Analyses et vérification des programmes à aspects

Simplice Djoko Djoko

UNIVERSITÉ DE NANTES
UFR SCIENCES ET TECHNIQUES

———

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

N° attribué par la bibliothèque

ANNÉE 2009

Analyses et vérification des programmes à aspects

———

THÈSE DE DOCTORAT

Discipline : Informatique et applications
Spécialité : Méthodes formelles et langages de programmation

Présentée
et soutenue publiquement par

Simplice Djoko Djoko

Le 26 juin 2009, devant le jury ci-dessous

Rapporteurs	J. LAWALL	*Professeur associé à l'Université de Copenhague*
	T. D'HONDT	*Professeur à la VUB (Vrije Universiteit Brussel)*
Examinateurs	C. Attiogbé	*Professeur des universités à l'Université de Nantes*
	J-P. Banâtre	*Professeurs des universités à l'Université de Rennes 1*

Directeur de thèse : Pascal Fradet Chargé de recherche à l'INRIA de Grenoble

Co-encadrant de thèse: Rémi Douence Maître assistant à l'EMN de Nantes

ED : ……
(Uniquement pour STIM et MTGC)

Remerciements

A *Mme Djoko*, née *Nguemchie Micheline*,
ma mère . . . .

Je voudrais commencer par remercier tout particulièrement les membres de mon jury :

Monsieur Jean-Pierre Banâtre, Professeur à l'Université de Rennes 1, pour m'avoir fait l'honneur d'accepter d'être le président de mon jury de thèse. Je tiens à souligner sa disponibilité comme celle de tous les autres membres du jury notamment lors du choix d'une date de soutenance qui n'est pas toujours chose facile.

Madame Julia Lawall, Professeur associé à l'Université de Copenhague, et *Monsieur Théo D'hondt*, Professeur à l'Université de Bruxelles, pour avoir accepter de juger ce travail. Donner son avis sur la qualité d'une thèse n'est pas chose aisée car il faut la lire intégralement, relever les points positifs et négatifs, et les améliorations possibles. De plus, c'est aux rapporteurs que revient la lourde charge d'autoriser (ou pas) la soutenance d'une thèse.

Monsieur Christian Attiogbe, Professeur à l'Université de Nantes, qui a accepté d'être membre de ce jury. Je suis sensible aux conseils et encouragements que lui et les autres membres du jury m'ont prodigués.

Monsieur Pascal Fradet, Chargé de recherche à l'Inria Rhône Alpes, qui est le directeur de cette thèse, et sans lequel ce travail n'aurait pas pu aboutir. Ces trois années et plus, passées à travailler à ses côtés ont été très enrichissantes tant sur le plan humain que professionnel. Je n'oublierai pas, ces discussions animées et fortes d'enseignements sur des sujets qui n'avaient aucun rapport avec la thèse. Tout comme ce sérieux, et soucis du détail et du travail bien fait.

Monsieur Rémi Douence, Maître assistant à l'École des Mines de Nantes, qui est le co-encadrant de cette thèse, sans qui aussi, je n'aurai pas pu réussir à aller au bout de cette aventure. En particulier grâce à son soucis du détail.

En général, je voudrais commencer par remercier le réseau d'excellence Européen AOSD-Europe qui a financé cette thèse. *Monsieur Alain Girault* et *Monsieur Pierre Cointe* respectivement responsables de l'équipe PopArt de l'Inria Rhône Alpes et de l'équipe ObAsCo de L'École des Mines de Nantes, pour m'avoir accueillis dans leurs équipes et mis à ma disposition les moyens nécessaires pour réaliser ce travail. Un clin d'oeil particulier pour le premier qui m'a initié au football du mardi, élément qui a été important pour mon intégration et mon équilibre pendant ces années de thèse.

Je n'oublie pas mes collègues de la machine à café et des parties de football, pour nos pauses instructives et pleines de joie qui m´ont permises de mieux replonger dans le travail. Il s´agit de *Franck Perignon*, *Vincent Acary*, *Jérôme Malick*, *Florent Cadoux*, *Bernard Brogliato*, *Jacques Noyé*, *Hervé Grall*, *Mario Sudholt*, *Angel Nunez*, *Luis Daniel Benavides Navarro*, *Kelly Garcés*, *Ali Assaf*, *Mayleen Lacouture*, *Gregor Goessler*, *Gwenael Delaval*, *Jean-Baptiste Raclet*, *Adrien Richard*, *Lies Lakhdar-Chaouch*, *Bhasker Vanamali*. Je ne manque pas aussi de remercier, *Diana Gaudin*, personnel administratif à l´école des mines de Nantes, pour son aide très utile dans les tâches administratives.

Enfin, je remercie les membres de ma famille et mes amis, qui m´ont toujours soutenus et encouragés pendant toutes mes années d´études. Avant ma soutenance, j´ai perdu un de mes grands frères, décédé, *Jean-Pierre Tchuinte Djoko*, à qui je dédie cette thèse.

Résumé

La programmation par aspects est un paradigme de programmation qui permet de mieux séparer les préoccupations d'une application. Un aspect est défini pour chaque préoccupation qui ne peut pas être isolée dans un module. Les aspects sont ensuite ajoutés au programme de base par un processus automatique appelé tissage. Cependant, l'expressivité des langages d'aspect généraux permet de modifier totalement la sémantique du programme de base (*par ex.,* un aspect peut remplacer certains appels de méthode par du code arbitraire). Ce comportement peut entraîner la perte des avantages (lisibilité, maintenabilité, réutilisabilité, *etc.*) d'une meilleure modularisation des préoccupations. Il devient impossible de raisonner sur le programme de base sans regarder le programme tissé.

Cette thèse apporte une réponse aux problèmes ci-dessus en définissant des catégories d'aspects dont l'impact sur la sémantique du programme de base reste sous contrôle. Pour chaque catégorie d'aspects, nous déterminons l'ensemble des propriétés du programme de base qui est préservé par tissage. L'appartenance d'un aspect à une catégorie est garantie par construction grâce à des langages d'aspect dédiés pour chaque catégorie. L'utilisation de ces langages assure que le tissage préservera l'ensemble des propriétés associé à la catégorie concernée. Les propriétés préservées sont représentées comme des sous ensembles de LTL et de CTL*. Nous prouvons formellement que quelque soit le programme de base, le tissage de n'importe quel aspect d'une catégorie préserve les propriétés de la catégorie correspondante.

Ces langages et catégories sont définis dans un cadre formel indépendant de tout langage de base ou d'aspect. L'expressivité de ce cadre est montrée en décrivant des primitives complexes de langages d'aspect comme AspectJ et CaesarJ et en effectuant une preuve de correction de transformation d'aspect.

Mots-clé : langage de programmation, programmation par aspect, sémantique formelle, logiques temporelles, preuves

Abstract

Aspect oriented programming is a paradigm aiming at improving the separation of concerns. Typically, an aspect is defined for a concern that can not be isolated in a module. Aspects are then added to the base program through an automatic process called weaving. However, the expressiveness of general aspect languages allows to completely change the semantics of the base program (*e.g.,* an aspect may insert arbitrary code). This jeopardizes the benefits (readability, maintainability, reusability, *etc.*) expected from a better modularisation of concerns. In particular, it may become impossible to reason on the base program without examining the woven program.

This thesis provides an answer to the above problem by defining categories of aspects whose semantic impact remains under control. For each category of aspects, we specify the class of properties of the base program that is preserved by weaving. The membership of an aspect to a category is guaranteed by construction through aspect languages dedicated to each category. The use of these languages ensures that weaving preserves all properties of the corresponding class. These properties are represented as subsets of LTL and CTL*. We formally prove that, for any program, the weaving of any aspect in a category preserves any property of the related class.

These languages and categories are defined in a formal framework independent of any base or aspect language. The expressiveness of that framework is shown by providing the semantics of complex primitives of aspect languages such as AspectJ and CaesarJ, and by proving the correctness of a standard aspect program transformation.

Keywords : programming language, aspect oriented programming, semantics, temporal properties, proofs

Table des matières

Table des figures

Introduction

Le développement des applications informatiques repose sur leur décomposition modulaire. Cette décomposition facilite les activités de programmation, de maintenance et de réutilisation des applications. Cependant, il existe des préoccupations (*par ex.,* sécurité, gestion des exceptions, persistence des données) appelées *transverses*, qui parfois se décomposent mal et se retrouvent entrelacées dans tout le code de l'application. La programmation par aspect [KLM+97] est un paradigme de programmation qui propose une solution à ce type de problème. Avec ce paradigme, chaque préoccupation transverse, peut être programmée séparément des fonctionnalités de base de l'application (*programme de base*), dans un *aspect*. Les aspects sont ensuite ajoutés automatiquement au programme de base pour donner le *programme tissé* qui correspond à ce qu'un programmeur aurait produit manuellement sans aspect. Cette séparation peut améliorer la lisibilité et programmation des applications, mais aussi leur maintenabilité et réutilisabilité. Néanmoins, les langages d'aspect généraux comme AspectJ [KHH+01a] et AspectC++ [SLU05], qui sont utilisés notamment en entreprise [CVK06], sont très expressifs et délicats à maîtriser. Ils peuvent complètement modifier la sémantique du programme de base.

Les langages d'aspect généraux permettent d'écrire n'importe quel type d'aspect. Ils peuvent modifier les variables du programme de base et remplacer ses appels de méthode par du code arbitraire. De plus, le tissage des aspects n'étant en général pas commutatif, ils peuvent interagir de façon non souhaitée par le programmeur. Ces comportements font qu'il est impossible de raisonner sur les aspects et le programme de base sans regarder le programme tissé. Aussi, comme ces langages d'aspect sont souvent accompagnés de sémantiques informelles, il est difficile de dire de façon précise, quel est l'impact des aspects sur le programme de base. Cette situation pourrait entraîner la perte des avantages (*c.-à-d.,* facilité de la programmation, maintenabilité et réutilisabilité) qu'offrent la programmation par aspect. Ainsi la recherche dans le domaine de la programmation par aspect porte entre autres sur la définition des sémantiques formelles [FL06], qui permettent de décrire précisément l'influence des aspects sur le programme de base. Elle porte également sur la définition des démarches qui permettent une réflexion effectivement modulaire sur le programme de base et les aspects *c.-à-d.,* raisonner sur le programme de base et les aspects sans regarder le code tissé. L'une de ces démarches est la définition de langages expressifs et adaptés à des domaines d'application dédiés (*par ex.,* synchronisation dans les applications distribuées, tolérance aux fautes, *etc.*). Malgré le fait qu'ils sont pour la plupart décrits de manière informelle, l'avantage de ces langages est qu'ils ont une influence spécifique et contrôlée sur le programme de base.

Cette thèse[1], propose un cadre qui permet de décrire formellement les langages d'aspect généraux, indépendamment du langage de programmation, du paradigme de programmation (*c.-à-d.,* impératif, objet, fonctionnel, ...) et sans nécessiter d'étape de traduction. Dans ce cadre, le tissage des différents types d'aspect est décrit séparément en introduisant uniquement les constructions nécessaires du langage de base. Nous utilisons ensuite ce cadre pour définir formellement des catégories d'aspects en fonction de leur impact sémantique sur le programme de base (modification du flot de contrôle ou des variables du programme de base). Pour chacune de ces catégories, nous identifions des propriétés du programme de base qui sont préservées (*c.-à-d.,* propriétés du programme de base qui restent satisfaites par le programme tissé) quelque soit l'aspect tissé appartenant à la catégorie correspondante. Avec cette démarche, il suffit de connaître la catégorie d'un aspect, pour déduire les propriétés du programme de base qu'elle permet de préserver. En complément, nous présentons des langages d'aspect qui assurent par construction que les aspects appartiennent à la catégorie correspondante.

Les contributions de cette thèse sont d'une part, de proposer un cadre formel original, qui permet de décrire précisément les exécutions des programmes tissés, indépendamment des types et paradigmes du langage de base, d'autre part d'utiliser ce cadre pour développer une approche, qui permet de raisonner sur le programme de base et les aspects sans regarder le programme tissé. Ces contributions ont été publiées dans deux conférences internationales (PEPM 08 [DDDF08a], SEFM 08 [DDDF08b]) et les livrables du projet AOSD-Europe [DDDF06, DDDF07, DDDF08c].

Ce mémoire s'organise en cinq chapitres.

Le Chapitre 1 présente les principes et fondations de la programmation par aspect. Nous exposons d'abord les caractéristiques principales de la programmation par aspect. Nous survolons un ensemble de travaux, qui ont défini des sémantiques pour décrire les programmes tissés, ou qui ont proposé des approches de raisonnement modulaire (de la vérification à la classification) pour la programmation par aspect. L'un des objectifs de ce chapitre, est de montrer que les cadres sémantiques utilisés pour raisonner sur la programmation par aspect sont complexes. L'autre objectif est de montrer que les langages d'aspect généraux ne permettent pas un raisonnement modulaire, mais aussi que les approches qui proposent un raisonnement modulaire sont soit coûteuses et particulières (la vérification), soit imprécises ou particulières (la classification).

Le Chapitre 2 présente notre cadre formel pour raisonner sur la programmation par aspect. Il s'agit d'une sémantique opérationnelle qui décrit toutes les étapes d'exécution d'un programme tissé. Ce cadre est indépendant de tout type et paradigme du langage de base. Dans ce cadre, l'exécution des instructions est séparée du tissage qui est effectué dynamiquement. Le tissage des différents types d'aspect est décrit de façon abstraite et séparément, en introduisant à chaque fois, les hypothèses sur le langage de base.

Le Chapitre 3 utilise le cadre présenté au Chapitre 2 pour proposer une approche qui permet de raisonner modulairement sur les aspects. Cette approche consiste à identifier des catégories d'aspects en fonction de leur impact sur la sémantique du programme de base. Pour chaque catégorie, nous identifions une classe de propriétés du programme de base

[1] financée dans le réseau d'excellence *AOSD-Europe* (www.aosd-europe.net)

qui est préservée quelque soit l'aspect tissé appartenant à la catégorie correspondante. Les classes de propriétés sont présentées comme des sous ensemble de LTL (resp. CTL*) dans le cas des programmes déterministes (resp. non déterministes). Il suffit donc de connaître uniquement la catégorie d'un aspect pour caractériser son impact sur le programme de base.

Le Chapitre 4 complète l'approche présentée au Chapitre 3 en proposant pour des catégories identifiées précédemment, un langage d'aspect général ou spécifique, qui assure par construction que chaque aspect appartient à la catégorie correspondante. Les sémantiques des langages proposés sont une application du cadre présenté au Chapitre 2.

Le Chapitre 5 complète les applications du cadre présenté au Chapitre 2, en décrivant d'une part, certaines constructions très utilisées par différents langages d'aspect. Dans le but d'être indépendant d'un langage particulier, ces constructions sont décrites séparément les unes des autres. Nous montrons également que ce cadre peut être utilisé pour faire des preuves de correction. A cet effet, comme le code des aspects est ajouté automatiquement en des points particuliers du programme qui sont sélectionnés en utilisant des motifs, nous prouvons que tout motif qui teste un prédicat sur l'état du programme peut être transformé en une instruction conditionnelle à l'intérieur du code à ajouter au programme.

La Conclusion fait une synthèse et présente des perspectives de recherche.

Enfin, nous présentons en **Appendices**, des preuves de correction de l'approche présentée aux Chapitres 3 et 4, et la sémantique d'un sous ensemble d'AspectJ basé sur un sous ensemble de Java. Cette sémantique a pour but de montrer que notre cadre formel peut décrire l'interaction entre les constructions que nous avons décrites séparément au Chapitre 5.

Chapitre 1

Programmation par aspect : principes et fondations

1.1 Le paradigme des aspects

1.1.1 Présentation générale

Les applications informatiques actuelles doivent répondre à diverses préoccupations. Les paradigmes de développement utilisés pour leurs mises en œuvre reposent sur le fait que ces préoccupations puissent se décomposer en une collection de modules (procédures, fonctions, classes, *etc.*). Cependant, certaines préoccupations notamment celles qui assurent des contraintes non fonctionnelles ne respectent pas en général cette décomposition et se retrouvent dispersées dans tous les modules de celles-ci. Elles sont alors dites transverses. Un des exemples de préoccupation transverse, utilisé pendant l'activité de programmation est le débogage : du code est ajouté généralement de façon manuelle pour tracer les valeurs de variables utilisées dans plusieurs modules de l'application. Nous pouvons citer aussi comme autres exemples la gestion de la sécurité, la mise en œuvre des transactions et la gestion de la synchronisation des processus. Ces préoccupations transverses peuvent avoir des conséquences négatives comme :

- un traçage difficile : les différentes préoccupations d'une application deviennent difficilement identifiables à l'implémentation, rendant complexe l'activité de validation des applications ;
- une diminution de la productivité : la prise en compte de plusieurs préoccupations au sein d'un même module empêche le programmeur de se concentrer sur son but premier (la fonctionnalité du module) ;
- une diminution de la qualité du code : le programmeur devant se concentrer sur des contraintes dispersées dans un ou plusieurs modules, est tenté d'utiliser des effets de bords qui entraînent des erreurs ;
- une diminution de la réutilisation du code : un module pouvant implémenter plusieurs

préoccupations, d'autres applications nécessitant une fonctionnalité similaire ne peuvent plus réutiliser le module tel quel, entraînant de nouveau une diminution de la productivité ;
– une maintenance et évolution difficile : faire évoluer les applications demande la modification de plusieurs modules. Ce qui pourrait introduire des incohérences.

Pour remédier à ces inconvénients, Kiczales et coll. [KLM+97] ont proposé la programmation par aspect. Elle consiste à décrire chaque préoccupation transverse dans un module séparé appelé *aspect*. Cette séparation rend les aspects et les autres modules de l'application (appelés par la suite application ou programme de base) plus lisibles, faciles à développer, réutilisables et maintenables.

La programmation par aspect repose sur un langage qui permet de définir dans un aspect, *où* et *comment* les préoccupations transverses modifient l'application de base. Ces aspects sont ensuite transmis à un *tisseur* qui les ajoute automatiquement à l'application de base. Le tisseur peut être vu comme un moniteur qui, observe l'exécution du programme de base et en fonction du lieu (*où*) spécifié dans l'aspect ajoute automatiquement le code (*comment*) correspondant à chaque préoccupation transverse. En général, il est implémenté comme un préprocesseur qui transforme statiquement le code du programme de base en fonction des aspects. La figure 1.1 présente dans sa partie gauche l'exemple d'une application (de deux modules mais on pourrait imaginer un nombre arbitraire de modules) entrelacée par une partie du code liée à la synchronisation des processus, à la sécurité et à la persistance des données. Avec la programmation par aspect, chaque préoccupation transverse est décrite séparément dans un aspect, et, par l'intermédiaire d'un tisseur, ajoutée au reste de l'application pour donner un code qui correspondrait à l'application initiale.

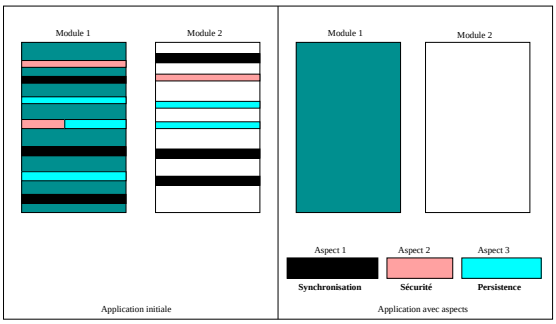


FIG. 1.1 – Préoccupations transverses et programmation par aspect

La programmation par aspect propose une solution qui permet une meilleure séparation des préoccupations d'une application pendant son développement. Elle permet de définir de manière séparée dans un aspect, chaque préoccupation transverse de l'application. Cette séparation offre une meilleure décomposition de l'application. Ces aspects sont ensuite ajoutés automatiquement à l'application de base par transformations de programme. Cependant, en général, les transformations qui peuvent être effectuées peuvent modifier complètement la sémantique du programme de base. Cette caractéristique rend difficile tout raisonnement sur l'application avec aspects sans prendre en compte le programme tissé. Un des défis majeurs de la programmation

par aspect est donc de permettre un raisonnement modulaire. C'est à dire qu'il faut pouvoir écrire et raisonner sur l'application de base et les aspects sans étudier le code tissé.

1.1.2 Domaines d'application des aspects

La séparation des préoccupations transverses et l'ajout du code à des endroits précis permet l'application de la programmation par aspect dans de nombreux domaines comme par exemple le débogage, la tolérance aux fautes et les intergiciels ("middleware") adaptables. Nous présentons ces trois domaines d'application ci-dessous avant de lister quelques exemples d'utilisation supplémentaires.

Le débogage

L'une des pratiques usuelles des programmeurs est l'ajout de code dans toute l'application afin de tester des assertions ou tracer des valeurs. Ce code représente une préoccupation transverse qui peut être regroupée dans un aspect comme l'ont décrit Kiczales et coll. dans [KHH+01a]. Une fois les erreurs détectées, ces aspects de débogage peuvent être désactivés et éviter ainsi de commenter ou de supprimer les lignes de code ajoutées. Cela permet un gain de temps pendant le développement des applications et surtout celui des grandes applications tout en supprimant les erreurs qui peuvent être introduites par l'oubli de commenter le code du débogage. Ces aspects permettent aussi une meilleure maintenance de l'application car pendant l'évolution de celle-ci, il suffit de les retisser pour chercher des nouvelles erreurs.

La tolérance aux fautes

Un système est dit tolérant aux fautes lorsqu'il préserve le service qu'il rend malgré l'occurrence des fautes. Une faute est définie comme étant une cause qui provoque une erreur. Une erreur a un état corrompu dans lequel se trouve le système ou l'application ; un tel état pouvant engendrer alors une défaillance. Une défaillance, quant à elle, se manifeste sous la forme d'une déviation du comportement de l'application par rapport au comportement correct défini par la spécification de l'application. La mise en œuvre de tels systèmes est donc basée sur la détection et le traitement de l'occurrence d'une faute. On distingue des fautes matérielles (panne d'un serveur) et des fautes logicielles (dépassement du délai d'exécution d'une tache, erreur du programmeur, *etc.*). Les fautes matérielles sont généralement traitées par la redondance du matériel tandis que les fautes logicielles sont traitées par la détection des comportements anormaux qui ne sont pas gérés (*par ex.,* les exceptions qui ne sont pas capturées). Ces traitements peuvent s'entrelacer dans toute l'application en fonction des fautes potentielles qui peuvent se retrouver dans plusieurs modules. Ils peuvent donc être vus comme une préoccupation transverse

qui peut être définie de manière séparée dans un aspect. Il en découle une meilleure compréhension, maintenabilité et réutilisabilité de l'application de base comme l'a montré Afonso et coll. [ASB+08]. L'autre apport fondamental de la programmation par aspect est qu'elle permet d'ajouter de la tolérance aux fautes à un système qui n'en possède pas en minimisant les erreurs introduites par la modification du système. C'est le cas du système JSR (*Java Server Recovery*), proposé par Bouchenak et coll. [BdPK05], qui permet d'ajouter de la tolérance aux fautes logicielles à toute application J2EE. Ce travail considère qu'une faute est une exception levée qui n'est pas capturée. Il utilise la programmation par aspect pour ajouter à l'application la capture et le traitement des fautes.

Intergiciel adaptable

Le développement d'internet et la présence de plus en plus importante de l'informatique dans notre vie (*par ex.,* les téléphones portables) a permis l'essor des intergiciels. Un intergiciel est une couche d'application qui se trouve entre les applications de haut niveau et les systèmes d'exploitation. Ils ont pour principales fonctions de masquer l'hétérogénéité des systèmes et matériels sous-jacents, de rendre la répartition aussi invisible que possible et de fournir des services repartis d'usage courant et une interface de programmation de haut niveau. Ces fonctions ont pour but de faciliter la programmation répartie en permettant un développement, une évolution et une réutilisation des applications, une portabilité des applications entre plates-formes et une interopérabilité d'applications hétérogènes. Les besoins de telles applications évoluant en permanence (qualité de service, capacité de croissance, *etc.*) et celles-ci s'exécutant dans un environnement changeant (la mobilité), les intergiciels et les applications doivent donc pouvoir s'adapter et de façon dynamique. La programmation par aspect fournissant la possibilité d'ajouter dynamiquement du code à une application tout en respectant les caractéristiques des intergiciels (évolution facile, réutilisation des applications, portabilité, *etc.*) est donc une technique adéquate pour ce domaine. C'est le cas par exemple des intergiciels JBOSS AOP [1], JAC [PSD+04] et DyMAC [LJ06], basés sur la programmation par aspect, qui offrent aux développeurs d'applications de haut niveau des aspects afin qu'ils décrivent de manière séparée et transparente les comportements non fonctionnels (persistance, gestion du cache, transactions distribuées, gestion des utilisateurs, *etc.*) des applications. Par exemple, JBOSS Cache AOP qui est implémenté en utilisant JBOSS AOP, permet la gestion du cache dans les applications distribuées J2EE de manière transparente. En effet, les applications J2EE masque leur distribution en utilisant un cache qui permet de maintenir la cohérence des données sur les machines distantes. A chaque modification d'une donnée qui se trouve dans le cache d'une machine, un message de diffusion est envoyé aux machines distantes afin qu'elles mettent à jour leur cache. JBOSS Cache AOP utilise la programmation par aspect pour mettre à jour les caches des machines distantes à chaque modification d'un cache.

[1] www.jboss.com

Autres domaines d'application

On trouve plusieurs autres utilisations possibles des aspects. Citons [CF00] qui utilise des aspects pour imposer des politiques de sécurité (gestion de permissions). Lors du tissage, des analyses peuvent être effectuées pour minimiser le nombre d'instructions ajoutées et éventuellement n'insérer aucune instruction si la politique de sécurité est vérifiée par l'application. Ces optimisations améliorent les performances du code tissé. La séparation des politiques de sécurité dans les aspects permet plus facilement leurs modifications et leurs réutilisations dans d'autres applications. Dans [BCD+06], un système de gestion du cache des pages web dynamiques basé sur les aspects est proposé. Ce système, grâce aux aspects, peut être ajouté de façon transparente à toute application web J2EE s'appuyant sur une base de donnée. Il assure que le contenu dynamique des pages (les données) est cohérent en mettant à jour le cache après chaque requête. Dans [LL00], la programmation par aspect est utilisée pour simplifier la gestion des exceptions dans une application. Les auteurs montrent que sans les aspects, la gestion des exceptions produit un code qui n'est pas réutilisable et qui se répète dans plusieurs modules rendant l'application parfois illisible et difficilement maintenable. Cette situation se simplifie lorsqu'on utilise les aspects pour coder de manière séparée cette gestion et apporter une meilleure compréhension, maintenabilité et réutilisabilité de l'application (application de base et aspects). D'autres exemples d'utilisation seront présentés en guise d'illustration tout au fil de ce document de thèse.

La programmation par aspect est une technique récente qui a apporté des solutions intéressantes à certains problèmes industriels notamment dans le domaine des intergiciels adaptables. La recherche sur ce paradigme continue d'être active et porte entre autres sur la mise en œuvre de langages plus expressifs ou plus adaptés aux préoccupations transverses, et l'amélioration de la formalisation des questions liées à ce paradigme comme la sémantique des langages d'aspect ou la détection des interférences entre aspects. Dans la suite de ce chapitre, nous nous attardons sur des langages d'aspect représentatifs et sur des travaux de formalisation dont le but est de comprendre et de cerner l'impact des aspects.

1.2 Les langages d'aspect

La programmation par aspect repose sur des langages fournissant une syntaxe qui permet aux programmeurs de définir modulairement les préoccupations transverses d'une application. Les travaux actuels sur les langages d'aspect distinguent les *langages d'aspect généraux* et les *langages d'aspect dédiés*. Les langages dédiés permettent de définir un type de préoccupation transverse lié à un domaine d'application. Cette particularité limite l'impact sémantique des aspects sur le programme de base. Cependant, pour le moment, peu de travaux [SLS03, TN05, KL07] traitent de la composition de ces langages. Les langages d'aspect généraux fournissent des syntaxes expressives qui sont capable d'exprimer différentes préoccupations transverses d'une application. Cependant cette grande expressivité permet en général à l'aspect de modifier sans contraintes la sémantique d'une application et les interactions entre plusieurs aspects ne

sont pas toujours faciles à gérer. Elle offre également au programmeur la possibilité de rompre le caractère modulaire de l'aspect.

Dans cette section, nous présentons le langage d'aspect général le plus connu (AspectJ) et un langage d'aspect dédié (COOL) pour les applications distribuées.

1.2.1 Un langage d'aspect général : AspectJ

AspectJ [KHH+01a, KHH+01b] est un langage d'aspect général pour Java qui a été proposé par Kiczales et coll. pour montrer l'utilité de la programmation par aspect. D'autres langages d'aspect généraux que nous ne présentons pas ici existent. Citons, par exemple, CaesarJ [AGMO06] pour Java, Arachne [DFL+05] pour C et AspectML [DWWW08] pour ML.

AspectJ, comme les autres langages d'aspect généraux, indique où et comment les aspects modifient le programme de base. Par exemple, un aspect qui arrête un programme (le comment) après l'appel d'une méthode particulière (le où). Cette séparation entre le où et le comment est définie par l'intermédiaire d'entités syntaxique du langage appelées respectivement point de coupure (*point cut*) et action (*advice*). Un point de coupure est un motif qui permet de sélectionner un ou plusieurs points du programme *où* le code (l'action) est tissé. Ces points du programme sont appelés point de jonction (*join point*). Une action représente le code à tisser et spécifie *comment* l'aspect modifie le programme. Pour illustrer ces notions, nous utiliserons le programme de base Java ci-dessous

```
class Base {
  static int callsB = 0;

  void bar(){
    System.out.println("join point bar" + " " + this.callsB);
  }

  void foo(){
    System.out.println("join point foo");
    callsB = callsB + 1;
    bar();
  }

  public static void main(String args[]){
    new Base().foo();
  }
}
```

Ce programme appelle la méthode `foo` de la classe `Base`. Cette méthode incrémente l'unique

champ de `Base` avant d'appeler la méthode `bar` qui affiche la valeur de ce champ sur la sortie standard.

Dans AspectJ, on distingue des points de jonction statiques et dynamiques. Les points de jonction statiques sont par exemple les appels et les exécutions de méthode, l'accès aux attributs d'une classe (en lecture ou en écriture) et la capture d'une exception. Tandis que les points de jonction dynamiques sont représentés entre autres par les valeurs des arguments d'une méthode, les valeurs des attributs, l'objet appelant et la pile des appel de méthode. Ces points de jonction sont filtrés par des points de coupure qui sont définis par des motifs d'expression logiques. Chaque motif permet de sélectionner un ensemble de point de jonction. Par exemple :

- `call(void Base.foo())`
  filtre tout appel de la méthode `foo()` de la classe `Base` dont le type de retour est `void`. En plus de `call`, AspectJ fournit un motif `execution` qui permet de filtrer l'exécution du corps des méthodes ;
- `call(void Base.*)`
  filtre tous les appels de méthode de la classe `Base` dont le type de retour est `void` ; l'utilisation des jokers (`*`) offre la possibilité au programmeur de développer un aspect indépendamment du programme de base ;
- `get(int Base.callsB)`
  filtre tous les accès en lecture de l'attribut `callsB` de la classe Base et de type `int` ;
- `set(int Base.callsB)`
  filtre tous les accès en écriture de l'attribut `callsB` de la classe Base et de type int ;
- `handler(IOException)`
  filtre toute capture d'exception `IOException` ;
- `call(void Base.foo()) && if(Base.callsB = 1)`
  définit un point de coupure dynamique qui filtre tout appel de la méthode `foo` de la classe `Base` à condition que son attribut `callsB` soit égal à `1`. Tout comme "et" (`&&`), les autres opérateurs logiques "ou" (`||`) et "non" (`!`) peuvent être utilisés entre motifs ;
- `pointcut Foo(Base r) : call(void Base.foo()) && target(r)`
  déclare un point de coupure dynamique nommé `Foo` qui filtre tout appel de la méthode `foo()` de la classe `Base` et dont le receveur est `r`. Le receveur est filtré en utilisant la primitive `target(r)` où `r` est une variable de motif qui peut être utilisé dans le code de l'action. Lorsqu'un appel de `foo()` est filtré par le point de coupure `Foo`, la valeur de l'objet appelant est associée à `r`. Outre `target`, les motifs `args` et `this` (utilisé avec `execution`) permettent respectivement de récupérer la valeur des arguments d'une méthode filtrée et l'objet appelé ;
- `cflow(call(void Base.foo()))`
  est un autre point de coupure dynamique qui filtre tout point de jonction qui se trouve dans le flot de contrôle d'un appel à la méthode `foo` de la classe `Base`. Par exemple, la conjonction de ce point de coupure avec un motif sur un appel de méthode,

  ```
  call(void Base.bar())
  && cflow(call(void Base.foo()))
  ```

  dénote tout appel de méthode `bar` de la classe `Base` effectué pendant l'exécution de toute méthode `foo` de la même classe. Notons qu'avec
  `cflow(call(void Base.foo()))`,

la méthode `foo` se trouve dans son flot de contrôle. Si on veut l'exclure de celui-ci, on utilise le motif `cflowbelow`.

Lorsqu'un point de jonction est filtré par un point de coupure, l'action qui correspond à ce point de coupure est tissée. AspectJ possède trois types d'actions appelés `before`, `after` et `around` qui spécifient le lieu où le code de l'action est tissé par rapport au point de jonction. Le code d'une action de type `before` (resp. `after`) est exécuté avant (resp. après) le point de jonction filtré. Par exemple, l'action suivante

```
before(): call(void Base.foo()){
  callsb = callsb + 1;
}
```

compte le nombre d'appels `callsb` de la méthode `foo` de la classe `Base`. Pour cela, elle incrémente `callsb` avant chaque appel de cette méthode. Une instruction particulière `proceed` peut être utilisée dans le cas des actions de type `around`. Lorsqu'elle comporte une instruction `proceed`, une action `around` exécute la partie de son code avant `proceed`, ensuite par l'intermédiaire de `proceed` elle exécute le point de jonction filtré et enfin termine avec l'exécution du code qui se trouve après `proceed`. En général, une action `around` peut comporter plusieurs `proceed` et dans ce cas, le point de jonction est exécuté autant de fois que de `proceed`. Si `proceed` n'est pas présente, le point de jonction n'est pas exécuté et le code de l'action le remplace complètement. Les variables des points de jonction peuvent être modifiées par l'intermédiaire des variables de motifs et de `proceed` qui prend en paramètre le même nombre et les mêmes types d'argument que le point de coupure. Par exemple, l'action :

```
void around(Base r): Foo(r) {
 if(callsa < 500) {
    callsa = callsa + 1;
    r = new Base();
    proceed(r);
 } else System.exit(1);
}
```

qui utilise le point de coupure `Foo` défini précédemment, compte le nombre d'appel de la méthode `foo` de la classe `Base` si celui-ci est inférieur à `500`. Avant d'exécuter cette méthode par l'intermédiaire de `proceed`, elle modifie l'objet appelant qu'elle a récupéré dans `r`, en affectant à celui-ci un nouvel objet de type `Base`. Si le nombre d'appel de la méthode `foo` est supérieur à `500`, l'action remplace l'appel courant par `System.exit(1)` qui arrête l'exécution du programme. A côté de ces actions classiques, AspectJ possède des types d'action particuliers notamment `around throws` et `after throwing` qui portent sur les exceptions. Une action `around throws` a la même sémantique qu'une action `around` mais ici, le point de jonction filtré pourrait lever une exception. Une action `after throwing` est exécutée après que le point de jonction filtré lève une exception.

Dans AspectJ, un aspect est une abstraction au même titre qu'une classe. Ainsi, en plus des points de coupure et des actions, il peut contenir des attributs, des méthodes et étendre une classe ou un autre aspect. Chaque aspect a donc un état et peut être instancié comme une classe. Mais cette création d'instance ne se fait pas comme dans le cas d'une classe en utilisant un constructeur. Elle utilise des primitives d'AspectJ telles que `percflow`, `pertarget`. Par exemple, associé à un aspect `A`, `percflow(call(void Base.foo()))`, crée une instance de `A` à chaque fois que l'on entre dans le flot de contrôle de la méthode `foo` de la classe `Base` tandis que `pertarget(call(void Base.foo()))` crée une instance de `A` pour chaque objet appelant la méthode `foo` de la classe `Base` (tout comme `pertarget` est lié au motif `call`, une primitive `perthis` utilisé avec le motif `execution` permet de créer une instance d'aspect pour chaque objet appelé *c.-à-d.,* `this`). Lorsqu'aucune de ces primitives n'est spécifiée, une instance unique est créée une fois pour toute pour chaque aspect. Le code ci-dessous

```
  aspect ProfileBar pertarget (call (void Base.bar())) {
    int callsA = 0;
    pointcut barDuringFoo(Base r):
        call(void Base.bar())
        && target(r)
        && cflow(call(void Base.foo()));
    void around(Base r): barDuringFoo(r) {
        callsA = callsA + 1;
        proceed(r);
    }
}
```

définit donc un aspect `ProfileBar` qui crée pour chaque objet appelant la méthode `bar()` une instance de `ProfileBar` qui va compter le nombre d'appel de la méthode `bar()` pendant l'exécution de la méthode `foo`. Ce nombre d'appel est sauvegardé dans l'attribut `callsA` qui représente l'état de chaque instance d'aspect.

Lorsque plusieurs aspects peuvent s'appliquer sur un même point de jonction, AspectJ fournit la primitive

```
declare precedence: Aspect_1, ..., Aspect_n
```

pour définir l'ordre d'application des aspects. Lorsqu'elle est appliquée à des aspects de type `after`, ceux-ci sont exécutés dans l'ordre inverse de celui indiqué par la primitive et pour tous les autres cas, l'ordre est celui indiqué par la primitive. Si aucun ordre n'est indiqué, l'ordre d'exécution des aspects n'est pas défini. En général, le comportement d'un programme tissé définit avec AspectJ est donc non-déterministe. Par exemple, les aspects `before` et `around` ci-dessus, qui comptent respectivement avec `callsb` et `callsa` le nombre d'appel de la méthode `foo`, peuvent en fonction de l'ordre d'application des actions, dans le cas où

`callsa > 500`, arrêter le programme sans incrémenter `callsb` ou incrémenter `callsb` avant d'arrêter le programme.

AspectJ est un langage d'aspect général qui repose sur des directives d'ordre syntaxique plus que sémantique. Cela oblige à réfléchir sur le programme tissé. Or l'intérêt de séparer les préoccupations transverses du programme de base réside dans le fait de pouvoir raisonner indépendamment sur le programme de base et l'aspect. Comme les aspects permettent de tisser du code Java arbitraire, ils peuvent modifier les attributs des objets, les arguments des méthodes ou supprimer des appels de méthode (`around` sans `proceed`) et ainsi complètement changer la sémantique du programme de base. De plus, ces modifications arbitraires font que le tissage de deux aspects n'est pas commutatif et les aspects peuvent interagir entre eux de manière non souhaitée par le programmeur.

1.2.2 Un langage d'aspect dédié : COOL

Pour remédier à ces difficultés, les langages dédiés proposent des langages d'aspect qui décrivent une préoccupation transverse d'un domaine d'application spécifique tout en essayant de maîtriser l'impact sémantique. Par exemple, c'est le cas de COOL que nous présentons ici mais aussi de RIDL [VL97] et KALA [FETD08] qui permettent respectivement de décrire la transmission des données et des transactions dans les applications distribuées.

