherit-...
```

Par défaut, aucune de ces méthodes n'est définie sur la classe `Meta-Link`. Ces comportements sont définis sur des sous-classes de la classe `Meta-Link`. Ainsi la méthode `inherit-implies` définie sur la classe `Reactive-Meta-Link` est invoquée pour gérer l'héritage lié à l'opérateur `implies`.

Architecture complète. La gestion de l'héritage et l'interprétation du comportement dynamique (voir en 8.2.3) des opérateurs nous ont conduit à introduire de nouvelles classes. Nous obtenons l'architecture illustrée par la figure 8.3.

La classe `Reactive-Meta-Link` spécifie l'héritage lié à l'opérateur `implies`. La classe `Reactive -Link` instance de la classe `Reactive-Meta-Link` définie la méthode `firing` pour l'interprétation de l'opérateur `implies`. La classe `Guarded-Reactive-Meta-Link` combine la gestion des opérateurs `implies` et `permitted-if`. La classe `Guarded-Reactive-Link` assure l'interprétation de ces deux opérateurs.

Où l'on apporte des précisions sur la définition. Les traitements (héritage et interprétation) des opérateurs imposent donc des contraintes par rapport à la nature des dépendances. Pour chaque type d'opérateurs du comportement dynamique, (a) la classe de la dépendance doit gérer l'héritage (donc la bonne sous-classe de `Meta-Link` doit être choisie) et (b) la dépendance doit hériter de classes de dépendances sous-classes de la classe `Link` gérant l'interprétation des opérateurs.

Ces choix ou vérifications sont effectués par la macro `deflink` qui facilite la définition de dépendances. L'utilisateur peut préciser explicitement la classe de la dépendance et les classes dont elle hérite. Le mot clé `:is` permet de spécifier explicitement la méta-classe dont la dépendance est

[3] Du fait de l'héritage entre dépendances, le terme message compensatoire d'une dépendance fait référence à tous les messages compensatoires hérités ou définis localement. De même, la garde fait référence aux gardes héritées et définies localement.

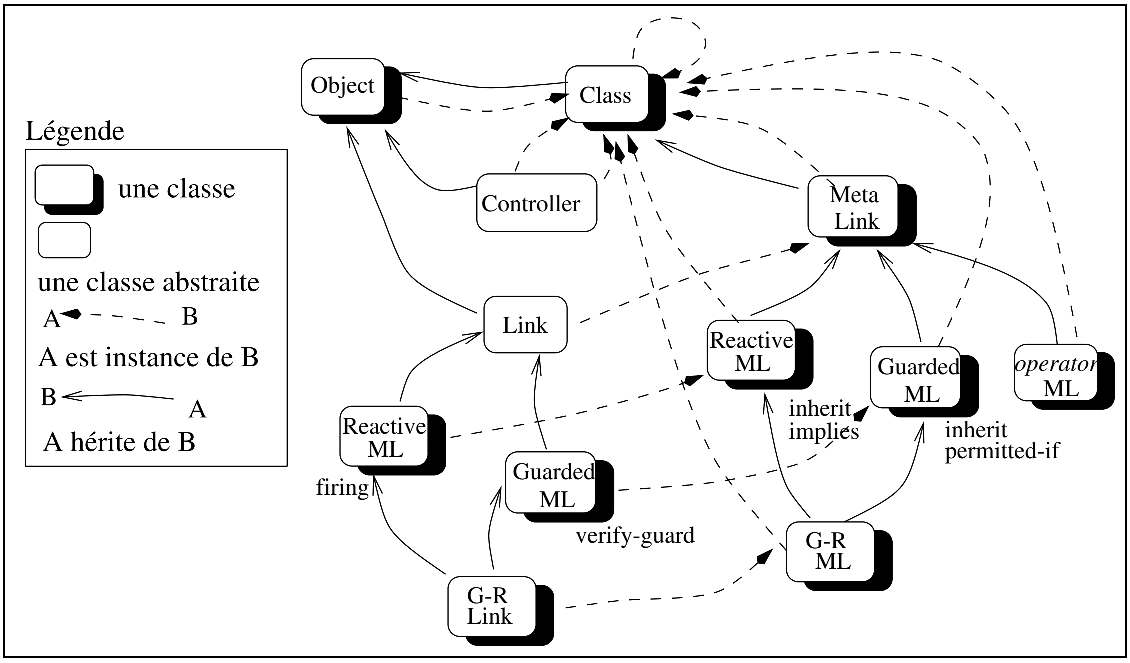


Figure 8.3: Conséquence de la gestion des opérateurs sur l'architecture de FLO au niveau des dépendances.

instance. Le mot clé `:inherit` permet de spécifier les classes dont la dépendance hérite. Lorsque de telles informations sont précisées le système s'assure que ces informations sont cohérentes avec celles déduites automatiquement de la présence des opérateurs ; c'est-à-dire que la classe spécifiée est une sous-classe des classes gérant les opérateurs de la dépendance.

```
(deflink mémorisée-par (:stack :memory)
  :behavior
   (((pop :stack) implies (store :memory result))
    ((pop :stack) permitted-if (not-null? :memory))))
```

Ainsi la définition de la dépendance *mémorisée-par* est *expansée* pour donner le code suivant[4] en STklos.

```
(define-class mémorisée-par (Guarded-Reactive-Link)
   ;; elle hérite de Guarded-Reactive-Link
  ()           ;; il n'y a pas de variables d'instances
  :metaclass Guarded-Reactive-Meta-Link
    ;; elle est instance de Guarded-Reactive-Meta-Link
:behavior ....)
```

Liaison classes / opérateurs. Il se pose alors la question de savoir comment le système associe un opérateur aux classes le traitant correctement. Notre solution est basée sur une réification des opérateurs. Ainsi la définition de l'opérateur `implies` se fait comme suit :

```
(define implies (make operator :name 'implies :link-class Reactive-Link
                               :link-meta-class Reactive-Meta-Link))
```

Comme nous le montrons au chapitre suivant, l'opérateur réifié n'intervient pas en tant qu'objet

[4] Le résultat de cette transformation dépend du langage hôte, ainsi en ObjVlisp il s'agira d'un message de création explicite à la méta-classe `Meta-Link`.

dans le maintien de la cohérence. Seul suffit l'utilisation du symbole associé, c'est pourquoi nous avons choisi d'associer l'opérateur à un symbole (une variable globale en SCHEME). La réification des opérateurs permet essentiellement de regrouper toutes les informations structurelles impliquées par le traitement des opérateurs : classe de dépendances définissant l'interprétation et méta-classe de dépendances gérant l'héritage.

8.2.5 Protocoles d'introspection

Le protocole doit offrir la possibilité d'accéder aux dépendances de la même manière qu'il est possible de le faire sur la structure des classes. Des outils d'aide à la programmation peuvent alors être mis en place. Nous présentons les méthodes qui définissent un protocole d'introspection pour les dépendances.

Sur `Meta-Link`. Toutes les dépendances doivent pouvoir être décrites ou visualisées. Il doit être possible d'accéder aux composants d'une dépendance. Ainsi les méthodes suivantes sont définies :

- (GIVE-BEHAVIOR $\langle meta-link \rangle$) : cette méthode renvoie une copie du comportement dynamique qui n'est pas modifiable directement.
- (GIVE-INHERITANCE $\langle meta-link \rangle$) : cette méthode rend la clause d'héritage de la dépendance.
- (GIVE-ACCESSORS $\langle meta-link \rangle$) : cette méthode rend la liste de tous les accesseurs définis pour la dépendance.

Sur `Link`. La classe `link` définit principalement trois méthodes de base d'accès à partir desquelles toutes les autres peuvent être définies.

- (GIVE $\langle link \rangle$ $\langle active \rangle$) : cette méthode permet d'accéder aux participants d'une dépendance étant donnée une variable active. Elle est utilisée pour la création des accesseurs.

  `(give p1-mp-p1:stack)` rend `p1`

- (GIVE-ALL-OBJECTS $\langle link \rangle$) : cette méthode permet d'accéder à tous les participants d'une dépendance. Cette méthode rend une liste dont chaque élément est un objet participant et la variable à laquelle il est associé.

  `(give-all-objects p1-mp-p1)` rend `((p1:stack) (m1:memory))`

- (OPERATORS $\langle link \rangle$) : cette méthode rend la liste des opérateurs de la dépendance.

8.3 Quelques extensions du protocole

Nous illustrons maintenant l'utilisation du protocole par quelques exemples. Nous précisons les besoins rencontrés et comment la spécialisation d'un aspect du protocole apporte une réponse.

8.3.1 Déclenchement conditionnel

Besoin. Le déclenchement des réactions s'avére parfois temporairement inutile [WIDO 96]. Par exemple, il est inutile de réafficher une fenêtre lorsqu'elle n'est pas visible. De plus, un déclenchement conditionnel peut être souhaité afin d'éviter des répétitions. Dans certains cas, la condition n'est pas liée à la dépendance mais à une donnée globale ou temporaire, il n'est donc pas possible de définir cette condition dans la réaction.

Solution. Le protocole initial prévoit simplement d'appeler systématiquement la méthode `firing` lorsqu'un message perturbant associé à l'opérateur `implies` pour cette dépendance a été reçu par un des objets contrôlés par celle-ci.

Ce déclenchement systématique est facilement soumis à condition. Pour cela, il suffit de définir une classe `Conditional-Firing-Link` dont la méthode `firing` ne déclenche le message compensatoire que si la condition est vérifiée. Nous avons choisi de forcer le programmeur à redéfinir la méthode `condition?` : par défaut, la méthode `condition?` rend la valeur `#f`.

```
(define-class Conditional-Firing-Link (Reactive-Link))

(define-method condition? ((l Conditional-Firing-Link))
   #f)
(define-method firing ((l Conditional-Firing-Link) selector args . opt)
   (when (condition? l)
         (next-method)))
```

Exemple. Prenons l'exemple d'un réaffichage inutile dans une fenêtre cachée. Il suffit de définir une méthode `condition?` sur la dépendance responsable de la cohérence de la fenêtre, nommons-la *affichage-cohérent* assurant le réaffichage.

```
(define-method condition? ((l affichage-cohérent))
    (visible? (give l :fenêtre)))
```

Variante : l'inhibition temporaire. Les conditions du déclenchement peuvent être associées à la dépendance par le biais de variables d'instance. Définissons une classe `Possible-Inhibited-Link` héritant de la classe `Conditional-Firing-Link` qui définit une variable d'instance `inhibited?`.

```
(define-class Possible-Inhibited-Link (Conditional-Firing-Link)
  ((inhibited? :initform #f :init-keyword :inhibited?
               :accessor inhibited?)))
(define-method condition? ((l Possible-Inhibited-Link))
   (not (inhibited? l)))
(define-method inhibit ((l Possible-Inhibited-Link))
  (set! inhibited? l #t))
(define-method activ ((l Possible-Inhibited-Link))
  (set! inhibited? l #f))
```

Distinction par rapport aux gardes. Le déclenchement conditionnel est différent de la notion de garde. Une garde peut interdire l'application d'une méthode à la manière des méthodes auxiliaires `:around` de CLOS [Keen 89]. Un déclenchement conditionnel n'interdit pas l'application d'une méthode mais contraint les messages compensatoires dus à cette application.

8.3.2 Une première détection de cycles

Besoin. Il est fréquent que des dépendances définissent des situations dans lesquelles les propagations mènent à des cycles (voir en 6.2).

Solution. Par défaut, le système ne propose pas de gestion des cycles. Une première solution pour le contrôle dynamique des cycles est basée sur le contrôle du déclenchement des dépendances. Nous avons montré au chapitre 6 les limites d'une telle solution quand elle se limite au niveau des dépendances et ne prend pas en compte l'aspect global du graphe de dépendances. Cependant, cette solution a fait ses preuves et est utilisable avec la connaissance de ses limites.

La solution consiste à définir une dépendance `No-Reenter-Link` sous-classe de `Reactive-Link` ayant une variable d'instance permettant de savoir si au cours d'une propagation elle a déjà été déclenchée et de modifier la méthode responsable du déclenchement `firing`.

```
(define-class No-Reenter-Link (Reactive-Link)
  ((state:initform 'ok)))

(define-method firing ((lk No-Reenter-Link) selector args . opt)
  (if (equal? (slot-ref lk 'state) 'ok)
      (begin
        (slot-set! lk 'state 'occuped)
        (next-method)
        (slot-set! lk 'state 'ok))
      'nothing))
```

Remarques. Plusieurs variantes peuvent être proposées suivant la granularité du test de redéclenchement de la dépendance. Ainsi pour les dépendances possédant de nombreuses règles d'interaction, il possible de tester le redéclenchement au niveau de la règle et non de la dépendance (voir en 6.2).

8.3.3 Déclenchement si nécessaire

Certaines dépendances ont un caractère symétrique qui pousse naturellement le programmeur à énoncer des dépendances menant à des cycles (voir en 6.2). La maintien purement réactif d'une dépendance n'est pas adapté. Il devient nécessaire de proposer au programmeur la possibilité d'énoncer les conditions de cohérence de la dépendance afin de ne propager les messages compensatoires que si nécessaire.

Solution. Il s'agit là d'un cas de déclenchement conditionnel. Nous définissons donc une nouvelle dépendance `Link-With-Coherence` sous-classe de `Conditional-Firing-Link` et spécialisons la méthode `establish-link` pour spécifier qu'il n'est pas possible de créer des dépendances incohérentes.

Le sous-protocole est basé sur trois nouvelles méthodes : `coherent?`, `do-coherence` et `coherence-error`. Comme le montre la méthode `establish-link`, la sûreté de la dépendance est assurée : lorsque la dépendance n'est pas cohérente (`coherent?`), la méthode `do-coherence` est invoquée. Si après cela, la dépendance n'est toujours pas cohérente alors la méthode `coherence-error` est invoquée. Elle détruit la dépendance et provoque une erreur.

```
(define-method establish-link ((l Link-With-Coherence))
    (next-method)
    (unless (coherent? l)
            (do-coherence l)
            (unless (condition? l) (coherence-error))))

(define-method do-coherence ((l Link-With-Coherence)))
(define-method coherent? ((l Link-With-Coherence)) #t)
(define-method coherence-error ((l Link-With-Coherence))
   (remove-link l)
   (error "link ~a that was not coherent, is removed~%" l))
(define-method condition? ((l Link-With-Coherence))
   (not (coherent? l)))
```

Le déclenchement conditionnel assure que la dépendance n'est déclencher que lorsque cela est nécessaire ; c'est-à-dire lorsque la dépendance est incohérente.

Les méthodes spécialisables définies sur la classe `Link-With-Coherence` sont :

- (COHERENCE-ERROR ⟨*link*⟩) : cette méthode teste si *link* est cohérent. Par défaut, si ce n'est pas le cas, la dépendance est détruite en appelant `remove-link` et une erreur est signalée.
- (COHERENT? ⟨*link*⟩) : cette méthode est appelée pour savoir si une dépendance est cohérente. Par défaut, cette méthode rend la valeur `#t`.
- (DO-COHERENCE ⟨*link*⟩) : cette méthode est appelée pour assurer la cohérence d'une dépendance. Par défaut, elle ne fait rien.

Illustration. Voici à titre d'exemple, une définition possible de la dépendance *même-abscisse* qui assure que de points ont la même abscisse.

```
(deflink  aligner (:point1 :point2)
 :inherit ((Link-With-Coherence))
 :access ((point1 :point1) (point2 :point2))
 :behavior
  (((move :point1 x y)        implies  (move :point2 x (y :point2)))
   ((move :point2 x y)        implies  (move :point1 x (y :point1)))
   ((translate :point2 dx dy) implies  (translate :point1 dx dy))
   ((translate :point1 dx dy) implies  (translate :point2 dx dy))))

(define-method coherent? ((lk aligner))
   (= (x (point1 lk)) (x (point2 lk))))

(define-method do-coherence ((lk aligner))
   (move (point2 lk) (x (point1 lk)) (y (point2))))
```

8.3.4 Action à la création

Besoin. Il est parfois important de pouvoir définir une action sur les objets participants lors de la création ou de la destruction d'une dépendance. Cette action peut définir une part importante de la sémantique de la dépendance et utiliser le comportement dynamique.

> Lors de la déclaration la dépendance d'exclusion-mutuelle entre boutons, il peut s'avérer nécessaire de spécifier que l'un d'entre eux soit sélectionné.

Solution. Dans le protocole initial, la méthode `action-before-effective` est invoquée *avant* que la dépendance ne soit réellement déclarée (voir 8.2.2). Introduire la possibilité de spécifier une action après la déclaration d'une dépendance est simple. Pour cela, il suffit de définir une nouvelle dépendance `Link-With-Action` et de spécialiser la méthode `establish-link` pour que la méthode `action-after-creation` soit invoquée.

```
(define-method establish-link ((l Link-With-Action))
   (next-method)
   (action-after-creation l))
(define-method action-after-creation ((l Link-With-Action)) '())
```

De même, la méthode `action-before-removal` est appelée automatiquement par la méthode `remove-link` avant que la dépendance soit détruite. Par défaut, ces deux nouvelles méthodes ne font rien.

```
(define-method remove-link ((l Link-With-Action))
   (action-before-removal l)
   (next-method))
(define-method action-before-removal ((l Link-With-Action))  '())
```

En CLOS, ces méthodes pourraient être implémentées comme des méthodes auxiliaire `:before` et `:after` de la méthode `establish-link`.

Exemple. Ainsi si l'on veut exprimer une variante de la dépendance *exclusion-mutuelle-reactive* : la dépendance *toujours-un* qui spécifie qu'il y a *toujours* un et un seul bouton sélectionné. La méthode `action-after-creation` spécifie ici qu'à la création de la dépendance le premier bouton est sélectionné.

La variable `deselection?` indique si un des boutons peut être déselectionné (ligne 2 et 6). Ce qui par rapport à la contrainte de départ n'arrive que si un autre bouton vient d'être sélectionné. La sélection d'un bouton[5] conduit donc à indiquer que le précédent bouton peut être désélectionné (ligne 9), puis à le désélectionner (ligne 10) et à de nouveau empêcher toute autre désélection (ligne 11).

```
1  (deflink toujours-un (:boutons)
2    :var '((deselection? :initform #f  :accessor deselection?)
3             (old  :initform'() :accessor old))
4    :inherit ((link-with-action))
5    :behavior
6     (((deselect :boutons-receiver) permitted-if (deselection? link))
7      ((select :boutons-receiver) implies
8                  (unless (null? (old link))
9                          (set! deselection? link #t)
10                         (deselect (old link))
11                         (set! deselection? link #f))
12                         (set! old link :boutons-receiver))))

13  (define-method action-after-creation ((l exclusion-mutuelle-reactive))
14     (select (car (give l :boutons))))
```

Les méthodes sont définies sur la classe `Link-With-Action` :

- (ACTION-BEFORE-REMOVAL ⟨*link*⟩) : cette méthode permet de spécifier une action à effectuer avant la destruction d'une dépendance. Par défaut, elle ne fait rien.
- (ACTION-AFTER-CREATION ⟨*link*⟩) : cette méthode permet de spécifier des actions à effectuer après la création d'une dépendance. Par défaut, elle ne fait rien. Elle est appelée après la déclaration de la dépendance donc les actions spécifiées par cette méthode sont contrôlées.

8.4 Conclusion

Dans ce chapitre, nous avons présenter l'architecture du langage FLO et de son MOP. Celui-ci est basé sur la définition de trois nouveaux méta-objets sous forme de classes : `Link` qui représente le comportement des dépendances effectives, `Meta-Link` qui représente le comportement des dépendances et `Controller`. Cette dernière fait l'objet du prochain chapitre. Nous avons détaillé les interactions entre les comportements de ces méta-objets et montré quelques extensions.

8.4.1 Vision d'ensemble du MOP au niveau des dépendances

Dans ce chapitre, la présentation du MOP s'est limitée aux protocoles liés aux comportements des dépendances. La figure 8.4 en présente une vision d'ensemble. Le chapitre suivant complète ce MOP en présentant la partie dédiée aux contrôleurs.

[5] autre que celui sélectionné si l'on suit toujours la précondition choisie dans les précédents exemples définis en 5.1.2.

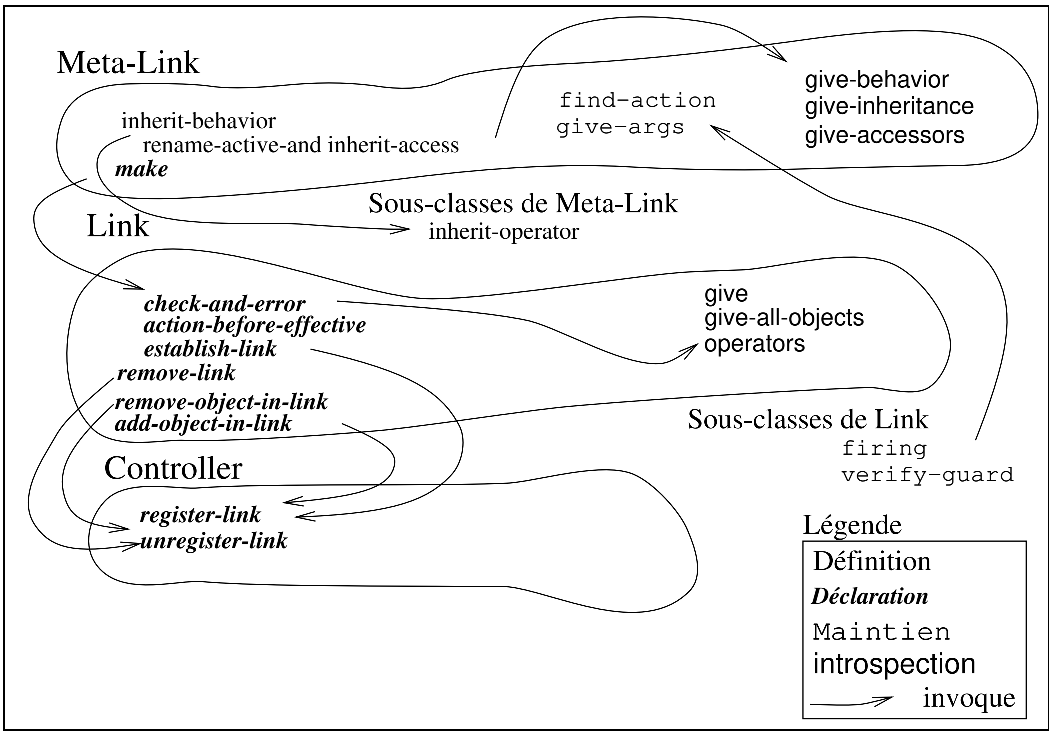


Figure 8.4: Une vue d'ensemble du MOP au niveau des dépendances.

9

Contrôleurs et extensions

« Work on metaobject protocols suggest a new view, in which abstractions expose their implementation, but do so in a way that makes a principled division between the functionality they provide and the underlying implementation [...] so that the client doesn't have to be confronted with implementation issues all the time, and, moreover, can address some implementation issues without having to address them all. » [Kicz 92a].

Ce chapitre complète la présentation de notre langage et de son protocole. Le maintien de la cohérence des dépendances est basé sur le contrôle des messages (voir en 3.1 et au chapitre 4), aussi présentons nous rapidement différents travaux sur la mise en place du contrôle de l'envoi de messages. Ensuite nous précisons quel type de contrôle nous avons choisi et nous introduisons les comportements des contrôleurs. Nous finissons en exposant comment les contrôleurs offrent un nouveau degré de spécialisation du langage.

9.1 Contrôle et abstraction de l'envoi de messages

Le contrôle de l'envoi de messages est réalisable de différentes manières. Cependant, hormis l'aspect technique d'un tel contrôle se pose la question de l'abstraction de celui-ci. L'utilisation de méta-objets offre une abstraction et une séparation claire entre le contrôle et les objets contrôlés.

9.1.1 Mécanismes

De nombreux travaux ont proposé diverses solutions pour introduire un contrôle de l'envoi de messages [Pasc 86, Maes 87b, Mura 89, Ferb 89, Pach 93, Ibra 91, Kicz 91, Chib 93a, McJa 95, Mule 95a, Riva 96a]. Nous les présentons sous formes de familles pour lesquelles nous nous sommes efforcés de faire ressortir les informations qui les caractérisent le mieux.

Sous-classes et redéfinition de méthodes. Il est possible de sous-classer la classe des instances à contrôler, de redéfinir les méthodes pour introduire un appel au contrôle des messages et de changer de classe ces instances (lorsque le langage le permet). Cette approche est choisie dans PluS [Boua 94]. Elle multiplie cependant le nombre de classes et nombre de méthodes ne servent qu'à prendre le contrôle.

Pièges à messages. KSL [IBRA 88, IBRA 91, CUMM 95] offre la possibilité de contrôler des envois de messages sur des instances particulières. Pour cela, ils offrent un mécanisme nommé *piège* qui intercepte les messages envoyés à une instance et permet d'effectuer d'autres actions : si un piège correspond au message reçu par un objet, le piège est effectué à la place en remplacement de la méthode invoquée. Le langage fournit la possibilité d'invoquer à partir d'un piège la méthode piégée grâce à un mécanisme similaire au `call-next-method` de CLOS: les méthodes `Eval` et `Next-Eval`.

Encapsulateurs et Espions. Les capsules ou encapsulateurs [PASC 86] ou les espions [PACH 93] proposent un contrôle des messages basé sur la substitution physique d'identité entre l'objet contrôlé et une capsule à l'aide de la primitive `:become` en SMALLTALK (voir figure 9.1). Ainsi les messages envoyés à l'objet sont en fait envoyés à la capsule, un objet dit *minimal*, qui ne connaissant pas le message déclenche le mécanisme de récupération d'erreurs de SMALLTALK: la méthode `doesNotUnderstand:`. Il est alors possible de contrôler le message.

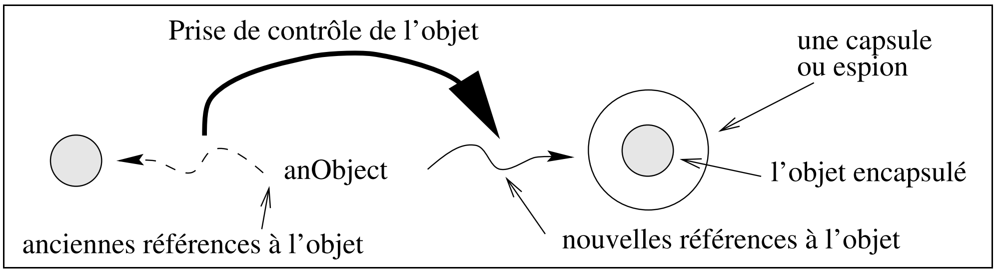


Figure 9.1: Prise de contrôle à l'aide d'une capsule ou d'un espion : substitution physique d'identité entre l'objet et la capsule.

Il existe principalement trois types de problèmes liés à cette approche : le coût du contrôle (le message contrôlé a déjà été cherché en vain), seuls les messages provenant de l'extérieur de l'objet sont contrôlables (la variable `self` n'étant pas une vraie variable) et les classes ne peuvent pas être espionnées [1]. La première version de FLO dans CLASSTALK utilisait les espions pour contrôler les messages.

Compilateurs et réflexion. F. RIVARD dans NEOCLASSTALK [RIVA 95, RIVA 96a] permet de spécialiser le compilateur de SMALLTALK afin d'enrichir ou de modifier la sémantique des méthodes : il se propose ainsi d'ajouter des pré et post-conditions à la EIFFEL ou l'envoi de messages asynchrones pour le langage ACTALK [BRIO 89b, BRIO 96]. Cette solution est efficace car elle modifie directement le fonctionnement du compilateur et offre plusieurs centres d'intervention : construction des noeuds, analyseur, arbre syntaxique, génération de code, environnement de génération. De plus, NEOCLASSTALK est basé sur le concept de changement dynamique de classes. F. RIVARD [RIVA 96c, RIVA 96b] propose différentes méta-classes permettant de contrôler différents aspects du langage dont la recherche des méthodes (méta-classe `LookupApply`), leur application (méta-classe `Apply`).

Cette approche va dans le même sens que celle de la nouvelle version de OPENC++ version 2 [CHIB 95] qui offre la possibilité de contrôler la compilation des programmes plutôt que leur exécution dynamique.

L'implémentation de FLO dans NEOCLASSTALK utilise précisément ces possibilités. Le changement dynamique de classe couplé au contrôle de l'application des méthodes permet un contrôle au niveau d'une instance. Lorsqu'une instance doit être contrôlée, elle change de classe et devient instance d'une classe sous-classe de celle-ci permettant le contrôle des messages.

---

[1] « *The natural solution would be to spy classes, and instance creation method. This is not feasible in practice, mainly because classes can't* become *non-classes objects (this is a limitation that we can forgive to Smalltalk interpreter!).* » [PACH 93]

Clos et son MOP. CLOS permet de spécialiser une méthode sur une instance particulière à l'aide des *spécialiseurs* de forme `eql`, ce qui dans une certaine mesure permet de prendre le contrôle de méthodes pour des instances particulières. Cette solution peut s'avérer coûteuse en place et en temps. En effet, le temps de parcours de la liste des méthodes lors du calcul de la méthode effective[2] augmente et pénalise l'ensemble des objets utilisant cette fonction générique.

CLOS, au travers de son MOP [Kicz 91], offre la possibilité de contrôler différents aspects de l'application des fonctions génériques : au niveau de leur application même avec la fonction apply-generic-function, de l'application des méthodes composant une fonction générique avec la fonction apply-methods ou au niveau d'une seule méthode avec la fonction apply-method.

Dans un but d'optimisation, ces fonctions génériques sont remplacées de manière effective dans le MOP par d'autres fonctions génériques [Kicz 91][page 128]. Ainsi, (apply-generic-function gf args) est équivalent à : (apply (compute-discriminating-function gf) args) Les fonctions apply-methods et apply-method sont soumises à une même transformation (voir [Kicz 91] page 130). Le net avantage de cette décomposition est la possibilité de mémoriser le résultat de la fonction générique. Ce contrôle de l'envoi de messages nous a permis de proposer une première version de FLO [Duca 93, Duca 94a]. Haarslev et Møller dans [Haar 90] utilisent le MOP de CLOS et ajoutent aux méta-classes un mécanisme de notification (démons associé aux variables d'instances, démons associés aux méthodes).

Remarque. Ces travaux, s'ils offrent la possibilité de contrôler l'envoi de messages, n'offrent pas une réelle distinction entre l'objet contrôlé et le contrôle lui-même. Pour cela, l'introduction de nouveaux objets est nécessaire. Les travaux suivants offrent à la fois un contrôle de l'envoi de messages et l'abstraction de ce contrôle.

9.1.2 Méta-objets

« *The meta-object holds information about the implementation and interpretation of the object... It is possible to create abstraction of the behavior of an object (i.e. ready-made meta-objects), and to temporarily attach such a special behavior to an object.* » [Maes 87a].

3-KRS. Les travaux de P. Maes [Maes 87b, Maes 87a, Maes 88] sur l'intégration d'une architecture réflexive à la 3-Lisp dans le langage KRS [Stee 88] constitue la première définition d'un langage à objets comportementalement réflexif : le langage 3-KRS. P. Maes introduit la notion de *méta-objet* afin de clairement distinguer les opérations réflexives des autres opérations. Ainsi à chaque objet du langage est associé un méta-objet. L'objet ne détient que les informations relatives à l'entité qu'il représente. Le méta-objet définit toutes les informations réflexives, c'est-à-dire nécessaires pour l'implémentation et l'interprétation de l'objet : comment il hérite des informations, comment l'objet s'affiche, comment de nouvelles instances peuvent être créées, comment l'objet réagit aux messages. Ainsi, il est possible de définir des abstractions de la gestion du comportement d'un objet et de les associer temporairement à certains objets.

Open C++. S. Chiba, dans la première version de OpenC++ [Chib 93a, Chib 93b], propose un MOP dans le cadre de la programmation distribuée. Celui-ci repose alors sur le contrôle de l'envoi de messages et l'abstraction de celui-ci à l'aide de méta-objets. Lorsque le comportement d'un objet doit être contrôlé (l'objet est dit *réflexif*), un méta-objet lui est associé. Ainsi lorsqu'une méthode *réflexive* est invoquée, son exécution est contrôlée par son méta-objet. La clé de voûte du contrôle des méthodes réside dans deux méthodes `Meta_MethodCall` et `Meta_HandleMethodCall` définies sur la classe `MetaObj` racine du graphe d'héritage des méta-objets. La première est invoquée lors de l'invocation d'une méthode réflexive, la seconde exécute la méthode initialement

[2] La méthode effective est l'ensemble des méthodes applicables et triées qui sont appliquées lors d'une invocation de la fonction générique.

invoquée. Le code ci-après définit un méta-objet affichant une trace à chaque invocation d'une méthode réflexive.

```
class VerboseMetaObj: public MetaObj {
public:
  void Meta_MethodCall(Id met, Id category, ArgPac& args, ArgPac& reply)
    { printf("***reflect met % s() was called.", Meta_GetMethodName(met));
         Meta_HandleMethodCall(met, args, reply); };
};
```

Le contrôle des messages de LAYOM [BOSC 95b] utilise une architecture similaire pour implémenter les Layers.

Moostrap : le protocole lookup ∘ apply. P. MULET définit dans [MULE 93a, MULE 95b, MULE 95a] un langage à prototypes réflexif. Ce langage, qui introduit de la réflexion comportementale dans un modèle à prototypes, est basé sur le protocole de décomposition réflexive de l'envoi de messages lookup ∘ apply[3] défini dans [MALE 92]. L'idée est de réifier l'envoi de messages en deux phases distinctes : tout d'abord la recherche d'un champ[4] (`lookup`) puis ensuite l'application du résultat obtenu (`apply`). En MOOSTRAP, tout objet possède un méta-objet (même les méta-objets) responsable de la recherche du champ.

CodA. Nous avons déjà présenté le protocole CODA proposé par J. MC AFFER [MCJA 95] dans chapitre 7. Nous tenons à ajouter que CODA en se concentrant en particulier sur la communication entre objets propose un protocole dont le pouvoir d'expression est beaucoup plus riche que la plupart des MOPs. Il permet de distinguer clairement chacune des étapes de la communication entre objets. Ainsi entre-t-il en jeu dès l'émission des messages et pas seulement à leur réception.

9.1.3 Des méta-objets comme contrôleurs dans un langage à classes

Alors que les travaux de P. MAES se situaient dans le contexte d'un langage ne proposant pas de distinction entre classes et instances, J. FERBER dans [FERB 89] répond à la question : «comment intégrer des méta-objets dans un langage à classes?» Il propose alors trois solutions : l'utilisation des classes comme des méta-objets, la définition des méta-objets comme instances d'une classe `Meta-Object` ou la réification de messages. Avec la première solution contrôler une seule instance n'est pas impossible, mais nécessite de créer de nouvelles classes : toutes les instances d'une classe partageant alors le même contrôle.

Le concept de méta-objet tel qu'il est défini par P. MAES et utilisé par S. CHIBA ou J. FERBER possède les caractéristiques souhaitées pour le rôle de *contrôleur* (voir en 6.1) : un contrôleur est alors un méta-objet spécialisé dans la gestion des dépendances. C'est pourquoi la deuxième solution est celle qui correspond le mieux à nos exigences :

- tout d'abord, la séparation entre un objet et la gestion des dépendances est logiquement définie. Un méta-objet contrôleur définit les informations relatives aux dépendances. Les classes sont utilisées pour la description *structurelle* (définition de la structure d'une instance et de l'ensemble des opérations applicables) et les méta-objets pour la description comportementale (comment un message est interprété, ...) [FERB 89].

- L'utilisation de méta-objets permet de ne contrôler que certaines instances et de ne pas pénaliser toutes les instances d'une classe dont une instance doit être contrôlée. En cela, une telle approche correspond aux besoins de notre modèle de dépendances.

[3] Attention, la composition lookup ∘ apply ne doit pas se lire dans l'acceptation mathématique de la composition de fonctions mais à l'envers ! Ainsi apply ∘ lookup est équivalent à (apply (lookup)) qui s'écrit en mathématiques apply ∘ lookup.

[4] MOOSTRAP offre une vision unifiée pour la représentation des méthodes et des variables d'instances sous forme de champs.

- Un méta-objet permet d'abstraire le contrôle de l'envoi de messages qu'il offre. La réification d'un méta-objet facilite la spécialisation du comportement du contrôle des messages.

Avec la troisième solution, il est impossible de particulariser le comportement d'une instance. Citons J. Ferber lui-même : *« reification of communications does not say anything on objects: it is impossible to monitor objects, to represent specific information about the receiver, or to represent the behavior of a single object »* [Ferb 89]. Il faut cependant noter que cette solution est orthogonale aux autres solutions et peut être utilisée conjointement à celles-ci.

Remarques. Notre approche diffère cependant sensiblement de celle de P. Maes ou S. Chiba. Dans FLO, un méta-objet est responsable des dépendances, c'est pourquoi nous préférons employer le terme de «contrôleur» à celui de «méta-objet». Un méta-objet contrôleur peut être associé à un groupe d'objets qui partagent alors le même contrôle des méthodes. Cette gestion d'un groupe d'objets par un seul contrôleur est rendue nécessaire par le fait qu'un contrôleur peut être utilisé pour maintenir globalement un ensemble de dépendances. Le fait d'associer un méta-objet par objet rendrait cette tâche difficile.

FLO utilise la réflexivité structurelle des langages hôtes (méta-classes) pour définir les dépendances (voir le chapitre précédent) et la réflexivité comportementale apportée par l'introduction de méta-objets.

Architecture complète de FLO. L'architecture de FLO est illustrée par la figure 9.2. Remarquons que les instances de la classe `Bouton` ne possèdent pas toutes un contrôleur : l'une d'entre-elles n'a pas de contrôleur. De plus, ce contrôleur pourrait être différent pour chacune d'entre elles. Nous reviendrons en détail sur l'association d'un contrôleur à un objet en 9.4.2.

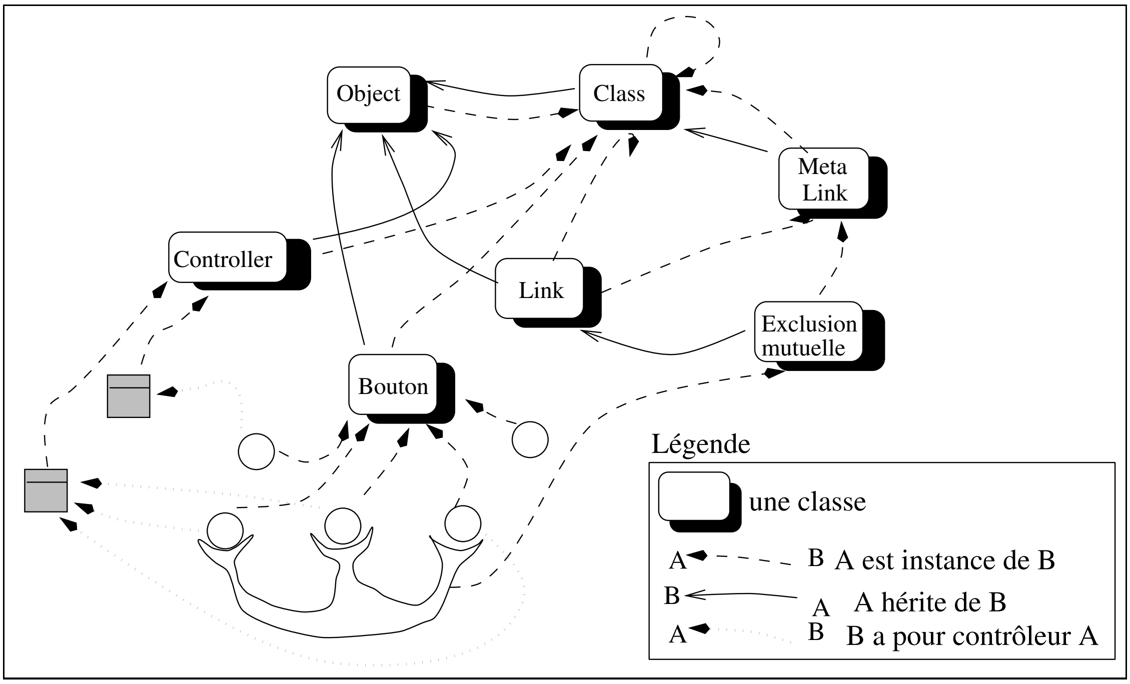


Figure 9.2: Architecture de FLO simplifiée mais montrant tous les acteurs : contrôleurs, classes, méta-dépendances, dépendances et dépendances effectives.

9.2 Modélisation de la capture de l'envoi de messages

Dans cette partie, après avoir comparé la prise du contrôle avec l'approche de J. FERBER et celle de S. CHIBA, nous explicitons les différents aspects du maintien de la cohérence des dépendances.

La seconde solution de J. FERBER pour l'intégration de méta-objets dans un modèle à classes est proche de notre solution, tout comme la solution de S. CHIBA, aussi allons-nous comparer ces approches. Notre modèle étant implémenté dans différents langages, pour cette comparaison, nous utilisons le modèle OBJVLISP et les mêmes conventions syntaxiques que celle de [FERB 89].

9.2.1 Une solution.

Pour la prise de contrôle, J. FERBER redéfinit la primitive `send` responsable de l'envoi de messages comme suit :

```
1 (define send (obj sel . args)
2    (let ((meta (meta-of obj)))
3      (if meta
4          (send meta :control
5                      (send Message :new :sender meta :receiver obj
6                                     :selector sel :arguments args))
7          (apply (default-lookup obj sel) obj args))))
```

Ainsi lorsqu'un objet possède un méta-objet, le message envoyé est réifié (lignes 5 et 6) et passé en argument lors de l'appel de la méthode `control`[5] du méta-objet. Quand un objet ne possède pas de méta-objet, le message est normalement appliqué. L'accès au méta-objet d'un objet est effectué à l'aide d'une primitive et non d'un envoi de messages pour éviter une régression infinie.

La méthode `control` est alors définie de la manière suivante :

```
8 (define-method control ((self MetaObject) msg)
9    (let* ((sel  (send msg :selector))
10          (args (send msg :args))
11          (rec  (send msg :receiver))
12          (meth (send self :lookup (class-of rec) sel)))
13     (if meth
14         (send meth :apply rec args)
15         (send rec :Doesnotunderstand sel args))))
```

Le message est alors décomposé (lignes 9 à 13) et la méthode invoquée est recherchée (ligne 13). Quand elle n'est pas trouvée, un message d'erreur est envoyé à l'objet (ligne 15).

Le modèle proposé par J. FERBER suit la philosophie première de P. MAES: un méta-objet est responsable de l'interprétation de l'objet. Ainsi la phase de recherche des méthodes est de la compétence des méta-objets. Elle est effectuée par le méta-objet et non par la classe de l'objet.

9.2.2 Notre solution pour le contrôle

Le contrôle que nous proposons est plus proche de celui de S. CHIBA [CHIB 93a]. En effet, la phase de recherche de la méthode reste du domaine de la classe, la méthode résultante étant passée en argument lors de l'appel de la méthode `control` (ligne 4). Nous ne réifions pas les messages ; cependant une telle solution peut facilement être mise en place.

[5] Originalement nommée `:Handlemsg` dans [FERB 89].

```
1 (define send (obj sel .args)
2   (if (and (has-controller? obj)
3            (is-control-mandatory? (controller obj) obj sel))
4       (send (controller obj) control obj sel args (lookup (class-of obj) sel)))
5       (apply (default-lookup obj sel) obj args))))
```

Ce code doit être vu comme une description possible de la prise de contrôle. Celle-ci dépend principalement du langage dans lequel le modèle est implémenté ; elle diffère pour chaque implémentation. Cependant, cette description permet de préciser le découpage choisi.

1. Pour des raisons d'efficacité, le contrôle des messages n'est pas directement lié au fait qu'un objet possède un contrôleur. Un message particulier n'est contrôlé que s'il est adressé à un objet particulier (ligne 3). Ainsi nous limitons le contrôle aux seuls messages nécessaires.

   La fonction `is-control-mandatory?` rend vrai si le message doit être contrôlé, faux sinon.

2. La fonction `default-lookup` réalise le mécanisme la recherche standard des méthodes. C'est elle qui déclenche un message d'erreur (comme le message `doesNotUnderstand:` de Smalltalk) lorsque la méthode invoquée n'est pas trouvée.

3. La fonction `lookup` réalise une recherche des méthodes ne déclenchant pas d'erreur lorsque la méthode n'est pas trouvée[6]. En cas de succès cette fonction rend la méthode recherchée ; en cas d'échec elle rend une valeur permettant de savoir qu'il y a eu échec. La différence entre la fonction `lookup` et `default-lookup` est fondamentale. En effet, en cas de contrôle des messages, l'émission d'une erreur est de la responsabilité du contrôleur. Ceci permet en particulier d'implémenter de la délégation de messages : au lieu que l'échec mène à une erreur, le contrôleur gère cette absence de méthodes.

9.2.3 Primitives d'accès

Notre intégration de méta-objets dans un langage à classes, nous oblige à enrichir la structure minimale d'un objet. Tous les objets de notre langage peuvent se voir associer un contrôleur. Nous aborderons en détail les conditions de cette association en 9.4.2.

Cependant, contrairement à des langages comme Moostrap dans lequel chaque objet possède un méta-objet, notre approche est basée sur une association objet méta-objet dite *paresseuse*. Un méta-objet n'est associé à un objet que si cet objet participe à une dépendance. Les primitives suivantes sont proposées :

- (controller ⟨*object*⟩) : rend le contrôleur associé à un objet.
- (has-controller? ⟨*object*⟩) : précise si un objet possède un contrôleur.
- (all-controllers) : rend la liste de tous les contrôleurs du système.

Aucune primitive n'est offerte pour changer explicitement le méta-objet d'un objet. Les raisons de ce choix sont expliquées en 9.4. Par contre, le programmeur peut, au moment de l'instanciation d'un objet préciser son contrôleur, comme le montre l'exemple suivant :

```
(make stack :controller aController)
```

[6] Dans le cadre d'un langage à base CLOS, nous avons redéfini la fonction générique `no-applicable-method` afin qu'elle ne fasse pas d'erreur directement mais rende à la place une valeur permettant d'identifier une absence de méthode.

9.3 Contrôleurs et maintien de la cohérence en FLO

La classe abstraite `Controller` est la racine du graphe d'héritage des contrôleurs (voir figures 8.3 et 9.2). Elle définit les principales méthodes du comportement des contrôleurs : l'enregistrement des dépendances, la gestion des dépendances ... et surtout le maintien de la cohérence des dépendances.

Dans cette partie, nous commençons par présenter les comportements relatifs à la gestion des dépendances. Ensuite, nous décrivons le maintien de la cohérence des dépendances qui est le cœur même d'un contrôleur.

9.3.1 Comportements des contrôleurs

Nous présentons maintenant les comportements minimaux définis pour un contrôleur.

Déclaration. Lors de la déclaration d'une dépendance, cette nouvelle dépendance doit être portée à la connaissance des contrôleurs des objets impliqués. Ce n'est qu'après son enregistrement que sa présence influe sur le comportement des objets.

Pour cela, la méthode `establish-link` définie sur la classe `Link` invoque la méthode `register -link` définie sur la classe `Controller`. De la même manière, lorsqu'une dépendance est détruite, la méthode `remove-link` invoque la méthode `unregister-link` définie sur la classe `Controller`. A des fins d'optimisations, nous ne contrôlons que les messages perturbants. Notre sélection tient compte à la fois du sélecteur et du receveur du message. Dans ce but, la méthode `register-link` de la classe `controller` utilise la fonction `declare-control`. Inversement la méthode `unregister-link` invoque la fonction `declare-uncontrol`.

Lors de la réception par un objet contrôlé d'un message, un contrôleur doit savoir si parmi les dépendances qu'il a enregistré certaines nécessitent d'être maintenues. Etant donnés un objet, un opérateur et un sélecteur, la méthode `get-links` définie sur la classe `controller` rend la liste des dépendances devant être maintenues.

Les trois comportements, `register-link`, `get-links` et `unregister-link` sont les méthodes de base de la gestion des dépendances au niveau des contrôleurs. Elles sont dépendantes les unes des autres. Ainsi `register-link` ou `unregister-link` utilisent une clé dont le codage doit être connu et utilisé par la méthode `get-links`.

- (GET-LINKS ⟨*controller*⟩ ⟨*object*⟩ ⟨*operator*⟩ ⟨*selector*⟩) : cette méthode rend la liste des dépendances possédant une règle d'interaction d'opérateur *operator* pour l'objet *object* et la méthode de sélecteur *selector*.

- (REGISTER-LINK ⟨*controller*⟩ ⟨*key*⟩ ⟨*link*⟩) : enregistre la dépendance *link* comme étant associée à la clé *key*. Après l'exécution de cette méthode la dépendance effective *link* est prise en compte lors du contrôle des messages envoyés aux objets sur lesquels elle porte. Par défaut, la clé est constitué d'un objet, d'un sélecteur et d'un opérateur.

- (UNREGISTER-LINK ⟨*controller*⟩ ⟨*key*⟩ ⟨*link*⟩) : après l'exécution de cette méthode, la dépendance effective *link* n'est plus prise en compte lors du maintien de cohérence des dépendances.

Les fonctions d'optimisation du contrôle :

- (DECLARE-CONTROL ⟨*controller*⟩ ⟨*object*⟩ ⟨*selector*⟩) : spécifie que les messages de sélecteurs *selector* et de receveur *objet* sont contrôlés.

- (DECLARE-UNCONTROL ⟨*controller*⟩ ⟨*object*⟩ ⟨*selector*⟩) : spécifie que les messages de sélecteurs *selector* et de receveur *objet* ne sont plus contrôlés.

Introspection. Il est parfois nécessaire de connaître tous les objets ou les dépendances sous la tutelle d'un contrôleur. Pour cela deux méthodes existent sur la classe `Controller` : la méthode `objects` qui rend la liste de tous les participants et la méthode `links` qui rend la liste de toutes les dépendances associées à un contrôleur.

- (LINKS ⟨*controller*⟩) : rend la liste de toutes les dépendances dont le contrôleur doit assurer la cohérence.
- (OBJECTS ⟨*controller*⟩) : rend la liste de tous les objets contrôlés par le contrôleur.

9.3.2 Maintien

Au chapitre 6 en 6.1.3, nous avons précisé que le maintien de la cohérence doit être spécifique, optimisé, permettre la combinaison et la définition de nouveaux opérateurs. Nous avons conclu qu'il doit être minimal et adaptable. Celui-ci assure la gestion de différents opérateurs et la possibilité d'en définir de nouveaux.

Le maintien de la cohérence des dépendances réside dans l'interprétation des opérateurs du comportement dynamique lors du contrôle d'un message perturbant. Il est mis en œuvre par la méthode `control` définie sur la classe `Controller`. C'est en définissant de nouvelles méthodes ou en spécialisant cette méthode que la combinaison d'opérateurs ou la définition de nouveaux opérateurs est gérée. La méthode `control` joue un rôle central du comportement d'un contrôleur ; ceci a pour conséquence d'offrir une hiérarchie de spécialisation *«plate et peu profonde»* (cf. figure 9.3).

Nous présentons maintenant le comportement minimal de cette méthode ainsi que les comportements liés aux opérateurs `implies` et `permitted-if`. Un comportement à la fois gardé et réactif est défini par héritage multiple entre les classes `Guarded-Controller` et `Reactive-Controller` (voir figure 9.3).

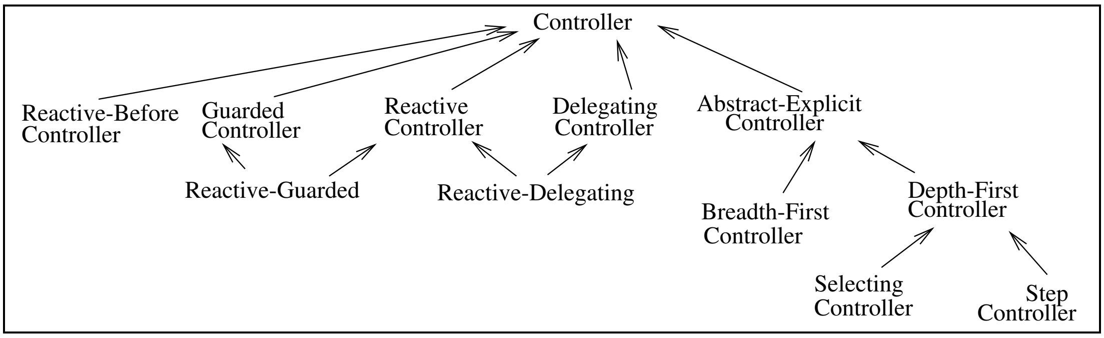


Figure 9.3: Hiérarchie et combinaison d'opérateurs par héritage multiple entre classe de contrôleurs.

Comportement minimal. La méthode `Control` définie sur la classe `Controller` applique simplement la méthode contrôlée[7] passée en argument.

```
(define-method control ((ctrl Controller) rec sel args defmet)
  (if (is-method? defmet)
      (apply defmet rec args)
      (error rec sel args)))
```

[7] Cependant, ce comportement n'est invoqué qu'indirectement au travers des spécialisations de la méthode `control` définies sur des sous-classes de la classe `Controller`. En effet, appliquer directement cette méthode n'a guère de sens, c'est pourquoi la classe `Controller` est abstraite.

Remarque. Tester si la méthode a été trouvée par la fonction `lookup` est important. Car bien que tous les messages perturbants (c'est-à-dire appartenant à une dépendance) soient déclarés comme devant être contrôlés (`declare-control`) durant la déclaration d'une dépendance, il est possible que le sélecteur d'un tel message ne corresponde pas à une méthode. Cette façon de procéder est liée au fait que nous ne vérifions pas la validité des dépendances (voir en 8.2.2).

Comportement réactif. Le comportement de maintien le plus simple est défini sur la classe `Reactive-Controller` qui hérite de la classe `Controller`.

