Composition non modulaire modulaire

Rémi Douence

UNIVERSITÉ DE NANTES
FACULTÉ DES SCIENCES ET DES TECHNIQUES

**ÉCOLE DOCTORALE SCIENCES ET TECHNOLOGIES
DE L'INFORMATION ET DE MATHÉMATIQUES**

Année : 2016

N° attribué par la bibliothèque

Composition non modulaire modulaire

HABILITATION À DIRIGER DES RECHERCHES

Discipline : Informatique

Spécialité : Informatique

Présentée
et soutenue publiquement par

Rémi Douence

Le 20 01 2015
à Mines Nantes

Devant le jury ci-dessous :

Président	:	Christian Attiogbe, Professeur	Université de Nantes
Rapporteurs	:	Christophe Dony, Professeur	Université de Montpellier 2
		Jean-Louis Giavitto, Directeur de Recherche	CNRS Ircam
		Lionel Seinturier, Professeur	Université de Lille 1
Examinateurs	:	Théo D'Hondt, Professeur	Vrije Universiteit Brussel
		Jean-Marc Jezequel, Professeur	Université de Rennes 1

Équipe d'accueil	:	Ascola EMN/INRIA, LINA
Laboratoire d'accueil	:	Laboratoire d'Informatique de Nantes

ED : .....

Composition non modulaire modulaire

Modular non-modular composition.

Rémi Douence

Université de Nantes

Remerciements

Je remercie Inria qui en m'accordant un détachement m'a permis d'entamer la rédaction de ce document. Je remercie Mines Nantes qui m'ont incité à le finir.

Je remercie aussi et surtout des humains.

Je remercie mon jury.

Je remercie mes coauteurs avec qui j'ai pris du plaisir à travailler.

Je remercie mes potentiels coauteurs passés.

Je remercie mon relecteur attentif pour ses remarques et mon chef d'équipe pour ses conseils.

Je remercie mes testeurs de soutenances.

Je remercie les membres des équipes durant mes séjours dans Solidor, Lande, Able, Compose, Objet, Obasco, Ascola, Popart et Ascola. Ainsi que ceux de l'équipe Contrainte et Tasc de l'EMN et Pleiad de l'université du Chili. Pour le travail, mais pas que.

Je remercie mes étudiants de thèse, et les coencadrants/directeurs.

Je remercie mes enseignants.

Je remercie mon encadrant de thèse grace à qui tout a pu commencer.

Enfin je remercie mes proches qui m'ont accompagné dans l'autre partie du voyage.

Table des matières

Table des figures

Liste des tableaux

Listings

Chapitre 1

Introduction

Sommaire

DEPUIS ma thèse j'ai fait un certain nombre de travaux de recherche. Comment les décrire ? Comment structurer ce document ?

J'ai envisagé un temps la rédaction par aspect de ce document : un unique énorme chapitre "à plat", une simple succession de petites sections et plusieurs tables des matières offrant différentes structures du document, différents points de vue, différents itinéraires de lecture. Et puis, j'ai choisi, comme la plupart des gens, de réécrire l'histoire, de lui donner autant que faire se peut une forme synthétique, de factoriser, d'éviter les références en avant, les répétitions.

1.1 Contributions

Ce document balaye l'essentiel de mes travaux de recherches réalisés après ma thèse. Ces travaux ont été validés par différentes publications dans des conférences et journaux avec comité de lecture. J'ai choisi d'être exhaustif ou presque dans ce document. On retrouvera donc l'essentiel de mes travaux de recherche réalisés une fois en poste aux Mines de Nantes. J'ai du toutefois ajouter un chapitre un peu fourre tout (le chapitre 9) pour évoquer différents travaux annexes.

1.2 Organisation du document

Le document est structuré en suivant le fil rouge de la programmation non modulaire modulaire. Il comporte deux parties.

La première partie décrit le contexte de mes recherches. Le chapitre 2 introduit le problème de la modularité dans la conception des logiciels. Le chapitre 3 introduit trois travaux de recherches fait par d'autres que moi, qui forment les briques de base de mes différents travaux décrits dans ce document.

La seconde partie du document décrit cinq grandes catégories de travaux de recherche auxquels j'ai participé et qui concernent tous la programmation non modulaire modulaire, que j'appellerai aussi la programmation par aspects. C'est à dire la proposition de supports langage pour programmer de façon unitaire des fonctionnalités dont le code (ou son flot de contrôle) sera éparpillé dans l'application finale. Je commence par des travaux qui voient la programmation par aspects comme basée sur un moniteur d'exécution. Le chapitre 4 concerne le tissage du flot

dynamique de contrôle dans les programmes séquentiels. Le chapitre 5 concerne le tissage du flot dynamique de contrôle dans les programmes concurrents. Le chapitre 6 concerne le tissage du flot dynamique de contrôle dans les programmes distribués. Le chapitre 7 concerne le tissage de flot dynamique de données.

Le chapitre 8 voit la programmation par aspects comme basée sur la transformation statique de programmes.

Le chapitre 9 passe en revue mes autres travaux de recherche non détaillés dans ce document. Il en profite aussi pour aborder mon activité en terme de prototypes. Et il revisite un éditorial (c.a.d. position paper) que j'ai rédigé seul il y a presque dix ans et qui définissait les frontières de la programmation non modulaire modulaire.

Le chapitre 10 fait un bilan de toutes ces activités et esquisse un programme de recherche.

Enfin, un dernier chapitre (chapitre 11) donne mon curriculum vitae élargi et liste mes publications. Par soucis de clarté dans tout le document mes travaux sont référencés numériquement (par exemple [46]), alors que les travaux extérieurs sont référencés alphabétiquement (par exemple [Hoa85]).

Première partie

Contexte

Chapitre 2

La programmation modulaire et ses limites

Sommaire

DEPUIS que j'ai découvert l'informatique (en achetant un ZX81 quand j'étais en sixième) j'ai toujours été passionné par les langages de programmation. Je suis très vite allé voir au delà du langage Basic qui était fourni avec ma machine, que ce soit en deçà avec l'assembleur ou au delà avec Forth. Tout au long de mes études j'ai exploré l'espace des langages de programmation en découvrant différent langages dans mon cursus (par exemple Pascal, C, Lisp, Prolog, Gamma) ou en allant de par moi même chercher ailleurs (par exemple Caml, LML, C++, Eiffel). Au cours de ces années j'ai découvert de nombreux langages, convaincu que certains langages étaient mieux que d'autres et même qu'un langage les dépasserait tous. Ma quête a été à la fois fructueuse (j'ai rencontré Haskell), à la fois infructueuse (j'ai compris que différents problèmes appelaient différents langages) et à la fois désenchantante (j'ai réalisé qu'un langage permettait d'exprimer une structure alors que certains problèmes ont plusieurs structures). Ce chapitre dresse une petite histoire très subjective de la modularité dans les langages de programmation et montre que cette notion atteint ses limites.

2.1 Une certaine notion de modularité

Dans ce chapitre je rapproche la notion de module et celle de structure. Programmation structurée et modulaire : "même combat". Dans un cas comme dans l'autre le but est de penser et réaliser des entités en isolation que l'on pourra par la suite composer pour obtenir une application complexe. Et ce, que l'approche soit ascendante (bottom-up) ou descendante (top-down).

Je passe donc en revue des mécanismes de la programmation structurée comme support de la programmation modulaire.

J'ai été élevé à la programmation structurée et modulaire (Pascal [W71] et Scheme [SS75]). Après un détour par la programmation logique (Prolog [CR93]), je me suis rapidement intéressé à la programmation fonctionnelle pure et paresseuse (LML [A84]). J'ai été fasciné par la facilité de composition soutenue par la pureté (c.a.d. l'absence d'effets de bord), l'ordre supérieur et la paresse. Pourtant, mes travaux sur plusieurs grosses applications (maintenance d'un logiciel de suivi d'affaires écrit dans un progiciel pendant un job d'été pour GDF, placement d'antennes en fonction du relief géographique programmé en C++ pendant mon service militaire, et évaluateur partiel de C programmé en SML pendant mon année d'ingénieur expert à l'Inria) m'ont montré que dès qu'un système grossit les architectures logicielles atteignent leurs limites en terme de modularité. Il n'existe pas une architecture permettant de programmer modulairement toutes les fonctionnalités de l'application. La seule architecture permettant cela serait le canevas qui consiste à truffer de trous une ossature à remplir de code et qui rend peu prévisibles et contrôlables les interactions entre ces bouchons de code.

2.2 Petite histoire subjective de la modularité

Les langages de programmation permettent de manipuler des entités primitives (fonctions, constantes, variables, instructions, ...). Ils permettent de les composer par exemple fonctionnellement pour construire des expressions complexes ou séquentiellement pour définir des blocs. (Pour un joli balayage de ces concepts dans un cadre typé voir le livre de David Schmidt [S84].) Des instructions de rupture de contrôle (par exemple `goto`) permettent de changer (possiblement conditionnellement) le flot de contrôle pour ignorer certaines instructions ou au contraire les exécuter plusieurs fois. Ces ruptures de flots de contrôle peuvent être rendues régulières avec des instructions de boucle (`for` et `while`) ou de récursion (`map` et `fix`). Il est possible de définir un flot de contrôle parenthèsé avec des paires d'instructions `call/return` qui gèrent une pile d'appels afin de revenir à l'appelant le plus récent. Puisque un morceau de code peut être exécuté plusieurs fois, des mécanismes de paramètrage (passage d'arguments, possiblement d'ordre supérieur) ont été proposés. Des opérateurs (d'héritage [DN66], de mixin [BC90]) permettent de construire un programme incrémentalement par ajout (mais aussi masquage) de calculs. Enfin la notion d'abstraction permet de cloisonner des calculs afin d'en exposer une interface limitée (types abstraits [LZ74], contrats [H69], modules et autre foncteurs [AZ02]).

Tous ces mécanismes participent à la programmation modulaire : la définition locale d'entités logicielles qui peuvent ensuite être assemblées (sans les modifier) pour construire une application complexe.

2.3 Les limites

Comme on l'a vu, différents moyens existent pour exprimer une décomposition modulaire.

Dans son livre "mythique" [Bro95] Fred Brooks insiste sur la nécessité de bien choisir la décomposition primaire car elle perdurera dans la suite du processus de développement. Une décomposition bien choisie facilitera la réalisation des modules alors qu'une mauvaise décomposition augmentera les échanges entre modules et complexifiera le logiciel.

La décomposition modulaire a de nombreux avantages : division (du travail) pour régner, raisonnement local, adaptabilité (par exemple en remplaçant un module par un autre), dans une certaine mesure réutilisation, ... Cependant l'approche modulaire a aussi des limites.

Il existe en général plusieurs manières de structurer, de décomposer un problème. Une décomposition modulaire favorise une décomposition parmi plusieurs. C'est justement cette décomposition qui rend par essence d'autres préoccupations non modulaires *vis à vis* de la décomposition primaire. En effet, cela n'a pas de sens de dire qu'une préoccupation est intrinsèquement modulaire ou non modulaire. Je répète ce point essentiel et pourtant souvent ignoré : cela n'a pas de sens de dire qu'une préoccupation est intrinsèquement modulaire ou non modulaire ! Elle l'est (ou ne l'est pas) par rapport à une décomposition conséquence de l'expression d'une autre préoccupation. Cette limitation est connue sous le nom de la tyrannie de la décomposition primaire [bCg]. Dit autrement, il faut choisir une décomposition mais si il peut exister de mauvaises décompositions pour une préoccupation donnée, il n'existe pas en général une décomposition adaptée à toutes les préoccupations d'un problème complexe. On choisira donc une décomposition qui permet d'exprimer de façon modulaire la première préoccupation (ou un maximum de préoccupations) programmée(s). Les autres devront tout de même être exprimées dans ce cadre et en conséquence de façon non modulaire. On pourrait arguer que les canevas (par exemple les conteneurs EJB) ne favorisent pas une préoccupation particulière mais offrent des trous à compléter par du code correspondant à différentes préoccupations. Ce à quoi je répondrais que ce type de gruyère logiciel qui anticipe un maximum de préoccupations a perdu toute structure et se rapproche plus de la méta-programmation, voire de la programmation réflexive.

Ce problème de la tyrannie de la décomposition primaire a été identifié depuis longtemps et dans différents domaines de l'informatique. Par exemple, les bases de données relationnelles permettent au concepteur de choisir une décomposition modulaire de l'information sous forme de relations. On favorise des relations en forme normale qui garantissent de bonnes propriétés que je résumerai abusivement à la cohérence et non redondance des informations. Ces propriétés facilitent l'interrogation et l'entretien de la base. Mais le choix d'une décomposition modulaire particulière en exclus d'autres et peut rendre complexes certaines requêtes qui militeraient pour une autre décomposition. Afin d'attaquer ce problème, la notion de vue [CGT75] a été très tôt introduite dans les bases de données. Une vue peut être considérée comme une relation mais cette relation ou table virtuelle n'est pas mémorisée (en fait pour des raisons d'efficacité elle peut l'être) mais (re)calculée à partir des autres (vraies) relations.

Ce problème est récurrent dans d'autres domaines de l'informatique.

Par exemple, dans les langages fonctionnels le filtrage par motif (pattern matching [FGP64]) est souvent utilisé pour sa concision et sa lisibilité. D'un autre côté le filtrage par motif met à jour un choix de représentation des données, par exemple une liste est-elle récursive par la gauche, par la droite, ou par une position arbitraire. Wadler propose la notion de vues [Wad87] qui permet aux programmeurs de spécifier plusieurs décompositions et les liens entre ces décompositions. Il peut ensuite utiliser dans ses motifs la décomposition la mieux adaptée à son algorithme. Le compilateur est responsable en dernière instance de réaliser une seule décomposition et de traduire tous les motifs utilisés dans cette décomposition (avec un surcoût de complexité si nécessaire).

Dans le domaine des spécifications formelles le même constat a été fait. Par exemple, Jackson montre comment faire cohabiter plusieurs spécifications d'un éditeur de texte [Jac95]. Une représentation du texte et de la position courante en deux piles de caractères facilite l'expression des fonctions de déplacement horizontaux. Une représentation du texte par une imbrication de listes (c.a.d. un mot est une liste de caractères, une ligne est une liste de mots) facilite l'expression de déplacement verticaux. Dans ce cas, les deux moitiés de spécification peuvent cohabiter assez facilement en exprimant les invariants qui relient les deux représentations possibles du texte.

Enfin, des données arborescentes peuvent avoir plusieurs représentations (c.a.d. structuration, décomposition). Un langage dédié à la transformation de ces données d'une représentation

à une autre a été proposé [FGM07]. Ce langage a la particularité d'être inversible. Il est ainsi possible de convertir les données dans une représentation adaptée à un traitement, de les modifier, puis automatiquement de les reconvertir dans la représentation d'origine.

2.4 Programmation

Je reprend ici différents point évoqués précédemment et les illustre à l'aide d'exemple de code et de ma vision de la chaîne de développement. En effet, il est important de ne pas perdre de vue que les programmes sont le produit final que l'on cherche à construire. D'un autre coté les programmes sont souvent de grandes tailles et je me limite ici bien naturellement à des exemples jouets. Le lecteur ne devra pas céder à la facilité consistant à imaginer un réusinage : dans les applications patrimoniales le changement d'architecture est loin d'être toujours possible et peut tout simplement déplacer les problèmes. Dans cette section, j'utilise le paradigme de la programmation à objets en général, et de Java en particulier. Néanmoins, le discours peut être transposé à d'autres paradigmes (par exemple fonctionnel, concurrent, etc.).

Attention, il s'agit bien d'analogie, je ne remet pas ici en cause certaines caractéristiques des langages à objets, mais je dis que dans un programme de grande taille des concepts (représentés ici par les attributs de la classe) pourraient être représentés par des entités logicielles découplées (modulaire). L'ajout ou la modification de fonctionnalités peut en général remettre en question cette modularité.

2.4.1 Programmation modulaire

La programmation modulaire consiste à définir des entités logicielles en isolation puis à les composer. En définissant ces entités, des choix sont faits et ces choix ont des conséquences.

Par exemple, considérons un point dans le plan avec des coordonnées :

```
1  class Point {
2    double x,y;
3    Point(double x, double y) {
4      this.x = x; this.y = y;
5    }
6    void translaterX(double dx) {
7      x += dx;
8    }
9    void translaterY(double dy) {
10     y += dy;
11   }
12 }
```

On a choisi de représenter un point par ses deux coordonnées cartésiennes et des opérations de translation. On voit que chaque fonctionnalité est bien modulaire (c'est à dire indépendante l'une de l'autre). Ici la méthode translaterX affecte uniquement l'attribut x et n'utilise pas y.

Dans l'esprit de la méthode de conception Agile [aM01] il est naturel de développer un logiciel incrémentalement. Dans le cadre de la programmation à objets la définition de sous classe est un mécanisme qui soutient cette approche : il permet d'ajouter des attributs et des méthodes sans modifier la super classe. Par exemple, un point peut être doté d'une couleur de façon orthogonale au code pré-existant.

```
class PointCouleur extends Point {
  Color c;
  PointColor(double x, double y, Color c) {
    super(x,y);
    this.c = c;
  }
  void eclaircir() {
    c = c.brighter();
  }
}
```

On peut aussi contraindre un attribut existant en sous classant Point pour représenter l'axe des ordonnées comme suit :

```
class PointAxeY extends Point {
  PointAxeY(double y) {
    super(0,y);
  }
  void translaterX(double dx) {
    // court circuite la modification de x
  }
}
```

Dans un cas (couleur) comme dans l'autre (axe des ordonnées) les méthodes définies impactent seulement une partie de la classe : un nouvel attribut c dans le cas de la couleur, et la coordonnée x dans l'autre cas. Les incréments de programmation sont donc locaux (ne concernent pas plusieurs attributs de la classe).

2.4.2 Programmation non modulaire

Nous avons vu qu'un point est représenté par ses coordonnées cartésiennes. Ce choix n'est pas sans conséquence. Par exemple, l'ajout d'une méthode de rotation nécessite d'accéder et de modifier la valeur des deux attributs.

```
class PointRot extends Point {
  PointRot(double x, double y) {
    super(x,y);
  }
  void tourner90() {
    double tmp = x; x = y; y = -tmp;
  }
}
```

La fonctionnalité de rotation pourrait être plus modulaire. Par exemple, dans une représentation polaire des coordonnées, elle se résume à une simple addition de l'angle. En contrepartie, la translation selon les abscisses n'est plus modulaire et doit modifier les deux attributs (angle et distance).

```
class PointPol {
  double r,t;
```

```
3     Point(double r, double t) {
4       this.r = r; this.t = t;
5     }
6     void tourner90() {
7       this.t = this.t + Math.PI/2;
8     }
9     void translaterX(double dx) {
10      double x = this.r*Math.cos(this.t);
11      double y = this.r*Math.sin(this.t);
12      this.r = Math.sqrt(x^2+y^2);
13      this.t = Math.atan(y,x);
14    }
15  }
```

Ceci illustre le problème de la tyrannie de la décomposition primaire qui fait qu'un choix a des conséquence et qu'en général il n'existe pas de choix idéal qui rendrait toutes les fonctionnalités modulaires. Ce problème a été nommé par Wadler comme le *expression problem* [Wad98]. Dans son cas, il consiste à définir un type de données pour représenter des expressions et des fonctions associées (par exemple pour calculer la valeur de l'expression, l'afficher, la simplifier, etc.). Le choix peut être fait de structurer le programme autour du type de données (ce qui facilite l'ajout de nouvelles expressions) ou autour des fonctionnalités (ce qui facilite l'ajout de nouvelles fonctionnalités). En général d'autres choix peuvent apparaitre : par exemple, la représentation explicite (ou non) du flot de contrôle/données, ou encore celles des communications distribuées (et de leur causalité).

2.4.3 Programmation non modulaire modulaire

La programmation non modulaire modulaire consiste à fournir un support (langages, compilateurs, analyseurs, etc.) de programmation permettant de programmer de manière modulaire des fonctionnalités qui ne seraient pas programmables de manière modulaire avec les langages et outils existants.

Ces supports peuvent faire cohabiter plusieurs représentations, ou bien en choisir une de base puis exprimer les autres en fonction de la première. Le défi est d'avoir accès aux bonnes abstractions permettant d'exprimer les liens entres plusieurs vues d'un même système. Dans l'idéal ces abstractions sont au bon niveau (applicatif plutôt que langage), déclaratives et robustes (elles résistent à des modifications plus ou moins cosmétiques du programme de base [OMB05]). Mais aussi de fournir opérateurs de composition pour faire cohabiter ces vues et des langages d'actions (plus ou moins limités) qui décrivent les nouveaux comportements introduits (sans pour autant bouleverser le programme de base initial). L'objectif reste de concevoir et comprendre un programme en considérant un module à la fois (c'est à dire sans aller consulter un éventuel code final produit par des outils). Le prochain chapitre présente, entre autres choses, AspectJ un des tout premier langages soutenant la programmation non modulaire. Le reste du document présente mes travaux dans cette même veine.

2.5 Conclusion

La décomposition modulaire permet de réaliser des logiciels complexes. Une décomposition adaptée à des préoccupations simplifie leur expression. En contrepartie une décomposition peut complexifier l'expression d'autres préoccupations. Il n'existe pas en général de décomposition où

toutes les préoccupations s'expriment simplement. Ce problème est connu depuis longtemps comme la tyrannie de la décomposition primaire. Ce problème se retrouve dans différents domaines de l'informatique.

Ce problème est le thème central de mes recherches décrites dans la seconde partie de ce document.

Chapitre 3

Les bases

Sommaire

Après ma thèse, mes travaux de recherche m'ont ouvert à différents univers qui constituent les briques de base de beaucoup de mes travaux de recherche réalisés depuis. Je passe ici en revue, trois de ces travaux initiaux et les concepts auxquels ils m'ont initié.

3.1 Processus Séquentiels Communicants (CSP)

Pendant mon post-doc à CMU j'ai été pour la première fois confronté à la notion d'architecture logicielle. Dans ce cadre j'ai travaillé sur le langage d'architecture logiciel Wright et plus particulièrement sur son extension afin de modéliser des reconfigurations dynamiques d'architecture [29].

La sémantique du langage Wright est définie en CSP [Hoa85]. Elle permet d'analyser statiquement une architecture afin de vérifier ses bonnes propriétés (essentiellement, mais pas uniquement, la compatibilité des protocoles entre composants). J'ai étendu la syntaxe de Wright pour décrire un configureur réagissant à des événements et déclenchant la création/destruction de composants ainsi que leur déconnexion/connexion avec le reste du système. La sémantique que j'en ai donnée en CSP encodait les différentes configurations et permettait de sélectionner la configuration active. Cette sémantique permettait aussi de vérifier avec une analyse statique que les reconfigurations étaient réalisées aux "bons moments" et ne perturbaient pas le bon fonctionnement du système.

Un processus séquentiel définit une séquence d'événements. L'opérateur $\rightarrow$ permet de préfixer un processus P par un événement e comme suit $e \rightarrow P$. Deux processus peuvent être composés avec opérateur de choix $P_1|P_2$ pour définir des séquences arborescentes.

Plus intéressant encore, la composition parallèle de deux processus peut être réécrite comme un seul processus avec deux règles. La première :

$$(e \to P_1) \| (e \to P_2) = e \to (P_1 \| P_2)$$

transforme la composition parallèle de deux processus qui souhaitent émettre un même événement et donc se synchronisent par rendez-vous en un seul processus qui commence par émettre cet événement (les deux branches sont unifiées en une seule dans le processus résultat). La seconde règle :

$$(e_1 \to P_1) \| (e_2 \to P_2) = (e_1 \to (P_1 \| e_2 \to P_2)) | (e_2 \to (e_1 \to P_1 \| P_2))$$

transforme la composition parallèle de deux processus qui souhaitent émettre des événements différents et donc entrelacent leur exécution en un seul processus qui commence par émettre soit un événement, soit l'autre événement (deux branches sont créées avec l'opérateur de choix dans le processus résultat). Ces règles prennent deux processus CSP et produisent un seul processus CSP. J'insiste : ces règles prennent deux programmes et produisent un seul programme.

Et en plus, cette façon de faire est compositionnelle. Le résultat de la composition étant un processus CSP, il peut de nouveau être composé avec un autre processus, qui va à la fois l'enrichir (en créant de nouvelles branches) et l'appauvrir (en sélectionnant certaines branches et faisant disparaitre les autres).

Un processus CSP peut être vu comme une spécification mais aussi comme un programme qui émet ou reçoit des événements. La composition parallèle peut donc être vue comme une transformation de programmes qui prend deux programmes en entrée et produit un unique programme en sortie, synthèse des deux autres. Dans CSP la composition parallèle est symétrique : les règles P_1 et P_2 sont interchangeables.

Dans le travail que j'ai effectué sur Wright, il y avait deux parties dissymétriques : le programme qui effectue un calcul et un reconfigurateur qui passivement ignore la plupart du temps le calcul en question et à certains moments arrête le programme, change son architecture, puis le redémarre. Cette dissymétrie a pu être codée en CSP grâce aux notions d'alphabets et de choix (déterministe et non-déterministes) de CSP.

3.2 Langages réflexifs

En arrivant à l'Ecole des Mines de Nantes je me suis intéressé à ce qui se faisait sur place. A l'époque les membres de mon équipe de recherche avait une forte activité en langages réflexifs.

En théorie, les langages réflexifs permettent d'écrire des programmes qui s'observent (on parle d'introspection) et se modifient (on parle d'intercession) alors même qu'ils s'exécutent. En pratique les langages réflexifs offrent une API qui définit le pouvoir d'observation et de modification des mécanismes réflexifs. Certains langages offrent une API puissante (par exemple, le Meta Object Protocol de Smalltalk [KR91]). D'autres langages offrent une API beaucoup plus limitée. Par exemple Java a des capacités d'introspection limitées à sa partie objet et permet de réifier différentes entités (`java.lang.Class`, `java.lang.reflect.Method`, ...) mais Java a des capacités d'intercession très très limitées qui se résument à : charger une classe dynamiquement, créer une instance, accéder à un champ ou encore appeler une méthode. En Java il est par exemple impossible de changer la structure d'une classe ou encore sa relation d'héritage (et je ne parle même pas de la partie non objet impérative du langage). Ces capacités réflexives limitées permettent tout de même d'écrire du code générique. En particulier, les premières versions de Java RMI nécessitaient la génération de talons serveur sur mesure qui désérialisaient une représentation de l'appel distant et effectuaient l'appel effectif. Depuis Java 1.2 et l'API de

réflexion, RMI repose sur un talon serveur unique qui désérialise la représentation de l'appel à effectuer, en construit une représentation réifiée dans l'API réflexive et enfin invoque la méthode.

Les API des différents langages réflexifs ont toutes une limite (par exemple même le MOP de Smalltalk ne permet pas de changer la sémantique de l'exécution séquentielle) et semblent souvent arbitraires et reposer sur des traditions (par exemple l'appel de méthode se décompose en deux étapes-`lookup` et `apply`). Pour comprendre les bases des langages réflexifs je me suis plongé dans les travaux fondateurs du domaine. Dès 1984, Smith [RS84] proposait pour les langages réflexifs un modèle général et formel basé sur une tour (potentiellement) infinie d'interprètes méta-circulaires. Des opérateurs permettaient à un code de s'exécuter dans un interprète plus haut dans la tour ou au contraire dans un interprète plus bas dans la tour. Il était ainsi possible de gravir les étages pour exposer (introspection) et modifier (intercession) des mécanismes d'interprétation, ou de redescendre des étages pour y exécuter du code sous les niveaux supérieurs modifiés. Le caractère arbitraire d'une API réflexive dans ce cas apparaissait spontanément en changeant de définition d'interprète (il y a de nombreuses manières d'écrire un interprète d'un même langage) et en sélectionnant les mécanismes réifiables dudit interprète.

J'ai en particulier appliqué cette approche à Java en définissant différents interprètes méta-circulaires et une transformation de programme systématique pour rendre réifiable n'importe quelle partie de ces interprètes [8]. Ce travail a montré comment générer des langages réflexifs sur mesure (pour des raisons de coût ou encore de sécurité). Il m'a aussi permis de maitriser les notions de introspection, intercession.

3.3 Le cflow de AspectJ

AspectJ [KHH01] est une extension de Java qui soutient la programmation modulaire de programmes non modulaires lorsqu'ils sont programmés en Java. AspectJ est à la fois un langage et une implémentation de référence (bien qu'elle ait changée plusieurs fois). Ce compilateur prend en entrée un programme Java et un programme AspectJ et génère en sortie un programme Java (des versions plus récentes visent directement le bytecode de la machine virtuelle Java standard). AspectJ opère statiquement par transformation de programme mais peut aussi être vue comme un moniteur d'exécution.

3.3.1 Un aperçu d'AspectJ

Un aspect dans un programme AspectJ définit un point de coupe (pointcut) qui sélectionne certains événements d'exécution (joinpoint) du programme Java original (aussi appelé programme de base). Un programme AspectJ définit aussi un morceau de code (advice) qui va être exécuté avant, après ou même à la place des événements d'exécution sélectionnés. C'est en ça[1] qu'il soutient la programmation non modulaire : un aspect réunit un ou plusieurs morceaux de code qui vont être tissés à différents endroits de l'application qui ne peuvent pas être désignés par une entité modulaire du programme de base (sinon inutile de parler d'aspects, la programmation modulaire classique est suffisante). AspectJ permet ainsi d'intercepter les appels de méthodes et accès aux champs des objets pour les modifier ou les remplacer. Un aspect ressemble par certains côtés à une classe : désignation relative de code (avec `proceed`/`super`), définition de champs, instanciations multiples (via `eachObject`, `eachClass`, etc...).

[1] Une autre caractéristique est aussi souvent mise en avant : le programme de base est inconscient (oblivious) des aspects qui vont lui être appliqués et n'a pas à les anticiper.

Voici un aspect qui compte le nombre d'appels à la fonction recursive `fib` :

```
1 aspect CountFib {
2   int i = 0;
3   before(int n): execution(static int Function.fib(int)) && args(n) {
4     System.out.println(this.getClass() + " " + (++i) + "\tfib(" + n + ")");
5   }
6 }
```

Cet aspect se nomme `CountFib`. Il possède une variable d'instance `i` initialisée a zéro (ligne 2). Son point de coupe (`execution(static int Function.fib(int))`) sélectionne les points de jointure correspondants aux exécution de la méthode `fib` (ligne 3). Avant (`before`) chaque exécution le compteur `i` est incrémenté et affiché (ligne 4).