Le langage COOL permet de décrire la coordination des processus dans une application distribuée. Il a été proposé par Cristina Videira Lopes dans sa thèse [VL97]. Dans ce cadre, une application distribuée est un ensemble d'objets distribués et concurrents qui communiquent par des appels à leurs méthodes partagées. Cool va donc servir à séparer dans un aspect, la préoccupation de coordination entre processus sur des méthodes partagées. L'autre objectif de COOL est de minimiser son intrusion dans les objets sur lesquels il synchronise les processus et ainsi éviter de changer la sémantique de l'application de base. Ce langage permet de spécifier les exclusions sur les processus entre les différents appels de méthode en utilisant les constructions `selfex{}` et `mutex{}`. Les méthodes appartenant à un ensemble déclaré `selfex` sont exécutées par un seul processus à la fois tandis que celles dans un ensemble `mutex` ne sont pas exécutées en concurrences *c.-à-d.,* l'exécution d'une méthode de l'ensemble par un processus exclut l'exécution des autres par d'autres processus. Le blocage et le réveil des processus peuvent aussi être spécifiés en définissant des conditions qui doivent être satisfaites avant l'accès aux méthodes par des processus. Ces conditions sont introduites par la primitive `require`. L'état de celles-ci varient en fonction de l'état des variables locales à l'aspect qui sont modifiées à l'entrée (`on_entry`) et à la sortie (`on_exit`) de l'exécution des méthodes. En supposant que nous avons une classe `BoundedStack` qui définit une pile bornée avec deux méthodes `pop` et `push`, l'aspect :

```
coordinator boundedStackCoord: BoundedStack{
   selfex{pop, push};
   mutex{pop, push};
```

```
    condition boolean full = false;
    condition boolean empty = true;

    push:requires!full;
     on_exit{
      if(sp == MAX)
         full = true;
         empty = false;

     }

    pop:requires!empty;
     on_exit{
      if(sp ==0)
         empty = true;
         full = false;
     }
}
```

décrit la politique de coordination des processus sur les méthodes `pop` et `push`. Une seule méthode est exécutée à un instant donné car `pop` et `push` sont à la fois dans `selfex` et `mutex`. La méthode `pop` requiert que la pile ne soit pas vide tandis que `push` que la pile ne soit pas pleine. De tels aspects ont un impact limité sur la sémantique de l'application de base car ils ont été conçu par construction pour observer son exécution et assurer uniquement la coordination des processus. En effet, tout aspect (`coordinator`) de COOL respecte par construction et hypothèses les contraintes suivantes :

- toutes les opérations réalisées par les aspects sont faites de façon atomique ce qui permet d'éviter des conflits au niveau des conditions et de garantir une exécution sûre des processus ;
- les aspects ne peuvent pas modifier les attributs ou appeler les méthodes des objets. Par contre, ils peuvent accéder aux attributs de ceux-ci en lecture (c'est le cas de `sp` qui est l'attribut de la classe `BoundedStack` et qui contient le nombre d'éléments de la pile) ;
- les seules commandes possibles à l'intérieur des aspects (à l'exception des primitives dédiées) sont les affectations sur les variables de l'aspect ou les instructions conditionnelles (`if`) sur ces affectations. Cela assure la terminaison des aspects s'il n'existe pas d'interblocage entre les processus.

Dans cette partie, nous avons présenté les langages d'aspect en séparant les langages généraux des langages dédiés. Les langages généraux, avec leur expressivité peuvent décrire différents types de préoccupations transverses. Cette caractéristique vient du fait que l'aspect peut ajouter du code arbitraire au programme de base. Le risque est de rompre la modularité entre aspect et programme de base et changer complètement la sémantique du programme de base. Les langages dédiés quant à eux permettent un raisonnement plus modulaire en séparant dans l'aspect un unique type de préoccupation transverse. Cette séparation entre les rôles de l'application de base et des aspects est orthogonale et permet de raisonner indépendamment sur la

sémantique et l'implémentation de l'application. Il permet aussi de mieux maîtriser l'impact de l'aspect sur l'application de base. Cependant la composition de ces langages afin d'exprimer plusieurs préoccupations différentes (*par ex.,* coordination et transaction) est une activité de recherche récente.

1.3 Formalisation de la programmation par aspect

La majorité des travaux du domaine de la programmation par aspect est informelle et cela malgré la complexité du tissage et des langages d'aspect. Néanmoins certains travaux s'attachent à formaliser la programmation par aspect. Leur but est de décrire la sémantique des aspects (du tissage) afin de comprendre de façon sûre quel est l'impact de l'aspect sur le programme de base. Dans cette section, nous décrivons quelques uns de ces travaux qui portent sur la sémantique des langages d'aspect, la vérification des programmes avec aspects, la classification des aspects en fonction de leurs comportements et la conception de langages d'aspect garantissant des comportements.

1.3.1 Sémantiques formelles des langages d'aspect

Les langages d'aspect fournissent la syntaxe nécessaire pour décrire les préoccupations transverses d'une application. Le tissage intègre ensuite automatiquement ces préoccupations à l'application. Une sémantique formelle est indispensable pour prouver la correction du tissage, qu'un aspect a le comportement attendu, qu'il y a pas d'interférences entre aspects mais aussi pour clarifier les ambiguïtés de la sémantique informelle du langage. Par exemple, en exécutant le programme de base de la classe `Base` présentée à la Section 1.2.1, page 10, on obtient le résultat ci-dessous :

```
join point foo
join point bar 1
```

Après le tissage des aspects `A1` et `A2` suivants

```
aspect A1 {
 void around(Base r): target(r) && call(void Base.foo())
                      && if(r.callsB == 0) {
   System.out.println("Before around1");
   r.callsB = r.callsB + 1;
   proceed(r);
   System.out.println("Afeter around1");
 }
```

```
}


aspect A2 {
  void around(Base r): target(r) && call(void Base.foo())
                        && if(r.callsB == 1) {
    System.out.println("Before around2");
    proceed(r);
    System.out.println("After around2");
  }
}
```

le résultat ci-dessus devient :

```
Before around1
Before around2
join point foo
join point bar 2
After around2
After around1
```

car l'aspect `A1` s'exécute en premier et déclenche l'exécution de l'aspect `A2`. En effet, seul l'aspect `A1` filtre l'appel de la méthode `foo` par le programme de base (`r.callsB = 0`). Ensuite à l'exécution de `proceed`, l'aspect `A1` déclenche l'aspect `A2` (`proceed` $\Rightarrow$ `foo` et `r.callsB = 1`) qui s'exécute et repasse la main à l'aspect `A1`. On pourrait donc penser que l'action de l'aspect `A2` par exemple, est équivalent à l'action ci-dessous où le point de coupure `if(r.callsB == 1)` est transformé en instruction conditionnelle à l'intérieur de l'action.

```
void around(Base r): target(r) && call(void Base.foo()) {
 if(r.callsB == 1){
   System.out.println("Before around2");
   proceed(r);
   System.out.println("After around2");
 }
}
```

Mais ce n'est pas le cas. En effet, dans le cas où `r.callsB` $\neq$ `1` et `target(r) && call(void Base.foo())` filtre le point de jonction courant, l'action de `A2` exécute ce point de jonction tandis que son équivalent ne fait rien. Ce comportement se complique lorsque plusieurs aspects s'appliquent à un même point de jonction. Pourtant, la

sélection d'un point du programme par le point de coupure `if(...)` et son interaction avec les actions ne sont pas spécifiées dans la sémantique informelle d'AspectJ.

Nous distinguons deux catégories de sémantiques. Dans la première, les sémantiques décrivent des langages d'aspect particuliers [WKD04, CL06]. Elles sont souvent complexes et incomplètes. La seconde catégorie décrit des sémantiques plus génériques pouvant s'appliquer à plusieurs langages [BJJR04, CLW03]. Elles nécessitent une étape de traduction. Nous nous concentrons sur les travaux qui nous ont semblé les plus représentatifs de ces catégories.

Sémantique dénotationnelle d'un langage d'aspect général

Wand et coll. [WKD04] proposent une sémantique dénotationnelle d'un langage d'aspect qui possède les mêmes concepts qu'AspectJ (point de jonction, point de coupure et action). Le langage du programme de base est Scheme, un langage du premier ordre avec des procédures et des appels par valeurs. Les expressions des actions sont supposées être celles du langage de base avec en plus l'expression `proceed` qui peut être utilisée pour les actions de type `around`. Les points de jonction filtrés ici, sont les appels de procédure ou les ensembles d'instructions dans un appel de procédure (`cflow`). Ils sont définis par la grammaire ci-dessous

$$\begin{array}{lcl} jp & ::= & \langle\rangle \mid \langle k, pname, wname, v^*, jp\rangle \\ k & ::= & \texttt{pcall} \mid \texttt{aexecution} \mid \ldots \end{array}$$

L'ensemble des points de jonction peut être vu comme une abstraction de la pile des appels de méthode. Par exemple le point de jonction $\langle \texttt{pcall}, f, g, v^*, jp\rangle$ représente un appel de la procédure f avec les arguments v^* à partir de la procédure g, avec jp qui est le point de jonction précédent. Un point de jonction de type `aexecution` représente l'exécution d'une action (qui est transformée en procédure ; permet de filtrer les instructions des actions). Dans ce cas, les champs $pname$, $wname$ et jp sont vides. Ces points de jonction sont filtrés en utilisant des points de coupure pcd décrit comme suit :

$$\begin{array}{lcl} pcd & ::= & (\texttt{pcalls}\ pname) \mid \texttt{cflow}\ pcd \mid (\texttt{args}\ id_1 \ldots\ id_n) \\ & ::= & (\texttt{and}\ pcd\ pcd) \mid (\texttt{or}\ pcd\ pcd) \mid (\texttt{not}\ pcd) \end{array}$$

Un point de coupure pcd est, soit un motif `pcalls` $pname$ qui permet de filtrer un appel de procédure dont le nom est $pname$, soit un motif `cflow` pcd qui permet de filtrer tout point de jonction qui se trouve dans le flot de contrôle de pcd, soit enfin un motif `args` $id_1, \ldots, id_n$ qui permet de récupérer les valeurs des arguments d'une procédure. Il peut aussi être une conjonction, disjonction et négation de ces motifs. Le processus de filtrage des points de jonction est calculé par la fonction `match-pcd` qui prend un motif de point de coupure et un point de jonction en paramètres et retourne, soit $Fail$ si le point de jonction n'est pas filtré, soit des associations des variables à des valeurs qui vont permettre de substituer les variables de motifs (*par ex.,* $id_1, \ldots, id_n$) utilisées dans les actions par leurs valeurs.

$$\texttt{match-pcd}(\texttt{pcalls}\ pname)\langle k, pname', wname, v^*, jp\rangle = \begin{cases} [\,] & \text{si } k = \texttt{pcall} \wedge pname = pname' \\ Fail & \text{sinon} \end{cases}$$

$$\texttt{match-pcd}(\texttt{args}\ id_1 \ldots id_n)\langle k, pname, wname, (v_1, \ldots, v_m), jp\rangle = \begin{cases} [id_1 = v_1, \ldots, id_n = v_n] & \text{si } k = \texttt{pcall} \wedge n = m \\ Fail & \text{sinon} \end{cases}$$

...

Par exemple, comme esquisser par la définition ci-dessus, un motif `pcalls` $pname$ filtre tout appel de procédure de nom $pname$ en retournant une substitution vide tandis que `args` $id_1 \ldots id_n$ associe à chaque variable id_i la valeur des paramètres d'une procédure lors d'un appel. Cette fonction est utilisée par la fonction sémantique

$$\mathcal{A} : JP \rightarrow Proc \rightarrow Proc$$

qui prend en paramètres un point de jonction et une procédure et génère une nouvelle procédure correspondant à l'action. Par exemple, les actions de type `before` sont définies comme suit :

$$\begin{array}{l} \mathcal{A}[\![((\texttt{before}\ pcd)\ e)]\!]\ \phi\ \gamma = \lambda jp\ \pi\ v^*. \\ \quad \mathcal{PCD}[\![pcd]\!]jp \\ \quad\ (\lambda\rho.\textit{enter-join-point}\ \gamma \\ \quad\quad \textit{new-aexecution-jp} \\ \quad\quad\ (\lambda v^*.\textbf{let} \\ \quad\quad\quad v_1 \Leftarrow \mathcal{E}[\![e]\!](\rho[\%\texttt{proceed} = None])\ \phi\ \gamma; \\ \quad\quad\quad v_2 \Leftarrow (\pi\ v^*) \\ \quad\quad\quad \textbf{in}\ v_2) \\ \quad\quad \langle\rangle) \\ \quad\ (\pi\ v^*) \end{array}$$

Ainsi, pour le point de jonction courant (jp), la procédure qui lui correspond (π) et ses paramètres (v^*), $\mathcal{A}$ définit une procédure qui, dans un premier temps applique au point de jonction courant, le point de coupure pcd qui correspond à l'action. Ensuite, appelle la fonction *enter-join-point* γ qui, à partir de l'ensemble des autres actions existantes et sauvegardées dans γ (dont les points de coupure vont éventuellement filtrer les instructions de l'action courante), du corps e de l'action courante et de la fonction *new-aexecution*, va construire un nouveau point de jonction qui correspond à l'exécution de l'action, dans lequel son corps s'exécute ($\mathcal{E}[\![e]\!]$) avant le point de jonction ($\pi\ v^*$). L'action étant de type `before`, l'instruction `proceed` n'existe pas dans son corps e ($\%\texttt{proceed} = None$). Une fois les actions définies, la sémantique du programme tissé est décrite par une fonction qui calcule son résultat et qui est un point fixe sur les fonctions des procédures du programme de base et des actions.

Cette sémantique a permis, en les formalisant, une meilleure compréhension de certain éléments de la programmation par aspect. Par exemple, le fonctionnement des points de coupure, qui est décrit par la fonction `match-pcd`. Malheureusement à l'instar de la fonction $\mathcal{A}$ décrite ci-dessus, ses règles sont complexes. De plus, la gestion des exceptions, qui est l'une des caractéristiques importante des langages d'aspect généraux car elle permet de mieux gérer les comportements anormaux (voir Section 1.1.2, page 7), n'est pas décrite. C'est aussi le cas du point de coupure dynamique `if` d'AspectJ, qui, comme présenté à l'exemple de la page 16 peut engendrer des comportements complexes.

MiniMAO$_1$: Une sémantique opérationnelle d'un langage d'aspect général

Pour mettre en évidence tous les détails de l'exécution, Clifton et Leavens [CL06] proposent d'utiliser une sémantique opérationnelle à petit pas [NN92]. Cette sémantique est celle d'un sous ensemble d'AspectJ basé sur le sous ensemble de Java, Featherweight Java [MP05] dont un fragment est présenté ci-dessous :

$$\begin{array}{lll} P & ::= & decl^*\ e \\ decl & ::= & \texttt{class}\ c\ \texttt{extends}\ c\ \{field^*\ meth^*\} \\ meth & ::= & t\ m(form^*)\ \{\ e\ \} \\ e & ::= & \texttt{new}\ \ c() \mid e.m(e^*) \mid e.f \mid e.f = e \\ & & \texttt{cast}\ t\ e \mid e; e \mid \ldots \end{array}$$

Ce langage possède les expressions classiques des langages à objets (création d'objet, appel de méthode, accès à un champs, séquence, ...). L'exécution de ces expressions et par là celui du programme est décrite en utilisant la relation $\hookrightarrow$ après avoir construit les environnements statiques à partir des déclarations ($decl$) du programme. La relation $\hookrightarrow$ prend une expression, une pile et un état et retourne une expression ou une exception (en cas d'erreur), une pile et un état.

$$\hookrightarrow : \mathcal{E} \times Stack \times Store \rightarrow (\mathcal{E} \cup Excep) \times Stack \times Store$$

L'argument $Store$ représente les objets du programme en associant à chaque adresse d'un objet, l'ensemble de ses champs et leurs valeurs. L'environnement $Stack$ représente la pile de flot de contrôle. Dans cette sémantique, il n'est pas modifié par le programme de base mais permettra avec l'ajout des aspects de filtrer le point de jonction courant. $\mathcal{E}$ est l'ensemble des expressions évaluées. L'évaluation de ces expressions pourra lever une exception appartenant à l'ensemble des exceptions $Excep$. Par exemple, l'appel d'une méthode avec un objet `null` lève une exception de type `NullPointerException`. Un contexte d'évaluation noté $\mathbb{E}$ est utilisé pour spécifier la stratégie d'évaluation d'une expression.

$$\begin{array}{lll} \mathbb{E} & ::= & \mathbb{E}.m(e\ldots) \mid v.m(v\ldots\mathbb{E}e\ldots) \mid (l(v\ldots\mathbb{E}e\ldots)) \mid \\ & & \mathbb{E}.f \mid \mathbb{E}; e \mid \mathbb{E}.f = e \mid v.f = \mathbb{E} \\ \\ l & ::= & \texttt{fun}\ m\langle var^*\rangle.e : \ldots \end{array}$$

Une expression est évaluée de la gauche vers la droite. Dans le cas d'un appel de méthode, le receveur et les arguments sont évalués avant d'exécuter le corps de la méthode. Afin de distinguer l'appel d'une méthode et son exécution (les motifs `call` et `execution` d'AspectJ), une expression sémantique $\texttt{fun}\langle\ldots\rangle.e$ est utilisée pour représenter l'exécution du corps e d'une méthode. Les règles CALL et EXEC ci-dessous décrivent l'exécution d'une méthode après l'évaluation du receveur et des arguments de celle-ci.

$$\begin{array}{l} \langle \mathbb{E}[loc.m(v_1,\ldots,v_n)], J, S\rangle \\ \quad \hookrightarrow \langle \mathbb{E}[(\texttt{fun}\ m\langle var_0,\ldots,var_n\rangle.e : \ldots (loc, v_1,\ldots,v_n))], J, S\rangle \quad \text{CALL} \\ \\ \langle \mathbb{E}[(\texttt{fun}\ m\langle var_0,\ldots,var_n\rangle.e : \ldots (v_0,\ldots,v_n))]\rangle \\ \quad \hookrightarrow \langle \mathbb{E}[e\{v_0/var_0,\ldots,v_n/var_n\}], J, S\rangle \qquad \text{EXEC} \end{array}$$

La sémantique des aspects et du programme tissé est décrite en étendant le langage de base avec les aspects définis comme suit :

$$\begin{array}{lcl} decl & ::= & \ldots \texttt{aspect}\ a\ \{field^*\ adv^*\} \\ adv & ::= & t\ \texttt{around}(form^*) : pcd\ \{\ e\ \} \\ pcd & ::= & \texttt{call}(pat) \mid \texttt{execution}(pat) \mid \texttt{args}(form^*) \mid \ldots \\ & & pcd \texttt{\&\&} pcd \mid !pcd \mid pcd \texttt{||} pcd \\ e & ::= & \ldots \mid e.proceed(e^*) \end{array}$$

Les aspects sont ceux d'AspectJ avec uniquement des actions de type `around`, des expressions e identiques aux expressions du langage de base, et une expression $e.proceed(e^*)$, qui permet d'exécuter les points de jonction (après avoir éventuellement modifié le receveur et les arguments) filtrés par les motifs de pcd. Ceux-ci sont similaires aux motifs présentés par Wand et coll. avec en plus le motif $\texttt{execution}(pat)$ qui filtre le début de l'exécution du corps d'une méthode pat. Des expressions sémantiques ajoutées à e sont utilisées pour définir des points de jonction et construire l'ensemble des aspects qui les filtrent.

$$e \ ::= \ \ldots \mid \texttt{jointpt}\ j(e^*) \mid \texttt{under}\ e \mid \texttt{chain}\ \bar{B}, j(e^*)$$

Parmi ces expressions, $\texttt{joinpt}\ j(e^*)$ représente le point de jonction j avec e^* qui sont les arguments de l'expression du langage source (appel de méthode) à laquelle j correspond.
La construction $\texttt{chain}\ \bar{B}, j(e^*)$ représente la liste des actions $\bar{B}$ qui filtre le point de jonction $j(e^*)$. Dans cette liste, chaque action est de la forme $[\![b, loc, e, \ldots]\!]$ avec b qui associe à chaque variable de motif une valeur, loc qui est l'instance de l'aspect (Pas de création dynamique d'aspect. Chaque aspect possède une instance unique `this` à l'exécution de son action) et e le corps de l'action. L'expression $\texttt{under}\ e$ indique que e est prête pour l'exécution (ne peut pas être transformée en point de jonction). Le point de jonction j est représenté par la grammaire ci-dessous qui précise le type du point de jonction (appel de méthode (`call`), exécution de méthode (`exec`), *etc.*), la valeur du receveur (v), le nom de la méthode (m) et le corps de la méthode et ses arguments (l).

$$\begin{array}{lcl} j & ::= & (\!|k, v, m, l, \ldots|\!) \\ k & ::= & \texttt{call} \mid \texttt{exec} \mid \ldots \end{array}$$

Chaque instruction du langage source qui peut être filtrée est donc d'abord transformée en point de jonction (CALL_A) ; ensuite la liste des actions qui filtre ce point de jonction est déterminée (fonction $adviceBind$ qui utilise une fonction $matchPCD$ définie comme dans Wand et coll. [WKD04]) et tissée ($\ll e \gg_{\bar{B},j}$). Par exemple,

$$\ll e_0.proceed(e_1, \ldots, e_n) \gg_{\bar{B},j} = \texttt{chain}\ \bar{B}, j(e_0, \ldots, e_n)$$

$$\ll e.f = e' \gg_{\bar{B},j} = \ll e \gg_{\bar{B},j} .f = \ll e' \gg_{\bar{B},j}$$

Lorsqu'il n'existe plus d'action à tisser ($\bar{B}$ est vide : $\bullet$), le point de jonction est exécuté si l'aspect possède proceed (CALL_B) sinon l'action remplace le point de jonction (ADVISE). Par exemple, l'appel d'une méthode se décrit maintenant comme suit

$$\langle \mathbb{E}[loc.m(v_1, \ldots, v_n)], J, S \rangle \\ \hookrightarrow \langle \mathbb{E}[\texttt{jointpt}(\!|\texttt{call}, -, m, -, \ldots|\!)(loc, v_1, \ldots, v_n)], J, S \rangle \quad \text{CALL}_A$$

$$\langle \mathbb{E}[\texttt{jointpt}\ j(v_0,\ldots,v_n)], J, S\rangle \hookrightarrow \langle \mathbb{E}[\texttt{under chain}\ \bar{B}, j(v_0,\ldots,v_n)], j+J, S\rangle \quad \text{BIND}$$
$$\text{où } adviceBind(j+J,S) = \bar{B}$$

$$\langle \mathbb{E}[\texttt{chain}\ [\![b, loc, e, \ldots]\!] + \bar{B}, j(v_0,\ldots,v_n)], J, S\rangle \hookrightarrow \langle \mathbb{E}[\texttt{under}\ e'\{loc/this\}(v_0,\ldots,v_n)/b], j'+J, S\rangle \quad \text{ADVISE}$$
$$\text{où } e' = \ll e \gg_{\bar{B},j} \text{ et } j' = (\!|\texttt{this}, loc, -, -, \ldots|\!)$$

$$\langle \mathbb{E}[\texttt{chain}\ \bullet, (\!|\texttt{call}, -, m, \ldots|\!)(loc, v_1,\ldots,v_n)], J, S\rangle \hookrightarrow \langle \mathbb{E}[(l(loc, v_1,\ldots,v_n))], J, S\rangle \quad \text{CALL}_B$$
$$\text{avec } S(loc) = [t\ .\ F] \text{ et } methodBody(t,m) = l$$

Clifton et Leavens associe à cette sémantique opérationnelle, une sémantique statique basée sur un système de type qui permet de vérifier si un aspect est bien typé.

La sémantique opérationnelle à petit pas permet de décrire toutes les étapes d'exécution d'un programme tissé (avec des aspects de type `around`). Par conséquent, ce travail permet de mieux comprendre les effets des aspects sur le programme de base et sur d'autres aspects. Mais cette sémantique est celle d'un langage particulier (sous ensemble d'AspectJ) basé sur un sous ensemble de Java. De plus, des éléments importants d'AspectJ et de la programmation par aspect n'ont pas été décrits :

- l'utilisation d'un prédicat pour filtrer l'état du programme (le point de coupure `if(...)` d'AspectJ) et l'interaction entre les aspects utilisant de tels points de coupure ;
- les exceptions et les actions portant sur ces exceptions (`handler`, `around throws` et `after throwing` d'AspectJ) ;
- la création des instances d'aspects à l'exécution (`percflow` et `pertarget` d'AspectJ). Seule la création d'une instance unique correspondant à l'aspect est considérée.

Sémantique opérationnelle pour un ensemble de langages d'aspect général

Contrairement aux travaux précédents ([CL06] et [WKD04]) qui s'appuient sur un langage particulier, Clifton et coll. [CLW03] proposent une sémantique opérationnelle qui permet de décrire la sémantique de plusieurs langages d'aspect généraux. Pour cela, ils s'appuient sur un calcul $\varsigma_{asp}(\mathbf{M})$ où $\mathbf{M}$ est un langage de point de coupure qui varie en fonction du langage d'aspect à décrire. Le calcul $\varsigma_{asp}(\mathbf{M})$ étend le calcul ς définit par Abadi et Cardelli[AC96] pour la programmation par objets en ajoutant la notion d'aspect. Cette extension se décrit comme suit :

$$\begin{array}{lcl}
\mathcal{P} & ::= & a \otimes \vec{\mathcal{A}} \\
a,b,c & ::= & x \mid v \mid a.k \mid a.k \Leftarrow \varsigma(x)b \mid \\
 & & \texttt{proceed}_{VAL}() \mid \texttt{proceed}_{IVK}(a) \mid \texttt{proceed}_{UPD}(a, \varsigma(x)b) \mid \ldots \\
k & ::= & l \mid f \\
v & ::= & d \mid [\overline{l_i = \varsigma(x_i)b_i}^{i\in I}] \\
\mathcal{A} & ::= & pcd \triangleright \varsigma(\vec{y})b \\
\ldots & &
\end{array}$$

Un programme $\mathcal{P}$ est un terme a suivi d'une séquence d'aspects $\vec{\mathcal{A}}$. Le terme a représente le programme de base à évaluer. Il peut être une variable (x), une valeur (v), un appel de méthode ($a.l$), une modification de méthode ($a.l \Leftarrow \varsigma(x)b$), un accès à un champs en lecture ($a.f$) et un accès à un champs en écriture ($a.f \Leftarrow \varsigma(x)b$). Un objet est représenté par une séquence de méthode $\varsigma(x_i b_i)$ dont le corps est b_i et où x_i est son receveur ($[\overline{l_i = \varsigma(x_i)b_i}^{i\in I}]$). Les valeurs peuvent être des constantes d ou des objets. L'aspect $\mathcal{A}$ associe à chaque point de coupure pcd de **M** une action $\varsigma(\vec{y})b$ dont le corps b est un terme qui peut comporter un $\texttt{proceed}_{VAL}$ pour exécuter la valeur filtrée par pcd. C'est aussi le cas de $\texttt{proceed}_{IVK}$ et $\texttt{proceed}_{UPD}$ qui permettent d'exécuter respectivement l'appel et la modification de la méthode filtrée par pcd.

Le comportement d'un tel programme est ensuite décrit par une sémantique opérationnelle à grand pas [NN92]. Lorsque le point de jonction courant n'est pas filtré par un aspect, l'instruction courante est réduite. Par exemple, la règle RED SEL 0 ci-dessous décrit l'exécution d'un appel de méthode lorsqu'il n'est pas filtré par un aspect ($advFor_M(\langle \ldots \rangle, \vec{\mathcal{A}}) = \bullet$).

$$\textsc{Red Sel 0}\ \frac{\begin{array}{c}\mathcal{K} \vdash_{M,\vec{\mathcal{A}}} a \rightsquigarrow o \qquad l_j \in \overline{l_i}^{i\in I} \\ advFor_M(\langle IVK, \mathcal{K}, \overline{l_i}^{i\in I}, l_j\rangle, \vec{\mathcal{A}}) = \bullet \\ \texttt{ib}(\overline{l_i}^{i\in I}, l_j).\mathcal{K} \vdash_{M,\vec{\mathcal{A}}} b_j\{\{x_j \leftarrow o\}\} \rightsquigarrow v\end{array}}{\begin{array}{c}\mathcal{K} \vdash_{M,\vec{\mathcal{A}}} a.l_j \rightsquigarrow v \\ \text{avec } o \stackrel{\Delta}{=} [\overline{l_i = \varsigma(x_i)}^{i\in I}]\end{array}}$$

La règle consiste d'abord à réduire son receveur (a), ensuite à appliquer au point de jonction qui lui correspond l'ensemble des aspects $\vec{\mathcal{A}}$ ($advFor_M$) et enfin le corps de la méthode est évalué après avoir substitué dans celui-ci la variable receveur x_j par sa valeur o ($b_j\{\{x_j \leftarrow o\}\}$) et mis au sommet de la pile de flot de contrôle $\mathcal{K}$ la signature de la méthode ($\texttt{ib}(\overline{l_i}^{i\in I})$).

Lorsqu'un point de jonction est filtré, l'ensemble des actions ($\varsigma(y)b+B$) qui le filtre est tissé autour du point de jonction ($close_{IVK}(b, \{(B+\varsigma(x_j)b_j), \ldots, \ldots\})$) et le résultat de l'évaluation de ce terme tissé (b') est retourné par le point de jonction. La règle RED SEL 0 présentée ci-dessus devient :

$$\textsc{Red Sel 1}\ \frac{\begin{array}{c}\mathcal{K} \vdash_{M,\vec{\mathcal{A}}} a \rightsquigarrow o \qquad l_j \in \overline{l_i}^{i\in I} \\ advFor_M(\langle IVK, \mathcal{K}, \overline{l_i}^{i\in I}, l_j\rangle, \vec{\mathcal{A}}) = \varsigma(y)b + B \\ close_{IVK}(b, \{(B + \varsigma(x_j)b_j), \overline{l_i}^{i\in I}, l_j\}) = b' \\ \texttt{ia}.\mathcal{K} \vdash_{M,\vec{\mathcal{A}}} b'\{\{y \leftarrow o\}\} \rightsquigarrow v\end{array}}{\begin{array}{c}\mathcal{K} \vdash_{M,\vec{\mathcal{A}}} a.l_j \rightsquigarrow v \\ \text{avec } o \stackrel{\Delta}{=} [\overline{l_i = \varsigma(x_i)}^{i\in I}]\end{array}}$$

L'évaluation du terme tissé $\Pi_{IVK}\{\ldots\}$ consiste à exécuter le point de jonction courant (Règle RED SPRCD 0) une fois que toutes les actions sont exécutées (Règle RED SPRCD 1)

$$\textsc{Red SPrcd 1}\ \frac{\mathcal{K} \vdash_{M,\vec{\mathcal{A}}} a \rightsquigarrow o \qquad B \neq \bullet \qquad close_{IVK}(b, \{B, \bar{l}, l\}) = b' \qquad \texttt{ia}.\mathcal{K} \vdash_{M,\vec{\mathcal{A}}} b'\{\{y \leftarrow o\}\} \rightsquigarrow v}{\mathcal{K} \vdash_{M,\vec{\mathcal{A}}} \Pi_{IVK}\{(\varsigma(y)b + B), \bar{l}, l\}(a) \rightsquigarrow v}$$

$$\textsc{Red SPrcd 0}\ \frac{\mathcal{K} \vdash_{M,\vec{\mathcal{A}}} a \rightsquigarrow o \qquad \texttt{ib}(\bar{l}, l).\mathcal{K} \vdash_{M,\vec{\mathcal{A}}} b\{\{y \leftarrow o\}\} \rightsquigarrow v}{\mathcal{K} \vdash_{M,\vec{\mathcal{A}}} \Pi_{IVK}\{\varsigma(y)b, \bar{l}, l\}(a) \rightsquigarrow v}$$

Lorsque l'ensemble des actions qui filtre un point de jonction n'est pas vide ($B \neq \bullet$), la Règle RED SPRCD 1 l'exécute en transformant chaque action courante en un terme tissé (b') qui est ensuite évalué. Une fois toutes les actions évaluées, la Règle RED SPRCD 0 exécute le point de jonction en évaluant d'abord son receveur (a), et ensuite son corps après avoir substitué sa variable receveuse par sa valeur. Le terme tissé est obtenu en appliquant récursivement la fonction $close$ sur le corps de chaque action qui filtre un point de jonction. Cette fonction appliquée à un terme `proceed` retourne un terme Π qui varie en fonction du type de proceed. Par exemple, pour $\texttt{proceed}_{IVK}$ $close$ est définie comme suit :

$$close_{IVK}(\texttt{proceed}_{IVK}(a), \{B, S, k\}) = \Pi_{IVK}\{B, S, k\}(close_{IVK}(a, \{B, S, k\}))$$

où B représente les actions qui restent à exécuter, S la signature de l'objet receveur du point de jonction (l'ensemble de ses méthodes) et k la méthode appelée. Par exemple, en considérant l'objet point

$$[\texttt{n} = \varsigma(\texttt{y})0, \texttt{pos} = \varsigma(\texttt{p})\texttt{p.n}]$$

où `n` est un champ qui permet de sauvegarder la position d'un point et `pos` une méthode qui permet de l'interroger. L'aspect

$$\mathcal{A} \triangleq [\texttt{n}, \texttt{pos}].\texttt{pos} \triangleright \varsigma(x)\texttt{proceed}_{IVK}(\texttt{x.n} \Leftarrow \varsigma(\texttt{y})0)$$

permet de sélectionner l'appel de la méthode `pos` et de mettre à 0 la valeur du champs `n` de son receveur avant l'exécution de `pos`. Ainsi sans aspect (Règle RED SEL 0), l'appel de `pos` donne :

$$\epsilon \vdash_{M,\bullet} [\texttt{n} = \varsigma(\texttt{y})2, \texttt{pos} = \varsigma(\texttt{p})\texttt{p.n}].\texttt{pos} \rightsquigarrow 2$$

Après le tissage de $\mathcal{A}$ (Règle RED SEL 1, RED SPRCD 1 et Règle RED SPRCD 0) on obtient

$$\epsilon \vdash_{M,\mathcal{A}} [\texttt{n} = \varsigma(\texttt{y})2, \texttt{pos} = \varsigma(\texttt{p})\texttt{p.n}].\texttt{pos} \rightsquigarrow 0$$

Ce calcul d'aspects a été utilisé pour décrire AspectJ [KHH+01a], HyperJ [OT01] et Adaptive Methods [LOO01] en traduisant leurs langages de points de coupure dans celui de pcd décrit par la grammaire :

$$\begin{array}{rcl} pcd & ::= & VAL \mid IVK \mid UPD \mid \\ & & \texttt{k} = k \mid \texttt{S} = S \mid \ldots \\ & & \mid \neg pcd \mid pcd \wedge pcd \mid pcd \vee pcd \\ \ldots & & \end{array}$$

Le point de coupure pcd peut être un motif VAL, IVK et UPD qui permettent respectivement de filtrer toute valeur, invocation et mise à jour. Le terme $\texttt{k} = k$ permet de filtrer tout point de jonction dont l'identificateur est k. Le terme $\texttt{S} = S$ permet de filtrer tout point de jonction dont

la signature du receveur est S. Un point de coupure peut aussi être une conjonction, disjonction et une négation de ces motifs. Par exemple,

$\texttt{call(void Point.pos())}$ est équivalent à $IVK \wedge \texttt{S} = \{\texttt{n}, \texttt{pos}\} \wedge \texttt{k} = \texttt{pos}$

et $\texttt{set(int Point.n)}$ est équivalent à $UPD \wedge \texttt{S} = \{\texttt{n}, \texttt{pos}\} \wedge \texttt{k} = \texttt{n}$

Cette traduction, qui ne concerne que les points de coupure, suppose que les actions soient décrites dans ς_{asp}. Cela implique que ce calcul est fait pour les langages à aspect basés sur les langages à objets. De plus, comme dans **MiniMAO**$_1$ [CL06] des éléments clés comme l'utilisation des points de coupure pour tester l'état d'un programme, les exceptions, et la création dynamique d'instances d'aspect, ne sont pas décrits.

μABC : un calcul minimal pour la programmation par aspect

Jagadeesan et coll. [BJJR04] proposent un calcul minimal μABC qui ne dépend pas d'un paradigme de programmation. Ce calcul a pour but de décrire et de comprendre les mécanismes de la programmation par aspect ainsi que leurs interactions avec les primitives d'un programme. Dans μABC, l'exécution de chaque commande génère un événement (message) qui est transmis d'un émetteur à un récepteur. Par exemple, la commande ci-dessous

$$\texttt{let}\ x = p \rightarrow q : m$$

transmet le message m de l'émetteur p au récepteur q et sauvegarde la valeur de retour du message m dans la variable x. Les variables p et q sont appelées des *rôles*. Un message est une étiquette $f, \ldots, l$ ou une action $a, \ldots, e$. Une commande peut aussi spécifier une séquence de messages. Cela permet de modéliser les appels de méthode $p \rightarrow q : l$ et les séquences d'actions $p \rightarrow q : a, b$. Dans le cas où une séquence de messages est émise, celle-ci l'est de la droite vers la gauche. La seule commande qui calcule une valeur est $\texttt{return}\ v$ qui retourne la valeur v. Ainsi,

$$\texttt{advice}\ a[p \rightarrow q : k] = \texttt{let}\ x = p \rightarrow r : l;\ \texttt{return}\ x$$

définit une action a qui filtre toute émission du message k de p à q et la rédirige comme un message l vers r. Cette commande pourrait permettre de déléguer tout message envoyé à q vers r avant qu'il soit reçu par q. Le terme entre crochet représente le point de coupure et celui après la première égalité, le corps de l'action. Des variables de motif associées à des quantificateurs ($\forall$ et $\exists$) sont utilisées dans des points de coupure afin de définir des ensembles de points de jonction. Des substitutions particulières permettent d'utiliser ces variables dans le corps des actions. Par exemple, l'action :

$$\texttt{advice}\ a[\exists z_s.\exists z_t\ z_s \rightarrow z_t : k] = \sigma y_s.\tau y_t.\pi b.\ \texttt{let}\ x = y_s \rightarrow y_t : b, l; \texttt{return}\ x$$

convertit tout message k en un message l émis de la même source vers le même récepteur et continue avec l'exécution des actions b. En effet, le point de coupure de a permet de filtrer tout envoi du message k. La substitution σ permet de récupérer la valeur de la source, celle τ la valeur du récepteur et celle π appelée *substitution proceed* permet d'exécuter d'autres actions.

S'il n'existe pas d'autres actions qui filtrent ce point de jonction, alors l'action $y_s \rightarrow y_t : b, l$ est identique à $y_s \rightarrow y_t : l, b$.

Un programme est une suite de déclarations $\vec{D}$ de rôles (et d'actions dans le cas du programme tissé) et une suite d'envoi de message qui représente son corps. Par exemple, le programme suivant

$$\vec{D}; \texttt{let}\ x = p \rightarrow q : j, k; \texttt{return}\ x$$

est un programme tissé dont les déclarations $\vec{D}$ sont

$$\texttt{role}\ p; \texttt{role}\ q; \texttt{role}\ r;$$
$$\texttt{advice}\ a[p \rightarrow q : k] = \sigma y_s.\tau y_t.\pi b.(\texttt{let}\ z = y_t \rightarrow r : b; \texttt{return}\ y_s);$$

Chaque étape de son exécution se décompose en deux temps : dans un premier temps, elle remplace le message filtré par le nom de l'action qu'il déclenche. Le programme ci-dessus devient :

$$\vec{D}; \texttt{let}\ x = p \rightarrow q : j, a; \texttt{return}\ x$$

Dans un second temps, elle remplace la commande d'envoi de message sur l'action déclenchée par le corps de celle-ci. Ainsi, le remplacement de $p \rightarrow q : a$ donne :

$$\vec{D}; \texttt{let}\ z = q \rightarrow r : j; \texttt{return}\ p$$

car une première substitution remplace la variable de retour x par la nouvelle z, l'émetteur du point de jonction p par celui de l'action y_t dont la valeur est q, le récepteur du point de jonction q par celui de l'action r et continue l'exécution avec le reste des messages b dont la valeur ici, est j. Une deuxième substitution remplace ensuite dans la continuation du programme $\texttt{return}\ x$, la variable de retour de la commande filtrée x, par la valeur de retour de l'action p. Il faut noter que, comme les points de coupure portent sur des étiquettes et comme celles-ci ont un espace de nom différent de celui des actions, les commandes des actions ne peuvent pas être filtrées par d'autres actions.

Dans le but de montrer que ce calcul est suffisant pour décrire des notions importantes des langages à aspects existants, les auteurs ont traduit dans μABC un langage fonctionnel MinAML [WZL03] et un langage à objets sous ensemble d'AspectJ [JJR03]. Par exemple, dans MinAML, qui étend le lambda calcul avec la notion d'aspect, l'expression suivante

$$p.z \rightarrow Q \gg P$$

exécute l'action Q si p filtre le point de jonction courant, puis continue avec l'évaluation de P. Q est exécutée après toutes les actions précédentes qui ont p comme point de coupure. z est une variable de motif dont la valeur est celle du point de jonction courant. Cette expression est traduite dans μABC par l'action :

$$p.z \rightarrow Q \gg P \triangleq \texttt{advice}[p.\texttt{hook}] = \tau y.\pi b(\lambda z.let\ x = (y \rightarrow y : b)(z); Q); P$$

Celle-ci filtre tout envoi de message $\texttt{hook}$ à p. Ce message réservé est envoyé par tout point de jonction filtré par p. Lorsque le point de jonction est filtré, sa valeur est passée via z aux autres

actions qui l'ont filtré ($(y \rightarrow y : b)(z)$). Une fois ces autres actions exécutées, Q est exécutée en associant à x la valeur de retour de ces actions.