```
1 (define-method control ((ctrl Reactive-Controller) rec sel args defmet)
2    (let ((result (next-method))
3       (for-each (lambda (x) (firing x sel (cons rec args) result))
4                  (get-links ctrl rec 'implies sel))
5        result))
```

Cette méthode applique la méthode contrôlée (ligne 2) et invoque les messages compensatoires liés aux messages par le biais de la méthode `firing` (lignes 3 et 4). La méthode `get-links` rend la liste des dépendances dont des réactions doivent être déclenchées afin de garantir la cohérence de celles-ci.

Gardes. La gestion des gardes implique de pouvoir interdire l'application d'une méthode perturbant la cohérence de la dépendance. La classe `Guarded-Controller` qui hérite de la classe `Controller` définit la méthode `control` suivante :

```
1 (define-method control ((ctrl Guarded-Controller) rec sel args)
2   (when (every (lambda (x)
3                (verify-guard x sel (cons rec args)))
4                  (get-links ctrl rec 'permitted-if sel))
5         (next-method)))
```

Toutes les dépendances définissant des gardes pour le message contrôlé sont vérifiées (lignes 2 et 3), dans l'affirmative le message est appliqué (ligne 5).

- (CONTROL ⟨*controller*⟩ ⟨*objet*⟩ ⟨*selector*⟩ ⟨*arguments*⟩ ⟨*method*⟩) : cette méthode est appelée automatiquement sur le contrôleur de l'objet receveur lors d'un envoi de messages déclenchants. *object* est le receveur, *selector* le sélecteur, et *arguments* la liste des arguments du message contrôlé. *method* est la méthode contrôlée.

9.4 Sûreté et optimisations

Les contrôleurs ont la responsabilité de maintenir la cohérence des dépendances. Or tout le succès de cette entreprise réside dans l'adéquation entre la sémantique des opérateurs de la dépendance et leur traitement au niveau des contrôleurs des objets y participant. En effet, si le contrôleur d'un objet devant être soumis à une garde ne gère pas les règles d'interactions basées sur l'opérateur `permitted-if`, les gardes ne sont pas prises en compte. Cette contrainte est donc très importante pour la sécurité du système. Nous avons établi en 6.1.3 que le maintien de la cohérence doit être spécifique aux opérateurs, minimal et extensible. Nous montrons maintenant comment FLO offre un mécanisme automatique, mais particularisable, de vérification et de mise en place de la cohérence entre l'interprétation des opérateurs et leur interprétations par les contrôleurs.

9.4.1 Déclarations

Lors de la définition d'un nouvel opérateur, le méta-programmeur doit le déclarer. Pour cela, il doit définir quel est le contrôleur minimal satisfaisant la cohérence de cet opérateur. Ainsi pour l'opérateur `implies`, il faut déclarer que le contrôleur `Reactive-Controller` est la classe de contrôleur minimale gérant cet opérateur. Ceci se fait soit à la création de l'opérateur (ligne 4) ou soit ultérieurement (ligne 5).

```
1 (define implies (make operator:name 'implies
2                               :link-class Reactive-Link
3                               :link-meta-class Reactive-Meta-Link
4                               :controller-classes (list Reactive-Controller)))

5 (add-controller implies Breadth-First-Controller)
```

Contrairement aux classes de liens, deux contrôleurs sans relation de spécialisation peuvent assurer une gestion *sémantiquement correcte* d'un même opérateur. Ainsi l'opérateur `implies` est géré par le controller `Reactive-Controller` mais aussi par le controller `Selecting-Controller` qui filtre certains messages compensatoires, ou `Breadth-First-Controller` qui propage les messages compensatoires en largeur. Lors de la définition de chaque contrôleur gérant l'opérateur `implies` une telle déclaration est nécessaire, elle crée un nouveau chemin dans le graphe de compatibilité des contrôleurs (voir figure 6.5).

9.4.2 Vérification et mise en place

La figure 9.4 illustre la solution mise en place. Les contrôleurs gèrent avec le plus de finesse possible l'interprétation des opérateurs des dépendances. Ainsi, lorsqu'une nouvelle dépendance est déclarée, le système vérifie si les contrôleurs en place satisfont les nouvelles contraintes par le biais de la méthode `check-and-error` définie sur la classe `Link`. Lorsque ce n'est pas le cas, il affecte de nouveaux contrôleurs aux objets de telle sorte que les opérateurs des dépendances soient correctement pris en compte.

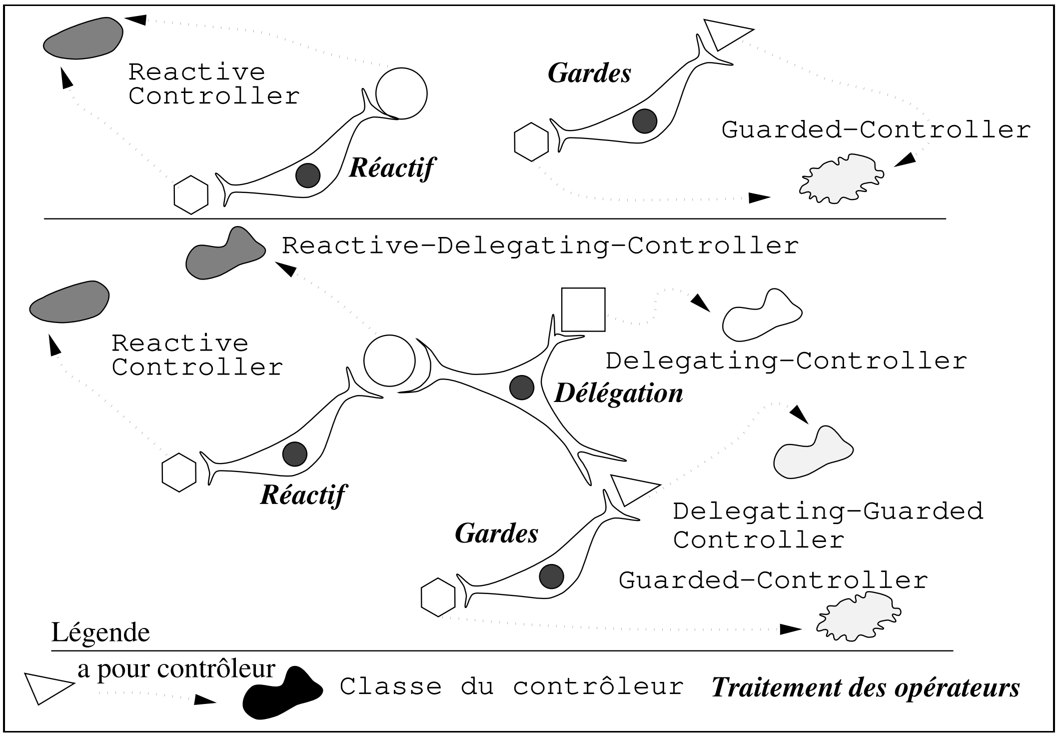


Figure 9.4: Prise en compte de la gestion d'un nouveau comportement lors de l'ajout d'une dépendance.

La méthode `check-and-correct` définie sur la classe `Link` est responsable de ce mécanisme.

Elle est invoquée par la méthode **make** lors de la déclaration d'une dépendance effective avant que ne soit appelée la méthode `establish-link` (voir en 8.2.2).

```
1 (define-method check-and-correct ((l Link))
2    ...
3   (let ((sem (operators l)))
4     (for-each
5        (lambda (object)
6          (unless (has-controller? object))
7                  (controller-set! object (find-controller sem))
8          (let ((ctrl (controller object))
9             (unless (is-compatible? ctrl sem)
10                    (change-compatible (find-compatible-controller ctrl sem)
11                                       object ))))
12    (give-all-objects l))))
```

Pour chaque participant à la dépendance, la méthode `check-and-error` procède en deux temps.

1. Tout d'abord, il s'agit de s'assurer qu'un object participant possède un contrôleur (ligne 5). Lorsque ce n'est pas le cas, un contrôleur gérant les opérateurs de la dépendance lui est associé (ligne 6). La fonction `find-controller` utilise le graphe de compatibilité des contrôleurs pour définir le contrôleur nécessaire à la dépendance.
2. La compatibilité entre le contrôleur de chaque participant et le comportement démandé par la dépendance est vérifiée (ligne 9) (méthode `is-compatible?`). Lorsque le contrôleur ne gère pas les opérateurs de la dépendance, la méthode `change-compatible` est appelée. Cette méthode a en charge d'affecter un nouveau contrôleur compatible à l'objet. La méthode `find-compatible-controller` a pour rôle de déterminer un contrôleur compatible. Elle utilise pour cela le graphe de compatibilité des contrôleurs.

On pourrait croire que le fait d'affecter le contrôleur spécifié par la dépendance dispense de vérifier si ce contrôleur est compatible avec celui pouvant être choisi par le programmeur (ligne 7).

La figure 9.4 illustre ce besoin. Ainsi bien que l'objet représenté par un cercle possède déjà un contrôleur, l'ajout de la dépendance nécessitant un contrôleur de délégation implique que le contrôleur du cercle gère l'opérateur réactif et de délégation.

Il faut d'ailleurs remarquer que l'utilisation de la primitive `controller-set!` peut causer de gros dégâts pour la cohérence du système si elle est mal utilisée. C'est pourquoi cette primitive est définie localement[8] en STklos à la méthode `check-and-correct` (représentée par l'ellipse) et ne peut en aucun cas être utilisé directement sur un objet.

Limites de cette solution lors de la globalité. Etant donné un objet, le choix de la nature de son contrôleur est déterminé soit (a) automatiquement par les opérateurs des dépendances comme nous venons de le montrer (réaction, délégation, gardes...), soit (b) par la gestion globale mise en œuvre par le programmeur (largeur, profondeur, filtrage ...) (voir en 6.1.1).

Ce dernier point pose un problème important : comment déterminer automatiquement le bon contrôleur dans un cas de gestion globale de dépendances? La solution ne peut s'appuyer sur la présence d'opérateurs car un même opérateur peut être traité de différentes manières tout en respectant son interprétation.

L'opérateur `implies` est géré par les classes de contrôleurs `Reactive-Controller`, `Selecting-Controller`...

[8]Nous faisons directement appel à une primitive C de l'interprète.

Le mécanisme proposé par la méthode `check-and-error` est très bien adapté à la situation (a). Il assure ainsi correctement la prise en compte de nouveaux opérateurs. Par contre, il ne gère pas le choix des contrôleurs liés à la globalité. Il est alors de la responsabilité du programmeur de spécifier les bons contrôleurs. Le système s'assure alors simplement que les contrôleurs sont cohérents avec les opérateurs. La figure 9.5 illustre le problème.

> Lorsque le programmeur souhaite que son système gère les dépendances en filtrant les messages compensatoires (il leur associe un contrôleur de la classe `Selecting-Controller`). Si une nouvelle dépendance est ajoutée et qu'un des nouveaux participants (l'objet hexagonal) n'a pas de contrôleur, le mécanisme ne choisit pas un contrôleur de propagation avec filtrage mais un contrôleur lié aux opérateurs spécifiant la dépendance, c'est-à-dire réactif simple. Or le contrôleur de l'objet hexagonal doit être un contrôleur gérant le filtrage des messages compensatoires et non un simple comportement réactif.

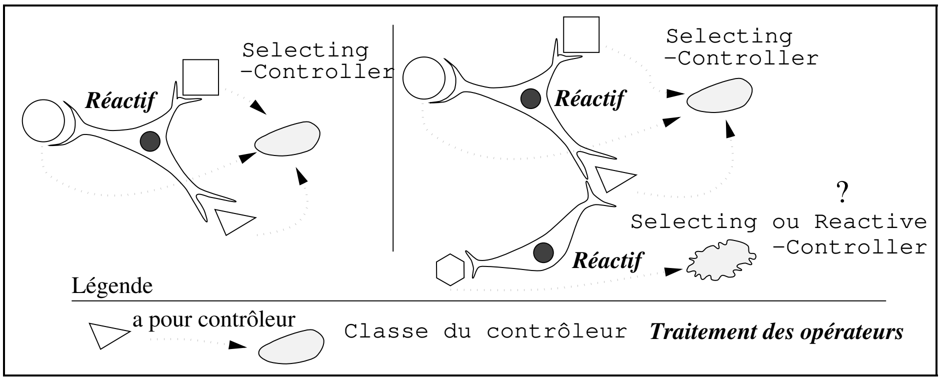


Figure 9.5: Problème de l'ajout d'une dépendance dans le cas de contrôleurs non déductibles par les opérateurs des dépendances.

Notre solution assure donc la *vérification* des contrôleurs mais ne déduit pas automatiquement le bon contrôleur dans une gestion globale. Une solution peut aider le programmeur à automatiser l'association objets/contrôleurs. Lors d'un maintien global de la cohérence, il est possible de spécialiser la méthode `is-compatible?` afin qu'elle ne rende vrai que pour une certaine catégorie de contrôleurs et la méthode `find-compatible-controller` pour qu'elle rende le contrôleur nécessaire. Cependant, cette solution ne nous satisfait que parce qu'elle oblige le programmeur à prendre ses responsabilités. De plus, toutes les autres solutions envisagées restaient peu satisfaisantes. La principale difficulté provient du fait qu'une dépendance spécifie des comportements locaux et n'a pas d'exigence quant à sa gestion de manière globale.

9.4.3 A propos de la composition de contrôleurs

Ce problème est similaire aux problèmes rencontrés avec les méta-classes explicites et nommés dans la littérature *compatibilité de méta-classes* [Grau 89, Danf 94a, Danf 94b] et souligne la nécessité de schémas de composition entre méta-objets [Mule 95b, Mule 95a].

Nous n'avons pas envisagé la création automatique de classes de contrôleurs dans le sens des méta-classes dérivées de SOM [Danf 94b]. Cela pour les raisons suivantes : tout d'abord car les changements impliqués lors de la définition de nouveaux contrôleurs sont importants pour le langage, aussi préférons nous que le méta-programmeur soit bien conscient de ses modifications. D'autre part, il nous faudrait déterminer un ordre global sur l'évaluation des opérateurs.

La combinaison des comportements nécessaires pour gérer différents opérateurs ne peut pas être définie de manière automatique. Les contrôleurs possèdent, contrairement aux méta-objets comportementaux et à la méthodologie décrite par P. Mulet dans [Mule 95b, Mule 95a], des comportements difficilement composables[9] automatiquement.

[9] Soient f et g deux gestions du maintien, f et g sont composables si f(g(x)) ou g(f(x)) a du sens.

Séquentialité des opérateurs. Pour composer le comportement d'un contrôleur gérant des messages compensatoires avec un contrôleur gérant des gardes ou des délégations de messages, la sémantique des opérateurs par rapport à l'application de la méthode doit être prise en compte. Il faut d'abord vérifier les gardes, puis si elles sont vraies appliquer la méthode et déclencher les messages.

Asymétrie des compositions. Certains comportements ne sont pas composables de manière symétrique. Ainsi soit CR le comportement gérant les messages compensatoires et CG celui gérant les gardes, CR(CG) est possible alors que CG(CR) n'a pas de sens.

Non composables. Certains comportements peuvent être simplement non composables. Ces comportements ne sont pas obligatoirement liés à un opérateur. Ainsi, la propagation en largeur ou le filtrage des messages ne sont pas composables avec le comportement réactif simple.

> On peut imaginer un opérateur `implies-substitute` tel que le résultat de la méthode contrôlée soit celui de l'action associée et non celui de l'application de la méthode. Dans ce cas, ce comportement ne peut pas être composé avec l'opérateur `implies`, `implies-before`.

9.5 Extensions

Dans cette partie, nous présentons quelques-unes des extensions possibles du comportement des contrôleurs. Certaines extensions sont à la fois simples de mise en œuvre mais complexes par leurs implications sur le langage ou par la connaissance des interactions entre les différents entités du langage.

9.5.1 L'opérateur corresponds

Dans le modèle en 4.2, nous avons introduit un opérateur de délégation : l'opérateur `corresponds`. Nous illustrons maintenant comment ce nouvel opérateur est défini.

Mise en œuvre. Tout d'abord, nous définissons la classe `Delegate-Meta-Link` et la méthode `inherit-corresponds`. Ensuite la classe `Delegate-Link`, instance de cette classe, et la méthode `apply-corresponds` sont définies. Finalement nous définissons une nouvelle classe `Delegating-Controller` qui hérite de la classe `Controller` et nous redéfinissons la méthode `control` comme suit :

```
1 (define-method control ((ctrl Delegating-Controller) rec sel args defmet)
2     (if (is-method? defmet)
3         (let ((links (get-links ctrl rec 'corresponds sel)))
4            (if (null? l)
5                (error rec sel args)
6                (map
7                  (lambda (x) (apply-correspond x sel (cons rec args) result))
8                  links)))
9         (next-method)))
```

Ainsi lorsque la méthode recherchée est trouvée, le contrôle standard est exécuté (ligne 9). Par contre, quand aucune méthode n'est trouvée (ligne 2), le contrôleur vérifie si des liens définissent un *message correspondant* (lignes 3 et 4). Dans ce cas, le contrôleur rend alors comme la liste des résultats des *messages correspondants* (lignes 6 et 7). Lorsque le message envoyé n'a pas de *messages correspondants*, une erreur est signalée (ligne 5).

Pour parachever le tout, il faut définir l'opérateur `corresponds` comme suit :

```
(define corresponds
       (make operator :name 'corresponds
                      :link-class Delegate-Link
                      :link-meta-class Delegate-Meta-Link
                      :controller-classes (list Delegating-Controller)))
```

Exemple. Supposons que nous ayons deux objets liés ou non par une structure de composition, une main `a` et un bras `h`. Nous pouvons lier ces objets afin que certains messages soient redirigés de l'un vers l'autre (voir en 5.5.1).

```
(deflink arm-hand (:hand :arm)
 :access '((hand-of :hand) (arm-of :arm))
 :behavior
  '(((open :arm) corresponds (open :hand))
    ((close :arm) corresponds (close :hand))
    ((turn-rigth :hand) corresponds (turn-rigth :arm))
    ((turn-left :hand) corresponds (turn-left :arm)))
    ((remove-link :link) implies (destroy :hand) (destroy :arm)))
```

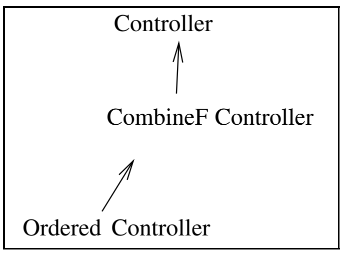
La dernière ligne de cette définition spécifie que lorsque la dépendance est détruite les objets qui la composent le sont également. Elle illustre le fait qu'une dépendance peut intervenir directement dans l'interaction (voir chapitre 10).

```
(define a (make arm))
(define h (make hand))
(define a-h (make arm-hand :hand h :arm a))
```

Remarques liées à Clos. En CLOS ou STklos, l'invocation de méthodes est basée sur l'application de fonctions génériques, une fonction générique regroupant plusieurs méthodes [Keen 89, Stee 90, Pae 93]. Ainsi introduire un opérateur du type de `corresponds` nécessite de créer des fonctions génériques afin que leurs applications puissent être contrôlées, de telles fonctions génériques étant alors *«vides»* de méthodes.

9.5.2 Ordre et sélection

Besoin. L'ordre d'émission des messages compensatoires est par défaut celui de la déclaration. Or, il est parfois nécessaire de pouvoir spécifier l'ordre ou une sélection des dépendances à déclencher. ceci est particulièrement utilisé dans les bases de données orientées objets actives [Hans 93, Wido 96].



Solution. Deux approches sont possibles pour ordonner les messages compensatoires : soit ordonner les dépendances lors de leur création, soit les ordonner dynamiquement. Cette dernière solution, qui est moins efficace, est par contre plus souple. Il est possible alors de ne déclencher qu'une certaine catégorie de dépendances. Ces deux approches peuvent être combinées, nous montrons ici la seconde car elle illustre une particularisation du contrôle des messages.

Une nouvelle classe de contrôleur `CombineF-Controller` héritant de `Controller` est définie (voir figure 9.5.2).

Mise en œuvre. La méthode `control` est spécialisée de la manière suivante :

```
(define-method control ((ctrl CombineF-Controller) rec sel args defmet)
   (let ((result (next-method)))
      (combine-firings ctrl
                       (get-links ctrl rec 'implies sel)
                        rec sel args result)
       result))
```

Par défaut, la méthode de combinaison des dépendances `combine-firings` invoque simplement tous les messages compensatoires.

```
(define-method combine-firings ((ctrl CombineF-Controller) sel lks args res)
   (for-each (lambda (x) (firing x sel (cons rec args) result))
            lks))
```

Il faut déclarer que cette nouvelle classe de contrôleurs gère l'opérateur `implies` et s'assurer que tous les objets d'une même dépendance ont bien le même type de contrôleur.

```
(add-controller implies CombineF-Controller)
```

Des dépendances pondérées. Supposons que l'on veuille déclencher des dépendances suivant la valeur d'un poids associé. Pour cela, on définit une nouvelle classe de dépendance `With-Weigth-Link` ayant une variable d'instance `firing-priority`. Associé un poids à une dépendance est fréquent dans les règles ECA des bases de données [Daya 88, Daya 96, Hans 93, Wido 96].

```
(define-class With-Weigth-Link (Link)
   ((firing-priority :initform 0 :init-keyword :priority :accessor firing-priority)))
```

On définit une sous-classe de `CombineF-Controller` : la classe `Ordered-Firings`, sur laquelle on spécialise la méthode `Combine-firings` comme suit :

```
(define-method combine-firings ((ctrl Ordered-Firings) sel lks args res)
  (for-each (lambda (x) (firing x sel (cons rec args) result))
      (sort lks (lambda (x y)
                  (<
                    (if (subclass? (class-of x)  With-Weigth-Link)
                        (firing-priority x) 0)
                    (if (subclass? (class-of y)  With-Weigth-Link)
                        (firing-priority y) 0))))))
```

Cette méthode accepte des dépendances ne définissant pas de priorité. Dans ce cas, elle les classe avec la plus basse priorité. Pour utiliser cette fonctionnalité, un programmeur définit simplement ses dépendances comme héritant de la classe `With-Weigth-Link` et associe la priorité désirée aux instances.

Remarques. Une solution limitant le travail des sous-classes de `CombineF-Controller` est de décomposer l'application des messages compensatoires en deux temps : le déclenchement et la manipulation de la liste des dépendances. Ainsi lorsqu'il ne s'agit que d'un ordonnancement, seule la méthode responsable de la manipulation de la liste est modifiée. Cependant, cette méthode introduit un nouvel appel de méthode et est donc plus coûteuse.

De la même manière, il est possible de filtrer les dépendances à déclencher en spécialisant la méthode `combine-firings`, il est possible de filtrer les dépendances à déclencher. Alors que par défaut toutes les gardes doivent être vérifiées, il est possible de mettre en place un mécanisme de sélection des gardes. Ainsi, on peut choisir de ne vérifier que la ou les gardes de priorité la plus élevée et permettre une *relaxation* de certaines dépendances [Born 87].

9.5.3 Profondeur et largeur

L'algorithme standard de maintien de la cohérence mélange la propagation locale et globale : à chaque fois qu'un objet reçoit un message à contrôler, le même contrôle est effectué. A cause de l'évaluation des *messages compensatoires*, cet algorithme est *implicitement* un algorithme en profondeur. Cela a pour conséquence de rendre le contrôle du flot de propagation peu modifiable ; par exemple, l'élimination de messages redondants est difficile à implémenter.

Maintenant, nous présentons un contrôle qui intègre une vision globale de l'algorithme de propagation. Supposons que le contrôle de m_1 implique m_{1i} comme *messages compensatoires*, qui implique à son tour d'envoyer les messages m_{1ij}. Nous obtenons ainsi un flot de propagation qui peut être en profondeur ou en largeur (cf. figure 9.6).

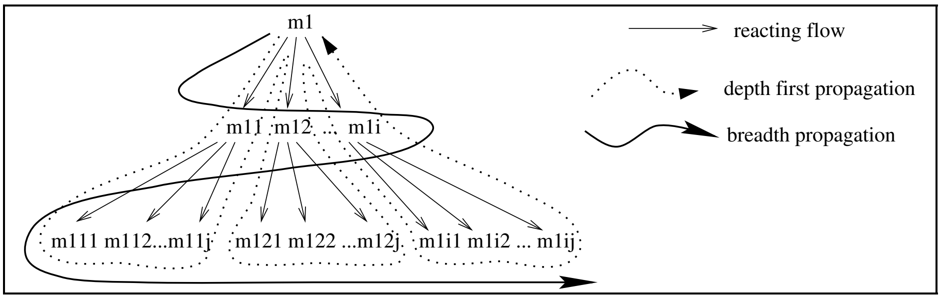


Figure 9.6: Deux flots de propagation lors d'une phase compensatoire : en profondeur ou en largeur.

Profondeur explicite. Nous rendons apparente la structure du contrôle. Pour cela, nous créons une nouvelle classe de contrôleur, appelée `AbstractExplicitController`, ayant la nouvelle variable d'instance `place`. Cette variable est utilisée afin de stocker les *messages compensatoires*. Trois méthodes sont nécessaires à sa gestion : `put`, `get` et `empty?`. La définition de telles méthodes abstraites nous permet de spécialiser le comportement du contrôle de l'envoi de messages. Une dernière méthode `apply-possible?` est définie : elle rend vrai quand l'application des messages stockés est possible.

```
(define-class AbstractExplicitController (Controller)
     ((place :initform '() :accessor place)))
```

Nous définissons une nouvelle méthode `return-messages` qui retourne la liste des *messages compensatoires*. Ces derniers sont alors stockés (`put` ligne 11) dans la variable d'instance `place` du contrôleur et sont utilisés (`get` ligne 13).

```
(define-method control ((ctrl Abstract-Explicit-Controller) rec sel args method)
  (let* ((result (apply method rec args)))
      (for-each (lambda (x) (put ctrl x))
                (return-messages ctrl
                         (get-links ctrl 'implies sel rec)
                         rec sel args result))
             (if (apply-possible? ctrl)
                     (apply (get ctrl)))
          result)))))
```

Contrôle en profondeur ou comportement standard. Afin d'obtenir une propagation en profondeur, nous créons un nouveau contrôleur qui hérite de `Abstract-Explicit-Controller`. Nous considérons `place` comme une pile. Ainsi la méthode `put` empile et `get` dépile des éléments (on considère que `get` rend une valeur particulière lorsque la pile est vide).

Contrôle de la propagation en largeur. Pour avoir un contrôle en largeur, nous créons une nouvelle classe de contrôleur `Breadth-First-Controller` qui hérite de `Abstract-Explicit-Controller`. Cette fois-ci `place` est considérée comme une file : la méthode `put` met dans une file et `get` prend le premier élément d'une file. La méthode `control`, quant à elle, n'est pas modifiée.

Pour ces deux contrôles, la méthode `apply-possible?` teste simplement s'il y a des éléments dans la pile ou la file.

Contrôle pas à pas. Une telle implémentation du contrôle facilite l'ajout de nouvelles fonctionnalités. En particulier, nous introduisons la possibilité d'avoir une trace en décomposant tous les *messages compensatoires*. Nous créons une nouvelle classe de contrôleurs (sous-classe de `Depth-First-Controller` ou `Breadth-First-Controller`), avec une nouvelle variable d'instance `step-mode` qui nous indique si le contrôle est en mode normal ou pas à pas. Nous définissons alors deux nouvelles méthodes : `toggle-step-mode` pour passer d'un mode à l'autre et `apply-next-implied` qui appelle la méthode `get` et applique la méthode retournée. De plus, la méthode `apply-possible?` est spécialisée afin que l'application automatique des messages ne soit possible qu'en mode normal.

```
(define-method apply-possible? ((ctrl Step-Controller))
  (not (or (step-mode ctrl) (empty?  ctrl))))

(define-method apply-next-implied ((ctrl Step-Controller))
    (unless (empty? ctrl)
            (apply (get ctrl))))
```

Filtrages des réactions. Filtrer les messages compensatoires est simple à mettre en œuvre, il suffit de spécialiser les méthodes d'ajout `put` ou de retrait `get` afin de n'ajouter ou retirer de la pile/queue que certains messages.

9.6 Conclusion

« It isn't a good idea to try and make the first version of a new kind of system open in this sense. Opening the implementation critically depends on understanding not just one implementation the clients might want, but also the various kinds of variability around that point they might want. » [Kicz 92a].

MOP complet. Au cours de ce chapitre et du précédent, nous avons montré l'architecture sur laquelle repose notre intégration des dépendances. Un protocole méta objet a été présenté (voir 9.7) et nous avons montré quelques unes de ses possibles extensions. Bien que ce protocole soit le fruit d'une lente maturation, nous pensons qu'il devrait être encore raffiné par de nombreuses autres utilisations. Nous sommes conscient que la conception d'un protocole est une tâche délicate et que le protocole que nous proposons possède sûrement des limites.

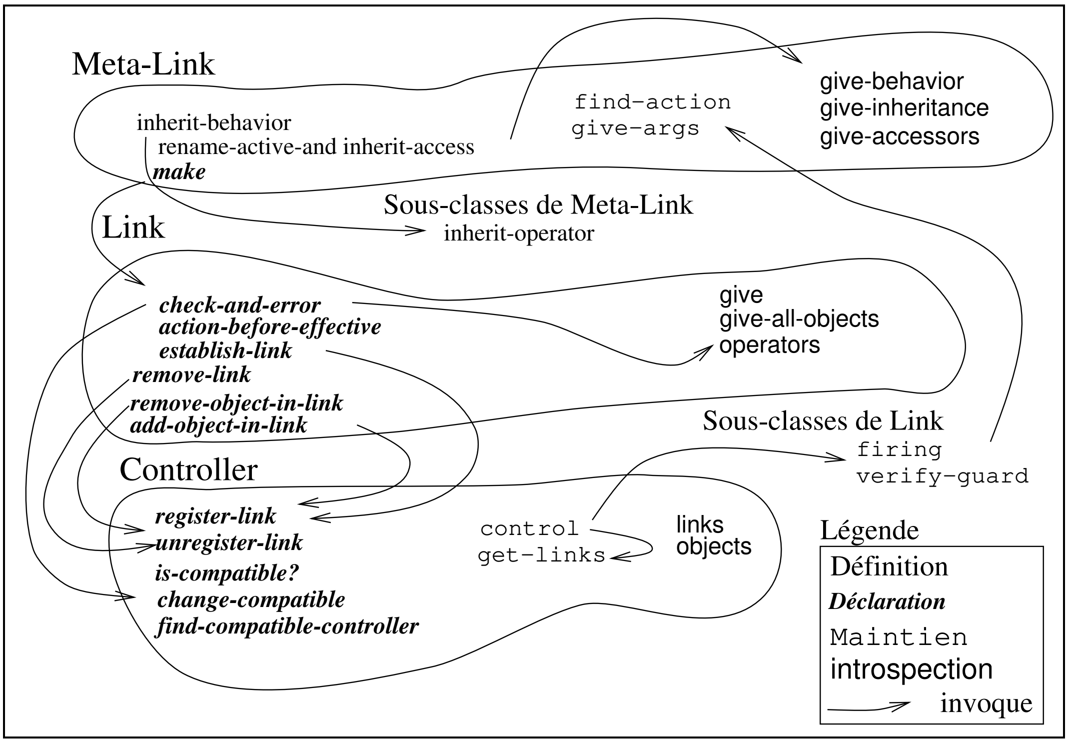


Figure 9.7: Vision d'ensemble du MOP.

Propriétés. Au chapitre 7, nous avons précisé les qualités qu'un MOP devait posséder [Kicz 92a] :

Étendue du contrôle. En FLO, différents aspects peuvent être particularisés (création, déclaration, comportement dynamique..) sur une dépendance, un groupe de dépendances, un contrôleur.

Séparation conceptuelle. Certains aspects de notre MOP peuvent être particularisés sans nécessiter sa compréhension globale.

Incrémentalité. Un méta-programmeur peut ajouter un nouveau comportement sans avoir à garantir la cohérence de l'ensemble.

Robustesse. Le noyau de base de FLO est robuste ; c'est-à-dire que l'ajout de fonctionnalités est prévu et certains mécanismes ont été proposés pour assurer la cohérence du système. Cependant, si un méta-programmeur commet une erreur lors de la définition de la méthode `control` pour la gestion d'un nouvel opérateur, notre système ne peut le détecter.

Réflexions et travaux futurs. Alors même que nous avons toujours gardé à l'esprit que l'efficacité d'un tel protocole était primordiale, nous n'avons pas mis en œuvre toutes les techniques connues pour le rendre moins pénalisant. Nous pensons en particulier que des techniques de *mémoization* pour les parties fonctionnelles du protocole pourraient être appliquées.

D'autre part, nous pensons que certains appels de méthodes du MOP concernant le maintien de la cohérence ne devrait être accessibles à la modification que sous la demande explicite du programmeur par le biais d'une déclaration d'utilisation. Dans la majeur partie des cas, certains appels pourraient être remplacées par l'appel de primitives et lorsque la nécessité voudrait qu'ils soient particularisés des appels aux méthodes du MOP seraient mis en place. Ainsi la méthode **`firing`** utiliserait une primitive à la place d'un appel à la méthode **`give-args`**. Ces considérations peuvent être prises en compte dans le cadre de langages non uniforme. Dans le cas de SMALLTALK, il n'est pas simple d'ajouter des primitives.

Des solutions devraient être trouvées pour garantir une meilleure gestion des comportements globaux. Nous pensons que l'utilisation de FLO dans le cadre d'une programmation distribuée doit valider ou montrer les limites de l'architecture et du MOP proposé ici.

Partie IV

Applications

Cette dernière partie présente différentes applications des dépendances. Nous abordons :

Implémentation des schémas de conception. Les schémas de conception *(design patterns)* restent avant tout des outils pour la conception d'applications. Nous montrons comment les dépendances peuvent être utilisées pour les implémenter.

Implémentation de modèles architecturaux en Interfaces Hommes-Machines. Les dépendances en assurant la communication entre objets sont un outil adapté pour la construction d'interfaces hommes-machines. Elles permettent une implémentation des modèles architecturaux comme le modèle ALV ou PAC.

Réification de l'héritage. Dans ce dernier chapitre, nous montrons comment le mécanisme d'héritage proposé dans le modèle ObjVlisp est réifié par le biais de dépendances. Ce choix nous conduit à étudier les problèmes de bootstrap et de stabilité de cette nouvelle implémentation de notre modèle.

10

Des dépendances pour l'implémentation de schémas de conception

« *Most authors propose design patterns as a mechanism specifically used during design...The disadvantage of a traditional language such as C++ is that no support for the representation of design patterns is provided by the language. This leads to traceability and implementation overhead problems related to the implementation of design patterns.* » [Bosc 96].

Dans ce chapitre, nous montrons comment les dépendances constituent une solution pour la prise en compte par les langages des schémas de conceptions. En particulier, nous montrons que les dépendances sont des entités permettant de mettre en œuvre de manière élégante et simple les *«schémas de conceptions» (design patterns)* utilisés lors de la phase de conception d'une application. Nous situons le contexte d'utilisation des schémas de conception puis présentons en quoi les dépendances sont intéressantes pour leur implémentation.

10.1 Contexte

« *Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it in the same way twice* » [Alex 77].

Les schémas de conception *(design patterns)* proviennent des travaux de l'architecte C. Alexander qui fut le premier à proposer des schémas *(patterns)* pour la conception en architecture [Alex 77]. Chaque schéma définit un problème récurrent et une solution générique. L'ensemble des schémas constitue alors un langage de schémas *(pattern language)* fournissant un ensemble de solutions composables pour un domaine particulier.

Le concept de schémas a naturellement trouvé sa place dans la communauté objet [Gamm 94]. Ces dernières années, les schémas de conception sont devenus très populaires au point d'avoir leur propre conférence. La plupart des auteurs proposent les schémas comme un outil de conception permettant de décrire des problèmes généraux de conception pouvant être particularisés pour des applications spécifiques. Ainsi les auteurs de [Gamm 94][page 3] définissent les schémas comme :

«descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.»

Les schémas de conception sont utilisés uniquement durant la phase de conception. Un schéma décrit alors une situation, les éléments qui entrent en jeu, leurs dépendances, leurs responsabilités et leurs collaborations. Un schéma offre une description abstraite du problème de conception et comment la mise en place des éléments qui le composent apporte une solution au problème. Certains auteurs proposent de possibles implémentations.

Inconvénients. Cependant, lors de l'implémentation des schémas des problèmes se posent comme le soulignent [Souk 95, Bosc 96]:

- Les schémas sont souvent perdus durant la phase d'implémentation car les langages de programmation n'offrent pas de concept correspondant. Le programmeur est alors contraint d'implémenter le schéma en le distribuant parmi les méthodes des objets participant au schéma.
- Pour Soukup [Souk 95], l'utilisation des classes pour représenter les schémas aboutit à des ensembles de classes mutuellement dépendantes.
- L'implémentation des schémas conduit trop souvent le programmeur à définir de nombreuses méthodes ayant un comportement simpliste, comme par exemple une redirection de message vers un autre objet. Ceci a pour conséquence de rendre le code produit moins compréhensible.

Soukup propose comme solution d'implémenter les schémas comme des classes C++. Cependant, il ne présente que peu de schémas et il n'est pas certain que tous les schémas puissent être définis comme des classes. La plupart des auteurs de schémas présentent des solutions qui malheureusement ne résolvent pas les problèmes décrits au-dessus. Nous montrons maintenant comment les dépendances permettent une implémentation de certains schémas. J. Bosch propose des solutions à ces problèmes en étendant son langage LayOM [Bosc 96]. Dans la suite, nous comparons nos solutions aux siennes.

Notre approche: des schémas à l'aide de dépendances. Les schémas ont une sémantique bien définie, aussi montrons-nous maintenant comment certains schémas peuvent facilement être implémentés à l'aide de dépendances. Afin de pouvoir comparer nos travaux à ceux de J. Bosch, nous présentons les mêmes schémas que ceux de [Bosc 96] tirés de [Gamm 94]: pour les schémas dit structurels *(structural design pattern)* l'`Adaptateur` *(Adapter)* et la `Facade` et pour les schémas dit comportementaux *(behavioral design pattern)* l'`Observateur`, *(Observer)*, et le `Médiateur`, *(Mediator)*. Pour chacun des schémas, nous présentons le but du schéma, les inconvénients dus à une implémentation traditionnelle et notre solution.

10.2 Schémas structurels

Les schémas structurels décrivent comment les classes et les objets doivent être composés pour former de nouvelles structures, par opposition aux schémas comportementaux qui décrivent plutôt les interactions entre objets.

10.2.1 Adaptateur

But et solution traditionnelle. Le schéma `adaptateur` est utilisé pour convertir l'interface d'une classe en une interface compatible avec celle attendue par un objet client. Il permet à des classes *a priori* incompatibles de coopérer. Dans un langage conventionnel, ce schéma s'implémente en utilisant un objet qui renvoie *(forward)* les messages après les avoir adaptés pour cet objet.

La figure 10.1 tirée de [Gamm 94] décrit les relations structurelles entre l'objet adapté et un tel adaptateur. L'objet `Client` collabore avec les objets par le biais d'une interface définie par l'objet `Target`. L'objet `Adapter` adapte l'interface de l'objet `Adaptee` afin qu'il soit conforme à l'interface attendue par l'objet `Client`. L'objet `Client` appelle des opérations sur une instance de l'`Adapter` qui lui invoque les opérations de l'objet `Adaptee`.

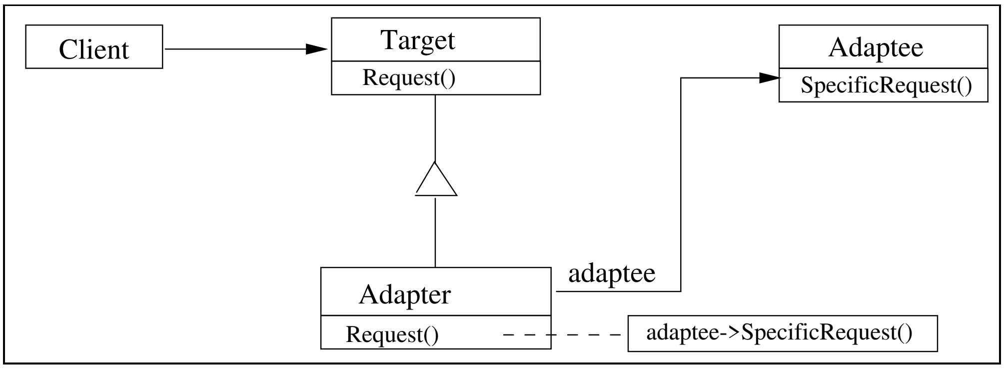


Figure 10.1: Représentation au niveau objet de l'architecture proposée dans Design Patterns pour le schéma Adaptateur.

Cette façon de procéder permet effectivement de faire coopérer des objets qui n'auraient autrement pas pu. Cependant, deux inconvénients sont à déplorer :

- premièrement, pour chaque élément de l'interface devant être adaptée, le programmeur doit définir une méthode pour invoquer la méthode de l'objet l'adapté. De telles méthodes sont alors juste des *relais* sans grand intérêt sémantique.
- Deuxièmement, mêmes les méthodes de l'adapté ne nécessitant pas une adaptation doivent être traitées avec le même mécanisme.

Solution. Notre solution est principalement basée sur l'utilisation de l'opérateur `corresponds`. La dépendance définit quels sont les messages devant être adaptés. Notre solution se situe principalement au niveau des instances et est complètement dynamique. Nous comparons notre approche à celle de LayOM en utilisant ses exemples (que nous présentons plus loin).

Voici la définition d'un schéma `Adaptateur`, nommé `adapter`, et sa mise en place sur une instance.

```
(deflink adapter (:obj)
:behavior
   (((mess1 :obj arg1 arg2) corresponds (newMessA :obj arg2 arg1))
    ((mess2 :obj arg1 arg2) corresponds (newMessB :obj arg1))
    ((mess3 :obj arg1 arg2) corresponds (newMessB :obj arg1))))

(make adapter :obj adaptee)
```

La définition de la dépendance `adapter` spécifie que les messages envoyés à un objet lui sont renvoyés après avoir été adaptés. Ainsi le message `mess1` est transformé en `newMessA` et `mes2` et `mess3` en `newMessB`. La figure 10.2 illustre l'action de la dépendance.

Pour offrir un adaptateur au niveau de classes, il suffit de définir une sous-classe de la classe devant être adaptée et de spécifier la création automatique d'un adaptateur sur les instances créées.

```
(define-method make ((cl objetadapté) args)
   (let ((ins (next-method)))
      (make Adapter :obj ins)
      ins))
```

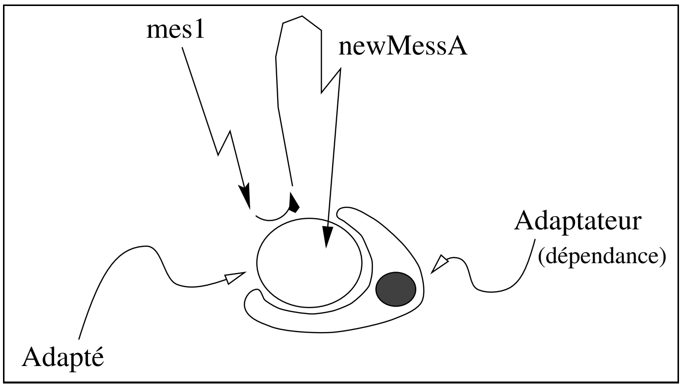


Figure 10.2: Un schéma adaptateur à l'aide de dépendances.

Outre une meilleure abstraction des schémas, avec notre approche seuls les messages nécessitant une adaptation seront sujets à une délégation. De plus, les dépendances favorisent encore une fois une réelle séparation entre les fonctionnalités de l'objet et d'éventuels comportements purement relationnels comme la délégation de messages.

Remarques et comparaison. Les Adaptors [Yell 94] sont une concrétisation du schéma `Adaptateur`. Cependant, les Adaptators intègrent la notion de protocoles que le schéma ne définit pas et la notion de compatibilité d'interfaces.

La solution proposée par LayOM [Bosc 96] est proche de la notre. Elle se situe aussi bien au niveau des instances que des classes. La syntaxe de la *«couche»* *(layer)* `Adapter` de LayOM est la suivante :

```
<i> : Adapter ( accept <mess-sel>+ as <new-mess-sel> , ...);
```

La définition d'un tel schéma peut alors être associée à une classe. Voici par exemple la classe représentant la situation décrite par la figure 10.1

```
 class adapter
     layers
      adapt : Adapter( accept mess1  as newMess1,
                      accept mess2, mess3  as newMessB);
      inh : Inherit(Adaptee);
end;
```

Au niveau d'une instance, il suffit de définir une couche directement sur l'instance :

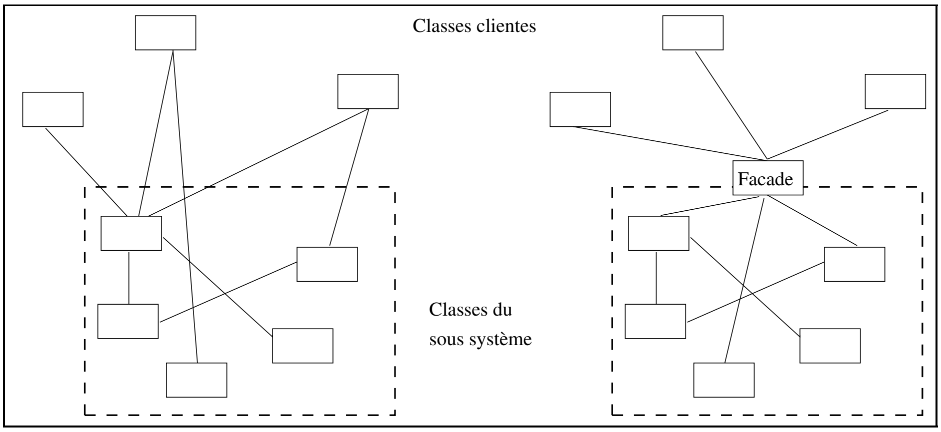
```
...
adaptedAdaptee : Adaptee  with layers
  adapt : Adapter( accept mess1  as newMessA,
                       accept mess2, mess3  as newMessB);
...
```

Il faut remarquer que les couches de LayOM contrôlent à la fois les messages reçus et ceux émis. Ceci permet de définir un adaptateur en spécifiant cette fois une adaptation des messages émis. Le modèle FLO se situe uniquement au niveau des messages reçus.

La solution offerte par la couche `Adapter` en LayOM est cependant moins expressive que notre solution au niveau des possibilités de manipulations des arguments d'appels. En effet, il n'est pas possible d'appliquer des fonctions aux arguments pour par exemple changer le type des arguments (conversion de type), il est seulement possible de spécifier qu'un message d'un certain sélecteur doit être accepté comme un message d'un autre sélecteur.

10.2.2 Façade

But et solution traditionnelle. Le but du schéma `Facade` est d'offrir une interface unifiée pour un ensemble de composants. Ce faisant la façade définit une interface de plus haut niveau d'abstraction rendant les composants plus facile à utiliser. L'utilisation d'une façade réduit les dépendances entre les différents composants [Lieb 89]. L'ensemble des classes représentant ces composants est appelé un *sous système* dans [Gamm 94].



Une des manières de représenter un schéma `Facade` consiste à utiliser une classe ayant pour parties les différentes classes des composants. L'objectif du schéma `Facade` est double : d'une part, il s'agit de coordonner les classes sous sa responsabilité, d'autre part d'offrir une interface homogène aux clients utilisant le sous système. Représenter le sous système par une classe satisfait le premier objectif. Pour le second objectif, on retrouve sensiblement les mêmes problèmes que pour le schéma `Adaptateur`, c'est-à-dire la nécessité de créer des méthodes dans le seul but de renvoyer les messages aux composants du sous système. Outre le nombre important de méthodes, la classe représentant le schéma `Facade` est un peu vide de sens.

Solution. La solution que nous proposons est d'utiliser une dépendance comme un objet représentant la coordination entre plusieurs objets. Une fois encore, nous utilisons l'opérateur `corresponds`. La dépendance `facade` décrite ci-après spécifie que les messages de sélecteur `mess1` et `mess2` envoyés à la dépendance sont renvoyés après une possible adaptation vers l'objet référencé par la variable `:Part01`. De même, le message `mess3` est renvoyé à l'objet `:Part02`.

```
(deflink facade (:Part01 :Part02)
 :behavior
  (((mess1 link a1 an) corresponds (mess1  :Part01 a1 an))
   ((mess2 link a2)    corresponds (mess1  :Part01 a2))
   ((mess3 link a3)    corresponds (mess3  :Part02 a3))))

(make facade :Part01 composant1 :Part02 composant2)
```

Remarquons l'usage de la variable `link` qui fait référence à la dépendance effective même. Ainsi il s'agit de déléguer les messages envoyés à cette dépendance effective qui n'est alors plus transparente du point de vue des objets clients. Ceci est illustré par la figure 10.3.

Certaines variations peuvent être apportées à l'exemple ci-dessus lorsque l'on considère que la façade doit avoir une structure représentant les éléments du sous-système. Pour cela, il suffit de définir des variables d'instance dans la dépendance de la même manière que l'on le ferait pour une classe. Le fragment de code suivant définit deux variables d'instance de dépendances permettant d'avoir une référence structurelle aux objets composant le sous-système.

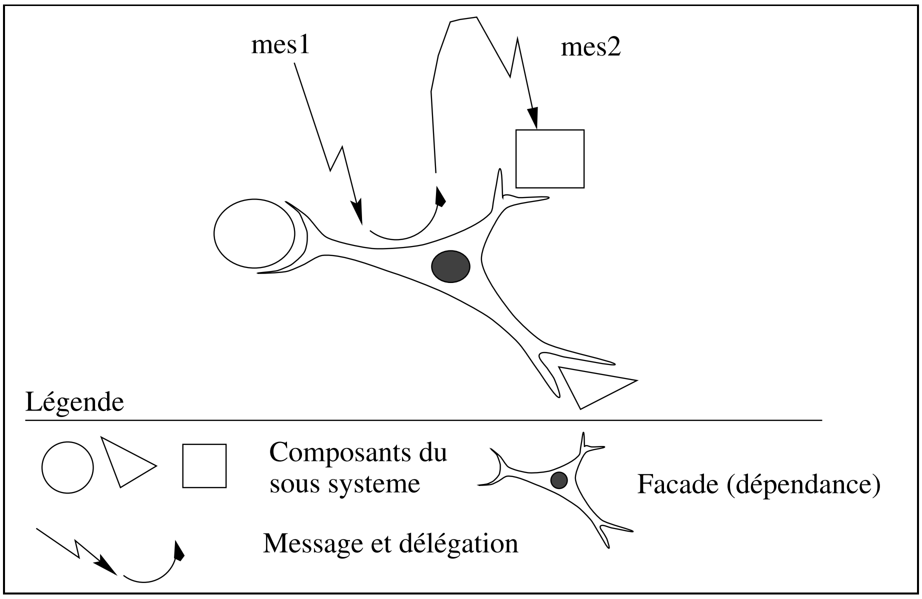


Figure 10.3: Une dépendance pour implémenter un schéma Facade.