Voici un aspect qui mémorise le résultat des appels à `fib` pour éviter du recalcul :

```
1 aspect MemoFib {
2   int[] memo = new int[100];
3   int around(int n): execution(int Function.fib(int)) && args(n) {
4     if (memo[n] == 0) {
5       memo[n] = proceed(n);
6     }
7     return memo[n];
8   }
9 }
```

On suppose que le domaine de définition de `fib` est l'intervalle de 0 à 99. L'aspect `MemoFib` alloue un tableau `memo` pour mémoriser les calculs déjà effectués (ligne 2). Lorsqu'un appel à `fib` est intercepté (ligne 3), il est remplacé par l'advice qui vérifie si le résultat demandé est dans le tableau (ligne 4). Si ce n'est pas le cas, le calcul est effectué en appelant la méthode originale avec `proceed` et le résultat est mémorisé (ligne 5). Dans tous les cas, le résultat est retourné (ligne 7).

En général un aspect peut faire n'importe quoi. Voici un aspect qui remplace n'importe quel programme (`main`) par le formatage du disque dur :

```
1 aspect Format {
2   void around(String[] a): execution(void *.main(String[])) && args(a) {
3     Runtime.getRuntime().exec("format c:");
4   }
5 }
```

Ces exemples se limitent à sélectionner un point d'exécution isolé et à agir. Nous abordons maintenant des aspects qui concernent plus d'un point. Ici nous parlons de plusieurs points à relier entre eux, et non pas de plusieurs points autonomes répondants à une méta-caractéristique quantifiée universellement et plus ou moins syntaxique comme par exemple toutes les méthodes dont le nom commence par la lettre “a”.

3.3.2 L'opérateur `cflow`

L'opérateur `cflow` d'AspectJ permet de définir des coupes qui sélectionnent et relient plusieurs points de jointure. En effet, `cflow(pointcut)` désigne tous les points de jointure entre l'appel et le retour à la méthode correspondant à `pointcut`. Par exemple, considérons l'aspect suivant :

```
1 aspect Quota { before(Client c, Worker w):
2   cflow(execution(void Client.ask()) && target(c))
3   && execution(void Worker.perform()) && target(w) {
4     System.out.println("Client:" + c + "␣Worker:" + w);
5   }
6 }
```

Le point de coupe, ligne 2 et 3, de cet aspect `Quota` est de la forme `cflow(pcd1) && pcd2` où `pcd1` désigne les exécutions de la méthode `Client.ask()` et `pcd2` désigne les exécutions de la méthode `Worker.perform()`. Le point de coupe `cflow(pcd1)` désigne tous les points de jointure pendant les exécutions de la méthode `Client.ask()`. Et donc, le pointcut complet désigne toutes les exécutions de la méthode `Worker.perform()` *pendant* les exécutions de la méthode `Client.ask()`. Et seulement celle là, si `Worker.perform()` est exécuté alors que `Client.ask()` n'est pas dans la pile d'appels, cette exécution ne sera pas un point coupe pour `Quota`. L'opérateur `cflow` permet donc de définir un point de coupe en relation avec un autre. Ces points peuvent être arbitrairement distants dans l'exécution : la pile d'appels peut contenir un nombre arbitraire d'appels entre les deux.

Cet opérateur permet lorsque deux points de coupe sont ainsi reliés de passer du contexte du premier vers le second. En effet, dans l'exemple ci dessus le point de jointure contient aussi deux lieurs : `target(c)` à la ligne 2 qui lie `c` au client receveur de `ask()` et `target(w)` à la ligne 3 qui lie `w` au travailler receveur de `perform()`. Il est dès lors possible de référencer le client et le travailleur dans l'action de l'aspect. Ici, on fait un simple affichage, mais un aspect de quota pourrait vérifier que le client a le droit de faire traiter une tâche par ce travailleur et dans le cas contraire d'ignorer la tâche, voire de générer une alerte.

Il est possible de définir `cflow` dans un sous ensemble d'AspectJ (sans utiliser `cflow`). Cette définition a l'avantage de donner une sémantique claire de cet opérateur et d'ouvrir la porte au codage d'autres opérateurs.

```
1  aspect Quota {
2    Stack c = new Stack();
3    before(Client c): execution(void Client.ask()) && target(c) {
4      this.c.push(c); // 1 store
5    }
6    after(): execution(void Client.ask()) {
7      c.pop(); // 3 discard
8    }
9    before(Worker w): execution(void Worker.perform()) && target(w) {
10     if (!c.isEmpty()) { // 2 access top (last, most recent)
11       System.out.println("Client:" + c.peek() + "␣Worker:" + w);
12   }
13 }
```

Pour des raisons de sécurité la pile d'appel en Java n'est pas introspectable, cet aspect va donc la réifier (ligne 2). Pour la maintenir à jour, avant chaque exécution de `ask` (ligne 3) le client est

empilé (ligne 4), et après chaque exécution de `ask` (ligne 6) le client est dépilé. Avant chaque exécution de `perform`, si il y a au moins un client dans la pile (ligne 10), l'identité du plus récent est affichée (ligne 11).

Enfin, il faut noter que cet opérateur est compositionnel et permet de définir des pointcuts abitrairement complexes. Par exemple le pointcut suivant désigne les appels à la méthode `C.c` pendant un appel à `B.b` lui même pendant un appel à `A.a` :

```
1 cflow(
2   cflow(execution(void A.a())) && execution(void B.b())
3 ) && execution(void C.c())
```

Enfin, tous les exemples développés ici se limitent à un seul aspect. En général, il peut y avoir plusieurs aspects en même temps. Dans ce cas, il doivent être composés pour obtenir le comportement souhaité. En effet, lorsque deux aspects ont des points de coupe en commun, deux actions doivent être tissées à un même point de programme. En AspectJ la composition se limite à ordonner les aspects (et donc leurs actions). En général, les problèmes peuvent nécessiter des compositions plus complexes (par exemple une action prime sur l'autre, ou bien les deux actions s'excluent et s'annulent mutuellement).

Cet opérateur `cflow` a retenu toute mon attention, car il permet de définir modulairement et déclarativement (inutile de consulter le code tissé pour anticiper ce qui va se passer) des coupes non modulaires (c'est-à-dire une coupe est définie par au moins deux points de programmes).

Dans mes travaux, je me suis notamment intéressé à enrichir les langages de coupes et la composition des aspects.

3.4 Conclusion

Durant mes premières années de jeune chercheur j'ai été amené à m'intéresser aux processus séquentiels communicants, aux langages de programmation réflexifs, à l'opérateur d'AspectJ de définition de point de coupe sur le flot de contrôle. Ces trois sujets et travaux associés m'ont convaincu qu'il était possible d'explorer des solutions langages (de programmation) au problème de la tyrannie de la décomposition primaire identifiée dans le chapitre précédent. En effet le formalisme CSP montrait comment des programmes avec des structures différentes pouvaient être fusionnés pour générer un programme unique. Les langages réflexifs offraient un cadre pour limiter le pouvoir d'observation et d'action d'un programme sur un autre (ou lui même). Enfin l'opérateur `cflow` d'AspectJ prouvait qu'il était possible de définir de façon modulaire des relations non modulaires entre deux programmes. Mes travaux qui en découlent sont décrits dans la seconde partie de ce document.

Deuxième partie

Contribution

Chapitre 4

Flot dynamique de contrôle séquentiel

Sommaire

Ce chapitre détaille mes recherches dans le domaine de la programmation non modulaire modulaire de flot de contrôle.

4.1 Point de coupe séquentiel

4.1.1 Introduction

On a vu dans la section 3.3.2 comment définir un aspect de quota. Son point de coupe utilise l'opérateur `cflow` pour sélectionner les points de jointure correspondant à l'exécution d'un service pendant l'exécution d'un client. Ce point de coupe permet aussi de passer du contexte pour que l'action puisse accéder à l'identité du client avant l'exécution du service pour vérifier son quota. Enfin, on a vu que cet opérateur cache une réification de la pile des appels.

Dans un système de traitement par lots, cet aspect ne fonctionne plus. En effet dans un système de traitement par lots, la requête d'un client est stockée dans une file et la méthode `Client.ask()` termine alors que `Worker.perform()` n'a pas encore été exécutée. Dans une telle architecture, l'exécution du service n'est plus dans le flot de contrôle de la requête du client.

La définition de l'aspect doit être adaptée pour prendre en compte cette nouvelle relation entre les deux points de jointure. Cela consiste tout naturellement à réifier la file (c.a.d. la séquence) des client comme suit :

```
1  aspect Quota {
2    Queue cs = new LinkedList();
3    before(Client c): execution(void Client.ask()) && target(c) {
4      cs.offer(c); // 1 store
5    }
6    before(Worker w): execution(void Worker.perform()) && target(w) {
7      // 2 access head (oldest)
8      System.out.println("Client:" + cs.remove() + " Worker:" + w);
9    }
10 }
```

A la ligne 2 une file est déclarée. A la ligne 4 l'identité du client qui fait une requête est mise dans la file. A la ligne 8 l'identité d'un client est extrait de la file lorsqu'un service est exécuté. A la différence du scénario basé sur le cflow, il n'y a pas de test dynamique à la ligne 7 pour vérifier que la file n'est pas vide. En effet, dans ce scénario on suppose qu'exactement un service est appelé pour réaliser la requête d'un client : son identité est extraite de la file quand qu'un service est appelé.

En général un aspect suppose que le programme de base suit un certain protocole. Par exemple, l'aspect suivant compte le nombre de sessions créées :

```
1  aspect Sessions {
2    boolean outOfSession = true;
3    int i = 0;
4    before(): execution(* *.login()) &&  if(outOfSession) {
5      i++;
6      outOfSession=false;
7    }
8    before(): execution(* *.logout()) && if(!outOfSession) {
9      outOfSession=true;
10   }
11 }
```

Il incrémente le compteur `i` (initialisé à la ligne 3) à chaque `login` (ligne 5). Il utilise aussi un booléen `outOfSession` initialement vrai (ligne 2). Lorsqu'il n'y a pas de session active (`if(outOfSession)` dans le point de coupe ligne 4) et que l'on rencontre `login`, le compteur de session est incrémenté et `outOfSession` est mis à faux (ligne 6) pour refléter l'entrée dans une session. Lorsqu'il y a une session active (`if(!outOfSession)` dans le point de coupe ligne 8) et que l'on rencontre `logout`, `outOfSession` est mis à vrai (ligne 9) pour refléter la sortie de session. Dans tous les autres cas, il ne se passe rien. En particulier, quand on est hors session un `logout` ne déclenche aucune action. De même, quand on est en session un `login` est ignoré : cet aspect suppose que le programme de base ne soutient pas les sessions imbriquées. Les hypothèses peuvent ici s'exprimer comme l'expression régulière `(login logout)*` qui spécifie que l'aspect s'intéresse à des séquences de `login logout`. Cette expression régulière peut être reconnue par un automate à deux états (encodés ici par le booléen). Pour des expressions régulières plus complexes, le booléen dans l'aspect doit être remplacé par une variable qui code l'état d'un automate

et chaque transition est codée par un point de coupe qui teste l'état courant et un advice qui modifie l'état courant[1]. Il faut remarquer ici qu'un aspect a désormais un flot de contrôle : la première action et la seconde ne peuvent être exécutées qu'en alternance. Un aspect se rapproche donc d'un programme. Mais ce flot de contrôle est passif, à l'écoute des points de jointures du programme de base : quand un point de jointure ne le concerne pas, aucun point de coupe ne le sélectionne et donc aucun advice n'est exécuté.

Un aspect sur un flot de contrôle séquentiel doit définir des points de coupe sensibles au flot de contrôle. Il doit spécifier comment différents points de jointure sont reliés. En particulier il doit offrir une notion de passage de contexte. Il peut offrir des notions de composition basées sur les flots de contrôle des aspects à composer. Lorsque le flot de contrôle d'un aspect peut se modéliser avec un automate à état fini, des analyses statiques (par exemple intersection de deux automates représentant des points de coupe sensibles au flot) sont possibles. Ces idées sont à la base des travaux décrits dans ce chapitre.

4.1.2 Point de coupe comme un analyseur syntaxique de points de jointures

Un premier travail [28] a consisté à présenter la programmation par aspect comme un moniteur d'exécution qui passivement écoute les points de jointures émis par l'exécution d'un programme de base. Le point de jointure peut déclencher (ou non) l'exécution d'une action dans l'aspect. Cette exécution est synchrone : l'aspect rend ensuite la main au programme de base qui poursuit son exécution.

Un aspect va alors devoir relier différents points de jointures entre eux. Par exemple, la séquence de points de jointures :

```
a=selectCustomer();
...
b=selectCustomer();
...
callService(c);
...
callService(d);
...
```

pourra être interprétée par un aspect comme "le client `a` déclenche le service `c` et le client `b` déclenche le service `d`" (associations entrelacées) ou bien par un autre aspect comme "`a` déclenche le service `d` et le client `b` déclenche le service `c`" (associations imbriquées). La distinction est bien sûr cruciale par exemple pour une action de facturation de services aux clients. Ce choix entre reconnaître des motifs parenthèsés ou entrelacés dépend de l'architecture de l'application de base (par exemple récursive ou séquentielle) et possiblement de la logique de facturation afin de facturer le bon client.

Afin d'exprimer ces différentes associations, nous avons proposé un langage dédié pour définir des points de coupe sensibles au flot de contrôle. Ce langage dédié (DSL) détaillé dans la figure 4.1 se base sur des analyseurs syntaxiques monadiques. En effet, ce cadre autorise la reconnaissance de motifs parenthésés (ce qui est requis par `cflow`). Il est constructif : il offre des opérateurs qui permettent à la fois de définir les séquences à reconnaître mais aussi de construire l'analyseur correspondant. Ces analyseurs peuvent être ambigüs et retourner plusieurs résultats ce que nous exploitons pour exprimer la reconnaissance de plusieurs séquences entrelacées.

[1] En général un aspect peut utiliser des variables et des conditions pour définir non pas une expression régulière mais une séquence arbitrairement complexe.

```
1 data PcD =
2     Return [Joinpoint]
3   | Bind ([Joinpoint] -> PcD)
4   | PcD `Seq` PcD
5   | Filter (Joinpoint -> Bool) PcD
6   | Abort
7   | PcD `Par` PcD
8   | First PcD
```

FIGURE 4.1 : Un langage dédié au point de coupes sensibles au contrôle

```
1 billingS :: PcD
2 billingS =
3   next "c" `Seq` Bind (\[e1] ->
4   next "s" `Seq` Bind (\[e2] ->
5   Return [e1,e2] `Par` billingS))
```

FIGURE 4.2 : Un point de coupe pour des séquences non entrelacées, non imbriquées

Nous avons choisi d'exprimer notre langage directement en Haskell ce qui nous donne une définition concise et opérationnelle de nos concepts. Le langage suit la structure monadique habituelle en fournissant deux opérateurs `Return` (ligne 2) et `Bind` (ligne 3). Ce langage définit des associations entres des points de jointure comme exprimé par la liste de points de jointure en paramètre de ces deux opérateurs. L'opérateur `Seq` (ligne 4) définit une séquence comme `Bind` mais en passant le contexte de façon implicite. L'opérateur `Filter` (ligne 5) permet de définir des points de coupe conditionnels. Les opérateurs `Abort` (ligne 6) et `Par` (ligne 7) correspondent aux opérateurs de la monade "zéro" et de la monade "plus". Ils permettent de reconnaitre en parallèle plusieurs points de coupe ou bien de mettre fin à une telle recherche. A l'aide de la récursion ces opérateurs permettent d'exprimer entrelacement et imbrication. Enfin `First` permet de recherche la première occurrence correspondant à un point de coupe.

Par exemple dans la figure 4.2 le point de coupe `billingS` cherche la prochaine occurrence d'un point de jointure concernant un client, ici pour simplifier `c` à la ligne 3, puis la prochaine occurrence d'un point de jointure concernant un service, ici pour simplifier `s` à la ligne 4, puis il retourne ces deux points liés à `e1` et `e2` et continue en parallèle à chercher d'autres paires, à la ligne 5.

Une sémantique du langage est donnée par des règles d'équivalence monadiques que l'on utilise comme des règles de réécriture pour obtenir un interpréteur d'un point de coupe `PcD`. Nous avons défini cette réécriture de terme en Haskell afin de tester la sémantique de différents points de coupe sur des liste de points de jointures. Nous avons aussi mis en œuvre un prototype en Java en traduisant systématiquement les termes `PcD` comme des arbres d'objets, les règles de réécritures associées comme des méthodes, et en introduisant des actions à exécuter quand un point de coupe a détecté une séquence de points de jointure.

4.1.3 Points de coupe généralisés et composition d'aspects

Le travail présenté dans cette sous section se base sur un article publié à la conférence française LMO [31]. Il introduit le terme de programmation par aspects événementiels (le modèle de la section précédente basé sur les moniteurs d'exécution) et décrit un prototype pour Java.

Ce prototype est plus puissant que celui décrit dans la sous section précédente. En effet il s'émancipe du cadre grammatical et permet de définir des séquences arbitrairement complexes de points de jointures. Pour ce faire la pleine puissance de Java est autorisée. Un point de coupe est un programme Java arbitrairement complexe qui fait appel à une API pour observer le prochain point de jointure. Afin d'alterner l'exécution du programme de base et celles des aspects, nous avons écrit une librairie de coroutines. Un aspect peut se mettre en pause lorsqu'il attend le prochain point de jointure, et le programme de base peut se mettre en pause dès qu'il génère un point de jointure. Les coroutines nécessitent un fil d'exécution par aspect et un autre pour le programme de base, ce qui a bien sûr un coût mais n'impose pas de forme particulière au programme de base ou aux aspects (par exemple l'API qui retourne le prochain point de jointure peut être appelée à n'importe quel endroit du code, comme par exemple dans le corps d'une boucle). Une alternative aurait été de transformer le code en style de passage de continuations, ce qui peut être complexe dans un langage impératif comme Java.

Ce prototype nous a aussi permis de proposer de nouveaux outils pour la composition d'aspects. La composition d'aspects dans AspectJ se limite à déclarer un ordre. Dans notre prototype les aspects sont les feuilles d'un arbre binaire. Chaque nœud est un opérateur binaire de composition de d'aspects. Cette approche est compositionnelle : un sous arbre peut être vu comme un aspect. Lorsque le programme émet un point de jointure et fait une pause, le point de jointure va parcourir l'arbre dans un parcourt descendant gauche droite possiblement altéré par les opérateurs.

La liste des opérateurs proposés est la suivante :

- `Seq` propage le point de jointure courant à son fils gauche puis à son fils droit.
- `Any` propage le point de jointure courant à ses fils dans un ordre arbitraire.
- `Fst` propage le point de jointure courant à son fils gauche. Chaque aspect primitif (et composition d'aspects) retourne un booléen `isCrosscuting` indiquant si le point de jointure a retenu l'intérêt du point de coupe. Lorsque le fils gauche retourne une valeur vraie le point de jointure n'est pas transmis au fils droit de `Fst`. Dans le cas contraire il est transmis au fils droit. Le booléen retourné par `Fst` est vrai si l'un des deux fils a retourné vrai.
- `Cond` propage le point de jointure courant à son fils gauche. Lorsque le fils gauche retourne une valeur vraie le point de jointure est transmis au fils droit de `Cond`. Dans le cas contraire il n'est pas transmis au fils droit. Le booléen retourné par `Cond` est vrai si l'un des deux fils a retourné vrai.

Notre prototype a été évalué sur une application idéalisée de commerce en ligne. Dans cette application, des clients effectuent des recherches dans le catalogue, remplissent leur panier, et valident leurs achats. Plus tard, par exemple quotidiennement, l'application traite par lots les commandes. Nous avons considéré deux aspects : un aspect de loterie (le millième client ne paye que la moitié du prix de sa commande) et un aspect de rabais (les achats successifs d'un client sont cumulés, quand la somme atteint 100 euros, le prix de sa prochaine commande sera réduit de 10%). La composition `Fst(Loterie,Reduction)` permet de s'assurer qu'un client qui gagne à la loterie ne bénéficie pas en même temps d'une réduction.

Notre prototype autorise des aspects sur des aspects : les actions d'un aspect émettent des points de jointure qui peuvent être utilisés par des aspects. Cela a nécessité que notre parcours d'arbre soit réentrant : si pendant le parcours l'action d'un aspect génère un point de jointure, alors le parcours de l'arbre est repris depuis la racine avec le nouveau point de jointure, et une fois fini on reprend le parcours de l'arbre avec l'ancien point de jointure. Afin de ne pas créer de verrou, un aspect donc l'action s'exécute signale qu'il rejette systématiquement les points de jointure. Un aspect qui a émis un point de jointure est en pause et n'observe pas les points de jointures. Ces protocoles sont soutenus par notre prototype qui permet essentiellement de se concentrer sur l'écriture du code fonctionnel (du point de coupe, de l'action, de l'enchaînement des points de coupes). Nous avons validé notre proposition en introduisant un aspect comptable qui cumule les rabais effectués. Cet aspect peut être composé séquentiellement avec les autres comme `Seq(Fst(Loterie,Reduction),CumulRabais)`.

4.2 Formalisation

Ces travaux nous ont conduit à définir une sémantique formelle de notre approche et à nous intéresser à l'analyse statique des interactions entre aspects en nous limitant aux points de coupe définissant des séquences régulières de points de jointure. Les travaux décrits dans cette section se basent sur deux articles de conférence [27] [24] et un chapitre de livre [44].

4.2.1 Programme de base et aspects

Notre cadre formel pour la détection et la résolution des interactions entre aspects est indépendant de tout langage de programmation. On suppose juste que la sémantique de ce langage peut être donnée sous la forme d'une sémantique petit pas. La partie de l'exécution utile pour le tissage est appelée la trace d'exécution observable. Elle peut être définie sur la base d'une sémantique petit pas du programme de base. La trace observable est une séquence de points de jointure qui sont des abstractions de l'état du programme de base.

Nos aspects sont définis par des règles de la forme : $C \rhd I$ où C est un point de coupe et I une action. Les points de coupe filtrent des points de jointure et les actions sont des schémas de code. L'intuition derrière une règle de base est que lorsque le point de coupe sélectionne le point de jointure courant, il retourne une substitution qui est appliquée au schéma de code qui est ensuite exécuté. Par exemple la règle : `error(`m`)` $\rhd$ `abort()` arrête l'exécution quand le programme de base appelle la fonction `error`. Les aspects composent des règles selon la syntaxe définie dans la Table 4.1. Une règle peut préfixer un aspect : ceci introduit une notion de séquence et d'état dans

A	$::=$	$C \rhd I; A$	préfixe
		$\mu a.A$	définition récursive
		$C \rhd I; a$	fin de séquence
		$A_1 \square A_2$	choix

TABLE 4.1 : Grammaire A des aspects

un aspect. Une fois que la règle a exécuté son action, la règle suivante est activée. Un aspect peut être défini récursivement (et une définition récursive contient au moins une règle). Un aspect peut composer deux aspects avec l'opérateur de choix $\square$: ceci permet à un aspect d'avoir plusieurs règles actives en même temps (la première règle à réagir sélectionnera une branche à la manière d'une conditionnelle, en cas de compétition la branche de gauche est prioritaire). Par exemple, l'aspect dans la Table 4.2 journalise les messages d'avertissement une fois que le fichier

de journalisation a été ouvert (l'action `skip` ne fait rien) et il arrête l'exécution du programme en cas d'erreur.

$$\texttt{openLog()} \rhd \texttt{skip}; \mu a. \quad \begin{array}{l} \texttt{warning(m)} \rhd \texttt{writeLog(m)}; a \\ \Box\ \texttt{error(m)} \rhd \texttt{abort()}; a \end{array}$$

TABLE 4.2 : Un aspect de journalisation et d'arrêt

4.2.2 Tissage silencieux et visible

Plusieurs aspects peuvent être utilisés en même temps. Dans ce cas ils sont composés comme : $A_1 || \ldots || A_n$. Le tissage définit comment le filtrage des points de coupe et l'exécution des actions sont entrelacés avec l'exécution du programme de base. Intuitivement, le tisseur prend une composition parallèle de n aspects et réalise les étapes suivantes à chaque point de jointure :

- Les règles applicables (dont le point de coupe sélectionne le point de jointure courant) sont déterminées par une fonction *sel*. Par exemple, dans le cas de l'aspect de la Table 4.2 la fonction *sel* retourne la règle `openLog()` $\rhd$ `skip` pour le premier point de jointure qui ouvre le fichier de journalisation. Elle retourne un ensemble vide pour tous les autres points de jointures qui ouvrent le fichier de journalisation.
- Toutes les règles sélectionnées sont appliquées (c.a.d. leurs actions sont exécutées) dans un ordre arbitraire.
- L'évolution des aspects est calculée par la fonction *next*. Ce même aspect n'évolue pas avant d'avoir rencontré la première ouverture du fichier de journalisation. Une fois que le point de jointure qui ouvre le fichier a été pris en compte, *next* retourne $\mu a.$`warning(`m`)` $\rhd$ `writeLog(`m`)`$; a \ \Box$ `error(`m`)` $\rhd$ `abort()`$; a$
- Le programme de base effectue un pas d'exécution ce qui génère le prochain point de jointure.

Ces étapes sont répétées jusqu'à l'arrêt du programme de base. L'exécution du programme tissé est formalisée dans la Figure 4.3. Le début et la fin d'un programme sont représentés par deux points de jointure spécialisés $\downarrow$ et $\uparrow$. La relation de transition $\rightarrow$ représente l'exécution standard. Si σ_0 est l'état initial, la trace observable du programme P est de la forme :

$$(\downarrow, P, \sigma_0) \rightarrow \ldots \rightarrow (j_i, P, \sigma_i) \rightarrow \ldots$$

Si la réduction termine, il existe un σ_n tel que $(\downarrow, P, \sigma_0) \stackrel{*}{\rightarrow} (\uparrow, P, \sigma_n)$ où $\stackrel{*}{\rightarrow}$ désigne la fermeture réflexive et transitive de $\rightarrow$. L'exécution tissée $\Longrightarrow$ est définie par l'application du moniteur suivie par une étape d'exécution standard. Elle retourne l'aspect $(next\ j\ A)$ à appliquer au point de jointure suivant. A chaque point de jointure, les règles applicables sont sélectionnées $(sel\ j\ A)$. La relation du moniteur $\models\!\!\Longrightarrow$ applique dans un ordre arbitraire les règles sélectionnées : si le point de coupe de la règle courante sélectionne le point de jointure courant, la substitution correspondante est appliquée à l'action qui est alors exécutée. Dans cette définition du tissage les actions sont silencieuses (c.a.d. non tissables) en effet c'est la réduction standard $\longrightarrow$ qui est utilisée pour les réduire. Il suffit de remplacer cette relation par la relation tissée $\Longrightarrow$ pour rendre les actions visibles (c.a.d. tissables). Cette variante permet de définir des aspects d'aspects. Elle permet aussi au programme tissé de diverger (quand par exemple un aspect se tisse lui même indéfiniment).

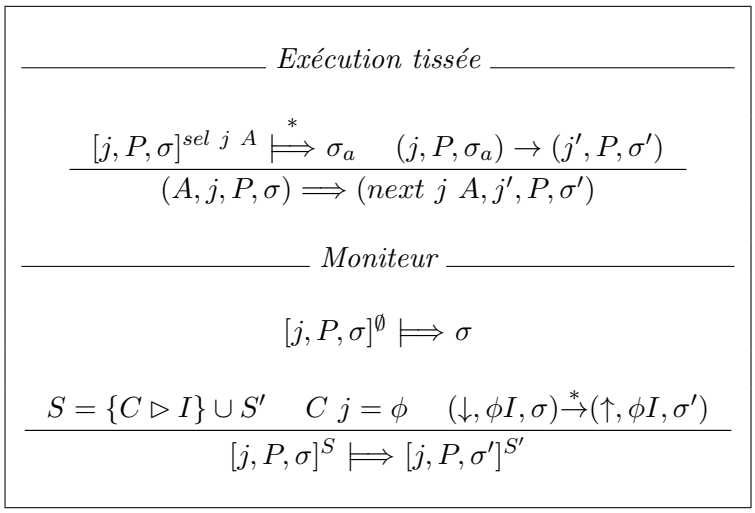


FIGURE 4.3 : Tissage

$$
\begin{array}{lcl}
let\ A & = & (C_1 \rhd I_1; A_1) \Box \ldots \Box (C_n \rhd I_n; A_n) \\
and\ A' & = & (C'_1 \rhd I'_1; A'_1) \Box \ldots \Box (C'_m \rhd I'_m; A'_m) \\
then\ A||A' & = & \Box_{i=1\ldots n}^{j=1\ldots m} C_i \wedge C'_j \rhd (I_i \bowtie I'_j); (A_i||A'_j) \\
 & & \Box_{i=1\ldots n} C_i \rhd I_i; (A_i||A') \\
 & & \Box_{j=1\ldots m} C'_j \rhd I'_j; (A||A'_j)
\end{array}
$$

FIGURE 4.4 : Composition de deux aspects

4.2.3 Indépendance forte et vis-à-vis d'un programme de base

Deux aspects interagissent quand ils sélectionnent un même point de jointure. Deux aspects sont indépendants si ils ne sélectionnent jamais en même temps un point de jointure. L'indépendance de deux aspects garantit que leur composition parallèle est bien définie : ils peuvent être tissés dans n'importe qu'elle ordre. Au contraire, deux aspects dépendants (interagissants) nécessitent que le programmeur supprime l'interaction en changeant les aspects où en les composant. Considérons par exemple, l'aspect `Aencryption` qui sélectionne des appels de fonction et crypte leurs arguments, et l'aspect `Alogging` qui journalise des appels de fonction. Si certains appels sont sélectionnés par les deux aspects, ces aspects interagissent et leur composition en parallèle n'est pas bien définie. Dans ce cas, puisque cette composition ne spécifie pas d'ordre d'exécution pour leurs actions, le tissage est non déterministe.

L'indépendance de deux aspects peut être vérifiée statiquement en calculant l'intersection de leurs points de coupe. Lorsque cette intersection est non vide, deux points de coupe peuvent sélectionner en même temps un point de jointure. Cette vérification peut être réalisée indépendamment de tout programme de base : dans ce cas on parle d'indépendance forte (quelque soit le programme de base) entre aspects. Pour ce faire, on utilise des règles qui prennent une composition parallèle de deux aspects et retourne un seul aspect. La principale règle inspirée de CSP est détaillée dans la Figure 4.4.

Cette règle considère deux aspects A et A' tous deux constitués de choix $\Box$. L'aspect résultant de la composition parallèle $A||A'$ est aussi constitué de choix divisés en trois catégories : les choix qui proviennent d'un point de coupe C_i de A et d'un point de coupe C'_j de A' et qui font progresser

les deux aspects, les choix qui proviennent d'un point de coupe de C_i de A où seul A progresse et les choix qui proviennent d'un point de coupe C'_j de A' où seul A' progresse. La composition des points de coupe $C_i \wedge C'_j$ peut dans certains cas être simplifiée statiquement (par exemple quand ils sélectionnent respectivement la fonction `f` et la fonction `g` le point de coupe résultant ne sélectionne aucune fonction (aucune fonction ne s'appelle à la fois `f` et `g`) et la branche de choix correspondante peut être supprimée.