Le calcul μABC permet de décrire les mécanismes clés de la programmation par aspect sans dépendre d'un paradigme particulier. Mais il se limite à des points de coupure qui permettent de filtrer l'exécution d'un évènement et ne permet pas de décrire des points de coupure tels que `cflow`. L'autre inconvénient est qu'il ne permet pas de filtrer les évènements générés par les actions, ce qui est le cas dans les langages d'aspect existants.

1.3.2 Vérification du programme tissé

Comme nous l'avons vu, les aspects en ajoutant du code au programme peuvent modifier les propriétés et la sémantique du programme de base. Des techniques comme la *vérification de modèles* ("model checking") [CEP00] ou l'analyse statique [CC77], utilisées pour vérifier des propriétés d'un programme ont été appliquées à la programmation par aspect pour vérifier qu'un aspect préserve les propriétés du programme de base. Mais elles peuvent être inefficaces. Une utilisation naïve de ces techniques tisse tous les aspects au programme de base et vérifie les propriétés du programme tissé. Des travaux ont amélioré cette technique en utilisant une méthode dite modulaire [KFG04, GK07].

Vérification modulaire des aspects

Krishnamurthi et coll. [KFG04] proposent une approche qui adapte la vérification de modèles de façon à permettre aux développeurs de vérifier les aspects sans accéder au programme de base. La technique consiste à générer automatiquement à partir du programme de base, d'un ensemble de propriétés qu'il doit satisfaire et d'un ensemble de points de coupure, des éléments qui vont permettre de vérifier uniquement l'action. Ces éléments qui sont vus comme des interfaces du programme de base, représentent des informations qui permettront à l'aspect de filtrer un point de celui-ci. Ils sont obtenus par un produit d'automates représentant le programme de base et les points de coupure. En effet, dans cette approche, le programme de base est représenté par son graphe de flot de contrôle qui est un système de transition étiqueté $M = (S, T, L, S_{src}, S_{sink}, S_{call}, S_{rtn})$ où

- S est l'ensemble des états. Ils représentent les commandes et expressions du programme de base ;
- $T \subseteq S \times S$ représente le flot de contrôle entre les états de M ;
- $L : S \rightarrow 2^{AP}$ sont des étiquettes associées à chaque état et qui représentent les propositions atomiques satisfaites par chacune d'elle. Les propriétés vérifiées ici sont celles de la logique temporelle CTL. Dans cette logique, chaque propriété est une proposition atomique ou une conjonction, disjonction et négation de ces propositions. Elle permet aussi d'exprimer des propriétés temporelles sur des formules CTL. Une description plus détaillée de CTL se trouve dans [QS82]. Par la suite nous utiliserons LTL et CTL* (voir

Section 3.3.2, page 59 et Section 3.5.2, page 69) qui sont des logiques proches de CTL ;
- $S_{src} \in S$ et $S_{sink} \in S$ sont respectivement l'entrée et la sortie du programme de base ;
- $S_{call} \subset S$ et $S_{rtn} \subset S$ sont respectivement des états d'appel et de retour de fonction. Chaque appel est lié à un retour et tout état S_{call} a pour transition $call(p)$ et S_{rtn} pour transition $return(p)$ qui correspondent à l'appel et au retour de la fonction p.

Les actions sont aussi représentées par ce système de transition. Comme les points de jonction filtrés par cette approche sont uniquement les appels de fonction, les points de coupure (PCD) sont représentés par des expressions régulières décrites par la Grammaire 1 où les atomes a sont des motifs sur les appels de fonction (`call`(f)) avec a^* qui est une séquence éventuellement vide de a. $PCD_1; PCD_2$ permet de filtrer une séquence d'évènements.

GRAMMAIRE **1.**

$$\begin{array}{lll} PCD & ::= & e \mid PCD_1; PCD_2 \mid \ldots \\ e & ::= & a \mid a^* \mid e_1 \vee e_2 \mid e_1 \wedge e_2 \mid \ldots \\ a & ::= & \texttt{true} \mid \texttt{call}(f) \mid \texttt{!call}(f) \end{array}$$

Par exemple le programme de base représenté par la Figure 1.2 invoque une fonction f qui appelle g. A la fin de f, la fonction h est appelée.

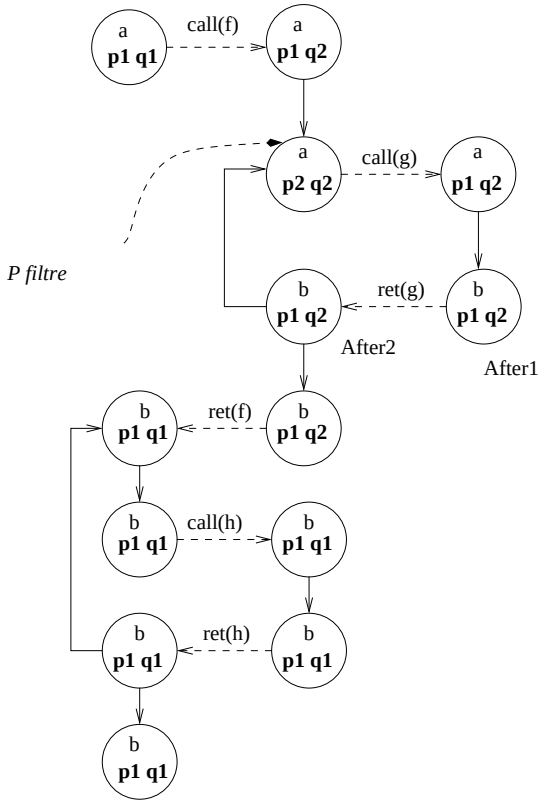


FIG. 1.2 – Produit du programme de base et des points de coupure

Ce programme satisfait la propriété $\forall\Box(\forall(a \cup b))$ qui spécifie que a est vraie à tous les états et

sur tous les chemins jusqu'à ce que b le soit. Soit A un aspect qui possède les points de coupure P et Q ci-dessous

$$P : \texttt{true}^*; \texttt{call}(g)$$
$$Q : \texttt{true}^*; \texttt{call}(f); \texttt{true}^*; \texttt{call}(h)$$

et dont l'action est représentée par la Figure 1.3, pour vérifier que A préserve la propriété

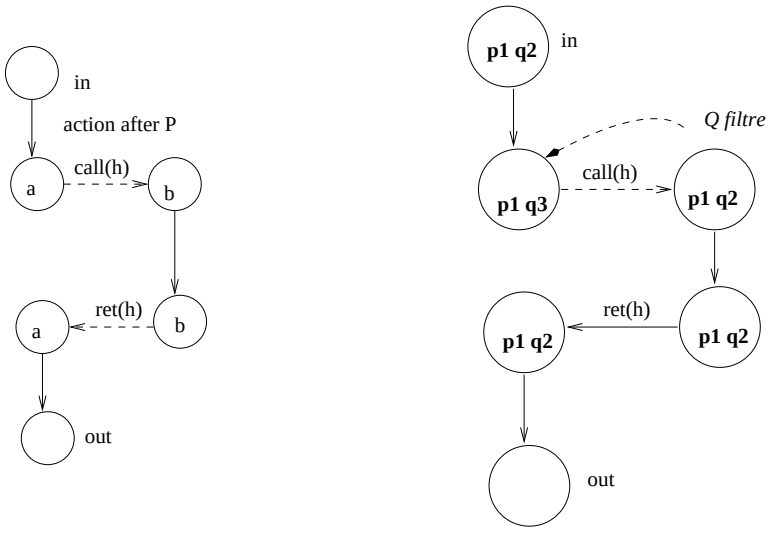


FIG. 1.3 – L'action A et son équivalent étiqueté par des points de coupure

satisfaite par le programme de base, les points de coupure P et Q sont transformés en automates et le produit de ceux-ci avec le programme de base permet d'étiqueter les états du programme de base avec les états $p1$ et $p2$ (resp. $q1$ et $q2$) de P (resp. Q). L'étiquette $p2$ est le témoin d'un appel de g tandis que $p1$ indique que le programme s'apprête à le faire. Il étiquette donc tous les états où g est appelée. De même $q1$ indique l'état initial de Q tandis que $q2$ signale que l'appel à f est dans la pile des appels. Ces étiquettes permettent ainsi d'identifier les états qui satisfont les points de coupure et où les actions sont appliquées (voir Figure 1.2). Le système de transition obtenu après le produit entre le programme de base et les points de coupure est ensuite passé au vérificateur avec la propriété à vérifier. Le vérificateur étiquette chaque état avec les sous formules de la propriété qui sont satisfaites à celui-ci. A chaque état qui satisfait un point de coupure, une interface qui va permettre de vérifier l'action à insérer de façon isolée est générée. Cette interface contient les étiquettes de l'état qui mène à l'action et ceux de l'état vers lequel l'état de retour de l'action mène. Dans notre exemple, uniquement P est satisfait et l'interface générée correspond aux étiquettes des états marqués After1 et After2. Ainsi, l'action (à gauche de la Figure 1.3) qui, seule, ne satisfait pas la propriété $\forall\Box(\forall(a \cup b))$ (car l'état avant l'état de retour de l'action satisfait a mais pas b) est vérifiée en associant à l'état source (in) de l'action, les étiquettes de After1 (avec b comme sous formule) et à l'état de sortie, les étiquettes de After2 (avec b, $\forall(a \cup b)$, $\forall\Box(\forall(a \cup b))$). Le vérificateur le fait en vérifiant que chaque état de l'action ne viole pas les sous formules de son état de sortie. Dans le cas où ces formules sont satisfaites (comme avec A), le programme tissé satisfait la propriété. Dans le cas contraire, l'action peut violer la propriété si l'état qui ne satisfait pas une sous formule dépend de la propriété.

Cette technique repose sur des approximations. Par exemple, son application aux actions des aspects `around` qui ne possèdent pas d'instruction `proceed` ne détecte pas en général la violation des propriétés de la forme $\forall\Diamond\varphi$ ou $\exists\Diamond\varphi$. La propriété $\forall\Diamond\varphi$ spécifie le fait que la propriété φ sera vraie dans tous les chemins tandis que $\exists\Diamond\varphi$ le fait qu'il existe un chemin où φ est vraie. C'est le cas de l'exemple de la Figure 1.4. Dans cet exemple, l'aspect `around` sans `proceed` et qui ne satisfait pas la propriété b est représenté en pointillé tandis que le programme de base est sans pointillé. Le programme de base satisfait la propriété $\forall\Diamond(b)$ car des trois états du programme de base, l'état du milieu satisfait b et le dernier état boucle sur l'état initial. En appliquant la technique de vérification décrite, le modèle de l'aspect a comme interface à l'état initial et à l'état final $\forall\Diamond(b)$. L'état du programme de base où b est satisfait est remplacé par un ensemble d'état qui ne satisfait pas b car l'aspect ne possède pas de `proceed`. La vérification de la formule $\forall\Diamond(b)$ de l'état de sortie sera satisfaite à tous les états de l'action (l'état initial la satisfait) pourtant elle est violée par l'aspect qui supprime l'état où b est satisfaite. Pour pallier à cette approximation, la propriété $\forall(\neg return \cup b)$ qui spécifie que la

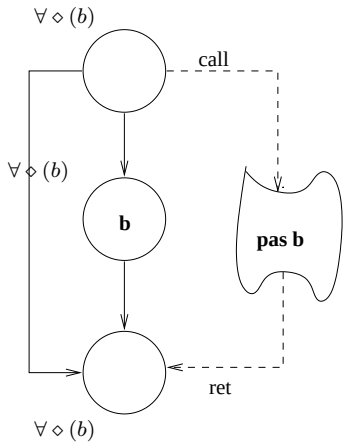


FIG. 1.4 – Exemple d'approximation

propriété $\neg return$ est satisfaite jusqu'à ce que b le soit, vérifiée par le programme de base, est générée comme une interface de l'état initial de l'action (état $call$). Il permet de vérifier que l'action possède un état qui satisfait b avant la fin de son exécution (état ret).

Des subtilités liées à cette approche sont décrites par les auteurs mais celle qui retient notre attention est la prise en compte par le modèle du fait que les instructions des actions peuvent être filtrées (caractéristique importante du tissage car elle peut mener à sa non terminaison. Elle est possédée par les langages de programmation par aspect). C'est le cas de l'action de A, qui, lorsqu'elle appelle h dans le programme tissé satisfait Q. Cela est détecté en analysant uniquement l'action en utilisant les étiquettes des points de coupure qui sont passées à l'action par l'intermédiaire des interfaces. Dans l'exemple, $p1$ et $q2$ sont passées à l'état in et comme $q2$ spécifie qu'on est dans le flot de contrôle de la fonction f, cela permet de savoir que Q est satisfaite à l'appel de h (voir à droite de la Figure 1.3).

Cet article présente une approche modulaire qui utilise la vérification des modèles afin de garantir que l'aspect ne modifie pas les propriété du programme de base. Il consiste à générer automatiquement à partir du programme de base des interfaces qui vont permettre de vérifier uniquement l'aspect. Un algorithme qui optimise cette technique permet de vérifier une propriété en un temps linéaire de la taille du modèle (le programme de base ou l'action). Cependant, c'est une technique incomplète (basée sur des approximations) qui s'applique à un programme de base donné et un aspect donné. De plus, elle suppose que, si l'aspect termine, il revient au point du programme de base qui suit directement le point de jonction. Ce travail a été étendu par Goldman et Katz [GK07] afin de vérifier que pour tout programme de base qui satisfait une hypothèse ψ, le programme tissé satisfait une propriété ζ éventuellement différente de ψ. Il suppose que l'aspect tissé peut modifier le flot de contrôle du programme de base à condition de retourner sur un état de celui-ci (n'importe lequel). Contrairement à Krishnamurthi et coll., l'aspect ne peut pas filtrer les instructions des actions.

1.3.3 Classification des aspects

La vérification de modèles vérifie les propriétés du programme tissé par rapport à un aspect donné ; de telle sorte, qu'en général une modification de l'aspect ou du programme de base nécessite une nouvelle vérification. Pour remédier à cet inconvénient, des travaux [CL02, RSB04, Kat06, DW06] caractérisent des catégories d'aspect en fonction de l'impact qu'ils peuvent avoir sur le programme de base. Cette approche consiste à déterminer les propriétés du programme de base qui restent satisfaites après le tissage de tout aspect appartenant à une catégorie [Kat06, DW06]. Il suffit après une exécution ou modification d'un aspect, de déterminer à quelle catégorie il appartient pour connaître les propriétés qui sont préservées après le tissage. De même, à la modification du programme de base, il suffira de connaître les propriétés satisfaites par le programme de base pour connaître celles qui sont préservées par le tissage. Ces travaux proposent pour caractériser les catégories d'aspect d'utiliser, soit l'analyse des programmes et des aspects [Kat06], soit des méthodes par construction [DW06, CLN07].

Classification d'aspects et classes de propriétés

Katz [Kat06] propose une classification des aspects en fonction des propriétés que ceux-ci permettent de préserver après le tissage. Les catégories définies sont : *spectateur*, *régulateur* et *invasif* ("spectative", "regulative" et "invasive"). Pour chaque catégorie, des classes de propriétés qui sont préservées par le programme tissé sont données. Ces classes sont les classes standards de propriétés exprimées en logique temporelle (sûreté, vivacité et invariant), et une classe dite d'*existence* définie par Katz. Une propriété de sûreté spécifie que quelque chose n'arrive jamais, une propriété de vivacité que quelque chose arrivera et un invariant que quelque chose arrive toujours. Ces propriétés se spécifient pour toutes les exécutions possibles d'une application. Une propriété d'existence est celle qui spécifie des propriétés sur quelques exécutions dans l'ensemble des exécutions d'une application.

Le modèle sémantique utilisé représente chaque programme par un graphe à états de toutes ses exécutions possibles. Chaque état du graphe associe les variables à leurs valeurs, et une arrête représente une instruction atomique du programme. La sémantique d'une exécution est une trace où chaque noeud et arête représentent respectivement l'état et l'instruction courante du graphe à états. Ainsi, il définit la sémantique du tissage des aspects `before`, `after` et `around` par transformation de graphe. Par exemple, lorsqu'un aspect `before` filtre une arête d'une trace, le graphe à états de l'action est tissé juste avant le noeud généré par l'instruction filtrée et le dernier état de l'action est relié à ce noeud par l'arête filtrée qui a été déplacée. La Figure 1.5 présente un exemple de transformation d'une trace du programme de base dont un fragment est la Figure 1.5 (à gauche) et qui a pour état x. Dans cette transformation, l'action d'un aspect `before` est tissée avant l'instruction filtrée p (voir à droite de la Figure 1.5). L'aspect possède un état w qui n'est pas modifié par le programme de base.

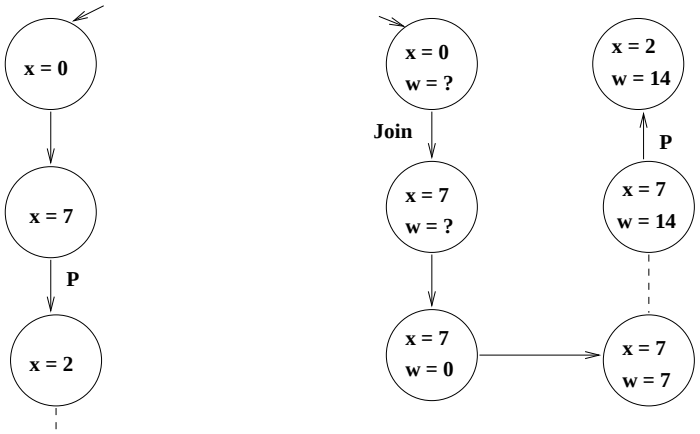


FIG. 1.5 – Fragment du graphe d'états d'un programme de base (à gauche) et son équivalent tissé par un aspect *before* (à droite)

Un aspect est dit spectateur, s'il ne peut pas modifier l'état et le flot de contrôle du programme de base. Il peut néanmoins modifier son propre état et insérer des instructions entre les instructions du programme de base. Il ne modifie pas le résultat final du programme de base. Plus précisément, un aspect est spectateur si la projection du graphe d'états tissé sur les variables du programme de base est identique au graphe d'états du programme de base à l'exception des états qui sont répétés et qui sont reliés par les instructions de l'action. La Figure 1.5 (à droite) présente l'exemple d'un fragment d'un graphe à états sur lequel a été tissé un aspect spectateur. Par exemple les aspects de débogage présentés en 1.1.2 sont des spectateurs. On peut déterminer par analyse de celui-ci si un aspect (à la AspectJ) est spectateur. Pour cela, Katz énumère les contraintes à vérifier par une telle analyse pour un aspect donné : l'aspect ne doit pas modifier les variables du programme de base, doit se terminer et les actions `around` doivent effectuer exactement un proceed. Katz se base sur la définition des aspects spectateurs pour prouver de façon informelle que toutes les propriétés de sûreté, de vivacité, d'existence et les invariants satisfaits (à l'exception des propriétés qui parlent de "l'état suivant d'un état" qui

peut être exprimé par l'opérateur $\bigcirc^2$("next")) par le programme de base restent satisfaits après le tissage de tout aspect appartenant à la catégorie des spectateurs.

Un aspect est dit régulateur s'il ne modifie pas les variables du programme de base mais peut modifier son flot de contrôle en restreignant ou supprimant certaines opérations ou en terminant l'exécution du programme de base. La suppression des opérations ici, n'est pas leurs remplacements par l'instruction vide (nop) mais le choix d'une opération entre deux opérations non déterministes. Aussi, restreindre une opération consiste à lui ajouter des contraintes de telle sorte que soit les contraintes sont respectées et l'opération est exécutée, soit les contraintes ne sont pas respectées et le programme est terminée. Par exemple les aspects de sécurité qui arrêtent l'exécution du programme lorsqu'une propriété critique va être violée sont des aspects régulateurs. Ainsi, en général, un aspect est régulateur si la projection du graphe d'états tissé sur les variables du programme de base est identique au graphe d'états du programme de base à l'exception des états qui se répètent et qui sont reliés par les instructions de l'action et des états et transitions supprimées. Comme pour les spectateurs, katz énumère des critères suffisants qu'une analyse peut vérifier pour assurer statiquement qu'un aspect est un régulateur. L'analyseur vérifie que l'aspect ne modifie pas les variables du programme de base, que son action termine et qu'il effectue exactement un `proceed` dans le cas d'un aspect `around`. Mais aussi, on peut permettre que l'aspect puisse lever une exception qui termine le programme ou utilise un mécanisme de sélection (*par ex.,* l'opérateur d'exclusion mutuel `wait`) d'une opération dans le cas d'exécutions concurrentes. Katz se base sur la définition des régulateurs pour prouver de façon informelle que le tissage de tout aspect appartenant à cette catégorie permet de préserver dans le programme tissé toutes les propriétés de sûreté et les invariants qui ne portent pas sur "l'état suivant d'un état". Les propriétés de vivacité ne sont pas préservées car l'aspect peut arrêter l'exécution du programme tissé avant que ces propriétés ne soient satisfaites. Aussi, les propriétés d'existence ne sont pas préservées car l'aspect peut choisir de supprimer les exécutions sur lesquelles portent ces propriétés.

Un aspect est dit invasif s'il peut modifier les variables et le flot de contrôle du programme de base. En général, ce type d'aspect peut modifier complètement la sémantique du programme de base et par conséquent ne préserver aucune propriété de celui-ci dans le programme tissé. Néanmoins, un aspect est *faiblement invasif* ("weakly invasive") lorsqu'il modifie les variables et le flot de contrôle du programme de base de manière à ce que, à chaque exécution d'une instruction du programme de base, l'état appartient à l'ensemble des états du programme de base. Ainsi, dans le graphe d'états du programme tissé, les transitions qui sont celles du programme de base commencent aux états qui existent dans le graphe d'états du programme de base et ce, même pour des entrées différentes des entrées du programme tissé. Les aspects de tolérances aux fautes qui permettent de restaurer des états précédents sûrs sont faiblement invasifs. En général, il n'est pas possible de déterminer statiquement qu'un aspect est invasif ou faiblement invasif. Katz affirme que les aspects faiblement invasifs ne préservent pas les propriétés de sûreté, de vivacité et d'existence car ils peuvent modifier les variables et le flot de contrôle du programme de base. Il prouve de façon informelle en se basant sur leur définition que le tissage de tout aspect faiblement invasif préserve tout invariant du programme de base à condition qu'il

[2] définit à la Section 3.3.2, page 59

soit satisfait à l'entrée de l'action et montré satisfait à tout état de l'action. Cela nécessite de vérifier chaque aspect faiblement invasif comme à la Section 1.3.2, page 27 avant de connaître si les invariants sont préservés.

Katz présente une approche qui identifie des catégories d'aspects et pour chacune des catégories, donne des classes de propriétés (sûreté, vivacité, invariant et existence) du programme de base qui restent satisfaites par le programme tissé quelque soit l'aspect tissé appartenant à la catégorie correspondante. Dans la plupart des cas, ce processus évite de vérifier le programme tissé et nécessite uniquement de savoir dans quelle catégorie appartient l'aspect tissé. Cependant il n'est pas formalisé et est dans certain cas imprécis. Ainsi, dans le cas des aspects spectateurs, lorsque les propriétés portent sur les événements, les classes de propriétés énoncées par Katz ne sont pas toutes préservées. Par exemple, la propriété de sûreté "l'événement `delete` n'a jamais lieu" satisfaite par un programme de base est invalidée par un aspect spectateur qui insère `delete`. Ce contre exemple indique qu'il faudrait distinguer les propriétés sur les événements de celles sur les états. Pour les régulateurs, le contre exemple présenté ci-dessus reste valable. De plus, les classes de propriétés préservées par les régulateurs excluent les propriétés de vivacité et d'existence. Pourtant, en considérant uniquement les aspects qui arrêtent l'exécution du programme, on pourrait préserver les propriétés d'existence. De même, en considérant uniquement les aspects qui sélectionnent des exécutions, on pourrait préserver les propriétés de vivacité. En ce qui concerne les aspects faiblement invasifs, comme l'exécution de chaque instruction du programme de base commence à un état qui existe dans le graphe d'état du programme de base, il semble que si les aspects terminent alors les propriétés de vivacité sur les invariants du programme de base sont préservées. De plus, certains aspects peuvent modifier les variables et le flot de contrôle du programme de base pendant l'exécution des actions sans violer ses invariants. Par exemple, un aspect de tolérance aux fautes qui restaure les états précédents du programme de base de manière atomique reste toujours dans l'ensemble des états atteignables par le programme de base. L'auteur propose aussi d'effectuer à *posteriori* des analyses pour déterminer si un aspect appartient à une catégorie. Ces analyses sont en générales indécidables et peuvent être complexes (*par ex.,* analyse d'alias, analyse de terminaison de l'action, . . .).

Classification d'aspects et langages

Dantas et Walker [DW06] proposent une catégorie d'aspects dit *inoffensifs* ("harmless"). Ces aspects n'interfèrent pas avec l'exécution du programme de base (*c.-à-d.,* ne modifient pas les variables du programme de base) mais peuvent modifier son flot de contrôle en le terminant. Ainsi, si le programme tissé termine, son résultat final est celui du programme de base. Cette catégorie représente un sous ensemble des régulateurs de Katz. Dantas et Walker présentent aussi un langage et un système de type qui assurent que tout aspect bien typé est dans la catégorie des inoffensifs. La sémantique de ce langage est décrite en utilisant le calcul défini par Walker, Zdancewic et Ligatti [WZL03]. Dans ce calcul, chaque point de jonction est étiqueté par un label l et l'exécution de toute expression $l[e_1]; e_2$ évalue d'abord e_1, ensuite exécute toutes les actions associées à l avec comme entrée la valeur v, résultat de e_1, et continue avec l'évaluation de e_2. Une action $\{pcd.x \rightarrow e\}$ est une expression qui est calculée lorsque l'exécution du pro-

gramme tissé atteint un label du flot de contrôle filtré par le point de coupure pcd représenté par un ensemble d'étiquette $\{l_1, \ldots, l_n\}$. Lorsque l'action est déclenchée, la valeur de l'expression associée au label est affectée à x qui peut être utilisée dans le corps de l'action e. L'opérateur $\Uparrow\ a$ active toute nouvelle action a. Lorsque l'exécution atteint un label appartenant au point de coupure, le corps de l'action est exécuté après toutes les actions précédemment activées. Par exemple, le code ci-dessous imprime `3 :hello world` (en supposant qu'il n'existe pas d'autres actions associées au label l dans l'environnement).

$\Uparrow \{\{l\}.x \rightarrow$ `printint x;print " :hello"`$\}$;
$\Uparrow \{\{l\}.y \rightarrow$ `print "world"`$\}$;
$l[3]$;

Le langage et le système de type qui assurent que les aspects sont inoffensifs s'appuient sur deux niveaux de langages. Un langage de haut niveau utilisé par le programmeur et un langage intermédiaire où toutes les analyses formelles sont effectuées. Cette démarche est justifiée par le fait que dans ce contexte, des analyses au niveau du langage source pourraient être complexes et générer des imprécisions ou des erreurs. Ce langage de haut niveau est un langage fonctionnel proche de ML, appelé AspectML [DWWW08], qui contient des objets et des aspects. Par exemple, le programme de base ci-dessous représente un objet `sys` qui implémente des appels systèmes. L'appel système `openW` ouvre un fichier (type de retour `file`) avec l'appel système `openO` si le fichier de nom `y` (paramètre d'entrée de type `string`) existe, sinon c'est l'appel `openC` qui est utilisé. `x` est la variable qui représente l'objet receveur. Pour les autres appels systèmes, seules les signatures sont présentées.

```
object sys = [
openW : file = x.y:string.
if (x.exists(y))
then (x.openO (y)) else (x.openC (y)),
openA : file = x.y:string. ...,
openR : file = x.y:string. ...,
openO : file = x.y:string. ...,
openC : file = x.y:string. ...,
write : int  = x.y:file * string. ...,
read : string = x.y:file * int. ...,
exists: bool  = x.y:string. ...,
...
]
```

L'aspect `limitdirectories`, est un aspect `before` qui empêche le programme de base d'ouvrir un fichier dans un répertoire différent de `tests`. Avant toute ouverture de fichier (filtrée par les points de coupure : `sys.openR`, `sys.openW`, `sys.openA`, `sys.openC`, `sys.openO`) par le programme, l'aspect teste si ce fichier est dans le répertoire `test`. Si tel est le cas, l'aspect ne fait rien (retourne `()`) et l'ouverture a lieu. Sinon, il imprime un message d'erreur sur la sortie standard et arrête l'exécution du programme. Notons que `x`, `y`, `s` et `n`

sont des variables de motif auxquelles sont respectivement associées après le filtrage d'un appel de méthode, le receveur, les arguments de la méthode, la pile de flot de contrôle et le nom de la méthode. Ces variables existent par construction dans les actions d'AspectML qui filtrent uniquement les appels de méthode.

```
limitdirectories:{
  (string alloweddirectory = "tests/")
  (before sys.openR, sys.openW, sys.openA, sys.openC,
            sys.openO (x,y,s,n) =
    let
      (string directory =
          substring (y, 0, (lastindexof (y, "/")+1)))
    in
      (if (directory == alloweddirectory)
        then ()
        else ((print "Forbidden directory.");
              (abort ())) ))
}
```

Le langage intermédiaire dans lequel est inclus le langage source est basé sur du lambda calcul typé contenant des chaînes de caractère, des booléens, des tuples, et des objets. Les actions des aspects sont traduites en des expressions $\Uparrow\{e.x \rightarrow_p e\}$. Pour s'assurer que les aspects appartiennent à la catégorie des inoffensifs, un système de type est utilisé pour vérifier que les aspects ne modifient pas les variables du programme de base. Pour cela, un domaine de protection (p) est attribué à chaque expression. Les domaines de protection sont organisés par une relation d'ordre ($\leq$) où $p_1 \leq p_2$ signifie que p_1 n'interfère pas avec p_2. Par souci de simplicité du système de type, pour éviter que des données ne puissent être transférées des aspects inoffensifs vers le programme de base, le type de retour du corps des actions doit être `unit` (*c.-à-d.,* ne retourne aucune valeur `()`). Un aspect est donc bien typé (*c.-à-d.,* de type $\texttt{advice}_{p'}$) si son domaine de protection (p') est inférieur à celui du point de jonction (p) qui le génère et si le type de retour du corps de son action est `unit` (Règle ACTION).

$$\textsc{Action}\quad \frac{\Gamma \vdash \tau\ \texttt{pcd}_p \qquad \Gamma, x:\tau; p' \vdash e : \texttt{unit} \qquad \vdash p' \leq p}{\Gamma \vdash \{v.x \rightarrow_{p'} e\} : \texttt{advice}_{p'}}$$

Par exemple, l'aspect `limitdirectories` est bien typé car son type de retour est `unit`. En effet, soit il retourne `()` qui est de type `unit`, soit il appelle des fonctions `print` et `abort` qui sont des fonctions prédéfinies de types `unit`. Cependant, cette contrainte imposée par le système de type exclut des aspects qui sont inoffensifs par définition (*c.-à-d.,* les aspects qui n'interfèrent pas avec le programme de base) mais qui ont un type de retour différent de `unit`. C'est le cas par exemple de l'aspect `profile` ci-dessous qui compte le nombre de fois (`z`) qu'un appel système est appelé pour l'ouverture d'un fichier.

```
profile:{
```

```
 (int z = 0)
 before sys.openR, sys.openW, sys.openA, sys.openC,
             sys.openO (x,y,s,n) =
 z := z + 1
}
```

D'après le système de type proposé par Dantas et Walker, le type de retour de l'action de `profile` est celui de $z+1$ *c.-à-d.,* de type `int` qui est différent de `unit`. Pourtant `profile` est inoffensif car il ne fait que modifier sa variable z.

Cet article présente de façon formelle, une méthode qui garantit par construction qu'un aspect appartient à la catégorie des inoffensifs. Il impose par un système de type que le type de retour de toute action d'un aspect inoffensif doive être `unit`. Cette approximation exclut des aspects qui sont inoffensifs par définition mais dont le type de retour n'est pas `unit`. Les inoffensifs représentent une classe restreinte des aspects et beaucoup d'autres aspects n'appartiennent pas à cette catégorie. De plus, les propriétés du programme de base (à l'exception du résultat final) qui sont préservées après le tissage des aspects ne sont pas décrites.

Clifton, Leavens et Noble utilisent dans [CLN07] une méthode basée sur des annotations de Java 1.5 afin de spécifier les effets des aspects sur le programme de base et sur d'autres aspects. Ces effets sont parmi ceux étudiés par Katz et d'autres. Ainsi, un aspect spécifier `@surround` ne modifie pas le flot de contrôle du programme de base (spectateur) tandis qu'un aspect `@curbing` peut le modifier en l'arrêtant (régulateur/inoffensif). Les propriétés spécifiées par ces annotations sont ensuite assurées par transformation automatique de programme et système de type. En supposant que les actions des aspects terminent, les aspects `before` et `after` qui sont spécifiés `@surround` sont transformés en "`try` action `catch (`Exception `e) { ;}`" afin de récupérer et supprimer toutes les exceptions levées par les actions car elles pourraient modifier le flot de contrôle. Pour les aspects `around`, il est vérifié au préalable qu'ils possèdent exactement un proceed ; ensuite la transformation ci-dessus est appliquée à la partie `before` et `after` de l'aspect de telle sorte que l'exception qui pourrait être levée par le point de jonction ne soit pas récupérée par la transformation et s'exécuterait ainsi comme dans le programme de base. La transformation s'appuie également sur le fait que les variables du programme de base qui sont dans les actions sont accessibles uniquement en lecture. L'annotation `@readonly` spécifie cela à la déclaration des variables utilisées dans les points de coupures et qui peuvent faire références aux variables du programme de base. Cette spécification qui est garantie sans utiliser une analyse d'alias suppose que toutes les variables des aspects auxquelles sont affectées les variables des points de coupures et les variables du programme de base doivent être déclarées `@readonly` et `final` (constante). Un aspect spécifié `@curbing` est assuré en utilisant les transformations des aspects `@surround` sans récupérer les exceptions levées par l'action. Les effets sur les variables du programme de base et sur celles des aspects peuvent être spécifiés en utilisant une notion de *domaine* qui utilise les types génériques de Java 1.5 afin de spécifier l'ensemble dans lequel appartient une variable à sa déclaration. Une annotation est ensuite appliquée à ces domaines afin de spécifier l'effet possible sur l'ensemble des variables du domaine. Par exemple, l'action ci-dessous

```
@writes({"Owner"})
after(@readonly Model<Other> m):
     target(m) && call(void Model<Other>.*)
{ corps de l'action }
```

spécifie que les variables du domaine `Owner` sont modifiables tandis que la variable `m` de type `Model` appartenant au domaine `Other` est accessible uniquement en lecture dans le corps de l'action. Les domaines sont construits statiquement avant d'être utilisés dans un système de type qui garantie les effets des annotations sur des variables du programme. Ce système de type est basé sur les langages de base et aspect utilisés dans **MiniMAO**$_1$ étendus avec les types génériques. La sémantique du tissage des aspects est celle de **MiniMAO**$_1$ (voir Section 1.3.1, page 20). Chaque type est de la forme $\delta\ T\langle\gamma_1, \ldots, \gamma_n\rangle$, où δ est soit `readonly` ou vide, T est le nom d'une classe ou d'un aspect et γ_i des domaines. Dans le cas où δ est vide, la variable peut être modifiée ou lue. Ainsi, par exemple, toute modification d'un champ f est bien typé si le domaine (γ_1) de f est dans l'ensemble des domaines qui peuvent être modifiés ($\hat{\gamma}$) et si le type de l'expression qui modifie f est un sous type du type de f ($s \preceq t$). Le domaine de f est obtenu en déterminant le type de son receveur ($\Gamma \centerdot \hat{\gamma} \vdash_{DT} e_1 : T\langle\gamma_1, \ldots, \gamma_n\rangle$). La fonction *fieldsOf* est appliquée à ce type pour déterminer le type de f (voir Règle T-SET).

$$\text{T-SET}\quad \frac{\begin{array}{c}\Gamma \centerdot \hat{\gamma} \vdash_{DT} e_1 : T\langle\gamma_1, \ldots, \gamma_n\rangle \\ \gamma_1 \in \hat{\gamma} \qquad fieldsOf(T\langle\gamma_1, \ldots, \gamma_n\rangle)(f) = t \qquad \Gamma \centerdot \hat{\gamma} \vdash_{DT} e_2 : s \qquad s \preceq t\end{array}}{\Gamma \centerdot \hat{\gamma} \vdash_{DT} e_1.f = e_2 : s}$$

Dans ce système de type, chaque aspect est un domaine. Ainsi, l'annotation `@spectator` permet de spécifier implicitement qu'un aspect est `@surround` et que les seules variables modifiables dans l'aspect sont celles de son domaine qui n'est pas partageable avec d'autres aspects (*c.-à-d.,* on ne peut pas avoir une variable déclarée appartenant à ce domaine dans d'autres aspects). Ces aspects `@spectator` sont similaires aux aspects spectateurs de Katz.

Cet article propose des annotations et des domaines de variable qui permettent aux programmeurs de définir à l'implémentation certains effets des aspects sur le programme de base et sur d'autres aspects. Ces effets sont ensuite garantis par transformation de programme et par un système de type. Les effets ne concernent que ceux des catégories spectateurs et inoffensifs. Ce travail suppose que les aspects ne contiennent pas d'alias, et comme pour les aspects inoffensifs, les propriétés préservées après le tissage des aspects ne sont pas étudiées.

1.4 Conclusion

Nous avons présenté dans ce chapitre les éléments de base de la programmation par aspect, les principaux domaines d'application et les travaux sur les fondements de ce paradigme. La programmation par aspect permet une meilleure décomposition des préoccupations d'une application en isolant chaque préoccupation transverse dans un aspect. Cela permet une meilleure

lisibilité, maintenabilité, réutilisabilité et évolutivité de l'application. Les aspects sont ajoutés automatiquement par transformation syntaxique de l'application de base pour donner l'application tissée. Pour que la décomposition proposée par la programmation par aspect soit bénéfique (*par ex.,* sans causer des comportements non souhaités suite à l'interaction entre plusieurs aspects), un des défis majeurs de celle-ci est de permettre de programmer et de raisonner sur l'application de base et les aspects sans étudier le code tissé. Les langages d'aspect dédiés apportent une solution à ce problème en se restreignant à un domaine d'application spécifique tout en minimisant par construction l'interaction entre les aspects et l'application de base (les aspects ne peuvent pas modifier les attributs ou appeler les méthodes de l'application de base). Cependant cette approche souffre d'un manque d'expressivité et d'une activité de recherche embryonnaire sur la composition des langages dédiés. Les langages d'aspect généraux permettent de définir différents types d'aspect. Mais leur expressivité (modification des attributs, appel et remplacement des méthodes du programme de base) rend difficile tout raisonnement sans étudier le code tissé. Des travaux de vérification à posteriori (Krishnamurthi et coll., Goldman et Katz, voir page 27) permettent néanmoins de raisonner sur la partie du code tissé correspondant à un aspect. Cependant ils dépendent d'un aspect et d'une propriété particulière. Ainsi à chaque modification de l'aspect ou de la propriété, il faut reprendre le processus de vérification ce qui minimise des avantages de la programmation par aspect (maintenabilité, réutilisabilité et évolutivité).

Cette thèse se situe dans la continuation des approches basées sur la classification des aspects. Ces approches proposent de regrouper les aspects ayant un comportement particulier dans une catégorie afin de déduire l'impact qu'un aspect a sur l'application de base en connaissant uniquement sa catégorie. Par exemple les aspects spectateurs de Katz ne modifient pas les variables et le flot de contrôle du programme de base. Tout aspect appartenant à cette catégorie préserve les propriétés de sûreté, de vivacité, les invariants et les propriétés d'existence du programme de base. A chaque modification d'un aspect, il suffit de connaître dans quelle catégorie se trouve le nouvel aspect pour connaître l'impact que celui-ci aura sur l'application de base. Ce raisonnement modulaire, ne restreint pas les aspects à un domaine spécifique. Cependant dans les approches actuelles, seul Katz identifie des catégories d'aspect tout en définissant les propriétés du programme de base qu'elles permettent de préserver. Cependant ce travail souffre d'imprécisions et d'ambiguïtés (voir Section 1.3.3, page 31). Par exemple, s'il nous semble vrai que les aspects spectateurs préservent les propriétés de sûreté sur les variables du programme de base (par définition des aspects spectateurs), cela n'est pas le cas lorsqu'il s'agit des événements. Par exemple, la propriété de sûreté "l'événement `diskformat` n'a jamais lieu" satisfait par un programme de base est violée par tout aspect spectateur qui insère la propriété `diskformat`. Notre approche qui a comme point de départ le travail de Katz vise à lever ses imprécisions en le formalisant. Ce faisant nous définissons des nouvelles catégories. Ainsi nous distinguons la catégorie des *observateurs* similaires aux spectateurs, des *adaptateurs* similaires aux régulateurs, des *faiblement intrusifs* similaires aux faiblement invasif, des *terminateurs* qui sont des observateurs qui modifie le flot de contrôle du programme en l'arrêtant, des *sélecteurs* qui sélectionnent des exécutions dans l'ensemble des exécutions possibles et des *verrouilleurs* qui conservent l'ensemble des états atteignables du programme de base. Contrairement à Katz, qui associe à chaque catégorie, des propriétés classiques et informelles (sûreté, vivacité, invariant et existence), nous définissons le sous ensemble de LTL préservé par chacune d'elle. Ces

sous ensembles sont ensuite généralisés pour les programmes non déterministes avec CTL*. Nos catégories sont reliées par une relation d'inclusion qui fait que leurs aspects peuvent interagir et se composer entre eux de façon déterministe quelque soit leur ordre. Cette démarche est indissociable du fait de déterminer la catégorie d'un aspect donné. Pour cela, Katz propose informellement des méthodes d'analyse (basées sur des analyses complexes comme l'analyse d'alias) a posteriori. Nous proposons, tout comme Dantas et Walker et les langages d'aspect dédiés, des langages d'aspect spécialisés (généraux ou spécifiques) qui garantissent par construction l'appartenance des aspects à une catégorie donnée.