```
...
 :var '((Part01)
        (Part02))
...
```

Cette solution possède principalement deux avantages : d'une part l'implémentation correspond exactement à la conception du système, d'autre part la séparation entre les classes «pures» et la représentation des aspects relationnels entre celles-ci est claire. Il ne s'agit plus d'avoir des classes dont les seules fonctionnalités sont de rediriger des messages.

Comparaison. La solution proposée par J. Bosch est très voisine de la notre. Il définit pour cela une nouvelle couche `Facade` permettant de spécifier la redirection de messages :
`<id> : Facade( forward <mess-sel>+  to <object>, ...);`

Cette nouvelle couche permet alors d'implémenter des façades :

```
class facade
   layers
    face : Facade ( forward mess1, mess2  to Part01
                    forward mess3  to Part02);
    part01 : PartOf(Classof01);
    part02 : PartOf(Classof02);
...
```

Il faut noter que, contrairement à la solution adoptée par LayOM, notre solution permet de manipuler les messages délégués. En cela nous considérons un schéma `Facade` comme une généralisation du schéma `Adaptateur` si l'on exclue la place différente accordée à la dépendance.

10.3 Schémas comportementaux

Les schémas comportementaux *(behavioral design patterns)* décrivent la coopération et les interactions entre objets. Ces schémas ne se contentent pas de définir les schémas structurels entre classes et objets mais aussi des schémas de communication entre ces objets.

10.3.1 Observateur

But et solution traditionnelle. Le but du schéma `Observer` est de définir une dépendance entre plusieurs objets de telle sorte que lorsqu'un objet, appelé le sujet change d'état ses dépendants soient avertis et adaptés de manière automatique.

Un des exemples les plus célèbres de ce schéma est le modèle MVC de SMALLTALK [KRAS 88]. Ce schéma est implémenté en SMALLTALK grâce à un mécanisme de dépendances. Nous présentons ce mécanisme et ses inconvénients dans l'annexe A, aussi pour éviter des répétitions résumons nous seulement la solution proposée. La figure 10.5 illustre la solution proposée pour l'implémentation traditionnelle de ce schéma : lorsqu'un objet, le sujet, change d'état, il avertit ses dépendants (méthodes `notify` ou `changed:` en SMALLTALK). Cette notification a pour conséquence l'invocation des méthodes `update` de chaque dépendant. Ces méthodes interrogent éventuellement le sujet pour avoir l'information nécessaire à leur remise à jour (méthode `GetState`) (voir figure 10.4).

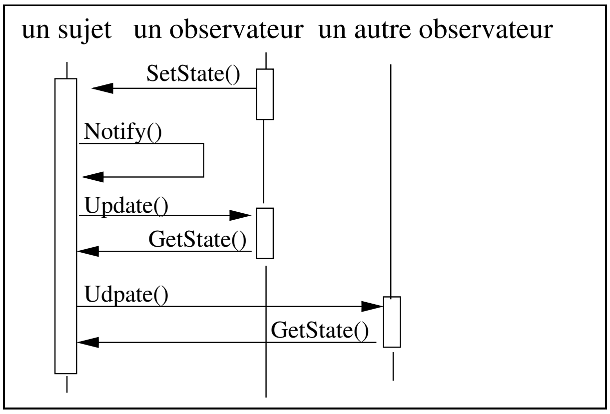


Figure 10.4: Séquence de messages dans le schéma Observateur

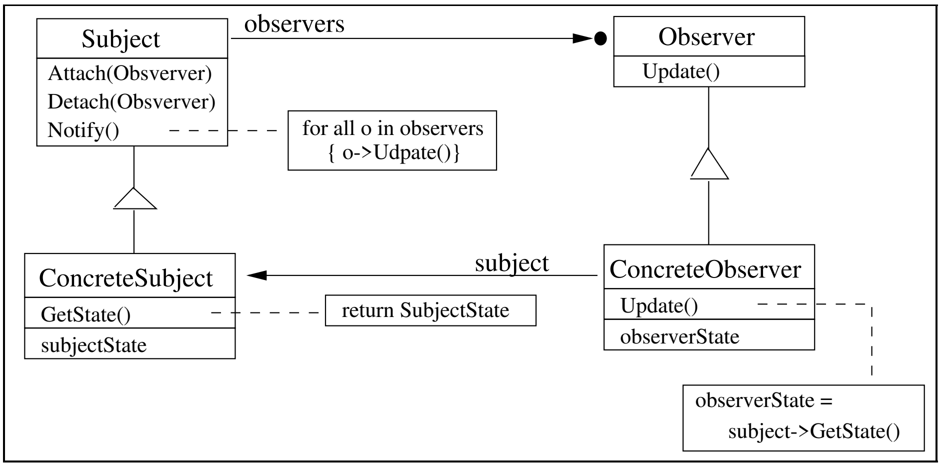


Figure 10.5: Relation structurelle entre les classes des objets participant à un schéma observateur

Solution. Nous représentons un tel schéma par une dépendance dont le rôle est d'assurer l'émission automatique des messages de mises à jour. De part la sémantique de nos dépendances, les méthodes de notification (`notify()`) et de mise à jour sont inutiles (`update()`) : une dépendance est directement exprimée à l'aide des méthodes modifiant l'objet. Différentes versions peuvent être proposées, nous montrons ici la plus simple et immédiate correspondant au schéma tel que décrit dans [GAMM 94]. Cependant, nous renvoyons le lecteur à la lecture de la dépendance *est-représentée-par* qui représente un schéma observateur entre l'objet et sa représentation (voir en 5.1.2).

```
(deflink observer  (:subject:observers)
 :behavior
  (((SetState :subject val)
              implies (map Update :observers (fct val)))))

(make observer:subject suj1:observers (list obs1 obs2 obs3))
```

Dans cette définition, la fonction `map` désigne un itérateur. De plus, outre le fait que la dépendance permet la manipulation des arguments d'appels des méthodes, elle est l'endroit idéal pour définir des conversions de types, d'unités... entre les valeurs des arguments.

Cet exemple n'est qu'un cas particulier des possibilités d'expressions offertes par les dépendances. En effet, il illustre exclusivement un changement de valeur de variables d'instance, mais comme nous l'avons déjà montré en détail lors des chapitres précédents, l'utilisation d'une dépendance permet de spécifier directement les méthodes des objets participants. De plus, l'utilisation de dépendances ne nécessite pas la modification des méthodes afin d'introsuire de messages de notification.

10.3.2 Médiateur

L'essence même des dépendances de FLO étant la prise en compte des interactions entre objets, leur utilisation pour implémenter le schéma `Mediator` est immédiate.

But. Le but de ce schéma est définir un objet, nommé *médiateur*, encapsulant l'interaction entre un ensemble d'objets, nommés *collègues*. Ce schéma fait diminuer le couplage qui existe entre les différents objets en permettant à ces objets de ne plus se référencer directement. Les collègues ne font plus référence qu'au médiateur. Aussi, au lieu d'envoyer des messages directement à ces collègues, un objet envoie des messages au médiateur qui lui le renvoie aux objets concernés. Ce schéma permet de modifier facilement le comportement de l'interaction. La figure 10.6 illustre les dépendances entre les différentes classes pour l'implémentation d'un tel schéma.

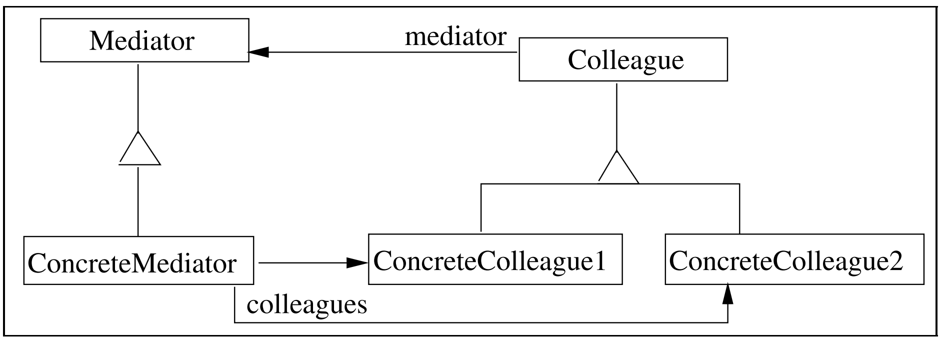


Figure 10.6: Relations structurelles traditionnelles entre les classes des objets participant à un schéma médiateur

Nous retrouvons pour ce schéma des inconvénients similaires à ceux montrés pour les schémas `Adapter` ou `Facade` : méthodes implémentant de simples redirections de messages et une diffusion du schéma.

Solution. Notre solution s'inspire de celles que nous avons proposées pour les schémas Adapter, Facade et Observer. Nous n'avons pas *a priori* de solution spécifique pour ce schéma. Afin de comparer notre solution à celle de LayOM pour ce schéma, nous commençons donc par la définition de la couche Mediator de LayOM:

```
<id>: Mediator (forward <mess-sel>+ from <client> to <object> ...);
```

La sémantique de cette nouvelle couche est qu'un message appartenant à l'ensemble <mess-sel> des messages envoyés par un objet client est renvoyé à un autre objet. LayOM offre la possibilité de spécifier que n'importe quel objet pourra envoyer de tels messages grâce au mot clé **Any**. La définition suivante permet de spécifier que les messages de sélecteur **mesA** provenant de n'importe quel objet sont renvoyés à l'objet ConcColleague1, de même les messages de sélecteur **mesB** envoyés par l'objet ConcColleague1 sont renvoyés à l'objet ConcColleague2.

```
class ConcreteMediator
  layers
    med: Mediator(forward messA from Any to ConcColleague1,
                  forward messB from ConcColleague1
                                 to ConcColleague2);
...
end;
```

Il faut bien noter qu'une couche en LayOM est étroitement liée à une classe contrairement aux dépendances proposées en FLO. Aussi les messages renvoyés dans la définition de la classe ConcreteMediator sont ceux envoyés au médiateur. FLO offre des possibilités plus riches. En effet, suivant les besoins du programmeur les objets collègues peuvent ne pas avoir besoin d'envoyer de message au médiateur. Dans une telle éventualité, une dépendance équivalente s'écrirait en faisant directement référence aux méthodes appelées sur les objets collègues à la manière de l'implémentation du schéma Observer. Ainsi supposons que deux collègues aient à interagir, l'un devant faire l'action A suite à l'action B de l'autre. Nous écririons par exemple :

```
(deflink mediator  (:CColleague1 :CColleague2)
:behavior
   (((B :CColleague1 a1 an) implies (A :CColleague2 a1))))

(make mediator :CColleague1 obj1 :CColleague2 obj2)
```

Lorsque le programmeur souhaite écrire une implémentation plus proche du schéma de conception, c'est-à-dire que les objets collègues appellent l'objet médiateur, la solution la plus proche de celle de J. Bosch est la suivante :

```
(deflink mediator   (:CColleague1 :CColleague2)
  :behavior
  (((messA  link b1 b2) corresponds (messA  :CColleague1 b1 b2))
   ((messB  link a1 an) corresponds (messB  :CColleague2 a1 an))))

(make mediator :CColleague1 obj1 :CColleague2 obj2)
```

Comme cet exemple le montre, FLO ne permet pas de désigner l'objet émetteur d'un message et donc de spécifier des interactions étant basées sur ce critère.

10.4 Conclusion

Notre discours n'est pas de présenter les dépendances comme la solution unique pour l'implémentation des schémas de conception. En effet, nous pensons qu'elles ne sont une réponse intéressante que pour l'implémentation de certains schémas structurels ou comportements. Elles offrent une meilleure lisibilité que les implémentations traditionnelles pour l'implémentation des schémas qui ne sont plus alors diffusés dans les classes. D'autre part, elles contribuent à une meilleure séparation entre objets et interactions et évitent la définition de classes *«creuses»*. Il serait intéressant d'étudier quels sont les critères des schémas permettant d'affirmer que l'utilisation de dépendances est adaptée.

De plus, cette utilisation des dépendances nous a confronté à certaines limitations qui devraient faire l'objet d'améliorations futures. D'une part, la syntaxe des interactions est assez rustre, ajouter un peu de sucre syntaxique permettrait une écriture plus compacte de certaines dépendances. D'autre part, la notion d'objet émetteur n'est pas traitée ce qui nous empêche d'exprimer certaines situations.

Alors que les dépendances que nous proposons peuvent être vues comme la composition de plusieurs schémas de conception, une question se pose : les schémas de conception proposés dans la littérature ne sont-ils pas prisonnier du modèle objet offert par les langages traditionnels ? Dit autrement, nous sommes persuadés que l'ajout de dépendances dans un modèle objet a des répercussions sur la façon même de concevoir les applications. En cela, nous allons dans le même sens que Rumbaugh quand il dit que l'utilisation d'associations explicites modifient la façon de penser des concepteurs. Dans cette optique, nous pensons qu'un prolongement à cette thèse devrait être l'élaboration d'une méthode de conception adaptée à ce nouveau moyen d'expression.

L'implémentation des schémas de conception rejoint complètement le besoin de pouvoir décrire au sein même du langage des interactions de première classes entre objets. En cela, les dépendances ne doivent plus être perçues comme exprimant uniquement des dépendances entre objets comme dans OTHELO mais des interactions complexes.

11

Des dépendances pour la construction d'interfaces

« The definition of an appropriate software architecture is an important problem in user interface design. Without an adequate architectural model, the resulting software may be hard to modify and maintain. Even worse, it may be unable to take the user's characteristics into account. » [Cout 89].

Les interactions entre humains et ordinateurs ont fait l'objet de nombreux travaux aussi bien en psychologie cognitive qu'en informatique. Les modèles de la psychologie cognitive tels que ACT* [Ande 83] et la théorie de l'action [Norm 86] offrent un moyen de comprendre la nature des processus cognitifs impliqués dans les interactions homme-machine. D'autres modèles permettent de prédire les performances humaines. Cependant, ces modèles ne conduisent pas à des définitions directement utilisables pour l'implémentation des systèmes interactifs.

Le but des modèles informatiques est d'offrir aux développeurs des outils pour élaborer des applications interactives. Cependant, le fossé existant entre les techniques de modélisation du comportement humain en psychologie cognitive et les formalismes et techniques informatiques complexifie la construction des interfaces utilisateurs. L'utilisation de modèles architecturaux[1] aide alors la conception d'applications interactives : il guide la décomposition de l'application, précise les différents éléments composants la constituant et spécifie comment ces composants coopèrent.

Dans ce chapitre, nous présentons rapidement les principaux modèles architecturaux : *«linguistique»* et multi-agents tels que les modèles PAC, ALV ou MVC. L'implémentation d'applications à l'aide de ces modèles n'est pas immédiate et pose des problèmes, nous présentons donc comment l'utilisation de dépendances offre des solutions à ces problèmes. Nous montrons comment les dépendances définissent de fait le modèle ALV et permettent d'implémenter le PAC. L'utilisation de dépendances pour implémenter le modèle PAC nous amène à une nouvelle version épurée de ce modèle. Nous finissons en proposant une modélisation des contrôleurs de dialogue dans le cadre de la resysnchronisation d'interfaces.

[1]Un modèle architectural est une méthode de structuration indépendante d'un langage donné.

11.1 Le modèle linguistique

Les premiers modèles comme celui de Seeheim [Pfaf 85, Gree 85] ou de Foley [Fole 84] traitent l'interaction homme-machine en s'inspirant du langage entre individus. Le dialogue est basé sur un langage commun qui est défini suivant trois niveaux : sémantique, syntaxique et lexical.

Principes. Le modèle Seeheim repose sur la décomposition d'une application interactive en trois parties : la présentation, le contrôle du dialogue et l'interface avec l'application. La présentation traite du niveau lexical du dialogue, le contrôle gère le niveau syntaxique et le niveau sémantique l'interface avec l'application logique (voir figure 11.1). Les informations échangées entre ces trois composants sont sous la forme de *tokens*.

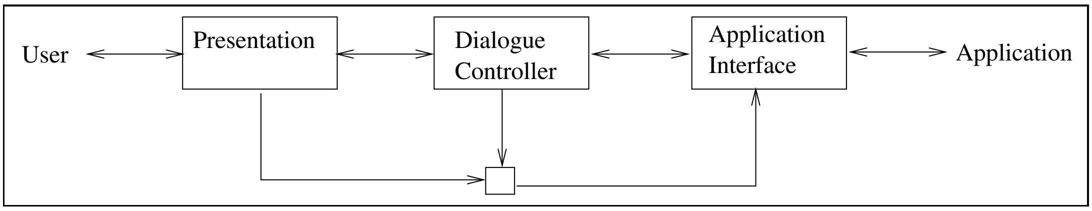


Figure 11.1: Organisation d'une application interactive suivant le modèle Seeheim.

- La présentation (niveau lexical) est responsable de l'apparence physique de l'interface (gestion de l'écran, de l'affichage des informations, clavier, souris). Elle capte les événements utilisateurs sur les dispositifs d'entrée et les traduit dans le langage de communication des autres composants.
- Le contrôleur de dialogue (niveau syntaxique) réalise la connexion entre l'interface de l'application et la présentation. Il convertit les informations provenant de la représentation en des structures représentant les commandes de l'utilisateur et les communique à l'application. De manière inverse, il convertit les informations provenant de l'application vers les différents éléments composant la présentation.
- L'interface avec l'application (niveau sémantique) modélise les concepts qu'un utilisateur et un système peuvent utiliser durant une interaction. Elle offre une description claire de l'application.

Avantages et Inconvénients. Le modèle linguistique possède trois avantages :

- L'architecture proposée est simple et peut facilement être mise en œuvre durant la conception d'applications.
- La modularité du modèle permet de modifier un des composants sans remettre en cause le reste de l'application.
- Le modèle est général et ne fait pas d'hypothèse concernant la nature de l'application ou des outils pour l'implémenter.

Les principaux inconvénients de ce modèle sont (voir [Cout 89] et [ten 90] pour plus de détails) :

- La centralisation des processus sémantique, syntaxique et lexicaux n'est pas compatible avec l'interactivité [Cout 89]. L'expérience montre que la frontière entre l'application (sémantique) et l'interface utilisateur (lexical et syntaxique) n'est pas toujours claire.
- La distinction, réalisée dans certaines applications [Fole 84], entre les langages utilisés par l'utilisateur et ceux utilisés par le système pour produire des sorties empêche de réutiliser celles-ci comme de nouvelles entrées.

- La centralisation du modèle pose des problèmes pour la conception d'applications distribuées.
- Le modèle Seeheim ne prend en compte qu'une interaction simple : il ne gèrent pas plusieurs événements simultanés [TEN 90].
- Le modèle Seeheim gère mal les phénomènes de surcharge. Ainsi alors qu'une sélection à la souris peut avoir des sens dépendants du contexte, il ne propose pas de solution claire pour les définir.

Le modèle Arch. Le modèle Arch et le méta-modèle Slinky [BASS 91] proposent une mise à jour du modèle Seeheim. Il est constitué par cinq composants (voir figure 11.2). Cette architecture met en avant la portabilité et la réutilisabilité des composants de l'application.

La *boîte à outils* offre un ensemble d'objets interactifs (widgets). Le composant de *présentation* assure la médiation entre le contrôleur de dialogue et la boîte à outils d'interaction. Il doit fournir au dialogue un ensemble d'objets indépendants d'une boîte à outils particulière. Le *contrôleur de dialogue* est la clé de voûte du système. Il fait la correspondance dans les deux sens entre les formalismes du noyau fonctionnel et de l'interface. Il assure la cohérence entre le noyau fonctionnel et la présentation. Le composant *adaptateur de domaine* assure la médiation entre le dialogue et le noyau fonctionnel. Il lance les tâches initiées par le noyau fonctionnel, réorganise les données du domaine et détecte les erreurs de sémantique. Le *noyau fonctionnel* fournit les fonctionnalités indépendantes de l'interaction. Ce composant est le symétrique de la boîte à outils.

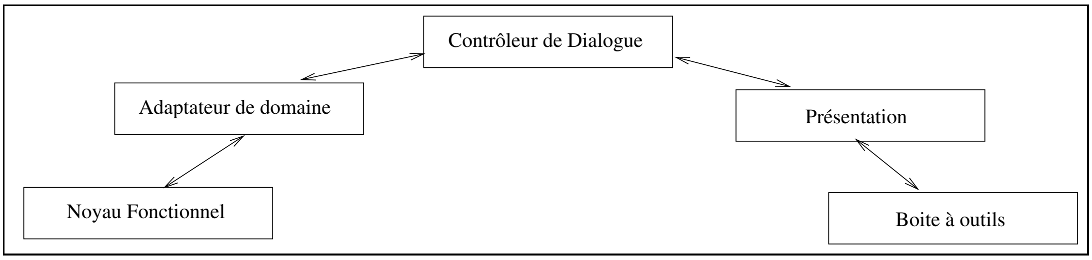


Figure 11.2: Organisation d'une application interactive suivant le modèle Seeheim.

11.2 Modèles Multi-Agents

D'un autre côté, plusieurs modèles multi-agents ont été définis : PAC, MVC ou ALV. Dans ces modèles, les applications sont organisées comme des ensembles d'agents réagissant à un ensemble d'événements extérieurs et engendrant des événements. Idéalement, un agent détient toutes les informations pour gérer les événements. Il possède alors un état et des fonctions associées, un processeur associé gérant les événements. Un agent est autonome et responsable de son état.

Ces modèles opposent une organisation modulaire fortement parallèle au modèle linguistique. Ainsi la centralisation de la gestion de la présentation et de l'état de l'interaction du modèle linguistique sont distribués parmi les agents. Cette modélisation est particulièrement adaptée aux applications distribuées. Ces modèles permettent une conception incrémentale : un agent peut être modifié sans modifier tout le système. La granularité de la modularité est plus fine que dans le modèle linguistique.

Un agent modélise une interaction. Cependant lorsque celle-ci devient trop complexe, cette interaction est décomposée et associée à un ensemble d'agents coopérants. Cette coopération est gérée différemment suivants les modèles multi-agents.

11.2.1 Le modèle PAC

Le modèle PAC [COUT 89] structure une application en trois parties : la Présentation qui définit les entrées et les sorties de l'application, l'Abstraction qui contient le noyau fonctionnel du système et le Contrôle qui maintient la cohérence entre l'Abstraction et la Présentation.

Cette triade se distingue de la décomposition proposée par le modèle linguiste par le fait que la Présentation est implémentée comme un ensemble d'agents spécialisés dans l'interaction, nommés *objets interactifs*. De tels agents sont également structurés de la même manière (voir la figure 11.3). La Présentation est un objet graphique représentant un réchaud. L'Abstraction définit les données et fonctionnalités d'un réchaud (une valeur représente la chaleur, une autre si le réchaud est allumé et une troisième si un objet est posé sur le réchaud). Le Contrôle maintient la cohérence entre l'Abstraction et sa Présentation. Il définit la conversion des données d'une représentation à une autre. Ainsi lorsque le réchaud est mis en route en sélectionnant l'interrupteur de la Présentation, le Contrôle modifie en conséquence l'état de l'Abstraction.

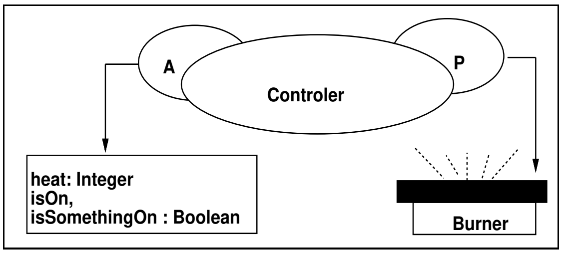


Figure 11.3: Un agent interactif du modèle PAC.

Un objet interactif peut être élémentaire ou composite. Un objet composite est structuré comme un agent dont le comportement dépend de lui-même et de ses composants. La Présentation et l'Abstraction d'un objet composite dépend à la fois de celles de ses composants et des informations définies à son niveau. De plus, son Contrôle gère la collaboration ou les dépendances entre composants.

Hiérarchie PAC. Une hiérarchie PAC décompose un système interactif en objets composites PAC. Au sommet de la hiérarchie, l'Abstraction correspond au noyau fonctionnel, la Présentation organise la visualisation de celui-ci et le Contrôle assure la communication entre elles (voir figure 11.4.

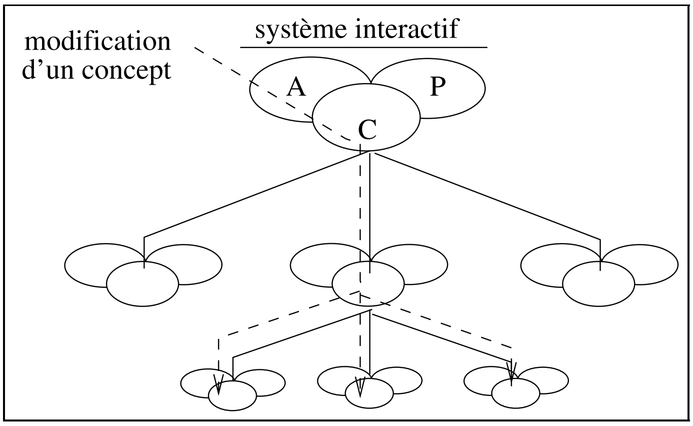


Figure 11.4: Décomposition d'un système interactif : un objet composite PAC.

Un rôle du Contrôleur (principal) du sommet de la hiérarchie est de lier les concepts du noyau fonctionnel avec les facettes abstraites des objets interactifs. Ainsi quand le noyau fonctionnel modifie les valeurs d'un concept, le Contrôleur principal convertit cette modification dans un formalisme compris par les facettes abstraites des objets interactifs, il envoie ces informations aux Contrôleurs plus bas dans la hiérarchie. Ces Contrôleurs propagent à leur tour les modifications à leur représentation ainsi qu'aux autres objets dépendants.

Inversement, une action d'un utilisateur est interprétée par la Présentation d'un objet interactif. Le Contrôleur local traduit cette interaction dans un formalisme et lance une propagation ascendante vers le noyau fonctionnel par une communication entre Contrôleurs.

Discussion. L'analyse du modèle PAC nous amène à faire deux remarques.

- Tout d'abord, la décomposition en objets PAC d'une application interactive n'est pas aussi récursive qu'il n'y paraît au premier abord. En effet, les Abstractions des objets élémentaires sont des parties (des variables) du noyau fonctionnel. Une hiérarchie PAC est une coopération d'agents dont les abstractions réfèrent l'abstraction du sommet de la hiérarchie. Un agent PAC explicite alors de manière locale comment une partie de l'Abstraction participe à l'interaction. Dans ce cas, nous ne considérons plus un objet interactif comme un agent autonome.
- La décomposition récursive se veut uniforme. Cependant, cette uniformité est mise à mal lors de l'utilisation du modèle. De nombreux objets interactifs ne définissent pas d'abstraction ou plus souvent de représentations.

11.2.2 Le modèle MVC

Le modèle MVC défini initialement comme le modèle architectural de SMALLTALK [GOLD 83, KRAS 88, BRIF 96] est basé sur la définition de triades (Modèle, Vue et Contrôleur) pour chaque élément de l'application participant à l'application. Ce modèle est utilisé par de nombreux systèmes d'interfaces (Andrew Toolkit [BORE 90], MVC++ [JAAS 95]).

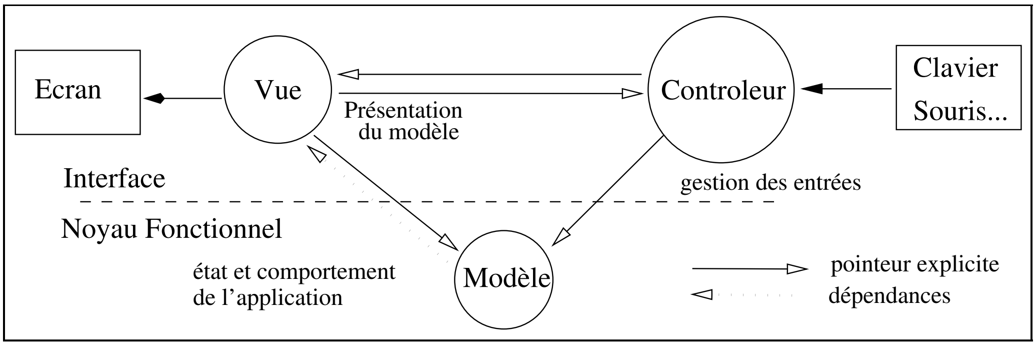


Figure 11.5: Organisation d'une application interactive suivant le modèle MVC.

Principe. Etant donné une triade : le Modèle représente une abstraction ; la Vue représente la visualisation de cette abstraction et le Contrôleur traite les entrées de l'utilisateur et les convertit en messages au Modèle. Contrairement au Contrôleur du modèle PAC, le Contrôleur du modèle MVC ne spécifie pas la cohérence entre une Abstraction et sa Présentation.

La Vue et le Contrôleur d'une triade sont étroitement liés. Ils se connaissent explicitement. De même, la Vue et le Contrôleur connaissent explicitement leur Modèle (voir figure 11.5). Pour des raisons de réutilisabilité et de modularité, le modèle ne connaît explicitement la Vue qui lui est associée. La Vue est dépendante du Modèle. Ainsi à chaque fois que le modèle change, la Vue qui lui est associée est remise à jour en utilisant les dépendances fonctionnelles. Plusieurs représentations d'un même modèle peuvent être mise en place sans que le modèle ne soit modifié.

Discussion. Cette modélisation pose des problèmes qui sont liés principalement à l'utilisation des dépendances. Nous renvoyons le lecteur à la lecture du mécanisme des dépendances en annexe A.3. Principalement, la Vue et le Contrôleur doivent connaître la structure interne du Modèle pour savoir comment réagir. Les classes de Modèle et des Vues sont liées ce qui va à

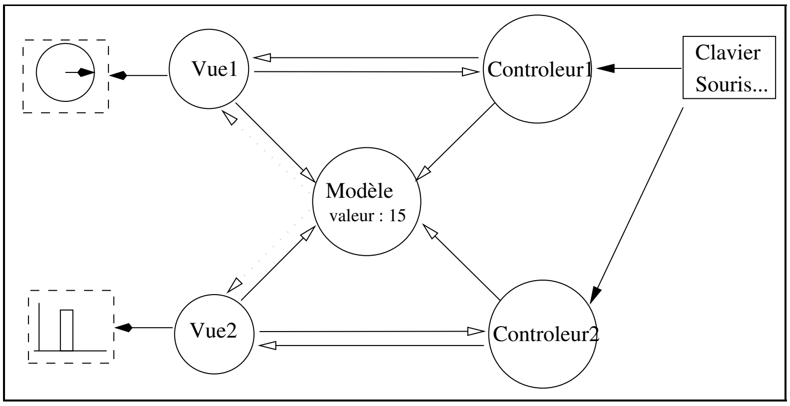


Figure 11.6: Un même objet représenté sous deux aspects différents.

l'encontre de la philosophie initiale du modèle. Des raffinements de ce modèle ont été proposés en introduisant les ValueHolder et les DependencyTransformer pour apporter des réponses à ces problèmes [Par 94, Brif 96].

La boîte à outils Interviews [Lint 89] ou ET++ [Wein 88] utilisent également la notion de dépendance dans leur implémentation de composants interactifs. Le modèle architectural est modifié : il n'y a plus de séparation entre les vues et les contrôleurs pour réduire les coûts de communication.

11.2.3 Le modèle ALV

Le modèle ALV *(Abstraction-Link-View)* [Hill 92] est le modèle architectural utilisé dans le langage objet Rendezvous [Hill 93b, Hill 94]. Comme le modèle PAC, il est un des rares modèles à expliciter le rôle du dialogue entre une Abstraction et sa Présentation.

Pour une triade ALV : l'Abstraction représente un objet du noyau applicatif, la Vue sa représentation et le Lien maintient la cohérence entre ces objets. Des contraintes sont utilisées pour spécifier ce maintien entre variables de l'Abstraction et de la Vue. Les liens sont utilisés comme outils de programmation (voir en annexe B.5). Ils constituent les seuls moyens de communication entre l'Abstraction et ses Vues.

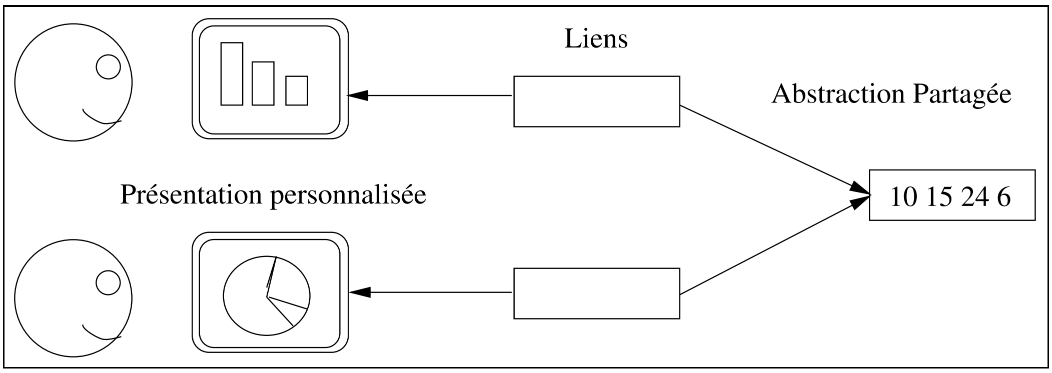


Figure 11.7: ALV le modèle architectural de RendezVous : Abstraction-Link-View

Une application est structurée sous la forme de différentes hiérarchies : hiérarchie des objets de l'Abstraction et hiérarchie des objets de Visualisation. Les liens assurent la cohérence entre certains éléments de ces hiérarchies (voir au chapitre 2). La figure 11.8 illustre cette situation dans le cas d'un jeu de Tic-Tac-Toe. Les liens assurent la cohérence du plateau de jeu ainsi que la cohérence entre les cellules composant le jeu.

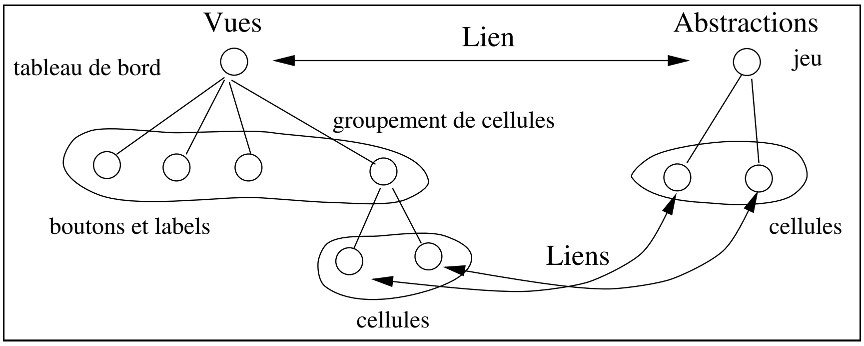


Figure 11.8: Hiérarchies et dépendances du Tic-Tac-Toe.

Le modèle ALV offre les possibilités suivantes pour la construction d'interfaces multi-utilisateurs : synchronisation des vues (des vues identiques pour les utilisateurs), personnalisation des vues (chaque utilisateur perçoit différemment la même abstraction), liberté d'action, prise en compte dynamique de nouveaux utilisateurs...[Hill 92].

11.3 Dépendances : une implémentation immédiate du modèle ALV

La séparation entre les objets et leurs représentations est un aspect important de la conception d'interfaces. Les dépendances en assurant un maintien de cohérence entre objets de manière explicite et disjointe des objets, sont un outil adapté pour l'implémentation d'interfaces utilisateurs. Les dépendances permettent une implémentation immédiate[2] du modèle ALV. Une dépendance est l'unique moyen de communication entre un objet applicatif et sa ou ses représentations.

Exemple. Nous avons implémenté à l'aide de FLO, une des applications chère aux auteurs de Rendezvous: un Tic-Tac-Toe. Dans notre version, le noyau fonctionnel représente le Tic-Tac-Toe. Il est possible d'y jouer en envoyant des messages aux objets en utilisant l'interprète de FLO. Aucune interface n'est alors prévue. Deux classes définissent le jeu : la classe `Tic` représente le jeu : elle contient une liste de cellules, un nombre indiquant quel est le prochain joueur ainsi que les scores respectifs des deux joueurs). Elle définit les méthodes `next-turn`, `one-more-turn`, `restart`, `reset` et `win`. La classe `Logical-Cell` représente une cellule (une position et un état). Elle définit les méthodes `mark-cell`, `reset` et `three-colinear?`.

Ajouter une interface graphique à ce jeu consiste à définir principalement des objets spécialisés représentant les cellules et l'état du jeu (le joueur courant...) et les dépendances entre ces objets graphiques et les objets modélisant le jeu (voir figure 11.9).

Deux principales dépendances doivent être définies : une dépendance *board-to-tic* entre le jeu et l'objet graphique le représentant. Cette dépendance gère en particulier le maintien de la cohérence des informations relatives au jeu (joueur courant, remise à jour du jeu...). La dépendance *cell-to-cell* maintient la cohérence entre l'état des cellules logiques et leurs représentations.

[2] La principale différence provient du fait que les dépendances sont définies en utilisant des messages alors que les liens de ALV sont exprimés en termes de changement d'état.

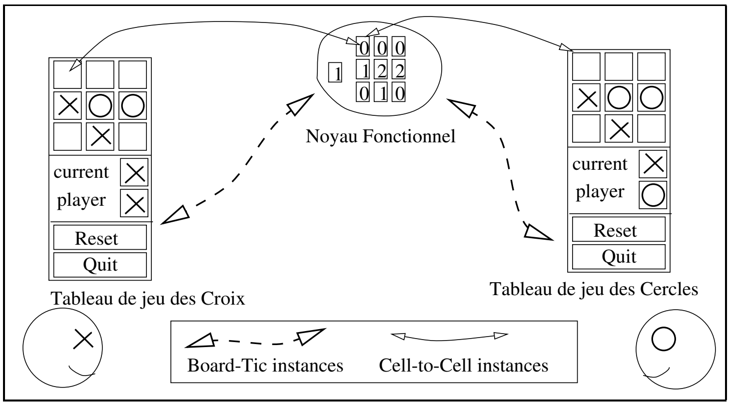


Figure 11.9: Architecture du Tic-Tac-Toe : une abstraction partagée, deux présentations personnalisées et des dépendances.

```
(deflink board-to-tic  (:logical :graphic)
  :inherit '((no-reenter-link))
  :behavior
   '(((reset :graphic) implies (reset :logical))
     ((reset :logical) implies (reset :graphic))
     ((set! current :logical val) implies (set! current :graphic val))))

(define b1totic  (make board-to-tic :logical master-tic :graphic bord1))
(define b2totic  (make board-to-tic :logical master-tic :graphic bord2))
```

Remarques. FLO, étant une mise en œuvre immédiate du modèle ALV offre aux concepteurs d'applications interactives tous les avantages de ce modèle : décomposition et séparation des composants, communication explicite, maintien de la cohérence automatique...[Hill 92] La création dynamique de dépendances permet d'offrir des applications multi-utilisateurs prenant en compte l'arrivée de nouveaux utilisateurs.

Cependant, FLO permet une expression plus riche des dépendances entre objets. En effet, les dépendances exprimées en termes de changement de variables ne sont qu'un cas particulier de méthodes. De plus, les dépendances de FLO proposent une expression de l'interaction entre objets plus riche et ne se limite pas à des mécanismes de propagations. .

11.4 Relecture du modèle PAC à l'aide de dépendances

Nous proposons une nouvelle vision du Contrôle du modèle PAC dans un cadre homogène[3]. Dans notre proposition, chaque facette d'un agent PAC est un objet autonome [Dery 96b]. L'Abstraction est limitée à un objet de l'application et la Présentation à un objet de l'interface. Le Contrôle initial du modèle PAC est raffiné en un Contrôle lié à l'Application (contrôle entre objets de l'Application), un Contrôle lié à la Présentation (contrôle entre objets de la présentation) et un Contrôle lié au Dialogue (contrôle entre un objet applicatif et un objet de présentation).

Avant de montrer comment cette approche transforme une hiérarchie PAC en des réseaux d'objets applicatifs et de présentation reliés entre eux par des contrôleurs, montrons qu'une dépendance est un Contrôleur PAC.

[3] L'abstraction, la Présentation et le Contrôle sont implémentés dans le même langage : ici FLO.

11.4.1 Un contrôleur PAC = une dépendance entre objets

Le modèle PAC reste essentiellement un *modèle* de structuration d'applications. L'implémentation naturelle d'un agent PAC est basée sur trois objets. Cependant, pour assurer le maintien de la cohérence, l'utilisation de langages classiques force à regrouper les trois parties en un ou deux objets. Dans cette solution le Contrôle est alors enfoui dans les autres objets (attributs, valeurs actives...). Ce faisant la modélisation se trouve modifiée lors de l'implémentation. L'utilisation des dépendances du langage FLO évite cet eccueil et permet de passer directement de la modélisation à l'implémentation. Une dépendance en assurant la communication et la cohérence entre objets implémente un Contrôleur.

Exemple. Nous reprenons ici la modélisation proposée par J. Coutaz dans [Cout 89] de l'agent représentant le réchaud (voir la figure 11.3). Le réchaud définit les méthodes `isOn?`, `on`, `off`, `computeHeat`. Lorsque le réchaud est allumé, la méthode `computeHeat` calcule la chaleur en fonction de la durée d'allumage. Cette méthode est appelée à intervalles de temps réguliers par l'objet réchaud. La Présentation du réchaud définit les méthodes `select-switch-on`, `select-switch-off` et `show-flames`. La Présentation est responsable de son comportement intrinsèque (changer la couleur de l'interrupteur après sélection...). Le contrôleur doit assurer entre l'Abstraction et la Présentation le comportement que nous représentons par la dépendance *burner-presentation-control*.

Lorsque l'on ouvre l'interrupteur, la Présentation indique ce changement d'état par un changement de couleur. Le Contrôleur doit répercuter cette modification à l'Abstraction. La ligne 3 spécifie donc que la réception d'un message `select-switch-on` implique d'envoyer le message `on` au réchaud.

De la même manière, fermer l'interrupteur conduit à éteindre le réchaud (ligne 4).

La Présentation en dessinant des effluves de différentes tailles représente la chaleur produite par le réchaud. Chaque nouveau calcul de la chaleur implique de redessiner les effluves (lignes 5 et 6).