4.2.4 Résolution des conflits

Lorsque le composition des points de coupe ne peut être simplifiée, l'action composite $I_i \bowtie I'_j$ indique un conflit puisque deux actions devront être exécutées à un même point de jointure. Le programmeur est responsable de remplacer ces actions conflictuelles par une action unique. Dans certains cas le remplacement peut être automatisé en remplaçant $\bowtie$ par un opérateur de composition d'actions `op`. On note la composition d'aspects : $A||_{\texttt{op}}\ A'$. Ainsi en cas de conflit, $A||_{\texttt{seq}}\ A'$ exécute l'action de A puis celle de A', $A||_{\texttt{fst}}\ A'$ n'exécute que l'action de A, $A||_{\texttt{snd}}\ A'$ n'exécute que l'action de A', $A||_{\texttt{qes}}\ A'$ exécute l'action de A' puis celle de A, et enfin $A||_{\texttt{skip}}\ A'$ exécute une action vide. Lorsque tous les conflits ne doivent pas être résolus de manière homogène il est aussi possible d'utiliser un opérateur de composition défini selon la grammaire de la Table 4.3. Cette grammaire a la même structure que celle des aspects. Les actions y sont

O	$::=$	$C \rhd (U, B); O$	préfixe
		$\mu o.O$	définition récursive
		$C \rhd (U, B); o$	fin de séquence
		$O_1 \square O_2$	choix
U	$::=$	$id \mid skip$	
B	$::=$	$\bowtie \mid seq \mid fst \mid snd \mid qes \mid skip$	

TABLE 4.3 : Grammaire O des opérateurs de composition

remplacées par une paire d'opérateurs : un opérateur unaire U à appliquer en l'absence de conflit et un opérateur binaire B à appliquer en cas de conflit. Cette grammaire permet par exemple de définir l'opérateur de composition par défaut comme :

$$|| = ||_{default} \text{ avec } default = \mu o.true \rhd (id, \bowtie); o$$

Le point de coupe $true$ sélectionne tous les points de jointure et donc $default$ remplace I par (idI) (c.a.d. I) lorsqu'il n'y a pas de conflit, et $I \bowtie I'$ par $I \bowtie I'$ en cas de conflit.

D'autres opérateurs plus intéressants peuvent être définis. Par exemple un opérateur $fair$ alterne les aspects en cas de conflit peut être défini comme :

$$|| = ||_{fair}\ avec\ fair = \mu o.true \rhd (id, fst);\ true \rhd (id, snd); o$$

4.2.5 Aspects réellement réutilisables

Parfois la définition d'un aspect n'est valable que pour une famille de programmes de base. Par exemple, l'aspect de journalisation définit dans la Table 4.4 n'est tissable que sur un programme de base qui n'autorise pas les sessions imbriquées. En effet, `flatLog` commence par attendre un point de jointure correspondant à un appel à `login`. Puis, soit le prochain point de jointure correspond à `logout` et l'aspect recommence, soit le prochain point de jointure correspond à la lecture d'un fichier f et le nom de ce fichier est journalisé, puis l'aspect attend de nouveau soit

$$\begin{array}{lll} \texttt{flatLog} = \mu a_1. & \texttt{login()} & \triangleright\ \texttt{skip}; \\ & \mu a_2. & \texttt{logout()} \triangleright \texttt{skip}; a_1 \\ & & \square\ \texttt{read}(f) \triangleright \texttt{addLog}(f); a_2 \end{array}$$

TABLE 4.4 : Un aspect de journalisation de session

la fin de session, soit une nouvelle lecture. Si les sessions sont imbriquées, la première fermeture de session (imbriquée ou non) sera considérée comme la fin de la session principale par l'aspect. Ainsi pour la séquence :

```
login(); login(); read(f1); logout(); read(f2); logout()
```

l'aspect journalisera l'accès à `f1` mais pas celui à `f2`.

Dans un tel cas, il peut être utile de spécifier les hypothèses qu'un aspect fait sur le programme de base (c.a.d. son domaine de validité). Ceci peut être fait sous la forme d'encore un aspect, comme détaillé dans la Table 4.5. L'aspect `flat` commence par attendre `login` (si il re-

$$\begin{array}{lll} \texttt{flat} = \mu a. & \texttt{login()} & \triangleright\ \texttt{skip}; \\ & & \texttt{logout()} \triangleright \texttt{skip}; a \\ & & \square\ \texttt{login()} \triangleright \texttt{abort()}; a \\ & \square\ \texttt{logout()} & \triangleright\ \texttt{abort()}; a \end{array}$$

TABLE 4.5 : Un aspect de détection des sessions non imbriquées

çoit `logout` il met fin à l'exécution avec l'action `abort()`. Une fois le début de la session repéré, il attend la fin de session `logout` puis recommence (si il reçoit un début de session `login` là encore il met fin à l'exécution. Cet aspect ne fait donc rien (`skip`) en cas de sessions non imbriquées et termine le programme sinon. Il suffit alors d'effectuer la composition `flat||flatLog` pour obtenir un aspect de journalisation instrumenté qui termine le programme si les conditions d'application ne sont pas vérifiées. Comme pour un aspect classique le résultat de la composition est susceptible d'être simplifié. Si toutes les occurrences de `abort` disparaissent l'aspect pourra être appliqué sans crainte à n'importe quel programme de base. Si au moins une occurrence persiste, l'aspect se comportera comme un test dynamique et terminera l'exécution en cas d'imprévu.

Un aspect qui prend en compte son contexte a pu être défini comme un aspect (résultant de la composition d'un aspect avec un autre explicitant les hypothèses sur son contexte). Cet aspect instrumenté peut de nouveau être composé avec d'autres aspects. Cette technique permet d'augmenter la précision et d'éviter parfois de faux conflits (un conflit entre deux aspects qui n'arrive que dans un cas interdit par le contexte et où l'action `abort` sera exécutée de toutes façons).

Une dernière possibilité consiste à analyser statiquement le programme de base pour en extraire une abstraction de son flot de contrôle. Une telle abstraction peut permettre de simplifier un aspect instrumenté et dans certains cas de supprimer des actions `abort`, voire des conflits. Nous ne détaillons pas cette dernière approche ici mais renvoyons le lecteur aux travaux originaux.

4.3 Applications

Nous avons appliqué les aspects sensibles aux flots de contrôle séquentiels dans différents domaines. Nous balayons ici ces différentes applications.

Dans le cadre des noyaux système nous nous sommes intéressés à la programmation avec un langage dédié à des ordonnanceurs. En effet, ces gestionnaires de ressources peuvent avoir des politiques très différentes suivant le type d'applications à gérer. Par exemple des applications multimédia, ou avec des contraintes de temps réel en général nécessitent des politiques très différentes de celles utilisées pour prolonger le temps d'utilisation des appareils portables. Les opérations d'ordonnancement peuvent être exécutées très fréquemment, elles sont donc en général optimisées et mélangées avec du code très bas niveau dans de nombreuses parties du noyau. Le DSL Bossa permet d'exprimer ces politiques de manière centralisée à un niveau bien plus haut et déclaratif. L'intégration de ce DSL nécessite d'instrumenter le noyau en de nombreux points. Dans ce cas, cette instrumentation a pu être envisagée comme une cible idéale de la programmation par aspects et Bossa a pu être revisité comme un aspect isolant des politiques, surveillant l'exécution du noyau et réagissant à des événements dynamiques [42].

Toujours dans le domaine des systèmes d'exploitation mais à un plus haut niveau nous nous sommes intéressés à Arachne, un tisseur dynamique de code pour des applications C. Dans ce contexte nous avons proposé un langage de coupes sensibles au flot de contrôle et offrant le passage de contexte. Cela nous a permis de définir des aspects pour l'adaptation d'un protocole de communication d'un mode connecté (TCP) en non connecté (UDP) afin d'améliorer les performances d'applications patrimoniales. Un autre aspect a permis de détecter et d'interdire les débordements de tampons et ainsi de sécuriser des applications. Enfin, notre proposition a permis d'introduire du préchargement sous la forme d'un aspect dans le gestionnaire de cache web Squid. Tous ces travaux et expérimentations [23][6] [7] ont été menés avec un souci constant de ne pas dégrader les performances. On peut noter que tous ces exemples utilisent une forme ou une autre de protocole et donc requièrent des aspects sensibles au flot de contrôle. On peut aussi noter que souvent plusieurs instances d'un même protocole sont entrelacées pendant l'exécution et donc requièrent le passage de contexte pour les désentrelacer et relier les bons points de jointures entre eux.

4.4 Conclusion

Les travaux sur les points de coupes les plus comparables aux nôtres sont très certainement ceux sur la notion de *trace matching* [ACH05]. Notons tout de même que nos point de coupe sont plus expressifs puisque à chaque étape de la séquence est associée une action arbitraire là où les *trace match* se contentent de collecter du contexte qui sera utilisé dans l'action finale associée à la fin de la séquence.

Nos travaux sur la programmation modulaire de préoccupations non modulaires sensibles au flot de contrôle ont été menés aussi bien dans un cadre théorique qu'appliqué. Ceci a permis d'homogénéiser notre approche (par exemple les opérateurs de composition d'aspects sont définis sous une forme similaire aux aspects et les hypothèses des aspects sont définis sous une forme d'aspect). Ceci a aussi permis de valider notre approche dans un cadre réaliste et contraint. Ces travaux ont été l'occasion d'élargir mon horizon en collaborant avec des membres de l'équipe ayant une sensibilité différente (à savoir système d'exploitation). Ces collaborations enrichissantes nous ont poussés à nous intéresser à des domaines réalistes et encore plus complexes que sont les applications concurrentes.

Chapitre 5

Flot dynamique de contrôle concurrent

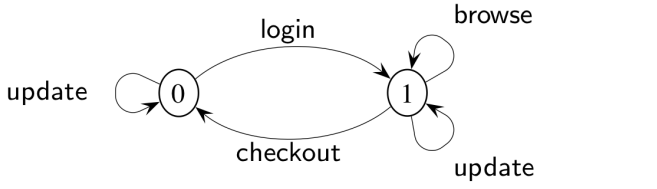
Sommaire

Les résultats décrits dans ce chapitre sont basés sur une succession de travaux [38] [37] [22]. Ces travaux montrent comment notre modèle de programmation par aspects sensibles au flot de contrôle peut être encodé dans un formalisme concurrent. Ce formalisme permet en effaçant certaines synchronisations d'étendre naturellement notre proposition pour obtenir des aspects sensibles au flot de contrôle concurrent.

5.1 Encodage dans un formalisme à états finis

Dans le cadre de la programmation concurrente, les programmes peuvent être modélisés dans des formalismes à états finis. Chaque programme est modélisé par un automate et la composition parallèle des programmes est modélisée par le produit synchronisé. Une telle formalisation peut être analysée statiquement afin de vérifier les propriétés du système (essentiellement l'absence d'interblocage). Dans le reste de ce chapitre nous utilisons la notation FSP (Finite State Process) pour spécifier le comportement de systèmes concurrents. Et nous analysons ces spécifications avec LTSA (Labelled Transition System Analyser). Ceci nous permet de spécifier et analyser des systèmes avec des aspects sensibles au flot de contrôle concurrent.

Mais commençons d'abord par spécifier une application de base inspirée du commerce en ligne. La figure 5.1 détaille la spécification d'une telle application à la fois sous forme graphique (en haut de la figure) et sous forme textuelle (en bas de la figure). Des clients se connectent à un site web en s'identifiant `login`, puis il peuvent feuilleter le catalogue en ligne `browse`. Une session se termine par le passage en caisse `checkout`. De plus, un administrateur du magasin peut mettre à jour le site à tout moment en publiant une nouvelle version `update`.



(0) Server = login → InSession
| update → Server,

(1) InSession = checkout → Server
| update → InSession,
| browse → InSession.

a) Automata representation b) FSP representation

FIGURE 5.1 : Modèle d'une application de base de vente en ligne

5.2 Flot de contrôle séquentiel

Considérons maintenant le problème d'annuler les mises à jour du site pendant une session, afin de garantir la cohérence des prix vus par le client. Cet exemple n'est pas nécessairement réaliste mais nous permet d'illustrer simplement notre approche. Voici un aspect sensible au flot de contrôle séquentiel qui permet de garantir cette cohérence

$$\mu a.\texttt{login};\mu a'.(\texttt{update} \rhd \texttt{skip log};a')\Box(\texttt{checkout};a)))$$

Cet aspect commence dans l'état a et attend que le programme de base émette un événement début de session `login` (les autres événements sont ignorés). Quand l'événement de début de session a lieu, l'aspect passe dans l'état a' et le programme de base qui était en pause reprend son exécution. L'aspect attend maintenant un événement de mise à jour `update` ou de fin de session `checkout` (encore une fois les autres événements sont ignorés). Si la mise à jour a lieu en premier, l'action associée `skip log` fait sauter au programme de base la commande de mise à jour (`skip` est un mot clé) puis l'aspect exécute la commande de journalisation `log` afin de garder une trace des mises à jour à redéclencher hors session. Ensuite, l'aspect revient dans l'état a' et le programme de base reprend son exécution. Si le passage en caisse a lieu en premier, l'aspect retourne dans l'état a et le programme de base reprend son exécution. Puisque les mises à jour sont ignorées dans l'état a, les mises à jour hors session sont tout simplement exécutées, alors que celles qui ont lieu pendant les sessions (état a') ne sont pas exécutées mais journalisées.

Notre approche fournit essentiellement des mécanismes pour synchroniser ou plus précisément entrelacer l'exécution du programme de base et celle de l'aspect à la façon des coroutines. La différence est ici que l'aspect est passif en écoute (lorsqu'un événement qui ne l'intéresse pas a lieu, le programme de base reprend simplement son exécution, il n'y a pas d'interblocage) mais actif en action (un aspect peut faire sauter au programme de base sa commande courante). Afin de réaliser ce protocole d'entrelacement entre programme de base et aspect, l'application de base doit être instrumentée avec différents événements administratifs comme illustré par le code grisé de la Figure 5.2. Le schéma d'instrumentation du programme de base est régulier : il commence par émettre un événement (par exemple `eventB_update`) pour l'aspect, puis il attend que l'aspect qui possiblement entrelace la première moitié de son action lui ordonne soit de continuer son exécution (`proceedB_update`), soit d'ignorer sa commande courante (`skipB_update`). L'intrumentation continue dans les deux cas par une paire d'événements (`proceedE_update`,`eventE_update`) ou bien (`skipE_update`,`eventE_update`) afin que le

programme de base fasse une nouvelle pause dans son exécution et permettent ainsi à l'aspect d'entrelacer l'exécution de la seconde partie de son action.

```
1 Server = login → InSession
2     | eventB_update →
3         ( proceedB_update →  update → proceedE_update → eventE_update → Server
4         | skipB_update → skipE_update → eventE_update → Server) ,
5 InSession = checkout → Server
6     | eventB_update →
7         ( proceedB_update →  update → proceedE_update → eventE_update → InSession
8         | skipB_update → skipE_update → eventE_update → InSession) ,
9     | browse → InSession.
```

FIGURE 5.2 : Programme de base instrumenté en FSP

La définition de l'aspect doit être instrumentée de façon complémentaire comme illustré dans la Figure 5.3 pour l'aspect qui garantit la cohérence des prix. Ici encore, le code grisé est le code administratif introduit par l'instrumentation. Dans l'état a les lignes 2 et 3 modélisent une boucle d'attente active non bloquante qui permet d'ignorer les événements et de rester dans l'état a. La ligne 2 est plus complexe que la 3 car pour la commande `update` le programme de base attend un ordre explicite d'exécuter ou d'ignorer la commande (dans ce cas de l'exécuter hors session). De même dans l'état a' la ligne 6 code une attente active non bloquante alors que la ligne 4 ordonne au programme de base d'ignorer les mises à jour pendant les sessions.

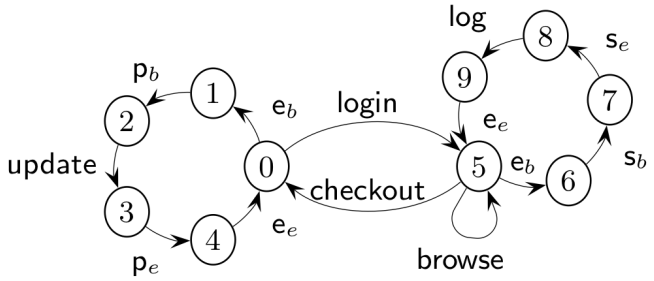
```
1 a = ( login → a'
2     | eventB_update → proceedB_update → proceedE_update → eventE_update → a
3     | checkout → a | browse → a ),
4 a' =( eventB_update → skipB_update → skipE_update → log → eventE_update →  a'
5     | checkout → a
6     | browse → a' | login → a' ).
```

FIGURE 5.3 : Aspect de cohérence en FSP

Le tissage d'un programme de base et d'un aspect se modélise alors simplement par la composition parallèle des deux définitions instrumentées. Le résultat est détaillé dans la Figure 5.4. On voit que hors session les mises à jour `update` ont lieu mais sont remplacées par la journalisation `log` en session.

5.3 Flot de contrôle concurrent

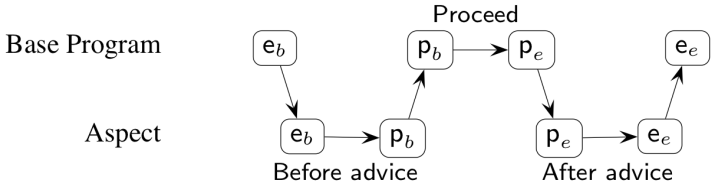
Dans la section précédente nous avons montré comment encoder un tissage d'aspect sensible au flot de contrôle séquentiel. Cet encodage repose sur un protocole qui entrelace le programme de base et l'aspect comme ci-dessous



where the following abbreviations are used:

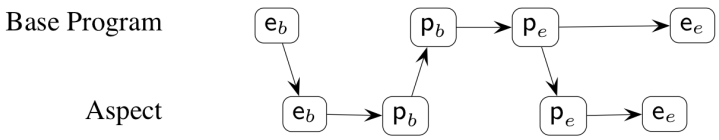
e_b	=	eventB_update
e_e	=	eventE_update
p_b	=	proceedB_update
p_e	=	proceedE_update
s_b	=	skipB_update
s_e	=	skipE_update

FIGURE 5.4 : Application de vente en ligne tissée par un aspect de cohérence



où les flèches représentent le flot de contrôle entre différents événements administratifs (e_b pour événement before, p_b pour procède before, etc.).

Notre encodage a été réalisé dans un formalisme dédié à la concurrence, il est donc très facile d'effacer des synchronisations pour introduire de la concurrence dans le programme tissé. Par exemple, lorsque l'événement e_e (ici `eventE_update`) qui modélise la fin de l'action d'un aspect est effacé du programme de base, alors la seconde moitié de l'action de l'aspect est exécutée en parallèle avec la reprise de l'exécution du programme de base (qui n'attend plus la fin de l'action). Ceci est modélisé par les deux flèches (flots de contrôle) les plus à droite ci dessous



Dans notre exemple, cela permet par exemple à l'aspect de journaliser `log` la mise à jour annulée sans freiner le programme de base. On obtient alors le programme tissé de la Figure 5.5. En particulier, on voit que dans l'état 7 alors que la journalisation n'a pas encore eu lieu il est déjà possible de feuileter le catalogue `browse`.

On peut rendre le programme tissé encore plus concurrent. En effet jusqu'à présent lorsque le programme de base exécute sa commande originale, l'action de l'aspect est en pause. En effaçant

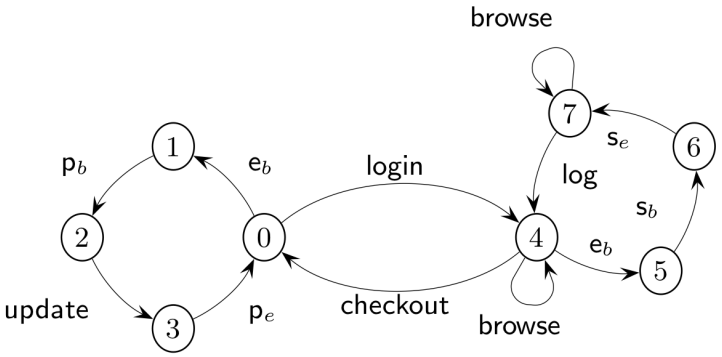


FIGURE 5.5 : Application de vente en ligne tissée par un aspect de cohérence et concurrente

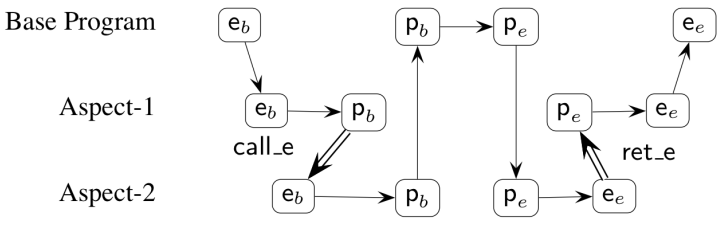
les événements `updateE_proceed` et `updateE_skip` la seconde moitié de l'action de l'aspect s'exécute en parallèle avec la commande de base. Par contre, cela n'a pas de sens d'effacer des synchronisation plus avant puisque le programme de base doit attendre que l'aspect lui ordonne d'exécuter ou de sauter sa commande originale.

5.4 Opérateurs de composition

Considérons un second aspect qui permet d'introduire de la sureté

$$\mu a''.(\texttt{update} \triangleright \texttt{rehash proceed backup}; a'')$$

à chaque fois qu'une mise à jour est effectuée (c.a.d. qu'un administrateur publie une nouvelle version du site), une base de données des liens est recalculée avant publication, puis après publication un archivage de la base de donnée est réalisé. Afin de tisser l'application de vente en ligne avec l'aspect de cohérence et l'aspect de sureté nous avons introduit un opérateur de composition d'aspects `Fun`. Il se comporte à la manière d'AspectJ en composant fonctionnellement les actions de deux aspects : ainsi l'action du premier aspect ne déclenche pas l'exécution (ou non) de la commande du programme de base mais l'action du second aspect, qui à son tour elle même déclenche (ou non) l'exécution de la commande du programme de base. Ceci est illustré ci-dessous :



Cet opérateur peut être défini formellement comme dans la Figure 5.6. Il utilise à la fois des renomages pour distinguer les différentes occurrences des mêmes événements administratifs et une séquence d'action qui une fois qu'un aspect à décidé d'ordonner d'ignorer une commande (c.a.d. `skip`) propage cette décision.

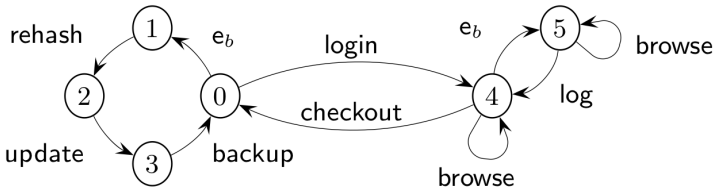
Cet opérateur permet d'ordonner précisément les aspects mais conduit à un programme tissé très séquentiel. Comme illustré par le programme tissé ci dessous, le cycle de gauche qui met à jour et entretien la base doit être entièrement terminé avant que l'application puisse être utilisée.

$\|\mathsf{FunArg}_1 = \mathsf{a}/\{\mathsf{call\_e/proceedB\_e},\ \mathsf{ret\_e/proceedE\_e},\ \mathsf{skipB\_e1/skipB\_e},\ \mathsf{skipE\_e1/skipE\_e}\}.$

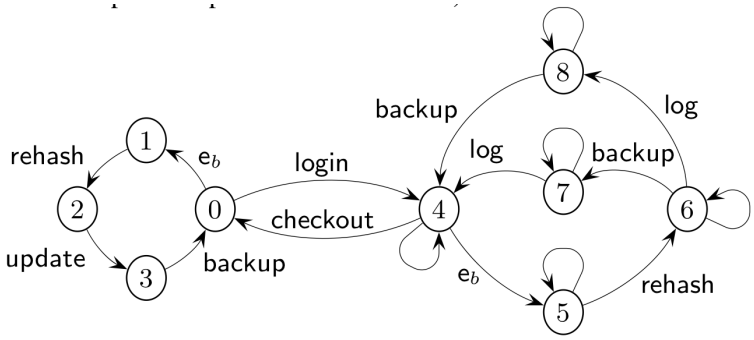
$\|\mathsf{FunArg}_2 = \mathsf{a''}/\{\mathsf{call\_e/eventB\_e},\ \mathsf{ret\_e/eventE\_e},\ \mathsf{skipB\_e2/skipB\_e},\ \mathsf{skipE\_e2/skipE\_e}\}.$

$$\begin{aligned}\mathsf{Fun} \quad = \ & (\ \mathsf{skipB\_e1} \rightarrow\ \mathsf{skipB\_e} \rightarrow\ \mathsf{skipE\_e} \rightarrow\ \mathsf{skipE\_e1} \rightarrow\ \mathsf{Fun} \\ & |\ \mathsf{skipB\_e2} \rightarrow\ \mathsf{skipB\_e} \rightarrow\ \mathsf{skipE\_e} \rightarrow\ \mathsf{skipE\_e2} \rightarrow\ \mathsf{Fun}).\end{aligned}$$

FIGURE 5.6 : L'opérateur de composition `Fun` pour l'événement e



Nous avons proposé un autre opérateur de composition `ParAnd` qui exécute les deux actions de ses aspects en parallèle. La commande originale du programme de base ne sera exécutée que si les deux actions ordonnent son exécution. Cet opérateur introduit beaucoup plus de parallélisme que le précédent et permet dans l'exemple du site de vente en ligne d'obtenir un programme tissé dans lequel la sauvegarde `backup` de la base de données peut être exécutée en parrallèle avec la journalisation `log` des mises à jour ignorées, ainsi que la consultation `browse` du catalogue. Ceci est illustré ci-dessous par les différents entrelacements de ces trois actions (pour ne pas surcharger la figure, l'action `browse` qui est possible dans de nombreux états est représentée par les transitions sans label).



Nous ne reproduisons pas ici la définition formelle de `ParAnd`, ni les schémas d'instrumentation du programme et des aspects qui sont détaillés dans l'article original [22]. Le lecteur pourra aussi y trouver une description d'un prototype Java permettant de reproduire les exemples.

5.5 Conclusion

Peu de travaux sur la programmation à aspects ont été dédiés à la concurrence. AspectJ étendu avec des *trace match* [ACH05] propose le mot clé `perthread` pour créer une instance d'aspect (attributs) pour chaque fil. En l'absence de ce mot clé, un aspect peut sélectionner des points de programmes dans différents fils, et l'action finale de l'aspect sera exécutée dans le dernier fil

sélectionné. Les algèbres de processus avaient déjà été utilisées pour décrire la sémantique d'un tisseur mais uniquement dans un cadre non concurrent [A01].

Notre traduction du tissage d'aspects sensibles au flot de contrôle séquentiel dans un cadre formel pour la programmation concurrente s'est avérée fructueuse. En particulier, elle a permis de montrer comment explorer de manière non adhoc (en considérant les différentes synchronisations à supprimer) les opportunités de parallélisme. Le vérification de modèles LTSA [MK99] nous a été utile lors du développement du processus d'encodage pour détecter quand notre traduction n'était pas correcte et créait un interblocage. Nous avons aussi pu vérifier formellement par exemple, que nos opérateurs de composition étaient associatifs, et au cas par cas nous avons pu vérifier que deux actions sont toujours exécutées dans un certain ordre. La génération automatique de tests de raffinement par rapport à des spécifications de plus haut niveau (par exemple qui ne s'encombrent pas d'événements administratifs) reste à explorer.

Chapitre 6

Flot dynamique de contrôle distribué

Sommaire

Les résultats décrits dans ce chapitre sont basés sur deux séries de travaux réalisés dans le cadre de nombreuses collaborations. Ces travaux montrent comment la programmation par aspects peut être étendue à un cadre distribué qui apporte de nouveaux défis. Une première série de travaux concerne la notion de flot de contrôle dans un langage de programmation par aspects avec distribution explicite. Une seconde série de travaux concerne l'exploration de la notion de portée pour contrôler l'application d'aspects dans un cadre distribué.

6.1 Aspects avec distribution explicite

Je n'ai pas participé à la conception initiale de Awed [NSV06] (un langage de programmation par aspects avec distribution explicite), mais j'ai contribué à différentes extensions et applications décrites ci dessous. Avant d'aborder ces travaux, je présente la version originale de Awed.

Après des travaux sur la programmation par aspects dans un cadre concurrent, il était logique d'étudier la programmation par aspects dans un cadre distribué. Au delà de la motivation théorique pour proposer Awed, une motivation pratique a été identifiée en étudiant JBoss. Ce serveur d'applications est bâti sur J2EE, la plate-forme intergicielle de Java, il offre (entre autres) un mécanisme de réplication de caches distribués afin d'améliorer les performances en stockant proches des clients les données dans une structure d'arbre (classe `TreeCache`). La cohérence des données répliquées est garantie par un protocole transactionnel reposant sur un mécanisme d'intercepteurs de JBoss. Ces préoccupations ne sont pas modulaires comme illustré dans la Figure 6.1 : le code de la classe `TreeCache` à gauche et du package des intercepteurs à droite

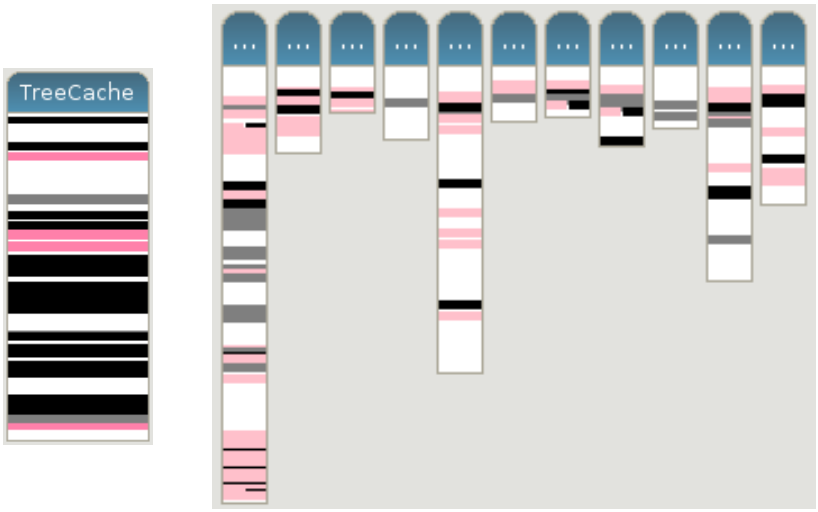


FIGURE 6.1 : Préoccupations non modulaires dans JBoss

```
1 Ad ::= [syncex] Pc { Stmt* }
2 Pc ::= Pc || Pc | Pc && Pc | !Pc | call(Msig)
3       | host(Group)
4       | on(Group)
5 Group ::= Host*
6 Host ::= localHost | jpHost | Ip:Port
7 Stmt ::= addGroup(Group) | removeGroup(Group) | ...
```

FIGURE 6.2 : Un langage d'aspects avec distribution explicite (Awed)

a été colorié en noir pour la réplication, en gris sombre pour les transactions et en gris clair pour les appels aux méthode de `TreeCache`. Plus précisément, la classe `TreeCache` inclut 188 méthodes et 1741 lignes de code : le code concernant la réplication est éparpillé dans 196 LOC ; celui concernant les transactions dans 228 lignes. La situation est similaire dans les 9 classes et 1236 lignes des intercepteurs : le nombres de lignes concernant la réplication, les transactions, et les appels à `TreeCache` sont respectivement 30, 41, et 73.