Notre étude repose sur un cadre formel sémantique appelé CASB ("Common Aspect Semantics Base"). Ce cadre décrit dans une sémantique petit pas l'exécution du programme tissé indépendamment de tout langage d'aspect et de tout paradigme de programmation. L'approche ne nécessite pas une étape de traduction supplémentaire contrairement aux travaux actuels (voir Section 1.3.1, page 16). La présentation de ce cadre est l'objet du prochain chapitre. Nous l'utilisons ensuite pour définir formellement nos catégories d'aspect, les propriétés qu'elles préservent et leurs langages.

Chapitre 2

Cadre sémantique pour la programmation par aspect (CASB)

2.1 Introduction

Nous avons vu au chapitre précédent que les cadres sémantiques actuels utilisés pour décrire la programmation par aspect dépendent soit d'un langage particulier [CL06] ou d'un paradigme particulier (impératif, objet, ...) [WKD04, CLW03], soit sont indépendants de tout langage mais nécessitent une étape de traduction supplémentaire [BJJR04]. Dans cette thèse nous proposons une nouvelle approche qui est indépendante de tout langage et de tout paradigme sans nécessiter d'étape de traduction. Précisons qu'ici, par traduction, nous entendons le passage d'un langage d'aspect source à un langage dans lequel la sémantique est décrite (*cf.* Sections 1.3.1 et 1.3.1). Cette approche repose sur un cadre où un programme est vu comme une séquence d'instructions. Les actions sont insérées à un point de la séquence par un tissage dynamique séparé de l'exécution des instructions. Notre formalisation décrit l'exécution du programme de base et du programme tissé par une sémantique opérationnelle petit pas. Elle modélise précisément l'implémentation et peut être vue comme une machine abstraite ou un compilateur. Nous avons utilisé ce cadre, pour décrire précisément des mécanismes des langages généraux tels que AspectJ et CaesarJ (*cf.* Chapitre 2 et Chapitre 5) et pour décrire l'impact des aspects sur le programme de base sans raisonner sur le code tissé (*cf.* Chapitre 3 et 4).

Comme la majorité des travaux existants (voir Section 1.3.1, page 16), nous décrivons des aspects de type `before`, `after` et `around` à la AspectJ. Ces types d'aspect se retrouvent aussi dans les autres langages d'aspect généraux (*par ex.,* AspectC [CKFS01], CaesarJ [AGMO06], AML [DWWW08], *etc.*). Nous décrivons les aspects le plus indépendamment possible du langage de base. Pour chaque type d'aspect, nous introduisons les constructions et environnements nécessaires pour qu'il ait un sens. Par exemple les aspects de type `before` ne nécessitent pas de mécanisme particulier pour être décrit mais le langage de base doit respecter quelques hypothèses (voir Section 2.2). Les aspects `around`, ont besoin, comme dans AspectJ,

d'une instruction spéciale `proceed`. Mais contrairement aux aspects `around` d'AspectJ qui filtrent uniquement les appels de méthode, les nôtres filtrent tout type d'instruction.

La Section 2.2 présente les hypothèses sur le langage de base. Nous considérons le tissage d'un seul aspect (Section 2.3), puis, étendons le modèle avec le tissage de plusieurs aspects (Section 2.4). Enfin, nous décrivons le processus de filtrage (Section 2.5) et concluons (Section 2.6).

2.2 Hypothèses sur le langage de base

Le langage de base est décrit par une sémantique petit pas, formalisée par la relation binaire $\rightarrow_b$ sur des configurations (C, Σ) composées d'un programme C et d'un état Σ.

Un programme C est une séquence d'instructions de base i terminée par l'instruction vide $\bullet$:

$$C ::= i : C \mid \bullet$$

L'opérateur ":" est supposé associatif et, implicitement, les programmes sont supposés être de la forme $i_1 : (i_2 : \ldots : (i_n : \bullet)\ldots)$. Nous abuserons de la notation et écrirons, par exemple, $C_1 : C_2$ pour la concaténation de deux programmes.

L'état Σ est maintenu aussi abstrait que possible. Il peut contenir différents environnements (*par ex.,* associant des variables aux valeurs, des noms de procédures aux codes), des piles ou des tas (*par ex.,* mémoire allouée dynamiquement), *etc.*

Une étape de réduction du langage de base est de la forme :

$$(i : C, \Sigma) \rightarrow_b (C', \Sigma')$$

Intuitivement, i représente l'instruction courante du programme et C sa continuation. Le composant $i : C$ peut être vu comme la pile de flot de contrôle. L'opérateur ":" séquence l'exécution des instructions. Nous utiliserons cette propriété pour définir la sémantique des aspects `before` et `after`. Les configurations finales sont de la forme $(\epsilon : \bullet, \Sigma)$.

EXEMPLE **1.** *La sémantique du langage arithmétique*

$$E ::= k \mid E_1 + E_2$$

peut être décrit dans ce contexte de la manière suivante :

$$\begin{array}{lll} (E_1 + E_2 : C, S) & \rightarrow_b & (E_1 : E_2 : + : C, S) \\ (+ : C, k_1 : k_2 : S) & \rightarrow_b & (C, k_1 + k_2 : S) \\ (k : C, S) & \rightarrow_b & (C, k : S) \end{array}$$

L'état est composé d'une pile d'évaluation S. Evaluer une expression $E_1 + E_2$ consiste à évaluer E_1 et E_2 avant d'effectuer l'addition. Les trois instructions correspondants à ces taches sont placées dans la pile de flot de contrôle. L'évaluation d'un entier le place sur la pile d'évaluation. Une addition remplace les entiers au sommet de la pile d'évaluation par leur somme.

EXEMPLE **2.** *La sémantique du langage impératif*

$$S ::= S_1; S_2 \mid f = S \mid call\ f$$

peut être décrit dans ce cadre comme suit :

$$\begin{array}{lll} (S_1; S_2 : C, \rho) & \rightarrow_b & (S_1 : S_2 : C, \rho) \\ (f = S : C, \rho) & \rightarrow_b & (C, \rho[f \mapsto S]) \\ (call\ f : C, \rho) & \rightarrow_b & (C' : C, \rho) \qquad \textit{if } \rho(f) = C' \end{array}$$

L'état est composé d'un environnement (une fonction) ρ associant des identificateurs (f) à leur code $(\rho(f))$. Evaluer $S_1; S_2$ consiste à évaluer S_1 puis S_2 à son tour. Une déclaration de procédure $f = S$ place le code S dans l'environnement ρ ($\rho[f \mapsto S]$). Un appel à f place le code associé à f dans la pile de flot de contrôle.

Une optimisation pourrait être de compiler le langage en traduisant toute séquence ";" (opérateur de séquence dans le langage source) par ":" (opérateur sémantique représentant la séquence). La première règle de sémantique disparaîtrait.

La plupart des instructions s'exécute sans supprimer ou faire référence à leur continuation. Nous disons que de telles instructions *respectent la séquence*. Formellement :

DÉFINITION **3.** *Une instruction i* respecte la séquence *si*

$$(i : \bullet, \Sigma) \rightarrow_b (C' : \bullet, \Sigma') \Rightarrow (i : C, \Sigma) \rightarrow_b (C' : C, \Sigma')$$

Toutes les instructions des exemples ci-dessus respectent la séquence alors que les sauts ou les exceptions ne la respecteraient pas.

Il est parfois utile de sauvegarder certaines structures pendant l'exécution du programme. Ainsi, les programmes sont étendus avec la notion de bloc pour représenter les sous-programmes :

$$C ::= i : C \mid \{C_1\} : C_2 \mid \bullet$$

Avec cette extension, une instruction peut être un bloc d'instructions $\{i_1 : \ldots : i_n : \bullet\}$. La règle de réduction d'un bloc est

$$(\{C_1\} : C_2, \Sigma) \rightarrow_b (C_1 : C_2, \Sigma)$$

EXEMPLE **4.** *Si nous considérons encore le langage impératif ci-dessus alors la réduction du programme*

$$(f = call\ g; i_3); (g = i_1; i_2); call\ f$$

conduit à la configuration $(i_1 : i_2 : i_3 : \bullet, \rho)$ où il n'est pas possible de distinguer les continuations des appels à f et g. Avec les blocs, la règle des appels de fonction peut s'exprimer comme

$$(call\ f : C, \rho) \rightarrow_b (C' : \{C\}, \rho) \ \textit{ if } \rho(f) = C'$$

Ainsi, la configuration précédente devient $(i_1 : i_2 : \{i_3 : \bullet\}, \rho)$ qui représente clairement que $i_1 : i_2$ sont des instructions de la fonction courante et i_3 une adresse de retour.

2.3 Tissage d'un Aspect

Dans la suite, nous considérons que l'état Σ dans un programme tissé est composé des environnements suivants :

- Σ^b est le sous ensemble de Σ correspondant à l'état du programme de base (*c.-à-d.,* variables, environnements, *etc.*modifiés par les instructions du programme de base et les actions) ;
- Σ^a est le sous ensemble de Σ correspondant aux états des aspects (*c.-à-d.,* variables, environnements, *etc.*) qui ne peuvent pas être modifiés par le programme de base mais uniquement par les actions ;
- Σ^ψ est le sous ensemble de Σ qui représente les aspects. C'est une fonction qui décide si l'instruction courante doit être tissée. Lorsqu'une nouvelle instance d'aspect est créée, Σ^ψ et Σ^a sont modifiés.

Donc si (C, Σ) est une configuration d'un programme tissé alors $\Sigma = \Sigma^b \cup \Sigma^a \cup \Sigma^\psi$.

La sémantique du programme tissé est décrite par la relation binaire $\rightarrow$ définie par :

$$\textsc{Reduce} \quad \frac{(C, \Sigma) \rightarrow_b (C', \Sigma') \quad w(C', \Sigma') = (C'', \Sigma'')}{(C, \Sigma) \rightarrow (C'', \Sigma'')}$$

Réduire une configuration du programme tissé, c'est réduire sa première instruction par $\rightarrow_b$ et ensuite appliquer la fonction de tissage w à la configuration obtenue. La fonction w se définit par deux règles :

- Soit l'instruction courante n'est pas filtrée par les aspects (Σ^ψ retourne nil) et w ne modifie pas la configuration ;

$$\textsc{weave}0 \quad \frac{\Sigma^\psi(C, \Sigma) = nil}{w(C, \Sigma) = (C, \Sigma)}$$

- soit l'instruction courante est filtrée par les aspects et Σ^ψ retourne une nouvelle configuration (C', Σ') :

$$\textsc{weave}1 \quad \frac{\Sigma^\psi(C, \Sigma) = (C', \Sigma') \quad w(C', \Sigma') = (C'', \Sigma'')}{w(C, \Sigma) = (C'', \Sigma'')}$$

  où
  - C' est le nouveau code dans lequel les actions ont été tissées avant, après ou autour de l'instruction courante de C ;
  - $\Sigma' = \Sigma^b \cup \Sigma'^a \cup \Sigma'^\psi$, avec Σ'^ψ contenant une nouvelle instance d'aspect et Σ'^a contenant l'état de celle-ci.

Notons que w s'applique récursivement sur du code des actions insérées par Σ^ψ jusqu'à ce que l'instruction courante ne soit pas filtrée. Nous supposons ainsi que le tissage ne dépend que de l'instruction courante et pas de la continuation. Pour éviter un tissage infini, nous introduisons la notion d'instruction taguée notée $\bar{i}$. Une instruction taguée $\bar{i}$ a une sémantique identique

à celle de i excepté le fait qu'elle ne puisse être tissée.

$$\text{TAGGED} \quad \frac{(i : C, \Sigma) \rightarrow_b (C', \Sigma')}{(\bar{i} : C, \Sigma) \rightarrow (C', \Sigma')}$$

$$\text{et } \Sigma^{\psi}(\bar{i} : C, \Sigma) = nil$$

Si des langages interdisent le tissage des actions (*par ex.,* les premières versions d'AspectJ [DFS02]), cela peut se traduire dans notre cadre par le taggage de toutes les instructions des actions. Dans AspectJ par exemple, la fonction w peut être vue comme le compilateur qui instrumente le code avec les actions.

Une réduction du programme tissé, commençant toujours par une réduction de $\rightarrow_b$, il est impossible de tisser la première instruction du programme de base. Cependant, il peut être utile d'exécuter une action avant la première instruction du programme de base. Pour permettre cela, nous introduisons une instruction spéciale $start$ et supposons que les configurations initiales sont de la forme $(start : C, \Sigma)$. La sémantique de $start$ est la même que celle de *nop* (l'instruction vide) :

$$(start : C, \Sigma) \rightarrow_b (C, \Sigma)$$

Ainsi un programme de base commence toujours par la réduction

$$(start : C_0, \Sigma_0) \rightarrow_b (C_0, \Sigma_0)$$

tandis qu'un programme tissé par

$$(start : C_0, \Sigma_0) \rightarrow (C'_0, \Sigma'_0) \qquad \text{avec } w(C_0, \Sigma_0) = (C'_0, \Sigma'_0)$$

et il est possible d'insérer une action après $start$ et juste avant la véritable première instruction.

Nous allons maintenant présenter les règles de sémantique des aspects `before`, `after` et `around`. La fonction Σ^{ψ} décide en deux étapes si un point de jonction est tissé ou pas. La première filtre uniquement les informations statiques tandis que la deuxième teste les informations dynamiques (le point de coupure dynamique `if`). Pour cela, Σ^{ψ} prend la configuration courante en paramètre et retourne une configuration dans laquelle a été insérée une fonction ϕ qui prend Σ en paramètre et retourne une action a. Nous supposons ici que le langage de a est celui du programme de base. Si aucune action ne filtre pas Σ (*c.-à-d.,* le prédicat du point de coupure `if` est faux), ϕ retourne ϵ dans le cas des aspects `before` et `after`, et `proceed` dans celui des aspects `around`.

2.3.1 Aspect `before`

Lorsqu'un aspect `before` Σ^{ψ} filtre une instruction courante, il la tague et insère l'instruction $test\ \phi$ avant celle-ci.

$$\Sigma^{\psi}(i : C, \Sigma) \ = \ (test\ \phi : \bar{i} : C, \Sigma)$$

L'instruction $test\ \phi$ est une instruction sémantique qui ne peut être filtrée. Lorsque $test\ \phi$ est l'instruction courante, la règle ADVICE applique la fonction ϕ à l'état Σ. Cette fonction retourne l'action à insérer a ou l'action vide ϵ.

$$\text{ADVICE}\quad \frac{}{(test\ \phi : C, \Sigma) \rightarrow (\phi(\Sigma) : C, \Sigma)}$$

Notons que les instructions des actions ($\phi(\Sigma)$), n'étant pas taguées, peuvent être filtrées.

EXEMPLE **5.** *Considérons le langage impératif de l'Exemple 2 (page 43) et l'aspect Σ^{ψ} qui insère un appel à bar avant chaque appel à foo.*

$$\forall (C, \Sigma).\ \ \Sigma^{\psi}(call\ foo : C, \Sigma) = (test\ \phi : \overline{call\ foo} : C, \Sigma)$$

$$\textit{avec}\ \phi(\Sigma) = call\ bar$$

La réduction de $(start : call\ foo : \bullet, \Sigma')$ où Σ' est l'état initial se déroule comme suit :

$$\begin{array}{rll} (start : call\ foo : \bullet, \Sigma') \rightarrow & (test\ \phi : \overline{call\ foo} : \bullet, & \Sigma') \\ \rightarrow & (call\ bar : \overline{call\ foo} : \bullet, & \Sigma') \\ \rightarrow^* & (\overline{call\ foo} : \bullet, & \Sigma'') \\ \rightarrow^* & (\epsilon : \bullet, & \Sigma''') \end{array}$$

Après le tissage (insertion de ϕ), la fonction foo étant taguée, elle ne peut plus être filtrée. L'instruction $test\ \phi$ tissée avant foo va insérer un appel à bar avant l'appel à foo.

2.3.2 Aspect `after`

Intuitivement, l'action d'un aspect `after` s'exécute après la fin de l'instruction qu'il filtre. Par conséquent, il ne peut s'appliquer qu'aux instructions qui respectent la séquence. S'il filtre l'instruction courante, un aspect `after` Σ^{ψ}, la tague et insère l'instruction $test\ \phi$ après celle-ci.

$$\Sigma^{\psi}(i : C, \Sigma)\ =\ (\bar{i} : test\ \phi : C, \Sigma)$$

L'instruction taguée $\bar{i}$ ne peut plus être filtrée. Nous évitons ainsi un tissage infini de i par Σ^{ψ} et la prochaine étape de réduction est celle de i par $\rightarrow_b$. Si l'instruction ne respecte pas la séquence (*par ex.,* elle peut lever une exception et supprimer sa continuation), l'action pourrait ne pas s'exécuter. Si c'est un appel de fonction ou de procédure, l'action est exécutée au retour de la fonction.

2.3.3 Aspect `around`

Pour les aspects `around`, le langage de base est étendu avec l'instruction `proceed` qui peut être utilisée dans le code de l'action.

L'exécution d'un aspect `around` commence par l'exécution de son action avant l'instruction filtrée, continue en exécutant l'instruction filtrée par l'intermédiaire de `proceed` et termine par la fin de l'action. L'action peut terminer son exécution sans exécuter l'instruction filtrée (elle n'utilise pas l'instruction `proceed`). Dans ce cas, l'action remplace complètement l'instruction. Comme indiqué ci-dessus, si la partie dynamique du point de coupure ne filtre pas l'état courant, l'action n'est pas tissée et l'instruction courante est exécutée. Cela est formalisé par la fonction ϕ qui retourne `proceed` (et non pas ϵ) lorsqu'elle ne correspond pas à Σ. Une implémentation possible de ce mécanisme est de transformer tout test dynamique dans les points de coupure (*par ex.,* `if` $x = 0$) en expression conditionnelle dans l'action (*par ex.,* `if` $x = 0$ `then` *action* `else proceed`).

En général, un aspect `around` peut contenir plusieurs `proceed` entraînant plusieurs exécutions de l'instruction filtrée. De plus, les instructions d'une action `around` peuvent être filtrées par un autre aspect `around` et par conséquent il faut garder un lien entre chaque `proceed` et l'instruction filtrée à laquelle elle correspond. Pour décrire le comportement d'un aspect `around`, nous introduisons les éléments sémantiques suivants :

- une pile Σ^P incluse dans Σ^a et appelée la pile proceed ;
- la fonction sémantique $push_p\ i$ qui place l'instruction i au sommet de la pile proceed ;
- la fonction sémantique pop_p qui retire l'instruction qui se trouve au sommet de Σ^P.

Un aspect `around` Σ^ψ, insère l'instruction $test\ \phi$ suivie de pop_p et place l'instruction filtrée sur la pile procced et ainsi, elle pourra être exécutée par une instruction `proceed`.

L'instruction `proceed` (Règle PROCEED) exécute l'instruction i placée au sommet de la pile proceed. Cette instruction est retirée de Σ^P, parce qu'elle peut appartenir à un autre aspect, dont le proceed doit exécuter l'instruction qui est au sommet de Σ^P et pas i. L'instruction $push_p\ i$ permet ensuite de replacer i au sommet de la pile au cas où l'action aurait plusieurs `proceed`.

La règle POP termine l'exécution de l'action courant en retirant l'instruction qui se trouve au sommet de la pile proceed.

$$\Sigma^\psi(i : C, X \cup \Sigma^P) \;=\; (test\ \phi : pop_p : C, X \cup \bar{i} : \Sigma^P)$$

$$\textsc{Proceed}\quad \frac{\Sigma^P = i : \Sigma'^P}{(\texttt{proceed} : C, X \cup \Sigma^P) \rightarrow (i : push_p\ i : C, X \cup \Sigma'^P)}$$

$$\textsc{Pop}\quad \frac{\Sigma^P = i : \Sigma'^P}{(pop_p : C, X \cup \Sigma^P) \rightarrow (C, X \cup \Sigma'^P)}$$

L'état $X \cup \Sigma^P$ représente l'état global où Σ^P est la pile proceed et X comprend Σ^b, Σ^ψ et le reste de Σ^a.

EXEMPLE **6.** *Considérons le langage impératif de l'Exemple 2 (page 43) et l'aspect Σ^ψ qui*

insère un appel à bar *(resp.* baz*) avant (resp. après) chaque appel à* foo *:*

$$\forall(C, \Sigma = X \cup \Sigma^P).\ \Sigma^\psi(call\ foo : C, X \cup \Sigma^P) = (test\ \phi : pop_p : C, X \cup \overline{call\ foo} : \Sigma^P)$$

$$\phi(\Sigma) = call\ bar; \texttt{proceed}; call\ baz$$

La réduction de $(start : call\ foo : \bullet, X \cup \epsilon)$ *où* ϵ *est la pile proceed vide, est :*

$$\begin{array}{ll}
& (start : call\ foo : \bullet, X \cup \epsilon) \\
\rightarrow & (test\ \phi : pop_p : \bullet, & X \cup \overline{call\ foo} : \epsilon) \\
\rightarrow & (call\ bar; \texttt{proceed}; call\ baz : pop_p : \bullet, & X \cup \overline{call\ foo} : \epsilon) \\
\rightarrow^* & (\texttt{proceed} : call\ baz : pop_p : \bullet, & X' \cup \overline{call\ foo} : \epsilon) \\
\rightarrow & (\overline{call\ foo} : push_p(\overline{call\ foo}) : call\ baz : pop_p : \bullet, & X' \cup \epsilon) \\
\rightarrow^* & (push_p(\overline{call\ foo}) : call\ baz : pop_p : \bullet, X'' \cup \epsilon) & \\
\rightarrow^* & (call\ baz : pop_p : \bullet, & X'' \cup \overline{call\ foo} : \epsilon) \\
\rightarrow^* & (pop_p : \bullet, & X''' \cup \overline{call\ foo} : \epsilon) \\
\rightarrow & (\epsilon : \bullet, & X''' \cup \epsilon)
\end{array}$$

Lorsque l'appel à foo *est filtré par l'aspect* Σ^ψ*, il est tagué et placé au sommet de la pile proceed. Puis, l'instruction* $test\ \phi$ *suivie de* pop_p *le remplace au sommet de la continuation.* $test\ \phi$ *se réduit en une séquence d'instructions qui correspond au code de l'action de* Σ^ψ*. Ces instructions exécutent un appel à* bar *suivi de* `proceed` *et un appel à* baz*. A l'exécution de* `proceed`*, l'instruction au sommet de la pile proceed* c.-à-d.*, l'appel à* foo *est exécuté. A la fin de l'exécution de l'action, l'instruction* pop_p *vide la pile proceed.*

Les règles ci-dessus supposent qu'un aspect `around` peut être filtré par un autre aspect `around`. Cela est aussi le cas dans AspectJ. Dans les langages où cela n'est pas possible, on ne peut jamais avoir pendant l'exécution d'une action des proceed appartenant à des aspects différents et la règle PROCEED doit être modifiée de telle sorte que le sommet de la pile proceed ne soit pas retiré et replacé à chaque proceed. Notre cadre est plus général que celui d'AspectJ. Par exemple, nos règles, permettent de filtrer tout type d'instruction, ce qui n'est pas le cas d'AspectJ qui lui, ne peut filtrer que les accès en écriture ou en lecture des variables, des appels de méthode et la capture des exceptions.

2.4 Tissage de plusieurs aspects

Nous considérons maintenant le tissage de plusieurs aspects au même point de jonction. Dans un premier temps, nous allons présenter le tissage de plusieurs aspects de même type (*c.-à-d.,* `before`, `after` ou `around`). Ensuite, nous présenterons le tissage de plusieurs aspects de types différents.

2.4.1 Aspects de même type

Plusieurs aspects de même type sont représentés par l'aspect Σ^{ψ} qui insère plusieurs fonctions $\phi_1 \dots \phi_n$ représentant les actions des aspects. L'ordre d'exécution des actions est celui de l'insertion des fonctions qui est déterminé par Σ^{ψ}.

Aspects `before`

Lorsque n aspects `before` filtrent une instruction i, leurs actions représentées par $\phi_1 \dots \phi_n$ sont exécutées avant la réduction de i. Comme dans le cas d'un seul aspect, Σ^{ψ} tague i pour l'empêcher d'être à nouveau filtrée et insère les fonctions correspondant aux actions.

$$\Sigma^{\psi}(i : C, \Sigma) = (test\ \phi_1 : \ldots : test\ \phi_n : \bar{i} : C, \Sigma)$$

Aspects `after`

Lorsque n aspects `after` filtrent une instruction i, leurs actions représentées par $\phi_1 \dots \phi_n$ sont exécutées après la réduction de i. L'aspect Σ^{ψ} insère alors les fonctions des actions après l'instruction taguée $\bar{i}$.

$$\Sigma^{\psi}(i : C, \Sigma) = (\bar{i} : test\ \phi_1 : \ldots : test\ \phi_n : C, \Sigma)$$

Aspects `around`

Lorsqu'une instruction est filtrée par plusieurs aspects `around`, Σ^{ψ} insère la fonction de l'action du premier aspect au sommet du code à exécuter et place les fonctions des autres actions ainsi que l'instruction filtrée sur la pile proceed. Comme dans le cas d'un seul aspect, une action peut avoir 0, 1 ou plusieurs `proceed`. Si n actions $(a_1, \ldots, a_n)$ s'appliquent autour d'une instruction i et si chaque action est de la forme a'_i : `proceed` : a''_i alors l'exécution des actions sera de la forme

$$a'_1 \rightarrow a'_2 \rightarrow \ldots \rightarrow a'_n \rightarrow i \rightarrow a''_n \rightarrow \ldots \rightarrow a''_2 \rightarrow a''_1$$

Dans le cas où certaines actions ont plusieurs ou zéro `proceed`, l'exécution est moins intuitive. Ainsi, si nous remplaçons a_1 par a'_1 : `proceed` : a''_1 : `proceed` : a'''_1 l'exécution sera

$$a'_1 \rightarrow \ldots \rightarrow a'_n \rightarrow i \rightarrow a''_n \ldots a''_2 \rightarrow a''_1 \rightarrow a'_2 \ldots \rightarrow a'_n \rightarrow i \rightarrow a''_n \ldots a''_2 \rightarrow a'''_1$$

qui devient

$$a'_1 \rightarrow a_2 \rightarrow a''_1 \rightarrow a_2 \rightarrow a'''_1$$

si a_2 n'a pas de `proceed`.

Donc Σ^{ψ} insère la fonction de la première action au sommet du code à exécuter ; cette fonction est suivie de $pop_p\ n$ qui va retirer les fonctions des autres actions et l'instruction filtrée de la pile proceed à la fin de l'exécution des aspects. La règle PROCEED* exécute la prochaine action *c.-à-d.,* l'instruction placée au sommet de la pile proceed. Comme dans le cas d'un seul aspect, si l'instruction à exécuter est une action, il pourrait exécuter l'instruction qui est sommet de la pile proceed par l'intermédiaire de l'instruction `proceed`. Une fois cette instruction exécutée, elle est replacée dans la pile proceed car elle pourrait être ré-exécutée par une autre occurrence de la même action `proceed`. L'instruction pop_p (Règle POP*) termine l'exécution des actions en retirant de la pile proceed les n premiers éléments correspondant aux fonctions des actions et l'instruction filtrée.

$$\Sigma^{\psi}(i : C, X \cup \Sigma^P) = (test\ \phi_1 : pop_p\ n : C, X \cup test\ \phi_2 : \ldots : test\ \phi_n : \bar{i} : \Sigma^P)$$

$$\textsc{Proceed*}\ \frac{\Sigma^P = x : \Sigma'^P}{(\texttt{proceed} : C, X \cup \Sigma^P) \rightarrow (x : push_p\ x : C, X \cup \Sigma'^P)}$$

$$\textsc{Pop*}\ \frac{\Sigma^P = x_1 : \ldots : x_n : \Sigma'^P}{(pop_p\ n : C, X \cup \Sigma^P) \rightarrow (C, X \cup \Sigma'^P)}$$

EXEMPLE **7.** *Considérons encore le langage impératif de l'Exemple 2 (page 43) et Σ^{ψ} regroupant deux aspects dont les actions sont représentées par ϕ_1 et ϕ_2. Les aspects filtrent chaque appel à foo. Ces aspects sont ordonnés de manière à exécuter ϕ_1 avant ϕ_2. L'action ϕ_1 insère un appel à bar, puis, passe la main à ϕ_2 (`proceed` de ϕ_1). L'action ϕ_2 exécute l'appel à foo par l'intermédiaire de son `proceed` avant de repasser la main à ϕ_1 qui termine son exécution avec l'appel à baz.*

$$\forall(C, \Sigma = X \cup \Sigma^P).\ \Sigma^{\psi}(call\ foo : C, X \cup \Sigma^P) = (test\ \phi_1 : pop_p\ 2 : C, X \cup test\ \phi_2 : \overline{call\ foo} : \Sigma^P)$$

$$\begin{array}{rcl} \phi_1(\Sigma) & = & call\ bar; \texttt{proceed}; call\ baz \\ \phi_2(\Sigma) & = & \texttt{proceed} \end{array}$$

La réduction de $(start : call\ foo : \bullet, X \cup \epsilon)$ où ϵ est la pile proceed vide, se déroule comme suit :

$$\begin{array}{ll}
(start : call\ foo : \bullet, X \cup \epsilon) & \\
\rightarrow\ (test\ \phi_1 : pop_p\ 2 : \bullet, & X \cup test\ \phi_2 : \overline{call\ foo} : \epsilon) \\
\rightarrow\ (call\ bar; \texttt{proceed}; call\ baz : pop_p\ 2 : \bullet, & X \cup test\ \phi_2 : \overline{call\ foo} : \epsilon) \\
\rightarrow^*\ (\texttt{proceed} : call\ baz : pop_p\ 2 : \bullet, & X' \cup test\ \phi_2 : \overline{call\ foo} : \epsilon) \\
\rightarrow\ (test\ \phi_2 : push_p\ test\ \phi_2 : call\ baz : pop_p\ 2 : \bullet, & X' \cup \overline{call\ foo} : \epsilon) \\
\rightarrow\ (\texttt{proceed} : push_p\ test\ \phi_2 : call\ baz : pop_p\ 2 : \bullet, & X' \cup \overline{call\ foo} : \epsilon) \\
\rightarrow\ (\overline{call\ foo} : push_p\ \overline{call\ foo} : push_p\ test\ \phi_2 : call\ baz : pop_p\ 2 : \bullet, & X' \cup \epsilon) \\
\rightarrow^*\ (push_p\ \overline{call\ foo} : push_p\ test\ \phi_2 : call\ baz : pop_p\ 2 : \bullet, & X'' \cup \epsilon) \\
\rightarrow^*\ (push_p\ test\ \phi_2 : call\ baz : pop_p\ 2 : \bullet, & X'' \cup \overline{call\ foo} : \epsilon) \\
\rightarrow^*\ (call\ baz : pop_p\ 2 : \bullet, & X'' \cup test\ \phi_2 : \overline{call\ foo} : \epsilon) \\
\rightarrow^*\ (pop_p\ 2 : \bullet, & X''' \cup test\ \phi_2 : \overline{call\ foo} : \epsilon) \\
\rightarrow\ (\epsilon : \bullet, & X''' \cup \epsilon)
\end{array}$$

Lorsque l'appel à foo est filtré par Σ^ψ, il est remplacé au sommet de la continuation par l'instruction $test\ \phi_1$ suivie de $pop_p\ 2$ où 2 est le nombre d'éléments insérés dans la pile proceed. Ces éléments sont $test\ \phi_2$ et l'appel à foo. L'exécution de $test\ \phi_1$ permet d'exécuter l'action ϕ_1 qui commence par l'appel à bar. A l'exécution de `proceed`*, l'instruction qui est au sommet de la pile proceed* c.-à-d., *$test\ \phi_2$ est exécutée. Cette exécution permet d'exécuter l'action ϕ_2. Celle-ci est une instruction* `proceed` *qui va exécuter l'appel à foo avant de passer la main à la suite de ϕ_1 (l'appel à baz). L'instruction $pop_p\ 2$ termine l'exécution des actions en retirant $test\ \phi_2$ et $\overline{call\ foo}$ de la pile proceed.*

2.4.2 Aspects `before`, `after` et `around`

Nous traitons le cas où plusieurs aspects différents `before`, `after` et `around` filtrent un même point de jonction par une transformation des aspects `before` et `after` en `around`. Les actions de ces aspects sont transformées en des actions d'aspect de type `around` en utilisant la fonction *toaround* définie comme suit :

$$\textit{toaround}(\phi_{\texttt{before}}) = \lambda\Sigma.(test\ \phi_{\texttt{before}} : \texttt{proceed})$$

$$\textit{toaround}(\phi_{\texttt{after}}) = \lambda\Sigma.(\texttt{proceed} : test\ \phi_{\texttt{after}})$$

L'action d'un aspect `before` est transformée en une action `around` en insérant l'action `before` (*c.-à-d.,* $test\ \phi_{\texttt{before}}$) suivie de `proceed` qui exécutera le prochain aspect ou l'instruction filtrée. De même, l'action `after` est transformée en `around` qui exécutera l'action `after` après l'exécution du prochain aspect ou de l'instruction filtrée. La fonction ϕ, prenant en paramètre un état, notre transformation commence par $\lambda\Sigma$. Ainsi, $test\ \phi_{\textit{before}}$ utilise l'état courant au moment de la transformation tandis que $test\ \phi_{\textit{after}}$ utilise l'état obtenu après l'exécution des autres aspects ou de l'instruction filtrée (*c.-à-d.,* `proceed`). La fonction Σ^ψ a donc la même définition que dans le cas de plusieurs aspects de type `around` ; sauf qu'ici, elle appelle préalablement la fonction *toaround*.

$$\Sigma^\psi(i : C, X \cup \Sigma^P) = (test\ \phi'_1 : pop_p\ n : C, X \cup test\ \phi'_2 : \ldots : test\ \phi'_n : \bar{i} : \Sigma^P)$$
avec $\phi'_1 = \textit{toaround}(\phi_1) \ldots \phi'_n = \textit{toaround}(\phi_n)$

2.5 Points de coupure

Jusqu'à présent, nous avons représenté les aspects par la fonction abstraite Σ^ψ. Dans cette section, nous précisons une partie de la structure de cette fonction en présentant les points de coupure.

En représentant les points de coupure par des motifs P, la fonction Σ^{ψ} est de la forme :

$$\begin{array}{ll} \Sigma^{\psi}(i : C, \Sigma) = & \textit{if } \textit{match}(P, i) \textit{ then } (C', \Sigma') \textit{ else } nil \\ & \text{avec } C' = \textit{toweave}(\sigma(\phi), i : C) \\ & \text{avec } \sigma \text{ la substitution telle que } \sigma(P) = i \end{array}$$

Un aspect est tissé, si l'instruction courante est filtrée par le motif de l'aspect P. Ce processus de filtrage est représenté par la fonction $match$ qui prend un motif et une instruction en paramètre et retourne un booléen.

$$\textit{match} : P \times Instruction \rightarrow \textit{bool}$$

où $Instruction$ représente l'ensemble des instructions. Dans le cas où l'instruction est filtrée, une nouvelle configuration (C', Σ') est retournée. La continuation C' est le nouveau code dans lequel l'action $\sigma(\phi)$ est tissée avant, après ou autour de i. La fonction *toweave* (qui n'est pas définie ici) place l'action à la bonne place dans la continuation suivant le type de l'aspect (`before`, `after` et `around`. *cf.* Sections 2.3 et 2.4). L'état Σ' est le nouvel état qui peut contenir de nouvelles instances d'aspects. Certaines informations telles que des noms, des types, des valeurs, *etc.* peuvent être passées d'une instruction à l'action par l'intermédiaire des variables de motif auxquelles des valeurs sont associées par la substitution σ telle que $\sigma(P) = i$.

Un motif standard peut être un terme ou une conjonction, disjonction et négation de motifs. La grammaire de P est définie par :

$$P ::= T_i \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid \neg P$$

Un terme T_i a la même syntaxe qu'une instruction mais peut comporter des variables de motif. Par exemple, le motif $call\ x$ filtre tout appel de fonction en associant à la variable x le nom de la fonction. La fonction *match* est définie comme suit :

$$\begin{array}{lcll} \textit{match}(T_i, i) & = & \textit{true} & \text{si } \exists\ \sigma \text{ tel que } \sigma(T_i) = i \\ & = & \textit{false} & \text{sinon} \\ \textit{match}(P_1 \wedge P_2, i) & = & \textit{match}(P_1, i) \wedge \textit{match}(P_2, i) & \\ \textit{match}(P_1 \vee P_2, i) & = & \textit{match}(P_1, i) \vee \textit{match}(P_2, i) & \\ \textit{match}(\neg P, i) & = & \neg \textit{match}(P, i) & \end{array}$$

La négation peut entraîner des complications car un motif avec une négation peut filtrer une instruction selon plusieurs substitutions. Par exemple, le point de coupure $\neg(call\ x)$, filtre toutes instructions différentes de $call$. Ce motif filtre par exemple une affectation avec une infinité de valeurs possible pour x. De même $\neg(x := 2)$ filtre toutes instructions différentes de $x := 2$. Il filtre par exemple un appel de méthode avec également une infinité de valeurs possible pour x. Dans ce contexte, nous supposons qu'une variable de motif utilisée dans l'action est associée à une unique valeur par le motif.

2.6 Conclusion

Nous avons présenté un cadre formel qui permet de décrire tout programme à aspect indépendamment d'un langage de programmation et sans nécessiter d'étape de traduction. Notre

cadre repose sur une sémantique opérationnelle petit pas qui décrit des aspects inspirés d'AspectJ tout en étant plus général (*par ex.,* les aspects `around` décrits filtrent n'importe quel type d'instruction). Comme présenté au Chapitre 1 (voir Section 1.3.1), Jagadeesan et coll. [BJJR04] proposent un calcul pour la description des programmes à aspect et qui est aussi indépendant d'un langage de programmation. Ce calcul utilise un langage basé sur l'envoi de messages pour décrire l'exécution des aspects. Cette démarche implique une traduction de tout programme de base et aspects vers le langage du calcul. De plus contrairement à notre cadre et aux langages d'aspect existants (AspectJ [KHH+01a], CaesarJ [AGMO06], AspectC [CKFS01], *etc.*), ce calcul ne permet pas de filtrer les instructions des actions. La sémantique que propose Clifton et coll. [CLW03] permet de décrire plusieurs langages d'aspect mais ils dépendent du paradigme objet. De plus, les points de coupure de ces langages doivent être traduits vers un langage de point de coupure général qui permet de filtrer des points de jonction très proches de ceux d'AspectJ. Clifton et Leavens [CL06] proposent une sémantique d'un sous ensemble d'AspectJ. Cette sémantique décrit uniquement des aspects de type `around` et seuls les appels et exécutions de méthodes sont filtrés par les aspects. Wand et coll. [WKD04] proposent une sémantique d'un langage qui possède les concepts du sous ensemble impératif d'AspectJ basé sur du Scheme. Ce langage sélectionne uniquement les appels et exécutions de méthodes, et les exécutions des aspects. Cette sémantique nous semble complexe et difficilement utilisable.