```
1(deflink burner-presentation-control (:BInterface :Babstract)
2 :behavior
3  (((select-switch-on :Binterface)  implies (on :Babstract))
4   ((select-switch-off :Binterface) implies (off :Babstract))
5   ((computeHeat :Babstract)
6        implies (show-flame :BInterface (convert result)))))
```

L'utilisation de dépendances pour implémenter des Contrôleurs PAC offre les avantages suivants : l'implémentation correspond exactement à la modélisation, les composants d'un agent sont réutilisables et le contrôle peut être dynamique (une instance de dépendance peut être créée ou détruite dynamiquement).

11.4.2 Une nouvelle vision des hiérarchies PAC

L'existence d'une hiérarchie PAC est motivée par le refus de références explicites entre Abstractions et Présentations. Le seul moyen de communication est le Contrôleur. Une hiérarchie PAC est alors dirigée par le contrôle ; c'est-à-dire que la hiérarchie représente la communication entre objets et entre contrôleurs.

Le problème est que certains contrôleurs d'une hiérarchie n'existent non plus pour lier un objet applicatif et sa représentation mais simplement pour régir des contraintes entre contrôleurs. D'autre part, certains contrôleurs spécifient seulement des contraintes entre objets applicatifs. Nous considérons que les agents dont une des deux parties (abstraction ou présentation) est absente, reflètent un problème de modélisation.

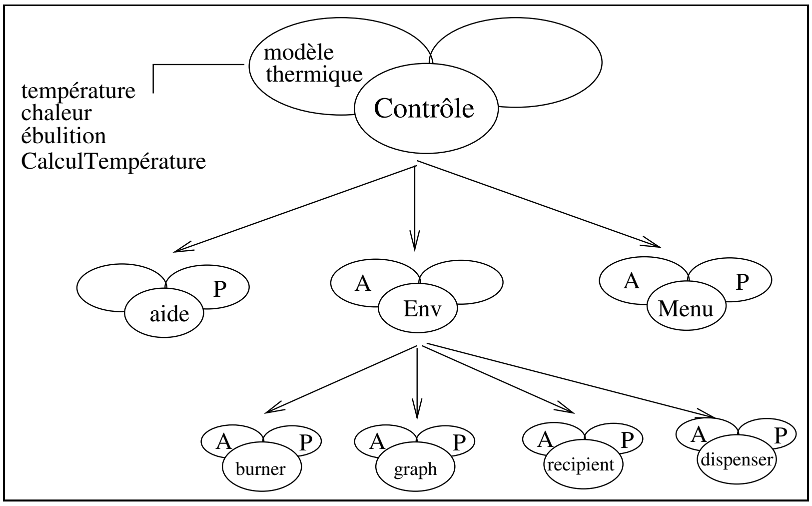


Figure 11.10: La hiérarchie PAC de l'exemple du réchaud.

Commentons la hiérarchie de l'exemple Thermo [Cout 89, Cout 90] illustré par la figure 11.10. Tout d'abord, la Présentation du sommet de la hiérarchie est vide. Le rôle de l'agent nommé Env est de faire respecter les lois d'environnement entre les différents objets : le réchaud attire le récipient, le réchaud repousse le dispenser d'eau, le dispenser attire le récipient... Cet agent n'a pas de Présentation, il modélise et contrôle les mouvements d'autres objets. Le rôle de cet agent n'est pas clairement exprimé, son comportement est assez complexe. Par exemple, quand un utilisateur bouge un objet graphique, Env vérifie si l'objet entre dans une zone d'attraction ou de répulsion. De plus, l'ajout de nouveaux objets implique de le modifier profondément.

Contrôleurs de Présentation et d'Abstraction. Nous considérons que les Contrôleurs PAC sont souvent utilisés à contre emploi pour exprimer un comportement entre objets applicatifs ou entre objets graphiques. Pour exprimer clairement de tels comportements, nous introduisons des Contrôleurs spécifiques.

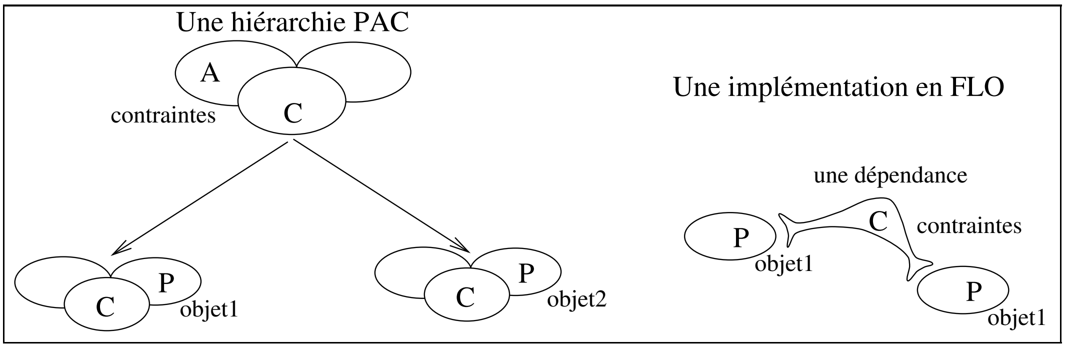


Figure 11.11: Contraintes de Présentation avec PAC et FLO

Les Contrôleurs de Présentation sont des Contrôleurs entre objets de la Présentation. Ils servent à contraindre et maintenir ces objets sans faire intervenir leurs abstractions. Dans ce cas, une part de la sémantique de l'interaction est déléguée à l'interface. La solution proposée par le modèle PAC est une hiérarchie à deux niveaux dont les agents ont une Abstraction vide (voir figure 11.11). Certains comportements peuvent être implémentés à l'aide de contraintes ou de mécanismes réactifs lorsque les boîtes à outils en offrent. En FLO, nous définissons simplement une dépendance entre les objets graphiques. De la même manière, l'utilisation de dépendances pour implémenter des Contrôleurs permet de définir des Contrôleurs entre objets de l'application.

L'utilisation de Contrôleurs spécifiques permet une modélisation plus fine d'une application. Cette modélisation explicite les différentes relations entre les objets et ne masque plus certains comportements dans les Contrôleurs de dialogue. Nous avons commencé à évaluer si des contrôleurs entre contrôleurs étaient nécessaires mais il semble que ce genre de contrôleur soit délicat à définir [Dery 96c].

11.5 Inhibition et resynchronisation des contrôleurs de dialogues

Dans cette partie, nous proposons une modélisation de l'inhibition et de la resynchronisation d'interfaces basée sur la définition de Contrôleurs de Dialogue. Ces nouveaux Contrôleurs relâchent les contraintes de synchronisation entre Présentation et Abstraction. Cette approche autorise une mise à jour asynchrone et optimisée de l'interface lorsque les calculs requis par l'application ne permettent pas de rétablir la cohérence de l'interface en temps réel.

11.5.1 Contexte

Différents modèles traitent le dialogue entre l'application et l'interface en terme de gestionnaire de dialogue, PAC [Cout 87], pac-Amodeus [Niga 91], ALV [Hill 92, Hill 94]). Le rôle du dialogue est de préserver la cohérence entre l'interface et l'application afin d'offrir aux utilisateurs une interface cohérente vis-à-vis de l'application. Cependant, il s'avère quelquefois nécessaire d'interrompre le contrôle de l'application et de retarder la mise à jour de l'interface. Ceci rend alors momentanément l'interface *«malhonnête»*. C'est le cas des applications qui demandent de longues étapes de calcul ou qui procèdent par succès - échec. Il semble que les collecticiels rentrent également dans ce cadre [Kars 93]. Cela peut également permettre de simplifier les actions d'affichage en combinant certaines répercussions redondantes ou complémentaires. Des problèmes se posent alors lorsqu'on souhaite revenir à une interface cohérente par un réaffichage minimum.

Dans cette partie, nous nous plaçons dans le cadre d'une modélisation objet en environnement homogène (application, dialogue et interface sont implémentés dans le même langage) ou hétérogène. Notre objectif est de maintenir la cohérence du système en modélisant la gestion du dialogue par des objets contrôleurs réactifs. Notre proposition repose sur une classification des contrôleurs basée sur différentes mises en oeuvre de la réactivité immédiate ou retardée. Dans ce dernier cas, le rétablissement de la cohérence de l'interface peut procéder par réordonnancement/compression d'événements ainsi que le suggère les travaux de Karsenty et Beaudouin-Lafon [Kars 93]. Le rôle essentiel du gestionnaire est de réagir aux événements de l'application (resp. l'interface) par la demande de mise à jour de l'interface (resp. de l'application) via les contrôleurs.

Nous présentons brièvement l'architecture logicielle choisie centrée sur un gestionnaire de dialogue puis les différentes techniques d'inhibition et resynchronisation des contrôleurs.

11.5.2 Gestionnaire de dialogue et contrôleurs

Tout comme le modèle Arch, nous proposons de gérer le dialogue entre une application et son interface graphique grâce à un Gestionnaire de Dialogue. Celui-ci est constitué d'un ensemble de contrôleurs [Niga 91, Salb 94]. Les contrôleurs gèrent la cohérence entre les objets logiques et leurs représentations graphiques sans aucune modification de ces objets [Dery 94].

Les contrôleurs sont définis en spécifiant essentiellement la liste des messages (ou événements) qui remettent en cause l'honnêteté de l'interface et les réactions minimales à effectuer pour la rétablir. Le gestionnaire de dialogue doit donc être informé de certaines actions sur les objets applicatifs afin de réagir par une demande de mise à jour des objets graphiques (et inversement) et maintenir ainsi de façon incrémentale la cohérence du système.

Certains objets de l'application ou de l'interface sont composites. Nous exploitons cet aspect au niveau du gestionnaire de dialogue en constituant une hiérarchie de contrôleurs 11.12. Cette hiérarchie correspond à des objets composites réels ou simulés et est analogue à une hiérarchie

PAC ou ALV. Elle présente différents avantages [Cout 87, Cout 90] dont : la gestion de nouveaux objets (création automatique des contrôleurs et des représentations correspondantes par exemple), la destruction immédiate d'un groupe d'objets, la propagation de messages aux objets composants, une gestion à différentes granularités de la communication.

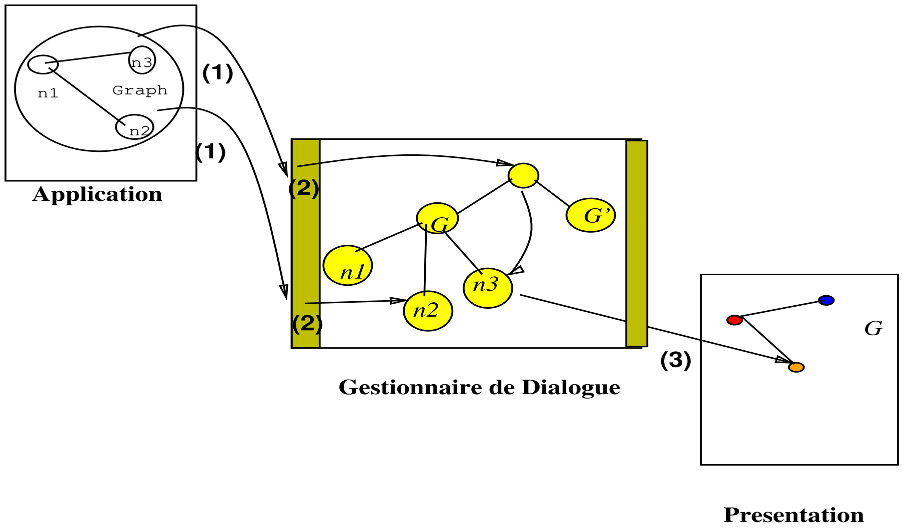


Figure 11.12: Architecture

Réactions immédiates. La figure 11.12 illustre les communications, entre les différents composants, dans le cadre des réactions immédiates. La propagation est schématisée de l'application vers l'interface, le processus est analogue de l'interface vers l'application.

(1) Toute action *perturbante* dans le monde applicatif ou graphique doit envoyer un signal vers le gestionnaire de dialogue. Est qualifiée de *perturbante* toute action qui invalide soit l'affichage soit les données de l'application.

(2) Lorsque le gestionnaire de dialogue reçoit des signaux avec la référence à l'objet concerné, il doit déterminer quel est ou quels sont les contrôleurs qui savent prendre en charge cette partie du dialogue.

(3) Un contrôleur concerné par un signal notifie le monde applicatif ou graphique de la réaction à exécuter.

Cette architecture se projette aussi bien dans un environnement homogène où tous les objets, de l'applicatif, du graphique et du gestionnaire de dialogue sont implémentés dans le même langage qu'en environnement distribué avec trois composants ou deux composants distincts (application et gestionnaire de dialogue / interface).

11.5.3 Inhibition et resynchronisation des contrôleurs

Une interface honnête à laquelle toute modification dans l'application est notifiée, entraîne des désagréments d'affichage lorsque certaines opérations nécessitent des calculs complexes comprenant des retours arrières et des modifications incrémentales des valeurs des objets. De façon symétrique, certaines actions sur l'interface telles que le déplacement d'objets contraints peuvent être momentanément incohérentes vis-à-vis de l'application. Il devient alors important de pouvoir déconnecter provisoirement l'interface de sorte que celle-ci ne soit plus en permanence le reflet de l'application. Il faut être ensuite capable de rétablir *a posteriori* la cohérence de l'interface et ce de la façon la plus intelligente possible. L'intelligence se situe à différents niveaux quelquefois contradictoires : minimiser les mises à jour en évitant un recalcul complet tout en restituant de façon schématique le déroulement des calculs. Des fonctions du gestionnaire de dialogue vont permettre d'implémenter les deux étapes: (1) arrêt de la réactivité, puis (2) resynchronisation. Toute la finesse de la resynchronisation dépend des contrôleurs.

Dans la suite, nous supposons pour simplifier une déconnexion complète de l'interface en phase d'interruption et de resynchronisation. Cette restriction peut être assouplie en autorisant des actions pour stopper l'interface. Une interaction plus forte nécessite une réflexion plus approfondie concernant l'état de l'application sur laquelle les opérations sont réalisées (état avant inhibition, état après resynchronisation finale ou intermédiaire).

L'*arrêt de la réactivité* peut prendre essentiellement deux formes: soit les signaux ne sont plus émis par l'application soit le gestionnaire de dialogue ne réagit pas immédiatement et suspend la réactivité en stockant les signaux. L'intelligence de la *resynchronisation* dépend du degré d'incrémentalité que l'on veut associer à la mise à jour de l'interface. Ce degré d'incrémentalité est tributaire des informations que l'on s'autorise à conserver pendant la phase d'interruption de la réactivité. Il s'agit d'aboutir à un compromis entre un réaffichage pseudo intelligent et le coût des notifications tolérées pendant le calcul.

Après interruption de la réactivité, une *resynchronisation des contrôleurs* implique principalement

- d'afficher les nouveaux objets de l'application,
- de supprimer la représentation des objets détruits et
- de réafficher les objets de l'application qui ont été modifiés.

Nous présentons différentes inhibitions-resynchronisations envisageables selon les contraintes imposées par les applications. Celles-ci reposent sur une classification au sens objet du terme des contrôleurs décrite dans la figure 11.13.

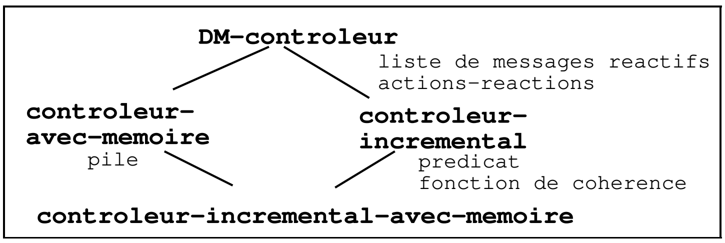


Figure 11.13: Hiérarchie de classes de contrôleurs

Interruption totale de l'émission de signaux.

Supposons que le coût d'émission des signaux soit jugé pénalisant dans le cas de traitements complexes sur le modèle applicatif ou que leur nombre soit trop grand pour les stocker. Nous proposons donc une méthode qui repose sur l'arrêt temporaire de l'émission des signaux ; les modifications des objets de l'application n'entraînent plus d'émission d'informations vers les contrôleurs. Les deux premières étapes de la resynchronisation sont réalisées par une différence ensembliste entre l'ensemble des objets représentés (gérés par les contrôleurs) et ceux à représenter (objets de l'application). Par contre le réaffichage des objets modifiés implique un réaffichage quasi complet des objets déjà visualisés de l'application.

Resynchronisation incrémentale. Le rafraîchissement peut être affiné si l'utilisateur est capable d'associer à chaque contrôleur, un prédicat qui exprime l'honnêteté *«locale»* de l'interface (testant la cohérence d'une représentation et de l'objet de l'application) et une *fonction de cohérence* qui rétablit la cohérence des représentations. Lorsqu'on veut resynchroniser un contrôleur de ce type (cf. contrôleur incrémental figure 11.13), il suffit de tester ce prédicat et lorsque celui-ci est faux d'exécuter la fonction de cohérence. Le prédicat et la fonction de cohérence existent dans les systèmes à base de contraintes mais ils sont beaucoup plus difficiles à écrire dans un cadre général de communication événementielle. Une telle solution n'a de sens que si le coût des tests est minimal par rapport au réaffichage complet de l'interface !

Un algorithme de resynchronisation consiste à appliquer la fonction de cohérence aux contrôleurs racines. La fonction de cohérence associée par défaut à un contrôleur composite est de demander la restitution de la cohérence de tous ses contrôleurs composants. Cependant cette fonction peut être redéfinie pour réinitialiser immédiatement (sans appel à d'autres contrôleurs) toute l'interface associée à la hiérarchie.

Lorsqu'il est possible d'associer contextuellement à la demande de resynchronisation une hiérarchie de contrôleurs correspondant aux objets de l'application concernés, il suffit alors d'examiner les contrôleurs de cette hiérarchie pour réaliser le réaffichage de l'interface. Ce n'est que lorsque la resynchronisation est terminée que le mode réactif peut être rebranché.

Réactions retardées.

Pour d'autres applications, on peut considérer que le coût d'émission de signaux est négligeable et leur stockage possible. Il s'agit alors de retarder la mise à jour de l'interface, sans annuler l'émission asynchrone des signaux vers le gestionnaire de dialogue ; seules les réactions vers l'interface sont provisoirement inhibées. Il faut alors utiliser des contrôleurs à mémoire (cf. figure 11.13). Ces contrôleurs contiennent une pile dans laquelle les signaux peuvent être stockés dans l'ordre de leur arrivée, en attendant d'être traités. La prise en compte de la demande d'interruption de la réactivité consiste alors à inhiber les contrôleurs avec mémoire afin de retarder leur réactivité. Le gestionnaire redirige les signaux vers les contrôleurs concernés et conserve ceux qui ne s'adressent pas directement à des contrôleurs existants (signaux qui, pour être gérés, nécessitent la création au préalable de nouveaux contrôleurs).

Lors de la resynchronisation, le gestionnaire de dialogue se charge de créer les éventuels nouveaux contrôleurs ce qui implique la création de la représentation graphique correspondante. Tous les contrôleurs à mémoire qui ont reçu des signaux sont ensuite activés.

Ce type de réactions n'a de sens que si les modifications sont effectivement localisées à certains contrôleurs, et que les réactions sont complémentaires ; il n'y a pas de redondance telle que le rafraîchissement de l'écran, par exemple.

Réactions retardées combinées. Il est important d'être capable de simplifier les signaux stockés dans la pile afin d'éviter des mises à jour inutiles ou trop partielles. De façon générale, on ne peut pas donner des règles de simplification. Par exemple, deux messages avec les mêmes arguments peuvent être soit redondants soit complémentaires. Cependant l'utilisateur est souvent capable de fournir des fonctions de combinaison des signaux [Clém 88] par contrôleur. Dans cette optique, nous proposons des fonctions prédéfinies de combinaisons dites *«propriétés»* que l'utilisateur peut associer à certains signaux dans chaque contrôleur. Celles-ci seront alors utilisées lors de la resynchronisation du contrôleur pour simplifier la liste des signaux auxquels le contrôleur doit réagir. En fonction de cette nouvelle liste, la réaction sera une conjonction des réactions relatives aux différents signaux.

Soit $S_O = \{s_1, ..., s_n\}$ l'ensemble des signaux adressés à un objet O concernant un contrôleur C. s_1 est le premier et s_n le dernier signal adressé à O pendant l'inhibition du contrôleur C. Voici quelques exemples de propriétés des signaux illustrées par la figure 11.14 :

absorbant(s) Un signal de sélecteur s est dit absorbant lorsque la présence d'un tel signal dans une liste de signaux doit entraîner la suppression des autres signaux.

Un signal s_i de même sélecteur que s dans un ensemble S_O réduit S_O à $\{s_i\}$. Si plusieurs signaux de même sélecteur que s existent dans S_O, seul le dernier signal est conservé.
La destruction d'un objet applicatif absorbe tous les autres signaux concernant cet objet.

absorbant(s,L) L'absorption avec contexte restreint l'absorbtion décrite ci-dessus à une seule liste de signaux L. Seuls les signaux de même sélecteurs que $[s_l, s_k, ..s_p]$ appartenant à L sont absorbés par le signal s.
La présence d'un signal s_i de même sélecteur que s absorbe tous les signaux de même sélecteur que $s_l, s_k, ...$ ou s_p antérieurs à s_i. S_O est alors réduit à $\{s_1..s_i - 1\} - \{s', s'', ..\} U \{s_i..s_n\}$
Le déplacement d'un objet à une nouvelle coordonnée absorbe tout déplacement relatif ou absolu précédent cette opération.
Notons que absorbant(s) est équivalent à $absorbant(s, S_O)$. L'article [KARS 93] présente cette propriété comme du masquage.

dernier(s) Un signal de sélecteur s est qualifié de dernier pour permettre de privilégier le dernier signal de ce type par rapport aux signaux précédents.
La présence de signaux $\{s_k, ..., s_l, s_p\}$ de même sélecteur que s dans S_O réduit S_O à $S_O - \{s_k, ...s_l\}$.
Si en cours de calcul, un noeud a pris différentes valeurs, seule la dernière doit être prise en compte par l'interface.
Notons que dernier(s) est équivalent à absorbant(s,{s}).

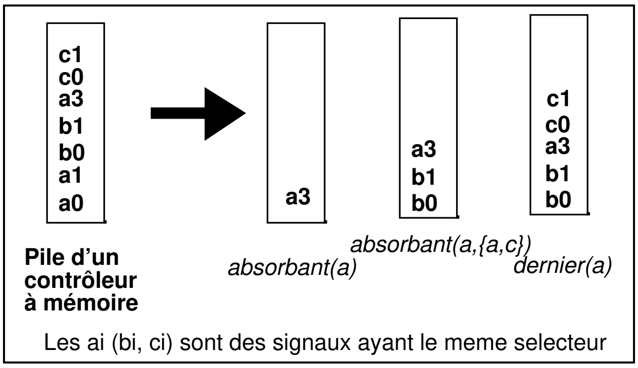


Figure 11.14: Propriétés des signaux

Les propriétés présentées ici sont simples et toutes basées sur le principe d'absorption. Nous envisageons de poursuivre cette étude en nous intéressant à la logique temporelle pour offrir un ensemble plus complet de propriétés. Une description à base de règles de production ou d'automates devrait également permettre une plus grande précision et un plus grand choix de critères. Cependant une telle expression des propriétés ne serait-elle pas trop complexe pour les utilisateurs?

11.5.4 Synthèse

Une architecture fondée sur un gestionnaire de dialogue entre l'application et le graphique permet d'identifier clairement les trois composants: application, dialogue et graphique. Dans notre approche, le gestionnaire de dialogue est essentiellement composé de hiérarchies de contrôleurs réactifs. Une telle architecture permet à la fois un contrôle décentralisé et global du dialogue. Nous avons abordé un point critique de la gestion des applications graphiques par réactivité: quel compromis peut-il y avoir entre l'honnêteté de l'interface et la nécessité d'interrompre momentanément sa mise à jour automatique? Le problème est alors de rétablir la cohérence de l'interface vis-à-vis de l'application. Pour cela, nous avons étudié différentes sortes de resynchronisation. Deux grandes familles se dégagent: l'une en interrompant totalement l'émission des signaux, l'autre en les stockant. Dans ce dernier cas, une analyse des signaux émis permet de réagir de

façon plus adaptée en réduisant les opérations de «undo» and «redo» et en éliminant certaines opérations inutiles. En basant ce travail sur une sémantique précise, des validations formelles de la couche de dialogue sont envisageables. A terme, il est utile de pouvoir combiner ces différentes formes de resynchronisation.
Pour certains contrôleurs, il n'y a pas de gain en terme d'incrémentalité si l'on stocke les signaux émis et une réinitialisation totale des représentations est facile à exprimer. Pour d'autres contrôleurs au contraire, l'analyse des signaux non seulement diminue le coût du réaffichage, mais une fonction de cohérence ne saurait toujours être exprimée. Le découpage du gestionnaire de dialogue en contrôleurs réactifs permet une telle évolution.

Résultat d'une réflexion menée en collaboration avec le CNET de Sophia Antipolis, afin de proposer une architecture de mise en oeuvre d'applications graphiques en administration réseaux, l'architecture se veut souple pour faciliter l'évolutivité des applications.

11.6 Conclusion

Dans ce chapitre, après avoir fait un rapide tour d'horizon des différents modèles architecturaux pour la construction d'interfaces hommes-machines, nous avons montré que les dépendances sont un outil d'implémentation adapté pour la construction de telles interfaces. En effet, les dépendances offrent une séparation logique entre les objets de l'application et les objets de l'interface, explicitent les interactions entre ces objets de manière distincte de leur code et assurent la communication et la cohérence entre objets de l'application et de l'interface.

Nous avons montré comment les dépendances permettent l'implémentation immédiate du modèle ALV [Hill 92] et comment les dépendances permettent une relecture du modèle PAC [Cout 89] en affinant le rôle des contrôleurs. Nous avons défini différentes alternatives pour la resynchronisation d'interfaces à l'aide de contrôleurs de dialogue spécialisés.

12

Réification de l'héritage, bootstrap et méta-stabilité

« *Two general categories of circularity arise in procedurally reflective languages:* bootstrapping *issues which are involved with how to get the system up and running in the first place, and* metastability *issues which have to do with how the system manages to run, and to stay running even while fundamental aspects of the implementation are being changed.* » [Kicz 91].

De nos jours, le mécanisme d'héritage est un des concepts les plus important de la programmation orientée objet et est, d'après P. Wegner [Wegn 87], intrinsèque à la définition même du terme *orienté-objet*. La littérature regorge de travaux sur : sa sémantique [Card 84, Card 85, Cook 89], son utilisation [Halb 87, Snyd 86], les problèmes liés à l'héritage multiple [Carr 90, Duco 92, Duco 94], de nouveaux modèles d'héritage [Brac 90, Jaga 92], des solutions alternatives [Lieb 86a, Stei 87]...

Comme nous l'avons écrit au début de cette thèse, nous considérons l'héritage comme un mécanisme de définition incrémentale dénué de toute notion de types. L'idée présentée maintenant est de considérer ce mécanisme comme une dépendance entre des objets particuliers : des classes et à ce titre de l'exprimer de manière distincte de la structure et des fonctionnalités des classes. Dans ce sens, nous rejetons le mécanisme d'héritage comme un mécanisme câblé à l'intérieur du langage. Nous proposons une réification du mécanisme d'héritage au moyen de dépendances entre classes. La réification de l'héritage au moyen de dépendances donne alors à FLO une définition complètement réflexive exclusivement basée sur deux concepts : les objets et les dépendances. L'héritage n'est plus un mécanisme global construit et figé dans le langage, mais devient décrit par le langage lui-même : il est géré comme les autres dépendances. FLO est alors auto-décrit, dans le sens où la sémantique de l'héritage est décrite dans les termes mêmes des méthodes de classes. Cette réintroduction de l'héritage dans un langage à classes s'apparente aux travaux de P. Mulet sur la réintroduction de la délégation dans un langage à prototypes [Mule 93a].

Nous avons choisi d'utiliser l'implémentation de FLO réalisée dans une implémentation du modèle ObjVlisp comme cadre d'expérimentation. Cette implémentation nous permet de manipuler tous les aspects du langage. Afin de ne pas prêter à confusion par rapport à la description de FLO des chapitres précédents, nous nommons FLO/ObjV cette présente implémentation.

Ainsi nous proposons non seulement un mécanisme d'héritage à base de dépendances [Duca 95c] mais surtout nous appliquons cette solution au modèle ObjVlisp en proposant alors un langage uniforme [Duca 96a]. Nous nous plaçons résolument d'un point de vue implémentation ; un potentiel utilisateur de ce langage n'aurait pas à se préoccuper de tels détails d'implémentation.

Le plan de ce chapitre est le suivant : premièrement, nous rappelons le modèle ObjVlisp avec un regard privilégié sur l'héritage et ces conséquences structurelles sur les classes. Deuxièmement, nous présentons comment l'héritage de ObjVlisp est réifié. Troisièmement, nous finissons par une mise en avant des problèmes liés aux bootstrap et la méta-stabilité du noyau de FLO/ObjV. Pour finir, nous présentons une première extension du mécanisme d'héritage.

12.1 Le modèle ObjVlisp

Le modèle ObjVlisp a fait l'objet de très nombreuses publications [Coin 87, Coin 88, Brio 87a, Brio 87b]. Ce modèle a été implémenté en Smalltalk [Brio 89a, Coin 92, Ledo 96]. Nous le décrivons rapidement en nous préoccupant principalement du mécanisme d'héritage.

12.1.1 Architecture d'ObjVlisp

"The ObjVlisp model supports a simple, clean and minimal architecture for explicit metaclasses." [Brio 89a]

Les classes en ObjVlisp sont explicitement et uniformément instances d'autres classes : des méta-classes. De plus, ObjVlisp est minimal de part sa complète auto-définition à l'aide de deux classes : la classe `Class`, instance d'elle-même, racine du graphe d'instanciation et la classe `Object`, instance de la classe `Class`, racine du graphe d'héritage comme le montre la figure 12.1.

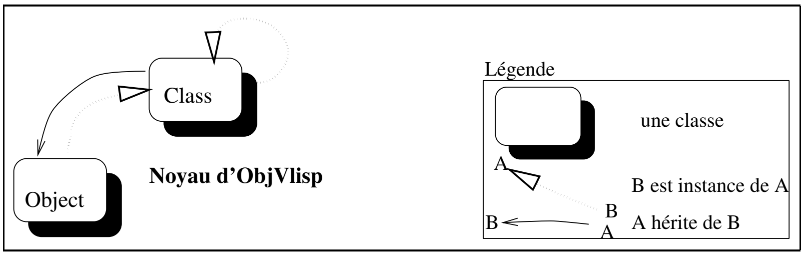


Figure 12.1: Architecture du noyau d'ObjVlisp.

Structure d'une classe. Les données représentant une classe sont : un nom, une liste de variables d'instance, un dictionnaire de méthodes et la liste des super-classes de la classe. De telles informations sont les variables d'instance de la classe décrivant les classes : la méta-classe `Class`. En particulier, la variable d'instance `supers` situe la classe dans le graphe d'héritage. Ainsi l'exemple montre que la classe `Class` hérite de la classe `Object`.

```
(send Class '? 'i-v)
> (isit name supers i-v keywords methods)
(send Class '? 'supers)
> (Object)
```

12.1.2 L'héritage en ObjVlisp

Le mécanisme d'héritage du modèle ObjVlisp se décompose de la même manière que celui de Smalltalk en deux comportements distincts : l'héritage *dynamique* des méthodes et l'héritage *statique* des variables d'instances.

Héritage des méthodes. Celui-ci est dynamique, il a lieu lors d'un envoi de message. L'envoi de message est décomposé en deux phases distinctes : la recherche de la méthode, correspondant au sélecteur du message et l'application de cette méthode. Chaque fois qu'un message est envoyé à un objet, la phase de recherche parcourt le graphe d'héritage, en utilisant les informations contenues par la variable d'instance `supers`, pour trouver la méthode recherchée. Quand la recherche n'aboutit pas dans la classe du receveur la recherche continue en profondeur ou largeur dans les superclasses. La recherche déclenche une erreur quand la méthode n'est pas trouvée c'est-à-dire quand la recherche échoue dans la classe `Object`. Il est important de bien considérer que l'algorithme de recherche est le même pour toutes les classes.

Héritage des variables d'instances. L'héritage des variables d'instances est statique et exécuté une seule fois lors de la création d'une classe. Quand une nouvelle classe est définie, ses variables d'instances sont calculées comme l'union de toutes les variables d'instances de ses superclasses et des variables d'instances précisées lors de sa définition. En particulier, toutes les classes héritent de la variable d'instance `isit` de la classe `Object`. Cette variable représente le lien d'instanciation entre un objet et sa classe : une instance connaît toujours sa classe.

12.1.3 Contrôle standard de l'envoi de messages en FLO/ObjV

Dans FLO, un méta-objet peut être associé à un objet. Afin d'introduire des méta-objets nécessaires au contrôle de l'envoi de message, nous modifions le noyau de ObjVlisp : chaque objet possède une variable d'instance supplémentaire `meta`. La primitive d'envoi de message doit être modifiée comme le montre la définition suivante.

Les dépendances écrites en FLO sont basées sur le contrôle des méthodes des objets intervenant dans la dépendance. Pour représenter le mécanisme d'héritage à l'aide de dépendances il est donc nécessaire que les fonctionnalités associées aux classes soient des méthodes. Ainsi dans FLO/ObjV, la primitive de recherche des méthodes `lookup` n'existe plus comme dans le modèle original ObjVlisp, une méthode `lookup` est définie sur la classe `Class`. Ce changement introduit une possible régression infinie dans la phase de recherche : envoyer un message implique d'envoyer le message `lookup` qui implique d'envoyer le message `lookup`...

Ainsi la nouvelle définition de la primitive `send` prend en compte ces deux besoins : (a) seuls les messages aux objets mis en dépendances sont contrôlés et (b) la gestion de la régression infinie de la phase de recherche n'interfère pas avec la gestion des dépendances. La régression infinie de la phase de recherche est stoppée (lignes 2 et 3) en testant le seul fait dont nous sommes sûrs : la classe `Class` possède la méthode `lookup` dans son dictionnaire de méthodes. Lors d'un tel message, la méthode `lookup` est extraite à l'aide de la primitive `get-method` (ligne 3). Lorsque la recherche a pour objet une méthode autre que la méthode `lookup` sur une classe différente de `Class`, la méthode est recherchée dans la classe de l'objet receveur (ligne 4).

```
1  (define send (lambda (recvr selector . args)
2    (let ((m (if (and (equal? selector 'lookup) (equal? recvr Class))
3                 (get-method Class 'lookup)
4                 (send (class-of obj) 'lookup selector recvr)))
5          (a-meta (meta-of recvr)))
6      (if (not-meta? a-meta)
7          (apply-method m recvr args)
8          (send a-meta 'Control m selector cons recvr args)))))
```

Ici, nous avons simplifié la condition nécessaire au contrôle des messages : dans FLO, le contrôle n'est effectué que si la méthode doit l'être effectivement, en FLO/ObjV, le contrôle est effectué dès lors que l'objet possède un contrôleur. Ainsi quand le receveur du message n'est pas lié à un contrôleur (ligne 6) le message est résolu normalement (ligne 7). Si le receveur possède un contrôleur, l'envoi du message `Control` au contrôleur prend en charge la gestion du message (ligne 8). Nous renvoyons le lecteur au chapitre 9 pour de plus amples détails au sujet de la gestion des dépendances.

12.2 Réification et héritage

FLO/ObjV est fortement basé sur le modèle ObjVlisp. La principale différence entre une implémentation traditionnelle de ObjVlisp et celle de FLO/ObjV est la gestion des dépendances et la réification du mécanisme d'héritage au moyen de celles-ci : l'héritage entre deux classes est représenté par une dépendance effective. La seule restriction que nous nous sommes posés est que cette nouvelle gestion de l'héritage ne doit pas nous conduire à modifier la primitive d'envoi de messages telle que nous l'avons décrite précédemment : l'héritage doit être introduit de manière uniforme par rapport aux dépendances.

Dans les sections suivantes, nous présentons les conséquences de la réification de l'héritage au niveau de la description et du comportement des classes et l'implémentation des dépendances gérant l'héritage.

12.2.1 Nouvelles structures

La classe `Stack` de notre premier exemple de dépendances n'était pas modifiée structurellement pour que ses instances puissent être liées à des instances de la classe `Memory` (aucune variable d'instance n'était ajoutée à la classe représentant la pile). Aussi enlevons nous les informations relatives à l'héritage de la description d'une classe.

La définition suivante décrit l'auto-définition[1]de la méta-classe `Class` classe de toutes les classes. Ainsi la variable d'instance `supers` n'existe plus (ligne 3).

La structure des objets est quand à elle simplement étendue pour prendre en compte une référence au contrôleur, la liste des variables d'instance est enrichie de la variable `meta` (ligne 10).

```
1   (send Class 'new
2      :name 'Class
3      :iv '(name iv keywords methods)
4      :methods  '(basicnew (lambda () (allocate (name self)))
5                  lookup   (lambda (selector obj) (get-method self selector))
6                  new      (lambda initargs
                              (send (send self 'basicnew) 'initialize
                                    initargs))
7                ...)))
8   (send Class 'new
9         :name 'Object
10        :iv '(isit meta)
11        :methods   '(?      (lambda (iv) (....))
12                     ?<-     (lambda (iv new-val) (...))
13                     error   (lambda ...)
14                   ...)))
```

La classe `Class` possède une méthode `lookup` qui exécute seulement une recherche *locale* de la méthode recherchée en utilisant la primitive `get-method` (ligne 5). Cette méthode rend faux (♯f) lorsqu'il n'existe pas de méthode locale correspondant à celle recherchée.

12.2.2 Héritage simple

Nous présentons maintenant comment un mécanisme d'héritage simple est exprimé à l'aide d'une dépendance entre classes.

[1]Cette auto-définition est rendue possible grâce à un mécanisme de bootstrap similaire à celui de ObjVlisp.

La dépendance *Inherit-from*

La sémantique du mécanisme d'héritage doit être explicitement exprimée : la recherche dynamique des méthodes et l'héritage statique des variables d'instances.

Héritage dynamique : deux cas. Pour l'héritage dynamique des méthodes, l'idée est simple : une dépendance contrôle la méthode `lookup` des classes (de la même manière que la méthode `pop` de la classe `stack` pour la dépendance *mémorisée-par*). La dépendance vérifie si la méthode a été trouvée localement, si ce n'est pas le cas, la dépendance lance la recherche dans la superclasse. Nous appellerons une telle dépendance : une *dépendance d'héritage*.

Le fait qu'une dépendance d'héritage lie localement des classes permet de spécifier le mécanisme d'héritage de manière locale. Par exemple, la gestion des erreurs (l'émission du message `error`) n'a pas à être définie pour toutes les dépendances d'héritage, seules celles dont une des classes impliquées dans la dépendance d'héritage est la classe `Object` doivent prendre en compte la gestion des erreurs[2]. Ainsi, suivant la position de la classe dans le graphe d'héritage deux cas se présentent. La définition de ces deux variantes est la suivante :

```
1 (deflink Inherit-from  (:superclass:class)
2     :inherit ((Inherit))
3     :behavior
4      '((lookup:class selector object)
5         implies-substitute
6        (if (is-method? result)
7            (begin (stock:class) result)
8            (send:superclass 'lookup selector obj))))
```

Le comportement dynamique de la dépendance *Inherit-from* est le suivant : si la méthode n'est pas trouvée dans la classe désignée par la variable active `:class`, la recherche est poursuivie dans sa superclasse (ligne 8). La définition est purement locale : la gestion des erreurs n'est pas prise en compte par cette dépendance. Nous reviendrons plus en détail sur le sens de l'opérateur `implies-substitute`, pour le moment il signifie que la méthode contrôlée est exécutée mais que le résultat rendu est celui du message compensatoire associé.

Lorsqu'une classe hérite directement d'une des classes racines du graphe d'héritage par exemple de la classe `Object`, la définition de la dépendance d'héritage doit prendre en compte cette situation particulière pour ne pas relancer la recherche et émettre une erreur lorsque la méthode n'est pas trouvée.

```
(deflink Inherit-f-w-error (:superclass:class)
  :inherit ((Inherit))
  :behavior
   '((lookup:class selector object)
     implies-substitute
     (if (is-method? result)
         (begin (stock:class) result)
         (let ((m (get-method:superclass selector)))
            (if (is-method? m)
                m
                (send object 'error selector))))))
```

Lorsque la méthode recherchée est trouvée dans la classe désignée par la variable active `:class`, elle est enregistrée afin que des appels aux méthodes masquées soient possibles à l'aide de `run-super` l'équivalent du `call-next-method` de CLOS puis elle est rendue. La recherche est alors terminée, la superclasse n'est pas mise à contribution. Lorsque la méthode n'est pas trouvée dans la classe

[2]Pour des raisons de régressions infinies, la classe `Object` n'est pas la seule classe pour laquelle un tel comportement doit être mis en place, nous reviendrons sur ce point précis dans la partie 12.3.

désignée par la variable d'instance `:class`, relancer la recherche au moyen des dépendances n'est pas possible : la dépendance en cours est la plus proche de la racine du graphe d'héritage. Si la méthode recherchée est définie dans la classe désignée par la variable `:superclasse` elle est rendue (ligne 9) sinon une erreur est déclenchée (ligne 10). En effet, on est alors sûr que la méthode n'est pas dans les classes du graphe d'héritage.

Les deux dépendances d'héritage *Inherit-from* et *Inherit-f-w-error* sont des instances de la classe `Meta-Link`. Elles héritent toutes les deux de la classe `Inherit` qui est une classe abstraite permettant de factoriser le comportement propre des dépendances agissant sur l'héritage.

De manière similaire au choix de J. Ferber [Ferb 89], la phase de recherche des méthodes est effectuée par les méta-objets, cependant dans notre approche ces méta-objets utilisent les informations décrites par les dépendances et non par les classes.

Héritage statique. Dans le modèle ObjVlisp, l'héritage statique des variables d'instance intervient lors de la création d'une classe. Dans FLO/ObjV, cet héritage est mis en place lors de la déclaration d'une dépendance d'héritage entre deux classes. Pour cela, nous utilisons un des points d'entrée du MOP de FLO: la méthode `action-after-creation`. Cette méthode est appelée chaque fois qu'une dépendance est créée et permet de définir alors une action.

Nous spécialisons la méthode `action-after-creation` comme illustré ci-dessous[3].

```
1 (define-method action-after-creation ((lk Inherit-from))
2    (let ((ancestor (send lk 'give:superclass)))
3          (descendant (send lk 'give:class)))
4      (keywords! descendant (add (keywords ancestor) (keywords descendant)))
5      (iv! descendant
6           (remove-duplicates (append (iv ancestor) (iv descendant)))))
```

Ainsi quand une dépendance effective d'héritage est créée entre deux classes, la méthode `action-after-creation` est invoquée pour cette nouvelle instance de dépendance. Les variables d'instances sont définies comme étant l'union des variables d'instance de la superclasse et des variables définies localement (ligne 5). Les mots clés, utilisés pour l'instanciation des instances [Coin 87], sont aussi calculés à cet instant.

Remarques. Dans le modèle ObjVlisp, l'héritage des variables d'instance était fortement associée à la création de la nouvelle classe par la méthode `initialize` de la classe `Class`. Dans FLO/ObjV, l'héritage est dissocié de la création des classes, il est lié à la déclaration des dépendances d'héritage. Le code présenté ici possède alors une précondition forte, la superclasse d'une nouvelle classe doit toujours avoir hérité de sa superclasse avant que la nouvelle classe n'hérite de la sienne. Par exemple, si A hérite de B, et B de C déclarer que A hérite de B puis que B hérite de C conduit à une erreur. Dans la section 12.4 nous proposerons une solution à ce problème.

Nous avons présenté ici exclusivement un héritage simple. Cependant, la particularisation d'une telle méthode permet l'expression de différentes mécanismes. Nous pourrions gérer les problèmes de conflits lors de l'héritage multiple, proposé un héritage différencié selon le statut des variables d'instances (publiques, privées ...).

12.2.3 Un exemple d'héritage.

Soit la hiérarchie très simple suivante : la classe `Point` hérite de la classe `Object` et la classe `Colored-Point` hérite de `Point` comme illustré dans la figure 12.2.

La mise en œuvre de cette hiérarchie est la suivante : tout d'abord, ces deux classes sont définies (lignes 1 à 6), ensuite les dépendances d'héritage sont créées entre ces classes : une instance de *Inherit-f-w-error* entre la classe `Point` et la classes `Object` (ligne 7) et une instance de *Inherit-from* entre la classe `Point` et la classe `Colored-Point` (ligne 8).

[3]Nous avons choisi de présenter la définition de méthodes de manière homogène par rapport aux précédents chapitres.

```
1  (define Point (send Class 'new :name 'Point :iv '(x y)
2                               :methods '(x       (lambda ...)
3                                          display (lambda ...))))
4  (define Colored-Point (send Class 'new :name 'Colored-Point :iv '(color)
5                           :methods '(color   (lambda ...)
6                                      display (lambda ...))))
7  (send Inherit-f-w-error 'new :class Point :superclass Object)
8  (send Inherit-from 'new :class Colored-Point :superclass Point)
9  (send Colored-Point '? 'iv)       >> (isit meta x y color)
10 (define cp2 (send Colored-Point 'new :x 33 :y 45 :color "blue"))
11 (send cp2 'x)                       >> 33
```

Comme le montre l'emploi de la méthode `?`[4], le message ligne 9 illustre à la fois le mécanisme d'héritage dynamique pour la recherche de méthodes et l'héritage de variables d'instance : les variables `isit` et `meta` sont héritées de la classe `Object` et `x` et `y` héritées de la classe `Point`.

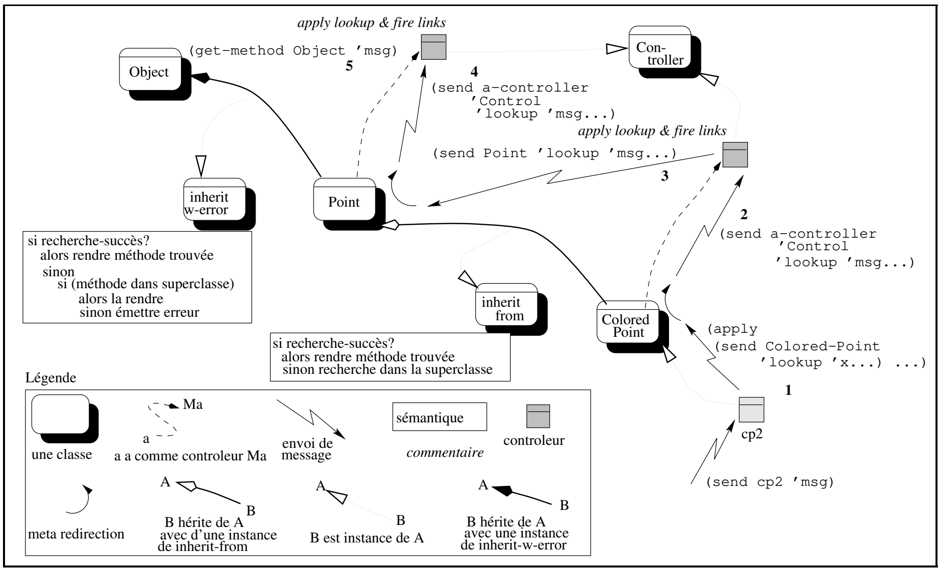


Figure 12.2: Utilisation de dépendances d'héritage : deux cas de base.

La figure 12.2 illustre la recherche d'une méthode lors de l'envoi de message. L'envoi d'un message `msg` à l'instance `cp2` conduit à trois situations :

- supposons que le sélecteur du message désigne une méthode définie dans la classe du receveur, par exemple dans la classe `Colored-Point`, la méthode est recherchée dans la classe (1). Cet envoi de message `lookup` est contrôlé (2). Comme la méthode est définie dans la classe `Colored-Point`, la phase de recherche est terminée.
- Supposons que la méthode soit définie dans la classe `Point` par exemple `x`, `y`..., le contrôle (2) du message `lookup` (1) conduit à poursuivre la recherche dans la classe `Point`. Pour cela, le message `lookup` est envoyé à la classe `Point` (3). Ce message est à son tour contrôlé, mais comme la méthode recherchée est définie dans la classe `Point` la recherche s'achève en retournant la méthode trouvée.
- Lorsque la méthode invoquée n'est définie ni dans la classe `Colored-Point` ni dans la classe `Point`, le cas précédent est étendu : si la méthode est définie dans la classe `Object` (5) elle est rendue sinon une erreur est émise.

[4] Cette méthode est définie sur la classe `Object`, elle permet de trouver la valeur d'une variable d'instance et correspond au `slot-value` de CLOS.

12.2.4 Un contrôle des messages différent

Lors de la définition des dépendances d'héritage, nous avons introduit l'opérateur `implies-substitute`. Cet opérateur est associé à un nouveau type de contrôleur. La sémantique de cet opérateur est la suivante : la méthode contrôlée est exécutée mais le résultat rendu n'est pas, contrairement à l'opérateur `implies`, celui de la méthode contrôlée mais celui de l'action qui lui est associée comme le montre la définition ci dessous :

```
(send Class 'new
     :name 'Substitute-Controller
     :methods
      '(Control (lambda (recvr selector args method)
                   (let ((res (run-super))
                         (lkf (send self :get-links 'implies-substitute
                                                    recvr selector)))
                      (if  (no-link? lkf)
                           res
                           (send lkf 'firing-substitute
                                        recvr selector args res)))))))
```

12.3 Architecture

Dans cette partie, nous abordons les problèmes d'implémentation dus aux régressions infinies et nous présentons l'architecture que nous avons choisi afin d'assurer une stabilité au système.

12.3.1 Régressions infinies

Afin de justifier nos choix architecturaux pour le noyau de FLO/ObjV, nous présentons maintenant un premier cas de régression infinie dû au contrôle de l'envoi de message inhérent au modèle de dépendances.

Supposons que nous voulions représenter le simple fait que la classe `Class` hérite de la classe `Object` en utilisant pour cela une dépendance d'héritage. Ce choix implique de contrôler les messages envoyés à la classe `Class` et d'associer un contrôleur à celle-ci. Cependant, ce simple choix conduit immédiatement à une régression infinie comme illustré par la figure 12.3.

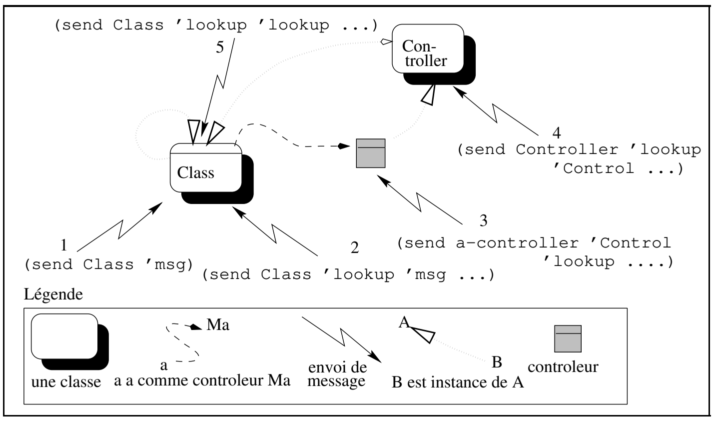


Figure 12.3: Une simple configuration menant à une régression infinie.

En effet, supposons que la classe reçoive un message `msg` (noté par le nombre 1 dans la figure 12.3 et dans la décomposition du message ci-dessous). Comme `Class` est instance d'elle-même, ce sélecteur est recherché dans `Class` elle-même en s'envoyant le message `lookup` (2). Ce dernier message doit être contrôlé pour laisser la dépendance d'héritage agir. Le message `control` est envoyé au contrôleur associé à la classe `Class` (3). Cet nouvel envoi de message implique alors de rechercher la méthode `Control` dans la classe du contrôleur (4) en envoyant la méthode `lookup`. Cette classe est elle-même instance de la classe `Class`, donc la recherche de la méthode `lookup` est envoyée à la classe `Class` (5). Et la boucle est bouclée.

```
1 (send Class 'msg)
2     (send Class 'lookup 'msg)
3         (send a-controller 'Control 'lookup 'msg Class)
4             (send Substitute-Controller 'lookup 'Control)
5                 (send Class 'lookup 'lookup)
6 ...
```

Différentes solutions peuvent être trouvées pour arrêter cette régression infinie. Par exemple, nous pouvons tester la recherche de la méthode de recherche sur la classe `Class` (6) (que la méthode `lookup` est recherchée pour trouver la méthode de recherche). Cependant, ce type de solution reste une solution *ad hoc* basée sur la modification de la primitive `send`. Une régression plus subtile apparaît dès lors que la classe `Substitute-Controller` hérite au moyen d'une dépendance d'héritage de la classe `Controller` ou `Object`. De la même, manière une autre boucle apparaît quand la classe `Meta-Link` hérite de la classe `Class` ou la classe `Link` de la classe `Object`. Chaque fois, la solution consiste à modifier la primitive et à y inclure une *batterie* de tests afin de stopper ces régressions.

En fait, l'observation du phénomène nous a conduit à privilégier une autre solution. Dans le premier exemple de régression, celle-ci était due à la nécessité de contrôler les messages envoyés à la classe `Class` pour permettre au mécanisme des dépendances d'entrer en jeu. La solution consiste donc à ne pas gérer l'héritage entre la classe `Class` et la classe `Object` au moyen d'une dépendance. Nous considérons un noyau minimal dans lequel les classes sont autonomes du point de vue de l'héritage. Ainsi le noyau représentant les classes responsable du contrôle de l'envoi de message est stable et figé comme le montre la figure 12.4. Pour ces classes exclusivement, le contrôle de l'envoi de message ne peut pas être lui-même contrôlé. Ces cinq classes héritent conceptuellement les unes des autres, mais sont définies de manière autonomes. C'est pourquoi nous pouvons les considérer comme autant de racines du graphe d'héritage du point de vue des dépendances auxquelles elles peuvent participer. Ces classes ne peuvent que jouer le rôle de la classe `:superclass` pour des instances de la dépendance d'héritage *Inherit-with-error*.

Par rapport au noyau initial de ObjVlisp, le noyau de FLO/ObjV inclut quatre nouvelles classes : la classe `Meta-Link` qui est la méta-classe de toutes les dépendances, la classe `Link` qui est la classe représentant toutes les fonctionnalités liées aux dépendances, la classe `Controller` qui définit toutes les fonctionnalités des contrôleurs et la classe `Inherit` qui est la classe regroupant les fonctionnalités des dépendances spécifiques à la représentation de l'héritage. Minimiser le nombre de ces `Class` pourrait être un de nos prochains travaux.

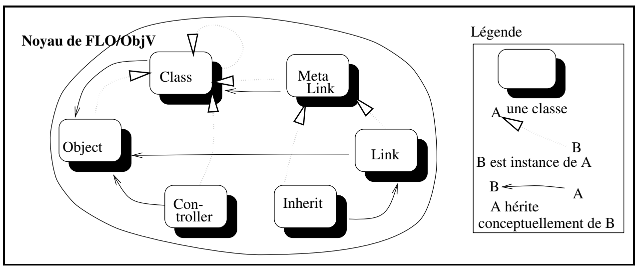


Figure 12.4: Le noyau de FLO/ObjV.

La figure 12.5 illustre l'architecture dans le cadre de l'exemple proposé plus avant.

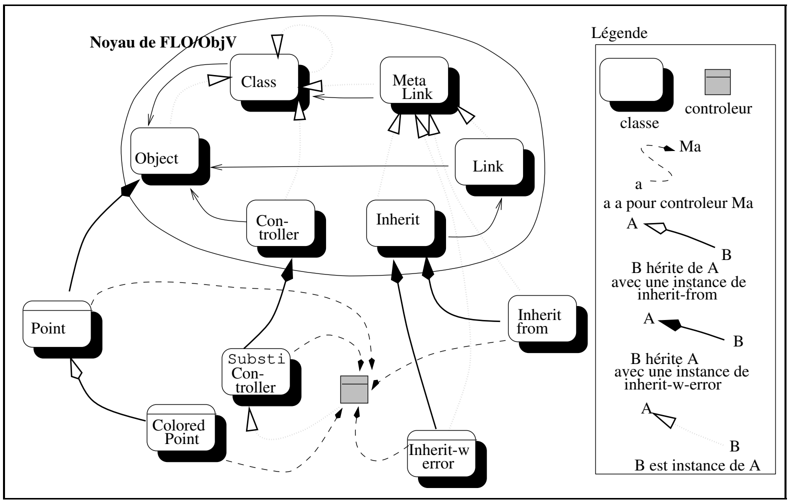


Figure 12.5: Le noyau de FLO/OBJV et un exemple d'héritage explicite entre les classe `Object`, `Point` et `Colored-Point`.

12.4 Une première extension : un héritage dynamique des variables d'instances

Comme nous l'avons exposé dans la partie 12.2.2, l'héritage des variables d'instances implique que les dépendances d'héritage soient instanciées de la classe la plus haute dans le graphe d'héritage vers les classes héritières. Nous montrons comment l'héritage des variables d'instances peut être dynamique ; c'est-à-dire que l'ajout d'une variable d'instance dans une superclasse est propagé à toutes ces sous-classes.

Nous définissons la méthode `set-ivs` sur une nouvelle méta-classe `Class-Dynamic`. Ensuite, nous créons trois nouvelles dépendances : la dépendance *Dynamic-Inherit* qui hérite de la dépendance *Inherit*, la dépendance *Dynamic-inherit-from* qui hérite des dépendances *Dynamic-Inherit* et *Inherit-from* et la dépendance *Dynamic-inherit-f-w-error* qui hérite de la dépendance *Dynamic-Inherit* et de la dépendance *Inherit-f-w-error*. Notons que ces héritages sont réalisés à l'aide de dépendances d'héritage.

La définition suivante spécifie que lors de la modification de la liste des variables d'une classe, ces modifications sont propagées à ses sous-classes.

```
1 (deflink Dynamic-Inherit  (:superclass:class)
2  :inherit '((Inherit-from))
3  :behavior
4    '(((set-ivs:superclass new-ivs)
4       implies-substitute
5       (send:class 'set-iv (remove-duplicates (append new-ivs (iv:class))))
6       (keywords!:class (make-keywords (iv:class))))))
```

Troisièmement, nous spécialisons la méthode `action-after-creation` de telle sorte qu'au lieu d'utiliser une primitive gérant l'accès aux variables, nous utilisons la méthode `set-ivs` (ligne 10). Ce faisant l'ordre de création des dépendances d'héritage n'est plus contraint.