Dans cette architecture la modification de la politique de réplication (par exemple pour ne répliquer que sélectivement certains objets) est complexe car non modulaire. L'étude menée a consisté à proposer un langage de programmation par aspect avec distribution explicite afin de rendre modulaire les préoccupations de réplication et de transaction, puis de modifier la politique de réplication.

Le langage de Awed ressemble à AspectJ mais l'étend avec différentes notions de distribution. Un extrait de la grammaire de Awed dans la Figure 6.2 met en évidence ces notions. Notamment, une action par défaut asynchrone peut être qualifiée de synchrone avec syncex. Le langage de point de coupe permet de sélectionner des points de jointures locaux à un groupe (ensemble d'hôtes) avec host(Group) et de préciser qu'une action doit s'exécuter ailleurs avec on(Group). Une action peut contenir des instructions qui ajoutent ou retirent l'hôte qui l'exécute à un groupe. Ce langage possède d'autres mécanismes que nous ne détaillons pas ici (par exemple des mots clés pour contrôler le déploiement et l'instanciation des aspects).

Ce langage a permis d'exprimer la réplication de cache comme un simple aspect (Figure 6.3).

```
1 all aspect CacheReplication {
2   pointcut cachePcut(Object key, Object o):
3     call(* Cache.put(Object,Object)) && args(key,o) && !on(jphost);
4   before(Object key, Object o): cachePcut(key,o) {
5     Cache.getInstance().put(key, o); }
6 }
```

FIGURE 6.3 : La réplication de cache comme un aspect distribué

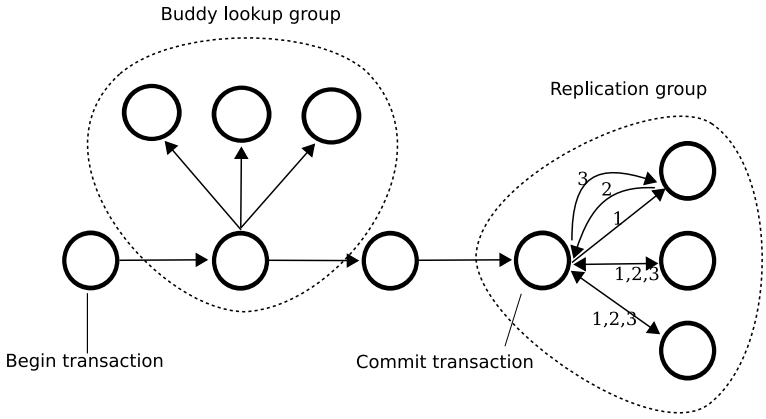


FIGURE 6.4 : Transaction et réplication dans JBoss

L'aspect CacheReplication est déployé sur tous (**all** à la ligne 1) les hôtes. Son point de coupe CachePcut (ligne 2 et 3) sélectionne les appels distants (!on(jphost)) à la méthode Cache.put qui écrit le cache et son action (ligne 4 et 5) réplique cette action (localement).

Je ne reproduis pas ici les autre résultats de l'utilisation de Awed dans JBoss. Par contre, on peut noter que cette étude a mis en évidence une forte régularité dans le code concernant le protocole pour gérer les transactions. Une première extension à laquelle j'ai contribué a consisté à proposer une extension langage à Awed pour exprimer cette régularité.

6.1.1 Patrons invasifs

Les travaux décrits dans cette section ont été publiés dans [21]. La figure 6.4 présente une vision haut-niveau à base de patrons de l'architecture de JBoss concernant les transactions et les réplications dans les caches. Une transaction est déclenchée par un appel de méthode représenté par le premier nœud du patron. Quand une valeur particulière n'est pas dans le cache, la valeur est cherchée dans les caches d'un groupe de nœuds voisins (buddy), comme illustré par les trois arcs verticaux du second nœud. A la fin de la transaction, le cache engage un protocole en deux phases. Des messages de préparation (numérotés 1) sont envoyés à des voisins (replication), qui répondent un accord/désaccord (numérotés 2). Finalement, en fonction de (dés)accords reçus, des messages de confirmation ou d'annulation (numérotes 3) sont envoyés aux voisins.

Ce schéma d'interaction distingue deux groupes de nœuds : ceux concernés par la recherche et ceux concernés par la réplication. Il utilise des patrons comme ceux définis dans la figure 6.5 où un nœud représente un hôte et un arc une communication. Informellement, `pipe` modélise une chaîne de calculs, `farm` la parallélisation de calculs et `gather` la collecte de resultats pour synthèse.

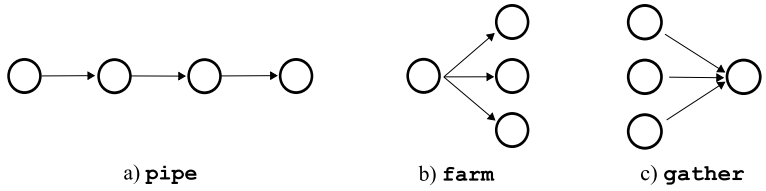


FIGURE 6.5 : Patrons d'interaction

```
1 P ::= patternSeq G1 A1 G2 A2 .... Gn
2 G ::= H* P*
3 A ::= aspect { around((H, id*)*):Pc SourceAdvice [sync] TargetAdvice }
```

FIGURE 6.6 : Un langage de patrons invasifs

Malheureusement la situation est moins claire dans le code de JBoss. Bien que JBoss utilise conceptuellement une structure à base de patrons, la mise en œuvre ne permet pas d'appliquer pratiquement les patrons à cause de l'éparpillement et de l'entrelacement des codes correspondants aux différentes fonctionnalités. Les interactions entre les fonctionnalités de base, les transactions, les réplications ne sont pas représentées explicitement dans le code. Un état peut se retrouver stocké par morceau dans plusieurs classes. Des préoccupations liées à la distribution nécessitent des communications entre différents groupes d'hôtes dont l'intersection n'est pas vide.

Dans cette situation, un style de programmation à base de patrons doit permettre la définition de patrons qui incluent l'accès aux données dont il dépend même si ces données ne sont pas locales, ainsi qu'à appliquer un patron à de nombreux endroits dans un programme. La non modularité (mono localité) des états d'exécution fait de la programmation par aspect un candidat idéal pour soutenir ce style de programmation. Nous étendons Awed avec différentes notions afin de permettre la programmation de patrons invasifs (Figure 6.6). Un pattern invasif P (ligne 1) est défini par une séquence de groupes G reliés par des aspects A. Un groupe (ligne 2) est constitué d'hôtes H et de groupes (ceux inclus dans P). Un aspect (ligne 3) définit un point de coupe Pc pour sélectionner des points de jointures sur le groupe à sa gauche dans le patron. Les variables H et **id** permettent de récupérer l'identité de l'hôte et de collecter des valeurs. Ces valeurs sont utilisées dans des actions asynchrones SourceAdvice exécutées sur le groupe à gauche de l'aspect et des actions possiblement synchrones TargetAdvice exécutées sur le groupe à droite de l'aspect dans le patron.

Ce langage de patrons invasifs permet de définir de façon modulaire les préoccupations de transaction et réplication dans JBoss comme illustré par la Figure 6.7. Ce patron est une composition imbriquée de quatre patrons. D'abord, nous appliquons un patron pipe afin de relier le début de la transaction avec l'opération de réplication, c.a.d. le nœud initial et le groupe final dans la Figure 6.4. Une fois l'engagement rencontré, un patron farm est utilisé pour paralléliser la phrase de préparation du protocole d'engagement à deux phases. Puis un patron gather est utilisé pour collecter les réponses des caches voisins (buddy). Finalement, une fois toutes les réponses reçues un patron farm distribue la décision finale d'engagement ou d'annulation. Le code de la figure considère trois caches répliqués. La réplication peut être déclenchée à partir de n'importe lequel des trois caches. Une fois le nœud déclencheur (h dans l'algorithme) sélectionné l'expression gcaches h représente le groupe de caches moins le déclencheur.

```
1  gCaches = {H1, H2, H3}
2  pipe([h],
3    Atransac,
4    farm(
5      gather(
6        farm([h], Aprepare, sync gCaches [h]),
7        Apresp,
8        [h]),
9      Acommit,
10     gCaches [h])
11   );
```

FIGURE 6.7 : Un patron invasif pour les transactions dans JBoss

Nous ne détaillons pas ici les aspects (et leur actions) de cet exemple. On trouvera dans l'article original la traduction d'un aspect définit dans la grammaire de la Figure 6.6 en Awed ainsi qu'une discussion sur les techniques de mise en œuvre utilisées.

6.1.2 Application à la grille

La section précédente a présenté les patrons invasifs : une extension de Awed pour programmer de façon modulaire des schémas de communication réguliers dans un contexte distribué et non anticipé. Nous avons illustré l'approche sur les caches de JBoss. Dans cette section nous décrivons comment les patrons invasifs peuvent être appliqués à un autre domaine d'application avec des communications régulières : les architectures de grilles [19].

Les grilles sont des architectures puissantes qui permettent d'exécuter des applications de large échelle aussi diverses que des calculs scientifiques ou des système d'informations de grande taille. Ce genre d'architecture qui est composée de multiples fédérations locales fournit aux utilisateurs un environnement hautement hétérogène. Pour dépasser ces problèmes d'hétérogénéité, les architectures de grilles et les applications sont typiquement construites en utilisant des intergiciels adaptés qui permettent de faire le pont entre des infrastructures existantes (souvent à base de composants).

Le développement des applications de grilles en utilisant de tels intergiciels est fréquemment impacté par deux problèmes : des moyens limités pour décrire les topologies et le manque de soutien pour la composition invasive des composants patrimoniaux. Par exemple, les topologies de grilles qui sous-tendent les applications sont souvent seulement définies implicitement à travers les passages de messages ou en utilisant des définitions de bas niveau, telles que les constructeurs de graphes dont les liens avec la grille ont été définis une fois pour toute. Concernant la composition de composants patrimoniaux de grille, il est souvent nécessaire de réécrire de façon significative ces composants, car leur composition nécessite de transmettre des données qu'ils n'exportent pas initialement.

Pour évaluer notre approche dans ce contexte, nous avons considéré le NAS Grid Benchmark. Ce canevas offre un cadre pour la mesure de performance dans les grilles de calcul en fournissant les mécanismes principaux : l'exécution de processus distribués communicants. Ce canevas est couramment utilisé pour tester des outils de programmation et des compilateurs optimisants. Afin d'illustrer le problème de la mise en œuvre d'algorithmes distribués sur les grilles et d'évaluer notre solution nous avons étudié un service fondamental des grilles : la récupération d'erreur

par points de reprise globaux. Ce service est essentiel pour les applications de calculs à grande échelle afin de minimiser les couts liés aux échecs du système qui arrêtent le calcul. Ce service sauvegarde périodiquement l'état de l'application. Une sauvegarde globale est composée de sauvegardes locales et un algorithme assure la cohérence des états. En conséquence un tel service doit inspecter et modifier de manière invasive les calculs locaux. Notre technique à base d'aspects permet de mettre en œuvre ce service en modifiant de façon transparente les interactions entre composants avec un impact négligeable sur les performances mémoires et autres. De plus, elle permet de définir des algorithmes pour la grille de façon déclarative et d'éviter la modification manuelle de code pénible et source d'erreurs.

6.1.3 Sémantique formelle

Nous venons de discuter des patrons invasifs et de leur mise en œuvre en utilisant Awed. La traduction de ces patrons invasifs en code Awed n'est pas triviale. En nous inspirant de nos précédents travaux sur la programmation par aspects avec flot de contrôle concurrent nous avons utilisé le langage FSP et le testeur de modèle LTSA pour définir une sémantique formelle de Awed ainsi que de la traduction des patrons invasifs [35]. Ce cadre nous a ainsi permis de tester des propriétés de vivacité et de sureté. Par exemple, pour l'aspect Awed présenté dans la section précédente qui réalise la réplication de caches distribués, nous avons pu montrer que la propriété "le chargement d'un cache local est suivi d'une réplication sur le cache distant" n'est pas vérifiée. En effet, dans ce cas LTSA a retourné une courte trace d'exécution qui viole la propriété testée. Les actions distantes étant asynchrones nous n'avons pas de garanti sur l'ordre d'exécution de ces deux actions. Si une telle propriété est nécessaire, un point de synchronisation explicite doit être introduit dans les actions et la propriété est alors vérifiée. De la même manière nous avons pu tester des propriétés de la mise en œuvre des patrons de conception. Nous nous sommes notamment rendu compte que les "vagues" de calculs définies par les séquences de patrons nécessitent plus de synchronisation que nous ne le pensions afin que la fin d'une vague et le début de la suivante ne s'exécutent jamais en parallèle. Cette modélisation formelle nous a permis d'affiner la mise en œuvre et la sémantique de Awed et des patrons.

6.1.4 Points de coupe causaux

Dans un contexte séquentiel la succession de deux événements peut être interprétée comme un lien causal. Dans un contexte concurrent ou distribué il peut ne pas y avoir de lien entre deux événements successifs. Les notions de temps et d'horloges logiques, introduites par Lamport, permettent d'identifier les causalités dans un système distribué.

Nous avons utilisé ces notions afin de définir des points de coupe séquentielle dans un contexte distribué. Le langage de coupes de Awed a été étendu avec des construction causales [20] : l'opérateur seqCausal définit des points de coupe dont les points de jointures sont causalement reliés. Cette construction permet de détecter les séquences causales de points de jointure. Si la cause est observée après la conséquence, par le jeux des exécutions distribués, la séquence ne sera pas repérée. Dans ce cas un second opérateur seqCausalOrder qui réordonne si nécessaire les points de jointure peut être utilisé. Ces extensions ont été mises en œuvre dans Awed en utilisant des vecteur d'horloges (pour détecter la causalité) et des queues (pour ne pas perdre de causalité). Nous avons utilisé ces nouvelles fonctionnalités afin d'assister à la mise au point d'applications. En particulier, nous avons montré comment programmer un aspect de détection des interblocages pour JBossCache. Les performances de la mise en œuvre ont été évaluées et elle se sont avérées ne pas introduire de surcout significatif dans JBossCache.

Nous avons mené un travail proche dans le cadre de JavaScript pour des applications Web [1]. Ici encore, je n'ai pas participé à la conception d'AspectScript un langage d'aspect pour Java Script mais j'ai contribué à l'étendre avec des notions de causalité pour obtenir la bibliothèque Weca permettant de la définition de coupes séquentielles causales dans les applications Web. Les mêmes ingrédients ont été utilisés : automates pour la définition de coupes séquentielles, vecteurs d'horloges pour détecter les causalités et queues pour réordonner causalement les points de jointures. Des stratégies de gestion des points de jointures distribués ont été introduites afin de pouvoir traiter différents scénarios : définition de services web composites (utilisation de queues pour assurer la logique applicative), traitement de flux vidéo en temps réel (pas d'utilisation de queue mais rejet des points de jointure en retard pour garantir la cohérence temporelle), visualisation de forums distribués (pas d'utilisation de queue mais rejet des points de jointure en avance pour garantir la cohérence des fils de discussion). Weca a aussi permis de prendre en compte la causalité entre tweets pour définir une notion précise de popularité. Ces différentes expérimentations ont permis de vérifier que le coût de Wecca était réaliste dans le cadre d'applications web interactives et que les solutions apportées étaient plus générales et flexibles que la concurrence.

6.2 Portée

Des notions de portée sont apparues dans la programmation par aspects dès AspectJ. Par exemple le point de coupe !within(A) indique une portée statique et doit être inclus dans les points de coupe de l'aspect A afin d'éviter qu'il ne se coupe lui même, ou encore cflow permet de parler de portée dynamique en sélectionnant des points de jointure dans la portée d'un appel de méthode. Dans le cadre d'une application distribuée les opportunités de contrôler finement la portée sont encore plus critiques.

6.2.1 Déploiement

Le déploiement dynamique d'aspects offre une grande flexibilité et un potentiel de réutilisation, mais nécessite un contrôle adapté de leur portée. Les questions de portée sont particulièrement cruciales dans un contexte distribué : un traitement adéquat des portées distribuées est nécessaire pour permettre la propagation d'instances d'aspects à travers les frontières des sites et pour éviter les incohérences dues à la diffusion non intentionnelle de données et de calculs dans un système distribué.

C'est dans ce cadre que nous nous sommes intéressés au déploiement des aspects [15, 4]. Nous avons isolé un scénario d'application distribuée basé sur un service de réservation pour voyageurs et des aspects de facturation, confidentialité et statistique ainsi que des comportements de propagation, d'activation et d'activation par transaction des différents aspects. Nous avons étudié les différentes solutions existantes et identifié leurs limitations (par exemple solutions manuelles, ou limitées à des portées statiques).

Nous avons alors proposé un premier modèle de portée pour des aspects non distribués. Ce modèle AspectScheme est basé sur le langage fonctionnel Scheme enrichi avec des Aspects. La sémantique opérationnelle du langage définie comme un interprète prend en compte un environnement d'aspects et leurs stratégies de déploiement. Une stratégie de déploiement est une paire de fonctions booléennes qui pour chaque point de jointure spécifient si l'aspect doit se propager dans la pile d'appel, et si l'aspect doit se propager à la construction d'une fermeture.

Ce modèle est étendu avec des primitives de programmation distribuée (essentiellement des exportations et liaisons de fonctions afin de fournir des appels distants synchrones). Les aspects

et leur déploiement sont étendus en conséquence : déploiement global sur un ensemble d'hôtes, passage des aspects par copie ou par référence lors d'appels distants, séparation de l'appel et de l'exécution d'une fonction qui peuvent être sur deux hôtes différents distants, séparation de la définition initiale d'une valeur de sa copie lors d'un appel distant. Ces nouvelles stratégies peuvent être prises en compte en étendant le modèle/l'interprète avec de nouveaux points de jointures et un nouvel environnement d'aspects (ceux qui sont utilisés lorsqu'une valeur est copiée).

Ce travail a permis de donner une définition opérationnelle et raisonnablement concise d'un langage distribué avec déploiement dynamique d'aspects avec portée. Cette définition se base sur des notions intrinsèques de distribution et a permis de répondre aux différents besoins identifiés dans le scénario de réservation distribuée de voyages. Ce cadre a aussi permis d'exprimer les différentes solutions existantes et de mieux les comparer.

6.2.2 Membranes

Pour nous abstraire d'un modèle de programmation particulier (par exemple le fonctionnel de Scheme dans la section précédente) nous avons proposé un modèle très général [33] de programmation par aspects et de contrôle des portées basé sur les membranes.

Une membrane suit l'analogie biologique et forme une barrière entre un calcul qu'elle contient et le monde extérieur. Les membranes peuvent être imbriquées hiérarchiquement arbitrairement. On peut aussi lier leurs canaux de communication ce qui permet par exemple à un calcul d'en observer un autre à la manière d'un aspect. Nous avons ainsi montré comment différentes topologies pouvaient reproduire différentes notions de portées.

Une membrane peut être active et filtrer, voire transformer ce qui la traverse. Nous avons aussi montré comment différentes membranes permettaient de restreindre ou de composer des calculs à la manière des aspects.

Pour rendre nos définitions concrètes nous avons défini des membranes dans un calcul de processus existant. Cela a permis de donner une sémantique formelle à notre approche, tout en ouvrant la voie vers une possible mise en œuvre basée sur ce calcul de processus.

6.3 Conclusion

Peu de travaux avaient été explicitement dédiés à la programmation à aspect dans un cadre distribués. Les plus connexes sont des travaux autour de la notion de portée (ce que peut "voir" un point de coupe, ce qui est "visible" d'une action) qui dans un cadre distribués prend un sens exacerbé et dynamique (sur quels hôtes, à partir de quels hôtes, etc.). On peut notamment citer CeasarJ [AGM06] et AspectJ Scheme [DT06] qui soutenaient la portée d'aspects déployés dynamiquement.

Un cadre distribué est naturellement propice à la notion de topologie et parfois de topologie régulière. Lorsque l'on parle de grille et de programmation à grande échelle on retrouve une forme de régularité et de quantification (par exemple dans patrons d'interactions) chères aux aspects qui fait qu'ils ont un rôle à jouer.

Un cadre distribué est naturellement propice à la notion d'observation (même si cette observation n'est que partielle) et s'intègre bien avec des notions de points de coupe. On a pu voir que les notions de causalité et de points de coupe se marient bien.

Enfin un cadre distribué introduit de nouvelles opportunités de localité et de portée. Nos travaux ont tenté d'embrasser la richesse et la complexité d'un tel cadre en général. D'autres tentatives pourraient être menées sur des modèles de distributions plus restreint.

Chapitre 7

Flot dynamique de données

Sommaire

Dans la section 3.3.2 on a vu qu'en AspectJ l'opérateur `cflow` permet de définir un point de coupe qui relie des points de jointure et permet de passer du contexte entre ces points. On a aussi montré que cet opérateur peut être simplement défini en réifiant puis inspectant la pile d'appels. Le passage de contexte a donc lieu dans ce cas vers le bas : d'un appelant vers un appelé.

Dans ce chapitre, nous nous focalisons sur le passage de contexte au delà de l'inspection de pile. La section 7.1 montre comment la réification de l'arbre d'appels permet aux points de jointure de survivre après le retour d'une méthode et donc relier des points pour passer du contexte vers le haut : d'un appelé vers un appelant. La section 7.2 montre comment la réification des points de jointure dans une base de faits et la définition de leurs relations en Datalog permet de relier des points pour passer du contexte du passé vers le présent.

7.1 Path : flots ascendants

Cette section est basée sur un travail publié à la conférence GPCE [25]. Il formalise et généralise le langage de point de coupe d'AspectJ.

7.1.1 Un langage généralisé de points de coupe pour les arbres d'appels

AspectJ permet de définir des points de coupe atomiques concernant les méthodes (appel et retour) et les champs (lecture et écriture). Dans une interprétation ensembliste, un point de coupe atomique dénote un ensemble de points de jointure. Il permet aussi de composer ces définitions entre elles avec des opérateurs logiques : `&&`, `||`, `!`. Dans une interprétation ensembliste ces opérateurs peuvent être compris comme l'intersection, l'union et le complémentaire. L'opérateur `cflow` prend un point de coupe en paramètre et permet de définir tous les points de jointure pendant l'exécution d'une méthode. (Il existe aussi une variante `cflowBelow` qui exclue le point courant). Comme montré à la fin de la section 3.3.2 des `cflow` peuvent être imbriqués. Une grammaire idéalisée de ce langage pourrait être :

```
PcD ::=
        points de coupe atomiques...
      | PcD && PcD
      | PcD || PcD
      | ! PcD
      | cflow PcD
      | cflowBelow PcD
```

Nous avons généralisé ce langage, tout en conservant sa régularité, en introduisant de nouveaux opérateurs :

- `filter` généralise les points de coupe atomiques en prenant en paramètre un prédicat arbitraire sur les points de jointure.
- `head` réduit un ensemble de points de jointure à son premier (au sens temporel) élément.
- `tail` est le complémentaire de `head` : il ignore le premier point de jointure d'un ensemble.
- `from` sélectionne tous les points de jointure qui se trouvent (dans la trace/le temps) après le point de jointure qu'il prend en paramètre.
- `path` sélectionne tous les points de jointure qui se trouvent dans l'arbre des appels sur le chemin entre le point de jointure qu'il prend en paramètre et le point de jointure racine de l'arbre (le retour à `main`).
- `pathAbove` est une variante de `path` qui exclut le point courant.

La grammaire complétée est :

```
PcD ::=
        filter Predicate
      | PcD && PcD
      | PcD || PcD
      | ! PcD
      | cflow PcD
      | cflowBelow PcD
      | head PcD
      | tail PcD
      | from PcD
      | path PcD
      | pathAbove PcD
```

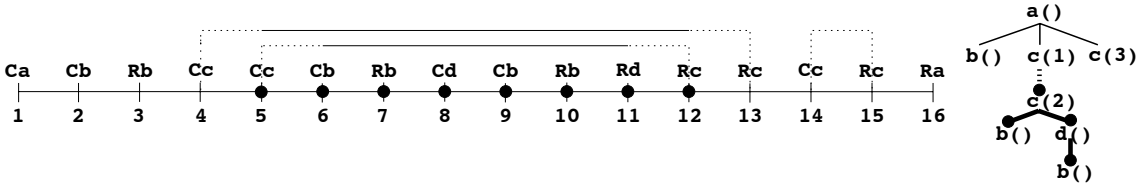


FIGURE 7.1 : Sémantique de `cflowBelow(C)`

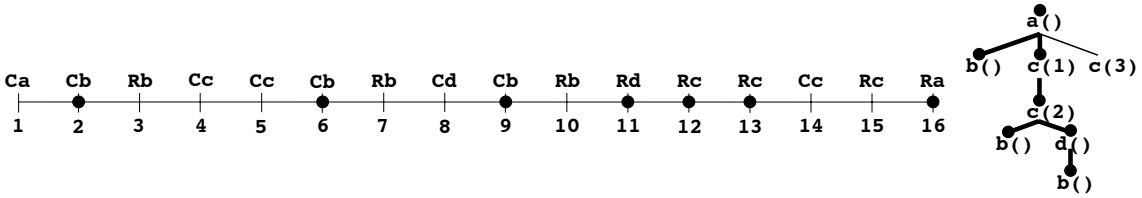


FIGURE 7.2 : Sémantique de `path(CallB)`

Une sémantique formelle de ce langage a été définie en Haskell. Cette sémantique prend la forme d'une fonction

```
eval :: Trace -> PcD -> [JoinPoint]
```

qui à partir d'une liste des points de jointure (`Trace`) et d'une définition de point de coupe `PcD` retourne les points de jointure sélectionnés par le point de coupe.

Par exemple, la Figure 7.1 montre cote à cote la trace d'exécution et l'arbre d'appel correspondant (pour chaque nœud de l'arbre qui représente une fonction on retrouve dans la trace deux points de jointure qui correspondent à son appel et son retour). Les points de jointure sélectionnés par `cflowBelow(C)` correspondent aux points de jointure pendant l'exécution de la fonction `C` (sauf l'appel et le retour à `C` comme spécifié par `Below`). On voit que les deux intervalles imbriqués dans la trace correspondent à un sous arbre.

Dans le cas de `path(CallB)` la Figure 7.2 montre que certains point dispersés dans la trace sont sélectionnés. La relation entre ces points est plus évidente dans l'arbre d'appels où l'on voit que tous ces points sont les points entre un appel à `B` et la racine de l'arbre.

Attention toutefois, seuls les points de jointure correspondant à un retour sont sélectionnés, il est en effet impossible de savoir en général quand `D` est appelée, qu'un appel à `B` aura lieu avant son retour. Par contre, une fois `B` appelée et revenue on peut sélectionner les points au dessus (sur le chemin du retour jusqu'à la racine). En d'autres termes, la fonction `eval` doit pouvoir s'évaluer incrémentalement : pour chaque nouveau point de jointure dans la trace, elle doit pouvoir décider si ce point est sélectionné par le point de coupe sans aller voir le point suivant. En effet, la décision d'exécuter une action doit être locale et ne peut présager du futur (voir la section 7.1.3 plus bas).

7.1.2 Passage de contexte généralisé

Notre langage généralisé de coupes permet de définir des points de coupes utiles et de passer du contexte. Considérons un système d'allocation de services. On se focalise sur trois classes principales : `Pool` qui groupe un ensemble de clients, `Caller` qui représente un client et `Worker` qui représente un service. Ces classes mettent en œuvre l'algorithme suivant : la méthode `Pool.doAllCallers` itère jusqu'à ce qu'un groupe soit vide en appelant répétitivement la méthode `Caller.doService`, qui alloue des services et appelle `Worker.doTask`. Cette architecture est illustrée par l'arbre d'appels dans la Figure 7.3.

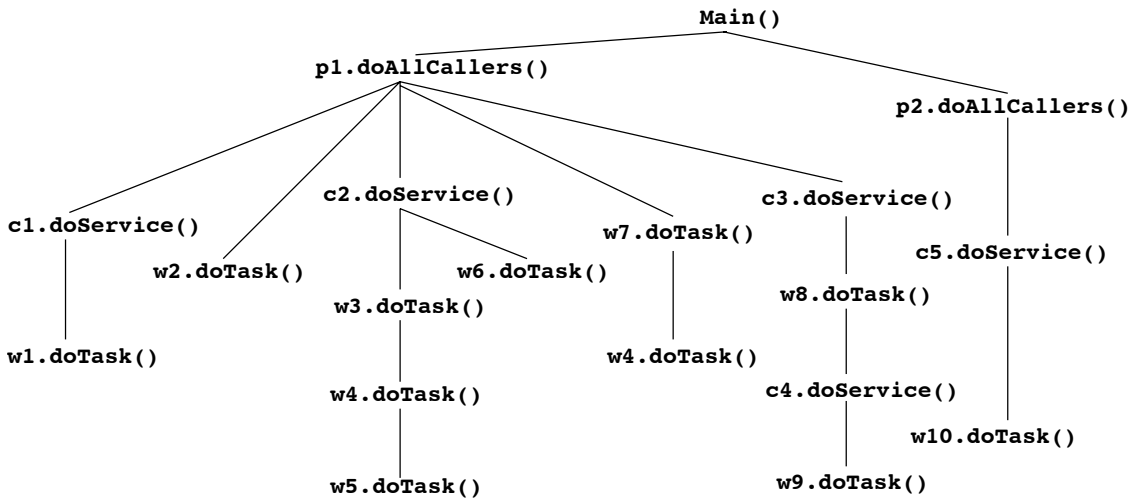


FIGURE 7.3 : Arbre d'appels d'une application à base de clients `Caller` et services `Worker`

Cet arbre ne détaille pas les appels intermédiaires (par exemple l'appel à un allocateur de service qui finalement appelle `Worker.doTask`). Dans cet exemple, il y a deux groupes de clients. Typiquement, comme illustré par la branche la plus à gauche, un groupe appelle un client, qui appelle un service. Un groupe peut aussi directement appeler un service comme dans la seconde branche pour réaliser une tache administrative. Un service peut déléguer sa tache à un autre service comme dans la troisième branche. Dans un telle chaine de délégation, le `doTask` le plus haut est qualifié de principal (par exemple `w3`) et celui le plus bas (qui assure effectivement le service) est qualifié d'exécutant (par exemple `w5`). Nous supposons que tous les services sont des instances d'une même classe et peuvent réaliser différent types de taches (par exemple, la seconde tache réalisée par `w4` est exécutante mais pas la première). Notre langage permet de distinguer ces différents cas.