L'originalité de notre approche réside dans le fait que les aspects sont décrits indépendamment d'un langage de base précis. Seules les constructions nécessaires du langage de base sont introduites (*par ex.,* la séquence pour les aspects `before` et `after`). Une autre originalité est la séparation du processus de filtrage en deux étapes : un filtrage statique qui permet de sélectionner l'instruction courante et un filtrage dynamique (*par ex.,* correspondant au prédicat du point de coupure `if(...)` d'AspectJ). Les travaux précédents (voir Section 1.3.1) utilisent un tissage similaire à notre filtrage statique *c.-à-d.,* pour chaque point de jonction, l'ensemble des aspects qui le filtre est tissé. Ils ne formalisent pas le filtrage dynamique qui rend difficile à anticiper la sémantique des programmes de style AspectJ. En effet, comme présenté à l'exemple de la Section 1.3.1, page 16, des aspects peuvent être activés pendant l'exécution des actions suite par exemple à une modification de variable qui rend vrai un prédicat d'un point de coupure `if(...)` qui ne l'était pas lors du filtrage statique.

Ce cadre est utilisé dans le Chapitre 3 pour raisonner sur le programme de base et les aspects sans regarder le code tissé. Dans le Chapitre 4, le cadre formel est utilisé pour concevoir des langages d'aspect qui assurent par construction que les aspects appartiennent à une catégorie déterminée d'aspects. Enfin, dans le Chapitre 5, nous illustrons l'expressivité de ce cadre, en l'utilisant dans un premier temps pour décrire de façon séparée des constructions importantes (des points de coupure, les exceptions et les instances d'aspect) des langages d'aspect différents, et dans un deuxième temps pour montrer la correction d'une transformation d'aspect.

Chapitre 3

Aspects et préservation de propriétés

3.1 Introduction

Si la programmation par aspect permet de coder de façon modulaire des traitements qui sont entrelacés dans le programme, elle peut en général, bouleverser la sémantique du programme de base. Le programmeur peut être amené à analyser le programme tissé pour comprendre sa sémantique. Cela est contraire à l'objectif de séparation des préoccupations. Afin de contrôler l'impact du tissage, nous proposons, plusieurs catégories d'aspects qui modifient la sémantique du programme de base de manière contrôlée. Pour chaque catégorie d'aspect $\mathcal{A}_x$, nous identifions une classe de propriété φ^x qui lui correspond et qui est préservée par le tissage. Plus précisément, soit P un programme qui satisfait n'importe quelle propriété $\varphi \in \varphi^x$, alors tisser n'importe quel aspect $A \in \mathcal{A}_x$ dans P produit un programme qui continue de satisfaire φ. Nos catégories d'aspects, inspirées des travaux de Shmuel Katz [Kat06], comprennent les observateurs, les terminateurs, les verrouilleurs et les aspects faiblement intrusifs :

- les observateurs ne modifient pas l'état et le flot de contrôle du programme de base. Dans cette catégorie, l'action peut modifier uniquement les variables locales de l'aspect ;
- les terminateurs sont des observateurs qui peuvent aussi arrêter l'exécution du programme de base. L'état du programme n'est pas modifié mais son flot de contrôle l'est car l'aspect peut mettre un terme à celui-ci ;
- les verrouilleurs peuvent modifier l'état et le flot de contrôle du programme de base mais ils assurent que tout état du programme tissé reste dans l'ensemble des états atteignables par le programme de base ;
- les aspects faiblement intrusifs peuvent modifier les états et le flot de contrôle sans aucune restriction lors de l'exécution des actions. Cependant, les états manipulés par les instructions du programme de base restent dans l'ensemble des états atteignables du programme de base.

Typiquement, les aspects de déboguage, de persistance, de profilage, de traçage et de journalisation sont des aspects observateurs. Les aspects assurant des propriétés de sûreté comme les aspects de sécurité sont des terminateurs. Les aspects de tolérance aux fautes qui retournent

vers des états passés de façon atomique sont des verrouilleurs. Ceux qui retournent vers des états passés de manière non atomique sont des faiblement intrusifs.

Les aspects observateurs insèrent des actions qui modifieront uniquement leurs propres variables locales. Intuitivement, ils devraient donc préserver beaucoup de propriétés mais il faut être prudent. Par exemple, des propriétés impliquant l'absence d'événements indésirables (*c.-à-d.,* des appels de procédures spécifiques) ne sont pas préservées en général car l'aspect insère des événements. Les propriétés de vivacité peuvent aussi être violées si l'action ne termine pas. Aussi, nous devons assurer que les programmes de base ne sont pas réflexifs car dans ce cas une action n'ayant aucun impact direct sur le programme de base pourrait modifier indirectement son flot de contrôle. En effet, un programme de base qui effectue un calcul précis en fonction du nombre d'instructions de sa continuation pourrait retourner un resultat différent si un aspect est tissé après celui-ci. Ces exemples montrent que ces catégories doivent être définies et étudiées de façon formelle.

Nous définissons ces catégories en utilisant le cadre du Chapitre 2. Les classes de propriétés sont définies comme des sous ensembles de LTL [MP92] pour les programmes déterministes et CTL* [CES83] pour les programmes non déterministes. L'objectif est de prouver formellement que, pour tout programme, le tissage de tout aspect d'une catégorie préserve toute propriété appartenant à la classe qui lui est reliée.

Ce chapitre dont les contributions ont été publiées à la conférence internationale PEPM 08 [DDDF08a], considère d'abord les programmes déterministes. La Section 3.2 présente la définition formelle des traces d'exécution du programme de base et tissé ainsi que les fonctions auxilliaires avec lesquelles nous raisonnons sur ces traces. La Section 3.3 rappelle la syntaxe et la sémantique de LTL. Nous définissons les catégories d'aspect et les classes de propriétés de LTL correspondant à chaque catégorie : les observateurs (Section 3.4.1), les terminateurs (Section 3.4.2), les verrouilleurs (Section 3.4.3) et les faiblement intrusifs (Section 3.4.4). La Section 3.5 présente le cas non déterministe de cette étude. Elle présente la sémantique des programmes non déterministes et la logique temporelle CTL*, puis, deux nouvelles catégories d'aspect (les sélecteurs et les adaptateurs) et les classes de propriétés de CTL* correspondant à chacune d'elle. La Section 3.6 présente les compositions et interactions d'aspects de nos différentes catégorie. La Section 3.7 conclut. L'appendice A.1 présente la preuve de préservation des propriétés correspondant aux observateurs. Les preuve pour les autres catégories ont la même structure que celle des observateurs.

3.2 Programme de base et programme tissé

Pour prouver que des propriétés sont préservées par le tissage, nous devons définir la sémantique du programme de base et du programme tissé. Nous le faisons en utilisant le cadre défini au Chapitre 2. Dans la séquence de code C du programme tissé, nous notons i_b une instruction du programme de base et i_a une instruction de l'action. L'instruction ϵ, qui représente

la dernière instruction du programme, est considérée comme une instruction du programme de base. Ainsi, pour une configuration (C, Σ) du programme tissé avec $\Sigma = \Sigma^b \cup \Sigma^a \cup \Sigma^\psi$, une réduction du programme tissé a deux propriétés. Premièrement, la réduction d'une instruction du programme de base modifie uniquement l'état du programme de base. Formellement :

$$\forall (C, \Sigma).(i_b : C, \Sigma) \rightarrow_b (C', \Sigma') \text{ avec } \Sigma' = \Sigma'^b \cup \Sigma^a \cup \Sigma^\psi$$

Deuxièmement, la réduction d'une instruction de l'action peut, en général, modifier l'état du programme de base et l'état local de l'aspect. Formellement :

$$\forall (C, \Sigma).(i_a : C, \Sigma) \rightarrow_b (C', \Sigma') \text{ avec } \Sigma' = \Sigma'^b \cup \Sigma'^a \cup \Sigma^\psi$$

Par la suite, les programmes sont représentés par leurs traces d'exécution. S'ils terminent, la configuration finale est de la forme $(\epsilon : \bullet, \Sigma)$. Par souci de simplicité et de régularité, nous considérons que toutes les traces sont infinies. Pour cela, l'instruction finale ϵ est supposée se réduire comme suit :

$$\forall \Sigma.(\epsilon : \bullet, \Sigma) \rightarrow_b (\epsilon : \bullet, \Sigma)$$

De cette manière, les exécutions des programmes qui terminent ou pas sont représentées par des traces infinies.

La trace d'exécution du programme de base, avec (C_0, Σ_0) comme configuration initiale est notée $\mathcal{B}(C_0, \Sigma_0)$.

DÉFINITION **8.**

$$\begin{array}{rl} \mathcal{B}(C_0, \Sigma_0) = & (i_1, \Sigma_1) : (i_2, \Sigma_2) : \ldots \\ \textit{avec} & \forall (j \geq 0).(i_j : C_j, \Sigma_j) \rightarrow_b (i_{j+1} : C_{j+1}, \Sigma_{j+1}) \end{array}$$

Comme les propriétés concernent uniquement les états et les instructions courantes, la continuation n'apparaît pas dans les traces. Nous notons $\mathcal{W}(C_0, \Sigma_0)$ la trace infinie du programme tissé.

DÉFINITION **9.**

$$\begin{array}{rl} \mathcal{W}(C_0, \Sigma_0) = & (i_1, \Sigma_1) : (i_2, \Sigma_2) : \ldots \\ \textit{avec} & \forall (j \geq 0).(i_j : C_j, \Sigma_j) \rightarrow (i_{j+1} : C_{j+1}, \Sigma_{j+1}) \end{array}$$

Remarquons que dans les deux définitions, l'instruction initiale i_0 (*c.-à-d.,* $start$) inutile ici, n'apparaît pas. Dans la suite de ce document, si α est une trace alors son $i^{\text{ème}}$ élément est noté α_i et ses préfixes, postfixes et sous traces sont notés comme suit :

$$\begin{array}{rcl} \alpha_{\rightarrow j} & = & \alpha_1 : \ldots : \alpha_j \\ \alpha_{j\rightarrow} & = & \alpha_j : \alpha_{j+1} \ldots \\ \alpha_{i\rightarrow j} & = & \alpha_i : \ldots : \alpha_j \end{array}$$

avec $i > 0$ et $j > 0$. La trace vide est notée $\alpha_{\rightarrow 0}$.

Par la suite, pour raisonner sur ces traces (*c.-à-d.,* définir les différentes catégories d'aspect), nous utilisons les fonctions $proj_b$ et $preserve_b$. La fonction $proj_b$ projette une trace de base ou tissée sur la séquence des instructions du programme de base qui a été exécutée. Nous notons $Traces_{\mathcal{B}}$ (resp. $Traces_{\mathcal{W}}$) l'ensemble des traces d'exécution du programme de base (resp. du programme tissé) et $Sequence_{i_b}$ l'ensemble des séquences d'instructions du programme de base.

$$\begin{array}{l} proj_b : Traces_{\mathcal{B}} \cup Traces_{\mathcal{W}} \rightarrow Sequence_{i_b} \\ proj_b((i_b, \Sigma) : T) \ = \ i_b : (proj_b\ T) \\ proj_b((i_a, \Sigma) : T) \ = \ proj_b\ T \end{array}$$

Le prédicat $preserve_b$ vérifie que les instructions de l'action dans une trace tissée ne modifient pas Σ^b. Chaque instruction i_a doit laisser l'état du programme de base inchangé.

$$\begin{array}{lcl} preserve_b : Traces_{\mathcal{W}} \rightarrow bool & & \\ preserve_b(\tilde{\alpha}) & = & \forall (j \geq 1).\ \tilde{\alpha}_j = (i_a, \Sigma_j) \\ & & \Rightarrow\ \tilde{\alpha}_{j+1} = (i, \Sigma_{j+1})\ \wedge\ \Sigma^b_j = \Sigma^b_{j+1} \end{array}$$

3.3 Logique temporelle linéaire

La logique temporelle linéaire (LTL) permet de définir une large gamme de propriétés sur l'exécution [MP92] des programmes. Dans cette section, nous définissons la syntaxe et la sémantique des formules de LTL basées sur nos traces d'exécution. Nous passons en revue les classes standards de propriétés en logique temporelle et discutons brièvement des raisons pour lesquelles, en général, ces classes ne sont pas préservées par le tissage.

3.3.1 Propositions atomiques

Dans notre contexte, une proposition atomique ap de LTL est, soit une proposition atomique sp sur l'état Σ (*par ex.,* $x \geq 0$), soit une une proposition atomique ep sur les instructions ou les événements (*par ex.,* `foo` qui est l'événement "la méthode `foo` est appelée").

Une proposition atomique ap est vraie à une étape α_j d'une trace (du programme de base ou tissé) si et seulement si α_j satisfait ap noté $\alpha_j \models ap$. Cela est défini en se basant sur les fonctions auxiliaires suivantes :

– Soit $Instruction$ l'ensemble des instructions et Ep l'ensemble des propositions atomiques sur les instructions, alors la fonction $m :: Instruction \times Ep \rightarrow bool$, retourne *true* si la proposition atomique satisfait l'instruction courante. La fonction m est surchargée

afin de prendre un élément d'une trace en paramètre :

$$\begin{array}{lcl} m :: Step \times Ep & \rightarrow & bool \\ m((i,\Sigma), ep) & = & m(i, ep) \end{array}$$

– Soit $State_B$ l'ensemble des Σ^b et Sp l'ensemble des propositions atomiques sur Σ^b, alors la fonction $l :: State_B \times Sp \rightarrow bool$, retourne *true* si la proposition est satisfaite par l'état passé en paramètre. La fonction l est aussi surchargée afin de prendre un élément d'une trace en paramètre :

$$\begin{array}{lcl} l :: Step \times Sp & \rightarrow & bool \\ l((i,\Sigma), sp) & = & l(\Sigma^b, sp) \end{array}$$

Donc, $\alpha_j \models ap$ est définie par :

$$\begin{array}{lcl} \alpha_j \models ep & \Leftrightarrow & m(\alpha_j, ep) = true \\ \alpha_j \models \neg ep & \Leftrightarrow & m(\alpha_j, ep) = false \\ \alpha_j \models sp & \Leftrightarrow & l(\alpha_j, sp) = true \\ \alpha_j \models \neg sp & \Leftrightarrow & l(\alpha_j, sp) = false \end{array}$$

3.3.2 Sémantique de LTL

La définition de LTL que nous considérons ici, est sa forme normale positive : la négation apparaît uniquement sur les propositions atomiques (Grammaire 2). L'opérateur $\bigcirc$ se lit "next", $\cup$ se lit "until", et W se lit "weak until".

GRAMMAIRE **2.**

$$\begin{array}{lcl} \varphi & ::= & ap \mid \neg ap \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \\ & & \bigcirc\varphi \mid \varphi_1 \cup \varphi_2 \mid \varphi_1 W \varphi_2 \end{array}$$

La sémantique d'une formule LTL est définie sur une trace α comme suit :

$$\begin{array}{lcl} \alpha \models ap & \Leftrightarrow & \alpha_1 \models ap \\ \alpha \models \neg ap & \Leftrightarrow & \alpha_1 \models \neg ap \\ \alpha \models \varphi_1 \vee \varphi_2 & \Leftrightarrow & \alpha \models \varphi_1 \vee \alpha \models \varphi_2 \\ \alpha \models \varphi_1 \wedge \varphi_2 & \Leftrightarrow & \alpha \models \varphi_1 \wedge \alpha \models \varphi_2 \\ \alpha \models \bigcirc\varphi & \Leftrightarrow & \alpha_{2\rightarrow} \models \varphi \\ \alpha \models \varphi_1 \cup \varphi_2 & \Leftrightarrow & \exists(j \geq 1).\alpha_{j\rightarrow} \models \varphi_2 \wedge \\ & & \forall(1 \leq i < j).\alpha_{i\rightarrow} \models \varphi_1 \\ \alpha \models \varphi_1 W \varphi_2 & \Leftrightarrow & \forall(j \geq 1).\ \alpha_{j\rightarrow} \models \varphi_1 \ \vee \ \alpha \models \varphi_1 \cup \varphi_2 \end{array}$$

Une proposition atomique ap (resp. $\neg ap$) est vraie sur α si ap est vraie (resp. fausse) sur le premier élément de α ; $\alpha_1 \vee \alpha_2$ est vraie si α_1 est vraie ou α_2 est vraie ; $\alpha_1 \wedge \alpha_2$ est vraie si α_1 est vraie et α_2 est vraie ; $\bigcirc\varphi$ est vraie si φ est vraie sur la trace qui suit immédiatement ; $\varphi_1 \cup \varphi_2$

est vraie si φ_1 est vraie jusqu'à ce que φ_2 le devienne ; finalement, $\varphi_1 W \varphi_2$ est vraie si φ_1 est toujours vraie ou $\varphi_1 \cup \varphi_2$ est vraie.

Les opérateurs dérivés $\Diamond$ et $\Box$ souvent utiles sont définis ainsi :
- $\Diamond\varphi = true \cup \varphi$ se lit "eventually φ" *c.-à-d.,* dans le futur, il existe une trace (postfixe) qui satisfait φ ;
- $\Box\varphi = \varphi \, W false$ se lit "always φ" *c.-à-d.,* toutes les traces (postfixes) de la trace satisfont φ.

3.3.3 Classes standards de propriétés en logique temporelle

Les classes standards de propriétés en logique temporelle sont :
- les propriétés de sûreté : "quelque chose (de mauvais) n'arrive jamais". Ces propriétés sont de la forme $\Box\varphi$ *c.-à-d.,* "toujours non quelque chose" ;
- les propriétés de vivacité : "quelque chose (de bon) arrivera". Ces propriétés sont de la forme $\Diamond\varphi$. Les propriétés de vivacité peuvent être répétées pour exprimer des propriétés d'équité (*c.-à-d.,* "quelque chose qui arrivera infiniment souvent"). Dans ce cas, elles sont de la forme $\Box\Diamond\varphi$;
- les invariants : "quelque chose est toujours vraie". Ils sont de la forme $\Box\varphi$ où φ est composée des propositions atomiques, des négations, disjonctions et conjonctions mais pas d'opérateurs temporels. Ils représentent le sous-ensemble des propriétés de sûreté qui ne dépendent pas de l'historique de l'exécution.

En général, ces classes de propriétés ne sont pas préservées par le tissage. Considérons, par exemple, la propriété de vivacité $\Diamond$`backup` spécifiant que "l'état du système sera sauvegardé". Un aspect `around` qui remplace les appels de la fonction `backup` par une action vide permet d'obtenir un programme tissé qui violera cette propriété. Pour les propriétés de sûreté, considérons un programme de base qui n'appelle jamais la fonction `diskformat` et satisfait par conséquent $\Box\neg$`diskformat`. Un aspect qui appelle cette fonction dans son action permet d'obtenir un programme tissé qui violera cette propriété.

La suite de ce chapitre est consacrée à l'identification des catégories d'aspect qui préservent des classes de propriétés temporelles.

3.4 Catégories d'aspect

Pour les programmes déterministes, nos catégories d'aspect sont : observateurs ($\mathcal{A}_o$), terminateurs ($\mathcal{A}_t$), verrouilleurs ($\mathcal{A}_v$) et faiblement intrusifs ($\mathcal{A}_f$). Pour chaque catégorie $\mathcal{A}_x$, nous présentons une classe de propriétés φ^x (un sous ensemble de LTL) qui est préservée par le tissage de tout aspect de $\mathcal{A}_x$. Ces catégories d'aspect sont liées par inclusion :

$$\mathcal{A}_o \subset \mathcal{A}_t \subset \mathcal{A}_v \subset \mathcal{A}_f$$

La catégorie observateur est la plus contrainte des catégories ; elle est incluse dans toutes les autres. La catégorie faiblement intrusif est la plus expressive ; elle inclut toutes les autres. Les classes de propriétés qui leurs correspondent sont aussi liées par inclusion :

$$\varphi^o \supset \varphi^t \supset \varphi^v \supset \varphi^f$$

Il n'est pas surprenant que la catégorie d'aspect la plus restreinte ($\mathcal{A}_o$) préserve la classe de propriétés la plus grande (φ^o) et que la chaîne d'inclusion soit dans la direction opposée.

Un point important à garder dans l'esprit est que nos preuves de préservation sont pour tout programme, tout aspect dans une catégorie et toute propriété dans une classe. Bien sûr, pour un programme et un aspect spécifique, plus de propriétés sont sans doute préservées. D'un autre côté, l'avantage de notre approche est que si on montre qu'un aspect appartient à une catégorie, alors on connaît une large classe de propriétés qui sera préservée quelque soit le programme.

Pour ces raisons, l'opérateur $\bigcirc$ ne peut apparaitre dans les classes de propriétés. En effet, une trace satisfait $\bigcirc\varphi$ uniquement si la séquence qui suit immédiatement satisfait φ. Le tissage d'un aspect même celui ayant le moins d'impact (par exemple un aspect insérant l'instruction nop) ne préserve pas ce type de propriété. Il suffit de le tisser juste avant que φ soit satisfait. Comme tous les aspects introduisent des étapes dans une trace d'exécution, aucune catégorie d'aspect ne préserve des $\bigcirc$-propriétés pour tout programme. Dans la suite de ce chapitre, pour illustrer nos catégories d'aspect et leurs classes, nous utiliserons des exemples de traces d'exécution où uniquement des informations intéressantes sont présentées. Par exemple :

$$x = 0 : x = 0 : (print, x = 1) : \epsilon : \epsilon : \ldots$$

représente une trace d'exécution où la première et la seconde étape satisfont $x = 0$ et la troisième satisfait $x = 1$ et a $print$ comme instruction courante. La seconde instruction qui n'est pas représentée ici a changé la valeur de x. Cette trace satisfait, par exemple, la propriété $(x = 0)Wprint$.

3.4.1 Les observateurs

Un observateur (Définition 10) ne modifie pas le flot de contrôle du programme de base mais insère uniquement les instructions de l'action i_a. Les traces d'exécution tissée et de base peuvent être projetées ($proj_b$) sur la même séquence d'instructions de base i_b. Un observateur ne modifie pas l'état du programme de base. C'est à dire que les instructions de l'action i_a ne changent pas l'état de base Σ^b. C'est ce que vérifie le prédicat $preserve_b$ (définit à la Section 3.2).

DÉFINITION **10.**

$$\begin{aligned} \forall(C,\Sigma).\ \Sigma^\psi \in \mathcal{A}_o \quad &\Leftrightarrow \quad proj_b(\alpha) = proj_b(\tilde{\alpha}) \\ &\qquad \wedge\, preserve_b(\tilde{\alpha}) \\ \textit{avec}\ \ \alpha = \mathcal{B}(C,\Sigma^b)\ \ &\textit{et}\ \ \tilde{\alpha} = \mathcal{W}(C,\Sigma) \end{aligned}$$

La définition 10 mentionne que les observateurs peuvent modifier la trace d'exécution en insérant des nouvelles instructions d'action (i_a) et un nouvel état local (Σ^a). Notons aussi que cette définition implique que l'action termine. En effet, lorsque l'action ne termine pas, les traces de base et tissée ne sont jamais égales lorsque l'on se projette sur les instructions du programme de base.

La classe de propriétés φ^o préservée par les aspects observateurs est définie par la Grammaire 3.

GRAMMAIRE **3.**

$$\begin{array}{rcl} \varphi^o & ::= & sp \mid \neg sp \mid \varphi_1^o \vee \varphi_2^o \mid \varphi_1^o \wedge \varphi_2^o \mid \varphi_1^o \cup \varphi_2^o \mid \\ & & \varphi_1^o W \varphi_2^o \mid true \cup \varphi'^o \\ \\ \varphi'^o & ::= & ep \mid \neg ep \mid sp \mid \neg sp \mid \varphi_1'^o \vee \varphi_2'^o \mid \varphi_1'^o \wedge \varphi_2'^o \mid \\ & & \varphi_1^o \cup \varphi_2^o \mid \varphi_1^o W \varphi_2^o \mid true \cup \varphi'^o \end{array}$$

Comme dans la section précédente, les sp et ep font respectivement référence à des propositions atomiques sur l'état de base et sur les instructions. Lorsque les propositions atomiques portent sur les états (sp), le langage φ^o est celui de LTL sans l'opérateur $\bigcirc$. Ainsi, il permet d'exprimer toutes les propriétés de sûreté, de vivacité et les invariants (sans $\bigcirc$) sur les états de base Σ^b. La classe est plus restreinte lorsqu'elle concerne les propositions atomiques sur les événements (ep). Ses propriétés sont uniquement de la forme $true \cup \varphi'^o$ et permettent de définir les propriétés de vivacité sur les événements. En effet, une propriété de vivacité $\Diamond\varphi'^o$ (resp. une propriété de vivacité équitable $\Box\Diamond\varphi'^o$) peut être réécrite comme $true \cup \varphi'^o$ (resp. $(true \cup \varphi'^o)W false$). D'autre part, ce langage interdit les propriétés de sûreté sur les événements. Une propriété de sûreté $\Box\neg\varphi$ est de la forme $(\neg\varphi)W false$ qui n'appartient pas à la Grammaire 3. Intuitivement, les propriétés de sûreté sur les événements interdisent certaine séquences d'instructions. Un aspect observateur introduisant des séquences d'instructions peut introduire des séquences interdites. Par exemple, la séquence de base

$$x = 0 : x = 0 : (print, x = 1) : \epsilon : \epsilon : \ldots$$

satisfait $(x = 0) \ \cup \ print$ et $(x = 0)W print$, mais après le tissage de l'instruction $write$ qui est l'action d'un observateur juste avant $print$, on obtient

$$x = 0 : x = 0 : (write, x = 1) : (print, x = 1) : \epsilon : \epsilon : \ldots$$

L'action $write$ d'un observateur modifie uniquement l'état local à l'aspect. La variable x conserve donc, pendant l'exécution de l'action, la valeur qu'elle avait lorsque l'aspect a sélectionné le point de jonction $print$ *c.-à-d.,* $x = 1$. Avec cette nouvelle trace, les propriétés satisfaites par la séquence de base ne le sont plus. De même, $read W false$ (*c.-à-d.,* toujours $read$) est satisfaite par une trace infinie d'instructions $read$

$$read : read : read : \ldots$$

mais après le tissage de l'action $write$ après le premier $read$

$$read : write : read : read : \ldots$$

la propriété n'est plus satisfaite.

Le Théorème 1 assure que le tissage d'un observateur préserve toutes les propriétés de φ^o qui sont satisfaites par le programme de base. La preuve de ce théorème se trouve à l'Appendice A.1.

THÉORÈME **1.**

$$\forall(C,\Sigma).\ \Sigma^{\psi} \in \mathcal{A}_o \ \Rightarrow\ \forall(p \in \varphi^o).\ \alpha \models p \Rightarrow \tilde{\alpha} \models p$$
$$\textit{avec}\ \ \alpha = \mathcal{B}(C,\Sigma^b)\ \ \textit{et}\ \ \tilde{\alpha} = \mathcal{W}(C,\Sigma)$$

Les aspects de persistance, de profilage, de débogage, de traçage et de journalisation appartiennent typiquement à la catégorie des observateurs. Les aspects de persistance qui sauvegardent uniquement les états du programme de base durant son exécution dans une base de données sont clairement des observateurs. Les aspects de débogage qui impriment les variables du programme de base pour tester des assertions ou tracer des valeurs d'un programme sont des observateurs. Les aspects de traçage, de journalisation ou de profilage, usuellement, observent uniquement l'exécution du programme de base et écrivent des informations (*par ex.,* appels de procédure, valeurs de paramètres, *etc.*) sur cette exécution dans un fichier ou sur la sortie standard (supposé ne pas être dans l'état du programme de base ou de l'aspect). Un exemple d'aspect de profilage sont des aspects d'analyse dynamique qui observent l'exécution du programme, détectent des comportements douteux et informent le programmeur. Par exemple, Govidranj et coll. [CVK06] présentent un outil nommé InfraRED. Il est basé sur des aspects développés avec AspectJ et qui observent l'exécution d'applications J2EE afin de détecter et d'analyser des problèmes de performance. La documentation d'AspectJ, possède plusieurs aspects de profilage tels que `telecom/TimerLog`, `tracing/lib/TraceMyClasses`, `tjp/GetInfo`, *etc.*

3.4.2 Les terminateurs

Un aspect terminateur (Définition 11) ne modifie pas l'état du programme de base. Comme dans le cas des observateurs, la trace tissée satisfait donc le prédicat $preserve_b$. Cependant, un terminateur peut modifier le flot de contrôle en terminant l'exécution du programme tissé. Cela est modélisé par l'instruction `abort` qui réduit toute configuration en une configuration finale :

$$\forall(C,\Sigma).\ (\texttt{abort} : C,\Sigma) \rightarrow (\epsilon : \bullet,\Sigma)$$

Si `abort` n'est pas exécutée, les projections sur les instructions du programme de base d'une trace de base et d'une trace tissée sont égales ; l'aspect terminateur se comporte comme un observateur. La projection sur les i_b d'une trace tissée terminée par `abort` est un préfixe de la même projection sur la trace du programme de base. Après `abort`, toutes les instructions courantes sont égales à ϵ.

DÉFINITION **11.**

$$\begin{array}{lll} \forall(C,\Sigma).\ \Sigma^{\psi} \in \mathcal{A}_t & \Leftrightarrow & preserve_b(\tilde{\alpha}) \wedge \\ & & proj_b(\alpha) = proj_b(\tilde{\alpha})\ \vee (\exists(i \geq 0). \\ & & \exists(j \geq i).\ proj_b(\alpha_{\rightarrow i}) = proj_b(\tilde{\alpha}_{\rightarrow j}) \\ & & \wedge \forall(k > j).\tilde{\alpha}_k = (\epsilon, \_)) \end{array}$$

avec $\alpha = \mathcal{B}(C,\Sigma)$ *et* $\tilde{\alpha} = \mathcal{W}(C,\Sigma)$

Notons que cette définition n'accepte pas les actions qui ne terminent pas ($proj_b(\alpha) = proj_b(\tilde{\alpha})\ \vee\ \forall(k > j).\tilde{\alpha}_k = (\epsilon, \_)$). En effet, quand une action qui ne modifie pas l'état du programme de base ne se termine pas, la projection de sa trace tissée sur les i_b est un préfixe de la projection sur la trace du programme de base. L'aspect n'est toutefois pas terminateur car l'exécution du programme tissé ne se termine pas par `abort`.

Les observateurs sont inclus dans la catégorie des terminateurs. L'ensemble des propriétés préservées par les terminateurs (Grammaire 4) est donc un sous ensemble de l'ensemble des propriétés préservées par les observateurs (Grammaire 3).

GRAMMAIRE **4.**

$$\varphi^t ::= sp \mid \neg sp \mid \varphi_1^t \vee \varphi_2^t \mid \varphi_1^t \wedge \varphi_2^t \mid \varphi_1^t W \varphi_2^t \mid\ true \cup \varphi'^t$$

$$\varphi'^t ::= \neg ep\ \mid \varphi'^t \vee \varphi^t\ \mid \varphi_1'^t \wedge \varphi_2'^t\ \mid true \cup \varphi'^t$$

Lorsque les propositions atomiques portent sur les états (sp), le langage φ^t est une proposition atomique ou une conjonction, disjonction et propriétés sur "weak until" de φ^t. Ce langage permet, sur les états, d'exprimer les propriétés de sûreté qui ne contiennent pas les opérateurs $\cup$ et $\bigcirc$. Les propositions atomiques sur les événements apparaissent uniquement sous une négation et dans une formule "eventually" (*c.-à-d.,* dans $true \cup \varphi'^t$). Ainsi, sur les événements, seules les propriétés de vivacité sur $\neg ep$ peuvent être exprimées. Par exemple, la propriété $true \cup \neg print$ (qu'on peut lire "un événement qui n'est pas $print$ arrivera") satisfaite par la séquence

$$print : print : print : read : \epsilon : \ldots$$

est préservée par tout terminateur. En effet, un terminateur, permettra d'atteindre l'instruction $read$ ou mettra un terme à l'exécution ; dans les deux cas, l'instruction courante sera différente de $print$ (ϵ n'est pas $print$). Nous supposons ici que ep ne peut pas dénoter l'instruction sémantique ϵ ; en effet, $true \cup \neg\epsilon$ ne serait pas préservée par un terminateur qui termine le programme avant l'exécution de la première instruction du programme de base.

Plusieurs propriétés préservées par les aspects observateurs ne sont pas préservées par les terminateurs. Cela vient de la possibilité d'arrêter l'exécution des programmes. Par exemple, $x = 0 \cup x = 1$ est satisfaite par la séquence suivante

$$x = 0 : x = 0 : x = 1 : \epsilon : \epsilon : \ldots$$

mais si un terminateur termine son exécution avant $x = 1$ alors la trace tissée devient

$$(\texttt{abort}, x = 0) : (\epsilon, x = 0) : (\epsilon, x = 0) : \ldots$$

et la propriété $x = 0 \cup x = 1$ n'est plus satisfaite. Les propriétés de la forme $\varphi_1^t W \varphi_2^t$ comme $x = 0 W x = 1$ sont préservées.

La préservation des propriétés de la Grammaire 4 par les terminateurs est formulée par le Théorème 2.

THÉORÈME **2.**

$$\begin{array}{l} \forall (C, \Sigma).\ \Sigma^{\psi} \in \mathcal{A}_t \ \Rightarrow\ \forall (p \in \varphi^t).\ \alpha \models p \Rightarrow \tilde{\alpha} \models p \\ \textit{avec}\ \ \alpha = \mathcal{B}(C, \Sigma^b)\ \ \textit{et}\ \ \tilde{\alpha} = \mathcal{W}(C, \Sigma) \end{array}$$

Les terminateurs les plus communs sont des aspects de sécurité qui détectent des états ou séquences d'instructions interdits. Ils peuvent aussi être des aspects qui garantissent qu'une exécution s'arrête après un délai. En général, un aspect qui teste si une condition est violée par le programme de base et lève une exception sans modifier l'état de base est un terminateur. Dans [CF00], les aspects sont des politiques de sécurité locale qui peuvent être tissées dans des applets non sécurisées. Ces aspects modifient uniquement leurs états mais arrêtent l'applet dès qu'elle essaye de violer la politique. Dans [FHTH07], les aspects temporisés terminent l'exécution des programmes afin de garantir la disponibilité des ressources partagées. Dans [CVK06], Wampler présente un outil nommé Crontract4J qui prend des invariants et génère des aspects qui assurent que ces invariants sont satisfaits par le programme tissé. Ces aspects observent l'exécution et l'arrêtent dès que l'invariant est violé.

3.4.3 Les verrouilleurs

Un aspect est un verrouilleur (Définition 12) si l'état du programme de base Σ^b dans toute configuration du programme tissé est un état atteignable par le programme de base. En général, les verrouilleurs peuvent modifier le flot de contrôle et l'état du programme de base.

L'ensemble des états atteignables à partir d'une configuration composée du programme C et de l'état Σ^b, noté par $Reach_b(C, \Sigma^b)$, se définit comme suit :

$$Reach_b(C, \Sigma^b) = \{\Sigma^{b'} \mid\ (C, \Sigma^b) \xrightarrow{*}_b (C', \Sigma^{b'})\}$$

La Définition 12 formalise le fait que l'état du programme de base de toute configuration dans la trace tissée est atteignable par le programme de base.

DÉFINITION **12.**

$$\begin{array}{l} \forall (C, \Sigma).\ \Sigma^{\psi} \in \mathcal{A}_v \ \Leftrightarrow\ \forall (j \geq 1).\ \tilde{\alpha}_j = (i, \Sigma_j) \\ \qquad\qquad\qquad\qquad \wedge\ \Sigma_j^b \ \in\ Reach_b(C, \Sigma^b) \\ \textit{avec}\ \ \alpha = \mathcal{B}(C, \Sigma)\ \ \textit{et}\ \ \tilde{\alpha} = \mathcal{W}(C, \Sigma) \end{array}$$

Les observateurs et les terminateurs sont inclus dans la catégorie $\mathcal{A}_v$ des verrouilleurs. L'ensemble des propriétés préservées par les verrouilleurs (Grammaire 5) est donc un sous ensemble de l'ensemble des propriétés préservées par les aspects terminateurs (Grammaire 4).

GRAMMAIRE **5.**

$$\varphi^v ::= sp \mid \neg sp \mid \varphi_1^v \vee \varphi_2^v \mid \varphi_1^v \wedge \varphi_2^v \mid \varphi_1^v W false$$

Le langage φ^v est restreint aux invariants (*c.-à-d.,* $\Box\varphi$ or $\varphi W false$) sur les états. Puisque les aspects verrouilleurs peuvent modifier le flot de contrôle des événements sans aucune restriction, aucune propriété dont les propositions atomiques portent sur les événements n'est préservée. Pour la même raison, les propriétés de sûreté telle que $\varphi_1^v W \varphi_2^v$ ne sont pas préservées par les verrouilleurs. Par exemple, l'exécution de base

$$x = 0 : x = 1 : x = 2 : \epsilon : \epsilon : \ldots$$

satisfait la propriété $x = 0 W x = 1$. Cependant, après le tissage d'un verrouilleur qui reste dans $Reach_b$, la séquence tissée peut être

$$x = 0 : x = 2 : x = 0 : x = 1 : \epsilon : \ldots$$

qui ne satisfait plus $x = 0 W x = 1$.

La préservation des propriétés de la Grammaire 5 par les verrouilleurs est formalisée par le Théorème 3.

THÉORÈME **3.**

$$\forall (C, \Sigma).\ \Sigma^{\psi} \in \mathcal{A}_v \ \Rightarrow\ \forall (p \in \varphi^v).\ \alpha \models p \Rightarrow \tilde{\alpha} \models p$$
$$\textit{avec}\ \ \alpha = \mathcal{B}(C, \Sigma^b)\ \ \textit{et}\ \ \tilde{\alpha} = \mathcal{W}(C, \Sigma)$$

Les exemples de verrouilleurs classiques sont des aspects de ré-initialisation qui restaurent l'état initial du programme de base, les aspects de tolérance aux fautes qui rétablissent un état d'exécution sûr à partir d'un point de contrôle, les aspects mémo qui raccourcissent une exécution en retournant directement des résultats de calculs ou requêtes déjà effectuées et sauvegardés. Dans tous les cas, afin de rester toujours dans l'ensemble des états atteignables, la restauration ou sauvegarde doit être atomique. Par exemple, une ré-initialisation non atomique crée temporairement des états non atteignables pendant la restauration. Dans ce cas, les aspects appartiennent à la catégorie suivante, les faiblement intrusifs.

3.4.4 Les faiblement intrusifs

Un aspect est faiblement intrusif (Définition 13) si dans une trace tissée, l'état du programme de base de chaque configuration avec comme instruction courante une instruction du programme

de base (*c.-à-d.,* i_b), est un état atteignable. En d'autres termes, un aspect faiblement intrusif peut produire des états du programme de base non atteignables pendant l'exécution de son action mais retourne toujours vers des états atteignables, quand il rend la main au programme de base. Les verrouilleurs sont des cas spéciaux de la catégorie des faiblement intrusifs.

La Définition 13 formalise le fait que, l'état du programme de base de toute configuration de la trace tissée avec une instruction courante du programme de base (i_b) est atteignable par le programme de base.

DÉFINITION **13.**

$$\begin{array}{l} \forall(C,\Sigma).\ \Sigma^{\psi} \in \mathcal{A}_f \ \Leftrightarrow\ \forall(j \geq 1).\ \tilde{\alpha}_j = (i_b, \Sigma_j) \\ \qquad\qquad\qquad\qquad \Rightarrow\ \Sigma^b_j \ \in\ Reach_b(C, \Sigma^b) \\ \textit{avec}\ \ \alpha = \mathcal{B}(C,\Sigma)\ \ \textit{et}\ \ \tilde{\alpha} = \mathcal{W}(C,\Sigma) \end{array}$$

Comme un aspect faiblement intrusif peut modifier le flot de contrôle et l'état du programme de base, il peut violer les invariants pendant l'exécution de l'action. Il n'existe donc aucune propriété de LTL préservée pour tout aspect faiblement intrusif et tout programme. Cependant, si l'exécution de tout faiblement intrusif termine (Définition 14), il préserve les propriétés de la forme $\Diamond\varphi^v$. Cela veut dire que, le programme tissé préserve les invariants φ^v (définie par la Grammaire 5) à un certain point de son exécution.

DÉFINITION **14.**

$$\begin{array}{l} \forall(C,\Sigma).\ \Sigma^{\psi}\ \textit{termine}\ \ \Leftrightarrow\ \ \exists(j \geq 1).\forall(k > j). \\ \qquad\qquad\qquad\qquad\ \ \tilde{\alpha}_k = (i_b, \Sigma_k) \\ \textit{avec}\ \ \alpha = \mathcal{B}(C,\Sigma)\ \ \textit{et}\ \ \tilde{\alpha} = \mathcal{W}(C,\Sigma) \end{array}$$

Par exemple, la trace du programme de base

$$x = 0 : x = 1 : x = 0 : (\epsilon, x = 1) : (\epsilon, x = 1) : \ldots$$

satisfait la propriété de φ^v $(x = 0 \vee x = 1)Wfalse$. La séquence tissée

$$x = 0 : x = 1 : x = 0 : x = 2 : (\epsilon, x = 0) : (\epsilon, x = 0) : \ldots$$

viole la propriété lorsque $x = 2$ (un état produit par l'exécution de l'action). Cependant la configuration finale $(\epsilon, x = 0)$ a un état $x = 0$ atteignable par le programme de base. Ainsi, $(x = 0 \vee x = 1)Wfalse$ sera satisfaite à partir de la configuration finale (*c.-à-d.,* $\Diamond((x = 0 \vee x = 1)Wfalse)$).