```
7 (define-method action-after-creation  ((lk Dynamic-Inherit))
8   (let ((s (send lk 'give:superclass))
9         (c (send lk 'give:class)))
10    (send d 'set-ivs (remove-duplicates (append (iv s) (iv c))))
11    (keywords! d (make-keywords (iv d)))))
```

La figure 12.6 montre une vision simplifiée des différentes dépendances d'héritage nécessaires à la gestion des variables dynamiques.

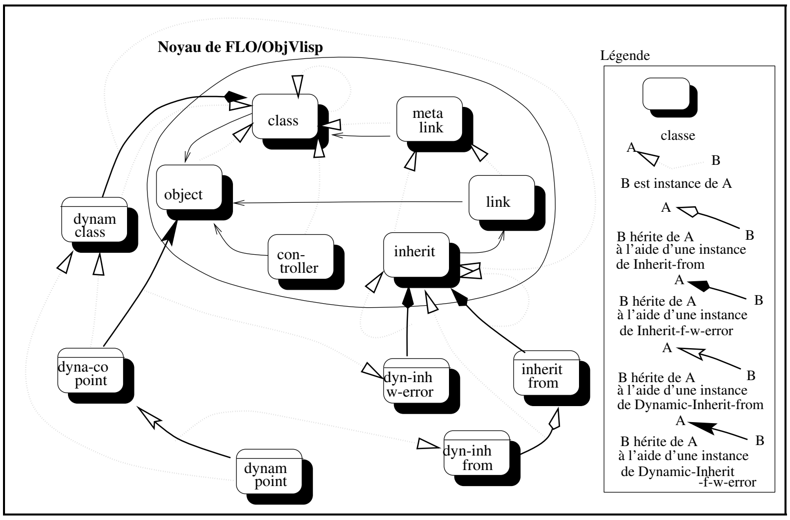


Figure 12.6: Une hiérarchie simplifiée d'une première extension des dépendances d'héritage : un héritage dynamique des variables d'instances.

Cette extension est simple et ne prend pas en compte l'état des instances créées avant l'ajout d'une variable d'instance. Un mécanisme similaire à celui de CLOS `update-instance-for-diffe -rent-class` pourrait être implémenté.

12.5 Autres travaux

Le contrôle de l'héritage au moyen de MOPs a déjà été abordé dans le langage CLOS ou MOOSTRAP. Nous discutons ici ces approches.

Un mécanisme d'héritage ouvert : la solution de Clos. L'héritage en CLOS [KEEN 89, KICZ 91, Pae 93] est fortement basé sur la linéarisation du graphe d'héritage [DUCO 92, DUCO 94], en construisant une *liste de précédence* des superclasses pour chaque classe. Cette liste de précédence est utilisée pour déterminer l'ordre dans lequel les classes seront parcourues pour hériter les variables d'instances ou pour trouver les méthodes à appliquer.

Tout d'abord, l'héritage des variables d'instances est réalisé durant le protocole de *«finition des classes», (Class Finalization Protocol p 93, 155 de [*KICZ *91]) « Class finalization is the process of computing the information a class inherits from its superclasses and preparing to actually allocate instances of the class »* [KICZ 91]. La fonction générique `finalize-inheritance` est responsable du calcul de la liste de précédence, de l'héritage des variables d'instances et de la définition de l'ensemble des arguments d'initialisation de la classe. Elle appelle respectivement la fonction générique `compute-class-precedence-list` et la fonction générique `compute-slots`. Plus précisément, la fonction `compute-slots` appelle la fonction `compute-effective-slot-definition` qui fusionne les variables d'instance de même nom.

Contrairement à CLOS, FLO/OBJV suit la même philosophie que celle de MOOSTRAP pour qui la délégation reste avant tout un mécanisme local au comportement du méta-objet [MULE 93a].

Moostrap et le protocole Lookup ∘ Apply. MOOSTRAP [MULE 95a] est un langage réflexif à prototype qui se distingue des autres langages à prototypes traditionnels (SELF [UNGA 87], NEWTON [SMIT 95]). En effet, MOOSTRAP ne définit *a priori* pas de mécanisme de délégation. Ce n'est que par le biais de la réflexion comportementale que la délégation est réintroduite dans le langage. MOOSTRAP est basé sur le protocole réflexif lookup ∘ apply défini dans [MALE 92] que nous avons déjà décrit au chapitre précédent. Dans [MULE 93a], les auteurs réintroduisent la délégation comme une succession de redirections élémentaires.

12.6 Conclusion

Outre le défi que représente cette fusion réflexive, ce travail suit la même philosophie que la réintroduction de la délégation dans un langage à prototypes proposée par P. MULET et P. COINTE [MULE 93a]. Dans FLO/OBJV, nous avons tout d'abord rejeté l'héritage en tant que mécanisme de base du noyau puis nous l'avons réintroduit par le biais des dépendances. Dans ce sens, cela tendrait à prouver que le lien *meta* est plus primitif que le lien d'héritage ou de délégation. Le mécanisme d'héritage est modifié grâce à la définition de nouvelles dépendances et à la spécialisation de certains points d'entrée du protocole. De plus, il est séparé de la définition des classes tant au niveau structurel et comportemental que du moment ou il intervient. Cette réification à l'aide de dépendances offre à FLO/OBJV une définition réflexive et uniforme basée sur deux concepts : les objets et les dépendances. L'héritage est alors géré comme les autres dépendances, ce qui permet de pouvoir définir et mélanger différents comportements.

D'autres extensions. Cette modélisation de l'héritage offre la possibilité d'exprimer explicitement le sens accordé à l'héritage. Nous pensons que différents comportements peuvent ainsi être spécifiés. Nous n'avons pas exploré plus en avant cet aspect de notre travail mais nous restons persuadés qu'une dépendance d'héritage est l'objet idéal pour des optimisations dans la recherche des méthodes en proposant un mécanisme de *cache* ou une gestion différente des variables d'instances (privées, publiques...). Cependant nous sommes conscient que ce travail reste un travail de modélisation de l'héritage.

Quelques points restent en suspend : avec la séparation de l'héritage des variables d'instance de la création des classes, de nouveaux problèmes apparaissent. Ainsi les actions devant être exécutées à la création des classes deviennent plus subtiles : par exemple, implémenter la méta-classe `AccessClass` qui offre la création automatique d'accesseurs n'est pas aussi simple que dans le modèle OBJVLISP. Alors que le système présenté fonctionne, nous voudrions savoir si diminuer le nombre de classes du noyau est possible.

13
Conclusion

«Human beings do not live in the objective world alone, nor alone in the world of social activity as ordinary understood, but are very much at the mercy of a particular language which has become the medium of expression for their society. It is quite an illusion to imagine that one adjusts to reality essentially without use of language and that language is merely an incindental means of solving specific problems of communication or reflection. The fact of the matter is that the 'real world' is to a large extent unconsciously built up on the language habits of the group... We see and hear and otherwise experience very largely as we do because the language habits of our community predispose certain choices of interpretation.» Edward Sapir.

Cette citation[1] souligne le fait que le langage influence directement notre perception du monde. C'est dans le but de permettre la prise en compte de nouveaux moyens d'expression et de modélisation que nous avons proposé dans cette thèse l'intégration de dépendances comportementales entre objets dans un modèle réflexif à classes.

Nous avons défini un modèle objet étendu dans lesquels les classes définissent les informations intrinsèques des objets, les dépendances spécifient les interactions entre ceux-ci et les contrôleurs assurent le maintien de la cohérence des dépendances. Nous avons précisé les caractéristiques et les interactions entre ces nouvelles entités et les autres entités du langage. En second lieu, au delà de proposer une implémentation de ce modèle, nous avons défini un protocole méta objet permettant de spécialiser notre approche des dépendances.

Un modèle. En plus des classes et des instances, notre modèle est basé sur l'introduction de deux nouvelles entités : les dépendances et les contrôleurs.

Une dépendance décrit de manière déclarative, explicite et locale l'influence de la réception d'un message par un des objets participants sur l'ensemble des objets participant à la dépendance Les dépendances sont définies de manière indépendante des classes des objets participants aux dépendances. Ainsi les classes définissent les comportements intrinsèques des objets et les dépendances les comportements relationnels ou interactifs entre ces objets.

En résumé, une dépendance :

- *explicite* les interactions entre *objets* et non entre variables d'instances. Elle est spécifiée dans les termes des interfaces des objets. Elle ne viole pas l'encapsulation de données.
- peut, en tant qu'entité à part entière, définir *ses propres variables et comportements.* Elle est le lieu pour définir des informations purement liées à l'interaction (conversion de données...).
- est *n-aire* et *non-orientée.* Elle gère la notion de groupe d'objets (ajout et retrait dynamique de participants à une dépendance).

[1] Cette citation provient d'une citation [BUDD 91] elle-même extraite de l'article "The Relation of Habitual Thought and Behavior to Language" de Benjamin Lee Whorf [WHOR 56].

- est définie de manière *distincte* de celle des classes. Elle peut être définie et déclarée ou retirée dynamiquement.
- peut être définie incrémentalement à l'aide d'un héritage spécialisé.

Un contrôleur utilise le comportement spécifié par les dépendances pour assurer la cohérence de l'ensemble des dépendances. Pour cela, un contrôleur contrôle les messages envoyés aux objets pouvant remettre en cause la cohérence de la dépendance. Un contrôleur peut être associé à un groupe d'objets dont il contrôle les comportements. De plus, un contrôleur abstrait le maintien de la cohérence des dépendances, ce qui permet d'introduire diverses possibilités de traitement global des dépendances comme la gestion des cycles. Les contrôleurs sont des entités distinctes à la fois des classes, des instances et des dépendances.

FLO : un langage et un MOP. Le modèle présenté est implémenté en STklos un langage objet de type CLOS. L'implémentation est basée sur le contrôle de l'envoi de messages. Elle utilise la réflexion structurelle en définissant des méta-classes spécialisées dans la gestion des dépendances (définition, héritage...) et la réflexion comportementale en introduisant des méta-objets comportementaux pour contrôler les envois de messages aux instances participant à des dépendances.

Le langage FLO met en œuvre toutes les propriétés définies par le modèle. Un utilisateur du langage programme des applications en spécifiant des classes et des dépendances. Une fois les dépendances déclarées entre instances, FLO en assure automatiquement la cohérence. Cependant, notre travail ne se limite pas à une simple implémentation du modèle. Le modèle présenté est extensible et modifiable (la gestion des dépendances peut être enrichie pour prendre en compte de nouveaux opérateurs, de nouveaux comportements de la propagation sont possibles). Pour cela, nous proposons donc un protocole méta-objet pour la gestion des dépendances. Ce protocole définit à la fois le comportement minimal de notre langage et les aspects pouvant être particularisés.

Applications. Notre modèle, en spécifiant les interactions entre objets de manière autonome et explicite sous forme de dépendances et en maintenant la cohérence de celles-ci, est particulièrement bien adapté pour la construction des interfaces hommes-machines. Les dépendances offrent un outil permettant une implémentation directe de modèles architecturaux multi-agents comme ALV[Hill 92] et PAC[Cout 89]. De la même manière, notre modèle offre des outils pour l'implémentation de schémas de conception.

Avenir et recherches futures. Au vu du travail accompli, nous présentons maintenant les points qui mériteraient un approfondissement.

Coût et Optimisation. Durant l'implémentation de FLO, nous nous sommes efforcés de minimiser le coût du contrôle des messages. Cependant, lors de la conception du MOP, toutes les techniques d'optimisations n'ont pas été mises en œuvre. Nous pensons donc qu'il serait intéressant d'analyser finement le coût du mécanisme proposé par rapport aux implémentations traditionnelles. Cette mesure devrait prendre en compte non seulement les temps de calcul mais des critères de complexité et de réutilisabilité des classes, ce qui n'est pas simple. Il serait possible d'utiliser des techniques d'optimisations comme la mémoization ans CLOS.

D'un autre côté, une approche *compilée* des dépendances devrait être étudiée. Dans ce cadre, nous pensons aux possibilités offertes par le compilateur réflexif de Smalltalk [Riva 95].

Modélisation. La possibilité de définir des dépendances entre objets donne un nouveau pouvoir d'expression. Cependant, la distinction entre informations intrinsèques à un objet et à une dépendance n'est pas toujours claire. Il serait donc intéressant : d'une part, d'étudier exactement comment les différentes méthodes de conception modélisent les interactions entre objets et de savoir si ces modélisations peuvent se définir sous formes de dépendances. D'autre part, la modélisation de plusieurs applications à l'aide de dépendances devrait amener à préciser une méthode adaptée à aux dépendances.

Architecture logicielle. Les dépendances décrivent les interactions entre objets. Nous voulons évaluer comment des telles entités peuvent être utilisées pour définir des connecteurs [Alle 94, Shaw 96, Nier 95] entre composants logiciels afin de proposer des architectures plus flexibles.

Distribution et objets actifs. L'utilisation de dépendances dans les applications distribuées ou pour contrôler des objets actifs reste pour nous la direction principale que nous souhaiterions donner à ces travaux. En effet, la distribution des applications ou le fait de considérer des objets actifs pose de nouveaux problèmes et nous souhaiterions savoir si l'architecture que nous avons proposée est suffisamment adaptable pour prendre en compte ces nouvelles dimensions. Il semble que des opérateurs de synchronisation ou d'atomicité soient nécessaires. De plus, à la manière des relations de Troll [Jung 96], les dépendances pourraient être utilisées pour spécifier la synchronisation entre processus.

Base de données. Les derniers travaux dans le domaine des bases de données font état de bases de données actives ; c'est-à-dire introduisant des mécanismes réactifs de maintien de la cohérence de relations entre éléments. Nous pensons qu'une suite à cette thèse pourrait être d'étudier les avantages du modèle proposé pour l'expression et le maintien de cohérence de relations dans le cadre de bases de données orientées-objets. La réalisation d'une implémentation de notre modèle à l'aide de OpenC++ par l'équipe BD du Cermics à Sophia-Antipolis pour offrir des dépendances dans le cadre d'une base de données orientée-objets est un premier pas intéressant dans cette direction.

Annexe

Nous avons délibérément choisi de présenter de manière détaillée les différents travaux qui constituent l'état de l'art en matière de dépendances dans les langages objets. Cependant, nous sommes conscient que le niveau de détail proposé peut parfois être gênant dans le cadre d'une lecture rapide de cette thèse. C'est pourquoi nous avons choisi de proposer une synthèse dans le cadre d'une lecture cursive de cette thèse et de présenter dans cette annexe ces travaux dans le détail.

L'état de l'art se compose de différents pôles :

- Premiers mécanismes : valeurs actives, démons, dépendances
- Contraintes et objets : KR, ThingLab, PluS
- Modélisation des interactions : Contract, ACT, Interactor, Troll...
- Coordination : Synchronizers, Procol

A

Premiers Mécanismes

Ce chapitre présente les premiers mécanismes qui ont été ou sont toujours utilisés pour représenter ou implémentés des dépendances entre objets. Nous abordons en détail les valeurs actives, les démons ainsi que les dépendances de Smalltalk.

A.1 Réactivité : valeurs actives, réflexes et démons

Les valeurs actives qu'offrent certains langages comme Loops et Klone [Naha 95] ou des générateurs de systèmes experts comme Smeci [Ilo 88] ou KEE [Int 85] peuvent servir à une mise en oeuvre de dépendances entre objets. Égérie un éditeur d'interfaces graphiques [Bout 93] est basée sur une vision unifiée de la notion de valeurs actives et de dépendances au travers de démons. Nous présentons ici les valeurs actives du langage Loops car il a été un précurseur dans l'utilisation de valeurs actives.

A.1.1 Valeurs Actives en Loops

L'originalité du langage Loops [Bobr 83, Stef 86a] a été d'ajouter un mécanisme permettant une programmation orientée par les accès : les valeurs actives. Cette dernière prend ses racines dans des langages comme Simula ou Interlisp-D [Sann 83] qui permettaient de convertir les accès à des enregistrements en un calcul. Les attachements procéduraux des langages de frames comme KRL [Bobr 77], FRL ou KL-One [Brac 83, Brac 85] ont également influencé l'apparition des valeurs actives.

Principes. La programmation dirigée par les accès permet d'invoquer des procédures lors de la lecture ou du changement d'état de données. De manière duale à l'approche objet où l'envoi d'un message provoque par effet de bord un changement d'état, dans la programmation orientée accès, un changement d'état provoque par effet de bord un envoi de message.

En Loops, la programmation orientée accès est basée sur la notion de *valeurs annotées*. Il existe deux sortes d'annotations : les *annotations de propriétés* et les *valeurs actives*. Les premières étendent la notion de liste de propriétés de langages à la Lisp*(property list)*, les secondes permettent l'invocation de méthodes lors de la lecture ou l'écriture de de variables d'instance. Une valeur active peut être installée sur n'importe quelle variable d'instance ou propriété. Une fois installée, n'importe quelle partie du programme accédant à la variable déclenche la valeur active. Les valeurs actives sont des objets : elles peuvent posséder leurs propres variables, leur comportement peut être spécialisé. Elles peuvent elles-mêmes être annotées. Cette réification des valeurs actives intègre parfaitement celles-ci dans un langage objet.

Implémentation. Une valeur active contient une variable d'instance, localState, qui stocke la valeur initiale de la variable sur laquelle porte la valeur active. Deux méthodes définissent le protocole d'accès aux valeur actives : PutWrappedValue invoquée lors de l'affectation d'une valeur et GetWrappedValue invoquée lors de la lecture de celle-ci (par défaut, GetWrappedValue rend la valeur de localState, PutWrappedValue en change la valeur). Ces deux méthodes définies sur la classe ActiveValue peuvent être spécialisées suivant les besoins.

Des langages comme Flavors [Moon 86], CLOS (en particulier avec les méthodes auxiliaires :after, :before et :around [Stee 90, Keen 89]) ou Smalltalk permettent une spécialisation des méthodes d'accès aux variables, cependant à la différence des valeurs actives de Loops, de telles méthodes sont applicables à toutes les instances d'une classe et ne sont dissociés des fonctionnalités d'un objet. Tout comme les *traps* du langage KSL [Ibra 88] (voir 9.1.1), les valeurs actives permettent d'associer dynamiquement des méthodes à des instances.

Discussion. Le comportement comportement réactif permet d'utiliser les variables actives pour maintenir des invariants entre variables ou entre variables et objets. La conjugaison de l'utilisation d'attributs et de valeurs actives permet de mettre en dépendance deux objets. Cependant, ces dépendances ne sont pas clairement exprimées. Hormis le déclenchement automatique des mises à jour et la possibilité de lier des instances spécifiques d'une classe, les valeurs actives ne répondent que partiellement aux problèmes posés (voir en 1.2) et ont presque les mêmes désavantages que la solution consistant à utiliser des attributs et des méthodes pour implémenter des dépendances.

Les valeurs actives ne permettent pas de lier un objet dans son identité globale à un autre, mais seulement une ou plusieurs variables à un objet. La dépendance ne s'exprime qu'à partir de modification de variables d'instances et non de messages (autres que des accesseurs). Aucun mécanisme ne permet de contrôler le déroulement du déclenchement.

A.2 Daemon

« A daemon which is attached to an object (whole), monitors the object bound to a particular instance variable (part). Whenever a specific message is sent to the part, the deamon invokes a method of a whole. » [NATA 91].
.

Murata et Kusumoto dans [MURA 87, MURA 89, NATA 91] introduisent la notion de *daemon* pour le maintien la cohérence et la communication dans des hiérarchies de composition. Ces *daemons* maintiennent la cohérence entre un objet composite et ses composants, tout en assurant un principe d'*indépendance de situation* de la part des *composants* qui ne connaissent pas l'objet composite auquel ils appartiennent.

Pour les auteurs de Daemon, il existe deux flux de messages: tout d'abord, l'application normale de méthodes établit un flux d'un objet composites vers ses composants et récursivement. La communication orientée en sens inverse; c'est-à-dire des composants vers les composites est prise en charge par l'invocation de démons. Du fait du principe d'indépendance de situation des composants, les Daemons sont proposés pour notifier les objets composites des changements de leurs composants.

L'exemple de la gestion d'un feu de signalisation tricolore tiré de [GOLD 83, MURA 89] nous permet de présenter cette approche. Il s'agit de faire respecter une contrainte entre les trois lampes: une seule ne doit être allumée à la fois. Nous montrons une solution traditionnelle puis celle utilisant les Daemon.

Approche Top-Down. Lorsque l'on considère que le feu est responsable de la cohérence de ses lampes. Des objets extérieurs ne doivent pas directement allumer ou éteindre les lampes. Donc tous les accès aux lampes doivent obligatoirement être adressés au le feu (cf. A dans la figure A.1). Par exemple, lorsque l'on veut savoir si la lampe rouge est allumée, on s'adresse au feu (f isRedOn) et le feu interroge la lampe rouge (lr isOn).

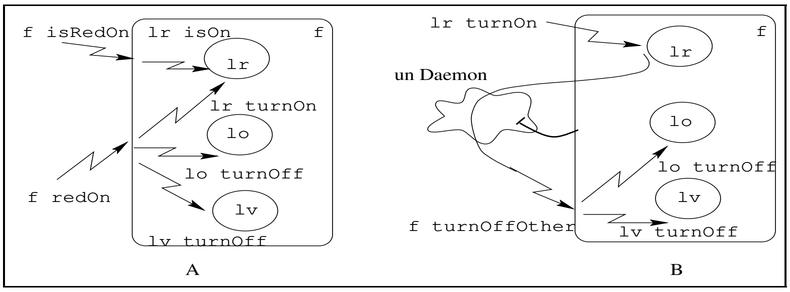


Figure A.1: Deux approches de la gestion de la cohérence entre un objet composite et ses composants: (A) top-down et (B) à l'aide de Daemons.

Cette approche conduit à un aplatissement de la hiérarchie des composants et la définition d'objets composites monolithiques dont certaines méthodes ne sont dues qu'à la nécessité d'une délégation vers leurs composants. Blake et Cook mettent clairement en avant ces problèmes dans [BLAK 87], ils écrivent à propos de cette solution: *« It flattens the part hierarchy and so removes the conceptual advantages of factoring that knowledge in an intuitive manner ... The net result is that the part hierarchy is replaced by a single monolithic whole as far as the external world is concerned. »* [BLAK 87]. Ils proposent quand à eux des messages composés (voir en 5.5.1).

Approche Bottom-Up. Avec cette solution, le feu ne cache plus ses lampes; ainsi un objet extérieur peut adresser directement des messages aux lampes. Se pose alors le problème du maintien de la cohérence du feu (qu'une seule lampe soit allumée à la fois) avec des lampes ne connaissent pas le feu (*principe d'indépendance de situation*).

Murata et Kusumoto proposent alors les *daemons* qui ont pour tâche de notifier les objets composites des changements de leurs composants. Ainsi un daemon est attaché à l'objet composite dont il contrôle les composants (cf. B dans la figure A.1).

Précisions. Les *daemons* sont attachés à un objet unique et ne peuvent donc être partagés. Ils sont décrits lors de la définition de la classe des objets composites dont ils contrôlent les parties. Un daemon est associé à une variable d'instance représentant le composant. Il ne contrôle d'une seule méthode (ainsi dans l'exemple du feu tricolore, deux daemons (pour les messages turnOn et turnOff) sont définis sur le feu pour chacunes des trois lampes).

Remarques, critiques et interrogations. L'introduction des *daemon* semble répondre aux problèmes accès, invocations des méthodes des objets composants dans des hiérarchies de composition [BLAK 87]. Cependant, des remarques sont nécessaires.

Nous émettons quelques réserves quand à la justification du fait qu'un objet soit défini comme un *tout* ou une *partie*. Murata et Kusumoto définissent un objet comme un *tout* à partir du moment où celui-ci possède un attribut qui pointe sur un autre objet (une *partie*). Nous pensons que la relation de composition ne se limite pas à cette définition, ainsi Civello dans [Cive 93] définit plusieurs types de relations de composition. Nous pensons que vouloir tout voir comme une *partie* et un *tout* est réducteur. Par exemple, Murata et Kusumoto présentent les vues du modèle MVC [Kras 88] (cf 11.2.2) comme des *tout*s et le modèle comme une *partie*, ils justifient cette représentation par le fait que lors d'un changement du modèle ses vues doivent être averties bien que celui-ci ne connaissent pas ces vues.

Nous avons montré en 5.5.1 comment notre solution est plus générale et s'applique aussi bien dans le cas de relation de composition que dans celui d'autres relations entre objets. Il est à noter que les informations données dans [Mura 87, Mura 89, Nata 91] ne donnent aucune indication sur la nature des *daemons* : s'agit-il d'objets de plein droit ? Est-il possible de leur associer des variables ou des méthodes. Ainsi aucune suggestion n'est faite au sujet de l'implémentation et la syntaxe des *daemons*.

De plus, sans parler d'implémentation, le mécanisme de *daemons* n'est pas clairement expliqué et nombre de points sont laissés dans l'ombre. Nous aurions préféré avoir de plus amples indications afin de comparer pleinement la solution proposée et notre solution.

A.3 Des dépendances

« (Observer) defines of one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. » [Gamm 94].

Le langage Smalltalk [Gold 83] a été un des premiers à proposer une des dépendances entre objets. Nous présentons ici le principe, la mise en oeuvre de cette approche dans le langage Smalltalk ainsi que les problèmes qu'elle génère. Malgré certains problèmes induits par cette approche, il est remarquable de noter l'utilisation systématique des dépendances, en particulier dans le modèle MVC qui constitue le modèle architectural de Smalltalk.

A.3.1 Principe

Les dépendances permettent la communication entre un ensemble d'objets : un objet que nous nommerons *influant*[1] ou *sujet* [Gamm 94] et un groupe d'objets appelés ses *dépendants* ou *observateurs* [Gamm 94]. Une dépendance permet l'envoi automatique de messages aux objets dépendants lorsque l'objet influant reçoit un certain message. Les objets dépendants pouvant à leur tour être influant d'autres objets, un envoi de message a comme conséquence une *propagation de messages*.

L'introduction des dépendances a pour but de permettre une indépendance entre les différents acteurs. Cependant les intervenants doivent avoir certaines fonctionnalités. L'objet influant

- doit offrir des possibilités d'ajout ou de retrait d'objets dépendents,
- doit *notifier* ses dépendants d'un changement d'état,
- mais lors de la notification de changement d'état, ne fait ni explicitement référence à ses dépendants, ni de supposition quand à leur nombre ou leurs structures.

Les objets dépendants doivent définir une action en réponse à une demande de mise à jour de l'objet influant.

Comme la notification de l'objet influant ne nécessite pas de spécifier explicitement d'objets, elle peut être vue comme un mécanisme de *broadcast* de messages. Ceci offre la liberté d'ajouter ou de retirer des objets dépendants de manière dynamique. En contre partie, les dépendants n'ayant pas connaissance les uns des autres ou de mécanisme centralisateur, une simple notification peut provoquer des propagations inutiles.

A.3.2 En Smalltalk

La gestion des dépendances s'exprime par un protocole (ensemble de méthodes) qui est défini sur la racine du graphe d'héritage : la classe `Object` ; ainsi tout objet Smalltalk peut prétendre avoir des dépendants.

Le mécanisme de propagation automatique, illustré par la figure A.2 est simple : pour informer les dépendants d'un changement d'état, l'objet influant *s'envoie* le message `changed`[2], ceci a pour conséquence d'appliquer la méthode `udpate:`[3] à tous les dépendants de cet objet (cette gestion automatique est spécifiée par la méthode `changed:with:` de la classe `Object`). Une dépendance est alors une relation orientée de l'objet influant vers les objets dépendants.

Les protocoles de gestion des dépendances se classent en deux groupes : l'un lié à la déclaration des dépendances et l'autre lié à la propagation de message. Durant la phase déclarative, le programmeur ajoute un objet comme dépendant d'un autre (`addDependent:`), retire un dépendant de la liste des dépendants (`removeDependent:`),

[1]Nous donnons ce nom à cet objet afin d'y faire référence plus aisément ; la littérature ne qualifie malheureusement pas cet objet. Par contre, le terme dépendant est lui le terme usuel dans les manuels traitant de Smalltalk.

[2]Il existe plusieurs versions permettant de préciser quelle modification a eu lieu. Ici, `xxx` qualifie le changement. Cette qualification est utilisée par la méthode `update:`.

[3]Plus précisément, la méthode `update:with:from:` est appelée. Elle appelle la méthode `update:with:` et cette dernière appelle alors `update:`.

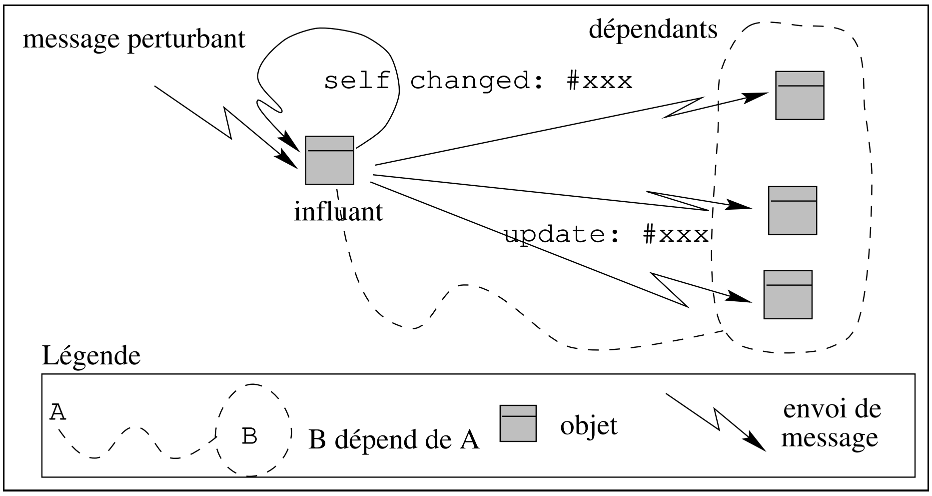


Figure A.2: Mécanisme de propagation automatique des dépendances Smalltalk.

enleve tous les dépendants (`release`). De plus, étant donné un objet, il est possible de connaître ses dépendants (`dependents`). Les protocoles de la phase dynamique sont principalement les méthodes: `changed` et `update:`. Un protocole plus élaboré permet de spécifier si la réaction doit avoir lieu: avec la méthode `changeRequest` le receveur veut signifier à ses dépendants qu'il veut changer, il interroge ses dépendants par le biais de la méthode `updateResquest` pour savoir s'il a le droit.

A.3.3 Mise en oeuvre par le programmeur

La déclaration de dépendance entre deux objets se fait par l'intermédiaire de la méthode `addDependent:`. Lorsqu'un programmeur veut mettre en dépendance deux objets, il doit travailler au niveau des classes des instances qu'il souhaite mettre en dépendance.

1. L'émission d'un changement de l'objet influant doit être spécifiée par le programmeur. Il doit préciser dans les méthodes de la classe de l'objet influant les changements en insérant le code d'émission (`self changed`).
2. De la même manière, le programmeur doit encore intervenir afin de spécifier la réaction des dépendants au changement d'état de l'objet influant. Pour cela, il doit spécialiser la méthode `update`[4] sur la classe des objets dépendants.
3. Le dernier point qu'un programmeur est amené à gérer des cycles lors des réactions. En effet, comme tout objet peut être à la fois influant et dépendant, il peut exister des situations de dépendances mutuelles. Un changement pouvant alors provoquer un cycle lors de la propagation de messages. Le mécanisme de dépendances de Smalltalk ne prévoit rien et c'est au programmeur de gérer ce type de situations.

A.3.4 Problèmes

« A relation represents an inherent constraint between objects of two or more classes. This constraint is not something to be hidden, but rather to be specified abstractly, without imposing an implementation. In a smalltalk program, the constraints would be buried within method code, hard to recognize. This kind of "information hiding" *hides semantic information and exposes implementation information, exactly backwards. »* [Rumb 87].

Ce modèle de dépendances n'est pas sans poser de nombreux problèmes que l'on classe en deux catégories: ceux liés directement à la mise en oeuvre des dépendances et ceux plus spécifiques de la spécification des dépendances. Commençons par les problèmes que posent l'implémentation des dépendances.

Anticipation. La nécessité de modifier les classes afin de gérer l'émission du message `changed` ou la réaction des dépendants nécessite de savoir *a priori* quels sont les objets susceptibles d'être mis en dépendance[5]. En conséquence, la déclaration dynamique *a posteriori* est impossible.

Couplage. L'obligation de spécifier au niveau des classes les réactions des dépendants met en avant le couplage qui existe entre les classes des dépendants et la classe de l'objet influant. De telles dépendances ne rendent pas réellement indépendantes de telles classes. Ainsi un objet influant n'a comme dépendant que des objets dont les comportements sont adaptés en fonction de cet objet.

[4] Par défaut, elle ne fait rien.

[5] La solution qui consiste à émettre systématiquement une notification lors de l'affectation d'une variable d'instance est limitée. Le programmeur doit pouvoir spécifier le moment de mises à jour.

Sous-classage inutile. Le simple fait que les dépendances ne soient pas exprimées aux niveau des instances mais à celui des classes conduit à la définition de classes dont la seule raison d'être est la mise en oeuvre de réactions adaptées. La question de la sémantique de ces classes est posée : s'agit-il de classes ayant une identité propre ou de classes induites par la faiblesse d'expression du langage ?

Absence de contrôle de la propagation. La propagation des messages n'est pas facilement contrôlable. L'exemple le plus flagrant est l'absence de gestion des cycles.

La qualité initiale mise en avant pour une utilisation de dépendances, c'est-à-dire l'indépendance entre l'objet influant et les objets dépendents, est mise à mal. D'autre part, des problèmes de spécification des dépendances se posent. Ces problèmes proviennent essentiellement du fait que les dépendances ne sont pas traitées au niveau conceptuel et restent un mécanisme simple d'implémentation.

Sémantique éparpillée. Il est possible de savoir si un objet dépend d'un autre mais sans la lecture des classes des objets mis en dépendance, il est impossible de connaître la nature de la dépendance. La sémantique d'une dépendance est littéralement enfouie dans le code de toutes les classes qui participent à la dépendance.

Absence de repères. Les émissions de changement d'état de l'objet influant sont elles aussi éparpillées dans le code, il est alors difficile de savoir exactement quand la dépendance est utilisée. Ces problèmes sont amplifiés lorsque plusieurs niveaux de dépendances s'entremêlent.

Compréhension difficile. *«Protocols are sometimes difficult to understand because one has to browse the whole class library to track the message flow»* [Barr 93].

Les classes qui utilisent intensément les dépendances sont difficilement compréhensibles car le programmeur doit suivre les propagations de messages au travers des différentes classes. Ce problème est particulièrement visible dans les classes `SelectionInList` et `SelectionInListView` de VisualWorks [Par 94]. De plus, les dépendances n'étant pas manipulables en tant qu'objets, l'élaboration d'outils d'aide à la conception (représentation des flots...) est rendue difficile.

A.3.5 Le feu de signalisation

L'exemple suivant montre comment les dépendances permettent d'exprimer qu'une seule des trois lampes d'un feu tricolore de signalisation ne doit être allumée à la fois. Cet exemple, tiré de [Gold 83], nous a servi de comparaison pour la gestion de la cohérence dans les hiérarchies de composition (voir 5.5.1). Il met en avant une utilisation des dépendances qui ne se réduit pas à une cohérence entre un objet et sa représentation graphique.

Voici tout d'abord le code de la classe `Light`. Une lampe émet une notification de changement d'état pour signaler qu'elle vient d'être allumée (`self changed` dans la méthode `turnOn`).

```
class name                Light
superclass                Object
instance variable names   status
class methods
 setOn                                        setOff
    ↑ self new SetOn                             ↑ self new SetOff
instance methods
 turnOn
    self isOff ifTrue: [status := true. self changed]
 turnOff
    self isOn ifTrue: [status := false]
 isOff
    ↑ status not
 isOn
    ↑ status
 update: aLight
    aLight == self ifFalse: [self turnOff]
 setOn                                        setOn
    status := true                                status := false
```

L'utilisation des dépendances n'est pas adaptée pour fournir un code dans lequel les interactions entre objets sont clairement exprimées. Le code de la méthode `update:` doit être interprété après compréhension des mises en dépendances de la classe `TrafficLight`.

Il est nécessaire de savoir que toutes les lampes d'un feu sont dépendantes de toutes les autres lampes du feu, cette mise en dépendance est le fait de la méthode `lights:` de la classe `TrafficLight`. La méthode `update:` indique alors simplement que lorsqu'une lampe doit se remettre à jour, elle s'éteint si elle n'est pas l'objet passé en argument lors de l'appel de la méthode `update:`. De ce fait, lors d'une modification d'une lampe toutes les autres reçoivent une demande de mise à jour, elles s'éteignent toutes sauf une.

```
class name                TrafficLight
superclass                Object
instance variable names   lights
class methods
with: numberOfLights
 ↑ self new light: numberOfLights

instance methods
turnOn: lightNumber
    (lights at: lightNumber) turnOn
release
    super release.
    lights do: [:eachLight | eachLight release].
    lights := nil
lights: numberOfLights
    lights := Array new: (numberOfLights max:1).
    lights at: 1 put: Light setOn.
    2 to: numberOfLights do:
               [:index | lights at: index put: Light setOff].
    lights do:
       [:eachLight|
         lights do:
           [dependentsLight|
               eachLight ~~dependentLight
                  ifTrue: [eachLight addDependent: dependentLight]]]
```

A.3.6 CLOS et les dépendances

À des fins d'optimisations, les classes et les fonctions génériques du langage CLOS sont sujettes à une *mémoization*[6]. CLOS utilise un mécanisme de dépendance afin de mettre à jour ces informations *mémoizées* ; par exemple, lorsqu'une classe est redéfinie, elle doit mettre à jour ses dépendants (ses sous-classes et fonctions génériques associées).

Les quatres fonctions génériques responsables de ce comportement sont spécifiées dans «le protocol de maintenance des dépendants» [Kicz 91] page 160. Ainsi les fonctions génériques `add-dependent` et `remove-dependent` permettent d'associer ou d'enlever un dépendent à un objet. La fonction générique `map-dependent` permet d'appliquer une fonction à tous les dépendents d'un objet. Cette fonction est utilisée pour appliquer la fonction générique `update-dependent` à tous les dépendents d'un objet. Par exemple, lorsqu'une méthode est ajoutée à une fonction générique le code suivant est exécuté :

```
(map-dependents  generic-function
                #'(lambda (dep)
                     (update-dependent  generic-function
                                       dep
                                       'add-method
                                        new-method)))
```

Les dépendances de CLOS sont plus anecdotiques que celles de Smalltalk de part leur utilisation très confidentielle et leur mécanisme moins élaboré.

A.3.7 Les dernières innovations de VisualWorks

La dernière version de VisualWorks (2.0) introduit un premier mécanisme de réification des dépendances. L'idée principale de cette réification est de permettre, en utilisant les anciennes dépendances, d'envoyer un message particulier à un objet lorsqu'un autre objet reçoit un certain message. Cette nouvelle possibilité est utilisée dans le modèle M2VC cas particulier du modèle MVC au travers de l'utilisation des `ValueHolder` [Brif 96]. Ce mécanisme provient de l'interaction entre la classe `Object` et la classe `DependencyTransformer`.

La classe Object définit de nouvelles méthodes pour gérer ces dépendances réifiées parmie lesquelles : `ExpressInterestIn:for:sendBack` et `retractInterestIn:for:`.

[6] Il d'une technique qui vise à limiter le recalcul d'une opération. Pour cela, lors de la première exécution le calcul est effectué normalement puis stocké, ainsi lors d'une prochaine demande de ce calcul et tant que le résultat est valide par rapport à son contexte d'utilisation, le résultat stocké sera rendu évitant un calcul inutile.

```
expressInterestIn: anAspect for: anObject sendBack: aSelector
    | dt |
    dt := DependencyTransformer new.
    dt setReceiver: anObject aspect: anAspect selector: aSelector.
    self addDependent: dt
```

Cette méthode crée une *dépendanceT*[7] (instance de `DependencyTransformer`) et l'ajoute dans la liste de dépendants d'un objet. La situation de dépendance représentée par la figure A.3 entre un objet `O1` et un objet `O2` est mise en place par le message suivant : `O1 expressInterestIn: #XXX for: O2 sendBack: #YYY`. On exprime que lorsque `O1` change avec la qualification `XXX`, la méthode de sélecteur `YYY` est invoquée sur `O2`.

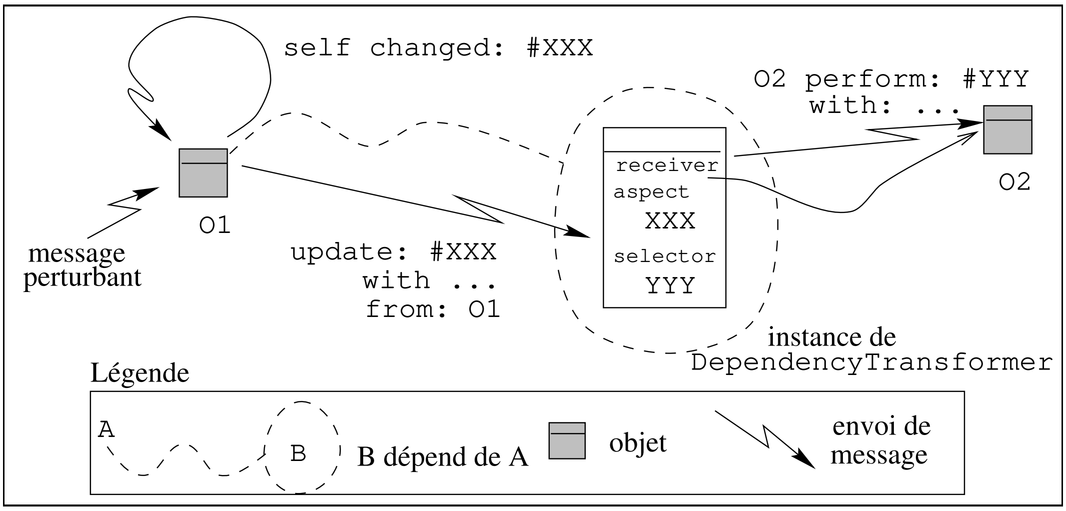


Figure A.3: DependencyTransformer et dépendances

La classe DependencyTransformer définit la structure et les méthodes de ces nouvelles dépendances.

class name	`DependencyTransformer`
superclass	`Object`
instance variable names	`receiver selector numArguments aspect`

Une instance de la classe `DependencyTransformer` a une variable d'instance `receiver` représentant le futur receveur du message (`O2` dans la figure A.3) dont le sélecteur est contenu par la variable d'instance `selector` (`YYY` dans la figure). La variable d'instance `aspect` contient elle un symbole (`XXX`) permettant de savoir si l'envoi de message doit être envoyé ou pas.

La méthode `update:with:from` de la classe `DependencyTransformer` est spécialisée de telle sorte qu'un message de sélecteur `selector` soit envoyé à l'objet `receiver` lorsque l'instance de la dépendance reçoit un message `update:with:from` dont l'aspect est celui de la dépendance.

```
update: anAspect with: arg from: anObject
 aspect == anAspect ifFalse:[↑self].
 numArguments == 0 ifTrue:[↑receiver perform: selector].
 numArguments == 1 ifTrue:[↑receiver perform: selector with: arg].
 numArguments == 2
   ifTrue:[↑receiver perform: selector with: arg with: anObject].
```

Quand `O1` reçoit un message perturbant (voir figure A.3), ses dépendants reçoivent un message `update:with:from:` et en particulier, la dépendanceT. Cette méthode invoquera la méthode `aSelector` sur l'objet `anObject`. Cependant, cette invocation ne se fait que si le message `update:` concerne réellement cette dépendanceT, c'est-à-dire si le paramètre associé au message `update:` (`XXX` dans la figure A.3) correspond à celui de la dépendanceT.

Discussion. Cette approche propose une réponse aux problèmes de l'ancien mécanisme. Cependant, il s'agit d'associer deux messages et deux objets, rien n'est pas prévu pour définir le code à émettre comme c'est le cas dans les *triggers* de ANIMUS [BORN 86b], des *propagator contraints* de PROCOL [VAN 91].

[7]Nous utiliserons ce terme pour bien distinguer cette dépendance des dépendances traditionnelles.

B

Contraintes

De nombreux travaux ont proposé une intégration de contraintes dans des modèles objets. L'utilisation de contraintes permet de spécifier certaines dépendances entre variables d'instances. Nous présentons dans ce chapitre les travaux les plus révélateurs de cette intégration. Nous montrons les limites de ces travaux par rapport aux propriétés fondamentales du modèle objet que nous avons choisi (voir 3.1).

B.1 Programmation et contraintes

La programmation par contrainte s'est fortement développée ces dernières années. Freeman-Benson, John Maloney et Alan Borning classent dans [Free 90a] les langages ou systèmes à base de contraintes en cinq grandes catégories suivant leur utilisation : le positionnement graphique, les simulations, les aides au raisonnement et à la conception, la gestion des interfaces hommes-machines et les langages des programmation dans lesquels les contraintes demeurent le concept principal. Nous ajoutons les systèmes à base de connaissances.

Positionnement géométrique. Sutherland avec Sketchpad fut un des précurseurs dans l'utilisation des contraintes pour permettre à l'utilisateur de spécifier des figures géométriques à l'aide d'entités graphiques primitives et de contraintes. ThingLab reprit les idées de Sketchpad et les mêla aux concepts de la programmation objet avec Smalltalk [Born 79, Born 86b]. D'autres systèmes comme Magritte [Gosl 83], Juno [Nels 85] ou [Liot 94] furent développés.

Simulation. Lors de la simulation, les lois, physiques par exemples, peuvent être exprimées sous forme de contraintes. Animus fut une extension de ThingLab qui intégrait la notion de temps et des contraintes événementielles permettant des animations.

Aide à la conception. Dans ce cadre, les contraintes sont utilisées comme des éléments de conception, de représentation des lois physiques. Par exemple TK!Solver est un système commercialisé offrant des contraintes pour des applications financières ou d'ingénierie. Dans Constraints [Suss 80], les contraintes sont utilisées pour représenter des circuits électriques.

Gestion des interfaces hommes-machines. Les contraintes sont aussi utilisées pour le maintien de la cohérence entre les données de l'application et leur représentation graphique, entre plusieurs représentations d'une même donnée, pour exprimer des dépendances entre les objets graphiques (ThingLab, Animus, Garnet). Garnet utilise les contraintes dans le cadre de la construction d'interfaces. Epstein et Lalonde dans [Epst 88] utilisent les contraintes pour le positionnement des fenêtres en Smalltalk. Rendezvous assure la cohérence et la communication entre les objets de l'application et leurs représentations par le seul biais de contraintes [Hill 94, Hill 92].

Programmation à base de contraintes. De nombreux langages utilisant les contraintes ont vu le jour. Steele dans [Stee 80] fut un précurseur dans cette utilisation des contraintes. Kaleidoscope fut une proposition de fusion entre la programmation impérative et les contraintes [Free 90b]. Horn [Horn 92] propose de décrire des objets à l'aide de contraintes et des termes de réécriture. Solver permet l'expression de contraintes dans une approche objet [Puge 94, Puge 93]. Solver est en particulier utilisé par Tropes.

Cependant, la plupart des langages utilisant des contraintes ont pour base la programmation logique (PrologIII, Chip, LOGIN, LIFE). Nous renvoyons le lecteur à [Jaff 94] pour un état de l'art complet et récent sur l'état de la programmation logique à base de contraintes.

Contraintes et représentation. L'utilisation de contraintes a permis l'augmentation du pouvoir d'expression des langages de représentations de connaissances. Ainsi TROPES [Gens 93] est un langage de représentation des connaissances intégrant des contraintes. Prose [Berl 92] est une boîte à outil proposant différents algorithmes intégrables à des systèmes experts (comme ce fut le cas avec SMECI [Ilo 88]). De même Cool [Aves 90] est une boîte à outils de satisfaction de contraintes intégré dans KEE [Int 85]. Un peu en marge, CSPoo [Koke 93] propose une modélisation objet du solveur de contraintes ce qui offre une meilleure spécialisation des différents comportements de celui-ci.

B.2 Nos choix de présentation

Le but de cette thèse n'est pas d'introduire des contraintes au sens logique du terme dans un modèle objet. C'est pourquoi nous avons limité notre étude aux travaux concernant l'introduction de contraintes dans les langages à objets tout en conservant les propriétés fondamentales du modèle. Cependant, le panorama offert ne se veut absolument pas exhaustif. Notre choix s'est porté vers la présentation de systèmes de contraintes dont les tentatives d'intégration sont caractéristiques ou historiques. De plus, nous nous sommes attachés aux systèmes basés sur un algorithme de propagation favorisant un maintien de cohérence[1].

A nos yeux, deux questions se posent lors l'intégration de contraintes dans un langage objet. Tout d'abord, il faut savoir sur quels entités les contraintes peuvent porter et en quels termes elles sont exprimées. Les contraintes ont souvent été décrites entre variables d'instances des objets et non entre des objets dans leur globalité. La seconde question importante est : comment l'encapsulation est-elle prise en compte par les contraintes ? En effet, il existe un antagonisme très fort entre ces deux concepts : un objet étant une entité masquant sa représentation interne et traitant des requêtes et une contrainte étant une entité restreignant les valeurs de variables (d'instances pour un objet). Bien que les travaux de Wilk se situe plus dans le domaine de la satisfaction de contraintes que du maintien de contraintes, il définit assez clairement notre point vue : *« ...we still might question whether the solver was suitable for our application. One complaint is its violation of object encapsulation. Encapsulation was first violated by the constraint expressions... Encapsulation was also violated by the solution, which told us how to set individual state variables. »* [Wilk 91]

Dans notre étude nous étudions plus les systèmes au vu de leur choix face à choix des entités sur lesquelles les contraintes peuvent porter, l'encapsulation, à la place donnée aux dépendances par rapport aux entités du langage (classes ou instances) plutôt qu'aux performances et propriétés des algorithmes mis en jeu. Nous offrons une présentation uniforme ; pour chacun des langages nous présentons le contexte général de son utilisation, les différentes dépendances offertes que nous illustrons par un exemple, quelques détails d'implémentation lorsque ceux-ci précisent le mécanisme mis en jeu et nous discutons des choix faits.

B.3 Formules et démons de KR

« An important feature of Garnet is that any slot of any object can contain a constraint *instead of a normal value. A constraint is a relationship that is declared once and automatically maintained by the system... Garnet encourages programmers to directly access and set slots of objects. »* [Myer 92].

KR [Gius 89, Gius 92] qui est un langage à prototype implémenté en Common Lisp [Stee 90] est la pierre de base de Garnet [Myer 90, Myer 92] un environnement[2] de développement d'interfaces utilisateurs. KR, comme son ancêtre CORAL [Szek 88] intègre un mécanisme de maintien de contraintes entre objets basé sur une propagation locale et l'évaluation paresseuse des contraintes. Ce langage n'est par ailleurs pas exclusivement dédié à Garnet.

KR et son modèle objet. Le langage KR s'inscrit dans la lignée des langages à prototypes [Lieb 86a, Lieb 86b, Unga 87, Cham 91]. Dans les termes du "Traité d'Orlando"[Stei 89], KR est un langage à prototype avec partage implicite (les objets héritent[3] de leurs parents et l'on ne peut pas déclarer explicitement comment les champs sont hérités) et dynamique (l'héritage peut être changé à n'importe quel moment).

En KR, les objets nommés *schémas* sont des agrégats de champs (*slots*) qui correspondent à la notion de variables d'instances des langages à classes. Les objets sont créés en spécifiant, grâce à la fonction `create-instance`, le prototype dont ils héritent et les propriétés qu'ils définissent ou redéfinissent à leur niveau [Gius 92, Myer 90].

Le code suivant tiré de [Gius 92], définit un objet nommé `objet-couleur` (lignes 1 et 2). Cet objet n'hérite d'aucun autre objet, il définit un champ nommé `couleur` dont la valeur est `bleu`. De même, un nouvel objet et créé (lignes 3 et 4), il hérite de l'objet `objet-couleur` et définit un champ `épaisseur` dont la valeur est `1`. Finalement l'objet nommé `Rectangle1` est défini, il hérite de l'objet `objet-boîte` et possède deux nouveaux champs `x` et `y`.

```
1 (create-instance 'objet-couleur NIL
2  (:couleur 'bleu))  ;; un premier objet ayant une couleur

3 (create-instance 'objet-boîte objet-couleur
4  (:épaisseur 1)) ;; objet-boîte hérite de objet-couleur et définit un nouveau champ

5 (create-instance 'Rectangle1 objet-boîte
6  (:x 10)       ;; Rectangle1 hérite de objet-boîte
7  (:y 20))      ;; et définit deux champs x et y
```

[1]Les systèmes permettant de la satisfaction de contraintes proposent souvent des algorithmes de propagation. Cependant, de tels langages se trouvant trop loin de notre centre d'intérêt principal : la programmation objet, nous ne les présenterons pas dans cette thèse.

[2]Garnet est l'acronyme de Generating an Amalgam of Real-Time, Novel Editors and Toolkits.

[3]Nous ne faisons pas ici de distinction entre l'héritage des langages à classes et son pendant pour les langages à prototypes la délégation. Tous deux permettent une définition incrémentale d'objets.

KR offre deux mécanismes distincts afin de permettre la propagation entre les objets grâce à l'utilisation des champs : la *propagation de valeurs* qui assure que les objets sont consistants après un changement et l'*invocation de démons* qui permet que certaines actions soient déclenchées lors de l'affectation de champs. Ce dernier mécanisme est indépendant du premier. En Garnet, un programmeur n'a pas à se soucier des démons, ceux-ci sont prédéfinis par Opal[4] et leur rôle est d'adapter de manière automatique la représentation graphique des objets de l'application.

La propagation de valeurs est basée sur la notion de dépendances entre *valeurs*. Ces dépendances sont exprimées par des *formules* qui sont associées aux champs des objets. Dans l'exemple, on définit un objet qui hérite de l'objet `Rectangle1`. Ce nouvel objet redéfinit le champ `x` avec la valeur `34` (ligne 1). Le champ `y` est redéfini à l'aide d'une formule : elle spécifie que la valeur du champ `y` est celle du champ `y` de l'objet dénoté par le champ `left-obj` à laquelle on ajoute `15` (ligne 3).

```
1 (create-instance 'Rectangle2 objet-boîte
2    (:x 34)
3    (:y (o-formula (+ (gvl :left-obj :y) 15)))
4    (:left-obj Rectangle1))
```

Lorsque la valeur de `y` de `Rectangle1` prend une nouvelle valeur, le champ `y` de `Rectangle2` n'est plus valide. Sa valeur n'est cependant calculée en fonction de cette nouvelle valeur que lorsque l'on en a besoin.

Quand la valeur d'un champ change, tous les champs qui en dépendent sont immédiatement invalidés, bien qu'ils ne soient pas immédiatement réévalués. Cette stratégie, l'évaluation paresseuse, assure alors que les valeurs ne sont recalculées que lorsque le système en a effectivement besoin.

KR a été le premier à introduire des contraintes pouvant porter sur des variables pointant sur des objets [Zand 91] *(Pointer Variables)* : les objets référencés par la contrainte peuvent changer dynamiquement. De plus, les contraintes proposées ici sont unidirectionnelles et la propagation est locale. KR propose une gestion de cycles dont l'idée est simple, lors de la réévaluation d'une formule si le système a besoin d'une valeur dépendant de cette formule alors il rend l'ancienne valeur associée au champ. Cette solution garantit que le système ne boucle pas, mais le résultat dépend en fait de l'endroit à partir duquel a été commencé la propagation.

Les démons sont des morceaux de code qui sont invoqués quand certaines actions sont exécutées sur un schéma. Par exemple, quand la valeur d'un champ change (directement ou comme le résultat d'une propagation de valeur), KR vérifie si un démon doit être invoqué.

Une application peut ainsi contrôler le changement des valeurs de champs grâce à deux démons invoqués à des moments différents : le démon *invalidate-demon* qui est appelé à chaque fois qu'une valeur est invalidée. Ce démon est contenu dans champ `:invalidate-demon` d'un objet (ligne 7). Le second démon *pre-set-demon* est invoqué immédiatement avant que la valeur de la formule soit modifiée. Le champ `update-slots` contient la liste des champs dont un changement de valeur déclenche un démon (ligne 6). Le code[5] suivant illustre comment un démon `invalide-démon` peut être défini puis invoqué lors du changement de valeur d'un champ soumis à une formule.

```
1 (defun inv-demon (schema slot v)
2     (format t "schema ~s, slot ~s est invalidé.~%" schema slot))

3 (create-schema 'A
4     (:left 10)
5     (:top (o-formula (1+ (gvl :left))))
6     (:update-slots '(:top))
7     (:invalidate-demon 'inv-demon))

 (g-value A :top)  => 11
 (s-value A :left 1)    schema #k<A>, slot :TOP est invalidé
 (g-value A :top)  => 2
```

Relations. KR propose des champs spéciaux appelés *relations*. Ces relations gèrent l'héritage entre schémas, ainsi il y a un champ `is-a` dans l'objet enfant et champ `is-a-inv` dans l'objet parent. En plus des relations prédéfinies par le système, il est possible de créer de nouvelles relations grâce à la fonction `create-relation`; une relation peut déclarer avoir une relation inverse. Dans ce cas, KR génère les champs responsables de cette relation et son inverse dans les objets mis en relation et gère l'ajout ou le retrait des objets intervenant dans une relation. L'exemple illustre la création d'une relation `has-parts` ayant deux relations inverses `part-of` et `subsystem-of`, ici indique qu'il ne s'agit pas de relations d'héritage.

```
(create-relation :has-parts NIL :part-of :subsystem-of)
```

[4] Opal est responsable de la définition des objets graphiques.

[5] La fonction `create-schema` diffère de la fonction `create-instance` par le fait qu'elle n'initialise pas l'objet créé et ne copie pas les formules d'un prototype lors de l'héritage.

Le langage PowerClasses d'Ilog [Ilo 95], aménagement de la sur-couche objet de Lelisp, propose des possibilités plus sophistiquées de création et gestion automatique de champs (gestion de la cardinalité des relations, accesseurs). Une première version confidentielle de ce langage, nommée alors Icos, a été développée par une équipe du cnet à Sophia-Antipolis. Les fonctionnalités de Icos étaient, d'après les auteurs mêmes d'Icos, inspirées du langage Othelo [Forn 90a].

Discussion. Le modèle proposé dans KR est simple et semble adapté à la manipulation d'objet graphiques (widget...) pour la création d'interfaces. Cependant, il n'est pas sans poser des problèmes. Les relations proposées en KR sont associées aux variables d'instances exprimées et dispersées entre les valeurs de champs de différents objets.

- L'encapsulation est inexistante. Les auteurs de Garnet et KR mettent en avant que l'utilisation du langage encourage le programmeur à accéder et à modifier directement les champs d'un objet. Pour nous ce choix nous paraît préjudiciable: plus aucun masquage d'informations n'est possible. Les objets perdent une partie de leur potentialité et deviennent semblables à un amas de données.

- Les dépendances exprimées au travers des formules ne sont pas des dépendances entre objets mais des dépendances entre champs. La possibilité d'expression donnée par les formules est restreinte; il s'agit toujours de propager un changement de valeur, à aucun moment il est possible d'exprimer qu'un message puisse en impliquer un autre.

- La distribution des contraintes à l'intérieur des objets nécessitent la connaissance exacte de la représentation interne de ceux-ci. Les dépendances ne sont pas clairement exprimées: la sémantique d'une dépendance est distribuée dans les formules des champs de tous les objets y participant.

B.4 Le système PluS

Le système PluS [Boua 95, Boua 94] propose des contraintes réifiées réactives et unidirectionnelles en Smalltalk et un noyau minimal et extensible pour leur gestion. Cette nouvelle approche des contraintes répond en partie aux problèmes posés par les dépendances (voir A.3). La solution proposée s'intègre au modèle MVC de façon à permettre la fusion entre ces nouvelles dépendances et les dépendances traditionnelles de Smalltalk. Dans PluS, cette intégration est importante car ces contraintes sont destinées à la création d'interfaces dans lesquelles il doit être possible d'utiliser les composants visuels prédéfinis de Smalltalk.

La réification des contraintes en tant qu'objet de première classe permet d'avoir des dépendances dont la sémantique est explicite, ainsi que la possibilité d'exprimer des contraintes d'ordre supérieur (contraintes sur les contraintes). De plus, les contraintes sont exprimées en termes d'envoi de messages et non en termes d'accès aux variables d'instances. La solution proposée respecte donc l'encapsulation des objets. Contrairement à des approches comme celle de ThingLab, tout objet peut être contraint et toute expression de Smalltalk peut constituer le corps de la contrainte. Le programmeur n'est plus obligé de sous-classer ou de modifier les classes et les méthodes des objets mis en dépendance. Les contraintes peuvent être posées dynamiquement sur les objets et les instances non contraintes ne sont pas pénalisées. L'implémentation choisie est basée sur la modification dynamique des méthodes ainsi que le changement dynamique de classe des objets contraints (voir 9.1.1).

Des contraintes de PluS. L'exemple suivant permet de poser une contrainte d'alignement entre deux fenêtres de telle manière que l'une se positionne à droite de l'autre. Les fenêtres sont déplacées à l'écran en leur envoyant le message `displayBox: aRectangle`. La méthode `attachTo:using:spy:` déclare une contrainte entre le receveur (`self`) et `anObject`. Ainsi ligne 6 une contrainte est déclarée entre `win1` et `win2`: lorsque `win1` reçoit le message `displayBox:` la contrainte (lignes 7, 8, 9 et 10) est invoquée pour maintenir `win2` alignée.

Une contrainte symétrique est posée (lignes 12 à 17). Le système gère les cycles. Cependant, au vu de l'algorithme de gestion des cycles [Boua 95], les deux contraintes semblent être déclenchées alors qu'une seule devrait normalement suffir (l'algorithme de détection de cycles est un marquage des contraintes en cours de propagation, une contrainte n'est déclenchée que si elle ne l'est pas déjà).