Le premier aspect que l'on considère est un aspect de journalisation. Quand un point de jointure correspond à un de ces points de coupe, son action affiche le receveur de la méthode `System.out.println(currentJoinPoint.getReceiver())`. Nous montrons maintenant comment générer différents journaux.

Tous les services réalisent effectivement un service quand leur méthode `doTask` est appelée :
`allW = Inter (Filter isCall) (Filter (isMethod "doTask"))`
Les service administratifs ne sont pas appelés par un client :

```
adminW = Diff allW (Cflow allC)
```

où `allC` est similaire à `allW` et sélectionne tous les clients. Donc, `adminW` sélectionne les services appelés pendant `Caller.doService` (c.a.d. `Cflow allC`). Dans ce cas, on obtient les services 2, 7 et (la seconde invocation de) 4.

Les services principaux sont les services au niveau le plus haut :

```
principalW = Diff allW (CflowBelow allW)
```

Donc, `principalW` sélectionne tous les services sauf (c.a.d. Diff) ceux pendant `Worker.doTask` (c.a.d. `CflowBelow allW`). Nous devons utiliser `CflowBelow` ici car `Cflow allW` inclut `allW`, donc `Diff allW (Cflow allW)` serait vide. On journalise les services 1, 2, 3, 6, 7, 8 et 10.

Les services principaux administratifs ont les deux propriétés :

```
principalAdminW = Inter principalW adminW
```

On journalise les services 2 et 7.

Les services executants sont les services les plus en bas dans chaque branche de l'arbre d'exécution :

```
allRetW = Inter (Filter (isMethod "doTask")) (Filter isRet)
execW = Diff allRetW (PathAbove allW)
```

ils peuvent être identifiés lors de leur retour (c.a.d. `allRetW`) mais excluent (c.a.d. `Diff`) les sous services impliqués (i.e., `PathAbove allW`). Dans ce cas, on journalise les services 1, 2, 5, 6, (second appel de) 4, 9 et 10.

Les services non exécutants sont tous les services sauf les exécutants :

```
noExecW = Inter Path(allW) allRetW
```

Ces sont les services (c.a.d. `allRetW`) qui sont aussi (c.a.d. `Inter`) sur le chemin d'un autre service (c.a.d. `Path allW`). Ce sont les services 3, 4, 7 and 8.

Les services non administratifs sont les services les plus en bas sous des clients (`Cflow allC`) :

```
execNonAdminW = Inter execW (Cflow allC)
```

On obtient les services 1, 5, 6, 9 et 10.

Les services principaux exécutants et non administratifs peuvent être journalisés avec :

```
principalExecNonAdminW = Diff execNonAdminW (CflowBelow allW)
```

Ce sont les services 1, 6 et 10 et ils correspondent au scénario standard (un client appelle directement un service) comme illustré par la brache la plus à gauche.

Les clients principaux entre le premier et le second service administratifs peuvent être journalisés avec :

```
firstPrincipalAdminW    = Head principalAdminW
secondPrincipalAdminW   = Head (Tail principalAdminW)
betweenAdminW           = Diff (From firstPrincipalAdminW)
                               (From secondPrincipalAdminW)
principalC              = Diff allC (CflowBelow allC)
principalCBetweenAdminW = Inter principalC betweenAdminW
```

On obtient seulement le client 2 qui est appelé entre les services 2 et 7.

Enfin, on peut noter que des services de niveaux (premier, second, troisième...) arbitraires peuvent être sélectionnés (notez aussi qu'il peut exister plusieurs définitions équivalentes d'un même point de coupe) :

```
topLevelW = Diff allW (CflowBelow allW)          (1, 2, 3, 6, 7, 8, 10)
noTopLevelW = Diff allW topLevelW                (4, 5, 4, 9)
secondLevelW = Inter (CflowBelow allW) allW      (4, 4, 9)
secondLevelW' = Diff noTopLevel (CflowBelow noTopLevel)
thirdLevelW = Inter (CflowBelow secondLevel) allW   (5)
...
```

Nous illustrons maintenant notre langage avec un aspect de facturation : un service doit facturer ses clients. Par exemple, dans la branche la plus à gauche de la Figure 7.3, service 1 doit facturer le client 1. Ces services non administratifs peuvent être sélectionnés par :

```
Inter (Cflow(Inter(Filter isCall)(Filter (isMethod "doService"))))
      (Inter(Filter isCall)(Filter (isMethod "doTask")))
```

```
1  aspect MyAspect {
2    pointcut all() : execution(* *.* (..));
3    PcD pcd = new Cflow(new Filter(new IsMethod("foo")));
4    before() : all() {
5      CallJp callJp = new CallJp(thisJoinPoint);
6      pcd.evalCallBefore(callJp);
7      if (pcd.isMatching()) myAdvice(pcd.getContext(), pcd.getCurrentJp);
8      pcd.evalCallAfter(callJp);
9    }
10   after() returning (Object ret) : all() {
11     RetJp retJp = new RetJp(thisJoinPoint, ret);
12     pcd.evalRetBefore(retJp);
13     if (pcd.isMatching()) myAdvice(pcd.getContext(), pcd.getCurrentJp);
14     pcd.evalRetAfter(retJp);
15   }
16   void myAdvice(Context context) { ... }
17 }
```

FIGURE 7.4 : Instrumentation du programme de base

Nous évoquons maintenant du passage de contexte en considérant un aspect de facturation. En effet, les points de jointure correspondants à un tel aspect représentent les appels à `doTask` sur un service lorsqu'il y a un appel en cours à `doService`. L'action doit connaître l'identité du client à facturer qui n'est pas décrite dans le point de jointure courant mais dans un point de jointure passé. A cette fin, l'arbre des appels est réifié au fur et à mesure de l'exécution du programme de base et du point de coupe. Pour des raisons d'efficacité c'est un arbre partiel qui ne contient que les point d'intérêt qui est réifié. Dans cet arbre, les points de jointure sont mémorisés. Cela permet à une action d'avoir accès aux points de jointure passés (par exemple dans un sous arbre d'appels ou bien dans le chemin jusqu'à la racine).

7.1.3 Prototype Java

Nous avons mis en œuvre un prototype pour Java qui évalue un point de coupe à la volée, au fur et à mesure de l'exécution du programme de base.

Notre prototype est basé sur le patron de conception interprète : chaque opérateur de notre langage est mis en œuvre par une classe qui étend la classe abstraite `PcD`. Un point de coupe est un arbre de syntaxe abstraite construit en instanciant ces classes : par exemple `new Cflow(new Filter(new IsMethod("foo")))`. Nous utilisons AspectJ comme un préprocesseur pour instrumenter le programme de base et appeler notre interprète à chaque point de jointure comme détaillé dans la Figure 7.4.

La classe abstraite `PcD` définit un champ booléen `isMatching` et quatre méthodes d'interprétation `evalCallBefore`, `evalCallAfter`, `evalRetBefore` et `evalRetAfter`. Chaque fois qu'une méthode du programme de base est appelée, un point de jointure est créé (ligne 5) et la méthode `evalCallBefore(CallJp)` est appelée (ligne 6) pour mettre à jour `pcd.isMatching`. Quand `isMatching` est `true` (c.a.d. le point de jointure courant est sélectionné par le point de coupe `pcd`) une action est exécutée (ligne 7) avec l'information de contexte comme argument. Puis `evalCallAfter(CallJp)` est appelée (ligne 8) pour maintenir les structures nécessaire au passage de contexte . Le même schéma est utilisé pour les retours de méthode (lignes 10 à 15). Cette mise

```
1  class Filter extends PcD {
2    Predicate pred;
3    void evalCallBefore(CallJp jp) { isMatching = pred.apply(jp); }
4    void evalCallAfter(CallJp jp)  { }
5    void evalRetBefore(RetJp jp)   { isMatching = pred.apply(jp); }
6    void evalRetAfter(RetJp jp)    { }
7  }
```

FIGURE 7.5 : Mise en œuvre de `Filter`

```
1  class Inter extends PcD {
2    PcD pcd1, pcd2;
3    void evalCallBefore(CallJp jp) {
4      pcd1.evalCallBefore(jp); pcd2.evalCallBefore(jp);
5      isMatching = pcd1.isMatching && pcd2.isMatching; }
6    void evalCallAfter(CallJp jp) {
7      pcd1.evalCallAfter(jp); pcd2.evalCallAfter(jp); }
8    void evalRetBefore(RetJp jp)  {
9      pcd1.evalRetBefore(jp); pcd2.evalRetBefore(jp);
10     isMatching = pcd1.isMatching && pcd2.isMatching; }
11   void evalRetAfter(RetJp jp)  {
12     pcd1.evalRetAfter(jp); pcd2.evalRetAfter(jp); }
13 }
```

FIGURE 7.6 : Mise en œuvre de `Inter`

en œuvre naïve instrumente toutes les méthodes comme indiqué par le point de coupe `all()`.

L'opérateur `Filter` évalue son prédicat pour chaque point de jointure pour mettre à jour le champ `isMatching` (ligne 3 et 5) dans la Figure 7.5.

Notre prototype fournit différents prédicats `isMethod`, `IsCall`, `IsRet`, `IsInClass`, `IsInstanceOf`, etc. D'autres peuvent être facilement ajoutés par le programmeur.

L'opérateur `Inter` évalue d'abord ses deux sous points de coupe `pcd1` et `pcd2` (ligne 4, 7 et 9, 12), puis il calcule la conjonction des deux booleens correspondants (ligne 5 et 10 dans la Figure 7.6).

Pour `Cflow`, `isMatching` est vrai depuis un point de jointure appel (ligne 5) jusqu'au point de jointure retour correspondant. Pour détecter ce retour (ligne 10) un compteur `nbPendingCalls` est maintenu (ligne 6 and 11 dans la Figure 7.7).

Enfin, `Path` dans la Figure 7.8 est vrai pour les points de jointures qui correspondent au sous point de coupe ou aux retours des méthodes appelantes. Afin de détecter ces points de jointure, un compteur `nbPendingCalls` est maintenu (ligne 4 et 9). Il est mis à `1` (resp. `0`) pour un point de jointure appel (resp. retour) correspondant au sous point de coupe. Ensuite, `isMatching` est vrai pour un point de jointure retour quand `nbPendingCalls` est négatif.

Quand une action est exécutée le point de jonction courant est disponible, mais une action peut aussi nécessiter des points de jonctions passés. Ils auront été stockés dans un arbre d'appel construit au fur et à mesure du calcul.

```
1  class Cflow extends PcD {
2    int nbPendingCalls = 0; PcD pcd;
3    void evalCallBefore(CallJp jp) {
4      pcd.evalCallBefore(jp);
5      isMatching = pcd.isMatching || nbPendingCalls >0;
6      if (isMatching) nbPendingCalls++; }
7    void evalCallAfter(CallJp jp) { pcd.evalCallAfter(jp); }
8    void evalRetBefore(RetJp jp) {
9      pcd.evalRetBefore(jp);
10     isMatching = nbPendingCalls > 0;
11     if (isMatching) nbPendingCalls  ; }
12   void evalRetAfter(RetJp jp) { pcd.evalRetAfter(jp); }
13 }
```

FIGURE 7.7 : Mise en œuvre de `Cflow`

```
1  class Path extends PcD {
2    int nbPendingCalls = 0; PcD pcd;
3    void evalCallBefore(CallJp jp) {
4      pcd.evalCallBefore(jp); nbPendingCalls++;
5      if (pcd.isMatching) nbPendingCalls = 1;
6      isMatching = pcd.isMatching;}
7    void evalCallAfter(CallJp jp) { pcd.evaCallAfter(jp); }
8    void evalRetBefore(RetJp jp) {
9      pcd.evalRetBefore(jp); nbPendingCalls   ;
10     if (pcd.isMatching) nbPendingCalls = 0;
11     isMatching = nbPendingCalls < 0 || pcd.isMatching;}
12   void evalRetAfter(RetJp jp) { pcd.evalRetAfter(jp); }
13 }
```

FIGURE 7.8 : Mise en œuvre de `Path`

7.1.4 Analyse des points de jointures

Un point de coupe sélectionne une liste d'appels/de retour de méthodes. Par exemple,

```
Inter (Cflow callsC) callsB
où  callsB = Inter (Filter isCall) (Filter (isMethod "b"))
    callsC = Inter (Filter isCall) (Filter (isMethod "c"))
```

sélectionne des appels à `b` pendant l'exécution de `c`. Donc, c'est une approximation sure de dire qu'il sélectionne certains appels à `b`. Nous appelons cette approximation l'alphabet du point de coupe. Il est possible d'analyser un point de coupe pour calculer son alphabet (un sur ensemble des points d'appel et de retour). Un tel alphabet permet d'obtenir une mise en œuvre réaliste qui n'instrumente que certaines méthodes. Il permet aussi de vérifier statiquement si deux points de coupe peuvent avoir des points des jointure communs à l'exécution. Lorsque les alphabets sont disjoints, il n'y aura jamais de point de jointure commun. Cette information permet ainsi de s'assurer de la non-interférence d'aspects.

7.1.5 Bilan

Nous avons montré comment le langage de coupes d'AspectJ peut être formalisé et généralisé pour définir différents points de coupe sur l'arbre d'appels. Nous avons aussi montré comment ces points de coupes permettent de passer du contexte au fur et à mesure de l'exécution. Ces points de coupe se basent sur l'arbre des appels. Notamment, on y retrouve la notion de sous arbre (avec `cflow`) et chemin/pile d'appel (avec `path`). La prochaine section généralise l'approche en montrant comment passer du contexte sans se baser sur l'arbre des appels mais en définissant des relation récursives entres différents points grâce à DataLog.

7.2 Datalog : flots relationnels

Cette section est basée sur l'article [36] qui montre les limites de l'inspection de pile par un aspect et propose d'utiliser à la place de la pile une base de faits Datalog pour définir les relations entre points de jointure et les interroger. Cette approche permet de définir des flots relationnels entre points de jointure.

7.2.1 Les limites de l'inspection de pile

On a vu au début de ce document dans la section 3.3.2 comment l'opérateur `cflow` permet de passer du contexte d'un point à un autre. Par exemple on a considéré une application avec une architecture fonctionnelle où lorsqu'un service est appelé, le client correspondant est en attente dans la pile des appels. On a aussi vu comment cet opérateur pouvait s'écrire avec des points de coupe plus primitifs qui réifient la pile des appels. Cette pile peut alors être inspectée pour aller y collecter du contexte.

Lorsque l'on considère une architecture différente, un point de coupe peut ne plus être valable. Par exemple, si on considère une application client service non plus fonctionnelle (avec des appels imbriqués) mais par lots (avec des appels entrelacés), alors l'exécution de la méthode du client place une tache dans une file et termine son exécution. Ainsi lorsqu'un service est appelé l'identité du client correspondant ne peut plus être retrouvée en inspectant la pile des appels. Dans ce cas la solution n'est pas de réifier la pile des appels mais de maintenir une table qui mémorise l'association d'un client à une tâche. Lorsqu'un service exécute une tâche il peut ainsi inspecter cette relation et identifier le client comme illustré dans la Figure 7.9.

```
1  aspect BillBatch {
2    Hashtable callers = new Hashtable();
3    after(Caller c) returning(RequestId rid):
4    execution(RequestId Caller.service())
5    && target(c) {
6      callers.push(rid,c);
7    }
8    before(Worker w,RequestId rid):
9    execution(void Worker.task(RequestId))
10   && target(w)
11   && args(rid) {
12     w.bill(callers.get(rid));
13   }
14   after():
15   execution(void Worker.task(RequestId))
16   && args(rid) {
17     callers.remove(rid);
18   }
19 }
```

FIGURE 7.9 : Un aspect de facturation pour les lots.

La table est déclarée à la ligne 2. La première action à la ligne 6 ajoute une relation entre un client et une requête dans la table. La seconde action à la ligne 12 facture le client lorsqu'un service traite une requête. La troisième action à la ligne 17 supprime la relation entre un client et une requête lorsque le service a fini son exécution.

7.2.2 Mise à jour et inspection d'une base de faits relationnels

Nous avons généralisé l'approche précédente (création de relation, interrogation de relations, destruction de relations) en utilisant Datalog. Ce formalisme offre en effet un bon compromis entre pouvoir d'expression et efficacité.

Datalog permet de définir des règles qui déduisent de nouveaux faits à l'aide de plusieurs faits. Ces nouveaux faits peuvent à leur tour être utilisés pour déduire de nouveaux faits. Datalog autorise les règles récursives. Il autorise aussi (avec certaines contraintes d'ordonnancement des faits en couches) de déduire des faits de l'absence de certains faits (c.a.d. d'utiliser la négation dans les règles). Enfin, Datalog garantit que les déduction terminent toujours et donc l'inspection des relations par un aspect ne peut pas introduire de divergence.

Nous avons proposé une syntaxe qui marie AspectJ et Datalog et évalué notre approche, que nous baptisons "les aspects relationnels", sur différents exemples. Datalog joue ici à la fois le rôle d'action qui crée et détruit des faits primitifs, mais aussi celui de langage de coupe qui en définissant des règles relie logiquement des points.

Notamment, nous avons montré comment une simple relation binaire bidirectionnelle permet de réifier la relation entre un élément et les ensembles auxquels il appartient. Les collections fournissent l'information dans un seul sens : les éléments contenus dans un ensemble mais pas les ensembles qui contiennent un élément. La représentation d'une telle relation par un aspect relationnel permet d'implémenter efficacement la suppression d'un élément et ainsi son retrait des ensembles qui le contiennent sans avoir besoin de parcourir tous les ensembles existants.

Les faits Datalog ne sont pas limités aux relations binaires. Ainsi il a été possible d'associer une zone d'affichage, un élément de figure graphique, l'identité d'un champ et un objet qui contient ce champ afin de réifier la relation suivante : "détecter les changements qui ont lieu sur des champs qui ont précédemment été lus dans le flot de contrôle du dernier appel à la méthode qui réaffiche tout". Cette relation permet de rafraîchir l'affichage en faisant un minimum de travail (on ne considère pas les objets dont les champs n'ont pas changé) et sans présager des champs significatifs pour l'affichage (on ne s'intéresse qu'aux champs qui ont été lus lors d'un affichage précédent).

Nous avons étendu un solveur de contraintes avec des aspects relationnels pour y ajouter des explications. Dans ce cadre, les explications représentent les dépendances (dynamiques) entre des modifications de variables. Cela permet de représenter des chaînes de modification et lorsqu'un échec arrive dans la recherche de remonter cette chaîne afin d'identifier un choix fautif. Dans ce cas précis, Datalog a permis de définir ces dépendances avec des règles récursives.

Enfin, nous avons montré comment dans un éditeur graphique les aspects relationnels permettent de propager une modification de couleur à tous les objets graphiques connectés entre eux. Nous détaillons cet exemple dans la Figure 7.10. L'action à la ligne 5 crée un fait `CONNECTSTART(cf,c)` pour représenter qu'une figure de connexion `cf` (par exemple une flèche) a été attachée à un connecteur `c` (le connecteur est un objet intermédiaire non graphique utilisé dans la mise en œuvre de l'éditeur graphique). L'action à la ligne 6 détruit (comme indiqué par `!`) le fait lorsqu'une connexion est rompue. Dans ce cas, le connecteur n'est pas connu comme indiqué par le symbole `_`. Les actions aux lignes 14 et 18 font de même pour l'autre extrémité de la connexion. Comme écrit ci dessus, le connecteur est un objet intermédiaire. Il se situe entre une figure graphique (par exemple une boîte) et une figure de connexion (par exemple une flèche). La ligne 20 crée un fait lorsqu'un connecteur est créé et associé à une figure. La règle ligne 23 introduit une nouvelle relation `CONNECTED` entre deux figures connectées pour ne pas s'embarrasser des détails (connecteur, figure de connexion, connecteur). Les règles aux lignes 25 et 27 définissent la fermeture commutative et transitive de `CONNECTED`. Le calcul de cette fermeture est déclenché à la ligne 33 pour déterminer les figures connectées à mettre à jour lorsqu'une couleur est changée.

7.2.3 Evaluation

Ces différents exemples nous ont permis d'évaluer (positivement) la puissance d'expression des aspects relationnels. La mise en œuvre d'un prototype (moteur Datalog) nous a aussi permis d'évaluer l'efficacité de notre approche. Même si en théorie une base de faits Datalog peut toujours être saturée et donc une requête ne peut pas diverger, en pratique une base de faits peut devenir très grande : par exemple un aspect relationnel peut créer un fait pour chaque appel à un accesseur. Dans ce cas, le coût en mémoire de stockage de la base de faits et sa saturation via des applications de règles peut devenir déraisonnablement coûteuse. D'un autre côté, certains aspects relationnels ne créent qu'épisodiquement une poignée de faits. Par exemple dans le cas de l'éditeur graphique qui propage les changements de couleur, le nombre d'objets connectés est souvent raisonnablement petit et l'aspect interactif de l'application rend indécelable un très léger surcoût au moment de cliquer sur l'action de changement de couleur. Les aspects relationnels sont donc un outil de plus pour le programmeur qui doit les utiliser avec discernement. Suite à des difficultés à publier ces résultats nous n'avons pas poussé plus avant l'étude des aspects relationnels. Dans certains cas, le code généré aurait pourtant pu être exactement celui qu'un programmeur écrirait.

```
1  aspect ColoredConnectedFigures {
2    before(ConnectionFigure cf,Connector c):
3    execution(void ConnectionFigure.connectStart(Connector))
4    && target(cf) && args(c) {
5      CONNECTSTART(cf,c);
6    }
7    before(ConnectionFigure cf):
8    execution(void ConnectionFigure.disconnectStart()) && target(cf) {
9      ! CONNECTSTART(cf,_);
10   }
11   before(ConnectionFigure cf,Connector c):
12   execution(void ConnectionFigure.connectEnd(Connector))
13   && target(cf) && args(c) {
14     CONNECTEND(cf,c);
15   }
16   before(ConnectionFigure cf):
17   execution(void ConnectionFigure.disconnectEnd()) && target(cf) {
18     ! CONNECTEND(cf,_);
19   }
20   after(Figure f) returning(AbstractConnector c):
21   execution(AbstractConnector.new(Figure)) && args(f) {
22     CONNECT(f,c);
23   }
24   // rules for connected figures
25   CONNECT(f1,c1) && CONNECTSTART(cf,c1)
26   && CONNECTEND(cf,c2) && CONNECT(f2,c2) => CONNECTED(f1,f2)
27
28   CONNECTED(f1,f2) && CONNECTED(f2,f3) => CONNECTED(f1,f3)
29
30   CONNECTED(f1,f2) => CONNECTED(f2,f1)
31
32   // repaint propagation
33   around(Figure f,Object value):
34   execution(public void Figure.setAttribute(String, Object))
35   && target(f) && args("FillColor",value) {
36     for f2 in CONNECTED(f,f2)
37       do f2.setAttribute("FillColor",value);
38   }
39 }
```

FIGURE 7.10 : Un aspect relationnel pour la propagation des couleurs entre figures connectées

7.3 Conclusion

Les points de coupes sensibles au flot de contrôles étaient présent dès le début de AspectJ sous la forme de l'opérateur `cflow`. Ils servaient néanmoins déjà aussi à passer du contexte et étaient donc orientés vers le flot de données. Une proposition de points de coupe complètement basés sur le flot de donnée avait été faites [MK03], mais la définition orientée langage nécessitait de procéder à une analyse dynamique de nombreuses instructions et nécessitaient des optimisations basées sur l'analyze statique pour rendre l'approche efficace. L'idée à d'ailleurs été poussée à la même époque en introduisant dans AspectJ le `pcflow` (pour *predictive* `cflow`), soit un opérateur de flot de contrôle mais basé sur une analyse statique du graphe d'appel programme. Nos approches restent dynamiques mais aussi efficaces en s'intéressant à un sous ensemble seulement des instructions du programme. (Elles généralisent mais différent peu de ce qu'un programmeur pourrait écrire manuellement).

Ce chapitre a présenté deux approches qui permettent d'exprimer de façon modulaire des flots de données qui ne sont pas modulaires dans une application. Les deux approches dépassent l'inspection de pile des appels et reposent sur la réification d'une structure de données plus (relations entre objets) ou moins (arbre des appels) arbitraire. Ces approches peuvent être couteuses en théorie. En pratique leur coût doit être évalué en considérant le surcoût qu'elles introduisent réellement dans l'application de base (ainsi que possiblement la solution manuelle qu'un programmeur pourrait produire). Dans un cas comme dans l'autre, il est naturel que de nouvelles fonctionnalités transverses aient un coût et vouloir limiter le pouvoir d'expression des langages de coupes revient souvent à rejeter le problème sur le programmeur des actions qui devra maintenir lui-même des informations de contexte.

Chapitre 8

Structures statiques

Sommaire

Mes travaux décrits dans les précédents chapitres ont montré comment les flots (de contrôle ou de données) de plusieurs programmes peuvent être reliés et les programmes tissés pour générer une application complète. Dans tous les cas les relations décrites entres les deux programmes étaient *dynamiques* et pouvaient s'apparenter à l'introduction d'un moniteur d'exécution. Dans ce chapitre je présente deux travaux qui reposent sur des structures *statiques* de données (Section 8.1) et de code (Section 8.2).

8.1 Structures statiques de données

Cette section est basée sur un travail publié à la confèrence SC [16]. Il étudie comment différentes représentations d'une même structure de données peuvent cohabiter efficacement.

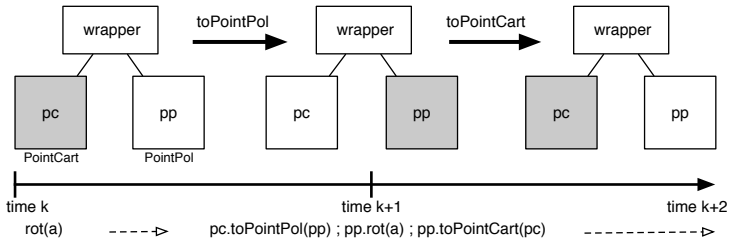


FIGURE 8.1 : Conversion automatique

8.1.1 Deux représentations

Considérons un point dans un espace à deux dimensions. Les coordonnées d'un tel point peuvent être représentées soit en utilisant des coordonnées cartésiennes, soit en utilisant des coordonnées polaires. Certaines opérations, par exemple les translations, sont plus simples à programmer et plus efficaces à exécuter dans la représentation cartésienne. D'un autre côté, d'autres opérations, par exemple les rotations, sont plus simples à programmer et plus efficaces à exécuter dans la représentation polaire.

8.1.2 Conversions automatiques

Nous proposons d'utiliser les deux représentations. Chaque opération est programmée seulement dans la représentation la plus appropriée (par exemple la translation est programmée pour les coordonnées cartésiennes, et la rotation pour les coordonnées polaires) et deux fonctions de conversions (de cartésien à polaire et de polaire à cartésien) sont définies une fois pour toutes. Un encapsulateur est généré de façon à faire collaborer les deux représentations. On choisit arbitrairement la représentation cartésienne comme représentation initiale. Quand une opération programmée dans la représentation cartésienne est appelée, l'encapsulateur délègue d'appel à la représentation cartésienne. Quand une opération programmée dans la représentation polaire est appelée, l'encapsulateur d'abord convertit les coordonnées cartésiennes en coordonnées polaires, puis il délègue l'appel à la représentation polaire, enfin il reconvertit les coordonnées polaires en coordonnées cartésiennes. La Figure 8.1 illustre ce comportement. Dans cette figure, une boîte est un objet Java et une ligne est une référence d'une instance à une autre : un encapsulateur cache les deux représentations `pc` (un point cartésien) et `pp` (un point polaire). A chaque instant dans le temps, la représentation grisée est valide. Initialement, le point est dans sa représentation cartésienne. Quand une opération doit être exécutée dans la seconde représentation (par exemple `rot(a)` à l'instant k), l'encapsulateur d'abord convertit `pc` dans la représentation polaire en appelant `toPointPol`, exécute l'opération dans `pp` (à l'instant $k+1$) et enfin reconvertit `pp` en `pc` en appelant `toPointCart`.

8.1.3 Conversion inversible

Le schéma décrit dans la section précédente introduit systématiquement des conversions quand une opération utilisant une représentation différente de celle par défaut est appelée. Ceci garantit que lorsqu'une opération est exécutée la représentation correspondante est à jour. Cependant, quand on considère une séquence d'opérations, certaines conversions peuvent parfois être évitées. Par exemple, pour exécuter successivement deux rotations, il est inutile de convertir les coordonnées (de polaires à cartésiennes puis de cartésiennes à polaires) entre les deux rotations.

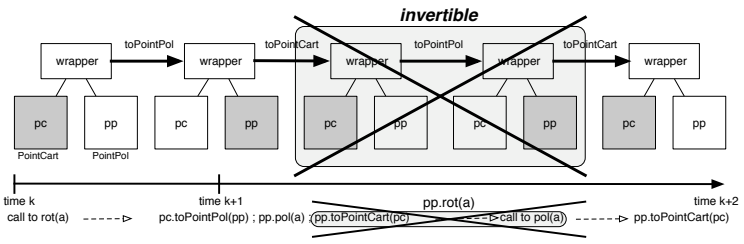


FIGURE 8.2 : Conversion inversible

Un tel scénario est dépeint par la Figure 8.2. Un appel à `rot(a)` mène à convertir `pc` en `pp`. Ensuite, la conversion automatique reconstruit `pc`. Quand un nouvel appel à `rot(a)` est immédiatement exécuté, `pc` doit être reconverti en `pp`. Ainsi, la séquence de conversions inverses `pp.toPointCart(pc); pc.toPointPol(pp)` peut être évitée.

Notre but est d'obtenir un schéma général qui puisse être appliqué à n'importe quelle représentation et algorithme de conversion. Pour éviter les conversions inutiles, nous proposons le schéma suivant. Premièrement, nous retardons les conversions aussi longtemps que possible. Deuxièmement, nous détectons dynamiquement et supprimons les séquences de deux conversions inverses lorsqu'elles se manifestent. Cette technique n'est pas aussi simple qu'elle y peut paraitre. Premièrement, pour retarder les conversions, nous devons être capables de détecter quand l'exécution d'une conversion retardée doit être forcée (autrement le calcul pourrait être incorrect). Deuxièmement, les appels au conversions peuvent être entrelacés avec n'importe quel autre code du programme. Ceci doit être pris en compte pour détecter les séquences de deux conversions inverses.