Le théorème 4 formalise le fait que si le programme de base satisfait un invariant p alors une exécution tissée avec un faiblement intrusif qui termine selon la Définition 14 satisfait $\Diamond p$

THÉORÈME **4.**

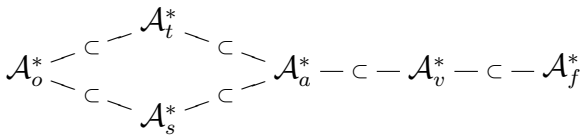
$$\begin{array}{l} \forall(C,\Sigma).\ \ \Sigma^{\psi} \in \mathcal{A}_f\ \wedge\ \Sigma^{\psi}\ \textit{termine}\ \ \Rightarrow \\ \qquad\qquad \forall(p \in \varphi^v).\ \alpha \models p\ \Rightarrow\ \tilde{\alpha} \models \Diamond p \\ \textit{avec}\ \ \alpha = \mathcal{B}(C,\Sigma)\ \ \textit{et}\ \ \tilde{\alpha} = \mathcal{W}(C,\Sigma) \end{array}$$

Les aspects de tolérance aux fautes qui restaurent un état précédent sûr de façon non atomique sont typiquement des aspects faiblement intrusifs. Ils peuvent produire des états non atteignables pendant l'exécution de l'action (*c.-à-d.,* la restauration) mais finiront par atteindre un état précédent sûr à la fin de l'action. Similairement, les aspects de réinitialisation non atomiques sont faiblement intrusifs.

3.5 Cas non-déterministe

Nous avons présenté des catégories d'aspect préservant des classes de propriétés pour des programmes déterministes. Le non déterminisme suggère deux nouvelles catégories d'aspect : les sélecteurs ($\mathcal{A}_s^*$) qui sélectionnent certaines exécutions parmi l'ensemble des exécutions, et les adaptateurs ($\mathcal{A}_a^*$) qui peuvent sélectionner mais aussi arrêter les exécutions.

Nous suivons la même démarche que dans le cas déterministe. Nous définissons la sémantique des programmes non déterministes comme un ensemble de traces d'exécution infinies. Les catégories d'aspect sont ensuite définies en se basant sur cette sémantique et les fonctions auxiliaires $proj_b$ et $preserve_b$. Les catégories observateur, terminateur, sélecteur, adaptateur, verrouilleur et faiblement intrusif forment une hiérarchie



où les terminateurs $\mathcal{A}_t^*$ et les sélecteurs $\mathcal{A}_s^*$ ne peuvent pas être comparés. Les propriétés sont définies en utilisant CTL* qui permet de quantifier les formules sur l'ensemble des traces d'exécution. Cette logique est plus expressive que LTL. Les classes de propriétés $\theta^o, \theta^t, \theta^s, \theta^a, \theta^v, \theta^f$ préservées par les catégories d'aspect qui leurs correspondent sont reliées par une hiérarchie d'inclusion duale.

Les exemples d'aspects discutés à la Section 3.4 restent valides dans le cas non déterministe. Par exemple, les aspects de debogage restent des observateurs même pour des programmes non déterministes. Dans cette section, nous ne présentons pas toutes les catégories mais nous nous focalisons sur les deux nouvelles (sélecteurs et adaptateurs) et les classes de propriétés qu'elles préservent.

3.5.1 Programme de base et programme tissé

Les exécutions du programme de base et du programme tissé sont représentées ici comme un ensemble de traces infinies notés $\mathcal{B}^*(C_0, \Sigma_0)$ (Définition 15) et $\mathcal{W}^*(C_0, \Sigma_0)$ (Définition 16). Comme dans le cas déterministe (Définitions 8 et 9), nous supposons que les programmes com-

mencent avec une instruction spéciale $start$ qui permet de tisser la première instruction du programme de base (celle après $start$). L'instruction $start$ est ignorée dans les traces.

DÉFINITION **15.**

$$\begin{aligned}\mathcal{B}^*(C_0,\Sigma_0) = \ & \{(i_1,\Sigma_1):(i_2,\Sigma_2):\ldots \mid \forall(j\geq 0).\\ & (i_j:C_j,\Sigma_j)\rightarrow_b (i_{j+1}:C_{j+1},\Sigma_{j+1})\}\end{aligned}$$

DÉFINITION **16.**

$$\begin{aligned}\mathcal{W}^*(C_0,\Sigma_0) = \ & \{(i_1,\Sigma_1):(i_2,\Sigma_2):\ldots \mid \forall(j\geq 0).\\ & (i_j:C_j,\Sigma_j)\rightarrow (i_{j+1}:C_{j+1},\Sigma_{j+1})\}\end{aligned}$$

3.5.2 La logique temporelle arborescente CTL*

Dans le cas non déterministe, les classes de propriétés préservées sont des sous ensembles de la logique temporelle arborescente CTL* [CES83]. La Grammaire 6 définie la forme normale positive des formules de CTL*.

GRAMMAIRE **6.**

$$\theta \ ::= \ ap \mid \neg ap \mid \theta_1 \vee \theta_2 \mid \theta_1 \wedge \theta_2 \mid \exists\omega \mid \forall\omega$$

$$\omega \ ::= \ \theta \mid \omega_1 \vee \omega_2 \mid \omega_1 \wedge \omega_2 \mid \bigcirc\omega \mid \omega_1 \cup \omega_2 \mid \omega_1 W \omega_2$$

Contrairement à LTL qui spécifie des propriétés sur une trace d'exécution, CTL* spécifie des propriétés sur un ensemble de traces d'exécution. Il étend LTL avec les quantificateurs logiques $\exists\omega$ ("il existe des traces qui satisfont ω") et $\forall\omega$ ("toutes les traces satisfont ω"). CTL* est donc plus expressif que LTL. En effet, toute propriété p de LTL pour une trace α est équivalente à la formule $\forall p$ pour l'ensemble $\{\alpha\}$. Dans la Grammaire 6, θ représente les propriétés sur les étapes d'une trace et ω les propriétés sur les traces avec la sémantique suivante :

$$\begin{array}{lcl}
T_j,\alpha_j \models ap & \Leftrightarrow & \alpha_j \models ap\\
T_j,\alpha_j \models \neg ap & \Leftrightarrow & \alpha_j \models \neg ap\\
T_j,\alpha_j \models \theta_1 \vee \theta_2 & \Leftrightarrow & T_j,\alpha_j \models \theta_1 \vee T_j,\alpha_j \models \theta_2\\
T_j,\alpha_j \models \theta_1 \wedge \theta_2 & \Leftrightarrow & T_j,\alpha_j \models \theta_1 \wedge T_j,\alpha_j \models \theta_2\\
T_j,\alpha_j \models \exists\omega & \Leftrightarrow & \exists(\alpha\in T_j).T_j,\alpha \models \omega\\
T_j,\alpha_j \models \forall\omega & \Leftrightarrow & \forall(\alpha\in T_j).T_j,\alpha \models \omega
\end{array}$$

$$\begin{array}{lcl}
T_j,\alpha \models \theta & \Leftrightarrow & T_j,\alpha_1 \models \theta\\
T_j,\alpha \models \omega_1 \vee \omega_2 & \Leftrightarrow & T_j,\alpha \models \omega_1 \vee T_j,\alpha \models \omega_2\\
T_j,\alpha \models \omega_1 \wedge \omega_2 & \Leftrightarrow & T_j,\alpha \models \omega_1 \wedge T_j,\alpha \models \omega_2\\
T_j,\alpha \models \bigcirc\omega & \Leftrightarrow & T_2,\alpha_2 \models \omega\\
T_j,\alpha \models \omega_1 \cup \omega_2 & \Leftrightarrow & \exists(i\geq 1).T_i,\alpha_i \models \omega_2 \wedge\\
 & & \forall(1\leq k<i).T_k,\alpha_k \models \omega_1\\
T_j,\alpha \models \omega_1 W \omega_2 & \Leftrightarrow & \forall(i\geq 1).\ T_i,\alpha_i \models \omega_1 \ \vee\ T_j,\alpha \models \omega_1 \cup \omega_2
\end{array}$$

Dans ces définitions, l'environnement T_j est l'ensemble des traces commençant par α_j. Dans notre contexte, T_j est initialement soit $\mathcal{B}^*(C_0, \Sigma_0)$ pour α_1, soit $\mathcal{W}^*(C_0, \Sigma_0)$ pour $\tilde{\alpha}_1$. Un ensemble de trace d'exécution à partir de α_j (T_j, α_j) satisfait ap si α_j satisfait ap (voir Section 3.3.1). Il en est de même pour pour $\neg ap$. Pour $\theta_1 \vee \theta_2$ (resp. $\theta_1 \wedge \theta_2$) l'ensemble des exécutions satisfait θ_1 (resp. θ_1) ou θ_2 (resp. et θ_2). Il satisfait $\exists\omega$ s'il existe une exécution $\alpha \in T_j$ (*c.-à-d.,* des traces commençant par α_j) qui satisfait ω. Pour $\forall\omega$, toutes les traces $\alpha \in T_j$ satisfont ω. La satisfaction de ω par une trace est définie comme dans LTL mais sur l'ensemble des traces d'exécution (T_j, α).

Les opérateurs $\Diamond$ et $\Box$ se définissent de la même manière dans CTL* mais avec des quantificateurs (*par ex.,* $\exists\Diamond$, $\forall\Diamond$).

3.5.3 Les sélecteurs

Un aspect sélecteur ne modifie pas l'état du programme de base. Cependant, un sélecteur peut modifier le flot de contrôle du programme de base en sélectionnant un sous ensemble des traces d'exécution dans l'ensemble de toutes les exécutions possibles. Puisque son rôle est de supprimer certain choix non déterministes, cette catégorie d'aspect n'a de sens que pour des programmes non déterministes.

Un sélecteur (Définition 17) n'introduit pas des nouvelles traces d'exécution. Ainsi, pour toute trace dans l'ensemble des exécutions tissées, il existe une trace dans l'ensemble des exécutions de base telle que les deux traces soient égales après une projection $proj_b$ sur les instructions du programme de base.

DÉFINITION **17.**

$$\forall(C,\Sigma).\ \Sigma^{\psi} \in \mathcal{A}_s^* \ \Leftrightarrow\ \begin{array}{l}\forall(\tilde{\alpha} \in \mathcal{W}^*(C,\Sigma)).\exists(\alpha \in \mathcal{B}^*(C,\Sigma^b)).\\ proj_b(\tilde{\alpha}) = proj_b(\alpha)\ \wedge\ preserve_b(\tilde{\alpha})\end{array}$$

Les propriétés définies par θ^s dans la Grammaire 7, sont préservées par les sélecteurs.

GRAMMAIRE **7.**

$$\begin{array}{lcl}\theta^s & ::= & sp \mid \neg sp \mid \theta_1^s \vee \theta_2^s \mid \theta_1^s \wedge \theta_2^s \mid \forall\omega^s\\ \\ \omega^s & ::= & \theta^s \mid \omega_1^s \vee \omega_2^s \mid \omega_1^s \wedge \omega_2^s \mid\\ & & \omega_1^s \cup \omega_2^s \mid \omega_1^s W \omega_2^s \mid true \cup \omega'^s\\ \\ \omega'^s & ::= & ep \mid \neg ep \mid \theta^s \mid \omega_1'^s \vee \omega_2'^s \mid \omega_1'^s \wedge \omega_2'^s \mid\\ & & \omega_1^s \cup \omega_2^s \mid \omega_1^s W \omega_2^s \mid true \cup \omega'^s\end{array}$$

La Grammaire 7 est celle des observateurs (*c.-à-d.,* une généralisation à CTL* de φ^o) qui ne contient pas le quantificateur existentiel ($\exists$) car une exécution du programme de base qui

satisfait une propriété $\exists\omega$ peut être supprimée par un aspect sélecteur. exemple, la propriété $\exists\,(x = 0 \cup x = 1)$ qui est préservée par l'ensemble des traces d'exécution suivant

$$\{\, x = 0 : x = 0 : x = 1 : \epsilon : \epsilon : \ldots\ ,\ x = 2 : x = 2 : \epsilon : \ldots \,\}$$

ne l'est plus après le tissage d'un aspect sélecteur qui supprime la trace

$$x = 0 : x = 0 : x = 1 : \epsilon : \epsilon : \ldots$$

Aussi, comme dans le cas des observateurs déterministes, les propriétés sur les traces $\omega_1^s \cup \omega_2^s$ et $\omega_1^s W \omega_2^s$ sont préservées que sur les états (sp) et pas sur les événements (ep). Par exemple, l'ensemble

$$\{\, x = 0 : x = 0 : (write, x = 1) : \epsilon : \epsilon : \ldots \,\}$$

satisfait la propriété $\forall\ (x = 0 \cup write)$ mais après le tissage d'un aspect sélecteur qui insère l'instruction $print$ avant $write$, elle ne l'est plus

$$\{\, x = 0 : x = 0 : (print, x = 1) : (write, x = 1) : \epsilon : \epsilon : \ldots \,\}$$

De même $\forall\ (writeWread)$ est satisfaite par l'ensemble

$$\{\, write : write : write : \ldots \,\}$$

mais après le tissage d'un aspect sélecteur qui insère l'instruction $print$ après le premier $write$, elle ne l'est plus

$$\{\, write : print : write : write : \ldots \,\}$$

La préservation de θ^s par les sélecteurs est exprimée par le Théorème 5

THÉORÈME **5.**

$$\begin{array}{l}\forall(C, \Sigma).\ \Sigma^{\psi} \in \mathcal{A}_s^* \ \Rightarrow \\ \forall(p \in \theta^s).\forall(\alpha \in \Gamma).\Gamma, \alpha_1 \models p \Rightarrow \forall(\tilde{\alpha} \in \tilde{\Gamma}).\tilde{\Gamma}, \tilde{\alpha}_1 \models p \\ \textit{avec } \Gamma = \mathcal{B}^*(C, \Sigma^b) \textit{ et } \tilde{\Gamma} = \mathcal{W}^*(C, \Sigma)\end{array}$$

Comme exemples de sélecteurs citons les aspects d'ordonnancement ou de raffinement qui suppriment certain choix non déterministes. Les aspects d'ordonnancement de [FHTH04] spécifient et appliquent des politiques d'ordonnancement à des réseaux de processus communicants. Ils sélectionnent un sous ensemble désiré de traces d'exécution dans l'ensemble de tous les entrelacements possibles. Ces aspects sont typiquement des sélecteurs.

3.5.4 Les adaptateurs

Les adaptateurs sont une combinaison des terminateurs et des sélecteurs. Un aspect adaptateur (Définition 18) ne modifie pas l'état du programme de base ($preserve_b$). Cependant, il peut modifier le flot de contrôle du programme de base, soit comme un terminateur en le terminant, soit comme un sélecteur en sélectionnant un sous ensemble des traces d'exécution. Pour toute trace $\tilde{\alpha}$ des exécutions du programme tissé :

– soit il existe une trace α dans l'ensemble des exécutions du programme de base qui a les mêmes instructions de base que $\tilde{\alpha}$. Donc l'aspect ne modifie pas le flot de contrôle du programme de base mais sélectionne ;
– soit il existe un préfixe $\alpha_{\rightarrow i}$ dans une trace d'exécution du programme de base et un préfixe $\tilde{\alpha}_{\rightarrow j}$ dans une trace d'exécution du programme tissé qui ont les mêmes instructions de base et le reste des instructions de la trace tissée est l'instruction finale ϵ.

DÉFINITION **18.**

$$\begin{aligned}\forall(C,\Sigma).\Sigma^{\psi} \in \mathcal{A}_a^* \Leftrightarrow\ & \forall(\tilde{\alpha} \in \mathcal{W}^*(C,\Sigma)).\exists(\alpha \in \mathcal{B}^*(C,\Sigma^b)).\\ & preserve_b(\tilde{\alpha}) \wedge\\ & proj_b(\tilde{\alpha}) = proj_b(\alpha) \vee\\ & \exists(i \geq 0).\ (\exists(j \geq i).\\ & proj_b(\alpha_{\rightarrow i}) = proj_b(\tilde{\alpha}_{\rightarrow j}) \wedge\\ & \forall(k > j).\ \tilde{\alpha}_k = (\epsilon, \_))\end{aligned}$$

Notons que, cette définition ne relie pas toutes les traces d'exécution du programme de base avec celles du programme tissé puisqu'un aspect adaptateur peut supprimer des traces dans l'ensemble des traces d'exécution du programme de base.

Les propriétés θ^a définies dans la Grammaire 8 sont préservées par les aspects adaptateurs.

GRAMMAIRE **8.**

$$\begin{aligned}\theta^a &::= sp \mid \neg sp \mid \theta_1^a \vee \theta_2^a \mid \theta_1^a \wedge \theta_2^a \mid \forall\omega^a\\ \omega^a &::= \theta^a \mid \omega_1^a \vee \omega_2^a \mid \omega_1^a \wedge \omega_2^a \mid \omega_1^a W \omega_2^a \mid true \cup \omega'^a\\ \omega'^a &::= \neg ep \mid \omega'^a \vee \theta^a \mid \omega_1'^a \wedge \omega_2'^a \mid true \cup \omega'^a \mid \forall\omega'^a\end{aligned}$$

La Grammaire 8 peut être vue comme l'intersection de la classe de propriétés préservées par les sélecteurs (θ^s) et la classe préservée par les terminateurs (*c.-à-d.,* une généralisation à CTL* de φ^a). Ainsi, comme dans le cas des sélecteurs, l'opérateur $\exists$ est exclu. Comme dans le cas déterministe, la propriété sur les traces $\omega_1^a \cup \omega_2^a$ n'est pas préservée car comme les terminateurs, un adaptateur peut arrêter le programme avant que ω_1^a ou ω_2^a ne soit satisfait. Par exemple, la propriété $\forall\ (x = 0 \cup x = 1)$ est satisfaite par l'ensemble

$$\{\ x = 0 : x = 0 : x = 1 : \epsilon : \epsilon : \dots\ \}$$

mais après le tissage d'un aspect adaptateur qui arrête l'exécution si $x = 0$, on obtient l'ensemble

$$\{\ (abort, x = 0) : (\epsilon, x = 0) : (\epsilon, x = 0) : \dots\ \}$$

qui ne satisfait plus la propriété. Pour les événements, seuls les propriétés de vivacité sur $\neg ep$ sont préservées. Par exemple, $\forall(true \cup \neg ep)$ est préservée même si $\neg ep$ sera satisfaite après l'arrêt de l'exécution (*c.-à-d.,* quand la configuration devient $(\epsilon : \bullet, \Sigma)$).

La préservation des propriétés de θ^a par les aspects adaptateurs est exprimée par le Théorème 6.

THÉORÈME **6.**

$$\forall(C,\Sigma).\Sigma^{\psi} \in \mathcal{A}_a^* \Rightarrow$$
$$\forall(p \in \theta^a).\forall(\alpha \in \Gamma).\Gamma, \alpha_1 \models p \Rightarrow \forall(\tilde{\alpha} \in \tilde{\Gamma}).\tilde{\Gamma}, \tilde{\alpha}_1 \models p$$
avec $\Gamma = \mathcal{B}^*(C,\Sigma^b)$ *et* $\tilde{\Gamma} = \mathcal{W}^*(C,\Sigma)$

3.6 Compositions et Interactions d'aspects

La relation d'inclusion qui existe entre nos catégories d'aspect leur confère une caractéristique intéressante : quelque soit l'ordre d'interaction, l'interaction entre deux aspects de nos catégories retourne un comportement qui correspond à celui d'un aspect appartenant à une de nos catégories. En effet, les aspects de nos catégories peuvent être composés et tissés dans n'importe quel ordre et retourner un comportement qui correspond à celui des aspects d'une de nos catégories. Cela nous permet de déterminer de façon déterministe, les propriétés préservées par l'interaction entre les aspects de nos catégories sans regarder le programme tissé. Par exemple, comme les observateurs sont inclus dans les terminateurs, la composition d'un observateur et d'un terminateur quelque soit leur ordre appartient à la catégorie des terminateurs. De même, lorsqu'on compose un terminateur et un verrouilleur quelque soit leur ordre, on obtient un aspect qui est dans la catégorie des verrouilleurs. Lorsqu'on compose un terminateur et un sélecteur on se retrouve dans la catégorie des adaptateurs car les sélecteurs et les terminateurs sont inclus dans les adaptateurs.

En général, comme c'est aussi le cas avec les langages d'aspect généraux, l'interaction entre les aspects de nos catégories reste non commutatif *c.-à-d.,* la sémantique du programme tissé varie en fonction de l'ordre du tissage des aspects. Par exemple, la trace ci-dessous correspond à l'exécution d'un programme de base où une variable x est initialisée à 0, puis incrémentée par la deuxième instruction avant d'être passée à l'instruction foo qui l'incrémente de nouveau.

$$x = 0 : x = 0 : (foo, x = 1) : (\epsilon, x = 2) : (\epsilon, x = 2) : \ldots$$

Considérons que l'instruction foo est filtrée par un aspect observateur et terminateur qui se partagent la variable y initialisée 0 et insèrent des actions juste avant foo. L'action de l'aspect observateur insère une instruction a_1 qui incrémente y. L'action de l'aspect terminateur insère une instruction a_2 qui arrête l'exécution du programme tissé si $y = 1$. Lorsqu'on tisse l'aspect terminateur avant l'aspect observateur on obtient la trace tissée suivante

$$(x = 0, y = 0) : (x = 0, y = 0) : (a_2, x = 1, y = 0) : (a_1, x = 1, y = 0)$$
$$: (foo, x = 1, y = 1) : (\epsilon, x = 2, y = 1) : (\epsilon, x = 2, y = 1) : \ldots$$

qui possède à chaque configuration l'état de la variable y des aspects. Comme à l'exécution de a_2, $y = 0$, l'exécution du programme tissé n'est pas arrêté. Par contre, lorsqu'on tisse l'aspect observateur avant l'aspect terminateur, a_1 incrémente y qui passe de 0 à 1 et entraîne l'arrêt du programme tissé par a_2.

$$(x = 0, y = 0) : (x = 0, y = 0) : (a_1, x = 1, y = 0) : (a_2, x = 1, y = 1)$$

$$: (\epsilon, x = 1, y = 1) : (\epsilon, x = 1, y = 1) : \ldots$$

Par contre, si les instructions des aspects ne sont pas filtrées par d'autres aspects, si tout état d'un aspect n'est pas accédé par d'autres aspects, alors sur les variables du programme de base (Σ^b), un observateur est commutatif avec un aspect de toutes les autres catégories. Un terminateur est commutatif avec un terminateur et un observateur. Dans les autres cas les aspects de nos catégories ne sont pas commutatif. Par exemple, si on se place dans le cas de l'interaction entre un observateur et un terminateur, quelque soit l'ordre du tissage, ils ne modifie pas les variables du programme de base par définition. De plus, comme les aspects ne se partagent pas leurs états et ont des instructions qui ne peuvent pas être filtrées par d'autres aspects, l'arrêt du programme tissé par l'aspect terminateur ne dépend pas de l'ordre du tissage. Donc la sémantique du programme tissé sur les variables du programme de base est la même quelque soit l'ordre du tissage entre observateur et terminateur. Les autres cas s'expliquent par un raisonnement similaire. Le tableau de la Figure 3.1 récapitule les commutativités entre les aspects des catégories. Dans ce tableau, 1 spécifie que le tissage est commutatif et 0 le contraire.

	$\mathcal{A}_o$	$\mathcal{A}_t$	$\mathcal{A}_s$	$\mathcal{A}_r$	$\mathcal{A}_v$	$\mathcal{A}_f$
$\mathcal{A}_o$	1	1	1	1	1	1
$\mathcal{A}_t$	1	1	0	0	0	0
$\mathcal{A}_s$	1	0	0	0	0	0
$\mathcal{A}_r$	1	0	0	0	0	0
$\mathcal{A}_v$	0	0	0	0	0	0
$\mathcal{A}_f$	0	0	0	0	0	0

FIG. 3.1 – Commutativité du tissage des aspects des différentes catégories

3.7 Conclusion

Dans ce chapitre, nous avons utilisé un cadre sémantique, indépendant de tout type de langage de programmation, pour définir formellement plusieurs catégories d'aspect : observateur, terminateur, verrouilleur, et faiblement intrusif. Pour chaque catégorie, nous avons donné un sous ensemble des propriétés de LTL préservé par le tissage, pour tout programme de base, et pour tout aspect de la catégorie correspondante. Cela a été prouvé par induction sur la structure des propriétés de la classe pour la catégorie des observateurs (voir Appendice A.1). La preuve des autres catégories a la même structure que celle des observateurs. Les catégories et classes ci-dessus ont été complétées et généralisées pour des programmes non déterministes en utilisant CTL*. Notre démarche permet de raisonner sur le programme de base et les aspects sans regarder le programme tissé. En effet, pour tout aspect, il suffit de connaître sa catégorie, pour déterminer quelque soit le programme de base, les propriétés qu'il permet de préserver.

Pour illustrer chaque catégorie d'aspect, des exemples ont été fournis. Typiquement, les as-

pects de profilage sont des observateurs ; les aspects qui assurent des politiques de sécurité sont des terminateurs ; les aspects de tolérance aux fautes ou de mémoisation sont soit verrouilleurs, soit faiblement intrusifs en fonction de leur implémentation. Bien sur, plusieurs aspects n'appartiennent pas à nos catégories. Par exemple :

- les aspects sur les exceptions (voir *par ex.,* [LL00]) peuvent être, des observateurs s'ils détectent et enregistrent des erreurs, ou des terminateurs s'ils capturent les erreurs en arrêtant l'exécution du programme (*par ex.,* la programmation par contrat est généralement implémenté comme des terminateurs). Toutefois, la capture des erreurs peut aussi retourner une valeur par défaut ou effectuer une action ou supprimer uniquement une portion de la trace. Dans ces cas, l'exécution du programme pourrait atteindre de nouveaux états et ne préserver en général aucune propriété.
- Les aspects de sécurité peuvent être des observateurs s'ils ne font qu'enregistrer les événements critiques, ou des terminateurs lorsqu'ils assurent une politique de sécurité. Lorsque les aspects sont utilisés pour implémenter des mécanismes de sécurité comme les contrôles d'accès, ils modifient généralement la sémantique du programme de base (*par ex.,* l'exécution change en fonction des droits d'accès).
- Les transformations de programme (optimisation) pourraient être vues comme des aspects préservant la sémantique. Elles préservent les propriétés sur l'état final. Comme elle peuvent changer complètement l'algorithme (*c.-à-d.,* la trace d'exécution) des programmes, elles n'appartiennent pas à nos catégories. Elles peuvent violer des propriétés (*par ex.,* sûreté) temporelles importantes. Une nouvelle catégorie préservant le résultat pourrait donc être introduite. Cependant, la classe de propriétés préservées est triviale (les propriétés sur les états du résultat final) et il semble difficile par construction ou par analyse statique d'assurer que les aspects appartiennent à cette catégorie.

Le point de départ de ce travail fut l'article de Katz [Kat06] (voir Section 1.3.3, page 31) qui introduit les catégories des spectateurs (similaires aux observateurs), des régulateurs (similaires aux terminateurs et adaptateurs) et des aspects faiblement invasifs (similaires aux faiblement intrusifs). Pour chaque catégorie, Katz indique les classes de propriétés standards (sûreté, vivacité et invariants) qui sont préservées. Cependant, cette étude est informelle. Les catégories d'aspect, les classes de propriété et les preuves ne sont pas formalisées, les définitions et résultat étant peu précis. Par exemple, les propositions atomiques sur les états (sp) et sur les événements (ep) ne sont pas clairement distinguées. Katz affirme que les aspects spectateurs préservent les propriétés de sûreté. Notre étude montre que les observateurs (spectateurs) préservent uniquement les propriétés de sûreté impliquant les propriétés sur les états (pas sur les événements). Katz énonce aussi que, les régulateurs (terminateurs) préservent les propriétés de sûreté mais pas de vivacité. Notre étude confirme cela, mais uniquement quand les propriétés impliquent exclusivement les propositions atomiques sur les états. D'un autre côté, nous montrons que les terminateurs ne préservent pas les propriétés de sûreté sur les événements, mais préservent les propriétés de vivacité impliquant uniquement les propositions atomiques sur la négation des événements ($\neg ep$).

Comme présenté au Chapitre 1 (voir Sections 1.3.2 et 1.3.3), Dantas et Walker [DW06] décrivent formellement une catégorie d'aspect dénommée inoffensif. Cette catégorie correspond à nos terminateurs. Ils proposent un système de type qui assure que les aspects inoffensifs ne modifient pas le résultat final du programme de base si l'exécution du programme tissé n'est

pas arrêtée par les aspects.

Krishnamurthi et coll. [KFG04] se focalisent sur les aspects qui retournent toujours au point du programme de base qui suit directement le point de jonction. Ils proposent une technique de vérification modulaire qui génère à partir du programme de base des interfaces qui vont permettre de vérifier uniquement l'aspect afin de déterminer qu'une propriété donnée du programme de base est préservée par le tissage. Ainsi, chaque aspect doit être analysé, contrairement à notre approche qui considère des catégories d'aspect. A chaque modification, l'aspect doit être de nouveau vérifié tandis que notre approche nécessite uniquement de déterminer sa catégorie pour connaitre les propriétés qu'il préserve. Le travail de Krishnamurthi et coll. a été étendu par Goldman et Katz à la catégorie des aspects faiblement invasifs (faiblement intrusifs).

Rinard, Salcianu et Bugara [RSB04] proposent des catégories d'aspect basées sur une classification informelle de leurs interactions avec le programme de base. Ils distinguent deux classes. La première traite des modifications du flot de contrôle : un aspect d'*augmentation* ne modifie pas le flot de contrôle ; un aspect de *restriction* peut supprimer des fonctions sélectionnées par les points de coupure ; un aspect de *remplacement* peut remplacer une fonction sélectionnée par un point de coupure par une autre ; un aspect de *combinaison* combine la fonction sélectionnée avec son action pour produire l'action courante. La seconde classe traite de la modification de l'état : un aspect *indépendant* ou la fonction qu'il sélectionne ne peut pas modifier une variable qui est lue ou écrite par d'autres aspects ou fonctions ; un aspect d'*observation* peut lire une variable que la fonction sélectionnée modifie ; un aspect d'*incitation* peut modifier une variable que la fonction sélectionnée lit ; un aspect d'*interférence* peut modifier une variable qui est aussi modifiée par la fonction sélectionnée. Ces catégories permettent d'avoir une intuition des impacts des aspects mais la préservation des propriétés n'est pas considérée par les auteurs. Les aspects d'augmentation et indépendants ressemblent aux observateurs. Les autres catégories peuvent modifier arbitrairement la sémantique du programme de base. Clifton et Leavens [RSB04] proposent deux catégories : des *observateurs* et des *assistants*. Comme les nôtres, les observateurs ne modifient pas la spécification du programme de base tandis que les assistants la modifient. Mais à partir de leurs exemples, les assistants semblent similaires aux terminateurs. Les auteurs utilisent la logique de Hoare pour expliquer le comportement du programme tissé. Les catégories elles mêmes, ne sont pas formalisées.

Pour compléter ce travail, nous introduisons au chapitre suivant des langages qui garantissent par construction qu'un aspect appartient à une catégorie.

Chapitre 4

Langages associés aux catégories d'aspect

4.1 Introduction

Au chapitre précédent, nous avons considéré plusieurs catégories d'aspect qui ont un impact particulier sur la sémantique du programme de base. Pour chaque catégorie d'aspects $\mathcal{A}_x$, nous avons identifié une classe de propriétés φ^x qui lui correspond et qui est préservée par le tissage des aspects. En d'autres termes, soit P un programme de base qui satisfait une propriété $\varphi \in \varphi^x$, alors tisser tout aspect $A \in \mathcal{A}_x$ sur P produira un programme satisfaisant φ. Il suffira donc, pour chaque aspect, de connaître sa catégorie, pour déterminer son effet sur le programme de base.

Dans ce chapitre, nous complétons cette étude en présentant des langages d'aspect spécialisés qui assurent que tout aspect appartient à une catégorie donnée. Les catégories que nous traitons pour le moment sont celles des observateurs, des terminateurs et des verrouilleurs. Pour les programmes non déterministes (concurrents), nous nous contentons de présenter quelques caractéristiques importantes des langages correspondants.

Cette démarche, qui garantit par construction l'appartenance d'un aspect à une catégorie, fait que nous devons fixer un langage de base afin de spécifier et d'assurer toutes les contraintes que les aspects doivent respecter. Nos langages d'aspect sont conçus pour un langage de base impératif simple et sans pointeurs. Cela pourrait être fait pour d'autres types de langages de base. Le choix de ce type de langage est dû au fait qu'il possède les points de jonction usuels (appels de procédures, accès aux variables, *etc.*) des langages d'aspect généraux sans la complexité des objets (*par ex.,* la présence d'alias). Les langages d'aspect partagent un même langage de point de coupure très expressif qui peut filtrer toute instruction du langage de base. Nous illustrons chaque langage par des exemples simples. Nous prouvons que tout aspect écrit avec le langage des observateurs est dans la catégorie des observateurs. Pour les terminateurs la structure de la preuve est similaire, tandis que pour les verrouilleurs elle est faite par définition du langage. Ces résultats ont été publiés à la conférence internationale SEFM08 [DDDF08b].

Ce chapitre est organisé comme suit. La Section 4.2 introduit le langage de base impératif, le langage de point de coupure qui lui est associé est décrit en 4.3 et les trois langages d'aspect correspondant aux observateurs, terminateurs et verrouilleurs sont présentés aux Sections 4.4, 4.5 et 4.6 respectivement. La Section 4.7 propose des pistes de réflexion pour la conception des langages d'aspect pour des programmes concurrents. La Section 4.8 conclut. La preuve d'appartenance des aspects du langage des observateurs à la catégorie correspondante se trouve dans l'appendice B.3.

4.2 Le Langage de Base

La syntaxe du langage de base est décrite par la Grammaire 9. Un programme $Prog$ est une séquence D de déclarations de variables globales (g) ou de procédures suivie d'une commande principale S. A côté des commandes usuelles (affectation, appel de procédure, séquence, conditionnelle, boucle `while`), l'instruction `abort` termine l'exécution du programme, `skip` ne fait rien et `loop(`A`)` S exécute A fois la commande S. Les expressions arithmétiques et booléennes sont décrites respectivement par les non terminaux A et B. Nous distinguons deux types de variable :

- les variables globales (g) qui sont déclarées dans D ;
- les variables locales (l) comme paramètres des procédures.

Ces deux types de variables peuvent être utilisés dans les affectations et expressions.

GRAMMAIRE **9.**

$$
\begin{array}{lcl}
Prog & ::= & D\ S \\
D & ::= & \texttt{var}\ g\ \texttt{:=}\ A \mid \texttt{proc}\ I\ \texttt{(}l_1\texttt{,}\ \ldots\texttt{,}l_n\texttt{)}\ S \mid D_1\ \texttt{;}\ D_2 \\
S & ::= & V\ \texttt{:=}\ A \mid I\ \texttt{(}A_1\texttt{,}\ \ldots\texttt{,}A_n\texttt{)} \mid S_1\ \texttt{;}\ S_2 \mid \\
 & & \texttt{if(}B\texttt{)}\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while(}B\texttt{)}\ S \mid \\
 & & \texttt{abort} \mid \texttt{skip} \mid \texttt{loop(}A\texttt{)}\ S \\
A & ::= & n \mid V \mid A_1 + A_2 \\
B & ::= & \mathit{true} \mid A_1\ \texttt{=}\ A_2 \mid A_1\ \texttt{<}\ A_2 \mid B_1\ \texttt{\&}\ B_2 \mid \texttt{!}B \\
V & ::= & g \mid l \\
I & ::= & p
\end{array}
$$

Notons que comme toutes les variables sont des entiers (n), nous n'utilisons pas de types. Cependant, le langage peut être facilement étendu avec d'autres types de base et des types construits. Comme requis par la CASB (Section 2.2, page 42), la sémantique est définie par une relation $\rightarrow_b$ sur des configurations (C, Σ) où C est une séquence de commandes S et Σ^b est composé d'environnements associant des variables globales et paramètres à leurs valeurs et d'une pile de retour pour les appels de procédures. La sémantique opérationnelle de ce langage est similaire à celle du langage *While* de Nielson et Nielson [NN92]. Une définition complète de cette sémantique se trouve à l'Appendice B.1. L'exemple 19 illustre notre langage avec un programme simple qui sera utilisé tout au long du chapitre.

EXEMPLE **19.** *Un programme qui calcule le quatrième élément de la suite de fibonacci peut s'écrire comme suit :*

```
var result := 0;
proc fib(x)
      if(x=0) then result := result + 1 else
      if(x=1) then result := result + 1 else
      fib(x-1); fib(x-2)
fib(4)
```

4.3 Langage de point de coupure générique

Nos langages d'aspect partagent le même langage de point de coupure qui est défini par la Grammaire 10.

GRAMMAIRE **10.**

$$
\begin{array}{lcl}
P & ::= & S^p \mid if(B^p) \mid P_1 \vee P_2 \mid P_1 \wedge P_2 \\
S^p & ::= & V^p \texttt{ := } A^p \mid I^p(A^p_1, \ldots, A^p_n) \mid S^p_1 \texttt{ ; } S^p_2 \mid \\
 & & \texttt{if(}B^p\texttt{) then } S^p_1 \texttt{ else } S^p_2 \mid \texttt{while(}B^p\texttt{) } S^p \mid \\
 & & \texttt{abort} \mid \texttt{skip} \mid \texttt{loop(}A^p\texttt{) } S^p \mid \beta_s \mid \neg S^p \\
A^p & ::= & n \mid V^p \mid A^p_1 + A^p_2 \mid \beta_A \mid \neg A^p \\
B^p & ::= & true \mid A^p_1 \texttt{ = } A^p_2 \mid A^p_1 \texttt{ < } A^p_2 \mid B^p_1 \texttt{ \& } B^p_2 \mid \texttt{!}B \mid \\
 & & \beta_B \mid \neg B^p \\
V^p & ::= & g \mid l \mid \beta_V \mid \neg V^p \\
I^p & ::= & p \mid \beta_I \mid \neg I^p
\end{array}
$$

Un point de coupure est soit une commande S^p avec des variables de motif (point de coupure statique), soit un prédicat $if(B^p)$ (point de coupure dynamique), ou une composition logique des points de coupure. Un motif permet pour chaque catégorie syntaxique qui le compose (expressions, variables, *etc.*), l'utilisation des variables de motif β_x ainsi que des motifs ayant une négation (*par ex.,* $\neg S$). Par exemple, A^p définit des motifs sur les expressions arithmétiques et leurs négations avec β_A comme variable de motif (capable de filtrer toute expression arithmétique). Le non terminal I^p définit des motifs sur les identificateurs de procédures. Le filtrage de motif S^p par rapport à une configuration $(i : C, \Sigma)$ affecte des valeurs à des variables de motif β_S, β_A, *etc.* Ces valeurs vont substituer les occurrences de ces variables dans les points de coupure dynamiques ainsi que dans l'action. La sémantique des motifs avec des négations (appelée *anti-patterns*) tels que utilisés dans la Grammaire 10 est décrite en détail dans [KKM07].

Les points de coupure dynamiques $if(b)$ doivent représenter des expressions booléennes après la substitution d'éventuelles variables de motif par leurs valeurs. Pour assurer cette pro-

priété, nous imposons que le motif $\neg B^p$ des expressions booléennes n'apparaisse pas dans les points de coupure dynamiques. En effet, $if(b)$ est utilisé pour tester si l'expression b est satisfaite à un point du programme. L'expression b, qui peut contenir des variables de motif, doit donc être une expression booléenne après la substitution de ses variables de motif. Cependant par souci de régularité et de simplicité, notre grammaire permet l'utilisation d'un motif de la forme $\neg B^p$ dans la construction if. Cette utilisation ne garantit pas d'obtenir une expression booléenne après la substitution des variables de motif. Par exemple, $p(\beta_A) \wedge if(\neg\beta_A)$ est un point de coupure permis par notre grammaire. Ce point de coupure filtre par exemple le point de jonction $p(3)$ en associant 3 à la variable de motif β_A. Mais après substitution on obtient $if(\neg 3)$ où $\neg 3$ n'est pas une expression booléenne mais un motif. D'autre part, comme présenté à la Section 2.5, page 51, un motif avec une négation peut filtrer une instruction selon plusieurs substitutions. Par exemple le point de coupure $\neg p(\beta_A)$ qui filtre toutes les instructions qui ne sont pas des appels de méthode, filtre l'affectation x `:=` 3 avec une infinité de valeurs possible pour β_A. Une telle variable de motif β_A ne doit pas apparaître dans une action ou dans un point de coupure dynamique. Donc, les variables de motifs utilisées dans les points de coupure dynamiques et dans les actions, ne doivent pas être dans la portée d'une négation d'un point de coupure statique. Cela permet de garantir une substitution unique et ainsi obtenir du code exécutable tant pour les prédicats $if(b)$ que pour les actions.