```
| win1 win2 c1 c2 |
win1 := ScheduledWindow new label: 'Win1'.
win2 := ScheduledWindow new label: 'Win2'.
win1 open.
win2 open.
c1 := win1 attachTo: win2
        using: [:w1 :w2 | | r |
                r := w2 displayBox.
                      w2 displayBox: (r align: r origin with:
                                        (w1 displayBox) topRight)]
          spy: #displayBox:.
c2 := win2 attachTo: win1
        using: [:w1 :w2 | | r |
                r := w2 displayBox.
                w2 displayBox: (r align: r topRight with:
                                (w1 displayBox) origin)]
          spy: #displayBox:.
```

La réification des contraintes permet de définir des contraintes entre contraintes. Dans l'exemple précédent, lorsqu'une fenêtre est fermée par l'envoi de la méthode `primClose`, les contraintes doivent être supprimées. Pour automatiser cette suppression, il est possible de définir une contrainte entre une fenêtre et une contrainte[6].

```
win1 attachTo: c1
     using: [:w :c | c release]
     spy: #primClose
     evaluate: false.

win2 attachTo: c2
     using: [:w :c | c release]
     spy: #primClose
     evaluate: false.
```

Cependant cette formulation n'est pas suffisante, lorsque la fenêtre `win1` est fermée, la contrainte `c1` est bien supprimée. Mais que se passe-t-il alors si l'on bouge la fenêtre `win2` ? La contrainte `c2` n'est pas supprimée alors qu'elle n'a plus aucune raison d'exister et fait encore référence à la fenêtre `win1`.

Discussion. La solution proposée dans PLUS répond en partie aux problèmes posés par les dépendances. Quelques remarques sont nécessaires :

- Bien que PLUS propose une propagation de méthodes et non de valeurs, les contraintes donnent l'impression de n'être qu'une simple amélioration des formules du langage KR du point de vue de la gestion de l'encapsulation (voir B.3). Cette vision des dépendances se limite à une vision *«mono-méthode»*[7] des dépendances entre objets. Or, il n'est pas rare que plusieurs méthodes d'un objet soient contraintes en même temps pour maintenir un invariant (voir 5.1.2).
- De plus, alors que le noyau proposé est la bonne propriété d'être minimal, il serait souhaitable de proposer au programmeur un protocole étendu du comportement des contraintes. Par exemple, lors de la création de la contrainte, la possibilité de définir des actions spécifiques et différentes de la formule associée au sélecteur est nécessaire.
- Les contraintes proposées ici sont des fonctions ayant simplement comme arguments les deux objets contraints, ceci est limitatif: la possibilité d'utiliser des arguments d'appels de la méthode contrôlée dans la contrainte est importante. De même, il ne semble pas avoir été prévu la possibilité de faire référence dans la contrainte au résultat rendu par la méthode contrôlée. Pour finir, les contraintes étant réifiées, elles sont le lieu privilégié pour définir des informations propres à la dépendance. Or PLUS n'offre pas la possibilité de faire référence à la contrainte à l'intérieur de celle-ci. Ce faisant, il limite beaucoup l'utilisation même des dépendances en tant qu'objet.
- Des contraintes n-aires devraient être proposées.
- Les contraintes représentent une abstraction des interactions entre objets. Or dans PLUS, la réutilisation des ces abstractions n'est pas claire. Il semble que la seule possibilité pour réutiliser une même contrainte entre deux couples d'instances différentes soient de copier la formule. Cette solution est loin d'être élégante et s'intègre très mal dans un contexte de réutilisation des langages objets. Il est souhaitable de pouvoir définir des abstractions de dépendances et de les instancier entre différentes instances. De plus, dans PLUS aucun mécanisme de définition incrémentale (de la famille de l'héritage entre classes) n'est défini pour réutiliser les relations.

[6] Dans ce cas, la méthode de déclaration des contraintes est enrichie de manière à pouvoir spécifier que la contrainte ne doit pas être satisfaite lors de la création de celle-ci.

[7] Une dépendance consiste à énoncer **une** action associée à une méthode.

B.5 Rendezvous et ses liens

« The links are bundles of constraints that maintien consistency between the views and the abstraction. In particular, they ensure that the redundant information stored in the views and abstraction is kept consistent. » [Hill 92].

Le but du langage Rendezvous [Hill 93b, Hill 94] est de permettre la construction de systèmes interactifs pouvant être utilisés simultanément par de multiples utilisateurs depuis plusieurs machines distantes. Les auteurs de Rendezvous ont choisi un modèle architectural basé sur une séparation claire entre l'interface utilisateur et l'application : le modèle ALV, pour *Abstraction-Link-View* [Hill 92]. Ainsi le maintien de la cohérence lors de changements dans l'application ou dans les différentes vues est entièrement pris en charge par les *liens* (figure 11.7).

Rendezvous. Le langage Rendezvous est basé sur CLOS auquel la notion d'événements et des contraintes sont ajoutés. À chaque objet est associé une liste de gestionnaires d'événements *(event handlers)*. Ces gestionnaires permettent une correspondance entre les événements (click souris, entrée dans une zone graphique, ...) et l'appel des procédures associées. Le mécanisme de base de la gestion des événements est une évolution de Sassafras [Hill 86, Hill 87]. Dans Sassafras, des règles (action condition) spécifiaient les réponses aux événements et des variables booléennes servaient à définir les conditions des règles.

Les contraintes proposées [Hill 93a] sont unidirectionnelles, *(one-way constraints)*, d'une variable dite *source* vers une variable dite *cible*. Cependant, les variables peuvent être la cible de plusieurs contraintes. La propagation de valeur est locale et basée sur une planification des contraintes [Hill 94, Hill 93a].

Dépendances et liens. La définition de contraintes se fait grâce à la fonction `defDependency`. Les dépendances sont définies sur un objet, c'est-à-dire qu'elles contraignent les valeurs des variables d'instances de l'objet sur lequel elles sont définies. Voici par exemple la définition de la dépendance `widthTwiceHeight` qui assure que la largeur d'un objet est toujours le double de sa hauteur.

```
1  (defDependency widthTwiceHeight (Graphic)
2     (:source (deltaY self))
3     (setf (deltaX self) (*2.0 (deltaY self))))
```

La ligne 2 spécifie que la variable d'instance `deltaY` de l'objet `self` est la source de la contrainte. La ligne 3 définit que la valeur de la variable d'instance `deltaX` de ce même objet est le double de celle de la variable source `deltaY`. Pour que la mise en dépendance de ces deux variables soit complète, c'est-à-dire que la largeur de cet objet soit le double de sa hauteur, il faut définir la dépendance inverse.

Les liens de Rendezvous sont des collections de dépendances. La classe `Link` définit des contraintes simplifiant la création, l'installation et la destruction des liens. Les liens ont principalement deux rôles : le premier est d'assurer le maintien de la cohérence entre les objets de la partie applicative et ceux de l'interface. Le second est de maintenir une cohérence structurelle entre les deux mondes. De tels liens sont nommés des *liens de maintient d'arbres (tree maintenance links)*. Ils interviennent automatiquement lorsqu'une vue ou une abstraction est modifiée, détruite ou créée ; quand un nouvel objet est créé dans l'application, sa représentation doit être créée ainsi qu'un lien entre ces deux objets.

Bien qu'un lien soit un objet et que les liens puissent créer des objets donc des liens, Rendezvous n'offre pas directement la possibilité de définir des liens créant ou détruisant d'autres liens. Ceci est dû à la difficulté de changer dynamiquement le graphe de contraintes lorsque le maintien de celles-ci est actif. Les auteurs de Rendezvous propose donc de différer ces changements jusqu'à ce que le solveur de contraintes soit inactif.

Un lien du Tic-Tac-Toe ! Nous illustrons ici les dépendances et les liens au travers d'un exemple cher aux auteurs de Rendezvous : un jeu de Tic-Tac-Toe [Hill 92, Hill 94]. Le Tic-Tac-Toe est composé d'une partie applicative complètement autonome, de deux représentations personnalisées et de liens gérant la cohérence entre ces différentes parties (figure 11.9). Par exemple, l'élément graphique qui indique quel est le joueur courant (`current`) est géré par un lien. De même, lorsqu'un des deux joueurs décide d'effacer la partie en activant le bouton `reset`, un lien propage cette modification à la partie applicative qui le répercute à la vue de l'autre joueur. La figure illustre principalement deux types de liens : des liens entre des cellules et leurs représentations et des liens entre le tableau de bord du jeu et sa partie applicative.

La définition ci-après présente le lien assurant la cohérence entre une cellule et sa représentation. Ce lien calcule, au travers de la dépendance `TTTCellPositionLink` ligne 3, les coordonnées de la représentation en fonction de la colonne et de la rangée de la cellule. La définition de cette dépendance est donnée à partir de la ligne 7. La seconde dépendance (lignes 4 et 5) assure que la valeur de la variable d'instance `state` de l'objet pointé par `view` est égale à celle de l'objet pointé par `abstraction`, et inversement.

```
(defRVClass TTTCellLink (Link)
   (:add-dependencies
       TTTCellPositionLink
       (TTTCellStateLink = (state (view self))
                           (state (abstraction self)))))

(defDependency TTTCellPositionLink (TTTCellLink)
    (setf (xOffsetFromParent (view self))
          (+ 2.0 (* 20.0
                    (float (:source (column (abstration self)))))))
    (setf (yOffsetFromParent (view self))
          (+ 2.0 (* 20.0
                    (float (:source (row (abstraction self))))))))
```

La mise en oeuvre d'un tel lien se fait par l'instanciation de la classe `TTTCellLink`: supposons que nous disposions d'un objet cellule `AbstractCell-151` et de sa représentation `GraphicCell-24`, le code suivant illustre la création d'un lien.

```
(create 'TTTCellLink
        :abstraction AbstractCell-151
        :view GraphicCell-24)
```

Discussion. La solution proposée par les auteurs de RENDEZVOUS est très proche de notre propre solution. Les liens de RENDEZVOUS sont des entités réifiées, établies entre objets mais contrôlant l'accès aux champs de ces objets. Les dépendances possèdent une expressivité riche. La sémantique dynamique est de la propagation de valeurs avec possibilité d'envoi de messages. Un héritage entre relations est possible.

Plusieurs points ont retenu notre attention et méritent d'être soulignés.

- Le mécanisme de maintien de la cohérence est fortement lié au solveur de contrainte. Ainsi, le modèle ALV ne propose aucun moyen de contrôler la propagation des valeurs.
- Des aveux mêmes des auteurs de RENDEZVOUS: *« Linking things other than values »* [HILL 94], lier uniquement des variables d'instances est limitatif. Nous ne pouvons pas savoir comment les liens de RENDEZVOUS évolueront dans le futur mais il est certain qu'à chaque fois que cette évolution tendra vers une mise en dépendance d'objets et non plus de variables, le modèle ALV se rapprochera de notre propre solution.
- De plus, les liens proposés se limitent exclusivement à un comportement réactif et ne permettent d'exprimer des interactions riches entre objets.

B.6 ThingLab et Animus

« Starting with a combination of ideas from SKETCHPAD *and* SMALLTALK*, a number of features have been built up. Part-whole and inheritance hierarchies are used to describe the structure of a simulation. Contraints are employed as a way of describing the relations among the parts of a simulation. »* [BORN 79].

THINGLAB est un laboratoire de simulation d'expériences physiques ou géométriques écrit en SMALLTALK [BORN 79, BORN 86b]. Dans le but de décrire les relations entre les différents objets THINGLAB introduit la notion de contraintes dans un langage à objet. ANIMUS est une extension permettant de construire des simulations à base de contraintes dans lesquelles intervient le facteur temps. Nous présentons les différentes sortes de contraintes proposées par THINGLAB et ANIMUS: contraintes *statiques* entre variables d'instances et contraintes *temporelles*, en particulier les contraintes événementielles *(trigger constraints)* qui ont souvent été oubliées dans la littérature.

Un modèle objet augmenté. THINGLAB utilise le modèle à classes de SMALLTALK. Cependant, chaque classe possède une instance prototypique[8] qui peut être ainsi manipulée et affichée directement.

Contraintes statiques. *« A static constraint describes a relation that must hold at all times »* [BORN 86a].

THINGLAB permet de spécifier au niveau d'une classe des contraintes entre les différentes parties de l'objet représenté par la classe. Par opposition, à des contraintes entre différents objets non liés par une relation de composition comme par exemple dans RENDEZVOUS, ces contraintes maintiennent une cohérence *intra-objet*. Une contrainte est décrit par un prédicat et un ensemble de procédures qui peuvent être invoquées pour satisfaire la contrainte.

Voici un exemple: la classe `MidPointLine`. Cette classe représente un segment dont un troisième point est explicité afin d'être toujours au milieu des deux extrémités.

[8] [BORN 86a] discute plus en détail de l'utilité des prototypes par rapport aux classes dans le cadre de la construction d'interfaces utilisateurs.

```
1 Class MidPointLine
2   Superclasses
3     GeometricObject
4   Part Descriptions
5     line: aLine
6     midPoint: aPoint
7   Constraints
8     midPoint = (line point1 + line point2) / 2
9     midpoint ← (line point1 + line point2) / 2
10    line point1 ← midpoint * 2 - line point2
11    line point2 ← midpoint * 2 - line point1
```

Une instance de cette classe est composée d'une ligne et d'un point (lignes 5 et 6). La valeur du point est sujette à une contrainte définie par le prédicat (ligne 8) qui indique si les coordonnées de ce point sont en accord avec celles de la ligne. La définition même du prédicat et des procédures implique que les composants de l'objet sur lequel porte la contrainte sont directement accessibles au travers d'accesseurs. L'ordre d'énumération des procédures (lignes 9, 10 et 11) indique un ordre décroissant quant à la procédure à appliquer pour resatisfaire la contrainte.

Il est possible de préciser que certaines variables sont des constantes *(anchor)* ou, que pour une contrainte particulière, elles ne sont pas modifiables *(reference-only)*. Ces déclarations sont prises en compte lors de la phase de planification. De plus, ThingLab permet de fusionner les parties d'un objet. Ainsi, dans un quadrilatère composé de quatre segments, les sommets sont fusionnés afin de n'avoir que quatre points et non les huit des quatre segments pris séparément.

Maintien. La maintien des contraintes statiques se décompose en deux phases : la première consiste à planifier les modifications de valeurs, c'est-à-dire déterminer une suite d'actions à exécuter afin de resatisfaire le graphe de contraintes, et de stocker cette planification en compilant une méthode qui est invoquée pour resatisfaire les contraintes. Cette planification s'effectue lors du changement de valeur d'une des variables, contrairement à Equate [Wilk 91] dans lequel la réécriture des contraintes à lieu lors de leurs définitions. La seconde phase invoque la méthode compilée lors de la précédente phase sur l'objet dont la contrainte doit être satisfaite. La planification essaye de minimiser le code à exécuter, pour cela les auteurs de ThingLab utilise une technique de propagation d'états connus et de propagation de degrés de liberté. Lorsque la planification est impossible pour cause de cycles ThingLab utilise une méthode numérique itérative de relaxation [Born 86b]. Les contraintes statiques sont satisfaites de manière locale contrairement aux contraintes temporelles qui les sont de manière globale.

Contraintes temporelles. *« A response represents an abstraction of some behavior, and the actual realization of the response as a stream of events may depend on parameters only known at run-time »* [Born 86b].

Deux sortes de contraintes dites *temporelles* sont proposées dans ThingLab : celles qui décrivent une évolution continue des objets, les contraintes évoluant en fonction du temps *(time function contraints)* et les contraintes à comportement différentiel *(time differential contraints)*, et celles qui décrivent une évolution discrete des objets, les contraintes événementielles *(triggers constraints)*.

Les contraintes événementielles sont singulièrement différentes des contraintes statiques présentées plus avant. Dans Animus, ces contraintes sont utilisées pour spécifier comment les animations bougent ou changent en réponse aux événements reçus par les objets qu'elles représentent. De telles contraintes sont basées sur la définition de *réponses* à effectuer lors de la réception par l'objet contraint de certains messages.

L'exemple ci-après illustre ces contraintes : il spécifie l'animation d'une liste lors du tri de ses éléments. Le système fait clignoter les éléments d'un tableau lorsqu'ils sont sujets à une comparaison et montre leur permutation. Ainsi la définition de la contrainte ligne 7 exprime que lorsqu'une instance de `SortQueue` reçoit un message `compare:with:` la contrainte fera clignoter les éléments comparés de cette instance. De même, la définition ligne 8 que la réception d'un message `swap:with:` nécessitera la mise en oeuvre d'une animation montrant le déplacement des deux éléments.

```
1 Class SortQueue
2   Superclasses
3    AnimusObject
4   Part Descriptions
5    list: anOrderedCollection
6   Constraints
7    list triggersOn: #compare:with: causing: (Flasher at: a1 Flasher at: a2)
8   list triggersOn: #swap:with: causing: (Trajectory from: a1 to: a2 Trajectory from: a2 to: a1)
```

Prolongements. ThingLab a donné suite à des nombreux travaux et améliorations. Ainsi une nouvelle version de ThingLab, ThingLabII [Malo 89], est destinée à la construction interactive d'interfaces utilisateurs. ThingLabII diffère de son ancêtre par les points suivants :

- Les objets `Things` se composent comme en Smalltalk d'un état et de comportement. Cependant, à l'exception des primitives les méthodes sont remplacées par des contraintes. Un nouvel objet est créé en déclarant qu'un ensemble d'objets précédemment créés sont les composants du nouvel objet ; ces composants sont liés par des contraintes.

- Le système permet d'exprimer des hiérarchies de contraintes [Born 87, Born 92]. Le programmeur a alors la possibilité d'associer différents poids aux contraintes: `required`, `strong`, `medium` et `weak`. La satisfaction de contraintes relâche le cas échéant certaines contraintes en tenant compte de leurs poids. L'algorithme de planification de ThingLab DeltaBlue a été amélioré: SkyBlue [Sann 93] prend en compte les cycles et les contraintes à sorties-multiples *(multi-output methods)*.
- Un compilateur [Free 89] permet de transformer un prototype en un *module* dont les performances sont améliorées en temps et espace.
- Les variables conservant l'historique de leurs valeurs permettent d'induire la notion de temps dans les contraintes de la même manière qu'en Animus.

ThingLabII à lui-même donné suite à Kaleidoscope [Free 90b, Free 92] langage fusionnant la programmation impérative et la programmation par contraintes.

Discussion. Quelques remarques doivent être faites :

- Bien que certains comme Wilk [Wilk 91] considèrent que le choix de ThingLab de définir des contraintes à l'intérieur même des objets assure un respect de l'encapsulation, nous considérons que la définition automatique d'accesseurs pour toutes les variables d'instance d'un objet bien au contraire affaiblit l'encapsulation.
- Alors que les contraintes événementielles de Animus peuvent être vues comme une tentative de dépendances entre *objets* : l'objet et sa représentation. Les contraintes de ThingLab sont elles des contraintes entre *valeurs de variables d'instances*. A aucun moment, il n'est possible d'exprimer de relation entre des *comportements* des objets autres que des modification des champs de ceux-ci.
- Les contraintes de ThingLab sont fortement liées à la structure des objets ; ainsi le fait qu'un point soit le milieu de deux autres s'exprime au travers d'une classe définissant une nouvelle notion de segment et non d'une contrainte sur des points. Les contraintes sont associées à la classe de l'objet qu'elles contraignent, ce qui empêche la réutilisation au niveau des contraintes.
- Une fois les contraintes posées, le système va essayer de satisfaire le graphe de contraintes en relaxant certaines d'entre elles le cas échéant à l'aide d'un algorithme de planification. L'ensemble du graphe peut être considéré comme une *boîte noire* dont le programmeur perçoit seulement le comportement extérieur.

C

Interaction

Dans cette partie, nous abordons les travaux proposant une réelle prise en compte des dépendances, des interactions ou des relations entre objets. Il ne s'agit plus de contraintes comme au chapitre précédent mais d'interactions ou de dépendances entre objets. Dans ces travaux la prise en compte des interactions ou dépendances n'est pas uniforme : dans CONTRACT, la définition même des classes se fait par référence aux interactions. Dans TROLL, les dépendances ne représentent qu'un aspect de la modélisation d'une application.

C.1 Contract : un langage de spécification d'interactions entre objets

« *The existence of behavioral compositions in a system, and in particular the* behavioral dependencies *that they imply, cannot be easily inferred; they are spread accross many class definitions in method implementation.* CONTRACT *aims to formalize the collaboration and behavioral relationships between objects.* » [HELM 90].

Afin de permettre la *spécification explicite* de *compositions comportementales*[1], HELM, HOLLAND ET GANGOPADHYAY ont défini un langage de spécification des interactions entre objets : CONTRACT [HELM 90]. Ce langage a été utilisé, en particulier, pour décrire les différentes *compositions comportementales* présentes dans des bibliothèques de classes telles que Interview [LINT 89] qui est dédiée à la construction d'interface utilisateurs.

Initialement le principal objectif de CONTRACT était seulement de permettre une *spécification* de haut niveau d'abstraction des objets et de leur inter-communication, les détails des contrôles logiques étant alors mis de côté et laissé aux soins du programmeur des classes d'objets. I. HOLLAND dans [HOLL 92] a, quand à lui, enrichi ce langage de spécification afin de prendre en compte ces contrôles logiques et de pouvoir générer de manière automatique l'implémentation des objets décrits par les contrats. En particulier, I. HOLLAND a montré que CONTRACT permet de représenter des algorithmes classiques (parcours de graphes, ...) en tant qu'abstractions sous formes d'objets réutilisables. Nous nous limitons ici à une présentation du langage telle que défini originellement dans [HELM 90].

Définition. L'élément de base de CONTRACT est le *contrat*. La spécification d'une *composition comportementale* à l'aide d'un *contrat* nécessite la donnée des objets *participants* à la composition, les *obligations contractuelles* de chacun d'entre eux, les invariants et les méthodes qui instancient le contrat.

Les *obligations contractuelles* définissent des obligations de type de la part des participants : un participant doit avoir certaines variables d'instances et méthodes. Ensuite elles définissent des *obligations causales* à la réception de certains messages, un participant doit effectuer une séquence ordonnée d'actions, en particulier l'envoi de messages aux autres participants, et rendre par ce biais certaines conditions vraies. En second, un contrat définit les *invariants* et les actions permettant de le resatisfaire. Et finalement, il définit des *préconditions* sur les participants et les méthodes qui mettront en oeuvre le contrat.

Exemple. Nous illustrons les différents aspects d'un *contrat* en commentant la définition du *contrat* SubjectView tiré de [HELM 90] (voir figure C.1). SubjectView spécifie les différentes actions que les objets participants doivent satisfaire afin de maintenir la cohérence entre un objet, Subject, et les différentes vues, Views, qui le représentent.

Obligations de types : l'objet Subject doit posséder une variable d'instance value de type non spécifié Value et, par exemple, une méthode Notify() (lignes 2 et 5).

Obligations causales : la réception d'un certain message doit *mener* à un certain comportement. Ainsi l'expression Draw() $\mapsto$ Subject() $\rightarrow$ GetValue() ligne 11 spécifie que chaque vue doit à la réception du message Draw() avoir un comportement qui mènera à l'envoi du message GetValue() à l'objet Subject. Dans la même idée, la réception du message Notify() par l'objet Subject doit l'amener à envoyer le message Update() à chacunes des vues (ligne 5). Messages Update() qui à leurs tours devront amener à l'envoi d'un message Draw() pour chacunes des vues (ligne 10). Ainsi la condition {*View* **reflects** *Subject.value*} ligne 11) sera maintenue.

[1] Une composition comportementales est un groupe d'objets coopérants pour maintenir un invariant ou accomplir une tâche.

```
    contract SubjectView
1       Subject supports [
2           value : Value
3           SetValue(val:Value)↦ Δvalue{value = val};Notify()
4           GetValue() : Value ↦ return value
5           Notify() ↦ ⟨|| v : v ∈ Views : v→ Update()⟩
6           AttachView(v:View) ↦ {v ∈ Views}
7           DetachView(v:View) ↦ {v ∉ Views}
8           ]
9       Views : Set (View) where each View supports [
10          Update() ↦ Draw()
11          Draw() ↦ Subject() → GetValue() { View reflects Subject.value}
11          SetSubject(s:Subject) ↦ {Subject = s}
12          ]
13      invariant
14          Subject.SetValue(val) ↦ ⟨∀v : v∈ Views : v reflects Subject.value⟩
15      instantiation
16          ⟨ || v : v ∈ Views :
17          ⟨Subject → AttachView(v) || v → SetSubject(Subject)⟩⟩
    end contract
```

Figure C.1: Définition du contrat SubjectView

Un contrat ne contrôle pas l'envoi de message et n'implique pas non plus que les messages seront envoyés, il spécifie simplement l'interaction entre les objets. Le mot clé **invariant** permet la définition de l'invariant du contrat. L'expression Subject.SetValue(val) est l'action qui conduit à une satisfaction de l'invariant. Pour être complet un contrat doit spécifier, à l'aide du mot clé **instantiation**, des *préconditions* sur les différents participants ainsi que les actions l'instanciant. Ainsi un contrat SubjectView est opérationnel lorsque les méthodes AttachView et SetSububject sont exécutées avec des arguments appropriés (lignes 16 et 17).

Affinage et inclusion. *Contract* offre deux opérations sur les contrats : l'*inclusion* et l'*affinage (refinement)*. Ces deux opérations permettent l'expression de contrats complexes en termes de contracts plus simples. L'*inclusion* définit des contracts par la composition de sous-contrats. L'*affinage* permet une spécialisation des obligations contractuelles et des invariants. Ces deux mécanismes permettent ainsi de créer et de réutiliser des abstractions basées sur le comportement qui sont orthogonales à celles basées sur les classes.

Nous illustrons rapidement l'*affinage* d'un contrat. Considérons le contrat ButtonGroup qui spécifie comment un ensemble de radio-boutons peut être implémenté (figure C.2). Chaque bouton du groupe représente la valeur de la sélection courante qui est définie par l'objet State. Suivant si la valeur représentée par un bouton correspond à celle de l'objet State, le bouton sera *éteint* ou *allumé*. La relation entre les boutons et l'objet State est une relation SubjectView affinée pour les radio-boutons : à chaque instant un bouton et un seul est *allumé* et chaque bouton représente une valeur distincte.

Le contrat ButtonGroup affine donc le contrat SubjectView : le rôle Subject est rempli par l'objet *State*, et *Views* par des objets *Buttons* (ligne 2 de la figure C.2). De nouvelles obligations de type sont spécifiées, par exemple, l'objet qui participe à ce contrat en tant que bouton doit posséder une variable d'instance myvalue (ligne 5) et de nouvelles méthodes Select(), Choose(), Unchoose() et Refresh() (lignes 7, 10, 11 et 8).

Deuxièmement les *obligations causales* sont héritées ou enrichies. Comme les obligations de Subject ne sont pas redéfinies dans le contrat ButtonGroup, le participant State en hérite. Par contre, la spécification de Update(), plus spécifique du contrat ButtonGroup, prend le pas sur celle du contrat SubjectView. Ainsi un appel à Update() doit conduire à un appel à Draw() comme spécifié dans le contrat SubjectView, cependant d'autres méthodes spécifiques à la sémantique des radio-boutons doivent être appelées entre celles-ci, comme Choose(), Unchoose() ou Refresh() (lignes 9, 10 et 11 figure C.2).

Troisièmement, la clause **instantiation** du contrat est étendue pour prendre en compte que chaque bouton représente une valeur unique et qu'un seul bouton n'est actif à la fois (lignes 17 et 18 figure C.2). Et finalement, l'invariant de ButtonGroup implique celui spécifié dans SubjectView en y incluant des conditions supplémentaires.

Conformité et implémentation. « *Implementation* (of contracts) *is then mapped to the contract specification with* conformance declarations. *A conformance declaration states how a class implementation conforms to a contract participant specification.* » [Holl 92].

```
contract ButtonGroup
    refines
        SubjectView(Views = Buttons, Subject = State)
    State supports []
    Buttons : Setof(Button) where each View supports [
        myvalue : Value
        chosen : Boolean
        Select() ↦State→SetValue(myvalue)
        Refresh() ↦ Draw()
        Update()↦if State → GetValue()= myvalue then Choose() else Unchoose()
    Choose() ↦ Δchosen {chosen = true};Refresh()
    Unchoose() ↦ Δchosen {chosen = false};Refresh()
    ]
    invariant
    Button.Select() ↦ ⟨∀b : b ∈ Buttons : b.chosen ⇔ b.myvalue = State.value⟩
                    ∧⟨∃!b : b ∈ Buttons : b.chosen⟩
    instantiation
        {∀b1, b2 : b1, b2 ∈ Buttons: b1 ≠ b2 ⇒ b1.myvalue = State.value}
        ⟨|| b : b ∈ Buttons : State → AttachView(b) || b → SetSubject(State) ⟩
end contract
```

Figure C.2: Définition du contrat ButtonGroup

Les contrats sont spécifiés indépendemment des classes. Cependant, lors de l'implémentation des classes des objets participants à ces contrats, celles-ci doivent se soumettre aux exigences dictées par les contrats. Ces *déclarations de conformité* d'une classe par rapport à un contrat spécifient comment cette classe remplit les *obligations* du participant spécifié par le contrat. Helm, Holland et Gangopadhyay donnent des exemples complexes afin d'illustrer ces déclarations de conformités dans [Helm 90]. Intuitivement, une classe est conforme à la définition d'un participant si et seulement si ses méthodes et variables d'instances satisfont à la fois les obligations de type et les obligations causales spécifiées dans la définition du participant.

Remarques Les contributions de *Contract* sont : premièrement de capturer des *dépendances comportementales* entre objets et d'offrir un formalisme pour leur abstraction.

Contract offre un langage très riche pour la spécification des dépendances entre objets. Cependant, il existe une différence entre l'approche que nous proposons à celle de Contract. Les contrats spécifient des règles d'échange de messages entre les participants. Ces participants se conforment à ces règles de part l'implémentation de leurs méthodes, le contract spécifie l'échange de messages entre objets et n'a aucun contrôle.

Le comportement des objets participant est spécifié par ses obligations. Cependant, comme un objet peut participer à plusieurs contrats, son comportement total peut être complexe. En fait, chaque contrat factorise ce comportement en parties séparées qui peuvent être comprises et modifiées séparément. Cette approche est similaire à l'approche de Ossher et Harisson [Ossh 92, Harr 93] qui séparent l'interface totale d'un objet en *Views*. D'autres approches factorisent l'interface d'un objet en parties nommées *Roles* [Arap 90, Pern 90], où un rôle représente une étape dans le cycle de vie d'un objet.

Nous percevons les contrats comme une forme de *«compilation»* de la notion de dépendance au sein des méthodes des objets participants; c'est-à-dire la spécification (le contract) explicite les différentes dépendances entre objets et la conformance des classes assure que celles-ci ont bien le comportement requis. Cette vision des contrats vient principalement du fait de la profonde différence entre le statut des classes de participants dans Contract et celle que nous avons choisi tout comme Notkin dans [Sull 92]. En effet, Contract accentue l'importance des dépendances entre objets par rapport aux classes des participants qu'elles relient. Ainsi Helm, Holland et Gangopadhyay déclare que la spécification d'une classe est dispersée dans les contrats et les déclarations de mise en conformité, et donc n'est pas localisée dans une seule définition de classe : «With Interaction-Oriented design, the specification of a class becomes spread over a number of contracts and conformance declaration, and is not localized to one class definition.» [Helm 90].

C.2 ACT

Dans [Aksi 94], les auteurs proposent le concept de Type Abstrait de Communication: ACT *(Abstract Communication Types)* afin de modéliser les interactions entre objets. Le modèle proposé est construit à partir du modèle de

Aksit Composition Filters [Aksi 88, Aksi 92b]. Nous ne présentons ici que le strict nécessaire[2] à la compréhension des ACTs.

Composition Filters. Dans le modèle Composition Filters, le modèle objet est étendu par l'introduction de filtres composables d'entrée et de sortie qui affectent les messages reçus et envoyés. Le langage sinaST met en œuvre ce modèle qui est résumé par la figure C.3. Un objet est décomposé en deux parties : une interface et une implémentation. L'interface s'occupe des messages reçus et émis. Pour cela, elle définit des filtres d'entrée et de sortie, des objets internes ou externes et les signatures des méthodes. Les filtres sont contrôlés par des *conditions* ou *états*. La figure C.3 dépeint le possible cheminement d'un message reçu : si le message passe tous les filtres d'entrée, il est soit délégué aux objets internes, aux méthodes ou aux objets externes. De plus, tous les messages dus aux exécutions des méthodes de l'objet et envoyés à des objets extérieurs doivent franchir les filtres de sorties.

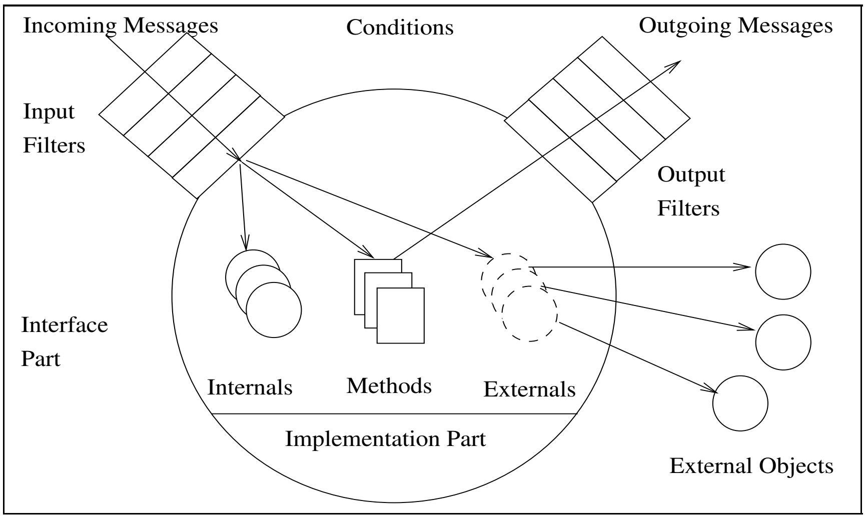


Figure C.3: Composants de l'interface et cheminement des messages dans le modèle objet Composition Filter

ACT. Un ACT est une classe sinaST ayant la même syntaxe et sémantique. C'est la façon dont son comportement est composé avec celui des objets participant à l'interaction qui la différencie des autres classes. Une classe ACT opère sur des messages réifiés. Un nouveau filtre nommé *Meta* permet la réification des messages. Ainsi lorsqu'un message est accepté, il est tout d'abord réifié (converti comme instance de la classe Message) et passé comme argument à d'un nouveau message envoyé à l'ACT. L'ACT a alors à sa disposition toutes les informations concernant le message (émetteur, récepteur, arguments...) et peut le modifier en invoquant les méthodes de la classe Message. Finalement, l'ACT reconvertit cette instance en l'exécution du message. Les ACT se classent en différentes familles suivant s'ils sont spécialisés pour la réception ART *(Abstract Receiver Type)* ou l'émission des messages AST *(Abstract Sender Type)*. L'utilisation d'un AST permet de mettre en place des communications asynchrones, du codage de messages. Un AST peut enregistrer les objets receveurs de messages et ainsi réaliser de la diffusion de messages *(multicast)*. L'utilisation d'un ART permet la définition de protocoles de sécurité, de solveurs de contraintes, de décodage de messages.

Illustrations. Parmi les différents exemples proposés dans [Aksi 94], contraintes unidirectionnelles, synchronisation, transactions atomiques, nous avons choisi de présenter ici le premier exemple car il nous permet de comparer notre approche a celle de ACT. Notre modèle objet n'intégrant pas de notion de concurrence entre les messages les autres exemples ne pourraient être exprimés dans notre langage.

Une instance de la classe `ReferencePoint` est supposée servir de référence à d'autres points qui constituent une figure. Le comportement souhaité est qu'à chaque changement de coordonnées du point de référence, tous les points dépendants soient mis à jour.

Pour comprendre la définition ci-après, il faut savoir que l'héritage proposé dans sinaST n'est pas un héritage traditionnel entre classes comme en Smalltalk mais d'un héritage simulé nommé héritage à base de délégation *(delegation-based inheritance)*. Une instance de la classe `ReferencePoint` est en fait constituée par une instance de la classe `Point` (`myPoint` ligne 7) à laquelle elle délègue les messages auxquels elle ne sait pas répondre (voir figure C.4).

[2]Le modèle Composition Filters a fait l'objet d'un certain nombre de publications dans lesquelles le lecteur pourra trouver toutes les informations [Aksi 88, Aksi 92b, Aksi 94].

```
1  class ReferencePoint  interface
2   comment  this class is a subclass of class Point and it used as a
3      reference point for a set of other points;
4   externals
5    figure: OneWayConstraints;  // instance of the ART class
6   internals
7    myPoint: Point;  // instance of the superclass
8   methods
9   display returns Nil;  // display itself on the current point
10  inputfilters
11   {
12     constraints: Meta =  True => [*.moveTo]figure.applyConstraint;
13     disp: Dispatch = True => myPoint.*, True => inner.*;
14   }
15 end;
```

Les filtres d'entrée doivent être lus séquentiellement. Un filtre est constitué d'un ou plusieurs *éléments de filtres* (1 pour *constraint* et 2 pour *disp*). Ces éléments sont composés de trois parties: une partie condition (avant le symbole `=>`, une partie reconnaissance de messages `[...]` et une partie substitution.

Le premier filtre *constraint* (ligne 12) de la classe `Meta` contient un unique élément de filtre. La condition `True` de cet élément signifie que seuls les messages satisfaisant le schéma de reconnaissance `[*.moveTo]` sont acceptés par ce filtre. Tous les messages ayant le sélecteur `moveTo` sont réifiés. Les autres messages sont passés aux filtres suivants. La partie substitution `figure.applyConstraint` spécifie qu'après la réification du message reconnu, celui est passé en argument à la méthode `applyConstraint` de l'objet `figure`.

L'objet `figure`, instance de la classe `OneWayConstraint`, est responsable du maintien de la cohérence de ses dépendants. Aussi après avoir adaptés ses dépendants, `figure` reconvertit le message réifié en un message qui est alors traité par le second filtre.

Le filtre *disp* (ligne 13) est responsable de l'appel des méthodes de l'objet. La première partie, `True => myPoint.*`, spécifie que tous les messages entrant sont délégués à l'objet *interne* `myPoint`, la seconde partie `True => inner.*` spécifie que les messages ne satisfaisant pas le premier filtre au lieu d'être délégués sont envoyés à la pseudo-variable *inner*, ceci permet un accès direct aux méthodes de l'objet.

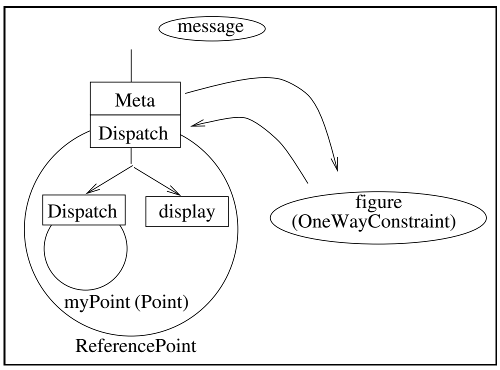


Figure C.4: Un filtre Meta et des contraintes unidirectionnelles

La classe `OneWayConstraint` dont l'interface est donnée ci-après, est un ACT. Elle assure la consistance entre objets dépendants. Cette classe définit cinq méthodes. Deux retiennent notre attention: `applyConstraint` permet de lancer la phase de maintien de cohérence et `putConstraints` de définir les contraintes. Celles-ci sont définies par des *blocks* (méthodes de SINASТ). Chaque block est une contrainte devant être satisfaite et correspond à l'objet qui est stocké avec le même indice dans la liste des objets dépendants. Par exemple, les contraintes des éléments de la figure peuvent exprimées par l'expression suivante: `[moveTo (message.argument(1) + dX, message.argument(2) + dY)]` dans laquelle `message` est l'argument de la méthode `applyConstraint` et `argument(i)` fait référence au i ème élément du message.

Comme toutes les classes, la classe `OneWayConstraint` peut être spécialisée afin de raffiner le comportement du maintien de la propagation.

```
class OneWayConstraint  interface
 comment  this class implements a one way constraint mechanism;
 methods
  applyConstraint(Message)  returns Nil; //  this is the independent reference message
  putDependants( OrderedCollection( Any ))  returns Nil;
  size  returns Integer;  //  number of dependant objects
  putConstraints (OrderedCollection(Block))  returns Nil; //  store constraints for dependants
  getConstraints  returns OrderedCollection(Block); // retrieve constraints
 inputfilters
  disp: Dispatch = True => inner.*;
end;
```

Discussion. La solution repose principalement le contrôle explicite de l'envoi de message. Bien que l'introduction, sûrement par un souci d'homogénéité, d'un filtre `Meta` semble un peu forcée, la solution proposée confirme que le contrôle de l'envoi de messages est le mécanisme nécessaire à la mise en œuvre de dépendances entre objets.

Cependant, la solution proposée est un peu rudimentaire, la réification des messages n'est qu'un moyen pour la mise en œuvre de dépendances. Aussi alors même que notre solution est aussi basée sur un contrôle, implicite cette fois, des envois de messages, nous pensons que ce contrôle reste un mécanisme important mais avec lequel le programmeur n'a pas à forcément être confronté.

L'exemple précédent est révélateur de différents points :

- Les ACTs et la (ou les) classes d'objets participant à l'interaction sont liées par la nécessité de spécifier au niveau des classes à quels ACTs les messages réifiés sont envoyés.
- Le programmeur doit explicitement spécifié quels messages doivent être spécifiés et à quelles méthodes des ACTs ils sont associées.

Bien que la dépendance soit traitée comme les autres entités du langage, sa mise en œuvre, en particulier le contrôle de l'envoi de message, nécessite un éclatement de celle-ci au travers des différentes classes y participant.

C.3 DynaSpecs et Interactor

Les auteurs de [Doda 95] proposent une approche unifiée pour la prise en compte des interactions comme des objets de plein droit : les *interacteurs*. Ils ont les caractéristiques suivantes :

- Les interacteurs sont instanciés lors de l'exécution du programme avec les objets participant à l'interaction. Une fois instanciés, ils sont traités comme n'importe quel autre objet. Par conséquent, des interacteurs peuvent être sujet d'autres interacteurs.
- La présence d'un interacteur est transparente pour un objet. Contrairement, à des approches telles que les Gluons [Pint 93] ou le concept de médiateurs [Gamm 94], un objet ne fait pas référence à un interacteur : il agit comme à son habitude.
- Les objets participants à une interaction offre exclusivement le comportement intrinsèque de l'objet. L'interacteur définit lui les comportements spécifiques à l'interaction et contrôle comment le comportement de base d'un objet est utilisé dans le contexte de l'interaction.

Spécification des Interactors : Dynaspecs. Dodani et al. prennent en compte l'expression explicite des interactions durant toutes les phases du développement. Ainsi ils proposent la spécification des interactions par la définition d'un *DynaSpec*. Une spécification DynaSpec possède deux parties : l'une définissant les attributs et méthodes que doivent posséder les objets coopérants et l'autre le comportement de l'interaction défini par une machine d'états finis (comme dans [Yell 94]). Les états de la machine représentent les changements d'états des objets par rapport à l'interaction. Une transition d'un état à un autre de la machine d'états finis correspond à la réception d'un message. Il est possible d'associer des pré- et post-conditions aux transitions. Une règle de transition est définie de la manière suivante : Etati pré-condition message post-condition → Etatj.

La règle 1 est l'état par défaut. Les préconditions spécifient les conditions devant être remplies avant la transition. Les post-conditions précisent l'état des attributs après la transition.

Dans l'exemple, d'un interacteur gérant la cohérence entre un modèle et une vue. Le modèle doit posséder un attribut `information` et une méthode `getValue`. La vue doit posséder une méthode `update` et l'interacteur une méthode `makeConsistent`.

Voici les règles de transitions d'un interacteur de consistance entre un modèle et sa représentation.

```
1. information = NIL → Ready
2. Ready  makeConsistent/getValue(aValue)information = aValue → Inconsistent
3. Inconsistent  getValue(aValue)/update(information) → Inconsistent
4. Inconsistent information ≠ NIL update(information)  → Ready
```

Par défaut, l'interacteur est cohérent. Lorsque le modèle change de valeur, l'interacteur devient inconsistent (2). La règle 4 précise que si la vue ne reflète pas l'information détenue par le modèle (état inconsistent) alors le message `udpate` reçu par la vue prend en charge la cohérence.

Implémentation des Interactors. Dodani et al. proposent un constructeur de composition `COMPOSE` permettant de définir des interactions. Il permet de définir les variables partagées par les différents participants de l'interaction, les interfaces des participants, les procédures et initialisation de l'interaction.

```
COMPOSE  ident ( Déclarations de variables)
[PARTS] CLASS  ident_participant1 ...
...
[PARTS] CLASS  ident_participantn ...
 Procédures
BEGIN  exécution END
```

Pour chaque participant défini grâce au mot clé CLASS (PARTS indique que plusieurs participants du même type sont définis) les méthodes éventuellement abstraites[3] qu'un participant doit satisfaire sont énumérées. Les procédures de l'interaction sont visibles par tous les participants.

La procédure `makeConsistent` est la méthode que le système invoque lorsqu'il maintient la consistance entre les différents participants.

```
COMPOSE CalculatorConsistency ()   ;; définition des variables partagées
  CLASS ModelParticipant ()
      ABSTRACT getValue():INTEGER;
  END;

  PARTS CLASS DependentParticipants ()
      ABSTRACT setValue (newValue: ModelParticipant);
      ABSTRACT update();
      PROCEDURE setDependentValue (newValue: ModelParticipant) =
        BEGIN SELF.setValue(newValue.getValue()) END;
  END;

  PROCEDURE makeConsistent (changeContext: INTEGER):INTEGER =
  VAR dependentValue: ModelParticipant;
      dependents: LIST DependentParticipants;
      dependent: DependentParticipants;
  BEGIN
      dependents:= CSELF.DependentParticipants ();
      WHILE LENGTH (dependents) > 0 DO
        dependentValue:= CSELF.ModelParticipant_getValue();
        dependent:= HEAD(dependents);
        dependent.setDependentValue(dependentValue);
        dependent.update();
        dependents:=TAIL(dependents);
      END;
  END;
BEGIN
END;

 ;; instanciation de l'interacteur
digitView:= NEW(DigitView);
calculator:= NEW(Calculator);
calulatorConsistency:= RELATE CalculatorConsistency(calculator, [digitView]);
```

Chaque participant peut se référencer en utilisant la variable `SELF`. `CSELF` est une pseudo-variable faisant référence à l'interacteur. La méthode `setDependentValue` copie apparemment la valeur du modèle dans un dépendant. La méthode `makeConsistent` spécifie que pour chaque dépendant, la valeur du modèle est récupérée[4] et la méthode `update` est invoquée.

Discussion. Nous aimerions avoir eu plus d'informations tant pour la compréhension globale des interacteurs qu'au niveau de l'implémentation. En particulier, les interacteurs sont décrits comme étant transparents du point de vue de l'envoi de message et nous aurions souhaité savoir si la technique utilisée était basée sur un contrôle de l'envoi de messages. Cependant, les auteurs n'ont pas donné suite à nos demandes.

[3]Les méthodes qualifiées d'abstraites sont remplacées par les méthodes des objets effectivement liés lors de l'exécution)

[4]Ce qui semble inutile.

C.4 Troll

« Relationships connect objects that are specified independently. Basically, relationships are language constructs to describe how system components are connected in order to describe the whole system. » [Jung 96].

Troll [Jung 93] est un langage de spécification intégrant les interactions entre objets. Troll concilie l'approche des langages de programmation orientée objets qui mettent en avant une manipulation fonctionnelle des objets au travers de leur interface (méthodes), à celle des bases de données orientées objets qui elles mettent en avant une vision structurelle des objets (au travers des attributs). Troll offre une vision structurelle des objets par le biais de leurs attributs et une manipulation du comportement des objects à l'aide d'*événements* constituant des abstractions des opérations de changements d'états ou des méthodes.

Nous montrons rapidement les spécificités de ce langage au travers d'un exemple de déclaration de classe, afin de pouvoir expliquer les relations proposées. Nous renvoyons le lecteur désirant avoir plus de détail à [Jung 93, Jung 96]

C.4.1 Un modèle objet étendu

En Troll, une classe est définie par la donnée d'un *espace d'identification* et d'une description générique d'instances. Cette dernière nécessite la définition d'une *déclaration de signature* et d'une *spécification*. La déclaration de signature consiste à introduire les symboles représentant les attributs et les événements (lignes 4 à 10). Ces symboles constituent l'interface de la classe : on peut lire les valeurs des attributs ou invoquer les événements de l'extérieur de l'instance. Les attributs peuvent être qualifiés de *dérivés*, lorsqu'ils sont le fruit du calcul d'autres attributs. La spécification est composée de sections pour la *dérivation* (ligne 12), la définition de *contraintes* (ligne 11), le calcul de valeur des attributs (ligne 13 et 14) et le comportement (lignes 15 à 30). Pour le comportement plusieurs formalismes sont proposés : des permissions définissant des conditions d'acceptations de événements, des obligations, des réactions *(commitments)* et des schémas *(patterns)*. Troll est très riche, aussi n'entrons nous pas dans les détails.

La classe ATM suivante répresente un distributeur automatique de billet.

```
1   class ATM
2   identification    No: nat;
3   template
4    attributes  CashOnHand: money;
5                derived Dispensed: bool;
6     events  birth set_up; death remove; ready; cancel;
7             read_card(in C:|CashCard|);
8             check_car_w_bank(in Acct:nat:, in PIN:nat); card_accepted;
9             bad_Pin_msg; bad_account_msg; issue_TA(in Acct:nat, in Amount:money);
10            TA_failed_msg; eject_car; dispense_cash(in Amount:money);
11    constraints initially CashOnHand = 10000;
12    derivation Dispensed = (CashOnHand < 100);
13    valuation variables m: money;
14     [dispense-cash(m)]CashOnHand = CashOnHand - m;
15    behavior
16      permissions variables n: nat; m: money; C: |CashOnHand|;
17         not Dispensed read_card(C);
18         m <= CashOnHand issue_TA(n,m);
19       patterns variables n,p: nat; m: money;
20         process CARD = read_card -> (exists n,p: nat)(check_card_w_bank(n,p))
21         end process;
22         process TA(n) = (exists m: money)(issue_transaction(n,m) ->
23                         case dispense_cash(m);
24                              TA_failed_msg esac)
25         end process;
26      CARD -> case
27              bad_account_msg -> eject ;
28              bad_PIN_msg -> eject ;
29              card_accepted -> TA -> eject;
30              esac
31    end class ATM;
```

Un fragment de définition d'un objet représentant la banque est :

```
object Bank
  template
    events
     birth establish; death close_down;
     ....
     verify_card(in Acct:nat, in PIN:nat, in ATM:|ATM|);
```

```
      no_such_account(out ATM:|ATM|);
      bad_PIN(out ATM:|ATM|);
      card_OK(out ATM:|ATM|);
      process_withdrawal(in Acct:nat, in Amount:money, in ATM:|ATM|);
      TA_failed(out ATM:|ATM|); TA-OK(out ATM:|ATM|);
      ....
end object Bank
```

C.4.2 Des relations en Troll

Les relations décrites en Troll lient des objets définis séparément. Troll permet de spécifier des contraintes et des communications globales entre objets.

Contraintes globales. Elles permettent la spécification des contraintes entre objets définis localement. Ainsi l'exemple suivant définit une contrainte entre deux instances de la classe `CheckingAccount`. Cette contrainte n'aurait pas pu être spécifiée de manière locale à une instance.

```
relationship CheckingHolder  between CheckingAccount C1, CheckingAccount C2;
  data types |BankCustomer|, nat;
  constraints (C1.Holder = C2.Holder) -> (C1.No = C2.No);
end relationship CheckingHolder;
```

Interactions globales. Elles définissent la communication entre objets. Du point de vue d'un processus, une relation décrit comment les processus impliqués dans la relation se synchronisent.

L'exemple qui suit tiré de [Jung 93] illustre la communication globale entre objets. Cet exemple définit comment un distributeur de billet *Automatic Teller Machine* et une banque communiquent. La relation suivante établit des relations d'appels *(calling relationships)* entre une instance de la classe `Bank` et une instance de la classe `ATM`. La relation d'appel `e1 » e2` entre deux événements e1 et e2 établit que lorsque l'événement e1 a lieu, l'événement `e2` doit aussi avoir lieu. Cependant, `e2` peut avoir lieu sans que `e1` ait lieu.