8.1.4 Conversion incrémentale

Si une représentation est très grande, sa conversion dans une autre représentation peut être couteuse. Quand une conversion complète n'est pas nécessaire, alors une optimisation consiste à convertir partiellement la représentation. Par exemple, considérons des points colorés. Une couleur peut être représentée soit dans le modèle RGB, soit dans le modèle HSB. Nous supposons qu'un point coloré est représenté soit comme un `PointCartRgb`, soit comme un `PointPolHsb`. On peut appliquer notre technique collaborative pour encapsuler ces représentations. Cependant, on peut noter que la représentation des coordonnées et la représentation de la couleur sont couplées : quand les coordonnées d'un point coloré sont représentées par des coordonnées cartésiennes, alors sa couleur est représentée par une valeur RGB. D'un autre coté, quand les coordonnées d'un point coloré sont représentées par des coordonnées polaires, alors sa couleur est représentée par une valeur HSB. Dans le cas de `PointCartRgb` et `PointPolHsb`, le couplage peut introduire des conversions inutiles. Par exemple, la séquence `rot(a1); setRed(r)` dans la Figure 8.3 nécessite de convertir une couleur RGB en HSB quand `rot` est appelée (bien que cette opération n'utilise pas la couleur).

Si la représentation des coordonnées et celle de la couleur étaient découplées, seules les conversions nécessaires seraient exécutées. Par exemple, dans la Figure 8.4 quand une rotation est exécutée, les coordonnées sont converties dans la représentation polaire mais la couleur reste une valeur RGB. Puis, quand `setRed` est appelée la couleur n'a pas à être convertie.

Il et possible de découpler la représentation des coordonnées de la représentation de la couleur sans modifier le programme. En effet, considérons la méthode conversion suivante :

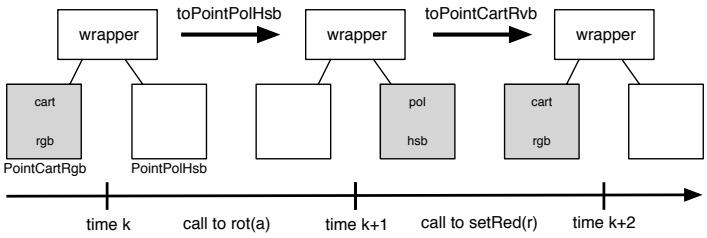


FIGURE 8.3 : Représentations couplées

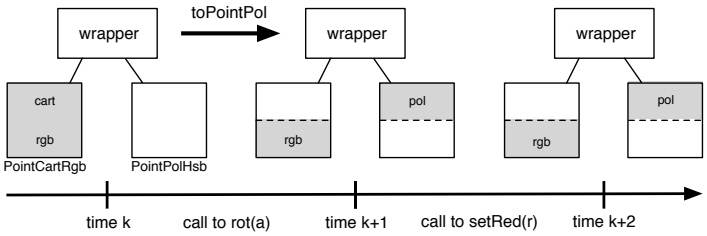


FIGURE 8.4 : Représentations découplées

```
1 void toPointPolHsb (PointCartRgb pcr) {
2   this.toPointPol(pcr);
3   this.toColorHsb(pcr);
4 }
```

Cette méthode appelle `toPointPol` pour convertir les coordonnées dans la representation polaire, puis elle appelle `toColorHsb` pour convertir la couleur dans sa valeur HSB. Afin de découpler les représentations, nous devons seulement rendre paresseuses et inversible les sous méthodes de conversion (`toPointPol` et `toColorHsb`). De cette façon la conversion des coordonnées et de la couleur deviennent indépendantes. Notre schéma général pour la collaboration des représentations repose sur la paresse et la détection et suppression dynamique (la simplification) des séquences de conversions inverses. Ceci nécessite que le programmeur déclare les dépendances entre les représentations (et leur méthodes associées).

8.1.5 Déclaration des dépendances

Nous avons défini un langage pour déclarer les représentations collaboratives. Ce langage fournit ce qui est nécessaire pour générer le code administratif qui met en œuvre une représentation collaborative. La Figure 8.5 présente la grammaire de notre langage.

Ce langage permet de spécifier une représentation collaborative comme suit. Le mot clé `wrapper` spécifie le nom de la classe encapsulateur à générer. Le mot clé `representation` liste deux (ou plus) représentations à encapsuler. Le mot clé `conversion` liste les conversions correspondantes. Notez que ce sont les principales méthodes de conversion que l'encapsulateur doit appeler (par exemple `toPointPolHsb` et `toPointCartRgb` mais pas `toColorHsb` dans l'exemple précédent). Le mot clé `inverse` liste des paires de méthodes inverses. Le mot clé `partition` introduit un ensemble d'atomes qui représente une partition des représentations. Chaque atome représente une partie d'une représentation (par exemple un point cartésien coloré peut être dé-

```
wrapper class {
  representation(class *);
  conversion(method *);
  inverse(method, method);*
  partition {pi*};
  lazy(method, {pi*}, {pi*});*
  strict(method, {pi*}, {pi*});*
}
```

FIGURE 8.5 : Langage de spécification pour les conversions automatiques

```
wrapper ColoredPoint {
  representations (PointCartRgb, PointPolHsb);
  conversions (toPointCartRgb, toPointPolHsb);
  inverse (toPointPolHsb, toPointCartRgb);
  partition {a1, a2} ;
  lazy(toPointPolHsb, {a1}, {a1,a2});
  lazy(toPointCartRgb, {a2}, {a1,a2});
  strict(getX, {a1}, {}));
  strict(setX, {}, {a1}));
  strict(getR, {a2}, {}));
  strict(setR, {}, {a2}));
}
```

FIGURE 8.6 : Représentations Collaboratives (couplées) d'un point coloré (extrait)

composé en un atome pour les coordonnées et un atome pour la couleur, ou un unique atome peut représenter les deux en même temps). Plus la décomposition est fine, plus les dépendances entre les fermetures seront précises. Le mot clé `lazy` spécifie qu'une méthode est paresseuse. Le deux arguments suivants spécifient quels atomes sont lus et écrits quand l'exécution d'une méthode paresseuse est finalement forcée. Le mot clé `strict` spécifie qu'une méthode est stricte. Les deux arguments suivants spécifient quels atomes sont lus et écrits quand une méthode stricte est exécutée. Ces atomes sont utilisés pour calculer les dépendances entre des méthodes paresseuses et strictes et pour forcer l'exécution d'une méthode paresseuse retardée avant qu'une stricte ne lise ou n'écrive les mêmes parties de la mémoire. Les méthodes strictes commencent par déterminer les méthodes paresseuses retardées dont l'exécution doit être forcée. Les méthodes ni paresseuses, ni strictes sont des méthodes Java standard et n'interfèrent pas avec les calculs retardés. La Figure 8.6 montre la spécification des représentations collaboratives pour des points colorés. On choisit de partitionner l'état en deux atomes : `a1` (qui représente `x`, `y`, `r`, `g` et `b`) et `a2` qui représente `rad`, `teta`, `h`, `s` et `b`). Cela couple la représentation des coordonnées avec celle de la couleur. En effet, l'accesseur `getR` lit la partie de la mémoire correspondant à `a2`, et la méthode de conversion `toPointPolHsb` écrit la partie de la mémoire correspondant à `a2`. Ainsi, ces deux méthodes sont dépendantes, et quand la composante rouge de la couleur est accédée, elle force en premier les appels retardés à `toPointPolHsb` (pour convertir couleurs et coordonnées).

Afin de découpler la représentation des coordonnées et de la couleur on doit seulement définir

une spécification basée sur une partition plus précise. Cette spécification aurait quatre atomes : `a1_1` correspondant aux coordonnées cartésiennes, `a1_2` correspondant aux coordonnées polaires, `a1_2` correspondant aux couleurs RGB, et `a2_2` correspondant aux couleurs HSB. Ceci permet de spécifier, par exemple, que `setR()` modifie seulement la représentation RGB (c.a.d. `a1_2`) mais pas les coordonnées cartésiennes `a1_1`. Plus la décomposition est fine, plus les dépendances entre fermetures sont précises. Cependant, plus la décomposition est fine, plus le calcul de dépendance entre fermetures est couteux et consomme de mémoire. Le choix des atomes d'une décomposition devrait être guidé par un compromis entre précision et efficacité (en particulier cela ne vaut pas la peine de dépenser plus de temps à décider si l'évaluation d'une fermeture peut être retardée qu'à l'évaluer). Nous résumons les étapes nécessaires pour nos représentations collaboratives :

1. Les programmeurs fournissent différentes mises en œuvre d'une structure de données.
2. Le programmeur de la représentation collaborative met en œuvre des fonctions de conversion et spécifie les méthodes paresseuses ainsi que les effets de bord.
3. Les classes pour l'encapsulateur et les fermetures sont automatiquement générées. Elles utilisent notre bibliothèque pour la programmation paresseuse sûre en Java (décrite dans la prochaine section).
4. Les programmeurs utilisent la structure de données de manière transparente. L'exécution des méthodes paresseuses est retardée. Quand une méthode stricte est appelée, notre bibliothèque force l'évaluation des méthodes retardées nécessaires au calcul du résultat de la stricte. Au fur et à mesure de l'exécution de l'application les données sont réparties dans les différentes représentations.

8.1.6 Dépendance des fermetures en Java

Dans notre bibliothèque Java, les calculs retardés sont représentés par des sous classes de la classe `Closure`. Les classes fermetures peuvent être automatiquement générées à partir des spécifications de leurs effets de bord décrites dans la section précédente. Les fermetures doivent fournir une méthode `eval` qui appelle la méthode retardée. Chaque fermeture doit aussi fournir deux methodes `reads` et `writes`. Ces méthodes sont utilisées pour calculer les dépendances entre fermetures. Elles retournent un ensemble d'atomes (un `BitSet` dans notre mise en œuvre). Considérons deux fermetures `c1` et `c2`, `c1` étant plus ancienne que `c2`. Quand `c2` doit être évaluée, `c1` doit être évaluée avant si soit `c2` lit un atome écrit par `c1`, ou bien `c2` écrit un atome lu par `c1`, ou bien `c2` écrit un atome écrit par `c1`. Quand un appel de méthode est retardé, la fermeture correspondante est créée et empilée sur `trace`. Quand une fermeture est empilée, si la fermeture inverse est déjà dans la pile, les deux fermetures sont supprimées. Quand une méthode stricte est appelée, la méthode `force` dans la Figure 8.7 est appelée pour calculer les dépendances et forcer l'évaluation de fermetures dépendantes dans `trace` avant l'évaluation de la méthode stricte. La méthode `force` balaye `trace` des fermetures les plus récentes aux plus vieilles. Afin de mémoriser la position dans `trace` une seconde pile est utilisée pour transvaser les fermetures, les unes après les autres. Si la fermeture courante doit être évaluée (ligne 5), elle est annotée (ligne 6) mais pas évaluée immédiatement puisque l'évaluation de fermetures plus anciennes doit être considérée auparavant, mais les effets de bords de la fermeture courante sont mémorisés (lignes 7 et 8). La mémoire écrite par cette fermeture `c` est ajoutée à la mémoire écrite par toutes les fermetures à évaluer `s.writes`. La mémoire lue par cette fermeture `c` est ajoutée à la mémoire lue par toutes les fermetures à évaluer `s.reads` et la mémoire écrite par `c` est retirée de `s.reads`. Puis le parcours de `trace` continue. Quand le début de la trace est atteint,

```
1  void force(AbstractState s) {
2    int i = this.ecart.size();
3    while (! this.trace.isEmpty())  {
4      Closure c = this.trace.pop();
5      if (c.isRequiredBy(s)) {
6        c.eval= true;
7        s.reads.diff(c.getWrites()).union(c.getReads());
8        s.writes.union(c.getWrites());
9      }
10     this.ecart.push(c);
11   }
12   while (this.ecart.size() != i) {
13     Closure c = this.ecart.pop();
14     if (c.eval) c.eval();
15     else this.trace.push(c);
16   }
17 }
```

FIGURE 8.7 : Calcul les dépendances entre fermetures et force leur évaluation

la seconde moitié de la méthode `force` (linges 12 à 15) balaye les fermetures de la plus ancienne à la plus récente. Si une fermeture est annotée alors elle est évaluée, sinon elle est réempilée sur `trace`.

8.1.7 Evaluation

On évalue maintenant notre approche sur un exemple réaliste concernant la théorie des graphes dans le contexte de l'implémentation de contraintes globales. Les contraintes globales représentent un outil irremplaçable de modélisation pour la programmation par contrainte. Elle reposent lourdement sur la théorie des graphes grâce à des propriétés comme les composantes fortement connexes ou les fermetures transitives (et d'autres beaucoup plus complexes). Une contrainte globale doit souvent combiner efficacement plusieurs propriétés. Notre bibliothèque permet de combiner efficacement et de façon sûre plusieurs méthodes liés à des propriétés de graphe. Quand un graphe est représenté par une liste des successeurs directs de chaque nœud (`GraphSucc`) il est simple et efficace d'exprimer une recherche parmi les descendants d'un nœud donné. D'un autre coté, quand un graphe est représenté par une liste des prédécesseurs directs de chaque nœud (`GraphPred`) la même recherche parmi les descendants d'un nœud donné devient plus complexe. Symétriquement, il est facile et efficace de chercher parmi les ancêtres dans `GraphPred`, mais il est plus complexe de chercher parmi les descendants. Dans ce cas, nous pouvons appliquer notre schéma de représentations collaboratives.

Les conversion entre `GraphSucc` et `GraphPred` sont basées sur deux méthodes `toNodePred(n)` et `toNodeSucc(n)` qui se comportent comme indiqué dans la Figure 8.8. La méthode `toNodePred(n)` convertit une colonne de `GraphSucc` en une ligne de `GraphPred`. La notion de colonne est une vue logique dans `GraphSucc` (une liste de listes de nœuds successeurs) qui n'a pas d'accesseur direct correspondant ; elle n'existe pas dans `GraphSucc`, aussi la méthode doit itérer sur chaque nœud. Si la liste de successeurs de `s` dans `GraphSucc` contient `n`, elle le retire de cette liste (ce qui libère la mémoire pour qu'un nœud n'existe que dans une seule représentation à la fois) et elle ajoute `s` aux ancêtres de `n` dans `GraphPred`. La conversion inverse (de `GraphPred` vers

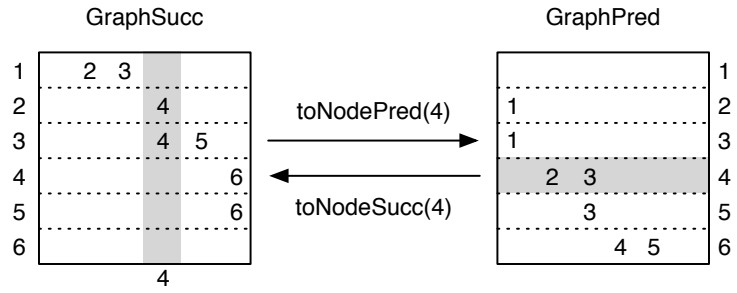


FIGURE 8.8 : Conversion partielle entre lignes et colonnes

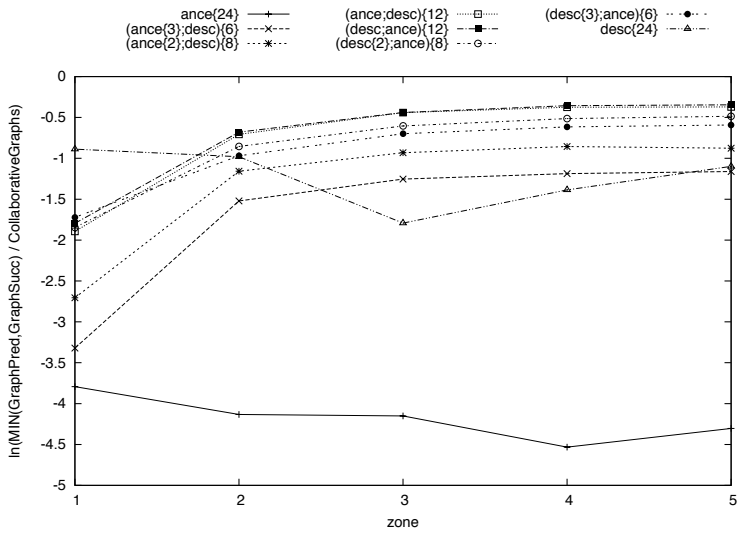


FIGURE 8.9 : Le meilleur de `GraphSucc` et `GraphPred` contre notre représentation collaborative

`GraphSucc`) est symétrique, et donc inversible : les conversions aller-retour d'un nœud laissent les deux représentations inchangées. Afin de convertir tous les nœuds d'un graphe, la méthode `toGraphSucc` (resp. `toGraphPred`) appelle `toNodeSucc` (resp. `toNodePred`) pour chaque nœud du graphe.

Nous avons évalué notre approche dans ce contexte. Nous comparons le temps d'exécution, sur 43200 graphes aléatoirement générés, en utilisant une représentation unique (soit liste des successeurs `GraphSucc`, soit liste des prédécesseurs `GraphPred`) et notre approche utilisant deux représentations collaboratives. Chaque instance de graphe est évaluée sur huit scénarios dépeints dans la Figure 8.9. Elle décrit des motifs utilisé pour ces expérimentations : `(ance{3}desc){6}` signifie que la méthode `ance` est appelée trois fois successivement, puis la méthode `desc` est appelée une fois, ceci est répété six fois en choisissant au hasard des nœuds arguments. Tous les motifs contiennent vingt quatre appels de méthode mais ils varient dans la fréquence d'alternance entre `ance` et `desc`. Dans cette expérimentation, nos graphes sont structurés comme des treillis et chaque scénario est évalué sur cinq types d'arguments groupés par zone : zone 1 signifie que les arguments sont choisis parmi tous les nœuds du treillis, zone 2 signifie que les arguments sont choisis dans la moitié inférieure du treillis, etc. Plus la zone est étendue, plus de nœuds seront convertis pour réaliser le calcul de `ance` ou `desc`.

Pour des scénarios avec le même nombre d'appels à chaque opération (par exemple `(ance;`

`desc){12}`)), les représentations successeurs et prédécesseurs ont des temps d'exécution comparables, mais pour un motif non symétrique (par exemple `ance{24}`) une représentation (inadaptée) peut être 150 fois plus lente que l'autre (adaptée). La Figure 8.9 compare sur chaque scénario notre approche avec la meilleure des deux représentations standard. Elle montre que pour tous les motifs, notre approche est moins efficace que la meilleure des deux représentations standard. Cependant, on peut noter que lorsque moins de nœud doivent être convertis (zones 2 à 5), notre approche est proche de la meilleure représentation standard. De plus, des mesures montrent que notre approche comparée à la pire des deux représentation standard peut être jusqu'à 90 fois plus rapide. Le nombre de nœuds convertis est statistiquement plus grand lorsque la zone considérée est 1 (tous les nœud sont candidats) dans ce cas notre approche crée de nombreuses fermetures et passe un temps non négligeable à tenter de les simplifier par paires inverses. Une solution dédiée à ce problème battrait notre approche. Cependant, notre proposition n'impose pas une complexité mémoire supérieure, et est en général plus efficace que les approches naïves. De plus elle offre une solution générale : elle est plus sure (du moment que les spécification des effets de bords sont correctes aucune évaluation de fermeture ne peut être oubliée), elle réutilise les représentation "naturelles" et elle peut être étendue à plus de deux représentations. Notre approche propose un compromis intéressant entre complexité du code et efficacité.

8.1.8 Dans un cadre fonctionnel

La programmation fonctionnelle repose sur la composition de fonctions. Dans ce cadre les dépendances entre différents calcul sont implicitement exprimées par la structure du programme : la composition de fonction correspond à un schéma producteur consommateur. Dans ce cadre fonctionnel pur la paresse est naturelle à exprimer et à mettre en œuvre.

Les mécanismes impératifs peuvent être élégamment mariés à la programmation fonctionnelle pure paresseuse à l'aide de monade. Dans ce cas deux mondes cohabitent : le fonctionnel pur paresseux et l'impératif strict. Les dépendances entre deux blocs impératifs est implicite : deux blocs sont forcément ordonnés par le programmeur même si leur (absence de) dépendance réelles font qu'ils sont commutatifs.

L'approche proposée pour rendre les programmes impératifs Java paresseux peut être adaptée aux mécanismes impératifs d'un langage fonctionnel. Ainsi, nous avons proposé [10] un canevas en Haskell qui permet de rentre paresseux des sous programmes impératifs comme le tri d'un tableau en place, ou encore l'appel à des fonctions étrangères (Foreign Function Interface). Nous avons ainsi montré comment rendre des tris à bulles ou encore des tris rapides (quicksort) paresseux. Les appels récursifs à l'algorithme afin de trier des sous tableaux sont paresseux et retardés. Lorsque la valeur d'une case du tableau trié est requise (par un calcul stricte comme par exemple une comparaison de valeur ou un affichage) alors les appels retardés nécessaires à décider la valeur de la case, et seulement ces appels la, sont déclenchés. Comme dans le cadre fonctionnel, cette technique favorise la réutilisation des programmes impératifs. Par exemple, le minimum d'un tableau peut être calculé en le triant paresseusement puis en accédant à sa première case. Notre cadre repose sur la réification des calculs, afin de déterminer leurs effets et leurs dépendances réciproques et sur des mécanismes impératifs non surs (notamment la fonction `unsafePerformIO` qui dans un contexte fonctionnel effectue finallement un calcul impératif retardé). Cette réification, nous à permis d'envisager les optimisations dynamiques de programmes (comme par exemple pour éviter de trier deux fois un tableau alors qu'une fois suffit). Enfin, ce travail dans un cadre fonctionnel nous a mené à définir une sémantique formelle de notre approche et à prouver sa correction.

```
1  abstract class Expr {
2    abstract Integer eval ();
3    abstract String show ();
4  }
5
6  class Num extends Expr {
7    int n;
8    Num (int n) { this.n=n; }
9    Integer eval() { return n; }
10   String  show() { return Integer.toString(n); }
11 }
12
13 class Add extends Expr {
14   Expr e1, e2;
15   Add (Expr e1, Expr e2) { this.e1 = e1; this.e2 = e2; }
16   Integer eval() { return e1.eval() + e2.eval(); }
17   String  show() { return "(" + e1.show() + "+" + e2.show() + ")"; }
18 }
```

FIGURE 8.10 : Décomposition selon les données

8.2 Structure statique de code

Cette section est basée sur un travail publié à la conférence CSMR [**?**]. Il étudie comment différentes décompositions d'un même code peuvent cohabiter.

8.2.1 Un évaluateur, deux architectures de code

Nous considèrons un évaluateur en Java. Le type `Expr` représente les expressions à évaluer. Le sous type `Num` représente les constantes entières et `Add` les additions. Deux opérations sont définies pour le type `Expr` : `eval` qui évalue une expression et `show` qui convertit une expression en chaîne de caractères. Un code fonctionnel définit le comportement de ces operations qui dépend des sous types. Il existe différentes manières d'architecturer un même code, c.a.d. d'organiser le code fonctionnel à l'intérieur d'une structure de classes. Ces différentes architectures ont un impact sur la modularité de la maintenance.

La première structure, dans la Figure 8.10, est basée le patron de conception Composite (ou Interpreter). Dans ce cas, le code fonctionnel pour un sous type se trouve dans une sous classe correspondante. Notamment la classe `Expr` est abstraite, la sous classe `Num` définit les méthodes `show` et `eval`, et la sous classe `Add` définit d'autres méthodes `show` et `eval`.

La matrice dans la Figure 8.11 est indicée par les sous types et les opérations. Les classes, qui définissent le code fonctionnel, forment une partition de la matrice selon les sous types. Toutes les informations concernant une donnée sont réunies dans une sous classe unique : par exemple, les codes fonctionnels pour les constantes entières sont réunis dans la sous classe `Num`. Par contre, les informations concernant une opération sont réparties dans plusieurs sous classes : le code fonctionnel de l'évaluation est réparti entre la méthode `eval` de `Num` et la méthode `eval` de `Add`.

Dans cette architecture, la maintenance d'un sous type est modulaire : quand un sous type évolue, les changements du code fonctionnel sont confinés à la sous classe correspondante. D'un

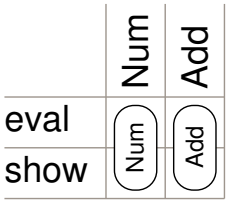


FIGURE 8.11 : Décomposition selon les données

autre côté, la maintenance d'une opération n'est pas modulaire : quand une opération évolue, les changements du code fonctionnel sont répartis dans toutes les sous classes.

La seconde structure, dans la Figure 8.12 est basée sur le patron de conception Visitor. Dans ce cas, les classes `Expr` et les sous classe `Num` et `Add` ne définissent pas de code fonctionnel mais la méthode `accept` du visiteur. Le code fonctionnel pour une opération se trouve dans une unique sous classe correspondante : par exemple, la sous classe `EvalVisitor` définit deux méthodes `visit` suivant que son argument représente une constante entière ou une addition.

La matrice dans la Figure 8.13 est indicée par les sous types et les opérations. Les classes, qui définissent le code fonctionnel, forment une partition de la matrice selon les opérations. Toutes les informations concernant une opération sont réunies dans une sous classe unique : par exemple, les codes fonctionnels pour l'évaluation sont réunis dans la sous classe `EvalVisitor`. Par contre, les informations concernant une donnée sont réparties dans plusieurs sous classes : le code fonctionnel des constantes entières est réparti entre la méthode `accept(Num)` de `EvalVisitor` et la méthode `accept(Num)` de `ShowVisitor`.

Dans cette architecture, la maintenance d'une opération est modulaire : quand une opération évolue, les changements du code fonctionnel sont confinés à la sous classe correspondante. D'un autre coté, la maintenance d'un sous type n'est pas modulaire : quand un sous type évolue, les changements du code fonctionnel sont répartis dans toutes les sous classes.

La dualité entre ces deux architectures illustre la tyrannie de la décomposition primaire : quelque soit la structure choisie, certaines opérations de maintenance ne seront pas modulaires. Dans la suite nous étudions ce problème de maintenance non modulaire.

8.2.2 Maintenance modulaire : le style fonctionnel

L'opposition entre décomposition selon les données et décomposition selon les opérations n'est pas spécifique à la programmation à objets. Dans les langages fonctionnels, les fonctions sont fréquemment définies par filtrage de motifs sur les données. Ceci correspond à une décomposition selon les opérations : maintenir une fonction existante est modulaire, mais maintenir un constructeur dans les données n'est pas modulaire (les changements du code fonctionnels sont répartis dans plusieurs fonctions).

Une seconde manière de définir des fonctions est d'utiliser des opérateurs de traversées (une généralisation de la fonction `fold`) qui prend en paramètre une fonction par constructeur de la structure de données. Puisque ces paramètres fonctionnels sont spécialisés pour des constructeurs, il est raisonnable de les grouper dans des modules contenant tous les codes fonctionnels pour un constructeur. Ceci correspond à une décomposition selon les données : maintenir un constructeur est modulaire mais maintenir une opération ne l'est pas (les changements dans le code fonctionnel sont répartis dans plusieurs modules).

```
1  abstract class Expr {
2    Integer eval(){ return(accept(new EvalVisitor())); }
3    String show(){ return(accept(new ShowVisitor())); }
4    abstract <T> T accept(Visitor <T> v);
5  }
6
7  class Num extends Expr {
8    int n;
9    Num (int n){ this.n=n; }
10   <T> T accept(Visitor <T> v){return v.visit(this); }
11
12 }
13
14 class Add extends Expr {
15   Expr e1, e2;
16   Add (Expr e1, Expr e2) { this.e1 = e1; this.e2 = e2; }
17   <T> T accept(Visitor <T> v){return v.visit(this); }
18 }
19
20 abstract class Visitor <T> {
21   abstract T visit(Num n);
22   abstract T visit(Add a);
23 }
24
25 class EvalVisitor extends Visitor <Integer> {
26   Integer visit(Num a){ return a.n; }
27   Integer visit(Add a) { return a.e1.accept(this) + a.e2.accept(this); }
28  }
29
30 class ShowVisitor extends Visitor <String> {
31   String visit(Num a){ return Integer.toString(a.n); }
32   String visit(Add a) { return "(" + a.e1.accept(this) +
33            "+" + a.e2.accept(this) + ")"; }
34 }
```

FIGURE 8.12 : Décomposition selon les opérations

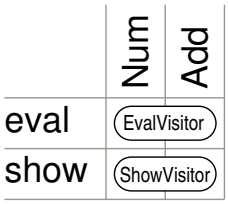


FIGURE 8.13 : Décomposition selon les opérations

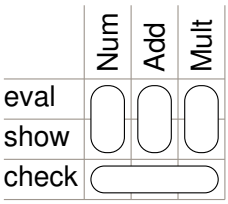


FIGURE 8.14 : Décomposition plus modulaire selon les données

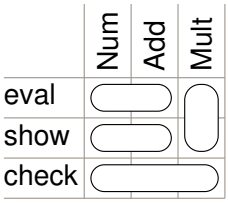


FIGURE 8.15 : Décomposition plus modulaire selon les opérations

8.2.3 Extension modulaire : le problème de l'expression

Le problème de l'extension est similaire au problème de la maintenance : dans la décomposition selon les données, ajouter un nouveau constructeur est modulaire : une nouvelle sous classes est définie dans un programme à objets, ou bien dans un langage fonctionnel, un nouveau module définit toutes les opérations pour le nouveau constructeur. Réciproquement, dans le décomposition selon les opérations, ajouter une nouvelle opération est modulaire : une nouvelle sous classe qui définit une méthode `accept` pour chaque cas est introduite, ou bien une nouvelle fonction est définie sur tous les constructeurs par filtrage de motifs. Ce problème d'extension modulaire est connu comme le problème de l'expression. Différentes solutions langages ont été proposées (voir [Zen05] pour un état de l'art). Cependant, après l'ajout modulaire grâce à une solution langage, le code ne peut plus être considéré comme modulaire. Par exemple, dans la Figure 8.14 après l'ajout d'une opération, le code n'est plus modulaire selon les sous types. Et dans la Figure 8.15 après l'ajout d'un constructeur, le code n'est plus modulaire selon les opérations. Pour cette raison, les solutions langages au problème de l'extension modulaire entre en conflit avec la maintenance modulaire.

8.2.4 Transformations inversibles de programmes

Les transformations présentées dans les sections précédentes peuvent être difficiles à mettre en œuvre. Nous avons décidé de les exprimer comme des séquences d'opérations fournies par un outil de réusinage. Cela garantit que nos transformations préservent la sémantique.

Nous avons mis en œuvre une transformation inversible pour Java et pour Haskell afin de passer d'une décomposition selon les données à une décomposition selon les opérations et inversement. Une telle transformation résout le problème de la maintenance modulaire : quand une opération de maintenance doit être réalisée sur un programme qui n'est pas dans la bonne décomposition, la transformation est appliquée afin de mettre le programme dans la forme convenable. Une séquence d'opérations de maintenance peut nécessiter des va et vient comme illustré dans la Figure 8.16. Dans le cas d'une extension non modulaire, la transformation préalable du programme dans la décomposition adéquat permet toujours d'exprimer l'extension modulairement.