EXEMPLE **20.** *Les exemples de motif ci-dessous illustrent les principales caractéristiques de ce langage :*

- x `:=` β_A *filtre toutes les affectations à* `x` *;*
- $(\neg x)$ `:=` β_A *filtre toutes les affectations mais pas celles à* x *;*
- $\neg(x$ `:=` $y)$ *filtre toutes les commandes mais pas* x `:=` y *;*
- `while (`β_B`)` β_S *filtre toutes les commandes* `while` *;*
- p `(`$3, \beta_A$`)` $\wedge$ `if(`β_A `=` $0)$ *filtre tous les appels à* p *avec deux paramètres dont le premier est* 3 *et le second est une expression arithmétique qui doit être égal à* 0.

4.4 Le langage des observateurs

Nous définissons ici un langage d'aspect qui assure que tout aspect défini avec ce langage est un observateur. Comme vu à la Section 3.4.1 (page 61), un observateur ne modifie pas le flot de contrôle du programme de base mais insère uniquement des instructions ne modifiant pas l'état de base (i_a). Afin d'être compatible avec AspectJ et la majorité des langages à aspects, nous considérons des aspects `around` dont l'action est composée d'instructions i_a avec une unique commande `proceed` pour exécuter l'instruction filtrée. Quand l'action termine son exécution, le programme de base continue à l'instruction qui suit celle filtrée.

Notre instruction `proceed` n'a pas de paramètre car dans le cas contraire, l'aspect pourrait modifier les paramètres des procédures et changer arbitrairement le flot de contrôle ou l'état du programme de base. L'action doit également terminer, sinon le programme de base ne peut poursuivre son exécution et son flot de contrôle est donc modifié. Nous assurons cette terminaison en empêchant l'emploi de la commande `while` dans l'action, en vérifiant qu'il n'existe

aucun cycle dans le graphe d'appel des procédures de l'action et en assurant que tout point de coupure ne puisse pas filtrer les instructions de son action. Une autre option serait de lever ces restrictions et de laisser ainsi le programmeur assurer cette terminaison. La préservation serait garantie modulo la terminaison des actions.

La seconde condition qu'un observateur doit respecter est de ne pas modifier l'état du programme de base (*c.-à-d.,* les instructions i_a ne changent pas Σ^b). Pour cela, le langage distingue les variables du programme de base (qui peuvent être lues par l'action) et les variables de l'aspect (qui peuvent être lues et écrites par i_a).

La sémantique de `proceed` est exprimée à l'aide d'une pile proceed (Σ^P) dans l'état global Σ. Lorsqu'un aspect `around` est tissé, l'instruction filtrée est placée au sommet de cette pile (voir Section 2.3.3, page 46). La commande `proceed` dépile et exécute l'instruction en sommet de pile proceed. Comme ici, l'action possède exactement un `proceed`, la Règle PROCEED (voir page 46) est simplifiée en supprimant l'instruction sémantique $push_p$. Cette instruction permettait de replacer l'instruction filtrée au sommet de la pile proceed afin qu'elle puisse être réexécutée par un autre `proceed` de l'action.

$$\textsc{Proceed}\ \frac{\Sigma^P = i : \Sigma'^P}{(\texttt{proceed} : C, X \cup \Sigma^P) \rightarrow (i : C, X \cup \Sigma'^P)}$$

La syntaxe des observateurs est définie par la Grammaire 11.

GRAMMAIRE **11.**

$$\begin{array}{lcl}
Asp^o & ::= & D^o\ \texttt{around}\ P\ \{S_1^o\ \texttt{;}\ \texttt{proceed}\ \texttt{;}\ S_2^o\} \\
D^o & ::= & \texttt{var}\ g^o\ \texttt{:=}\ A^o \mid \texttt{proc}\ p^o(l_1^o, \ldots, l_n^o)\ S^o \mid \\
 & & D_1^o; D_2^o \\
S^o & ::= & V^o\ \texttt{:=}\ A^o \mid p^o(A_1^o, \ldots, A_n^o) \mid S_1^o; S_2^o \mid \texttt{skip} \mid \\
 & & \texttt{if(}B^o\texttt{)}\ \texttt{then}\ S_1^o\ \texttt{else}\ S_2^o \mid \texttt{loop(}A^o\texttt{)}\ S^o \\
A^o & ::= & n \mid V' \mid A_1^o + A_2^o \mid \beta_A \\
B^o & ::= & \mathit{true} \mid A_1^o\ \texttt{=}\ A_2^o \mid A_1^o\ \texttt{<}\ A_2^o \mid B_1^o\ \texttt{\&}\ B_2^o \mid \\
 & & \texttt{!}B^o \mid \beta_B \\
V^o & ::= & g^o \mid l^o \\
V' & ::= & V^o \mid g \mid \beta_V
\end{array}$$

Un observateur Asp^o définit des variables g^o et des procédures p^o qui forme son état local. C'est un aspect du type `around` qui associe un point de coupure avec une action qui contient exactement un `proceed`. Pour simplifier la présentation, nous avons considéré que l'aspect a un point de coupure et une action mais cela peut être facilement généralisé à plusieurs points de coupure et actions. Les déclarations D^o ne doivent pas contenir de variables de motif car les valeurs étant associées aux variables de motif par l'intermédiaire des points de coupure, s'il existe de variables de motif utilisées dans les déclarations qui n'apparaissent pas dans les points de coupure, celles-ci se retrouveront sans valeur après la substitution. Par exemple, si

`var` $x := \beta_{\text{A}}$ est une déclaration alors β_{A} se retrouvera sans valeur après la substitution s'il n'apparait pas dans le point de coupure. Les autres instructions S^o sont similaires aux motifs S^p mais sans la négation $\neg$. En effet, une action doit être un code exécutable valide après la substitution de ses variables de motif (β_{A}, β_{B}, β_{V}). Remarquons que l'instruction `abort` ne peut pas être employée dans l'action car elle changerait le flot de contrôle du programme de base. De façon similaire, la variable de motif pour les commandes β_{S} ne peut pas être utilisée car elle pourrait filtrer et exécuter des affectations aux variables du programme de base. Les affectations dans l'action modifient uniquement les variables de l'aspect (V^o). Bien sûr les variables de l'aspect comme celles du programme de base (V') peuvent être lues. Enfin, une action ne peut appeler que des procédures définies (p^o) dans l'aspect car appeler une procédure du programme de base modifierait le flot de contrôle (et potentiellement l'état) du programme de base.

Un aspect qui compte les appels à `fib` (Exemple 19) est défini par l'Exemple 21. Cet aspect de profilage respecte la grammaire Asp^o et est donc un observateur

EXEMPLE **21.** *L'aspect suivant enregistre dans la variable* `n` *le nombre d'appels à* `fib`*.*

```
var n:=0;
around fib(βA) {n:=n+1; proceed; skip}
```

Il filtre tout appel à `fib` *en utilisant le point de coupure* `fib`(β_{A})*. Lorsqu'un appel à* `fib` *est filtré, l'action incrémente* `n` *ensuite exécute l'appel à* `fib` *(par l'intermédiaire de* `proceed`*) et termine par* `skip` *qui ne fait rien.*

La sémantique du tissage (Section 2.3, page 44) représente un aspect comme une fonction Σ^ψ qui prend la configuration courante (C, Σ) en paramètre et retourne soit une nouvelle configuration (C', Σ'), soit nil quand le point de coupure ne filtre pas l'instruction courante. Nous définissons la sémantique de notre langage d'aspect de façon à générer Σ^ψ à partir du code de l'aspect. La fonction résultante prend la configuration courante en paramètre et filtre la première instruction i.

$$
\begin{array}{l}
[\![\texttt{around } p\ s]\!] = \\
\quad \mathit{let}\ (p_1 \wedge \mathit{if}(b_1)) \vee \ldots \vee (p_n \wedge \mathit{if}(b_n)) = \mathit{Transf}(p)\ \mathit{in} \\
\quad \lambda(i : C, X \cup \Sigma^P). \\
\quad\quad \mathit{case} \quad \mathit{match}^s(p_1, i) = \sigma_1 \quad \mapsto \quad (\overline{a_1} : C, X \cup \bar{i} : \Sigma^P) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \cdots \\
\quad\quad\quad\quad\quad\quad \mathit{match}^s(p_n, i) = \sigma_n \quad \mapsto \quad (\overline{a_n} : C, X \cup \bar{i} : \Sigma^P) \\
\quad\quad\quad\quad\quad\quad \mathit{else} \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mapsto \quad \mathit{nil} \\
\quad \text{où } a_i = \sigma_i(\texttt{if}\,(b_i)\ \texttt{then}\ s\ \texttt{else proceed})
\end{array}
$$

L'instruction $\bar{i}$ et le code $\bar{a_i}$ sont tagués (voir Section 2.3, page 44) pour empêcher un tissage infini. Afin de distinguer sa partie statique de sa partie dynamique, la fonction Transf transforme tout point de coupure p en une disjonction de la forme $(p_1 \wedge \mathit{if}(b_1)) \vee \ldots \vee (p_n \wedge \mathit{if}(b_n))$ où les p_j sont mutuellement exclusifs. Elle procède de la manière suivante :

– le point de coupure p est mis sous une forme normale disjonctive $p'_1 \vee \ldots \vee p'_n$;
– chaque p'_j est alors converti sous la forme $p''_j \wedge \texttt{if}(b_j)$ tel que p''_j soit un point de coupure statique. On le fait en utilisant l'associativité et la commutativité de l'opérateur $\wedge$ et la règle suivante

$$\texttt{if}(b_1) \wedge \texttt{if}(b_2) \equiv \texttt{if}(b_1 \wedge b_2)$$

si p'_j est uniquement statique ou conditionnel, il peut toujours être converti avec les règles :

$$\texttt{if}(b) \equiv \beta_{\mathrm{s}} \wedge \texttt{if}(b) \qquad S^p \equiv S^p \wedge \texttt{if}(true)$$

– l'aspect maintenant de la forme

$$\texttt{around}\,((p''_1 \wedge \texttt{if}(b_1)) \vee \ldots \vee (p''_n \wedge \texttt{if}(b_n)))\; s$$

peut être transformé en plusieurs aspects de la forme $\texttt{around}\ (p_j \wedge \texttt{if}(b_j))\ s$. Par exemple, l'aspect

$$\texttt{around}\,((p_1 \wedge \texttt{if}(b_1)) \vee (p_2 \wedge \texttt{if}(b_2)))\; s$$

est converti en trois aspects

$$\begin{array}{l}\texttt{around}\,(p_1 \wedge \neg p_2 \wedge \texttt{if}(b_1))\; s\\ \texttt{around}\,(p_2 \wedge \neg p_1 \wedge \texttt{if}(b_2))\; s\\ \texttt{around}\,(p_1 \wedge p_2 \wedge \texttt{if}(b_1 \vee b_2))\; s\end{array}$$

Ces aspects sont mutuellement exclusifs (*c.-à-d.,* au plus un est appliqué). Ils peuvent être vus comme un aspect de la forme

$$\begin{array}{l}\texttt{around}\,((p_1 \wedge \neg p_2 \wedge \texttt{if}(b_1))\\ \qquad\qquad \vee (p_2 \wedge \neg p_1 \wedge \texttt{if}(b_2))\\ \qquad\qquad \vee (p_1 \wedge p_2 \wedge \texttt{if}(b_1 \vee b_2)))\; s\end{array}$$

où les points de coupure statiques sont mutuellement exclusifs. Cette transformation se généralise à n disjonctions. Si les preuves des premières transformations sont claires et standards, la preuve du passage d'un aspect à plusieurs aspects mutuellement exclusifs est faite à la Section 5.3.3, page 110.

La fonction Σ^{ψ} teste donc si l'instruction courante i est filtrée par un des points de coupure statique p_j. Si i n'est pas filtrée, la fonction retourne nil. Sinon, i est remplacée par un code a et placée au sommet de la pile proceed. Lorsqu'il est exécuté, a teste la partie dynamique b_j du point de coupure statique p_j qui a filtré i. Si b_j est satisfaite, l'action s est exécutée. Sinon, l'exécution continue avec l'instruction i (l'action n'est pas exécutée). Les variables de motifs dans b_j et s sont remplacées par leurs valeurs en utilisant la substitution σ retournée par $match^s$ qui est une surcharge de la fonction $match$ (définie à la Section 2.5, page 51) sur les *anti-patterns*.

Cette sémantique distingue l'évaluation de la partie statique du point de coupure de celle de la partie dynamique. Nous avons fait ce choix afin de modéliser les langages à la AspectJ où plusieurs aspects peuvent interagir (*c.-à-d.,* la partie dynamique d'un point de coupure peut dépendre de l'exécution de l'action précédente).

La propriété 22 formalise le fait que tout aspect dans Asp^o est un observateur.

PROPRIÉTÉ **22.** $\forall a \in Asp^o.[\![a]\!] \in \mathcal{A}_o$

En effet, comme les commandes de Asp^o modifient uniquement les variables déclarées par l'aspect, une réduction de toute commande de Asp^o par $\rightarrow_b$ modifie uniquement Σ^a (*c.-à-d.,* l'aspect ne modifie pas l'état du programme de base). De plus comme tout aspect de Asp^o possède un `proceed` sans paramètres et appelle uniquement ses méthodes, alors la sémantique de `proceed` ci-dessus et $\rightarrow_b$ garantissent que la séquence d'instructions de base dans la trace tissée est la même que la trace d'instructions du programme de base (*c.-à-d.,* l'aspect ne modifie pas le flot de contrôle du programme de base). Une preuve de cette propriété est présentée à l'Appendice B.3, page 128. Cette preuve est une induction sur la longueur de la trace de base.

4.5 Le langage des terminateurs

Un terminateur est un observateur qui peut terminer l'exécution du programme. Le langage des terminateurs est donc similaire à celui des observateurs. Sa grammaire Asp^t est exprimée exactement comme Asp^o, excepté la commande `abort` qui apparaît dans S^t. L'instruction `abort` réduit toute configuration en une configuration finale (voir Section 3.4.2, page 63).

L'Exemple 23 spécifie un aspect comptant le nombre d'appels à la procédure `fib` (Exemple 19). Si le nombre d'appels atteint 100.000, le programme est terminé par `abort`. Cet aspect peut être utilisé pour garantir un délai d'exécution. Il est défini dans Asp^t et c'est donc un terminateur.

EXEMPLE **23.** *L'aspect suivant limite les appels à* `fib` *à 100.000 au maximum.*

```
var nbCalls:= 0;
around (fib(β_A)) {
  nbCalls:=nbCalls+1;
  if (nbCalls=100000) then abort else skip;
  proceed; skip
}
```

La propriété 24 spécifie que tout aspect dans Asp^t est un terminateur.

PROPRIÉTÉ **24.** $\forall a \in Asp^t.[\![a]\!] \in \mathcal{A}_t$

La preuve de cette propriété est presque identique à celle du langage des aspects observateurs. La seule différence est qu'ici, la séquence d'instructions de base dans la trace tissée peut être un préfixe de la trace d'instructions du programme de base car l'aspect peut arrêter l'exécution du programme tissé par l'intermédiaire de l'instruction `abort`.

4.6 Langages de verrouilleurs

Les verrouilleurs peuvent arbitrairement modifier le flot de contrôle et l'état du programme aussi longtemps que cet état reste dans l'ensemble des états atteignables par le programme de base. Il est difficile de concevoir un langage général qui assure cette propriété car il doit permettre aux aspects d'exprimer différents comportements sur les variables du programme de base sans permettre à celles-ci de sortir de l'ensemble des états atteignables du programme de base. Cependant, deux langages spécialisés des verrouilleurs viennent à l'esprit :

- un langage d'optimisation dont les actions permettraient d'aller directement vers des états futurs atteignables ;
- un langage de tolérance aux fautes dont les actions retourneraient vers des états sûrs précédemment atteignables.

Ici, nous proposons un langage spécialisé pour définir des aspects `memo`. Un aspect `memo` est un aspect d'optimisation qui sauvegarde et restaure dans un cache des résultats de calcul. Lorsqu'un calcul à sauvegarder est effectué pour la première fois, l'aspect sauvegarde les arguments et les résultats de celui-ci. Quand ce calcul doit à nouveau être effectué, l'aspect court-circuite son exécution et retourne directement les résultats sauvegardés précédemment. La Grammaire 12 présente la syntaxe de ce langage.

GRAMMAIRE **12.**

$$\begin{array}{lcl} Asp^m & ::= & memo\ (I^m(A^p_1, \ldots, A^p_n) \wedge if(B^o)) \\ I^m & ::= & p \mid \beta_l \end{array}$$

Un aspect `memo` se résume à une primitive `memo` qui sauvegarde et restaure les résultats des appels de procédure. Ces appels de procédure sont filtrés par la partie statique du point de coupure qui lui est passé en paramètre si le prédicat de la partie dynamique est vrai.

Pour donner la sémantique des aspects `memo`, nous avons besoin des listes des variables lues et écrites par une procédure. Ces deux listes sont calculées par les fonctions $read$ et $write$. La fonction $read$ prend les déclarations D en paramètre et retourne une fonction qui associe à chaque nom de procédure sa liste de variables lues. Pour effectuer cette tache, $read$ parcours la séquence des déclarations $(d_1; d_2)$ et retourne l'union $(\uplus)$ des fonctions retournées par chaque déclaration. Lorsqu'une déclaration est une procédure, $read$ retourne une fonction qui associe au nom de la procédure (p) la liste (construite par la fonction $mklist$) des variables lues par son corps (s). Lorsqu'une déclaration est une variable, $read$ retourne la fonction vide $(\perp)$.

$$\begin{array}{lcl} read : Decl \rightarrow Identifier \rightarrow List(\mathcal{V}ar) & & \\ read[\![d_1\ \texttt{;}\ d_2]\!] & = & read[\![d_1]\!] \uplus read[\![d_2]\!] \\ read[\![\texttt{proc}\ p\ \texttt{(...)}\ s]\!] & = & [p \mapsto mklist(read_{\mathrm{s}}[\![s]\!]\{p\})] \\ read[\![\texttt{var}\ g\ \texttt{:=}\ a]\!] & = & \perp \end{array}$$

L'ensemble des variables lues est calculé par la fonction $read_{\mathrm{s}}$ qui prend en paramètres le corps de la procédure ainsi que l'ensemble des procédures déjà analysées (pour assurer la

terminaison de l'analyse).

$$
\begin{array}{lcl}
read_{\mathrm{s}} : \mathit{Statement} \to \mathbb{P}(\mathit{Identifier}) \to \mathbb{P}(\mathcal{V}ar) & & \\
read_{\mathrm{s}}[\![S_1 \texttt{ ; } S_2]\!]ps & = & read_{\mathrm{s}}[\![S_1]\!]ps \cup read_{\mathrm{s}}[\![S_2]\!]ps \\
read_{\mathrm{s}}[\![g \texttt{ := } a]\!]ps & = & read_{\mathrm{A}}[\![a]\!] \\
read_{\mathrm{s}}[\![\texttt{while(}b\texttt{) } s]\!]ps & = & read_{\mathrm{B}}[\![b]\!] \cup read_{\mathrm{s}}[\![s]\!]ps \\
read_{\mathrm{s}}[\![\texttt{loop(}a\texttt{) } s]\!]ps & = & read_{\mathrm{A}}[\![a]\!] \cup read_{\mathrm{s}}[\![s]\!]ps \\
read_{\mathrm{s}}[\![\texttt{if(}b\texttt{) then } s_1 \texttt{ else } s_2]\!]ps & = & read_{\mathrm{B}}[\![b]\!] \cup read_{\mathrm{s}}[\![s_1]\!]ps \cup read_{\mathrm{s}}[\![s_2]\!]ps \\
read_{\mathrm{s}}[\![p(a_1, \ldots, a_n)]\!]ps & = & read_{\mathrm{s}}[\![\texttt{body}(p)]\!](ps \cup \{p\}) \\
 & & \cup_{i=1\ldots n}\ read_{\mathrm{A}}[\![a_i]\!]\ si\ p \notin ps \\
read_{\mathrm{s}}[\![\_]\!]ps & = & \emptyset \quad \text{sinon}
\end{array}
$$

Les variables lues par une séquence de deux commandes sont l'union des variables lues par la première et la seconde commande. Les variables lues par une affectation sont les variables qui apparaissent dans la partie droite a. Ces variables sont collectées par la fonction $read_{\mathrm{A}}$ (définie dans l'Appendice B.2). Les variables lues par une boucle `while` (resp. `loop`) sont des variables lues par la condition b (resp. l'expression a) et les variables lues par les corps de ces boucles. Les variables lues par l'instruction conditionnelle `if` sont les variables lues par la condition et celles lues dans les deux branches. Enfin, les variables lues lorsqu'une procédure est appelée sont celles lues par le corps de la procédure. Le second argument de $read_{\mathrm{s}}$ (ps) enregistre les procédures analysées afin de déplier chaque procédure une seule fois. Dans les autre cas, tels que les commandes `skip` et `abort` ou un appel à une procédure déjà analysée, $read_{\mathrm{s}}$ retourne l'ensemble vide.

La fonction $write$ prend les déclarations D en paramètre et retourne une fonction qui associe à chaque procédure sa liste de variables écrites. Sa définition est similaire à $read$ (voir l'Appendice B.2).

Nous pouvons maintenant définir la sémantique d'un aspect `memo` comme une transformation de programme prenant en entrée l'aspect et les déclarations (D) du programme de base :

$$
\begin{array}{l}
T[\![memo(I^m(a_1,\ \ldots,\ a_n)\ \wedge\ if(B^o))]\!]D\ = \\
\quad \{\texttt{var cache := empty;} \\
\quad\quad \{\texttt{around(}I^m(a_1,\ \ldots,\ a_n)\ \wedge\ if(B^o)\texttt{)} \\
\quad\quad\quad \texttt{if contain(}I^m\texttt{,}\ a_1\texttt{:...:}a_n\texttt{,}\ read[\![D]\!]I^m\texttt{)} \\
\quad\quad\quad \texttt{then } write[\![D]\!]I^m\ \texttt{:=} \\
\quad\quad\quad \texttt{lookup(}I^m\texttt{,}\ a_1\texttt{:...:}a_n\texttt{,}\ read[\![D]\!]I^m\texttt{)} \\
\quad\quad\quad \texttt{else proceed; store(}I^m\texttt{,}\ a_1\texttt{:...:}a_n\texttt{,}\ read[\![D]\!]I^m\texttt{,}\ write[\![D]\!]I^m\texttt{)} \\
\quad \}\}
\end{array}
$$

Notons que, par souci de concision, nous avons défini l'action avec le langage de base étendu avec des structures de données (*c.-à-d.,* un cache implémentant une table de hachage, et des listes pour représenter les valeurs des variables lues et écrites) et une valeur de retour pour les procédures (*par ex.,* `contain`, `lookup`).

Un aspect `memo` défini initialement un cache vide qui va contenir les résultats sauvegardés. Une entrée du cache associe un triplet ($\texttt{contain}(I^m, a_1 : \ldots : a_n, read[\![D]\!]I^m)$) composé du nom d'une procédure, la liste de ses arguments et de ses variables lues, à la liste des valeurs des variables écrites par celle-ci $write[\![D]\!]I^m$.

Lorsque l'instruction courante est filtrée par le point de coupure, la substitution retournée σ est appliquée à l'action ce qui instancie complètement la procédure, ses arguments, ainsi que les listes des variables lues ($read[\![D]\!]I^m$) et écrites ($write[\![D]\!]I^m$). Quand l'action est exécutée, si le cache contient le résultat du calcul ($\texttt{contain}\,(I^m, a_1 : \ldots : a_n, read[\![D]\!]I^m)$) alors le résultat sauvegardé dans le cache est affecté aux variables écrites
($\texttt{lookup}\,(I^m, a_1 : \ldots : a_n, read[\![D]\!]I^m)$), sinon le calcul est effectué et le cache mis à jour ($\texttt{store}\,(I^m, a_1 : \ldots : a_n, read[\![D]\!]I^m, write[\![D]\!]I^m)$). Un tel aspect est verrouilleur uniquement si l'affectation ($write[\![D]\!]I^m \;\; \texttt{:= lookup(...)}$) est atomique. Si tel n'est pas le cas, l'affectation de plusieurs variables peut produire des états temporaires non atteignables.

L'Exemple 25 défini un aspect `memo` pour la procédure `fib` (Exemple 19). Il est facile de vérifier que `fib` ne lit aucune variable et écrit la variable `result`.

EXEMPLE **25.** *L'aspect memo ci-dessous mémorise les appels à* `fib` *uniquement si son argument est supérieur à 10 (pour amortir le coût de la sauvegarde dans le cache par exemple).*

```
memo (fib(β_A) ∧ if(β_A > 10))
```

La transformation T qui lui correspond est la suivante :

```
var cache := empty
{around (fib(β_A) ∧ if(β_A > 10))
    {if(contain(fib,[β_A],[]))
    then result:=lookup(fib,[β_A],[])
    else proceed; store(fib,[β_A],[],[result])}}
```

Notre version de `fib` *(Exemple 19) calcule plusieurs fois les mêmes appels et a une complexité exponentielle. L'aspect* `memo` *ci-dessus permet de ramener cette complexité à un temps linéaire.*

La propriété 26 formule le fait que tout aspect `memo` est un verrouilleur.

PROPRIÉTÉ **26.** $\forall a \in Asp^m.[\![T[\![a]\!]]\!] \in \mathcal{A}_v$

Nous n'avons pas prouvé cette propriété. Une preuve consisterait à montrer que dans la trace du programme tissé, les variables du programme de base ne sortent jamais de l'ensemble des états atteignables par l'exécution du programme de base. Ce comportement est garanti par la transformation T. Elle utilise uniquement des instructions dont la sémantique, soit restaure

le résultat d'une instruction du programme de base si elle a déjà été calculée et sauvegardée (`lookup`), soit exécute cette instruction (`proceed`) et sauvegarde (`store`) son résultat. Cela suppose comme mentionné ci-dessus que la restauration de plusieurs variables se fasse de façon atomique.

Dans le but d'implémenter des stratégies de mémoisation sophistiquées, un aspect `memo` peut être combiné avec un observateur. Comme les observateurs et les terminateurs sont inclus dans la catégorie des verrouilleurs, la composition d'un aspect verrouilleur avec un observateur ou un terminateur est aussi un verrouilleur. Par exemple, le programme de base pourrait être tissé avec un observateur qui collecte des statistiques sur les appels de procédures (*par ex.,* nombre d'appels, la profondeur d'une récursion, *etc.*) dans ses variables. Ensuite un aspect `memo` dont le prédicat teste ces variables peut être tissé.

4.7 Langages d'aspect non-déterministes

Dans les sections précédentes, nous avons présenté des langages d'aspect spécialisés qui préservent des classes de propriétés pour des programmes séquentiels. Les programmes non déterministes (*c.-à-d.,* concurrents) apportent de nouvelles catégories et classes de propriétés. Nous avons identifié à la Section 3.5 (page 68), les catégories sélecteur et adaptateur. Les sélecteurs sélectionnent des exécutions dans l'ensemble des exécutions possibles. Les adaptateurs sélectionnent ou terminent des exécutions dans l'ensemble des exécutions possibles. Pour obtenir un langage d'adaptateurs, il suffit d'ajouter `abort` au langages des sélecteurs. Nous discutons brièvement de la conception des langages spécialisés pour les catégories.

Nous supposons que, les commandes et les motifs sont étendus avec les commandes et motifs non déterministes suivants :

$$S \ ::= \ \ldots \mid \ S_1 \ \texttt{or} \ S_2$$

$$S^p \ ::= \ \ldots \mid \ S_1^p \ \texttt{or} \ S_2^p$$

La commande S_1 `or` S_2 exécute de façon non déterministe soit S_1 (Règle OR_1), soit S_2 (Règle OR_2)

$$\text{OR}_1 \quad \frac{(S_1 : C, \Sigma) \rightarrow_b (C', \Sigma')}{(S_1 \ \texttt{or} \ S_2 : C, \Sigma) \rightarrow_b (C', \Sigma')}$$

$$\text{OR}_2 \quad \frac{(S_2 : C, \Sigma) \rightarrow_b (C', \Sigma')}{(S_1 \ \texttt{or} \ S_2 : C, \Sigma) \rightarrow_b (C', \Sigma')}$$

Cette nouvelle commande doit être prise en compte par les langages d'aspect. Ainsi, en ajoutant S_1^o `or` S_2^o (resp. S_1^t `or` S_2^t) à S^o (resp. S^t) les langages des observateurs et des terminateurs restent valides. Concernant les verrouilleurs, la sémantique des aspects `memo` doit être adaptée de telle sorte que les fonctions $read_s$ et $write_s$ collectent des variables dans les deux branches non déterministes. Un autre langage dont les aspects sont aussi dans la catégorie des verrouilleurs est celui des aspects `rollback` qui représentent des aspects de tolérance aux fautes dont les actions restaurent des états précédents corrects. Ce langage qui n'a pas été étudié dans le cas déterministe (car une seule exécution est possible) peut l'être ici.

4.7.1 Le langage des `rollback`

Le aspects verrouilleurs `rollback` que nous définissons interagissent avec des observateurs dont la syntaxe est présentée par la Grammaire 13. Les aspects de Asp^o ici utilise le point de coupure S_1^p `or` S_2^p pour filtrer des exécutions non déterministes d'un programme. Pour chacune des instructions filtrées, l'aspect sauvegarde par l'intermédiaire de `proceedCommit` l'alternative non choisie de telle sorte que si une erreur survient dans la branche choisie, on puisse restaurer l'autre branche. La Grammaire 13 modifie la grammaire des observateurs présentée à la Section 4.4 en remplaçant les motifs de P par un motif dont la partie statique est S_1^p `or` S_2^p et la partie dynamique un prédicat B^o (défini dans la Grammaire 11) sur les variables du programme de base et de l'aspect. Les commandes de S^o sont étendues avec S_1^o `or` S_2^o. La commande `proceed` est remplacée par une nouvelle commande `proceedCommit`.

GRAMMAIRE **13.**

$$
\begin{array}{lcl}
Asp^o & ::= & D^o \texttt{ around } S_1^p \texttt{ or } S_2^p \wedge if(B^o)\ \{S_1^o \texttt{ ; proceedCommit ; } S_2^o\} \\
D^o & ::= & \ldots \\
S^o & ::= & \ldots \mid S_1^o \texttt{ or } S_2^o
\end{array}
$$

La commande `proceedCommit` transforme toute instruction S_1 `or` S_2 au sommet d'une pile proceed, en une autre instruction non déterministe qui en plus de l'exécution de la commande $S_1 \texttt{or} S_2$, enregistre le choix non sélectionné par l'intermédiaire de la fonction $commit$ (Règle PROCEEDCOMMIT et COMMIT). La sémantique d'un tel observateur utilise, en plus de la pile proceed, une pile Σ^S qui est un sous ensemble de l'état global Σ et qui contient les éléments sauvegardés par `proceedCommit`. Lorsque l'action est tissée, elle est suivie de l'instruction pop_{Σ^S} qui retire de la pile Σ^S, le choix enregistré lorsqu'aucune erreur n'intervient pendant l'exécution (Règle pop_{Σ^S}). Le tissage défini à la Section 4.4 devient

$$
\begin{array}{l}
[\![\texttt{around } s_1^p \texttt{ or } s_2^p \wedge if(B^o)\ s]\!] = \\
\quad \lambda(i : C, X \cup \Sigma^P \cup \Sigma^S). \\
\quad\quad if \quad match^s(s_1^p \texttt{ or } s_2^p, i) = \sigma \quad then \quad (\overline{a} : pop_{\Sigma^S} : C, X \cup \overline{i} : \Sigma^P \cup \Sigma^S) \\
\quad\quad else \quad nil \\
\quad \text{où } a = \sigma(\texttt{if (}B^o\texttt{) then } s \texttt{ else proceed})
\end{array}
$$

Le code généré par le tissage insère la commande `proceed` qui exécute le point de jonction lorsque le prédicat B^o n'est pas vrai.

$$\textsc{ProceedCommit}\ \frac{}{\begin{array}{c}(\texttt{proceedCommit} : C, X \cup S_1 \texttt{ or } S_2 : \Sigma^P \cup \Sigma^S) \\ \rightarrow ((commit(S_2 : C, X \cup \Sigma^P \cup \Sigma^S); S_1) \texttt{ or} \\ (commit(S_1 : C, X \cup \Sigma^P \cup \Sigma^S); S_2) : C, X \cup \Sigma^P \cup \Sigma^S)\end{array}}$$

$$\textsc{Commit}\ \frac{}{(commit(C', \Sigma') : C, X \cup \Sigma^P \cup \Sigma^S) \rightarrow (C, X \cup \Sigma^P \cup (C', \Sigma') : \Sigma^S)}$$

$$\textsc{Pop}_{\Sigma^S}\ \frac{\Sigma^S = x : \Sigma'^S}{(pop_{\Sigma^S} : C, X \cup \Sigma^P \cup \Sigma^S) \rightarrow (C, X \cup \Sigma^P \cup \Sigma'^S)}$$

Après le tissage de l'aspect observateur, si pendant l'exécution une erreur se produit, un aspect `rollback` de la Grammaire 14 est tissé.

GRAMMAIRE **14.**

$$Asp^r ::= \quad \texttt{around}\, error\, \texttt{rollback}$$

Cet aspect est un `around` qui filtre avec le motif $error$ des événements d'erreurs i générés pendant l'exécution. Ces événements qui ne sont pas formalisés ici, peuvent être des exceptions, certains appels de fonction ou valeurs de variables. Une fois l'erreur filtrée l'action `rollback` est exécutée. Remarquons que l'action n'ayant pas de commande `proceed`, l'instruction courante n'est pas placée sur la pile proceed.

$$\begin{array}{l}[\![\texttt{around}\, error\, \texttt{rollback}]\!] = \\ \quad \lambda(i : C, X \cup \Sigma^P \cup \Sigma^S). \\ \quad\quad \begin{array}{ll} if & match^s(error, i) = \sigma \quad then \quad (\texttt{rollback} : C, X \cup \Sigma^P \cup \Sigma^S) \\ else & nil \end{array}\end{array}$$

L'action `rollback` exécute la configuration qui est au sommet de Σ^S. Cette configuration correspond à l'exécution de l'autre choix non déterministe (Règle ROLLBACK).

$$\textsc{Rollback}\ \frac{}{(\texttt{rollback} : C, X \cup \Sigma^P \cup (C', \Sigma') : \Sigma^S) \rightarrow (C', \Sigma')}$$

Si une erreur est filtrée pendant l'exécution d'un choix non déterministe et Σ^S est vide (*c.-à-d.,* il y'a plus d'autres branches en attente), alors la commande `rollback` termine l'exécution du programme en générant une erreur dynamique sans modifier l'état Σ (Règle ERROR).

$$\textsc{Error}\ \frac{}{(\texttt{rollback} : C, X \cup \Sigma^P \cup \epsilon) \rightarrow (Dynamic\ Error : \bullet, X \cup \Sigma^P \cup \epsilon)}$$

L'action `rollback` permettant d'exécuter uniquement des choix d'exécution sauvegardés, on reste donc toujours dans l'ensemble des états atteignables par toutes les exécutions possibles. Par conséquent, l'aspect est dans la catégorie des verrouilleurs si la Règle ROLLBACK s'effectue de façon atomique.

L'interaction entre le langage des aspects `rollback` et celui des observateurs qui sauvegarde des choix d'exécution, peut permettre de parcourir toutes les exécutions possibles d'un programme non déterministe. Il suffit pour cela que lors de l'exécution de chaque branche, une erreur permet à l'aspect `rollback` d'exécuter l'autre branche et ainsi de suite.

4.7.2 Le langage des sélecteurs

Les sélecteurs sont des observateurs qui peuvent sélectionner des exécutions dans l'ensemble des exécutions possibles. Pour définir ce langage, nous étendons le langage des actions des observateurs avec les commandes `proceedLeft` et `proceedRight` en remplaçant la commande `proceed` par le non terminal ci-dessous :

$P^r ::= \texttt{proceedLeft} \mid \texttt{proceedRight} \mid \texttt{proceed} \mid \texttt{if(}B^o\texttt{) then } P_1^r \texttt{ else } P_2^r$

Comme `proceed`, les commandes `proceedLeft` et `proceedRight` exécutent l'instruction qui est au sommet de la pile proceed Σ^P si celle-ci est différente de S_1 `or` S_2 (Règle ALLPROCEED). Sinon `proceedLeft` exécute la partie gauche (Règle PROCEEDLEFT), et `proceedRight` la partie droite (Règle PROCEEDRIGHT).

$$\textsc{AllProceed}\ \frac{i = \texttt{proceedLeft} \vee \texttt{proceedRight} \vee \texttt{proceed} \wedge i' \neq S_1 \texttt{ or } S_2}{(i : C, X \cup i' : \Sigma^P) \rightarrow (i' : C, X \cup \Sigma^P)}$$

$$\textsc{ProceedLeft}\ \frac{}{(\texttt{proceedLeft} : C, X \cup S_1 \texttt{ or } S_2 : \Sigma^P) \rightarrow (S_1 : C, X \cup \Sigma^P)}$$

$$\textsc{ProceedRight}\ \frac{}{(\texttt{proceedRight} : C, X \cup S_1 \texttt{ or } S_2 : \Sigma^P) \rightarrow (S_2 : C, X \cup \Sigma^P)}$$

Par exemple, L'Exemple 27 ci-dessous présente un programme non déterministe qui
- calcule des nombres de fibonacci qui sont ensuite écrits sur un canal de communication ;
- ou lit des nombres écrits sur ce canal et les affiche sur la sortie standard.

EXEMPLE **27.** *Pour cet exemple, le langage de base a été étendu avec des instructions d'entrée-sortie (*par ex., *lecture sur l'entrée standard :*ι?*, écriture sur la sortie standard : o !). Le programme soit calcule les nombres de fibonacci des entiers compris entre* 1 *et* N *avec* $N \neq 0$ *et les écrits sur* dt *(dt!fib(i)), soit lit les valeurs écrites sur* dt *(*$dt?y$*) pour les écrire sur la sortie standard* o*.*

```
var i := 0;
while(true)
{(ι?N; i:=0; while(i < N) {i := i+1; dt!fib(i)}) or
       (o!0; while(!(i = 0)) {i := i - 1; dt?y; o!y})
}
```

Le caractère non déterministe de ce programme fait que celui-ci peut ne jamais lire les valeurs écrites sur dt, soit parce qu'aucune valeur y est écrite (*c.-à-d.,* l'exécution choisit toujours la partie droite de `or`), soit parce que le programme écrit infiniment sur dt (*c.-à-d.,* choisit toujours la partie gauche de `or`). L'aspect sélecteur de l'Exemple 28 permet de résoudre ce problème en transformant le programme de l'Exemple 27 en programme où les valeurs écrites sur dt sont ensuite lues.

EXEMPLE **28.** *L'aspect suivant sélectionne parmi les deux choix de l'Exemple 27 la lecture sur dt (`proceedLeft`) une fois que les valeurs y ont été écrites (`if ( !(i < N) )`).*

```
around ((ι?N; β_s1) or (o!0; β_s2))
        ∧  if (!(i < N) ∨ i=0)
 { if(i = 0) then proceedLeft else proceedRight }
```

4.8 Conclusion

Ce chapitre définit des langages d'aspect spécialisés qui assurent par construction qu'un aspect appartient à une catégorie spécifique et donc préserve la classe de propriétés associée. En particulier, nous avons proposé des langages généraux pour les observateurs et les terminateurs et un langage de verrouilleur dédié pour les aspects de mémoisation. Dans le cas des programmes concurrents, nous avons discuté d'autres langages d'aspect notamment des langages généraux pour les sélecteurs et les adaptateurs et un langage dédié pour les aspects de tolérance aux fautes (les aspects de `rollback` qui appartiennent aux verrouilleurs). Ces langages permettent de raisonner sur les aspects et le programme de base sans regarder le programme tissé. Ils assurent par construction que les aspects préservent une large classe de propriétés. Cependant, ils ne permettent pas de coder tous les aspects. Notamment les verrouilleurs qui ne sont pas des aspects `mémo` et `rollback`, et les aspects de la catégorie des faiblement intrusifs. Nous devons aussi montrer que les langages des observateurs et terminateurs permettent d'exprimer tous les aspects observateurs et terminateurs. Cette preuve peut se faire par cas sur la syntaxe du langage de base et des langages d'aspect en montrant que chaque langage d'aspect possède toutes les constructions du langage de base qui garantissent l'appartenance des aspects à la catégorie correspondante.