```
relationship RemoteTransaction Between Bank, ATM
 interaction
    variables atm:|ATM|; n,p:nat; m:money;
    /* vérification de la carte */
    ATM(atm).check_card_w_bank(n,p) >> Bank.verify_card(n,p,atm);
    Bank.no_such_account(atm) >> ATM(atm).bad_account_msg;
    Bank.bad_PIN(atm) >> ATM(atm).bad_PIN-msg;
    Bank.card_ok(atm) >> ATM(atm).card_accepted;
    /* transaction */
    ATM(atm).issue_TA(n,m) >> Bank.process_withdrawal(n,m,atm);
    Bank.TA_failed(atm) >> ATM(atm).TA_failed_msg;
    Bank.TA_OK(atm,m) >> ATM(atm).dispense_cash(m);
end relationships RemoteTransaction;
```

Dans cet exemple, la communication est décrite point à point. L'expression `ATM(atm).` fait référence à une instance de la classe `ATM`. Les variables utilisées dans les relations d'appels ainsi les variables `n` et `p` de l'expression `Bank.verify_card(n,p,atm)` sont liées aux valeurs des variables de l'expression `ATM(atm).check_card_w_bank(n,p)`.

Les relations sont aussi spécifiées à l'aide de *préconditions temporelles* pour spécifier l'ordre *(precedence relationships)* des relations. Ces préconditions peuvent faire référence à l'histoire de la communication et de gardes *(guards)*. Avec une garde, une communication peut seulement avoir lieu lorsque la précondition est vraie. En voici une illustration: la relation entre un distributeur automatique de billets et un usager.

```
1 relationship UseATM between ATMCustomer, ATM
2  interaction
3    variables C:|ATMCustomer|; atm:|ATM|; CC:|CashCard|; p:nat; m:money;
4    Customer(C).insert_card(CC,atm) >> ATM(atm).read(CC);
5     after(Customer(C).insert_card(CC,atm)) ==>
6      Customer(C).enter_PIN(p,atm) >>
7       ATM(atm).check_car_w_bank(CashCard(CC).ForAcct, p);
8    Customer(C).enter_Cancel(atm) >> ATM(atm).cancel;
9     after(Customer(C).enter_PIN(p,atm)) ==>
10     ATM(atm).card_OK >> Customer(C).card_accepter(atm);
    ...
end relationship;
```

Ainsi les lignes 5, 6 et 7 spécifient que la communication n'aura lieu que si la garde est vraie, c'est-à-dire si l'expression temporelle after(Customer(C).insert_card(CC,atm)) est vraie.

Implémentation. Les relations de TROLL ne sont pas réifiées, elles sont transformées dans des structures plus primitives nommées *canal*, *(canal object)* partagées par les objets communiquant. Chaque clause de communication est transformée en un événement de l'objet canal. Ainsi pour mettre en œuvre la relation d'appel `e1 » e2`, l'événement e1 d'un canal appel l'événement e2.

D

Coordination

La coordination des activités des objets est un aspect fondamental des langages de programmation concurrents et distribués. Dans de nombreux langages, cette coordination est exprimée explicitement dans les termes de protocoles de communication de bas niveau. L'expression de la coordination devient alors étroitement liée à l'implémentation des objets, ce qui gène la modification et réutilisation des schémas de coordination [Aksi 92a]. Différents travaux ont proposé des solutions à ce problème [Guer 92] souvent par l'introduction de contraintes de synchronisation [Nier 87, Neus 91, Frøl 92] exprimées à l'aide de conditions booléennes. Plus récemment, Agha et Frølund ont proposé l'introduction de *synchroniseurs* : entités responsables de la coordination de plusieurs objets.

Bien que les problèmes des langages à objets concurrents soient quelque peu différents des nôtres, l'idée de proposer une abstraction de la coordination entre objets tend vers des objectifs presque similaires aux nôtres : abstraction de l'interaction entre les objets et meilleure réutilisation des objets et de l'interaction.

D.1 Les *synchronizers*

« The behaviors of objects depend on the context in which they exist. Synchonizers allows us to express the contextual constraints of a part-whole hierarchy as an aspect of the hierarchy, not an aspect of the parts » [Frøl 93].

Agha et Frølund ont développé un langage pour l'expression de la coordination d'objets [Frøl 93, Frøl 94] basé sur l'abstraction de schémas de coordination. L'élément central de ce langage est la notion de synchroniseurs *(synchronizers)* : une contrainte exprimée entre plusieurs objets et spécifiée de manière indépendante de celle des protocoles requis pour implanter les propriétés souhaitées.

La spécification de ces synchronizers en termes exclusivement des interfaces des objets contraints permet d'une part une description, un raisonnement et des modifications de ces contraintes plus aisées et d'autre part, une séparation entre la coordination et les fonctionnalités des objets qui peuvent être alors plus facilement réutilisés.

Les synchronizers. Conceptuellement, un synchronizer est une entité spéciale qui observe et limite les invocations de l'ensemble des objets qu'il contraint. Plusieurs synchronizers peuvent contraindre un même objet. Les synchronizers peuvent être mis en place par n'importe quel objet, ce qui permet à un objet d'en contraindre d'autres.

La structure d'un synchronizer est définie à l'aide du constructeur `{ ... }`. Comme le montre la dernière règle de la figure D.1 qui présente la syntaxe abstraite des synchronizers, il est possible de nommer un synchronizer afin de le réutiliser. Mais regardons plutôt un exemple !

Exemple : un distributeur automatique. Il s'agit là d'un exemple de hiérarchie de composition *(part-whole hierarchy)* avec contraintes tiré de [Frøl 93].

Un distributeur automatique est décomposé en plusieurs parties : une partie, que nous nommeront *monnayeur*, accepte de l'argent et des compartiments contenant, par exemple, des fruits. Ces différents composants doivent être contraints afin d'offrir le comportement attendu : qu'il faut mettre assez d'argent pour pouvoir avoir un fruit, qu'une fois le fruit pris l'argent inséré n'est plus en compte pour un choix ultérieur et que si l'on appuie sur un bouton spécial du monnayeur l'argent inséré est rendu. Si l'on modélise chacune des parties de cet appareil par un objet distinct la contrainte de fonctionnement est énoncée par un synchronizer.

Le code suivant défini un possible synchronizer pour un distributeur ayant seulement deux compartiments[1] : un pour les pommes et un pour les bananes. Supposons que l'objet monnayeur définisse deux méthodes : `insert()` et `refund` et que les compartiments eux définissent une méthode `open`.

[1] L'exemple donné ici se veut avant tout simple, d'autres choix pourraient être proposés mais ce n'est pas le propos.

```
1  VendingMachine(accepter,apples,bananas,apple_price,banana_price)
2  {   init amount := 0
3      amount < apple_price   disables apples.open,
4      amount < banana_price  disables bananas.open,
5      accepter.insert(v)  updates amount := amount + v,
6      (accepter.refund  or apples.open  or bananas.open)
7       upates amount := 0 }
```

Le synchronizer nommé `VendingMachine` permet de contraindre le monnayeur (l'objet `accepter`) et les compartiments (les objets `banane` et `apple`). Des variables supplémentaires sont passées comme arguments du synchronizer: le prix associé à chaque compartiment (ligne 1).

Le synchronizer `VendingMachine` définit une variable locale `amount` qui représente l'argent que l'utilisateur met dans le monnayeur (ligne 2). Les lignes 3 et 4 spécifient que la méthode `open` d'un compartiment ne peut être activée que lorsque l'argent inséré dans la machine est supérieur ou égale au prix associé au compartiment. La ligne 5 spécifie que la valeur de la variable `amount` reflète constamment la somme insérée dans le monnayeur. Les lignes 6 et 7 sont complémentaires par rapport à la ligne 5, elles assurent que la valeur de la variable `amount` reflète bien la somme d'argent insérée depuis le dernier choix ou la dernière demande de remboursement.

1	*binding*	::=	*name* := *exp*
2		|	*binding1*;*binding2*
3	*pattern*	::=	*object.name*
4		|	*object.name*(name1, name2,..., namen)
5		|	*pattern1* **or***pattern2*
6		|	*pattern* **where** *exp*
7	*relation*	::=	*pattern* **udpates** *binding*
8		|	*exp* **disables** *pattern*
9		|	**atomic**(*pattern1*, ..., *patterni*)
10		|	*pattern* **stops**
11		|	*relation1* **,** *relation2*
12	*synchronizer*	::=	*name*(*name1* , ... ,*namen*)
13			{[**init** *binding*]
14			*relation*
15			}

Figure D.1: Syntaxe abstraite des synchronizers

Des précisions sur les synchronizers. L'observation et le contrôle des invocations de méthodes sont définies en utilisant de la reconnaissance de formes *(pattern matching)*. La deuxième règle de la figure D.1 exprime les différentes formes (*patterns*) possibles:

- *o.n* reconnaît tous les messages invoquant la méthode *n* de *o* (ligne 3).
- *o.n(x1, ..., xn)* reconnaît tous les messages invoquant la méthode *n* et lie les valeurs de paramètres de la méthode aux variables *x1, ..., xn* (ligne 4).
- Les messages reconnus par le pattern *p1*or*p2* sont des messages reconnus soit par *p1* soit par *p2* (ligne 5).
- Un message sera reconnu par le pattern *p* where*exp* si le message est reconnu par *p* et que l'expression *exp* est vraie.

Une fois la reconnaissance de méthodes présentée, il nous faut voir le coeur des synchronizers: les opérateurs updates, disables et atomic qui définissent la sémantique du synchronizer (règles relation lignes 7 à 11 figure D.1). Ces trois opérateurs permettent de spécifier les conséquences ou les conditions associées à la réception du message par un objet contraint: il s'agit des *relations* d'un synchronizer.

udpates: *p* updates *b* spécifie qu'à chaque invocation de méthodes reconnues par *p*, l'état du synchronizer change suivant *b*. Il est important de bien remarquer que *b*représente en fait une liste d'affectation de variables du synchronizer et non d'envoi de messages à d'autres objets. Il s'agit d'adapter l'état du synchronizer. Lorsque plusieurs formes reconnaissent une méthode pour un même synchronizer, l'ordre d'exécution est indéterminé et effectué après que toutes les reconnaissances aient eu lieu[2].

disables : *exp* disables *p* spécifie que les méthodes reconnues par *p* ne seront appliquées que si l'expression *exp* est vraie pour l'état courant du synchronizer. Lorsque *exp* est évaluée à faux, les méthodes sont suspendues. Cependant, il n'est pas clairement indiqué comment l'exécution de ces méthodes se fondra dans les appels de messages à ces objets, lorsque les expressions qui les conditionnent deviendront vraies.

atomic : l'opérateur atomique atomic(*p1,...,pn*) spécifie que les différentes méthodes reconnues par les *p1,...,pn* doivent s'exécuter comme une seule méthode, c'est-à-dire qu'aucune notion d'ordre due au temps ne doit intervenir. Ainsi lorsque l'invocation d'une méthode contrôlée par cet opérateur arrive, elle est stockée en attendant que l'invocation de toutes les autres méthodes.

Les observations faites par un synchronizer sont consistantes par rapport au temps : ainsi si on invoque la méthode m2 après la méthode m1, les synchronizers observent m2 après m1. Un message donné peut être reconnu à la fois par les patterns *updates* et *disables* d'un même synchronizer. Dans un tel cas, l'expression associée à l'opérateur disables est tout d'abord évaluée dans le contexte du synchronizer, état qui est changé ensuite par l'opérateur updates.

Au niveau de l'implémentation, le compilateur traduit ces synchronizers en messages entre les objets contraints. Les synchronizers ne sont pas des objets de pleins droits et ils ne sont accessibles par envoi de messages.

Discussion. Les Synchronizers permettent d'abstraire du code des objets leurs possibles synchronisations. La définition des objets est simple. La réutilisation à la fois de ces objets et de leur coordination est améliorée. La distinction entre les fonctionnalités des objets et les schémas de coordination est claire.

Les synchronizers introduisent l'interdiction de méthodes (opérateur disables). Il faut cependant préciser quelques points sur lesquels diffèrent nos travaux de ceux présentés ici. Ces divergences sont principalement dues au mode de programmation choisi et aux objectifs fixés. Notre cadre de programmation est une programmation séquentielle, et non concurrente, et nos objectifs ne sont pas la spécification d'une coordination entre objets mais d'une inter-communication. Pour notre part, nous pensons que l'inter-communication inclut la coordination entre objets[3].

La principale différence provient de la portée des opérateurs : les opérateurs (udpates et disables) des synchronizers ne portent que sur l'état des synchronizers. A aucun moment, ces opérateurs n'ont pour vocation de modifier l'état d'un autre objet contraint. Agha et Frølund sont conscients de cette lacune *« It might be desirable for synchronizers to be able to receive messages as well as trigger new activities. The ability to trigger activities would provide an elegant way of extending the vending machine to give back change: successful opening of a slot triggers a refund of the money remaining in the accepter. »* [Frøl 93]

Plusieurs synchronizers peuvent contrôler un même objet ; plusieurs gardes peuvent donc spécifier des conditions différentes pour l'acceptation d'un même message. La nécessité de vérifier toutes les gardes peut donc amener à des incohérences du comportement souhaité. L'expression locale de synchronizers peut à un niveau global poser des problèmes d'incohérence vis à vis de la sémantique locale d'un synchronizer.

D.2 Procol

« A protocol in a object regulates the message traffic to the object by ordering access in time, and by allowing/disallowing messages » [van 91].

Procol est un langage objet concurrent proposant des protocoles de délégation et des contraintes [van 91]. Nous n'entrerons pas ici dans le détail du modèle objet proposé par les auteurs de Procol, nous renvoyons le lecteur à [van 91, van 89]. Par contre, nous présentons les aspects originaux de Procol : les protocoles de délégation et les contraintes.

D.2.1 Protocoles

Procol associe des protocoles à chaque objet. Un protocole spécifie déclarativement un contrôle des accès à l'objet à l'aide d'expressions régulières définies en termes de l'état de l'objet et du passé de la communication. Il permet d'ordonner et de réduire l'accès aux méthodes (Actions) de l'objet. Un protocole n'est pas exécutable mais est utilisé lors de l'exécution pour déterminer les méthodes pouvant être invoquées à un instant donné.

Un protocole est défini à l'aide d'opérateurs (+ ou, ; séquence, * répétition) manipulant des *termes d'interactions*. Un terme d'interaction associe la réception d'un message à l'exécution d'une méthode de l'objet. Ces expressions sont enrichies de prédicats exprimés en fonction de l'état de l'objet : des *gardes* (opérateur:). Une garde est évaluée afin de déterminer si le message est accepté ou pas par l'objet. Lorsque plusieurs gardes existent elles doivent toutes être évaluées.

[2] Par comparaison, on peut penser à l'évaluation des définitions de variables locales dans la forme spéciale `let` du langage Scheme.

[3] Pour nuancer notre propos nous dirons que traiter l'inter-communication dans un monde concurrent inclut la coordination dans un tel monde). Cependant, dans cette thèse, nous n'avons pas exploré cet aspect, qui se précise pour nous comme une des principales voies pour nos futures recherches (voir13).

L'état d'un objet, et donc d'un procotole, change lorsqu'un message a été reconnu par un terme d'interaction, un nouveau terme d'interaction devient le terme courant. Une analogie peut être faite avec le processus d'analyse d'un compilateur dont la grammaire serait un protocole. Par exemple, le protocole suivant défini sur l'objet `S` : $A(msg1) \rightarrow actionA*; B(msg2) \rightarrow actionB$ spécifie que le premier objet a pouvoir accéder à l'objet `S` est `A` ; que l'envoi par `A` du message `msg1` impliquera l'invocation de la méthode `actionA`. L'objet `A` pourra répéter cet envoi de `msg1` un certain nombre de fois, mais la fin de ce protocole spécifie que cette répétition de messages doit être suivie d'un envoi du message `msg2` par l'objet `B`. Un protocole joue plusieurs rôles : il spécifie l'interface de l'objet à la manière des Adaptors de [Yell 94], séquence les interactions entre objets et contrôle les accès aux méthodes.

D.2.2 Propagateurs de contraintes

« Propagators can be used to maintain relationships between objects, or to visualize non-visual algorithms, or to simply monitor all accesses to a particular object. » [van 91].

Les auteurs de Procol ont souhaité une intégration de contraintes qui n'impliquent pas une modification du code des objets contraints et respectent le principe de l'encapsulation. Ils proposent des contraintes mono-directionnelles exprimées, non pas en termes de changement de valeurs des variables, mais en termes de méthodes. La sémantique d'une contrainte est qu'à chaque fois qu'un objet reçoit un message sur lequel porte une contrainte, le code associé à cette contrainte est exécuté.
Ainsi la contrainte suivante : `constraint Point1.Move(x, y)→Point2.Move(x + 5, y );` exprime qu'à chaque fois que l'objet `Point1` reçoit le message `Move`, l'objet `Point2` reçoit le message `Move` de telle sorte qu'il soit aligné à la droite de `Point1` à une distance de 5 unités.

Les contraintes de Procol permettent d'animer des algorithmes à la manière de la visualisation d'algorithmes de Animus [Born 86b]. L'exemple suivant permet de visualiser un tableau d'entier `array` par un objet de visualisation `bargraph`.

```
Obj       DEMO (INTARRAY array)
Declare   BARGRAPH bargraph;
          int i, val;
Init      new bargraph;
           constraint array.SetValue(i,val) → bargraph.SetValue(i,val);
EndObj    DEMO.
```

D.2.3 Discussion

Plus que de définir des relations entre objets, les protocoles définissent des interfaces plus élaborées des objets incluant le séquencement et des conditions d'acceptations des messages et sont à rapprocher des protocoles explicites de [Yell 94].

Le choix de préserver l'encapsulation en n'exprimant des contraintes en termes exclusivement de méthodes est pour nous fondamental. Cependant, ces contraintes sont étroitement liées aux classes des objets sur lesquelles elles portent. Les propagateurs de contraintes se rapprochent des contraintes de Animus.

E

Code de FLO/ObjVLisp

Nous donnons ici le code complet d'une version de FLO/ObjV. Le système résultant possède les caractéristiques décrites dans le dernier chapitre de cette thèse c'est-à-dire fusion de ObjVlisp et de FLO: bootstrap avec noyau de classes autonomes et gestion de l'héritage à base de dépendances. L'objectif n'est pas d'implémenter tout FLO mais seulement d'implémenter les mécanismes nécessaires à la définition de l'héritage par le biais de dépendances. Aussi nous nous sommes permis de nombreux raccourcis: la hiérarchie est simplifiée à l'extrême, les opérateurs gérés le plus simplement possible, la gestion de l'héritage entre dépendances n'est mis en place et certains points du mop ne sont pas implémentés. Malgré ces restrictions, ce code est complet et fonctionne.

Certaines parties du code présenté, comme le choix des noms des variables et des fonctions, sont inspirées du source du modèle ObjVlisp présenté dans [Coin 87].

```
;;;;*****************************************************************
;;; FLO/OBJVLISP: may 1995.
;;;;*****************************************************************
;;; structure of an objet
;;;    - class
;;;    - meta

;;; structure of class:
;;;    - class
;;;    - meta
;;;    - name
;;;    - iv
;;;    - keywords
;;;    - methods
(define *eval-trace* #t)
(load "pp") ;; un pretty printer.
(define *t* #f)
;;*****************************************************************
;; slots
;;*****************************************************************
(define *uninitialized* 'nv)

(define-macro (create-primitives slot offset)
  (let ((setname (string->symbol (string-append (symbol->string slot) "!"))))
    `(begin
       (define ,slot (lambda (instance) (vector-ref instance ,offset)))
       (define ,setname (lambda (instance val) (vector-set! instance ,offset val))))))

(create-primitives isit 0)
(create-primitives meta 1)
(create-primitives name 2)
(create-primitives iv 3)
(create-primitives keywords 4)
(create-primitives methods 5)

(define (iv* x) (eval x))
(define (class-of instance) (iv* (isit instance)))

(define (give-offset slot instance)
  (do ((i 1 (+ i 1))
       (x (cdr (iv (class-of instance))) (cdr x)))
      ((equal? (car x) slot) i)))
;; test: (give-offset 'name test-class) -> 2
```

```
(define (allocate class-name)
  (let ((vect (make-vector (length (iv (iv* class-name))) *uninitialized*)))
    (isit! vect class-name)
    vect))
;; test: (allocate 'Test-class)

(load "prim") ;; loading some primitives (every, some...)

(define (initargs->slotsvalues initargs keywords)
 ;; (key1 val1 key2 val2 ...) * (key2 keyn ... key1) -> (val2 valn ...val1)
  (map (lambda (x) (get-keyword x initargs *uninitialized*)) keywords))

;; test: (initargs->slotsvalues '(:lulu 5 :bob 2) '(:bob :titi :toto :lulu))
;; -> (2 nv nv 5)

(define (initialize-iv object initargs)
 ;; object x initargs list -> initialized instance
  (do ((i 1 (+ i 1))
       (x  (initargs->slotsvalues initargs (keywords (class-of object)))
           (cdr x)))
      ((null? x) object)
    (vector-set! object i (car x))))

;; test:
;; (define person (initialize-iv
;;              (allocate 'Test-Class)
;;              '(:name person :iv (age name address))))
;;****************************************************************
;; methods
;;****************************************************************
(define (prepare-methods l)
 ;; list of couple (selector, lambda) -> hash-table
  (if (list? l)
      (let ((h (make-hash-table)))
        (do ((lsel-meth l (cddr lsel-meth)))
            ((null? lsel-meth) h)
          (hash-table-put! h (car lsel-meth) (iv* (cadr lsel-meth))))
        h)
      (make-hash-table)))

(define (add-method class selector body)
 ;; class x symbol x lambda -> side effect
  (let ((m (methods class)))
    (if (equal? m *uninitialized*)  ;; there is no method before
        (let ((h (make-hash-table)))
          (hash-table-put! h selector body) ;; physic modification
          (methods! class h))
       ;; there are some methods, we add the new one
        (hash-table-put! (methods class) selector body))))

(define (selectors class)
  (hash-table-for-each (methods class)
                       (lambda (x y) (format *t* "~a~%" x)
                               (pp (procedure-body y)))))

(define-macro (declare object)
 ;; object -> side effect
  `(let ((n (name ,object)))
     (eval `(define ,n ,,object))))

;; the variable self refers the receiver of the message in the
;; definition of a method. Therefore, we only
;; write other elements of the signature of methods.
(define-macro (mkmethod s . l)
  `(lambda ,(cons 'self s) ,@l ))

(define *no-method* 'no-method)
(define (is-method? x) (not (equal? *no-method* x)))

(define (get-method class selector)
  (let ((res (hash-table-get (methods class) selector *no-method*)))
    (format *trace-1*
            "PRIM GET-METHOD:selector ~a sur ~a => method ~a~%"
            selector (name class) res)
    res))

(define (methods-down-all de a)
 ;; to move all the methods from an ancestor to a descendant class
  (hash-table-for-each (methods de)   (lambda (x y) (add-method a x y))))
```

```
(define (methods-down-selected de a lofselectors)
  ;; to move the selected methods from an ancestor to a descendant class
  (for-each (lambda (sel) (add-method a sel (get-method de sel)))
            lofselectors))

(define *isit* '())
;; this variable refers to the current class for the method lookup.
;; A preferable way to do this is to use a closure.

(define (not-meta? x) (equal? x *uninitialized*))
;; ****************************************************************
;; Some little tools
;; ****************************************************************
(define (fcl obj)
  ;; to make provide a clean trace
  (if (member (isit obj) '(class <metalink>))
      (name obj)
      (string-append "a-" (symbol->string (name (class-of obj))))))

(define (fargs l)
  ;; to make provide a clean trace
  (letrec ((farg (lambda (x) (cond ((vector? x) (fcl x)) (else x)))))
    (map farg l)))

(define f format)
;;;****************************************************************
;;; the send primitive.
;;;****************************************************************
(define (send obj selector . args)
  (let ((m (if (and (equal? selector 'lookup) (equal? obj class))
             (get-method class 'lookup)
             (send (class-of obj) 'lookup selector obj))))
    (let ((a-meta (meta obj)))
      (if (not-meta? a-meta)
          (if (is-method? m)
              (apply m obj args)
              (send obj 'error selector))
          (send a-meta 'control m selector (cons obj args))))))
;;;****************************************************************
;;; 1 class skeleton
;;;****************************************************************
(define class '())
(set! class
      (vector
       'class
       *uninitialized*
       'class
       '(isit meta name iv keywords methods)
       '(:meta :name :iv :keywords :methods)
       (prepare-methods
        '(lookup
          (mkmethod (selector object)
                    (format *t* "avB meth lookup sur ~a pour ~a~%"
                            (name self) selector)
                    (get-method self selector))
          basicnew
          (mkmethod ()
                    (format  *t* "avB allocate sur class~%")
                    (allocate (name self)))
          class?    (mkmethod () #t)
          new
          (mkmethod initargs
                    (format *t* "avB new sur class~%")
                    (send (send self 'basicnew) 'initialize initargs))
          initialize
          (mkmethod initargs
                    (format *t* "avB initialize sur class~%")
                    (initialize-iv self (car initargs))
                   ;; we don't have super
                    (keywords! self (make-keywords (iv self)))
                    (methods! self (prepare-methods (methods self)))
                    (declare self)
                    self)))))

(add-method  class 'name  (mkmethod () (name self)))
(add-method  class 'class? (mkmethod () #t))

;; test:
;; (apply (get-method class 'basicnew) (list class))
;; (send class 'lookup 'basicnew class)
```

```
;; (send class 'basicnew) -> #(class nv nv nv nv nv)
;; lulu inherits by hand from object.
;; take care to define initialize and isit and meta
'(
(send class 'new :name 'lulu :iv '(isit meta x y)
     :methods
      '(display    (mkmethod () (format *t* "jjkjlj~%"))
        initialize (mkmethod  initargs
                              (format  *t* "initialize de la classe Lulu ~a~%"
                                       (car initargs))
                              (initialize-iv self (car initargs)))))
(define l1 (send lulu 'new :x 3 :y 15))
(send l1 'display) ;; fine
)
(format *eval-trace* "skelet class~%")
;;;*****************************************************************
;; 2 object
;;;*****************************************************************
(send class 'new :name 'object :iv '(isit meta)
     :methods
      '(?          (mkmethod (iv) (vector-ref self (give-offset iv self)))
        ?<-        (mkmethod (iv val)  (vector-set! self (give-offset iv self) val))
        class      (mkmethod () (class-of self))
        metaclass?  (mkmethod () #f)
        class?     (mkmethod () (send (class-of self) 'metaclass?))
        error      (mkmethod (msg)
                          (format *t* "~%Unknown selector ~a for class ~a~%"
                                  msg (isit self)))
        initialize    (mkmethod  initargs
                                 (format *t*
                                         "initialize de la classe Object ~a~%"
                                         (car initargs))
                                 (initialize-iv self (car initargs)))))

(methods-down-selected object class '(? ?<- class error))
;; the class class is an autonomous class, so it inherits by hand
;; from object

(format *eval-trace* "object~%")
;;;*****************************************************************
;; 3 Meta-Object
;;;*****************************************************************
(send class 'new :name '<metaboot> :iv '(isit meta relations)
     :methods
      '(relations
        (mkmethod () (vector-ref self (give-offset 'relations self)))
        relations!
        (mkmethod (val) (vector-set! self (give-offset 'relations self) val))
        control
        (mkmethod (method selector args)
          (format *t*
                  "CONTROL:~% Method ~a Selector~a~% ARGS~a~%" method selector args)
          (let ((res (apply method args))
                (llkf (get-link (send self 'relations)
                                (do-key (car args)
                                        (with-op '-> selector)))))
               (format *t* "RESULTat de l'application ~a et liens~a~%" res llkf)
               (if (pair? llkf)
                   (car (map (lambda (x)
                               (send x 'firing '-> selector args res))
                             llkf))
                   res)))
          initialize
          (mkmethod initargs
                    (format *t* "initialize <meta>boot ~%")
                    (let ((i (initialize-iv self (car initargs))))
                      (send i 'relations! (make-hash-table))
                      i))))

(define *default-meta-res* (send <metaboot> 'new))
(methods-down-selected object <metaboot> '(? ?<- class metaclass? error))
(format *eval-trace* "metaboot~%")
;;;*****************************************************************
;;; 4 Metalink:
;;;*****************************************************************
(load "schemas-flo")
(load "new-hashes13")
(load "divers-flo-obj")

(send Class 'new :name '<metalink>
```

```
     :iv '(isit meta name iv keywords methods behavior)
     :methods
      '(behavior
        (mkmethod () (vector-ref self (give-offset 'behavior self)))
        behavior!
        (mkmethod (val)  (vector-set! self (give-offset 'behavior self) val))))

(add-method  <metalink>  'lookup
             (mkmethod   (selector object) (get-method self selector)))


(add-method <metalink> 'basicnew
            (mkmethod ()
                      (format *t* "allocate sur metalink~%")
                      (allocate (name self))))

(add-method  <metalink> 'new
             (mkmethod initargs
                       (format *t* "new sur metalink~%")
                       (let ((i (send (send self 'basicnew) 'initialize initargs)))
                         (send i 'manage-keywords initargs)
                         (send i 'establish-link)
                         i)))

(add-method <metalink> 'initialize
            (mkmethod initargs
                      (format *t* "INITIALIZE sur METALINK(on cree une classe lien)~%")
                      (initialize-iv self (car initargs))
                      (keywords! self (make-keywords (iv self)))
                      (methods! self (prepare-methods (methods self)))
                      (declare self)
                      (send self 'behavior!
                            (map
                             (lambda (y)
                               (let* ((rs (bsch y)))
                                 (do-schemas (oper y) (asch y)
                                             (do-absch (prepare-funct (funct rs))
                                                       (search-and-eval-fct (sign rs))))))
                             (get-keyword :behavior (car initargs) '())))
                      self))

(methods-down-selected class <metalink> '(name))
(methods-down-selected object <metalink> '(? ?<- class error metaclass?))
(format *eval-trace* "metalink~%")
;;;*****************************************************************
;;5 link: instance of metalink inheriting from object.
;;;*****************************************************************
(send <metalink> 'new :name '<link> :iv '(isit meta link-info)
     :methods
      '(link-info
        (mkmethod () (vector-ref self (give-offset 'link-info self)))
        link-info!
        (mkmethod (val) (vector-set! self (give-offset 'link-info self) val))))

(add-method  <link>
             'initialize
             (mkmethod initargs
                       (format *t* "INITIALIZE <link> (init link-info)~%")
                       (let ((i (initialize-iv self (car initargs))))
                         (send i 'link-info! (make-hash-table))
                         i)))

(add-method  <link>  'action-at-creation-time
             (mkmethod () (format *t* "action-at-creation-time de <link>~%") #t))

(add-method <link>  'establish-link
            (mkmethod ()
                      (format *t* "ESTABLISH on <link>~%")
                      (do-register 'relations self (send (class-of self) 'behavior))
                      (send self 'action-at-creation-time)))

(add-method  <link> 'manage-keywords
             (mkmethod (initargs)
                       (format *t* "MANAGE-KEYWORDS on <link>(on remplit link-info)~%")
                       (let ((h (send self 'link-info)))
                         (let loop ((ini initargs))
                           (unless (null? ini)
                                   (let ((symkey (key->symbol (car ini))))
                                     (hash-table-put! h symkey (cadr ini))
                                    ;; we stock keywords and its value
```

```
                                      (if (pair? (cadr ini))
                                          (for-each
                                           (lambda (x)
                                             (hash-table-put! h x
                                                              (create-symrec symkey)))
                                           (cadr ini))
                                         ;; we have a list we noted receiver
                                         ;; all its elements
                                          (hash-table-put! h (cadr ini) symkey)))
                                   ;; else we simply note the given associated keywords
                                    (loop (cddr ini))))
                         ;; and we add :link keyword
                          (hash-table-put! h 'link self)
                          (hash-table-put! h  self 'link))
                        self))

(define (do-register where lk l-sch)
  (for-each (lambda (x)
              (let* ((op (oper x))
                     (a  (asch x))
                     (gfname (funct a))
                     (ogf (with-op op gfname)) ; do 'omethodname
                     (key (car (sign a))))
                (if (is-receiver? key)
                   ;; we register all objects having this keywords
                    (for-each (lambda (obj)
                                (add-put-link (send (meta obj) where)
                                              (do-key obj ogf) lk)
                                )
                              (allreceiver lk (key->symbol key)))
                    (let ((obj (hash-table-get (send lk 'link-info)
                                               (key->symbol key))))
                      (add-put-link (send (meta obj) where)
                                    (do-key obj ogf) lk)
                      ))))
            l-sch))

(add-method  <link> 'firing
             (mkmethod (operator gfname args . opt)
                       (format *t* "FIRING on <link>~%")
                       (let* ((sch (schemas->schema (send (class-of self) 'behavior)
                                                    operator gfname
                                                    (symbol->key
                                                     (hash-table-get
                                                      (send self 'link-info) (car args)))))
                              (bs (bsch sch))
                              (nargs (give-args-for-small
                                      self (sign (asch sch)) args (car opt) (sign bs)
                                      (car args))))
                         (let ((f (funct bs)))
                           (if (symbol? f)
                               (apply send (car nargs) f (cdr nargs))
                               (apply f nargs)))
                         )))

(add-method  <link> 'give
             (mkmethod (what)
                       (format *t* "GIVE on <link>~%")
                       (if (is-receiver? what)
                           (allreceiver l (key->symbol what))
                           (hash-table-get (send self 'link-info) (key->symbol what)))))

(methods-down-selected object <link> '(? ?<- error class class? metaclass?))
(format *eval-trace* "link~%")
;; ****************************************************************
;; some new tools
(define (add-ordered l1 l2)
  (if (null? l2) l1
      (if (member (car l2) l1)
          (add-ordered l1 (cdr l2))
          (add-ordered (append l1 (list (car l2))) (cdr l2)))))

(define (remove-duplicates lslots)
;; list -> list
;; remove all the same elements but order is conserved
    (do ((l lslots (cdr l))
         (res ()))
        ((null? l) res)
      (unless (member (car l) res)
              (set! res (append res (list (car l)))))))
```

```
;; test: (remove-duplicates '(a a b d c d))
;; -> (a b d c)
;;;***************************************************************
;; le run-super:
(define-macro (run-super obj selector args)
  `(let ((i *isit*))
     (apply
      (send
       (hash-table-get
        (send (car (get-link (send (meta i) 'relations)
                             (do-key i '->lookup)))
               'link-info) 'ancetre)
       'lookup ,selector ,obj)
      ,obj ,args)))

;;;***************************************************************
;;; the last class of the kernel : inherit-from-last
;;;***************************************************************
;;;; inherit-from-last is another name for inherit-with-error.
;;;; this dependancy manage an error and has to be declared between
;;;; an class and a kernel class.
(send <metalink> 'new :name 'inherit-from-last :iv '(isit meta link-info)
      :behavior
      '((-> (lookup (:descendant selector obj))
            ((lambda (des anc obj selector lookupres)
               (if (is-method? lookupres)
                   (begin (set! *isit* des) lookupres)
                   (let ((m (get-method anc selector)))
                     (if (is-method? m)
                         m
                         (send obj 'error selector)))))
             (:descendant :ancetre obj selector :result)))))

(methods-down-all <link> inherit-from-last)

(add-method  inherit-from-last
             'action-at-creation-time
             (mkmethod ()
                       (let ((a (send self 'give :ancetre))
                             (d (send self 'give :descendant)))
                         (keywords! d (add-ordered (keywords a) (keywords d)))
                         (iv! d (remove-duplicates (append (iv a) (iv d)))))))

;;;; ***************************************************************
;;;; now we can program with this strange kernel
;;;; ***************************************************************
;;; for example, we define the dependancy inherit-simple using the
;;; inherit-from-last dependancy
(send <metalink> 'new :name 'inherit-from-simple   :iv '()
      :behavior
      '((-> (lookup (:descendant selector obj))
            ((lambda (des anc obj selector lookupres)
               (format *t*
                       "INHERIT SIMPLE avec desc ~a, meth ~a, res du lookup ~a~%"
                       (name des) selector lookupres)
               (if (is-method? lookupres)
                   (begin
                    ;; we mark the clas sin which we found the method
                    ;; for the run-super
                     (set! *isit* des)
                   lookupres)
                 (begin
                   (send anc 'lookup selector obj))))
            (:descendant :ancetre obj selector :result)))))

(meta! inherit-from-simple *default-meta-res*)
(define inherit-last-between-link-last-and-inherit-simple
  (send inherit-from-last 'new :ancetre inherit-from-last
                               :descendant inherit-from-simple))
;;;;*************************************************************
;;;; test  with point et colored point classes
;;;;*************************************************************
'(
(send class 'new :name 'point :iv '(x y)
       :methods
        '(point-display  (mkmethod ()
                                   (format #t "x: ~a y:~a ~%"
                                           (send self '? 'x)
                                           (send self 'y)))
```

```
         y  (mkmethod () (vector-ref self (give-offset 'y self)))))

(meta! point *default-meta-res*)
(define point-inherit-from-object
    (send inherit-from-last 'new :ancetre object :descendant point))
;; point class inherits from class object with an instance of inherit-from-last

(define p1 (send point 'new :x 12 :y 50))
(send p1 '? 'x)
(send p1 'y)
(send p1 'point-display)
;;(send p1 'lulu) ;; this leads to an error, ok

(send class 'new :name 'c-point :iv '(color)
       :methods
        '(point-display
          (mkmethod ()
                    (run-super self 'point-display '())
                    (format #t "color ~a~%" (send self 'color)))
          color
          (mkmethod ()
                    (vector-ref self (give-offset 'color self)))))

(meta! c-point *default-meta-res*)
(define c-point-inherit-from-point
  (send inherit-from-simple 'new  :ancetre point :descendant c-point))
;; (send c-point-inherit-from-point 'lulu)  this leads to an error

(define p2 (send c-point 'new :x 12 :y 15 :color 'blue))
(send p2 'color)
(send p2 'y)
(send p2 'point-display)
)
;;;;*****************************************************************
;;;; First extension : dynamic slot inheritance.
;;;; we define a new meta class for-dyn that uses add-iv instead a primitive.
;;;; second we define two dependancies :
;;;; inherit-with-dynamic-from-last and inherit-with-dynamic-simple

;;;; a best version should factorize the common behavior of these two
;;;; dependencies in a superclass inherit-with-dynamic.

'(
;; we define a new class for-dyn that inherits from class
;; this class use the method add-iv to add dynamic slots.
(send class 'new :name 'for-dyn :iv '()
     :methods '(add-iv  (mkmethod (val) (format #t "ADD-IV ~%")
                                 (iv! self val))))

(meta! for-dyn *default-meta-res*)
(define class-for-dyn  (send inherit-from-simple 'new :ancetre class :descendant for-dyn))
)

'(
(send for-dyn 'class?)
(send for-dyn 'name)
(send for-dyn 'basicnew)

;;test with dyn-point and without dynamique link
(send for-dyn 'new :name 'dyn-point :iv '(x y))

(meta! dyn-point *default-meta-res*)
(define point-inherit-from-object
  (send inherit-from-last 'new :ancetre object :descendant dyn-point))
(define p1 (send dyn-point 'new :x 12 :y 16))
(send p1 '? 'x)
)
;;;****************************************************************
'(
;; as we do not have define inheritance between dependancies we must to
;; rewrite some code. With inherhitance between dependancy we just have to
;; specify the second implies rule.

(send <metalink> 'new :name 'inherit-with-dynamic-from-last  :iv '()
     :behavior
      '((-> (lookup (:descendant selector obj))
           ((lambda (des anc obj selector lookupres)
              (if (is-method? lookupres)
                  (begin (set! *isit* des) lookupres)
                  (let ((m (get-method anc selector)))
```

```
                    (if (is-method? m)
                        m
                        (send obj 'error selector)))))
            (:descendant :ancetre obj selector :result)))
       (-> (add-iv (:ancetre new))
           ((lambda (des new)
              (send des 'add-iv  (remove-duplicates (append new (iv des))))
              (keywords! des (make-keywords (iv des))))
            (:descendant new)))))

(meta! inherit-with-dynamic-from-last *default-meta-res*)
(define inherit-simple-between-last-and-dynamic-last
  (send inherit-from-simple 'new
       :ancetre inherit-from-last :descendant inherit-with-dynamic-from-last))

(add-method  inherit-with-dynamic-from-last 'action-at-creation-time
             (mkmethod ()
                       (let ((a (send self 'give :ancetre))
                             (d (send self 'give :descendant)))
                         (send d 'add-iv (remove-duplicates (append (iv a) (iv d))))
                         (keywords! d (make-keywords (iv d))))))

;;; Now with simple
(send <metalink> 'new :name 'inherit-with-dynamic-simple :iv '()
     :behavior
      '((-> (lookup (:descendant selector obj))
           ((lambda (des anc obj selector lookupres)
              (if (is-method? lookupres)
                  (begin (set! *isit* des) lookupres)
                  (send anc 'lookup selector obj)))
            (:descendant :ancetre obj selector :result)))
       (-> (add-iv (:ancetre new))
           ((lambda (des new)
              (send des 'add-iv  (remove-duplicates (append new (iv des))))
              (keywords! des (make-keywords (iv des))))
            (:descendant new))) ))

(meta! inherit-with-dynamic-simple *default-meta-res*)
(define inherit-simple-between-simple-and-dynamic-simple
  (send inherit-from-simple 'new
       :ancetre inherit-from-simple :descendant inherit-with-dynamic-simple))

(add-method  inherit-with-dynamic-simple 'action-at-creation-time
 (mkmethod ()
          (let ((a (send self 'give :ancetre))
                (d (send self 'give :descendant)))
            (send d 'add-iv (remove-duplicates (append (iv a) (iv d))))
            (keywords! d (make-keywords (iv d))))))
)
'(
;;; test
;;; Now we define some instances of the for-dyn meta-class
(send for-dyn 'new :name 'dyn-colored-point :iv '(color))
(send for-dyn 'new  :name 'dyn-point :iv '(x y))
(meta! dyn-colored-point *default-meta-res*)
(meta! dyn-point *default-meta-res*)
(meta! object *default-meta-res*)

(define coloredpoint-inherit-from-point
  (send inherit-with-dynamic-simple
        'new :ancetre dyn-point :descendant dyn-colored-point))

(define point-inherit-from-object
  (send inherit-with-dynamic-from-last
        'new :ancetre object :descendant dyn-point))
;;; really slow !!!!
(define p1 (send dyn-point 'new :x 12 :y 16))
(send p1 '? 'x)
(define p2 (send dyn-colored-point 'new :x 24 :y 69))
(send p2 '? 'y)
)



;;;; ****************************************************************
;;;; a second extension : an inheritance dependancy with cache
;;;; ****************************************************************
(send <metalink> 'new :name 'caching :iv '(cache)
     :behavior
      '((-> (lookup (:descendant selector obj))
```

```
            ((lambda (des anc obj selector lookupres lk)
               (if (is-method? lookupres)
                   (begin (set! *isit* des) lookupres)
                   (begin
                     (format *t* "We look up ~a in the cache~%" selector)
                     (let ((c (look-in-cache lk selector)))
                       (if (is-method? c)
                           c
                           (let ((r (send anc 'lookup selector obj)))
                             (put-in-cache lk selector r)
                             r)))))))
            (:descendant :ancetre obj selector :result :link)))))

(meta! caching *default-meta-res*)
(define inherit-last-between-link-last-and-caching
  (send inherit-from-last 'new :ancetre inherit-from-last  :descendant caching))


(add-method  caching 'initialize
 (mkmethod initargs
           (let ((i (run-super self 'initialize initargs)))
             (send i 'cache! (make-hash-table))
             i)))

(add-method  caching  'cache (mkmethod () (vector-ref self 3)))

(add-method  caching  'cache! (mkmethod (val) (vector-set! self 3 val)))

;; primitive version
(define (cache lk)  (vector-ref lk 3))
(define (cache! lk v)  (vector-set! lk 3 v))
(define (look-in-cache lk selector) (hash-table-get (send lk 'cache) selector *no-method*))
(define (put-in-cache lk selector v)  (hash-table-put! (send lk 'cache) selector v))

;;;; ****************************************************************
;;;; an example
;;;; ****************************************************************
(send class 'new :name 'point :iv '(x y)
       :methods
        '(point-display
          (mkmethod ()  (format #t "x: ~a y:~a ~%" (send self '? 'x) (send self 'y)))
          x:
          (mkmethod (val) (vector-set! self (give-offset 'x self) val))
          x
          (mkmethod () (vector-ref self (give-offset 'x self)))
          y
          (mkmethod () (vector-ref self (give-offset 'y self)))))

(meta! point *default-meta-res*)
(define point-inherit-from-object
  (send inherit-from-last 'new :ancetre object :descendant point))
(define p1 (send point 'new :x 12 :y 50))
(send p1 '? 'x)
(send p1 'y)
(send p1 'point-display)

;;(send p1 'lulu)  ;;error fine

(send class 'new  :name 'c-point :iv '(color)
       :methods
        '(point-display  (mkmethod () (run-super self 'point-display '())
                              (format *t* "color ~a~%" (send self 'color)))
          color (mkmethod ()
                   (vector-ref self (give-offset 'color self)))))

(meta! c-point *default-meta-res*)
(define c-point-inherit-from-point
  (send caching 'new :ancetre point :descendant c-point))

;; (cache! c-point-inherit-from-point (make-hash-table))
;; (send c-point-inherit-from-point 'lulu)  fait bien une erreur