Une fois l'évolution finie, le programme peut être laissé dans sa dernière décomposition

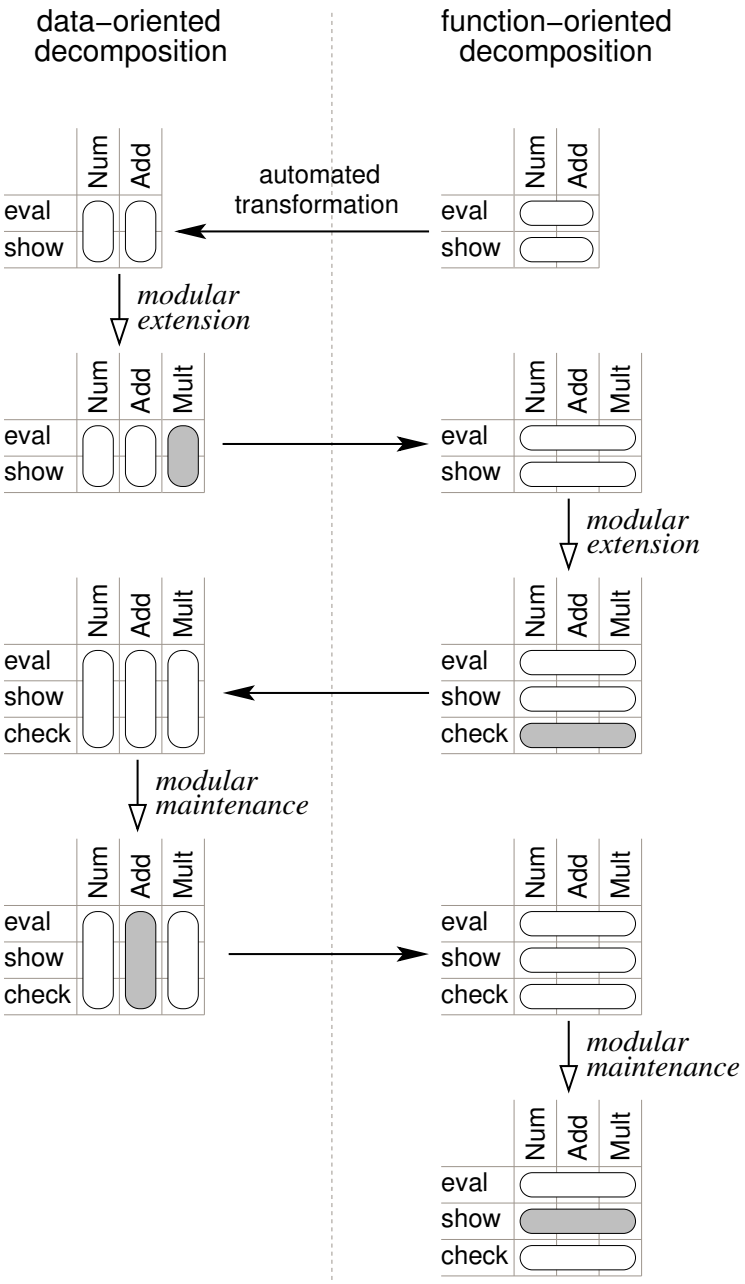


FIGURE 8.16 : Scénario d'évolution

ou transformé pour retrouver sa décomposition initiale dans laquelle les changements ont été répercutés.

8.2.5 Expérimentation basée sur un réusineur de code

Nous avons réalisé une expérimentation avec Java et Haskell. En Java, nous avons testé la transformation composite/visiteur avec les outils de réusinage intégrés dans Eclipse et IntelliJ. La transformation complète n'est pas encore complètement automatique.

En Haskell, nous avons testé la transformation donnée/opération avec le réusineur HaRe. La transformation a pu être complètement automatisée pour plusieurs exemples. Elle est définie par un script qui doit être légèrement adapté pour chaque programme. Ce travail a nécessité de modifier l'interface du réusineur.

Voici les conclusions que l'on peut tirer de ces expérimentations.

- La sémantique (et la sûreté) de type sont conservées par les transformations.
- Le code après un aller retour est exactement le même mais sa mise en forme peut avoir changé. En Java, la visibilité des éléments de la classe composite doit changer lorsque l'on passe dans la structure de visiteur. Ce n'est pas lié à notre transformation mais à la nature du patron de conception.
- Sur des programmes de petite taille (6 sous types et 6 opérations), les réusineurs de Java sont raisonnablement rapides (3 secondes pour tout transformer), alors que le réusineur Haskell peut nécessiter 3 minutes pour tout transformer (une opération de renommage peut prendre jusqu'à 30 secondes). Ceci s'explique comme le réusineur Haskell ne mémorise par la représentation interne d'un programme mais repart de zéro pour chaque opération.
- Nos transformations sont sensibles aux variations de la structure initiale (un programme équivalent mais sous une autre forme peut ne pas être transformable).
- Certaines opérations ne sont pas fournies par les outils. Pour Java, les transformations nécessitent encore quelques étapes manuelles. Pour Haskell, cinq opérations ont du être ajoutées au réusineur pour pouvoir définir une transformation complètement automatique.

8.3 Conclusion

Les représentations multiples d'une structure de données avaient déjà été abordées par Wadler et ses vues [Wad87]. Dans son cas une seule représentation était maintenue mais différentes reconnaissances de motifs pouvaient être exprimées (au prix de calculs parfois couteux par exemple pour accéder à une liste par sa fin). La structuration du code est un problème ancien [P72], que Wadler (encore lui) à remis à l'ordre du jour en le baptisant *expression problem* [Wad98].

Ce chapitre a présenté deux travaux basés sur les structures statiques de données et de code. Dans le premier cas (Section 8.2) plusieurs représentations d'une même structure de données (favorisant chacune certains traitements algorithmiques) existent en une alternance temporelle en fonction de la succession des traitements. Lorsque les conversions de représentations peuvent être partielles (par morceaux) l'évaluation paresseuse et la simplification dynamique de fonction inverse permet de gagner en efficacité dans un langage strict ou impératif. Dans le second cas (Section 8.2) les définitions des structures de données et des fonctions sont réorganisées pour être groupées selon les données ou les fonctions. Chacune de ces deux organisations favorisant la modification (ou l'extension) soit des données, soit des fonctionnalités. En plus d'être plus statiques

dans leur approche que ceux des chapitres précédents, ces deux travaux affichent aussi ouvertement des buts allant au delà de la correction. Notamment pour les représentations multiples de données la question de l'efficacité des algorithmes (bien adaptés à telle ou telle structure) est clairement abordée. Pour la réorganisation de code, c'est la problématique de la maintenance et de l'extension de programme qui sont visées.

Chapitre 9

Autres travaux

Sommaire

9.1 Autres travaux

J'ai choisi de structurer ce document autour d'une série de travaux sur des langages de programmation modulaires pour des notions de flots ou de structures. Dans ce chapitre, j'évoque d'autres travaux plus ou moins connexes non détaillés dans les chapitres précédents.

9.1.1 Composants et aspects

Dès 2005 j'ai contribué avec Jacques Noyé et Mario Südholt à un chapitre [43] de livre sur l'alliance des composants et des aspects. Au delà de la présentation des caractéristiques essentielles de la programmation par composants et de celle par aspects, nous nous sommes attachés à rapprocher ces deux paradigmes complémentaires. En effet, la programmation par composant offre une structure (de base) forte sur lequel différents services (par exemple, la persistance, les transactions, etc.) viennent se greffer. La mise en œuvre de ses services et souvent non modulaire vis à vis de la structure de composants. Les mécanismes d'interception (liés aux conteneurs par exemple) et de configuration (par exemple via des langages à base de balise XML) sont à rapprocher de la programmation par aspects. Nous ouvrions l'horizon à la fin de ce chapitre en concluant sur les langages de composants et aspects.

Plus récemment dans le cadre de la thèse de Hakim Hannousse [Han11], nous avons étudié la programmation par aspects dans un cadre à composants. En particulier, nous avons proposé un langage de définition de vues qui donne une autre structuration d'un système à composants

(que l'on peut rapprocher d'un langage de points de coupe) et nous nous sommes intéressés à la composition d'aspects [34]. Ce travail a été instancié à Fractal pour étendre la notion de contrôleurs et proposer des opérateurs de composition d'aspects [13]. Nous avons aussi modélisé l'approche dans le cadre formel du testeur de modèles Uppaal afin de détecter les interactions entre aspects [14].

9.1.2 Analyse dynamique de flots de mémoire

Nous nous sommes intéressés à l'analyse dynamique de l'exécution de programmes Java afin d'y vérifier des hypothèses. Dans ce contexte, nous avons proposé de réutiliser l'interface de débogage de la machine virtuelle Java afin de la considérer comme un générateur de point de jointures et de la connecter à un moteur prolog dont les règles contrôlent l'exécution du programme Java et détectent des propriétés. Ce travail [26] nous a notamment permis de détecter des patrons de conception en nous basant sur les notions d'association, d'agrégation et de composition. Ce travail à été mené en collaboration avec l'équipe Contraintes des Mines de Nantes, Lina, Inria. Collaboration que je détaille maintenant.

9.1.3 Travaux transverses

Depuis un certain nombre d'années je coopère avec des membres de l'équipe Tasc (programmation par contraintes) des Mines de Nantes, Lina. Les solveurs de contraintes sont des logiciels complexes dont les performances sont critiques. Un premier travail à consisté à définir un solveur idéalisé puis à l'étendre avec un mécanisme d'explications en utilisant la programmation par aspects. Ce travail a été publié à la fois dans la communauté programmation par contraintes [40] et la communauté programmation par aspects [41]. Dans ce cadre transverse, j'ai coencadré avec Xavier Lorca et Narendra Jussien de l'équipe Tasc la thèse de Charles Prud'homme. Ces travaux ont abouti à un DSL permettant piloter la phase de propagation dans les solveurs de contraintes discrètes [2].

9.1.4 Conservation de propriétés malgré la programmation non modulaire

Dans le cadre du laboratoire formel du réseau d'excellence AOSD j'ai coencradré avec Pascal Fradet la thèse de Simplice Djoko Djoko. Ce travail nous a amené à définir un cadre formel et abstrait, la CASB [45], pour définir la sémantique de différents systèmes de programmation par aspects. Nous avons par la suite exploité ce cadre pour étudier des catégories d'aspects qui préservent des classes de propriétés du programme de base. Un avantage de l'approche est que les preuves de préservation sont faites une fois pour toute pour chaque catégorie et classe. Chaque classe de propriété est définie comme un sous ensemble de la logique temporelle et chaque catégorie d'aspects est définie en énonçant les impacts que peut avoir un aspect sur la sémantique du programme. Ainsi les aspects de la catégorie la plus inoffensive, les aspects observateurs, peuvent lire l'état du programme mais ne peuvent pas le modifier, le flot de contrôle du programme de base peut être enrichi (mais pas modifié) en y insérant des actions, ces actions doivent toujours terminer (sinon le flot de contrôle du programme de base serait modifié) et ces actions ne peuvent pas contenir de code réflexif. La classe de propriétés conservées par ces aspects est (relativement) grande même si elle interdit l'usage de l'opérateur de logique temporelle *next*. Cette limitation est liée au problème de bégaiement (stuttering) identifié par Lamport [Lam83]. Une taxonomie des six classes de classes/catégories d'aspects à été ainsi proposée. Ces classes/catégories sont des restrictions/extensions de l'une ou l'autre. Plus une catégorie d'aspect a d'impact sur la sémantique du programme, moins la classe de propriétés

conservées est vaste. Cette taxonomie a été proposée à la fois dans un contexte déterministe et non déterministe (programmes concurrents) [17]. Par la suite afin de rendre peu coûteux le test d'appartenance d'un aspect à une catégorie, nous avons proposé des langages d'aspects restreints qui garantissent l'appartenance à une catégorie par construction [18]. La synthèse de ces travaux a finalement été produite [3].

9.1.5 Vieux amours

Ma thèse de doctorat porte sur la modélisation formelle des techniques de mise en œuvre de langages fonctionnels par transformation de programme. Quelques années après ma thèse j'ai eu l'occasion de me replonger dans cet univers et de proposer avec Pascal Fradet une modélisation de la machine de Krivine et de ses variantes [5].

9.2 Prototypes

Je ne suis pas un programmeur émérite mais j'aime confronter occasionnellement mes idées de recherche à la réalité de la programmation. Ainsi à plusieurs reprises j'ai participé à la conception et au développement de petits prototypes de recherche. Ces développements dans un contexte idéalisé m'ont permis de me concentrer sur l'essence des problèmes tout en vérifiant (parfois avec mauvaise surprise) la cohérence de définitions qui ont pu par la suite être corrigées ou simplement améliorées. Je passe ici brièvement en revus de tels prototypes :

- MetaJ : est un interpréteur réflexif pour un sous ensemble de Java. Cet interpréteur est en fait une collection d'interpréteurs dans la mesure où en suivant les travaux pionniers sur les reflexions dans le langage Lisp n'importe quelle classe de notre méta interpréteur Java peut être rendue réifiable. Nous avons ainsi pu montrer que l'essentiel des API de réflexion repose sur un folklore qui fait des hypothèses implicites quant à la mise en œuvre des mécanismes du langage (par exemple le protocole *lookup-apply*). En changeant la définition d'un interpréteur on change l'API de sa version réflexive. Et en choisissant les classes à réifier on contrôle le degré de réflexivité du langage.
- EAOP : est une bibliothèque Java utilisant des threads pour définir des coroutines et un arbre pour composer des aspects. En utilisant un outil de transformation de programme nous avons facilement pu instrumenter le programme de base (et possibles les actions des aspects) pour tisser des aspects à la façon d'un moniteur d'exécution.
- Cacao : est un solveur de contraintes discrètes idéalisé. Nous avons conçu ce programme avec un minimum de concepts (c.a.d. sans à priori), puis a posteriori nous l'avons étendu avec AspectJ pour y ajouter une analyse dynamique connue sous le nom d'explication. Ce travail nous a confronté à certaines limites d'AspectJ et nous a permis d'identifier des pistes de recherche que nous explorons encore à ce jour.
- AXL : est une bibliothèque Java offrant un interprèteur pour un langage de coupe généralisé sensible aux flots de données. Il utilise AspectJ pour instrumenter le programme de base et appeler l'interprèteur.
- CEAOP : est une bibliothèque Java mettant en œuvre notre modèle de programmation par aspect dans un cadre concurrent et notamment des opérateurs de composition d'aspects.

Ces prototypes n'ont pas franchi le stade du jouet d'étude mais ils ont réellement nourri nos réflexions et permis de raffiner nos propositions.

9.3 Conclusion

La recherche n'est pas un chemin en ligne droite. Le mien a peut être pris plus de tours et détours que d'autres, mais cette richesse a permis aussi d'alimenter mes reflexions. On peut tout de même noter qu'à une époque où Haskell connaît un regain d'intérêt je peux travailler de nouveau dans ce cadre (comme notamment mes derniers travaux avec Nicolas Tabareau sur l'évaluation impérative paresseuse précédemment évoqués).

Chapitre 10

Conclusion

Sommaire

10.1 Bilan

Qu'avons nous fait ? Comment l'avons nous fait ? Que reste-t-il à faire ? Cette section fait un bilan des travaux présentés dans ce document et esquisse un programme de recherche pour l'avenir.

L'objectif à ne pas perdre de vue est toujours le même : donner au programmeur des supports (langages) afin de l'aider à développer des applications complexes. Une application peut être complexe pour différentes raisons. L'une d'entre elles est je pense que les limites de la modularité vont amener le programmeur à prendre en compte différentes fonctionnalités en même temps avec pour résultat d'obtenir un code entrelacé ayant perdu toute structuration propre à chaque fonctionnalité. Une autre raison complémentaire est que le programmeur entrelaçant son code à la main va souvent devoir surspécifier son code (par exemple en choisissant un ordonnancement précis de deux instructions pourtant commutatives) et être tenté d'appliquer des optimisations qui ne sont pas forcément indispensables. Un code dupliqué va alors pouvoir évoluer de différentes façons suivant son contexte d'entrelacement et créer des opportunités de bogues (par l'oubli de la correction d'un bogue dans les deux branches à la fois ou par l'impossibilité de corriger le bogue de la même façon dans les deux branches).

Nous voulons inciter le programmeur à ne pas se censurer. En effet, un programmeur peut être amené à renoncer à programmer certaines fonctionnalités à cause de leur complexité. Par exemple une des motivations des travaux décrits à la section 8.1 était que les propriétés sont parfois tellement complexes à maintenir lorsque le graphe évolue qu'un schéma de programmation qui recalcule ces propriétés peut lui être préféré. Dans ce cas l'évaluation paresseuse et des optimisations dynamiques permettent de rendre ce schéma raisonnablement efficace. A l'inverse,

nous voulons inciter le programmeur à se censurer sur certaines optimisations non indispensables. Dans un premier temps, il doit se limiter à définir un code le plus général et le plus composable possible.

Tous les travaux présentés dans ce document (depuis la prise en compte du flot dynamique de contrôle, jusqu'au tissage statique de code en passant par la prise en compte des flots dynamique de données) vont dans ce sens. Avant de présenter un programme de recherche, je souhaite revenir sur une définition des aspects que j'avais proposé et la revisiter à l'aune des années écoulées.

10.2 Revisiter le passé

Dès 2004 je me suis fais une opinion générale sur la programmation par aspects ou plutôt sur son potentiel [39]. J'ai ainsi listé ce que la programmation par aspects était/pourrait être/devrait être mais aussi ce qu'elle n'était pas/ne pourrait pas être/ne devrait pas être. Je me propose ici une dizaine d'années après de revisiter ma définition et de la confronter à la réalité du domaine.

Mon article était structuré en deux grandes parties. La première partie "AOP is not" caractérisait en creux la programmation par aspect, la seconde partie "AOP is" caractérisait en plein la programmation par aspect. Enfin, je proposais une définition restreinte de la programmation par aspects dont je reprends ici les principaux points.

La programmation par aspects n'est pas :

- "l'introduction *dynamique* de nouvelles préoccupations". Le problème de la tyrannie de la décomposition primaire apparait statiquement lors de la programmation d'une nouvelle fonctionnalité. Même si dans certains cas, la nouvelle fonctionnalité doit prendre en compte des propriétés dynamiques du programme de base comme sa pile d'appel, son arbre d'appel ou encore sa trace d'exécution, la nouvelle fonctionnalité n'est pas introduite/programmée dynamiquement. Ceci n'a donc aucun rapport avec des mécanismes comme le chargement dynamique de classe ou encore le changement dynamique (et réflexif) de classe.

- "la réutilisation de code". Si un grand nombre de travaux permettent de factoriser du code (qui sera par la suite entrelacé avec le programme de base), la factorisation n'est qu'une étape vers la réutilisation qui elle transforme un code boîte blanche en boîte noire (voire grise).

- "la programmation réflexive". L'essentiel des recherches sur la programmation par aspects a été mené dans les langages non réflexifs ou peu réflexifs (à commencer par Java qui a des capacités de reflexion limitées). La réflexion peut par contre être un puissant outil de mise en oeuvre, mais au même titre que les transformations de programme.

- "restreinte aux langages à objets". La programmation par aspects a été démocratisée par AspectJ dans un cadre objet mais des travaux ont été menés dans d'autre cadres (par exemple fonctionnel [DWWW08] ou impératif [CKFS01]).

- "restreinte aux entités syntaxiques". Un aspect qui utilise l'opérateur cflow parle de la pile d'appel du programme de base. On a vu que d'autres aspects nécessitent d'inspecter sa trace d'exécution ou encore son flot de données par exemple.

- "restreinte aux préoccupations non fonctionnelles (c.a.d. les services à la Corba/EJB)". Pour exemple le tutoriel officiel d'AspectJ propose un aspect pour rafraîchir l'affichage d'un arbre d'objets graphique ou encore la detection de collision dans un jeu vidéo.

- "une solution à tous les problèmes", "magique", "gratuit". Après des promesses excessives la programmation par aspects a suivi un cycle de vie classique [Ste12]. Et de nombreux travaux se sont intéressés à optimiser ce qui avait un coût (par exemple en évaluant partiellement les points de coupe [KOL07]).

- "la factorisation du code". L'exemple introduisant l'opérateur cflow au début de ce document ne concerne pas la factorisation d'un code dupliqué mais la réunion de deux morceaux de code arbitrairement distants.

- "la quantification". Un point de coupe AspectJ très bien peut être totalement instancié, il ne contient alors pas de $*$ et donc aucune quantification.

La programmation par aspects est :

- "une façon d'attaquer un problème réel". On a vu que le problème de la tyrannie de la décomposition primaire est central au génie logiciel.

- "une structuration du code dispersé". A partir du moment où deux (ou plus) morceaux de code sont nécessaires à la programmation d'une fonctionnalité on peut légitimement se poser des questions de flots (de contrôles et de données) entre ces deux morceaux.

- "basée sur les points de coupe". On ajoute pas une fonctionnalité ex-nihilo (sinon elle est autonome par définition) donc il faut bien parler de l'existant/du programme de base.

- "basée sur les points de coupe sémantiques". Les points de coupes sémantiques reposent sur des informations dynamiques telles que la pile ou l'arbre d'appel. Ceci permet de programmer des aspects qui ont des liens complexes avec le programme de base.

- "basée sur des actions sémantiques qui vérifient des propriétés". La notion d'abstraction est centrale au génie logiciel. Un aspect doit être programmé au "bon" niveau, c'est-à-dire en décrivant de la manière la plus abstraite possible quels sont les rapports entre un aspect et le programme de base. On préférera ainsi spécifier des points de coupes comme : "l'objet graphique le plus petit possible mais qui englobe tous les objets graphiques qui ont été déplacés" ou encore "toutes les lectures de données qui dépendent du mot de passe".

- "basée sur la composition des aspects". Il est naturel de se poser la question de la composition des aspects entre eux puisqu'ils doivent cohabiter dans le logiciel complet. A noter que la composition de morceaux de code ne devrait pas se réduire à les ordonner (la composition d'un aspect introduisant du rouge avec un autre introduisant du bleu devrait elle produire du magenta ?) et que la composition ne se résume pas à parler des points de programmes communs à deux aspects (par exemple un aspect qui incrémente une variable à un point de programme et un aspect qui décrémente la même variable à un autre point peuvent interagir).

- "un mécanisme langage". Ma philosophie est qu'il faut proposer des langages aux programmeurs pour être utile et utilisé. C'est aussi le choix qu'on fait les concepteurs de AspectJ.

- "une solution à la programmation incrémentale". Si on considère toutes les fonctionnalités en même temps on pourra faire des choix de décomposition informés qui permettront parfois d'éviter le problème de la décomposition primaire. Quand on programme incrémentalement, le problème de la tyrannie de la décomposition primaire apparait encore plus rapidement.

- “une solution à la dispersion de code”. La réalisation effective d'une fonctionnalité peut nécessiter de disperser son code. La programmation par aspects réunit ces morceaux.

Ma définition restreinte de la programmation par aspects était : “Un ensemble de mécanismes langage qui permettent d'introduire dans une application de base des fonctionnalités non anticipées. Sans ces mécanismes, l'application de base devrait être modifiée en plusieurs endroits. La programmation par aspects permet de rendre ces fonctionnalités modulaires. Elle permet de comprendre une application sans en lire le code tissé.”

Dix ans plus tard je trouve cette définition toujours d'actualité. Et je pense que tous mes travaux ont été menés en cohérence avec elle.

Mon seul regret est peut être qu'aucune théorie fondatrice et unificatrice de la programmation par aspect ne soit apparue (à comparer au lambda-calcul pour les langages fonctionnels par exemple). Les travaux tiennent donc il me semble plus de l'artisanat, et pose à chaque fois un cadre.

10.2.1 Le futur

Que peut nous réserver demain le problème de la modularité ? Tout d'abord une constatation : la composition (et la décomposition) sont au cœur de la maitrise de toute activité complexe et ne risque pas de disparaître.

AOP est mort, vive AOP. La conférence phare AOSD créée en 2002 a été rebaptisée Modularity en 2012. On pourrait penser que la communauté est morte. De sa surspécialisation ? De ses promesses irréalistes évidemment non tenues ? [Ste12] Je pense plutôt que la communauté affiche tardivement sa thématique cœur qui a toujours été présente. Il était temps. Que restera-t-il alors de AOP demain ? Essentiellement, le mot “aspect” (et quelques autres comme peut-être “pointcut”, “crosscuting”), qui a le mérite de nommer et donc de reconnaitre l'existence du problème de l'impossible décomposition modulaire (unique) des problèmes suffisamment complexes.

Je pense que pendant encore longtemps on va assister au développement des canevas anticipants tels ou tels types de préoccupations non modulaires. Ces canevas pourront être soutenus par des langages dédiés de plus haut niveau destinés à des utilisateurs humains et proposant possiblement un certain nombre de garanties. Mais ces propositions seront dédiées à des préoccupations données bien particulières (sécurité, traçabilité, transaction, etc.).

A plus long terme, je pense que des nouveaux paradigmes plus généralistes de programmation non modulaire modulaire verront enfin le jour et proposeront des outils pour se démocratiser. Je pense surtout que la solution pourra venir de la convergence de la programmation “in the large” avec des algorithmes biologiques, ou stochastiques. De l'émancipation des programmes discrets, des mécanismes d'horlogerie si fragiles qu'une seule instruction “grain de sable” peut tout mettre en défaut. L'analogie avec le vivant a ici tout son intérêt et les systèmes du futur pourraient bien s'inspirer de la nature pour faire cohabiter différents systèmes (comme les systèmes nerveux et hormonaux par exemple), fournir de la redondance et du continu.

Enfin concernant la dérivation de programme, je pense qu'elle a un rôle important à jouer (pour obtenir des logiciels fiables et peut être enfin sortir de l'industrie de prototypes bricolés). Par contre je pense que si elle est bien adaptée aux structures statiques, il est beaucoup plus difficile de l'adapter à des structures dynamiques si présentes dans le cadre de la programmation par aspects. Le problème ici ne viendra pas tant que la technique mais de la difficulté à formuler des spécifications.

10.3 Un programme de recherche

Mes recherches passées ont été éclectiques (même si j'ai réussi, je l'espère, à raconter une histoire cohérente). Mon programme de recherche est dans la continuité. Il couvre donc des travaux assez variés mais qui s'intéressent tous de près ou de loin à la modularité.

10.3.1 Objectif(s)

Mes objectifs restent de fournir des supports langages, si possible spécialisés, afin de favoriser la programmation modulaire dans des situations classiquement non modulaires. Notamment, je souhaite inciter les programmeurs à ne pas se censurer (quand les fonctionnalités deviennent très difficiles à programmer de par leur non modularité, des programmeurs peuvent y renoncer), à ne pas surspécialiser ou surspécifier leurs programmes (plus les programmes sont généralistes plus ils sont réutilisables et simples).

Je liste maintenant différentes pistes de travail que j'ai structuré en deux grandes familles : les travaux dans un cadre dynamique (manipulation de l'exécution) et dans un cadre statique (manipulation de code avant l'execution).

10.3.2 Dans un cadre dynamique

Certaines formes de modularité sont intrinsèquement dynamiques (en tout cas dans leur définition la plus générale). Nous en décrivons ici deux qui nous semblent fondamentales.

10.3.2.1 La paresse

L'évaluation paresseuse est très naturelle dans un cadre fonctionnel pur où les dépendances entre expressions sont explicites et l'absence d'effet de bord permet d'ordonner librement l'évaluation des expressions. Dans un cadre impératif, la programmation paresseuse est beaucoup plus complexe. En effet les dépendances entre blocs sont à la fois sur-spécifées (obligation de composer séquentiellement deux blocs dans un ordre ou dans l'autre même lorsqu'ils sont commutatifs) et sous-spécifiées (la dépendance entre deux blocs passe implicitement par l'état global et les variables partagées). Pourtant, la programmation paresseuse est loin d'être absente dans un cadre impératif. De nombreux programmes se comportent en producteur-consommateur mais l'orchestration de ces comportement est à la charge du programmeur. On pourra penser à l'exemple d'un arbre de jeu potentiellement infini qu'un programme impératif doit construire et explorer en même temps, et non construire puis explorer, pour éviter la non terminaison ou des calculs inutiles prohibitifs. Pour des raisons de performance un certain nombre de cas ont déjà été traités et sont offerts via des bibliothèques aux programmeurs : par exemple les entrés-sorties sont typiquement tamponnées et à un grain plus gros les pipes de unix proposent un mécanismes général de batch paresseux (c.a.d. de flot).

La paresse est donc un bon candidat (utile et complexe) à la recherche de supports dans un cadre impératif. La paresse favorisera comme dans la cadre fonctionnel la programmation d'entités générales et modulaires, tout en garantissant des performances raisonnables. La paresse en soi et les différentes stratégies d'évaluation plus ou moins strictes peuvent être vues comme de la programmation non modulaire qui coupe à travers les composants producteurs et les composants consommateurs. La paresse introduira aussi la notion d'évaluation spéculative dans un cadre impératif, qui pourra servir d'angle d'étude de la programmation concurrente.

Enfin, il faut noter que la programmation paresseuse nécessite de réifier les calculs mis en attente. Ceci ouvre la voie aux optimisations dynamiques comme par exemple la simplification

de la double application d'un calcul idempotent ou encore la suppression pure et simple de deux calculs inverses l'un de l'autre.

10.3.2.2 L'espace temps

Depuis toujours la programmation est un compromis entre l'utilisation de ressources mémoire et calcul : une possibilité est de ne mémoriser que le minimum nécessaire et de recalculer à la demande ce qui en découle (par exemple le nombre d'éléments d'une collection), l'autre possibilité est de mémoriser des informations redondantes (comme les éléments d'une collection et leur nombre) ce qui se traduit par une consommation mémoire accrue au bénéfice de moins de recalculs (en général, car il faut tout de même mettre à jour toutes les informations mémorisées en cas de modifications d'une donnée afin de conserver leur cohérences). De manière générale, une approche n'est pas supérieure à l'autre. Cela dépend de la ressource à privilégier ; cela dépend aussi de la politique d'utilisation des données : si les données sont jamais ou rarement modifiées il peut être rentable de favoriser le temps au dépend de l'espace, si au contraire les données sont plus souvent modifiées que consultées la mise à jour de propriétés peut s'avérer couteuse et on pourra favoriser l'espace au profit du temps. Bien sûr, la politique d'utilisation des données peut évoluer pendant la durée de vie d'une application (plus de consultation à certains moment et plus de modifications à d'autres) et ne concerner qu'un sous ensemble des données (par exemple le sous ensemble affiché des données, les données reliées à un utilisateur actif à un moment donné).

On le voit le compromis espace temps est fondamental en programmation mais il peut être complexe et dynamique. Ce compromis est généralement laissé à la charge du programmeur qui doit gérer souvent manuellement et explicitement une sorte de gestionnaire de mémoire plus ou moins sophistiqué. Par exemple, dans certains cas, il va devoir jongler entre les données présentes en mémoire et celles recalculables ou bien stockées sur une mémoire de masse via un système de pooling ou encore de pointeurs faibles.

Dans l'idéal, on aimerait que le programmeur fournisse d'une part les données et les algorithmes de calculs et d'autre part une politique (possiblement dynamique) de gestion du compromis espace temps. La solution proposée permettrait de spécifier des politiques plus ou moins sophistiquées et surtout de garantir leur cohérence : en évitant les calculs inutiles lorsque par exemple une donnée est modifiée plusieurs fois avant que les propriétés qui en dépendent ne soient consultées, ou encore en assurant qu'une propriété est bien à jour vis à vis de ses données lorsqu'elle est consultée. On peut imaginer des politiques qui répercutent les modifications au plus tôt ou au plus tard, qui effectuent des simplifications dynamiques (suppression des mises à jour inutiles) ou encore qui passent d'une politique à l'autre (bascule vers une représentation redondante des données ou inversement).

10.3.3 Dans un cadre statique

L'exécution d'un programme est en général hautement dynamique (en général les chemins d'exécutions sont différents pour chaque jeu de données sauf dans des cas très particuliers, par exemple certaines applications météorologiques qui exécutent toujours les mêmes calculs). Par contre la définition d'un programme est intrinsèquement statique, si l'on omet les fantaisies de la réflexion et autre chargement dynamique de classes. Nous balayons ici différents travaux qui s'attachent à faciliter la tache du programmeur : à savoir écrire du code.

10.3.3.1 Tissage et tranchage

Le tranchage aussi connu sous le nom anglais de slicing est une transformation de code qui extrait un sous ensemble du programme nécessaire pour calculer un sous ensemble du résultat du programme. Cette transformation a essentiellement été utilisée jusqu'à présent à des fins de débogage. Intuitivement elle correspond à une projection des instructions de calcul et de contrôle du programme sur un sous ensemble des variables et de leurs dépendances. On peut voir le tissage d'aspects comme une transformation inverse qui injecte dans un programme (de base) des morceaux de code (advice) permettant de calculer en plus du résultat de base des résultats complémentaires. Cette définition a du sens pour des aspects observateurs (qui ne modifient pas le flot de contrôle du programme de base). On peut probablement l'étendre un peu, en considérant que le slicing peut générer des programmes dont le flot de contrôle peut être simplifié. En tout cas, les intuitions sont si proches qu'il serait intéressant d'étudier sous quelles conditions ces deux transformations sont bien inverses et d'explorer les différents tisseurs résultants de l'inversion des algorithmes de slicing existants.

10.3.3.2 Les transformations inverses

De manière naturelle on pourrait s'intéresser aux transformations de programmes et à leurs inverses ou aux transformations de programmes bidirectionnelles. Ces dernières sont des transformations de programmes limitées mais inversibles par construction, qui exploitent le fait de construire un contexte lors de la transformation "aller" pour mémoriser les information perdues (par projection) et utilisent ce contexte lors de la transformation "retour" pour injecter dans le terme initial, représenté par le contexte collecté, les transformations effectuées.

Qui dit "transformations", dit "vues". Et qui dit "inversion", dit "liens entre ces vues". Dans cette approche il serait intéressant d'étudier les (paires, et plus de) transformations inversibles célèbres, comme par exemple la transformation en passage de continuation/en style direct, ou encore la transformation en lambda-lifting/lambda dropping, afin de voir dans quelle mesure ces paires de transformation pourraient permettre de décrire un unique programme dans deux vues et que les modifications dans une vue puissent être répercutées dans l'autre afin de générer un unique code source à partir de programmation dans des vues multiples. La programmation fonctionnelle de part ses bonnes propriétés est le lieu privilégié des transformation de programme. Une exploration des transformations existantes devrait fournir une première liste de transformations candidates à une telle étude.

Les transformations bidirectionnelles sont plus générales que les transformations inversibles, puisque la collecte d'un contexte lors de la transformation aller permet de s'affranchir de la contrainte d'être naturellement une bijection. Cette famille de transformations devrait donc ouvrir plus de portes à l'exploration de la programmation modulaire non modulaire.

10.3.3.3 Dérivation de programme par fusion de code

Il est naturel pour des raisons de modularité de décomposer un programme en producteurs et consommateurs. Dans la cadre fonctionnel la transformation de déforestation consiste à fusionner un producteur et un consommateur afin d'éviter la création de structures de données intermédiaires jetables (d'où le nom de la transformation) et à factoriser et fusionner les passes de visite/itération sur les structures de données.

Dans un cadre plus général et plus puissant comme celui de la dérivation de programme avec outil de preuve, comme Coq, il serait intéressant d'explorer dans quelle mesure la fusion de programmes permettrait de dériver des algorithmes complexes pourtant simples à exprimer

dans la vue producteur et dans la vue consommateur. C'est l'approche, Coq mis à part, qu'utilise Bird dans beaucoup de ses perles [B10]. Par exemple, les algorithmes de filtrages utilisés dans la programmation par contraintes peuvent être vus comme l'optimisation de deux algorithmes force brute de génération d'une part et test d'autre part. Certains algorithmes sont probablement trop ad-hoc pour retrouver un lien avec un algorithme force brute mais je pense que ce n'est pas le cas de tous. Par exemple, j'ai commencé à regarder l'algorithme de consistance des bornes de la contrainte de somme croissante [BCPC12] et je pense que cet algorithme pourrait être formellement dérivé de l'algorithme force brute et des invariants du problème. Cette approche devrait être répétée avec plusieurs algorithmes afin de voir si une stratégie/structuration de preuve pourrait se dégager. Dans le cas ou cela s'avérerait possible on obtiendrait un cadre pour déclarer des algorithmes de filtrage et générer leur version optimisée fusionnée.

10.3.3.4 Tissage de documents en langue naturelle

Je finis ce programme de recherche par une dernière piste plus atypique.

La programmation de code repose sur des principes comme : ne pas dupliquer mais factoriser/abstraire le code, introduire une sous-classe qui amende sinon réutilise certaines définitions, diviser pour régner et décomposer hiérarchiquement le code (bottom-up/top-down), éviter/limiter les références en avant qui complexifient la construction de données, choisir une structuration et vivre avec même si certaines fonctionnalités vont s'exprimer de manière transverse, etc.

Je pense que ces principes (et d'autres) se retrouvent dans la rédaction de documents techniques en langue naturelle. Par exemple, on va tenter de ne pas copier coller du texte qui pourrait ensuite par le jeu des éditions successives diverger. On va commencer par des généralités, puis aller vers du particulier en ne précisant que ce qui change. On va structurer hiérarchiquement un document en parties, chapitres, sections, sous-sections, etc. On va éviter/limiter les références en avant qui rendent la lecture non linéaire. On va choisir une structuration et ajouter différents guides de lecture qui concernent un sous ensemble du document et traversent la structure, etc.

Notamment, un document peut être lu par différents lecteurs et pour différentes raisons. Si je me place par exemple dans le domaine des règles de jeux de société, ces documents sont clairement techniques et peuvent être utilisés par un joueur novice qui souhaite apprendre à jouer, par un joueur novice qui souhaite n'utiliser qu'un sous ensemble des règles, par un joueur novice avec ce jeu mais qui connait d'autres jeux de la même série et souhaite en connaître rapidement les points communs et les divergences, par un testeur qui souhaite évaluer l'impact d'une nouvelle règle, par un joueur expérimenté qui a besoin d'un rappel procédural des étapes à suivre, par un joueur expérimenté qui cherche à verifier un point de règle qui peut s'appliquer à de nombreuses étapes du tour de jeu, par un joueur qui a sélectionné un certain nombre de règles optionnelles et souhaite avoir une vision instanciée/spécialisée de la règle dans cette configuration.

Toutes ces utilisations et bien d'autres montrent que différentes vues sont utiles : vision de haut niveau à la façon table des matières, vision bas niveau procédurale à la façon séquence de jeu détaillée, vision différentielle entre deux versions de règles, vision déstructurée en isolant chaque mécanisme qui peut intervenir à plusieurs étapes, vision spécialisée en remplaçant une référence à une règle par la règle elle même afin d'avoir localement un texte autoporteur même si c'est au prix de duplication de texte, etc.

En pratique ces différents besoins existent et amènent à différent documents. Par exemple, la société GMT publie des jeux de société de simulation historique qui comportent la plupart du temps un livre de règles (commençant parfois par "n'essayez pas de lire ce document en premier") et un play book destiné à l'apprentissage du jeu (contenant souvent entre autre un

exemple de partie détaillée). Les règles sont aussi émaillées de notes de conception équivalent des commentaires/spécifications dans le monde du logiciel. Ce "play book" et ce "rule book" référencent bien sur les même règles mais étant développés indépendamment et manuellement, ils sont la proie de nombreuses incohérences. De plus, les règles du jeu bénéficient après leur sorties d'errata et de faq qui doivent être intégrés dans des "living rules" à destination des joueurs connectés, voire des futurs clients d'une réimpression ou d'une nouvelle édition. Toutes ces manipulations se font simplement en éditant des documents et en tentant de ne pas introduire d'incohérences.

Dans l'idéal un unique document de référence devrait exister et les autres devraient être automatiquement générés à partir de lui. Sans définir une sémantique complète de la langue naturelle (on ne cherche pas à programmer un arbitre), il devrait aussi être possible d'analyser plus ou moins syntaxiquement un document de référence puis à partir de description de structures alternatives de générer les autres. Ainsi, on pourrait imaginer de concevoir des analyses pour lever des alertes lorsque deux paragraphes sont à insérer au même endroit indice d'un conflit potentiel de règles. Ou encore lorsque la suppression d'une règle crée une étape vide.

Il existe certainement d'autres domaines plus "sérieux" que celui des jeux qui permettraient d'étudier la conception, la génération et l'analyse de documents techniques modulaires non modulaires. Une piste à explorer me semble être celle des modes d'emploi. Ce travail est à rapprocher un peu de HyperJ [TOHS99] qui si il ne propose pas de générer différentes vues d'un programme, s'intéresse à l'organisation gros grain du code (et non à la sémantique précise de chaque ligne) et définit l'application complète comme la superposition de plusieurs tranches (slices) en se basant sur les identifiants qu'elles partagent.

10.4 Conclusion

La tyrannie de la décomposition primaire a encore de beau jour devant elle. Elle est intrinsèque à tout problème complexe qui contient plusieurs structures. Dans le cadre du génie logiciel elle est particulièrement présente, les logiciels étant des artefacts complexes et hautement structurés (notamment pour en maitriser l'échelle). Devant un problème il faut l'identifier puis tenter de lui apporter des solutions. C'est ce que le communauté de la programmation par aspects s'est attachée à faire avec plus ou moins de succès.

Je pense que les solutions doivent viser le fabriquant de logiciel qui est en première ligne : le programmeur. A ce niveau toute la rigidité d'un langage de programmation se fait ressentir et n'est pas négociable. Il y a du travail dans le contexte dynamique à revisiter des compromis fondamentaux, comme dans le contexte statique à faire cohabiter, voire coopérer, différentes structures. Les documents structurés en langue naturelle semble une voie prometteuse puisqu'ils allient différentes structures pour différents utilisateurs à une facilité de composition des "instructions" de base (juxtaposition de deux paragraphes).

Ma connaissance limitée (!) des autres domaines scientifiques m'a exposé à des instances de notions que j'avais appris dans le monde de l'informatique. Par exemple, les classifications du monde essentiellement arborescentes, les structures fractales (récursive) dans la nature, le calcul de point (plus ou moins) fixe des système dynamiques, l'abstraction d'un mobile à son centre de gravité pour calculer sa trajectoire, les systèmes en couches et autres membranes biologiques, etc. Je n'ai jamais trouvé jusqu'à présent une structuration d'un mécanisme non informatique en terme d'aspects. Il y a certes l'analogie de la maison et de ses différents plans (de masse, électrique, de chauffage, etc.) correspondants à différents corps de métier. Mais même dans ce cas là, je trouve qu'il y a une absence de régularité dans les structures qui justifierait l'existence de points de coupe déclaratifs/sémantiques. En attendant de trouver la preuve du contraire, j'ai

aujourd'hui la conviction que la programmation par aspects est et restera un concept purement artificiel et informatique. Ce qui ne lui enlève rien de son importance. Bien au contraire.

Chapitre 11

Publications

Bibliographie

Journaux internationaux avec comité de lecture

[1] Paul Leger, Éric Tanter, and Rémi Douence. Modular and flexible causality control on the web. Sci. Comput. Program., 78(9) :1538–1558, 2013.

[2] Charles Prud'homme, Xavier Lorca, Rémi Douence, and Narendra Jussien. Propagation engine prototyping with a domain specific language. Constraints, 2013.

[3] Simplice Djoko Djoko, Rémi Douence, and Pascal Fradet. Aspects preserving properties. Sci. Comput. Program., 77(3) :393–422, 2012.

[4] Éric Tanter, Johan Fabry, Rémi Douence, Jacques Noyé, and Mario Südholt. Scoping strategies for distributed aspects. Sci. Comput. Program., 75(12) :1235–1261, 2010.

[5] Rémi Douence and Pascal Fradet. The next 700 krivine machines. Higher-Order and Symbolic Computation, 20(3) :237–255, 2007.

[6] Marc Ségura-Devillechaise, Jean-Marc Menaud, Nicolas Loriant, Rémi Douence, Mario Südholt, Thomas Fritz, and Egon Wuchner. Dynamic adaptation of the squid web cache with arachne. IEEE Software, 23(1) :34–41, 2006.

[7] Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. An expressive aspect language for system applications with arachne. In Awais Rashid and Mehmet Aksit, editors, Transactions on Aspect-Oriented Software Development I, volume 3880 of Lecture Notes in Computer Science, pages 174–213. Springer, 2006.

[8] Rémi Douence and Mario Südholt. A generic reification technique for object-oriented reflective languages. Higher-Order and Symbolic Computation, 14(1) :7–34, 2001.

[9] Rémi Douence and Pascal Fradet. A systematic study of functional language implementations. ACM Trans. Program. Lang. Syst., 20(2) :344–387, 1998.

Conférences internationales avec comité de lecture

[10] Rémi Douence, and Nicolas Tabareau. Lazier Imperative Programming. PPDP, 2014.

[11] Julien Cohen, Rémi Douence, and Akram Ajouli. Invertible program restructurings for continuing modular maintenance. In Tom Mens, Anthony Cleve, and Rudolf Ferenc, editors, CSMR, pages 347–352. IEEE, 2012.

[12] Diana Allam, Rémi Douence, Hervé Grall, Jean-Claude Royer, and Mario Südholt. A message-passing model for service oriented computing. In Karl-Heinz Krempels and José Cordeiro, editors, WEBIST, pages 136–142. SciTePress, 2012.

[13] Abdelhakim Hannousse, Rémi Douence, and Gilles Ardourel. Composable controllers in fractal : Implementation and interference analysis. In EUROMICRO-SEAA, pages 51–54. IEEE, 2011.

[14] Abdelhakim Hannousse, Rémi Douence, and Gilles Ardourel. Static analysis of aspect interaction and composition in component models. In Ewen Denney and Ulrik Pagh Schultz, editors, GPCE, pages 43–52. ACM, 2011.

[15] Éric Tanter, Johan Fabry, Rémi Douence, Jacques Noyé, and Mario Südholt. Expressive scoping of distributed aspects. In Kevin J. Sullivan, editor, Proceedings of the 8th International Conference on Aspect-Oriented Software Development, AOSD 2009, Charlottesville, Virginia, USA, March 2-6, 2009, pages 27–38. ACM, 2009.

[16] Rémi Douence, Xavier Lorca, and Nicolas Loriant. Lazy composition of representations in java. In Alexandre Bergel and Johan Fabry, editors, Software Composition, volume 5634 of Lecture Notes in Computer Science, pages 55–71. Springer, 2009.

[17] Simplice Djoko Djoko, Rémi Douence, and Pascal Fradet. Aspects preserving properties. In Robert Glück and Oege de Moor, editors, Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2008, San Francisco, California, USA, January 7-8, 2008, pages 135–145. ACM, 2008.

[18] Simplice Djoko Djoko, Rémi Douence, and Pascal Fradet. Specialized aspect languages preserving classes of properties. In Antonio Cerone and Stefan Gruner, editors, Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa, 10-14 November 2008, pages 227–236. IEEE Computer Society, 2008.

[19] Luis Daniel Benavides Navarro, Rémi Douence, Fabien Hermenier, Jean-Marc Menaud, and Mario Südholt. Aspect-based patterns for grid programming. In 20th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2008, October 29 - November 1, 2008, Campo Grande, MS, Brazil, pages 141–148. IEEE Computer Society, 2008.

[20] Luis Daniel Benavides Navarro, Rémi Douence, and Mario Südholt. Debugging and testing middleware with aspect-based control-flow and causal patterns. In Valérie Issarny and Richard E. Schantz, editors, Middleware, volume 5346 of Lecture Notes in Computer Science, pages 183–202. Springer, 2008.

[21] Luis Daniel Benavides Navarro, Mario Südholt, Rémi Douence, and Jean-Marc Menaud. Invasive patterns for distributed programs. In Robert Meersman and Zahir Tari, editors, On the Move to Meaningful Internet Systems 2007 : CoopIS, DOA, ODBASE, GADA, and IS, OTM, volume 4803 of Lecture Notes in Computer Science, pages 772–789. Springer, 2007.

[22] Rémi Douence, Didier Le Botlan, Jacques Noyé, and Mario Südholt. Concurrent aspects. In Stan Jarzabek, Douglas C. Schmidt, and Todd L. Veldhuizen, editors, Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proceedings, pages 79–88. ACM, 2006.

[23] Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. An expressive aspect language for system applications with arachne. In Mira Mezini and Peri L. Tarr, editors, Proceedings of the 4th International Conference on Aspect-Oriented Software Development, AOSD 2005, Chicago, Illinois, USA, March 14-18, 2005, pages 27–38. ACM, 2005.

[24] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In Gail C. Murphy and Karl J. Lieberherr, editors, Proceedings of the 3rd International Conference on Aspect-Oriented Software Development, AOSD 2004, Lancaster, UK, March 22-24, 2004, pages 141–150. ACM, 2004.

[25] Rémi Douence and Luc Teboul. A pointcut language for control-flow. In Gabor Karsai and Eelco Visser, editors, Generative Programming and Component Engineering : Third International Conference, GPCE 2004, Vancouver, Canada, October 24-28, 2004. Proceedings, volume 3286 of Lecture Notes in Computer Science, pages 95–114. Springer, 2004.

[26] Yann-Gaël Guéhéneuc, Rémi Douence, and Narendra Jussien. No java without caffeine : A tool for dynamic analysis of java programs. In 17th IEEE International Conference on Automated Software Engineering (ASE 2002), 23-27 September 2002, Edinburgh, Scotland, UK, pages 117–. IEEE Computer Society, 2002.

[27] Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In Don S. Batory, Charles Consel, and Walid Taha, editors, Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, Pittsburgh, PA, USA, October 6-8, 2002, Proceedings, volume 2487 of Lecture Notes in Computer Science, pages 173–188. Springer, 2002.

[28] Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. In Akinori Yonezawa and Satoshi Matsuoka, editors, Reflection, volume 2192 of Lecture Notes in Computer Science, pages 170–186. Springer, 2001.

[29] Robert Allen, Rémi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In Egidio Astesiano, editor, Fundamental Approaches to Software Engineering, 1st Internationsl Conference, FASE'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings, volume 1382 of Lecture Notes in Computer Science, pages 21–37. Springer, 1998.

[30] Rémi Douence and Pascal Fradet. Towards a taxonomy of functional languages implementations. In Manuel V. Hermenegildo and S. Doaitse Swierstra, editors, Programming Languages : Implementations, Logics and Programs, 7th International Symposium, PLILP'95, Utrecht, The Netherlands, September 20-22, 1995, Proceedings, volume 982 of Lecture Notes in Computer Science, pages 27–44. Springer, 1995.

Conférences francophones avec comité de lecture

[31] Rémi Douence and Mario Südholt. Un modèle et un outil pour la programmation par aspects événementiels. L'OBJET, 9(1-2) :105–117, 2003.

[32] Charles Prud'homme, Xavier Lorca, Rémi Douence, and Narendra Jussien. Prototyper des moteurs de propagation avec un dsl. In JFPC, 2013.

Ateliers internationaux avec comité de lecture

[33] Éric Tanter, Nicolas Tabareau, and Rémi Douence. Taming aspects with membranes. In Shmuel Katz, Gary T. Leavens, and Hidehiko Masuhara, editors, FOAL, pages 3–8. ACM, 2012.

[34] Abdelhakim Hannousse, Gilles Ardourel, and Rémi Douence. Views for aspectualizing component models. In Proceeding of the 9th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS 2010), 03 2010.

[35] Luis Daniel Benavides Navarro, Rémi Douence, Angel Nunez, and Mario Sudholt. Lts-based semantics and property analysis of distributed aspects and invasive patterns. In Proc. of the 3rd International Workshop on Aspects, Dependencies, and Interactions (ADI'08), July 2008.

[36] Rémi Douence. Relational aspects for context passing beyond stack inspection. In International Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT'06), March 2006.

[37] Rémi Douence, Didier Le Botlan, Jacques Noyé, and Mario Sudholt. Towards a model of concurrent AOP. In International Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT'06), March 2006.

[38] Rémi Douence and Jacques Noyé. Towards a concurrent model of event-based aspect-oriented programming. In European Interactive Workshop on Aspects in Software (EIWAS 2005), Brussels, Belgium, September 2005.

[39] Rémi Douence. A restricted definition of aop. In European Interactive Workshop on Aspects in Software (EIWAS 2004), Berlin, Germany, September 2004.

[40] Rémi Douence and Narendra Jussien. Non-intrusive constraint solver enhancements. In Colloquium on Implementation of Constraint and Logic Programming Systems (CICLOPS'02), Copenhagen, Denmark, July 2002.

[41] Rémi Douence and Narendra Jussien. Non-intrusive constraint solver enhancements. In First AOSD workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), Enschede, The Netherlands, April 2002. University of Twente.

[42] Luciano Porto Barreto, Rémi Douence, Gilles Muller, and Mario Sudholt. Programming OS schedulers with domain-specific languages and aspects : New approaches for OS kernel engineering. In International Workshop on Aspects, Components, and Patterns for Infrastructure Software at AOSD, April 2002.

Chapitres de livres

[43] Jacques Noyé, Rémi Douence, and Mario Sudholt. Composants et aspects. In Mourrad Oussalah, editor, Ingénierie des composants : Concepts, techniques et outils, chapter 6, pages 169–195. Vuibert, 2005.

[44] Rémi Douence, Pascal Fradet, and Mario Sudholt. Trace-based aspects. In Mehmet Aksit, SiobhÃąn Clarke, Tzilla Elrad, and Robert E. Filman, editors, Aspect-Oriented Software Development, pages 201–218. Addison-Welsey, 2004.

Rapports

[45] Simplice Djoko Djoko, Rémi Douence, and Pascal Fradet. Proof of correctness of aspect transformations in the casb. Research Report D88, Network of Excellence in AOSD (AOSD-Europe), July 2007.

Thèse

[46] Rémi Douence. Décrire et comparer les mises en oeuvre de langages fonctionnels. PhD thesis, Université de Rennes I, 1996.

Travaux extérieurs référencés

[A01] James H. Andrews. Process-Algebraic Foundations of Aspect-Oriented Programming. REFLECTION, pp 187–209, 2001.

[AZ02] Davide Ancona et Elena Zucca. A Calculus of Module Systems. JFP, 12(2), PP 91–132, 2002.

[aM01] Agile Manifesto `http://agilemanifesto.org` 2001

[ACH05] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie J. Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam et Julian Tibble. Adding trace matching with free variables to AspectJ OOPSLA, pp 345–364, 2005.

[AGM06] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, et Klaus Ostermann. An overview of CaesarJ. TAOSD, LNCS 3880, pp 135 ?173, 2006.

[A84] Lennart Augustsson. A Compiler for Lazy ML. LISP and Functional Programming, pp 218–227, 1984.

[bCg] Software by Composition group. Morphogenic software, ibm t.j. watson research center. `http://researchweb.watson.ibm.com/morphogenic/` (via `archive.org` dÃl'sormais)

[BCPC12] Nicolas Beldiceanu, Mats Carlsson, Thierry Petit, Jean-Charles Régin. An O(nlog n) Bound Consistency Algorithm for the Conjunction of an alldifferent and an Inequality between a Sum of Variables and a Constant, and its Generalization. In ECAI, pages 145–150. IOS Press, 2012.

[BC90] Gilad Bracha et William Cook. Mixin-based Inheritance. OOPSLA/ECOOP, pp 303–311, 1990.

[NSV06] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvée. Explicitly distributed aop using awed. In Robert E. Filman, editor, AOSD, pages 51–62. ACM, 2006.

[B10] Richard Bird. Pearls of Functionnal Algorithm Design. Cambridge University Press, 2010.

[CR93] Alain Colmerauer and Philippe Roussel. The Birth of Prolog. HOPL, pp 37–52, 1993

[RS84] Jim des Rivières and Brian Cantwell Smith. The implementation of procedurally reflective languages. In Proceedings of the 1984 ACM Symposium on LISP and functional programming, LFP '84, pages 331–347, New York, NY, USA, ACM, 1984.

[CGT75] D. D. Chamberlin, J. N. Gray, and I. L. Traiger. Views, authorization, and locking in a relational data base system. In Proceedings of the May 19-22, 1975, national computer conference and exposition, AFIPS '75, pages 425–430, New York, NY, USA, ACM, 1975. ACM

[CKFS01] Yvonne Coady, Gregor Kiczales, Michael J. Feeley and Greg Smolyn. Using aspectC to improve the modularity of path-specific customization in operating system code ACM ESEC, pp 88–98, 2001.

[DWWW08] Daniel S. Dantas, David Walker, Geoffrey Washburn and Stephanie Weirich. AspectML : A polymorphic aspect-oriented functional programming language. ACM Toplas, 30(3), 2008.

[DT06] Christopher Dutchyn, David B. Tucker, et Shriram Krishnamurthi. Semantics and scoping of aspects in higher-order languages. SCP, 63(3), pp 207 ?239, 2006.

[FGM07] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations : A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst., 29(3), 2007.

[Han11] Abdelhakim Hannousse. Aspectualizing Component Models : Implementation and Interferences Analysis. PhD thesis, École des Mines de Nantes, Université Nantes Angers Le Mans, November 2011.

[Hoa85] C. A. R. Hoare. Communicating Sequential Processes. Prentice-Hall, 1985.

[Jac95] Daniel Jackson. Structuring z specifications with views. ACM Trans. Softw. Eng. Methodol., 4(4) :365–389, 1995.

[Bro95] Frederick P. Brooks Jr. The mythical man-month - essays on software engineering (2. ed.). Addison-Wesley, 1995.

[DN66] Ole-Johan Dahl et Kristen Nygaard. SIMULA - an ALGOL-based simulation language. CACM, 9(9), pp 671–678, 1966.

[FGP64] David J. Farber, Ralph E. Griswold et I. P. Polonsky. SNOBOL, A String Manipulation Language. JACM, 11(1), pp 21–30, 1964.

[H69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. CACM, 12(10), pp 576–580, 1969.

[KHH01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In Jørgen Lindskov Knudsen, editor, ECOOP, volume 2072 of Lecture Notes in Computer Science, pages 327–353. Springer, 2001.

[KR91] Gregor Kiczales and Jim Des Rivieres. The Art of the Metaobject Protocol. MIT Press, Cambridge, MA, USA, 1991.

[KOL07] Karl Klose, Klaus Ostermann and Michael Leuschel. Partial Evaluation of Pointcuts. PADL, LNCS 4354, pp 320–334, 2007.

[Lam83] Leslie Lamport. What Good is Temporal Logic ? IFIP Congress, pages, 657–668, 1983.

[LZ74] Barbara Liskov et Stephen N. Zilles. Programming with Abstract Data Types. SIGPLAN Notices, 9(4), pp 50–59, 1974.

[MK99] Jeff Magee et Jeff Kramer. Concurrency - state models and Java programs. Wiley, 1999.

[MK03] Hidehiko Masuhara et Kazunori Kawauchi. Dataflow Pointcut in Aspect-Oriented Programming. APLAS, pp 105–121, 2003.

[OMB05] Klaus Ostermann, Mira Mezini et Christoph Bockisch. Expressive Pointcuts for Increased Modularity. ECOOP Proc. LNCS 3586, pp 214–240, 2005.

[P72] David Lorge Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. CACM, 15(12), pp 1053–1058, 1972.

[S84] David A. Schmidt. The structure of typed programming languages. Foundations of computing series, MIT Press, 1984.

[Ste12] Friedrich Steimann. The paradoxical success of aspect-oriented programming. In Proceedings of OOPSLA, October 22-26, 2006, USA

[SS75] Gerald Jay Sussman and Guy L Steele Jr. Scheme : An interpreter for extended lambda calculus. in MEMO 349, MIT AI LAB, 1975.

[TOHS99] Peri L. Tarr, Harold Ossher, William H. Harrison and Stanley M. Sutton Jr. N Degrees of Separation : Multi-Dimensional Separation of Concerns. ICSE, pp 107–119, 1999.

[Wad87] Philip Wadler. Views : A way for pattern matching to cohabit with data abstraction. In POPL, pages 307–313, 1987.

[Wad98] Philip Wadler. The expression problem Discussion on the Java-Genericity mailing list, 1998.

[W71] Niklaus Wirth. The Programming Language Pascal In Acta Inf., 1, pp 35–63, 1971.

[Zen05] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem In FOOL, 2005.

Composition non modulaire modulaire

Rémi Douence

Mots-clés : composition, module, programme, flot de contrôle/données statique/-dynamique de programmes séquentiels/concurrents/distribués.

Ce document retrace mes différentes activités de recherche depuis ma thèse. Le fil conducteur de ces travaux est ma fascination pour la programmation modulaire mais aussi ses limites lorsque qu'il n'existe pas une décomposition modulaire d'un problème mais plusieurs décompositions qui doivent coexister. Ces recherches sont déclinées selon plusieurs axes : flot de contrôle et flot de données, comportements statiques et dynamiques, contextes séquentiels, concurrents et distribués.

Modular non-modular composition.

Rémi Douence

Keywords : composition, module, program, static/dynamic control/data flow of sequential/concurrent/distributed programs

This document survey my different research activities since I have defended my PhD. Thesis. The thread of these work is my fascination for modular programming but also its limits when there is not a single modular decomposition but several decompositions that must coexist. These researches are structured according to several axes: control and data flow, static and dynamic behavior, sequential, concurrent and distributed context.