(define p2 (send c-point 'new :x 12 :y 15 :color 'blue))
(define p3 (send c-point 'new :x 12 :y 15 :color 'blue))
(procedure-body (look-in-cache c-point-inherit-from-point 'initialize))
(send p2 'color)
(send p2 'y)
(look-in-cache c-point-inherit-from-point 'y)
(procedure-body (look-in-cache c-point-inherit-from-point 'y))
(send p2 'point-display)
```

Bibliographie

[Ilo 88] Ilog. *SMECI Manuel de référence*, 1.4 edition, 1988. (pp 205, 213)

[Ilo 91] Ilog. *Broker*, 1991. Reference Manual. (p 32)

[Ilo 95] Ilog. *Power Classes*, 1995. Reference Manual. (p 216)

[Int 85] Intellicorp. *KEE v.2. Software Development System, User's Manual*, 1985. (pp 205, 213)

[AKSI 88] M. Aksit. *Data Abstraction Mechanisms in Sina/st.* In Proceedings of OOPSLA'88, pages 267–275, Septembre 1988. (p 226)

[AKSI 92a] M. Aksit and L. Bergmans. *Obstacles in Object-Oriented Software Development.* In Proceedings of OOPSLA'92, pages 341–358, 1992. (pp 11, 14, 233)

[AKSI 92b] M. Aksit, L. Bergmans, and S. Vural. *An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach.* In Proceedings ECOOP '92, LNCS 615, pages 372–395. Springer-Verlag, Juillet 1992. (p 226)

[AKSI 94] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. *Abstracting Object Interactions Using Composition Filters.* In Object-Based Ditributed Programming (ECOOP'93 workshop), LNCS 791, pages 152–184, 1994. (pp 15, 20, 26, 29, 30, 40, 225, 226)

[ALEX 77] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. A pattern language. Oxford University Press, 1977. (p 159)

[ALLE 94] R. Allen and D. Garlan. *Formalizing Architectural Connection.* In Proceedings of ICSE'94, 1994. (p 199)

[ANDE 83] J. Anderson. The architecture of cognition. Harvard University Press, 1983. (p 169)

[ANDE 92] E. P. Andersen and T. Reenskaug. *System Design by Composing Structures of Interacting Objects.* In Proceedings of ECOOP'92, LNCS 615, pages 133–152. Springer-Verlag, Juin 1992. (p 20)

[ARAP 90] C. Arapis. *Specifying Object Life-Cycles.* In Object Management, pages 133–195. Dennis Tsichritzis, 1990. Centre Universitaire de Genève. (p 225)

[ARAP 92] C. Arapis. *Object Behavior Composition: a Temporal Logic Based Approach.* In Proceedings of MFD-KBS, LNCS 495, pages 308–324, 1992. (p 26)

[ATTA 89] G. Attardi, C. Bonini, M. Boscotrecase, T. Flagella, and M. Gaspari. *Metalevel Programming in CLOS.* In S. Cook, éditeur, Proceedings of ECOOP'89, pages 243–256. Cambridge University Press, Juillet 1989. (p 115)

[AVES 90] P. Avesani, A. Perini, and F. Ricci. *COOL: An Object System with Constraints.* In Proceedings of TOOLS'90, pages 221–228, 1990. (p 213)

[BARR 93] P. Barril, M. Boufaida, and J.-F. Brette. *Class cooperation in a dedicated object system: the force authoring environment.* In Proceedings of the 10th TOOLS International Conference, pages 115–123, Versailles, Mars 1993. (pp 15, 209)

[BASS 91] L. Bass, R. Little, R. Pellegrino, S. Reed, R. Seacord, S. Shepperad, and M.R.Szezur. *The Arch Model: Seeheim Revisited.* In User Interface Developers' Workshop, ACM SIGCHI, Avril 1991. (p 171)

[BAST 95] R. Bastide and P. Palanque. *A Petri net based environement for the design of event-driven interfaces.* In Proceedings of ATPN'95, 1995. (pp 26, 27)

[BENO 89] C. Benoit, M. Bidoit, L. Henninger, and R. Velly. *Lore: An Object-based Programming Environment.* In Technology of Object-Oriented Languages and Systems, pages 469–481, CNIT Paris - La défense - France, November 1989. (p 31)

[BENV 91] A. Benveniste and G. Berry. *The Synchronous Approach to Reactive and Real-Time Systems.* In Proceedings of the IEEE, volume 79, sep 1991. (p 102)

[BERL 92] P. Berlandier. *A Study of Constraint Interpretation Mechanism and of their Interpretation in a Knowledge Language.* PhD thesis, University of Nice, 1992. (in French). (pp 25, 28, 40, 213)

[BERN 92] M. Berndtsson and B. Lings. *On Developing Reactive Object-Oriented Databases.* Data Engineering, Special issue on active databases, vol. 15, no. 1-4, pages 31–34, 1992. (pp 25, 32, 34, 35)

[Blah 92] M. Blaha, W. P. F. Eddy, W. Lorensen, and J. Rumbaugh. Object-oriented modeling and design. Prentice-Hall, 1992. (p 76)

[Blah 95] M. Blaha, W. P. F. Eddy, W. Lorensen, and J. Rumbaugh. Modélisation et conception orientées objet. Masson–Prentice-Hall, 1995. seconde édition. (pp 12, 14, 32, 76)

[Blak 87] E. Blake and S. Cook. *On Including Part Hierarchies in Object-Oriented Languages, with an Implementation in Smalltalk.* In Proceedings of ECOOP'87, LNCS 276, pages 41–50. Springer Verlag, Juin 1987. (pp 84, 85, 206)

[Bobr 77] D. Bobrow and T. Winograd. *An Overview of KRL, a Knowledge Representation Language.* Cognitive Science, vol. 1, no. 1, 1977. (pp 26, 205)

[Bobr 83] D. Bobrow and M. Stefik. *The LOOPS Manual*, 1983. (p 205)

[Bobr 86] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. *CommonLoops: Merging Lisp and Object-Oriented Programming.* In Proceedings OOPSLA'86, pages 17–29, November 1986. (p 111)

[Bobr 93] D. Bobrow, R. Gabriel, and J. White. *CLOS in Context – The Shape of the Design.* In Object-Oriented Programming: the CLOS perspective, pages 29–61. MIT Press, 1993. (p 112)

[Booc 91] G. Booch. Object-oriented design with applications. Benjamin–Cummings, 1991. (pp 11, 12, 14)

[Bore 90] N. Borenstein. Multimedia applications development with the andrew toolkit. Prentice Hall, 1990. (p 173)

[Born 79] A. Borning. *ThingLab – A Constraint-Oriented Simulation Laboratory.* PhD thesis, Stanford University, 1979. (pp 213, 219)

[Born 86a] A. Borning. *Classes versus prototypes in object-oriented languages.* In H. S. Stone and S. Winkler, éditeurs, Fall Joint computer conference, pages 36–40, Dallas, Texas, November 1986. IEEE. (pp 12, 13, 14, 15, 219)

[Born 86b] A. Borning and R. Duisberg. *Constraint-Based Tools for Building User Interfaces.* Transactions on Graphics, vol. 5, no. 4, pages 345–374, oct 1986. (pp 25, 27, 28, 31, 32, 37, 40, 70, 85, 100, 211, 213, 219, 220, 236)

[Born 87] A. Borning, R. Duisbserg, B. Freeman-Benson, A. Kramer, and M. Woolf. *Constraint Hierarchies.* In Proceedings of OOPSLA'87, pages 48–60, oct 1987. (pp 100, 151, 221)

[Born 92] A. Borning, B. Freeman-Benson, and M. Wilson. *Constraint Hierarchies.* Lisp and Symbolic Computational, no. 5, pages 223–270, 1992. (pp 100, 221)

[Borr 87] P. Borras. *CENTAUR : the system.* Rapport de recherche RR-777, INRIA, Decembre 1987. (p 95)

[Bosc 95a] J. Bosch. *Relations as Object Model Components.* Journal of Programming Languages, 1995. (pp 11, 12, 14, 20, 26, 28, 32)

[Bosc 95b] J. Bosch. *Layered Object Model: investigating paradigm extensibility.* PhD thesis, Lund University, Octobre 1995. Department of Computer Science. (pp 20, 138)

[Bosc 96] J. Bosch. *Language Support for Design Patterns.* In Proceedings of TOOLS'96, pages 197–210, 1996. (pp 159, 160, 162)

[Boua 94] M. Bouabsa, J. Brette, and P. Barril. *Les dépendances comme objets de première classe.* Rapport de recherche, Institut Blaise Pascal, 1994. MASI 94/24. (pp 14, 25, 40, 135, 216)

[Boua 95] M. Bouabsa. *Elaboration et prototypage d'interfaces homme-machine au moyen de contraintes: le système Plus.* PhD thesis, Université de Paris VI, 1995. (pp 14, 28, 34, 216)

[Boul 94] F. Boulanger, H. Delebecque, and G. Vdal-Naquet. Intégration de modules synchrones dans un cycle de développement par objets. 1994. (p 102)

[Bous 91] F. Boussinot and R. de Simone. *The Esterel Language.* In Proceedings of the IEEE, volume 79, 1991. (pp 16, 39, 102)

[Bout 93] V. Bouthors. *Egérie: un interprète embarqué dans un système de gestion d'interfaces homme-machine.* PhD thesis, Université de NICE SOPHIA-ANTIPOLIS, 1993. (pp 26, 32, 34, 205)

[Brac 83] R. Brachman. *What IS-A is and isn't: an analysis of taxonomic link in semantic network.* Computer, vol. 16, no. 10, pages 30–36, 1983. (pp 26, 205)

[Brac 85] R. J. Brachman. *On the Epistemological Status of Semantic Networks.* In R. J. Brachman and H. J. Levesque, éditeurs, Readings in Knowledge Representation, pages 191–215. Morgan Kaufmann Publishers, Inc, California, 1985. (pp 26, 205)

[Brac 90] G. Bracha and W. Cook. *Mixin-based Inheritance.* In Proceedings OOPSLA/ECOOP'90, pages 303–311, Octobre 1990. (pp 12, 185)

[Bran 96] S. Brandt and R. Schmidt. *The Design of a Meta-Level Architecture for the BETA Language.* In Proceedings of Reflection'96, 1996. (p 111)

[BRIF 96] X. Briffault and G. Sabah. Smalltalk: programmation orientée objet et développement d'applications. Eyrolles, 1996. ISBN: 2-212-08914-7. (pp 15, 27, 32, 173, 174, 210)

[BRIO 87a] J. Briot and P. Cointe. *The ObjVlisp Model: Definition of a Uniform, Reflexive, and Extensible Object-Oriented Language.* In Advances in Artificial Intelligence II, pages 225–232. Elsevier Science (North-Holland), 1987. (p 186)

[BRIO 87b] J. Briot and P. Cointe. *An Uniform Model for Object-Oriented Languages Using The Class Abstraction.* In Proceedings of IJCAI'87, 1987. (p 186)

[BRIO 89a] J.-P. Briot and P. Cointe. *Programming with Explicit Metaclasses in Smalltalk-80.* In Proceedings OOPSLA'89, pages 419–432, Octobre 1989. (pp 109, 111, 186)

[BRIO 89b] J. Briot. *Actalk: A Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment.* In S. Cook, éditeur, Proceedings ECOOP'89, pages 109–129. Cambridge University Press, Juillet 1989. (p 136)

[BRIO 94] J.-P. Briot. *Modèlisation et classification de langages de programmation concurrente à objets: l'experience Actalk.* In Langages et Modèle à Objets, pages 153–161, Grenoble, October 1994. IMAG, INRIA et PRC GDR IA. (p 111)

[BRIO 96] J. Briot. *An Experiment in Classification and Specialization of Synchronization Schemes.* In Proceedings of ISOTAS'96, LNCS 1049, pages 227–249. JSSST-JAIST, mar 1996. (pp 111, 136)

[BRIS 95] P. Brissi and R. Rousseau. *IREC: an Object-Oriented Abstract Representation to Handle Software Components in a Persistent Framework.* In Object-Oriented Technology for Database and Software Systems, pages 6–21. A. Vangalur and R. Missaoui, 1995. World Scientific Pub Co: ISBN 9810221703. (p 35)

[BUDD 91] T. Budd. An introduction to object-oriented programming. Addison-Wesley, 1991. (p 197)

[CARD 84] L. Cardelli. *A semantic of multiple inheritance.* In Lecture Notes in Computers Science, Semantics of data types, volume 173, pages 51–67. Springer-Verlag, New-York, 1984. (pp 12, 185)

[CARD 85] L. Cardelli and P. Wegner. *On Understanding Types, Data Abstraction, and Polymorphism.* Computing Surveys, vol. 17, no. 4, pages 471–521, December 1985. (pp 12, 185)

[CARL 93] P. Carle, J. Ferber, T. Furet, and D. Mousseau. *Synchronisation de messages par utilisation de la réflexivité du langage d'acteurs Mering IV.* In Journees Francophones des langages applicatifs. INRIA, 1993. (p 111)

[CARR 90] B. Carré and J.-M. Geib. *The Point of View notion for Multiple Inheritance.* In Proceedings of OOPSLA/ECOOP'90, pages 312–321, Ottawa, October 1990. ACM. (pp 12, 52, 185)

[CARR 96] B. Carré and G. Vanwormhoudt. *Conception modulaire en CROME. Le cas de l'interfaçage graphique d'objets.* In IHM'96, 1996. (p 52)

[CHAB 93] B. Chabrier. *Interfaces par Contraintes.* PhD thesis, Université de NICE SOPHIA-ANTIPOLIS, 1993. (pp 13, 52, 104)

[CHAB 94] B. Chabrier and P. Franchi-Zannettacci. *Délégation sémantique par contraintes réactives pour les interfaces graphiques. Semantic delegation with reactive contraints for graphical interfaces.* Technique et Sciences Informatiques, vol. 13, no. 4, pages 539–566, 1994. (p 104)

[CHAK 92] S. Chakravarthy, E. Hanson, and S. Su. *Active Data/Knowledge Base Reserch At The University of Florida.* Data Engineering, Special issue on active databases, vol. 15, no. 1-4, pages 35–38, 1992. (pp 25, 28, 32)

[CHAM 91] C. Chambers, D. Ungar, B.-W. Chang, and U. Holze. *Parents and Shared Parts of Objects: Inheritance and Encapsulation in SELF.* LISP and SYMBOLIC COMPUTATION: An international journal, vol. 4, no. 3, 1991. (p 214)

[CHIB 93a] S. Chiba and T. Masuda. *Designing an Extensible Distributed Language with a Meta-Level Architecture.* In Proceedings ECOOP'93, LNCS 707, pages 483–502, Kaiserslautern, Germany, Juillet 1993. Springer-Verlag. (pp 91, 111, 135, 137, 140)

[CHIB 93b] S. Chiba. *Open C++ Release 1.2 Programmer Guide.* Rapport de recherche 93-3, Dept of Information Science, University of Tokyo, 1993. (p 137)

[CHIB 95] S. Chiba. *A Metaobject Protocol for C++.* In Proceedings of OOPSLA'95, pages 285–299, Austin, Octobre 1995. ACM. (pp 111, 136)

[CIVE 93] F. Civello. *Roles for composite objects in object-oriented analysis and design.* In Proceedings of OOPSLA '93, ACM SIGPLAN Notices, pages 376–393, Octobre 1993. (pp 84, 85, 207)

[CLEM 91] D. Clement, F. Montagnac, and V. Prunet. *Integrated Software Components: a Paradigm for Control Integration.* In Proceedings of the European Symposium on Software Development Environments and CASE Technology, June 1991. (pp 28, 29)

[CLÉM 88] D. Clément and J. Incerpi. *Specifying the Behaviour of Graphical Objects Using Esterel.* Rr no 836, INRIA, 1988. (p 182)

[COAD 91] P. Coad and E. Yourdon. Object oriented analysis. Yourdon Press, 1991. 2 ed. (p 12)

[COIN 87] P. Cointe. *Metaclasses are first Class: The ObjVlisp Model.* In OOPSLA'87 Proceedings, pages 156–165, October 1987. (pp 21, 41, 42, 111, 112, 186, 190, 237)

[COIN 88] P. Cointe. *The ObjVlisp Kernel: A Reflective Lisp Architecture to Define a Uniform Object-Oriented System.* In Meta-Level Architectures and Reflection, 1988. (p 186)

[COIN 92] P. Cointe. *Clos and Smalltalk: a comparaison.* In Object oriented Programming: the CLOS Perspective, pages 215–250. MIT Press, 1992. (pp 109, 186)

[COLL 96a] P. Collet and R. Rousseau. *Classification et réification des assertions– Application au langage Eiffel.* In Proceedings of LMO'96, pages 10–27, 1996. (p 35)

[COLL 96b] P. Collet and R. Rousseau. *Assertions are object too.* In Proceedings of WOON'96, pages 25–39, 1996. (p 35)

[COOK 89] W. Cook and J. Palsberg. *A Denotational Semantics of Inheritance and its Correctness.* In Proceedings OOPSLA'89, pages 433–443, Octobre 1989. (pp 12, 185)

[COUT 87] J. Coutaz. *The Construction of User Interfaces and the Object Paradigm.* In Proceedings of ECOOP'87, LNCS 276, pages 121–130. Springer Verlag, Juin 1987. (pp 179, 180)

[COUT 89] J. Coutaz. *Architecture Models for Interactive Software.* In Proceedings of ECOOP'89, pages 383–399, 1989. (pp 22, 169, 170, 171, 177, 178, 184, 198)

[COUT 90] J. Coutaz. Interface homme-ordinateur: Conception et réalisation. Dunod informatique, 1990. (pp 178, 180)

[CUMM 95] F. Cummins and M. Ibrahim. *A Model of Reflection in Object-Oriented Languages.* In Proceedings of the IJCAI'95 workshop on Reflection and Meta-Level Architectures and their Applications in AI, pages 19–29, 1995. (p 136)

[DANF 94a] S. Danforth and I. R. Forman. *Reflections on Metaclass Programming in SOM.* In ACM, éditeur, Proceedings of OOPSLA'94, volume 29, pages 440–452, Portland, Octobre 1994. ACM. (pp 42, 111, 147)

[DANF 94b] S. Danforth and I. R. Forman. *Derived Metaclass in SOM.* In Proceedings of TOOLS EUROPE'94, pages 63–73, 1994. (pp 94, 147)

[DAYA 88] U. Dayal, A. P. Buchmann, and D. R. McCarthy. *Rules are Objects Too: A knowledge Model For An Active, Object-Oriented Database System.* In LNCS 334, Advances in OO Database Systems, pages 129–143, 1988. (pp 25, 32, 34, 35, 150)

[DAYA 96] U. Dayal, A. Buchmann, and S. Chakravarthy. The hipac project, chapitre 7, pages 177–206. Morgan Kaufman Publishers, 1996. (pp 25, 34, 40, 150)

[DE C 91] D. de Champeaux. *Object Oriented Analysis and Top-Down Software Development.* pages 360–375, 1991. (p 12)

[DEME 95] F.-N. Demers and J. Malenfant. *Reflection in logic, functional and object-oriented programming: a Short Comparative Study.* In Proceedings of the IJCAI'95 workshop on Reflection and Meta-Level Architectures and their Applications in AI, pages 29–38, 1995. (pp 111, 112)

[DERY 94] A. Dery and M. Fornarino. *Dépendances comportementales et modèle PAC.* In IHM'94, Lille, Decembre 1994. (pp 22, 179)

[DERY 96a] A. Dery, S. Ducasse, and M. Fornarino. *Objets et Dépendances.* In O. Mourad, éditeur, Ingénièrie Objet. Inter-éditions, 1996. (pp 21, 22)

[DERY 96b] A. Dery, S. Ducasse, and M. Fornarino. *Inhibition et resynchronisation des contrôleurs de dialogue.* In IHM'96, 1996. (pp 22, 77, 90, 176)

[DERY 96c] A. Dery, S. Ducasse, and M. Fornarino. *A new vision of control in the PAC modelization.* Rapport de recherche RR-96-??, I3S, 1996. (pp 22, 179)

[DISS 96] S. Dissoubray. *Une approche synchrone pour l'intégration du contrôle.* PhD thesis, Université de Nice-Sophia Antipolis, 1996. (pp 95, 102)

[DODA 95] M. Dodani, B. K. Gan, L. Velasquez, and X. Yang. Modeling object interactions as first class citizens. Poster Panel at OOPSLA'95, 1995. (pp 15, 20, 29, 30, 40, 228)

[DUBO 91] A. Dubois, M. Fornarino, and A. Pinna. *A tool for modelling and reasoning. Application to the building energy analysis.* In 13th IMACS World Congress on Computation and Applied Mathematics, Dublin, 1991. (p 69)

[DUCA 93] S. Ducasse and M. Fornarino. *Protocol for Managing Dependencies between Objects by controlling Generic Function Invocation.* In OOPSLA'93 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming. ACM, 1993. RR–94–62. (pp 21, 22, 137)

[DUCA 94a] S. Ducasse and M. Fornarino. *Protocole pour la gestion des dépendances entre objets grâce au contrôle des fonctions génériques.* Numeéro 11 in Collection didactique, pages 239–258. Journées Francophones des Langages Applicatifs, INRIA, Février 1994. RR–94–62. (pp 21, 22, 137)

[DUCA 94b] S. Ducasse, M. Fornarino, and A.-M. Pinna. *Embedding behavioral relationships between objects using computational relection.* Rapport de recherche RR-94-60, I3S, 1994. (p 22)

[DUCA 95a] S. Ducasse, M. Blay-Fornarino, and A.-M. Pinna. *A Reflective Model for First Class Dependencies.* In Proceedings of OOPSLA'95, pages 265–280, Austin, Octobre 1995. ACM. RR–95–24. (p 22)

[DUCA 95b] S. Ducasse, M. Fornarino, and A.-M. Pinna-Dery. *Control et PAC model.* Rapport de recherche RR-95-03, I3S, 1995. (p 22)

[DUCA 95c] S. Ducasse. *Inheritance Mechanism Reification by Means of First Class Object.* In Proceedings of the IJCAI'95 workshop on Reflection and Meta-Level Architectures and their Applications in AI, pages 39–49, 1995. RR–95–12. (pp 22, 185)

[DUCA 96a] S. Ducasse. *Reifying Inheritance in a Reflective Language.* 1996. RR–95–28. (pp 22, 186)

[DUCA 96b] S. Ducasse, M. Blay-Fornarino, and A.-M. Pinna. *Object and Dependency Oriented Programming in FLO.* In Proceedings of ISMIS'96, LNCS 1079, pages 295–304. Springer Verlag, Juin 1996. (p 22)

[DUCO 87] R. Ducournau and M. Habib. *On Some Algorithms for Multiple Inheritance in Object-Oriented Programming.* In Proceedings ECOOP'87, LNCS 276, pages 243–252. Springer Verlag, Juin 1987. (p 12)

[DUCO 89] R. Ducournau and M. Habib. *La multiplicité de l'héritage dans les langages à objets.* TSI, vol. 8, no. 1, pages 41–62, Janvier 1989. Numéro Spécial Langages orientés objet. (p 12)

[DUCO 92] R. Ducournau, M. Habib, M. Huchard, and M. L. Mugnier. *Monotonic Conflict Resolution Mechanisms for Inheritance.* In Proceedings OOPSLA'92, ACM SIGPLAN Notices, pages 16–24, Octobre 1992. (pp 185, 195)

[DUCO 94] R. Ducournau, M. Habib, M. Huchard, and M. L. Mugnier. *A Proposal for a monotonic Multiple Inheritance Linearization.* In Proceedings OOPSLA'94, ACM SIGPLAN Notices, pages 164–175, Octobre 1994. (pp 185, 195)

[DUGE 87] P. Dugerdil. *Les mécanismes d'hériatge d'OBJLOG: vertical et sélectif multiple.* In Actes de CARFIA'87, pages 259–273, 1987. (p 84)

[EPST 88] D. Epstein and W. Lalonde. *A Smaltalk Window System Based On Constraints.* In Proceedings of OOPSLA'88, pages 83–94, 1988. (p 213)

[FERB 84] J. Ferber. *Quelques aspects du caractère self réflexif du langage MERING.* Bigre + Globule, vol. 41, no. 41, pages 277–290, 1984. Deuxième journée d'étude du groupe de travail AFCET sur les langages orientées objets. (p 111)

[FERB 88] J. Ferber. *Conceptual reflection and actor languages.* In P. M. North-Holland and D. Nardi, éditeurs, Meta-level Architectures and Reflection, pages 177–193, 1988. (p 111)

[FERB 89] J. Ferber. *Computational Reflection in Class Based Object Oriented Languages.* In N. Meyrowitz, éditeur, Proceedings of OOPSLA'89, pages 317–326. ACM, October 1989. (pp 19, 91, 135, 138, 139, 140, 190)

[FOLE 84] J. Foley and A. V. Dam. Fundamentals of interactive computer graphics. Addison Wesley, 1984. (p 170)

[FOOT 93] B. Foote. *Architectural Balkanization in the Post-Linguistic Area.* In OOPSLA'93 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming, 1993. (p 119)

[FORN 90a] M. Fornarino and A.-M. Pinna. *Un modèle objet logique et relationnel. Le langage Othelo.* PhD thesis, Université de Nice - Sophia Antipolis, INRIA Sophia Antipolis, Avril 1990. (pp 11, 12, 13, 14, 20, 21, 30, 40, 52, 69, 77, 105, 216)

[FORN 90b] M. Fornarino and A. Pinna. *Expression des relations et maintien de la cohérence : le concept de lien.* Rapport de recherche 1346, INRIA, November 1990. (p 69)

[FORN 90c] M. Fornarino and A.-M. Pinna. *Une approche génie logicielle de la représentation des connaissances.* In J. L. G. Gouardires and M. White, éditeurs, Expersys-90, pages 145–151, 1990. (p 69)

[FREE 89] B. Freeman-Benson. *A Module Mechanism for Constraints in Smalltalk.* In Proceedings of OOPSLA'89, pages 389–396, oct 1989. (pp 96, 221)

[FREE 90a] B. Freeman-Benson, J. Maloney, and A. Borning. *An incremental Constraint Solver.* Comminications of the ACM, vol. 33, no. 1, pages 55–63, 1990. (p 213)

[FREE 90b] B. N. Freeman-Benson. *Kaleidoscope: Mixing Objects, Constraints, and Imperative Programming.* In Proceedings of OOPSLA'90, pages 77–88, Ottawa, October 1990. ACM. (pp 213, 221)

[FREE 92] B. Freeman-Benson and A. Borning. *Integrating Constraints with an Object-Oriented Language.* In Proceedings of ECOOP'92, LNCS 615, pages 268–286, Utrecht, June 1992. Springer-Verlag. (p 221)

[FRØL 92] S. Frølund. *Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages.* In Proceedings of ECOOP'92, LNCS 615, pages 185–196. Springer-Verlag, June 1992. (pp 14, 83, 233)

[FRØL 93] S. Frølund and G. Agha. *A Language Framework for Multi-Object Coordination.* In Proceeding of ECOOP'93, LNCS 707, pages 346–360. Springer Verlag, Juillet 1993. (pp 13, 14, 17, 25, 28, 29, 30, 32, 33, 37, 40, 51, 69, 104, 233, 235)

[FRØL 94] S. Frølund. *Constraint-Based Synchronization of Distributed Activities.* PhD thesis, University of Illinois at Urbana-Champaign, 1994. (pp 25, 65, 233)

[GALL 94] E. Gallesio. *STklos: A Scheme Object Oriented System dealing with the Tk Toolkit.* In ICS, éditeur, Xhibition 94, San Jose, CA, pages 63–71, June 1994. (p 109)

[GALL 96] E. Gallésio. *Designing a Meta Protocol to Wrap a Standard Graphical Toolkit.* In Proceedings of ISOTAS'96, LNCS 1049, pages 135–156. JSSST-JAIST, Mars 1996. (pp 30, 109, 111)

[GAMM 94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Elements of reusable object-oriented software. Addison-Wesley, 1994. (pp 159, 160, 163, 166, 207, 228)

[GATZ 92] S. Gatziu and K. R. Dittrich. *SAMOS: an Active Object-Oriented Database System.* Data Engineering, Special issue on active databases, vol. 15, no. 1-4, pages 23–26, 1992. (pp 25, 27, 29, 30, 34, 35, 40)

[GEHA 92] N. Gehani and H. Jagadish. *Active Database Facilities in Ode.* Data Engineering, Special issue on active databases, vol. 15, no. 1-4, pages 19–22, 1992. (pp 25, 27, 34, 40)

[GENS 93] J. Gensel. *Expression et satisfaction de contraintes dans TROPES.* In RPO, pages 51–62, 1993. (pp 25, 28, 29, 40, 213)

[GIUS 89] D. Giuse. *KR: Constraint-based knowledge representation.* Rapport de recherche CMU-CS-89-142, Carnegie Mellon University, Avril 1989. (p 214)

[GIUS 92] D. Giuse. *KR: Constraint-based knowledge representation.* Rapport de recherche, Carnegie Mellon University, November 1992. Kr V2.0. (pp 15, 28, 32, 34, 38, 40, 214)

[GOLD 83] A. Goldberg and D. Robson. Smalltalk-80: The language and its implementation. Addison-Wesley, 1983. (pp 11, 12, 14, 15, 27, 57, 86, 173, 206, 207, 209)

[GOSL 83] J. Gosling. *Algebraic Constraints.* PhD thesis, Carnegie Mellon University, 1983. (p 213)

[GRAU 89] N. Graube. *Metaclass compatibility.* In Proceedings of OOPSLA'89, pages 305–315. ACM, October 1989. (pp 42, 94, 147)

[GREE 85] M. Green. *The University of Alberta User Interface Management System.* In Computer Graphics, volume 19, pages 205–213, Juillet 1985. (p 170)

[GUER 92] R. Guerraoui, R. Capobianchi, A. Lanusse, and P. Roux. *Nesting Actions through Asynchronous Message Passing: the ACS Protocol.* In Proceedings of ECOOP'92, LNCS 615, pages 170–184. Springer-Verlag, Juin 1992. (p 233)

[HAAR 90] V. Haarslev and R. Møller. *A Framework for Visualizing Object-Oriented Systems.* In Proceedings OOPSLA/ECOOP'90, pages 237–244, Octobre 1990. (p 137)

[HALB 87] D. C. Halbert and P. D. O'Brien. *Using Types and Inheritance in Object-Oriented Languages.* In Proceedings ECOOP'87, LNCS 276, pages 20–31. Springer Verlag, Juin 1987. (p 185)

[HALB 91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. *The Synchronous Data Flow Programming Language LUSTRE.* In Proceedings of the IEEE, volume 79, sep 1991. (pp 16, 39, 102)

[HANS 93] E. Hanson and J. Widom. *An overview of production rules in database systems.* Knowledge Engineering Review, vol. 8, no. 2, pages 121–143, 1993. (pp 105, 149, 150)

[HARR 86] D. Harris. *A Hybrid Structured Object and Constraint Representation Language.* In Proceedings of the AAAI'86, pages 986–990, 1986. (pp 28, 40)

[HARR 93] W. Harrison and H. Ossher. *Subject-Oriented Programming (A Critique of Pure Objects).* In OOPSLA'93, pages 411–428, Washington DC, October 1993. (pp 36, 52, 225)

[HELM 90] R. Helm, I. Holland, and D. Gangopadhyay. *Contracts: Specifying Compositions in Object-Oriented Systems.* In Proceedings of OOPSLA'90, pages 169–180, Ottawa, October 1990. ACM. (pp 11, 14, 20, 33, 73, 223, 225)

[HILL 86] R. Hill. *Supporting Concurrency, Communication, and Synchronisation in Human–Computer Interaction–The Sassafras UIMS.* ACM Transactions on Graphics, vol. 5, no. 3, pages 179–210, Juillet 1986. (p 218)

[HILL 87] R. Hill. *Event Reponse Systems – a Technique for Specifying Multithreaded Dialogues.* In Proceedings of CHI+GI, pages 214–248, 1987. (p 218)

[HILL 92] R. D. Hill. *The Abstraction-Link Paradigm: Using Contraints to Connect User Interfaces to Applications.* In Proceedings of CHI'92: the Conference on Human Factors in Computing Systems, pages 335–342. ACM, Mai 1992. (pp 22, 28, 29, 35, 40, 174, 175, 176, 179, 184, 198, 213, 218)

[HILL 93a] R. Hill. *The Rendezvous Constraint Maintenance System.* In Proceedings of UIST'93, pages 225–234, 1993. (pp 38, 45, 99, 100, 218)

[HILL 93b] R. Hill, T. Brinck, J. Patterson, S. Rohall, and W. Wilner. *The Rendezvous Language and Architecture: Tools for Constructibg Multi-User Interactive Systems.* Communications of the ACM, vol. 36, no. 1, pages 62–67, 1993. (pp 45, 174, 218)

[HILL 94] R. D. Hill, T. Brinck, S. L. Rohall, J. F.Patterson, and W. Wilner. *The Rendezvous Architecture and Language for Constructing Multi-User Applications.* ACM Transactions on Computer-Human Interaction, vol. 1, no. 2, pages 81–125, June 1994. (pp 25, 28, 30, 35, 95, 174, 179, 213, 218, 219)

[HOLL 92] I. M. Holland. *Specifying reusable components using Contracts.* In Proceedings of ECOOP'92, LNCS 615, pages 287–308, Utrecht, June 1992. Springer-Verlag. (pp 20, 223, 224)

[HORN 92] B. Horn. *Constraint Patterns As a Basis for Object-Oriented Programming.* In Proceedings OOPSLA '92, pages 218–233, Octobre 1992. (p 213)

[IBRA 88] M. H. Ibrahim and F. A. Cummins. *KSL: A Reflective Object-Oriented Programming Language.* In Proceedings of the International Conference on Co,puter Languages, pages 186–193. IEEE, October 1988. (pp 136, 205)

[IBRA 91] M. H. Ibrahim, W. E. Bejeck, and F. A. Cummins. *Instance specialization without delegation.* Journal of Object-Oriented Programming, vol. 4, no. 3, pages 53–56, June 1991. (pp 135, 136)

[ISHI 91] Y. Ishikawa. *Reflection Facilities and Realistic Programming.* SIGPLAN Notices, vol. 26, no. 8, pages 101–110, August 1991. (p 111)

[JAAS 95] A. Jaaski. *Implementing Interactive Applications in C++.* Software Practice and Experience, vol. 25, no. 3, pages 271–289, Mars 1995. (p 173)

[JAFF 94] J. Jaffar and M. Maher. *Constraint Logic Programming : a survey.* The Journal of Logic Programming, no. 19,20, pages 503–581, 1994. (p 213)

[JAGA 92] S. Jagannathan and G. Agha. *A Reflective Model of Inheritance.* In Proceedings of ECOOP'92, LNCS 615, pages 351–371, Utrecht, June 1992. Springer-Verlag. (p 185)

[JOHN 88] R. Johnson and B. Foote. *Designing reusable classes.* JOOP, vol. 1, no. 2, pages 22–35, jun 1988. (p 11)

[JUNG 93] R. Jungclaus, T. Hartmann, and G. Saake. *Relationships between Dynamic Object.* In Proceedings of the second European-Japanese Seminar, Information Modelling and Knowledge Bases IV : Concepts, Methods and Systems, pages 425–438, Amsterdam, 1993. IOS Press. (pp 12, 26, 27, 63, 230, 231)

[JUNG 96] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. *Troll – A langugae for Object-Oriented Specification of Information Systems.* ACM transactions on Inofrmation Systems, vol. 14, no. 2, pages 175–211, Apr 1996. (pp 11, 12, 13, 14, 20, 26, 27, 199, 230)

[KARS 93] A. Karsenty and M. Beaudouin-Lafon. *An algorithm for distributed groupware Applications.* In IEEE, éditeur, Proceeding of ICDCS'93 Intyernational Conference on Distributed Computing Systems, Mai 1993. (pp 44, 90, 179, 183)

[KEEN 89] S. E. Keene. Object-oriented programming in common-lisp. Addison-Wesley, 1989. (pp 26, 109, 116, 129, 149, 195, 205)

[KICZ 91] G. Kiczales, J. des Rivieres, and D. G. Bobrow. The art of the metaobject protocol. MIT Press, 1991. (pp 94, 109, 111, 112, 115, 116, 135, 137, 185, 195, 210)

[KICZ 92a] G. Kiczales. *Towards a New Model of Abstraction in the Engineering of Software.* In Proc. of IMSA'92 Workshop on Reflection and Meta-Level Architecture, 1992. (pp 113, 115, 135, 152, 153)

[KICZ 92b] G. Kiczales and J. Lamping. *Issues in the Design and Documentation of Class Libraries.* In Proceedings OOPSLA'92, ACM SIGPLAN Notices, pages 435–451, Octobre 1992. (p 115)

[KIM 90] W. Kim. Introduction to object-oriented databases. Mit Press, 1990. (p 26)

[KOKE 93] T. Kokeny. *CSPOO: un système á résolution de contraintes orienté objet.* In Représentation par objets, pages 39–49, 1993. (pp 25, 213)

[KRAS 88] G. E. Krasner and S. T. Pope. *A cookbook for using the Model-View-Controller User interface paradigm in Smalltalk-80.* JOOP, pages 26–49, Août 1988. (pp 15, 27, 165, 173, 207)

[KRIS 93] B. B. Kristensen. *Transverse Activities: Abstractions in Object-Oriented Programming.* In Proceedings of ISOTAS'93, LNCS 742, Lecture Notes in Computer Science, pages 277–296. JSSST-JAIST, Springer-Verlag, November 1993. (p 20)

[KRIS 94] B. B. Kristensen. *Complex Associations: Abstractions in Object-Oriented Modeling.* In ACM, éditeur, Proceedings of OOPSLA'94, volume 29 of *ACM Sigplan Notices*, pages 272–286, Portland, October 1994. ACM. (p 20)

[KRIS 96] B. Kristensen and D. May. *Activities: Abstractions for Collective Behavior.* In Proceedings of ECOOP'96, 1996. (p 20)

[LEDO 96] T. Ledoux and P. Cointe. *Explicit Metaclasses as a Tool for Improving the Design of Class Libraries.* In Proceedings of ISOTAS'96, LNCS 1049, pages 38–55. JSSST-JAIST, mar 1996. (p 186)

[LIEB 86a] H. Lieberman. *Delegation and Inheritance: Two mechanisms for sharing Knowledge in Object-Oriented Systems.* Bigre + Globule, vol. 48, pages 79–89, 1986. (pp 62, 185, 214)

[LIEB 86b] H. Lieberman. *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems.* In Proceedings OOPSLA'86, pages 214–223, November 1986. (p 214)

[LIEB 88] K. Lieberherr. *Object-oriented programmnig with class dictionaries.* Journal on Lisp and Symbolic Computation, vol. 1, no. 2, pages 185–212, 1988. (p 11)

[LIEB 89] K. Lieberherr and I. Holland. *Assuring a Good Style for Object-Oriented Programs.* IEEE Software, pages 38–48, sept 1989. (p 163)

[LINT 89] M. Linton, J. Vlissides, and P. Calder. *Composing User Interfaces with Interviews.* IEEE Computer Magazine, pages 8–22, 1989. (pp 174, 223)

[LIOT 94] J. Liotard. *Contraintes, objets et application: la CAO.* In Langages et Modèle à Objets, pages 181–193, Grenoble, Octobre 1994. IMAG, INRIA et PRC GDR IA. (p 213)

[MAES 87a] P. Maes. *Concepts and Experiments in Computational Reflection.* In Proceedings of OOPSLA'87, pages 147–155. ACM, October 1987. (pp 62, 91, 111, 137)

[MAES 87b] P. Maes. *Computational Reflection.* PhD thesis, Laboratory for Artificial Intelligence, Vrije Universiteit Brussel, Brussels Belgium, Janvier 1987. (pp 112, 135, 137)

[MAES 88] P. Maes. *Issues in Computational Reflection.* In D. N. P. Maes, éditeur, Meta-Level Architectures and Reflection, pages 21–35. Elsevier Science Publishers B.V. (North-Holland), 1988. (pp 62, 116, 137)

[MAGN 94] M. Magnan. *Réutilisation de composants: les exceptions dans les objets composites.* PhD thesis, Université de Montpellier II, 1994. (pp 84, 85, 87, 88)

[MALE 92] J. Malenfant, C. Dony, and P. Cointe. *Behavioral Reflection in a prototype-based language.* In A. Yonezawa and B. Smith, éditeurs, Proceedings of Int'l Workshop on Reflection and Meta-Level Architectures, pages 143–153, Tokyo, November 1992. RISE and IPA(Japan) + ACM SIGPLAN. (pp 138, 196)

[MALE 95] J. Malenfant. *On the Semantic Diversity of Delegation-Based Programming Languages.* In Proceedings of OOPSLA'95, pages 215–230, Austin, Octobre 1995. ACM. (p 62)

[MALO 89] J. Maloney, A. Borning, and B. Freeman-Benson. *Constraint Technology for User-Interface Construction in ThingLabII.* In Proceedings of OOPSLA'89, pages 381–388, oct 1989. (p 220)

[MASU 92] H. Masuhara, S. Matsuoka, T. Wanatabe, and A. Yonezawa. *Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently.* In Proceedings of OOPSLA'92, pages 127–144. ACM, October 1992. (p 111)

[MATS 92] S. Matsuoka, T. Watanabe, and A. Yonezawa. *Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming.* In Proceedings of ECOOP'92, LNCS 615, pages 231–250. Springer-Verlag, June 1992. (p 111)

[MCJA 95] J. McJaffer. *Meta-Level Programming with CodA.* In Proceedings of ECOOP'95, LNCS 952, pages 190–214. Springer-Verlag, Août 1995. (pp 111, 117, 135, 138)

[MEDE 91] C. B. Medeiros and P. Pfeffer. *Object Integrity Using Rules.* In Proceedings of ECOOP'91, LNCS 512, pages 218–230. Springer-Verlag, Juillet 1991. (pp 25, 27, 28, 34, 35, 36, 70)

[MEYE 90a] B. Meyer. Conception et programmation par objets. Intereditions, 1990. (p 11)

[MEYE 90b] B. Meyer. *Tools for a new culture: Lessons from the design of Eiffel libraries.* Communications of the ACM, vol. 33, no. 9, pages 68–88, Septembre 1990. (p 11)

[MINS 87] N. H. Minsky and D. Rozenshtein. *A Law Based Approach to Object-Oriented Programming.* In Proceedings of OOPSLA'87, pages 482–493. ACM, Octobre 1987. (p 89)

[MINS 89] N. H. Minsky and D. Rozenshtein. *Controllable Delegation: An Exercise in Law-Governed Systems.* In Proceedings OOPSLA'89, pages 371–380, Octobre 1989. (p 89)

[MOON 86] D. A. Moon. *Object-Oriented Programming with Flavors.* In Proceedings OOPSLA'86, pages 1–8, November 1986. (pp 26, 205)

[MOUR 94] P. Moura and E. Costa. *Logtalk: Object-Oriented Programming in Prolog.* In Proceedings of the Second Portuguese Conference and Exhibition in Object-Oriented Technology, Lisbon, 1994. 3i Consultores. (p 69)

[MULE 93a] P. Mulet and P. Cointe. *Definition of a reflective kernel for a prototype-based langage.* In Proceedings of ISOTAS'93, LNCS 742, pages 128–144. JSSST-JAIST, Springer-Verlag, November 1993. (pp 62, 111, 112, 138, 185, 195, 196)

[MULE 93b] P. Mulet and P. Cointe. *Définition d'un noyau réflexif pour un langage à prototypes.* In Représentation par objets, pages 101–115, La grande motte, 17-18 June 1993. Ec2. (p 111)

[MULE 95a] P. Mulet. *Réflexion et langage à prototypes.* PhD thesis, École des Mines de Nantes, 1995. (pp 42, 62, 111, 112, 117, 135, 138, 147, 195)

[MULE 95b] P. Mulet, J. Malenfant, and P. Cointe. *Towards a Methodology for Explicit Composition of MetaObjects.* In Proceedings of OOPSLA'95, pages 316–330, Austin, Octobre 1995. (pp 116, 138, 147)

[MURA 87] M. Murata and K. Kusumoto. *Daemon: A Mediator that Keeps Wholes Consistent with their parts.* Rapport de recherche, Fuji Xerox, 1987. (pp 206, 207)

[MURA 89] M. Murata and K. Kusumoto. *Daemon: Another Way of Invoking Methods.* JOOP, vol. 2, no. 2, pages 8–12, Juillet 1989. (pp 84, 86, 135, 206, 207)

[MYER 90] B. Myers, D. Giuse, R. Dannenberg, B. V. Zanden, D. Kosbie, E. Pervin, A. Mickish, and P. Marchal. *Garnet: Comprehensive Support for Graphical Highly-Interactive User Interfaces.* IEEE Computer, vol. 23, no. 11, pages 71–85, 1990. (pp 25, 28, 214)

[MYER 92] B. A. Myers, D. A. Giuse, and B. V. Zanden. *Declarative programming in a prototype-instance system: object-oriented programming without writing methods.* In Proceedings of OOPSLA'92, pages 185–199, Vancouver, October 1992. ACM. (p 214)

[NAHA 95] C. Nahaboo. *Klone Reference Manual.* Inria, 1995. (p 205)

[NANC 92] D. Nanci, B. Espinasse, B. Cohen, and H. Heckenroth. Ingenierie des systemes d'information avec merise. Sybex, 1992. (p 77)

[NATA 91] H. Natatsuyama, M. Murata, and K. Kusumoto. *A new framework for separating user interfaces from application programs.* S.I.G.C.H.I, vol. 23, no. 1, pages 88–92, Janvier 1991. (pp 206, 207)

[NELS 85] G. Nelson. *A constraint-based graphics system.* In Proceedings of SIGGRAPH'85, pages 235–243, 1985. (p 213)

[NEUS 91] C. Neusius. *Synchronizing Action.* In Proceedings of ECOOP'91, LNCS 512, pages 118–132. Springer-Verlag, Juillet 1991. (pp 14, 233)

[NIER 87] O. Nierstrasz. *Active Object in Hybrid.* In Proceedings of OOPSLA'87, pages 243–253, Octobre 1987. (pp 14, 41, 233)

[NIER 91] O. Nierstrasz, D. Tsichritzis, V. de Mey, and M. Stadelmann. *Objects + Scripts = Applications.* In Object Composition. D. Tsichritzis, 1991. Université de Genèves. (p 28)

[NIER 95] O. Nierstrasz and D. Tsichritzis. Object-oriented software composition. Prentice-Hall, 1995. (p 199)

[NIGA 91] L. Nigay and J. Coutaz. *Building user interfaces: organizing software agents.* In ACM, éditeur, Esprit'91 Conference Proceedings, 1991. (p 179)

[NORM 86] D. Norman and S. Draper. User centered system design. Lawrence Erlbaum Ass. Publisher, 1986. (p 169)

[OSSH 92] H. Ossher and W. Harrison. *Combination of Inheritance Hierarchies.* In Proceedings OOPSLA'92, ACM SIGPLAN Notices, pages 25–40, Octobre 1992. (p 225)

[OSSH 95] H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal. *Subject-Oriented Composition Rules.* In Proceedings of OOPSLA'95, pages 235–250, 1995. (p 52)

[OUSS 96] M. Oussalah. Ingénierie objet. Inter-Editions, 1996. (pp 25, 100)

[PACH 93] F. Pachet, F. Wolinski, and S. Giroux. *Spying as an Object-Oriented Programming Paradigm.* In Proceedings of TOOLS EUROPE'93, pages 109–118, 1993. (pp 135, 136)

[PAEP 90] A. Paepcke. *PCLOS: Stress Testing CLOS.* In Proceedings of OOPSLA'90, pages 194–211, Ottawa, Octobre 1990. ACM. (p 111)

[PAEP 93] A. Paepcke. *User-Level Language Crafting.* In Object-Oriented Programming: the CLOS perspective, pages 66–99. MIT Press, 1993. (p 115)

[PASC 86] G. A. Pascoe. *Encapsulators: A New Software Paradigm in Smalltalk-80.* In Proceedings OOPSLA'86, pages 341–346, November 1986. (pp 135, 136)

[PERN 90] B. Pernici. *Class Design and Meta-Design.* In Object Management, pages 133–195. Dennis Tsichritzis, 1990. Centre Universitaire de Genève. (p 225)

[PFAF 85] G. Pfaff and P. Hagan. Seeheim workshop on user interface managament systems. Springer-Verlag, 1985. EuroGraphics Seminars. (p 170)

[PINT 93] X. Pintado. *Gluons: a support for Software Component Cooperation.* In Proceedings of ISOTAS'93, LNCS 742, pages 43–60. JSSST-JAIST, Springer-Verlag, November 1993. (pp 26, 28, 29, 30, 40, 71, 228)

[PUGE 93] J. Puget and P. Albert. *SOLVER: Contraints + Objects ← Descriptions.* In Workshop of IJCAI'93, 1993. (pp 25, 28, 40, 213)

[PUGE 94] J. Puget. *A C++ implementation of CLP.* In Proceedings of SPICIS'94, 1994. (p 213)

[RAO 91] R. Rao. *Implementational Reflection in Silica.* In Proceedings of ECOOP'91, LNCS 512, pages 251–267. Springer-Verlag, Juillet 1991. (p 111)

[RISC 92] T. Risch and M. Skold. *Active Rules based on Object-Oriented Queries.* Data Engineering, Special issue on active databases, vol. 15, no. 1-4, pages 27–30, 1992. (p 25)

[RIVA 95] F. Rivard. *Extension du compilateur Smalltalk: application à la paramétrisation de l'envoi de message.* In JFLA95, 1995. (pp 136, 198)

[RIVA 96a] F. Rivard. *Smalltalk et Réflexivité.* PhD thesis, Ecole des Mines de Nantes, 1996. (pp 109, 112, 135, 136)

[RIVA 96b] F. Rivard. *Reflective Facilities in Smalltalk.* Février 1996. (pp 111, 136)

[RIVA 96c] F. Rivard. *Pour un lien d'instanciation dynamique dans les langages à classes.* In JFLA96. INRIA - collection didactique, Janvier 1996. (p 136)

[RUMB 87] J. Rumbaugh. *Relations as Semantic Constructs in an Object-Oriented Language.* In Proceedings OOPSLA'87, pages 466–481, Decembre 1987. (pp 11, 12, 14, 15, 25, 32, 35, 208)

[SALB 94] D. Salber, L. Nigay, and J. Coutaz. *Extending the Scope of PAC-Amodeus to Cooperative Systems.* In Workshop of Conference on Software Architectures for Cooperative Systems. ACM'94, Octobre 1994. (p 179)

[SANN 83] M. Sannella. *The Interlisp-D Reference Manual.* Xerox Parc, Palo Alto, 1983. (p 205)

[SANN 93] M. Sannella. *The skyblue constraint solver.* Rapport de recherche 92-07-01, Department of Computer Science and Engineering, University of Washington, Février 1993. (pp 38, 100, 221)

[SHAH 89] A. Shah, J. Rumbaugh, J. Hamel, and R. Borsari. *DSM: An Object-Relationship Modeling Language.* In Proceedings of OOPSLA'89, pages 191–202, Octobre 1989. (pp 11, 20, 32)

[SHAW 96] M. Shaw and D. Garlan. Software architecture: Perspectives on an emerging discipline. Prentice-Hall, 1996. (p 199)

[SMIT 84] B. Smith. *Reflection and Semantics in Lisp.* In Proceedings of POPL'84, pages 23–3, 1984. (p 111)

[SMIT 95] W. Smith. *Using a Protype-based Language for User Interface : The Newton Project's Experience.* In Proceedings of OOPSLA'95, pages 61–73. ACM, Octobre 1995. (p 195)

[SNYD 86] A. Snyder. *Encapsulation and Inheritance in Object Programming Languages.* In Proceedings of OOPSLA'86, pages 38–45. ACM, Septembre 1986. (p 185)

[SOUK 95] J. Soukup. *Implementing Patterns.* In Addison-Wesley, éditeur, Patterns Languages of Program Design, pages 395–412. Coplein and Schmidt, 1995. (p 160)

[STEE 80] G. Steele. *The definition and implementation of a comuter programming language based on constraints.* PhD thesis, MIT, 1980. (p 213)

[STEE 88] L. Steels. *Meaning in knowledge representation.* In P. M. North-Holland and D. Nardi, éditeurs, Meta-level Architectures and Reflection, pages 51–59, 1988. (p 137)

[STEE 90] G. Steele. Common lisp the language, second edition. Digital Press, 1990. (pp 11, 26, 32, 109, 116, 149, 205, 214)

[STEF 86a] M. Stefik and D. Bobrow. *Object-Oriented Programming: Themes and Variations.* The AI Magazine, vol. 6, no. 4, pages 40–62, winter 1986. (pp 14, 57, 205)

[STEF 86b] M. Stefik, D. Bobrow, and K. Kahn. *Integrating Access-Oriented Programming into a Multiparadigm Environment.* IEEE Software (USA), vol. 3, no. 1, pages 10–18, Janvier 1986. (pp 26, 32, 34, 40)

[STEI 87] L. A. Stein. *Delegation is Inheritance.* In Proceedings of OOPSLA'87, pages 138–146, October 1987. (p 185)

[STEI 89] L. A. Stein, H. Lieberman, and D. Ungar. *A Shared View of Sharing: The Treaty of Orlando.* In Object-Oriented Concepts, DataBases, and Applications, pages 31–48. ACM Press, Addison-Wesley, 1989. (p 214)

[STRO 86] B. Stroustrup. The c++ programming language. Addison-Wesley, 1986. (p 11)

[STRO 95] R. Stroud and Z. Wu. *Using Metaobject Protocols to Implement Atomic Data Types.* In W.Olthoff, éditeur, Proceedings of ECOOP'95, LNCS 952, pages 168–189. Springer-Verlag, Août 1995. (p 111)

[SULL 92] K. Sullivan and D. Notkin. *Reconciling Environment Integration and Software Evolution.* Transactions on Software Engineering and Methodology, vol. 1, no. 3, pages 228–268, Juillet 1992. (pp 28, 32, 225)

[SUSS 80] G. Sussman and G. Steele. *CONSTRAINTS : a language for expressing almost-hierarchical descriptions.* Artificial Intelligence, vol. 14, no. 1, pages 1–39, 1980. (p 213)

[SZEK 88] P. Szekely and B. Myers. *A User Interface Toolkit Based On Graphical Objects and Contraints.* In Proceedings of OOPSLA'88, pages 36–4. ACM, Septembre 1988. (p 214)

[TANZ 95] C. Tanzer. *Remarks on object-oriented modeling of associations.* JOOP, pages 43–46, Février 1995. (p 12)

[TEN 90] P. ten Hagen. *Critique of the Seeheim model.* In Proceedings of the workshop :User Interface Management and Design, pages 3–6. Springer-Verlag, 1990. (pp 170, 171)

[TROM 97] G. Trombettoni. *Maintien de cohérence des systémes de contraints fonctionnels par propagation locale.* PhD thesis, INRIA, 1997. Titre provisoire. (pp 99, 101)

[UNGA 87] D. Ungar and R. B. Smith. *Self: The Power of Simplicity.* Published as SIGPLAN Notices, vol. 22, no. 12, pages 227–242, 1987. (pp 62, 195, 214)

[VAHD 92] A. Vahdat. *The design of a Meta-Object Protocol controlling the behavior of a scheme interpreter.* Rapport de recherche, Xerox Parc, 1992. (p 114)

[VAN 89] J. van den Bos and C. Laffra. *PROCOL – A Parallel Object Language with Protocols*. In Proceedings OOPSLA'89, pages 95–102, Octobre 1989. (p 235)

[VAN 91] J. van den Bos and C. Laffra. *PROCOL : a Concurrent Object-Oriented Langugae with Protocols delegation and constraints*. Acta Informatica, vol. 28, pages 511–538, 1991. (pp 14, 25, 28, 30, 40, 70, 104, 211, 235, 236)

[VAUT 96] S. Vauttier, M. Magnan, and C. Oussalah. *COBRAS : An Active Model for the Management of the Behavior of Composite Objects*. Rapport de recherche, EERIE, 1996. N:970307. (pp 26, 32, 34, 84, 87, 88)

[WATA 88] T. Watanabe and A. Yonezawa. *Reflection in an Object-Oriented Concurrent Language*. In Proceedings of OOPSLA'88, pages 306–315. ACM Sigplan Notices, Septembre 1988. (p 111)

[WEGN 87] P. Wegner. *Dimensions of Object-Based Language Design*. In Proceedings OOPSLA'87, pages 168–182, Decembre 1987. (p 185)

[WEIN 88] A. Weinand, E. Gamma, and R. Marty. *ET++ - An Object-Oriented Application Framework in C++*. In Proceedings of OOPSLA'88, pages 46–57. ACM, Septembre 1988. (p 174)

[WHOR 56] B. Whorf. Language thought and reality. MIT Press, 1956. (p 197)

[WIDO 96] J. Widow and S. C. (editors). Active database systems : Triggers and rules for advanced database processing. Morgan Kaufman Publishers, 1996. (pp 25, 26, 105, 128, 149, 150)

[WILK 91] M. R. Wilk. *Equate : An Object-Oriented Constraint Solver*. In Proceedings of OOPSLA'91, pages 286–298. ACM, 1991. (pp 214, 220, 221)

[WIRF 90] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. Designing object-oriented software. Prentice Hall, 1990. (pp 12, 14)

[WOLI 91] F. Wolinski and J.-F. Perrot. *Représentation of Complex Objects : Multiple Facets with Part-Whole Hierarchies*. In Proceedings of ECOOP'91, LNCS 512, pages 288–306, Juillet 1991. (p 84)

[YELL 94] D. M. Yellin and R. E. Strom. *Interfaces, Protocols, and the Semi-Automatic Construction of Software Adaptors*. In Proceedings of OOPSLA'94, pages 176–190. ACM, Octobre 1994. (pp 26, 28, 29, 40, 71, 162, 228, 236)

[ZAND 91] B. V. Zanden, B. Myers, D. Giuse, and P. Szeleky. *The importance of Pointer Variables in Constraint Models*. In Proceedings of UIST'91, pages 155–164, 1991. (p 215)

[Pae 93] A. Paepcke, éditeur. Object Oriented Programming : the CLOS perspective. MIT Press, 1993. (pp 94, 109, 112, 115, 116, 149, 195)

[Par 94] Parc Place. *VisualWorks 2.0*, 1994. User Guide, Cookbook, Reference Manual. (pp 27, 32, 174, 209)

[R4R 90] *R4RS. Revised Report on the Algorithmic Language Scheme*. ACM Sigplan Notices, vol. 21, no. 12, Decembre 1990. (p 115)

Résumé

Dans un système à base d'objets, des groupes d'objets coopèrent dans le but d'assurer différentes tâches ou de maintenir des invariants entre eux. Nous appelons de tels comportements collectifs des *dépendances* entre objets. Or les langages à classes traditionnels n'offrent pas la possibilité d'exprimer facilement de telles dépendances. En réponse à ce problème, nous proposons un modèle objet extensible qui intègre l'expression et le maintien de dépendances entre objets. Il propose : une réification du concept de dépendance, une expression locale du maintien de cohérence, des propriétés liées aux dépendances et la résolution des problèmes liés à une expression locale des dépendances par un maintien global en terme de contrôleurs de dépendances. Le langage FLO est une implémentation de ce modèle dans un langage à classes. Dans FLO, les dépendances ont un statut égal à celui des classes (abstraction de la sémantique d'une dépendance, définition incrémentale et indépendance vis-à-vis des classes). FLO utilise les caractéristiques réflexives de langages tels que CLOS, comme le contrôle de l'envoi de message, afin d'assurer le maintien de la cohérence des dépendances.

Dans le but d'offrir un langage conforme au modèle proposé, nous avons choisi de définir un langage adaptable dans lequel peuvent être projetées des dépendances de différentes sortes. L'adaptabilité d'un langage pose deux questions essentielles : d'une part quelles sont les fonctions du langage qui peuvent être sujettes à une adaptation sans mettre en péril sa cohérence ? D'autre part, quelles sont les fonctions du langage qu'il faut rendre adaptables afin de générer toutes les fonctionnalités souhaitées ? Une part importante de notre travail peut être perçue comme l'élaboration d'un *«jeu de Lego pour la gestion des dépendances»* permet au programmeur de se construire son propre environnement de travail.

Mot-clés : Langages Orientés Objets, Dépendances, Méta-Programmation, Protocoles Méta-Objets, Interfaces Hommes-Machines.

Summary

In object-based systems, groups of objects often cooperate to perform some task or maintain some invariants. We called such behaviors *inter object dependencies*. However, most of the class-based languages do not take into account the ability to express such inter object dependencies. In this thesis, to resolve this problem, we propose a new object model that integrates specification and consistency management of inter object dependencies. This model is global enough to manage associations, contraints and, more generally, reactive inter object dependencies. It proposes : the reification of the dependency concept, the local expression of management consistency, the dependency properties, and the global resolution of consistency problems by means of controllers. The FLO langage is an implementation of this model. In FLO, inter object dependencies have the same status as classes (abstraction of the inter object dependency semantics, incremental définition, independence from classes). FLO uses the reflective features of languages such as CLOS, i.e. message passing control, to ensure the consistency of dependencies. In order to provide a language close to the proposed model, we chose to define an adaptable language in which different kinds of inter objet dependencies may be speficied. However, providing an adaptable language leads to two questions : on the one hand, which parts of the langage can be made adaptable without endangering the global consistency of the langage? On the other hand, which parts have to be made adaptable to provide all of the desired fonctionalities? Thus, an important part of our work can be seen as the elaboration of *"Lego game for inter object dependencies"*, with which the programmer may construct his own programming environment.

Keywords : Object Oriented Languages, Dependencies, Meta-Programming, Metaobject