Katz [Kat06] propose d'utiliser des méthodes d'analyse, pour déterminer à posteriori la catégorie d'un aspect. Ces analyses peuvent être complexes, à l'instar de l'analyse d'alias pour vérifier qu'un aspect ne modifie pas une variable du programme de base. Dantas et Walker [DW06] qui ont identifié la catégorie des aspects inoffensifs (similaires aux terminateurs), ont proposé un langage qui assure par construction (via un système de type) qu'un aspect appartient à cette catégorie. La catégorie des inoffensifs ne modifie pas le résultat final du programme de base si le programme tissé n'est pas arrêté par l'aspect. Mais la préservation de propriétés temporelles n'est pas étudiée. Clifton et Leavens [CLN07] (voir page 34) utilisent les annotations et les types génériques de java pour spécifier qu'un aspect est observateur et terminateur. Ces spécifications sont ensuite assurées automatiquement par transformation de programme et un système de type comme dans [DW06]. Ici encore, la préservation des propriétés du programme de base n'est pas étudiée.

Plusieurs langages d'aspect dédiés ont été proposés dans la littérature. Lopes a proposé deux langages d'aspect spécifiques RIDL et COOL [VL97] qui permettent respectivement de décrire la transmission des données et la synchronisation des processus dans les applications

distribuées (Section 1.2.2, page 14). Fabry et coll. ont proposé KALA [FETD08], qui permet de décrire les transactions dans les applications distribuées. Mendhekar et coll. [MKL97] ont présenté un langage d'aspect qui utilise une primitive similaire à `memo` pour optimiser un système de traitement d'image. Fradet et Hong Tuan Ha [FHTH07] ont défini un langage semblable à celui des terminateurs pour empêcher les famines causées par la non libération de ressources partagées. Bien que ces langages contrôlent l'impact de leurs aspects sur le programme de base (voir Section 1.2.2, page 14), aucune étude ne porte sur les propriétés du programme de base qu'ils permettent de préserver.

La sémantique formelle des langages présentés dans ce chapitre, a été décrite en utilisant la CASB présentée au Chapitre 2. Ce cadre indépendant de tout type de langage de programmation, a pour but de décrire précisément tout type de langage de programmation par aspect. Mais jusqu'à présent, seuls des langages d'aspect assez simples ont été décrits. Dans le but de montrer l'expressivité de ce cadre, nous décrivons au chapitre suivant, des constructions sophistiquées des langages de programmation par aspect.

Chapitre 5

Autres applications du cadre

5.1 Introduction

Au Chapitre 2, nous avons défini un cadre formel qui décrit toutes les étapes d'exécution d'un programme à aspect (sémantique opérationnelle petit pas) afin de mieux comprendre son fonctionnement. Ce cadre est indépendant de tout type de langage de programmation et peut par conséquent permettre de décrire différents langages d'aspect. Dans ce cadre, les aspects de type `before`, `after` et `around` qui se retrouvent dans la plupart des langages d'aspect généraux sont décrits en introduisant à chaque fois, les constructions minimales du langage de base nécessaires. Nous l'avons utilisé aux Chapitres 3 et 4 pour proposer une approche qui permet de raisonner sur le programme de base et les aspects sans regarder le programme tissé.

L'objectif de ce chapitre, est de montrer que ce cadre est suffisamment expressif pour décrire tout type de langage d'aspect et étudier d'autres propriétés. Pour cela, nous décrivons des mécanismes sophistiqués utilisés par différents langages d'aspect. Ces mécanismes sont les points de coupure `cflow` et `cflowbelow`, les aspects sur les exceptions (les `around throws` et `after throwing` d'AspectJ), le déploiement dynamique des aspects (le `deploy` de CaesarJ) et les instances d'aspect (`pertarget` et `percflow`). Pour être indépendant des différents langages de base et aspect, nous introduisons pour chacun de ces mécanismes, les constructions du langage de base dont nous avons besoin. Par exemple, pour décrire `cflow` et `cflowbelow`, le langage de base doit posséder des instructions d'appel et de retour de méthode, de procédure ou de fonction. Pour les actions sur les exceptions, nous ajoutons au langage de base, les structures qui lèvent et récupèrent les exceptions (*c.-à-d.,* bloc `try - catch`). En général ces mécanismes interagissent et il est utile de décrire un langage d'aspect complet. Il est fait en décrivant à l'Appendice C un sous ensemble d'AspectJ (qui comporte les aspects de type `around`, les `cflow` et les instances d'aspect) basé sur un sous ensemble de Java (Featherweight Java avec des affectations [MP05]). Excepté `cflow` et `cflowbelow`, les différents mécanismes n'avaient jamais été formellement décrits auparavant.

Afin de montrer que notre cadre formel peut être utilisé pour faire des preuves de correction générales, nous présentons à la Section 5.3.3 la preuve que tout aspect `around` avec un point de coupure dynamique `if(...)` est équivalent à un aspect `around` où le point de coupure `if(...)` est transformé en une instruction conditionnelle dans le corps de l'action.

5.2 Constructions particulières des langages à aspect

Dans cette section, nous décrivons des constructions particulières des langages à aspect classiques. Il s'agit des points de coupure `cflow` et `cflowbelow`, des exceptions (`around throws`, `after throwing` et `handler`) et de l'instantiation (`pertarget`, `percflow` d'AspectJ, et `deploy` de CaesarJ). Ces constructions impliquent l'ajout de certaines instructions dans le langage de base présenté à la Section 2.2. Par souci de clarté, ces constructions sont décrites de façon isolée. Ces descriptions fournissent les briques de base pour la description d'un langage d'aspect complet.

5.2.1 Points de coupure `Cflow (below)`

Le point de coupure $\texttt{cflow}(P)$, permet de filtrer toutes les instructions qui sont dans le flot de contrôle des instructions filtrées par P (*par ex.,* des appels de fonctions), y compris les instructions filtrées par P. $\texttt{cflowbelow}(P)$ est similaire à $\texttt{cflow}(P)$ mais exclut les instructions filtrées par P. Par la suite, nous supposons que P ne puisse filtrer que des appels de fonction ou de procédure. Cette hypothèse est aussi celle d'AspectJ et des langages qui offrent ces points de coupure. D'où, pour décrire la sémantique de $\texttt{cflow}(P)$ (resp. $\texttt{cflowbelow}(P)$), nous introduisons les déclarations et les appels de procédure dans le langage de base.

$$Prog ::= (T\ \texttt{proc}\ p()\ S)^*\ S$$

$$T ::= \texttt{void} \mid \texttt{int} \mid \ldots$$

$$S ::= p() \mid \ldots$$

Dans cette grammaire, $T\ \texttt{proc}\ p()\ S$, représente la déclaration de la procédure p et dont le type de retour est T (*par ex.,* `void`, `int`, *etc.*). Un programme, est une collection de déclarations de procédures suivie d'une commande. L'appel de procédure se note $p()$. La sémantique de ces instructions est décrites par les règles ci-dessous. Elle utilise les environnements Σ^ρ et Σ^F qui sont des sous-ensembles de Σ. L'environnement Σ^ρ est une fonction qui associe à chaque nom de procédure p son corps et son type de retour. La pile Σ^F contient les signatures des procédures qui sont en train de s'exécuter. Le programme est donc dans le flot de contrôle des appels de procédure qui sont dans Σ^F. Ainsi, un appel de procédure insère un bloc représentant son adresse de retour et le corps de la procédure contenu dans Σ^ρ au sommet du code à exécuter. Il place aussi sa signature au sommet de Σ^F. Cette signature est retirée par l'adresse de retour à

la fin de l'exécution de la procédure.

$$\textsc{Call}\quad \frac{\Sigma^{\rho}(p) = (C', t)}{(p() : C, X \cup \Sigma^{\rho} \cup \Sigma^{F}) \rightarrow_{b} (C' : \{C\}, X \cup \Sigma^{\rho} \cup (t)p : \Sigma^{F})}$$

$$\textsc{Ret}\quad \frac{}{(\{C\}, X \cup \Sigma^{\rho} \cup (t)p : \Sigma^{F}) \rightarrow_{b} (C, X \cup \Sigma^{\rho} \cup \Sigma^{F})}$$

Le point de coupure $\texttt{cflow}(P)$ filtre une instruction si elle se trouve dans le flot de contrôle des instructions filtrées par P. Chaque signature est une paire (p, t). Si l'instruction est un appel de procédure, p est le nom de la procédure et t son type. Pour les autres instructions, p est l'instruction et `void` leur type de retour. La sémantique de *cflow* et *cflowbelow* est décrite par la fonction $match_f$. Elle étend la fonction de filtrage *match* décrite à la Section 2.5, afin qu'elle prenne en compte la pile Σ^F. La fonction $match_f$ prend en paramètres un motif, une signature et une pile de signature.

$$match_f : P \times Sig \times Sig^* \rightarrow bool$$

où P est l'ensemble des points de coupure, Sig l'ensemble des signatures et Sig^* la pile des signatures correspondant aux appels de procédures en cours. Avec la syntaxe suivante pour les points de coupure

$$P ::= (P_T)P_I \mid \texttt{cflow}(P) \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid \neg\, P$$

$$P_T ::= x \mid \texttt{void} \mid \texttt{int} \mid \ldots$$

$$P_I ::= x \mid p$$

Dans cette grammaire, $(P_T)P_I$ est un motif filtrant toute signature (p, t) avec p filtré par P_I et t par P_T. Le motif P_I (resp. P_T) est soit un identificateur (resp. un type) ou une variable de motif x pouvant filtrer tout type ou identificateur. La fonction $match_f$ a la définition suivante :

$$\begin{array}{lcl}
match_f((P_T)P_I, i, \Sigma^F) & = & match(P_T, t) \ \wedge \ match(P_I, p) \ \ si\ i = (p, t) \\
match_f(\texttt{cflow}(P), i, \epsilon) & = & false \\
match_f(\texttt{cflow}(P), i, i' : \Sigma'^F) & = & match_f(P, i, i' : \Sigma'^F) \\
 & & \vee\, match_f(\texttt{cflow}(P), i', \Sigma'^F) \\
 & & \\
match_f(\texttt{cflowbelow}(P), i, \epsilon) & = & false \\
match_f(\texttt{cflowbelow}(P), i, i' : \Sigma'^F) & = & match_f(\texttt{cflow}(P), i', \Sigma'^F) \\
 & & \\
match_f(P_1 \wedge P_2, i, \Sigma^F) & = & match_f(P_1, i, \Sigma^F) \wedge match_f(P_2, i, \Sigma^F) \\
match_f(P_1 \vee P_2, i, \Sigma^F) & = & match_f(P_1, i, \Sigma^F) \vee match_f(P_2, i, \Sigma^F) \\
match_f(\neg P, i, \Sigma^F) & = & \neg match_f(P, i, \Sigma^F)
\end{array}$$

Une instruction est filtrée par $\texttt{cflow}(P)$ si P la filtre ou l'appel de procédure qui précède cette instruction est dans le flot de contrôle de P. Elle est filtrée par $\texttt{cflowbelow}(P)$ si l'appel de procedure qui précède cette instruction est dans le flot de contrôle de P.

5.2.2 Exceptions et aspects

Nous introduisons les exceptions dans le langage de base en utilisant les instructions suivantes :

$$S ::= \texttt{try}\ S_1\ \texttt{catch}\ ex\ S_2 \mid \texttt{throw}\ ex \mid \ldots$$

L'instruction $\texttt{try}\ S_1\ \texttt{catch}\ ex\ S_2$ déclare une nouvelle exception ex qui peut être levée pendant l'exécution de S_1 et capturée pendant l'exécution de S_2. L'instruction $\texttt{throw}\ ex$ lève l'exception ex.

L'état abstrait Σ contient une pile Σ^E qui contient toutes les exceptions qui ont été déclarées. Chaque élément de Σ^E est une paire de type $I \times C$ où I est l'ensemble des identificateurs des exceptions et C un code. Une paire (ex, C) de Σ^E fournie le code C à exécuter lorsque l'exception ex est attrapée. Ces paires sont placées dans la pile dans l'ordre des déclarations des blocs `try-catch`. Lorsqu'une exception ex est levée, la continuation courante est remplacée par le code associé à ex dans Σ^E. Si l'exception ne se trouve pas dans Σ^E, le programme s'arrête avec une erreur dynamique de type "*uncaught exception*".

La sémantique des exceptions est décrite par la relation $\rightarrow_b$ ci-dessous. L'exécution d'un bloc $\texttt{try}\ S_1\ \texttt{catch}\ ex\ S_2$, place dans Σ^E la paire $(ex, S_2 : C)$ et exécute le bloc S_1. L'instruction pop_e retire cette paire de Σ^E après l'exécution de S_1. Lorsqu'une exception ex est levée, la continuation courante C est remplacée par le code C' associé à la première occurrence de ex dans Σ^E. Toutes les exceptions sauvegardées après ex sont retirées de Σ^E ; en effet, une levée d'exception supprime toutes les exceptions introduites depuis sa déclaration.

$$\textsc{Try}\ \frac{}{(\texttt{try}\ S_1\ \texttt{catch}\ ex\ S_2 : C,\ X \cup \Sigma^E) \rightarrow_b (S_1 : pop_e : C,\ X \cup (ex,\ S_2 : C) : \Sigma^E)}$$

$$\textsc{Pop}_e\ \frac{\Sigma^E = x : \Sigma'^E}{(pop_e : C,\ X \cup \Sigma^E) \rightarrow_b (C,\ X \cup \Sigma'^E)}$$

$$\textsc{Throw}\ \frac{\Sigma^E = (ex_0, C_0) : \ldots : (ex_k, C_k) : (ex, C') : \Sigma'^E}{(\texttt{throw}\ ex : C,\ X \cup \Sigma^E) \rightarrow_b (C',\ X \cup \Sigma^E)}$$
$$\text{avec } \texttt{ex}_i \neq \texttt{ex} \text{ et } 0 \leq i \leq k$$

$$\textsc{Uncaught}\ \frac{}{(\texttt{throw}\ ex : C,\ X \cup (ex_0, C_0) : \ldots : (ex_k, C_k) : \epsilon) \rightarrow_b \text{" } \textit{Uncaught exception} \text{"}}$$
$$\text{avec } \texttt{ex}_i \neq \texttt{ex} \text{ et } 0 \leq i \leq k$$

Le langage AspectJ, utilise les constructions syntaxiques telles que `around throws`, `after throwing` et `handler`, pour prendre en compte les exceptions dans les aspects. Nous présentons maintenant la sémantique de ces constructions.

Aspects `Around throws`

Un aspect `around throws` P P_{ex} filtre une instruction qui est filtrée par P et qui pourrait lever une exception filtrée par P_{ex}. Son exécution est celle d'un aspect `around` dans lequel le point de coupure et le filtrage est modifié afin de prendre en compte les exceptions. Nous utilisons la grammaire suivante pour P_{ex}

$$P_{ex} ::= * \mid id$$

où $*$ représente toutes les exceptions et id l'identificateur d'une exception. Nous utilisons aussi la fonction $excep$ qui prend une instruction et retourne la liste des exceptions lex que ce point de jonction peut lever. Ainsi, la fonction $match_{ex}$ retourne $true$ s'il existe au moins une exception dans la liste des exceptions qui est filtrée par le motif des exceptions P_{ex} ($match_{ex}(lex, *) = true$). Un aspect `around throws` est décrit en redéfinissant Σ^{ψ}. Cette définition, qui effectue un tissage statique, associe chaque instruction à la liste d'exceptions qu'elle peut lever. Un aspect `around throws` P P_{ex} est donc défini par la fonction Σ^{ψ} ci-dessous.

$$\begin{array}{rl} \Sigma^{\psi}(i : C, X \cup \Sigma^{P}) = & \textit{if } \textbf{match}(P, i) \wedge match_{ex}(excep(i), P_{ex}) \\ & then\ (\sigma(test\ \phi) : C, X \cup \overline{i} : \Sigma^{P})\ else\ nil \\ & \text{avec } \sigma\ tel\ que\ \sigma(P) = i \end{array}$$

Aspects `After throwing`

Les aspects `after throwing` s'appliquent sur les procédures qui propagent des exceptions à l'exécution. Nous supposons que les appels de procédure et leurs retours sont formalisés en utilisant la pile Σ^F comme dans la Section 5.2.1, page 96. La pile Σ^F contient toutes les signatures des appels de procédures en train de s'exécuter. Pour filtrer un appel de procédure qui propage une exception, la pile Σ^F, l'exception et la continuation lorsqu'on entre dans un bloc `try - catch`, doivent être sauvegardées. Les deux règles TRY et POP$_e$ précédentes sont donc redéfinies comme suit :

$$\textsc{Try}\ \frac{}{\begin{array}{c}(\texttt{try}\ S_1\ \texttt{catch}\ ex\ S_2 : C,\ X \cup \Sigma^F \cup \Sigma^E) \\ \rightarrow_b\ (S_1 : pop_e : C,\ X \cup \Sigma^F \cup (ex,\ S_2 : C, \Sigma^F) : \Sigma^E)\end{array}}$$

$$\textsc{Pop}_e\ \frac{\Sigma^E = x : \Sigma'^E}{(pop_e : C,\ X \cup \Sigma^F \cup \Sigma^E)\ \rightarrow_b\ (C,\ X \cup \Sigma^F \cup \Sigma'^E)}$$

Lorsqu'une exception est levée, le programme est remplacé par le code C' associé à cette exception et la pile Σ^E est modifiée comme dans la règle THROW précédente. Au lieu de remplacer la pile courante Σ^F par la pile Σ'^F associée à l'exception levée, Σ^F est modifiée récursivement par l'instruction Ret_p. Ce processus itératif nous permettra de filtrer tous les appels de procédures dans Σ^F qui ont propagé cette exception.

$$\textsc{Throw}\ \frac{\Sigma^E = (ex_0, C_0, \Sigma_0^F) : \ldots : (ex_k, C_k, \Sigma_k^F) : (ex, C', \Sigma'^F) : \Sigma'^E}{\begin{array}{l}(\texttt{throw}\, ex : C,\ X \cup \Sigma^F \cup \Sigma^E) \\ \rightarrow_b\ (Ret_p\ ex\ \Sigma'^F : C',\ X \cup \Sigma^F \cup \Sigma'^E) \text{ avec } ex_i \neq ex \text{ et } 0 \leq i \leq k\end{array}}$$

La fonction $Ret_p\ ex\ \Sigma'^F$ retire le sommet de la pile Σ^F jusqu'à ce qu'elle soit égale à la valeur qu'elle avait à l'entrée du bloc `try-catch` correspondant à l'exception ex (règle $\textsc{Ret}_p^1$). Chaque signature retirée, est celle d'une instruction qui a propagé l'exception ex et donc candidate à l'insertion d'une action `after throwing`. La règle $\textsc{Ret}_p^2$ exécute la continuation de l'exception lorsque Σ^F possède la valeur qu'elle avait à l'entrée du bloc `try-catch` correspondant à l'exception qui a été propagée.

$$\textsc{Ret}_p^1\ \frac{}{(Ret_p\ ex\ \Sigma'^F : C, X \cup (t)p : \Sigma^F \cup \Sigma^E)\ \rightarrow_b\ (Ret_p\ ex\ \Sigma'^F : C, X \cup \Sigma^F \cup \Sigma^E)}$$

$$\textsc{Ret}_p^2\ \frac{}{(Ret_p\ ex\ \Sigma^F : C, X \cup \Sigma^F \cup \Sigma^E)\ \rightarrow_b\ (C, X \cup \Sigma^F \cup \Sigma^E)}$$

Si l'aspect `after throwing` Σ^ψ filtre l'appel de procédure $p()$ correspondant à la signature qui est au sommet de Σ^F et p propageant l'exception ex, alors l'action de l'aspect `after throwing` est placée au sommet de la continuation associée à ex et la signature de p est taguée. Ainsi, l'instruction correspondant à cette signature ne sera pas de nouveau filtrée et pourra être retirée de Σ^F par $Ret_p\ ex\ \Sigma'^F$. Cela est nécessaire afin de garantir que le tissage termine. Notons qu'ici Σ^ψ filtre l'instruction au sommet de Σ^F contrairement aux fonctions Σ^ψ précédentes qui filtraient l'instruction courante.

$$\begin{array}{l}\Sigma^\psi(Ret_p\ ex\ \Sigma'^F : C, X \cup (t)p : \Sigma^F \cup \Sigma^E) = \\ \quad \textit{if } \textit{match}(P, p())\ then\ (Ret_p\ ex\ \Sigma'^F : \sigma(test\ \phi) : C, X \cup \overline{(t)p} : \Sigma^F \cup \Sigma^E)\ else\ nil \\ \quad \text{avec } \sigma\ tel\ que\ \sigma(P) = p()\end{array}$$

EXEMPLE **29.** *Considérons le programme $Prog$ et l'aspect Σ^ψ défini comme suit*

$$\begin{array}{lcl} Prog & = & \texttt{try}\ foo()\ \texttt{catch}\ ex_1\ \epsilon \\ void\ foo()\ ex_1 & = & goo() \\ void\ goo()\ ex_1 & = & throw\ ex_1 \end{array}$$

la déclaration des procédures est modifiée pour permettre de spécifier qu'elles puissent propager une exception (T `proc` $p()$ ex S). $Prog$ appelle la procédure foo dans un bloc `try-catch`*. Cette procédure, qui peut propager une exception ex_1, appelle la procédure goo qui lève ex_1.*

$$\begin{array}{lcl} \Sigma^\psi & = & \phi \\ & & \textit{avec } P = (x)x \\ \phi(\Sigma) & = & baz() \\ void\ baz() & = & \epsilon \end{array}$$

L'aspect Σ^{ψ} tisse un appel à baz après tout appel de procédure se terminant en retournant une exception. La réduction de $Prog$ avec l'aspect Σ^{ψ} est celle-ci :

$$
\begin{array}{ll}
(start : \texttt{try}\ foo()\ \texttt{catch}\ ex_1\ \epsilon : \bullet, X \cup \epsilon \cup \epsilon) & \\
\rightarrow & (\texttt{try}\ foo()\ \texttt{catch}\ ex_1\ \epsilon : \bullet, X \cup \epsilon \cup \epsilon) \\
\rightarrow & (foo() : pop_e : \bullet, X \cup \epsilon \cup (ex_1, \epsilon, \epsilon) : \epsilon) \\
\rightarrow & (goo() : \{pop_e : \bullet\}, X \cup (void)\ foo : \epsilon \cup (ex_1, \epsilon, \epsilon) : \epsilon) \\
\rightarrow & (\texttt{throw}\ ex_1 : \{\{pop_e : \bullet\}\}, X \cup (void)\ goo : (void)\ foo : \epsilon \cup (ex_1, \epsilon, \epsilon) : \epsilon) \\
\rightarrow & (Ret_p\ ex_1\ \epsilon : \bullet, X \cup (void)\ goo : (void)\ foo : \epsilon \cup \epsilon) \\
\rightarrow & (Ret_p\ ex_1\ \epsilon : test\ \phi : \bullet, X \cup \overline{(void)\ goo} : (void)\ foo : \epsilon \cup \epsilon) \\
\rightarrow & (Ret_p\ ex_1\ \epsilon : test\ \phi : test\ \phi : \bullet, X \cup \overline{(void)\ foo} : \epsilon \cup \epsilon) \\
\rightarrow & (Ret_p\ ex_1\ \epsilon : test\ \phi : test\ \phi : \bullet, X \cup \epsilon \cup \epsilon) \\
\rightarrow & (test\ \phi : test\ \phi : \bullet, X \cup \epsilon \cup \epsilon) \\
\rightarrow & (baz() : test\ \phi : \bullet, X \cup \epsilon \cup \epsilon) \\
\rightarrow & (\{test\ \phi : \bullet\}, X \cup (void)\ baz : \epsilon \cup \epsilon) \\
\rightarrow & (test\ \phi : \bullet, X \cup \epsilon \cup \epsilon) \\
\rightarrow & (baz() : \bullet, X \cup \epsilon \cup \epsilon) \\
\rightarrow & (\{\epsilon : \bullet\}, X \cup (void)\ baz : \epsilon \cup \epsilon) \\
\rightarrow & (\epsilon\bullet, X \cup \epsilon \cup \epsilon)
\end{array}
$$

Lorsqu'une exception est levée, la continuation courante est remplacée par le code du bloc $\texttt{catch}\ ex_1$ *(ici, ϵ) et $Ret_p\ ex_1$ qui va récursivement retirer de la pile Σ^F les deux signatures qui ont propagé l'exception levée. A chaque fois, l'aspect Σ^{ψ} insère l'appel à baz.*

Point de coupure Handler

Dans AspectJ, le point de coupure `handler` P_{ex}, filtre tout point de jonction qui attrape une exception ex (*c.-à-d.,* `catch` ex) filtrée par le motif P_{ex}. Il est supporté uniquement par les aspects `before`. Lorsqu'une exception est levée, comme dans le cas de la règle THROW, la continuation courante est remplacée par le code associé à l'exception levée (*c.-à-d.,* le code du bloc `catch` qui attrape l'exception). L'aspect Σ^{ψ} insère donc son action ($test\ \phi$) avant ce code si l'exception levée est filtrée par P_{ex}.

$$
\begin{array}{l}
\Sigma^{\psi}(\texttt{throw}\ ex : C,\ X \cup (ex_0, C_0) : \ldots : (ex_k, C_k) : (ex, C') : \Sigma^E) = \\
\quad if\, match_{ex}(ex, P_{ex})\ then\ (test\ \phi : C',\ X \cup \Sigma^E) \\
\quad \text{avec } ex_i \neq ex \text{ et } 0 \leq i \leq k
\end{array}
$$

5.2.3 Déploiement des aspects

Les aspects, tout comme les classes, peuvent être instantiés dynamiquement. Par exemple, un aspect peut être instantié à l'entrée d'un bloc et l'instance détruite à sa sortie. Ce type de

déploiement d'aspect, s'oppose au déploiement statique où les aspects sont instantiés une fois pour toute. Nous décrivons ici, la sémantique d'un déploiement dynamique d'aspect similaire à celui que permet la construction `deploy` du langage CaesarJ [AGMO06].

Nous considérons l'instruction `deploy id` S qui permet d'activer l'aspect du nom id pendant l'exécution du bloc S et de le désactiver après. L'aspect Σ^ψ est donc représenté par une pile qui sauvegarde tous les aspects actifs. Elle contient tous les aspects qui sont instantiés par l'instruction `deploy` mais aussi ceux qui sont instantiés de façon statique. Ces aspects globaux sont à la fin de la pile. Lorsque l'instruction `deploy id` S est exécutée, la nouvelle instance d'aspect Σ^ψ_{id} est placée au sommet de Σ^ψ et le bloc S est exécuté. Après l'exécution de S, l'instruction pop_{Σ^ψ} désactive l'aspect qui est au sommet de Σ^ψ en le dépilant.

$$\textsc{Deploy}\ \frac{}{(\texttt{deploy id}\ S : C,\ X \cup \Sigma^\psi)\ \rightarrow_b\ (S : pop_{\Sigma^\psi} : C,\ X \cup \Sigma^\psi_{id} : \Sigma^\psi)}$$

$$Pop_{\Sigma^\psi}\ \frac{}{(pop_{\Sigma^\psi} : C,\ X \cup \Sigma^\psi_{id} : \Sigma^\psi)\ \rightarrow\ (C,\ X \cup \Sigma^\psi)}$$

Durant l'exécution, appliquer un aspect Σ^ψ à une configuration (C, Σ) revient à appliquer la pile des aspects actifs à (C, Σ). Ces aspects doivent être ordonnés en fonction de leurs priorités. Nous supposons que ces priorités sont données par la fonction $priority$ qui peut être définie par le programmeur similairement à la déclaration `declare precedence` d'AspectJ. Appliquer une pile d'aspects $(\Sigma^\psi_{id1} : \ldots : \Sigma^\psi_{idn} : \epsilon)$ à (C, Σ), se définit par

$$(\Sigma^\psi_{id1} : \ldots : \Sigma^\psi_{idn} : \epsilon)(C, \Sigma)\ \ =\ \ \textit{priority}(\Sigma^\psi_{id1}, \ldots, \Sigma^\psi_{idn})(C, \Sigma)$$

où la définition de $\textit{priority}(\Sigma^\psi_{id1}, \ldots, \Sigma^\psi_{idn})(C, \Sigma)$ est équivalente à la définition de $\Sigma^\psi(C, \Sigma)$ dans le cas général de plusieurs aspects de types différents (voir Section 2.4.2, page 51).

Le déploiement d'aspect peut être vu comme une version simple de l'instantiation des aspects que nous présentons dans la section qui suit.

5.2.4 Association d'aspect

L'association d'aspect est un mécanisme qui associe un aspect particulier à une structure sémantique d'une instruction (*par ex.,* un objet, un flot de contrôle). Dans notre cadre, cela est effectuée par la fonction Σ^ψ prenant une configuration (C, Σ) en argument et créant une nouvelle instance d'aspect associée à l'instruction courante. Chaque instance possède un état privé sauvegardé dans l'état des aspects Σ^a et pouvant évoluer pendant l'exécution. Comme à la Section 2.3, Σ^ψ peut aussi insérer une action ($test\ \phi$) avant, après et autour de l'instruction courante de C. Dans cette section, nous présentons comment Σ^ψ crée des nouvelles instances dans le sens des instructions `pertarget` et `percflow` d'AspectJ.

Par opposition aux instances d'aspect qui sont créées une fois pour toute ou pour le temps d'un bloc (`deploy`), certains aspects peuvent être associés à certaines entités dynamiques telles que des objets. Ainsi, plusieurs instances d'un même aspect peuvent co-exister à l'exécution. Par exemple, une instance d'aspect peut être associée à chaque objet d'une classe donnée. Comme il n'est pas possible en général de connaître statiquement le nombre d'instances d'une classe, les aspects doivent être instanciés dynamiquement.

Instantiation d'aspect

Les instances d'aspect pouvant être générées à l'exécution, Σ^{ψ} doit évoluer dynamiquement pour prendre en compte toutes les instances d'aspect. Pour cela, nous le représentons comme un environnement qui associe à chaque identificateur d'aspect unique, une instance Σ^{ψ}_{id} ; avec $\mathit{dom}(\Sigma^{\psi})$ qui est l'ensemble des identificateurs des aspects existants.

Chaque fois qu'une instruction est tissée, la fonction Σ^{ψ} peut être mise à jour par la fonction *update*. Cette fonction teste si l'instruction courante déclenche la création d'une instance. Si c'est le cas, elle est créée et le nouvel environnement est appliqué à la configuration courante. Sinon, les aspects élémentaires de Σ^{ψ} sont ordonnés et appliqués comme dans le cas du déploiement des aspects (voir Section 5.2.3, page 101). L'état de chaque instance étant sauvegardé dans Σ^{a}, *update* prend également Σ en paramètre et retourne un nouvel état Σ' dans lequel Σ^{ψ} et Σ^{a} ont été modifiés.

$$
\begin{array}{rcll}
\Sigma^{\psi}(i : C, \Sigma) & = & \mathit{if} & \mathit{update}(i, \Sigma) = \Sigma' \\
 & & \mathit{then} & \Sigma'^{\psi}(\bar{i} : C, \Sigma') \\
 & & \mathit{else\ if} & \mathit{update}(i, \Sigma) = \Sigma \\
 & & \mathit{then} & \mathit{priority}(\Sigma^{\psi}_{id1}, \ldots, \Sigma^{\psi}_{idn})(i : C, \Sigma) \\
 & & & \text{avec } \mathit{dom}(\Sigma^{\psi}) = \{id1, \ldots, idn\} \text{ et } \Sigma' = X \cup \Sigma'^{a} \cup \Sigma'^{\psi}
\end{array}
$$

La fonction *update* est idempotente (*c.-à-d.,* une unique instance d'un même aspect peut être associée à une instruction i)

$$\mathit{update}(i, \Sigma) = \Sigma' \ \mathit{then}\ \mathit{update}(i, \Sigma') = \Sigma'$$

et les instructions taguées ne génèrent pas de nouvelles instances

$$\mathit{update}(\bar{i}, \Sigma) = \Sigma$$

Un autre rôle de la fonction *update* pourrait être de supprimer les instances d'aspect lorsque les entités auxquelles elles sont associées n'existent plus (ramasse miettes). Mais nous ne détaillons pas ce point.

Exemple (pertarget)

Nous illustrons l'instanciation des aspects par un exemple qui associe un aspect `counter` à chaque instance d'une classe `Point`. Ce type d'instanciation (par objet) correspond à la directive `pertarget` d'AspectJ. Chaque aspect `counter` tisse avant chaque appel à une méthode m de `Point`, une action qui calcule le nombre d'appel de m dans un programme. Premièrement, nous allons décrire le mécanisme d'association et d'instanciation. Ensuite, nous présenterons les détails de l'action.

Nous avons deux options : soit nous créons une instance de `counter` à chaque fois qu'un nouvel objet de la classe `Point` est créé, soit nous la créons uniquement lorsque la méthode m est invoquée par un objet de la classe `Point` pour la première fois. Nous considérons ici le deuxième cas mais la première option pourrait également se décrire dans notre modèle.

Les nouvelles instances d'aspect sont créées en utilisant un générateur d'aspect noté G. Dans le cas de `pertarget`, un générateur est une fonction qui attend un objet x et un état Σ, et retourne une instance d'aspect et un nouvel état Σ' qui contient l'état de l'instance et le nouvel environnement d'aspect. Une instance d'aspect est une fonction Σ^{ψ}_{id}. Par convention, l'identificateur est de la forme `Nom_`x qui est le nom de l'aspect tagué par la référence de l'objet auquel il est associé. Ainsi, dans notre exemple, *update* est définie comme suit, avec $x.m$ qui représente l'appel de la méthode m sur l'objet x

$$
\begin{array}{lcll}
\textit{update}(\texttt{x.m}(), \Sigma) & \triangleq & \Sigma' & \text{si } \texttt{counter\_}x \notin \textit{dom}(\Sigma^{\psi}) \\
 & & & \text{avec } G(x, \Sigma) = (\Sigma^{\psi}_{\texttt{counter\_}x}, \Sigma') \\
 & & & \text{et } \Sigma'^{\psi} = \Sigma^{\psi}\{\texttt{counter\_}x \rightarrow \Sigma^{\psi}_{\texttt{counter\_}x}\} \\
 & & & \text{et } \Sigma' = X \cup \Sigma'^{\psi} \cup \Sigma'^{a} \\
\textit{update}(i, \Sigma) & \triangleq & \Sigma & \text{sinon}
\end{array}
$$

Dans le premier cas, une instance est créée en appelant G. Cette instance $\Sigma^{\psi}_{\texttt{counter\_}x}$ est ensuite ajoutée à l'environnement des aspects Σ^{ψ} et son état sauvegardé dans Σ^{a} qui devient Σ'^{a}. Dans le second cas, l'instruction ne déclenche pas la création d'une instance et Σ n'est pas modifié. La généralisation à d'autres classes et aspects est immédiate.

Afin d'illustrer la sauvegarde d'un nouvel état dans Σ^{a}, nous présentons les détails de `counter`. Nous supposons que Σ^{a} est un environnement qui associe aux variables d'aspect des valeurs. Nous supposons aussi que la fonction $incr$ qui prend une variable et l'incrémente est donnée. Formellement, le générateur G associé à l'aspect `counter` est défini comme

$$
\begin{array}{lcll}
 & G(x, \Sigma) & \triangleq & (\Sigma^{\psi}_{\texttt{counter\_}x}, \Sigma') \\
\text{où} & \Sigma^{\psi}_{\texttt{counter\_}x} & \triangleq & \begin{cases} (\texttt{x.m}() : C, \Sigma) \mapsto ((\lambda\Sigma.\textit{incr}\ z) : \overline{\texttt{x.m}()} : C, \Sigma) & \text{si } \texttt{classOf}(x) = \texttt{Point} \\ (i : C, \Sigma) \mapsto \textit{nil} & \text{sinon} \end{cases} \\
\text{et} & \Sigma'^{a} & \triangleq & \Sigma^{a}\{z \mapsto 0\}
\end{array}
$$

Le nouvel environnement Σ'^a est défini comme $\Sigma^a\{z \mapsto 0\}$, qui est l'environnement Σ^a dans lequel la nouvelle variable z est initialisée à 0. L'instance $\Sigma^{\psi}_{\texttt{counter\_}x}$ est un aspect `before` défini comme

$$(\texttt{x.m}() : C, \Sigma) \mapsto ((\lambda\Sigma.incr\ z) : \overline{\texttt{x.m}()} : C, \Sigma)$$

où $(\lambda\Sigma.incr\ z)$ est le *test* ϕ de l'instance $\Sigma^{\psi}_{\texttt{counter\_}x}$

Exemple (percflow)

Dans cet exemple, nous présentons la sémantique d'une instanciation de type $\texttt{percflow}(P)$ utilisé dans AspectJ. Un aspect $\texttt{percflow}(P)$ est instancié chaque fois que le programme entre dans le flot de contrôle de P. Plus précisément, un programme est dans le flot de contrôle de P si son instruction courante est filtrée par $\texttt{cflow}(P)$ (voir Section 5.2.1, page 96). Lorsqu'un programme passe d'une instruction qui n'est pas filtrée par $\texttt{cflow}(P)$ à une instruction qui est filtrée par $\texttt{cflow}(P)$, on dit qu'il entre dans le flot de contrôle de P. Inversement, on dit qu'il sort du flot de contrôle de P.

Formellement, nous supposons que nous avons un générateur d'aspect, qui, prend en argument l'état Σ et l'instruction i qui permet d'entrer dans le flot de contrôle de P, et retourne une instance Σ^{ψ}_{id} et un nouvel état Σ'

$$G(i, \Sigma) = (\Sigma^{\psi}_{id}, \Sigma')$$

Une instance Σ^{ψ}_{id} est générée lorsque l'instruction courante est un appel de méthode dont la signature est filtrée par P ($match_f$ utilisée lors de la définition de $\texttt{cflow}(P)$) et si on ne se trouve pas déjà dans le flot de contrôle de P ($id \notin dom(\Sigma^{\psi})$). Inversement, l'aspect est supprimé de Σ^{ψ} chaque fois que le programme sort du flot de contrôle de P. Cela intervient à chaque reduction exprimée par la règle RET définie à la Section 5.2.1, page 96. La fonction *update* qui implémente la création d'instance par un aspect $\texttt{percflow}(P)$ lors d'un tissage est définie comme suit :

$$\begin{array}{lcl}
update(i, X \cup \Sigma^{\psi} \cup \Sigma^F) & \triangleq & X' \cup \Sigma'^{\psi} \cup \Sigma^F \\
 & & \quad \text{si } match_f(P, i, \Sigma^F) \text{ et } id \notin dom(\Sigma^{\psi}) \\
 & & \quad \text{avec } G(i, X \cup \Sigma^{\psi} \cup \Sigma^F) = (\Sigma^{\psi}_{id}, X' \cup \Sigma'^{\psi} \cup \Sigma^F) \\
 & & \quad \text{et } \Sigma'^{\psi} = \Sigma^{\psi}\{id \rightarrow \Sigma^{\psi}_{id}\} \\
update(\{C\}, X \cup \Sigma^{\psi} \cup \Sigma^F) & \triangleq & X \cup \Sigma^{\psi} - id \cup \Sigma^F \\
 & & \quad \text{si } \neg match_f(P, \{C\}, \Sigma^F) \\
update(i, X \cup \Sigma^{\psi} \cup \Sigma^F) & \triangleq & X \cup \Sigma^{\psi} \cup \Sigma^F \\
 & & \quad \text{si } \neg match_f(P, i, \Sigma^F) \text{ ou } id \in dom(\Sigma^{\psi})
\end{array}$$

Le premier cas de cette définition, correspond à l'entrée du programme dans le flot de contrôle de P. Le deuxième cas, supprime l'instance id à la sortie du flot de contrôle de P. Dans le troisième cas, l'instance n'est pas créée si elle existe déjà ou si on n'est pas dans le flot de contrôle de P.

5.3 Preuve de correction d'implémentation

Nous voulons maintenant montrer, qu'il est possible d'utiliser notre cadre pour raisonner sur des transformations de programme ou d'aspect. Nous considérons des transformations qui peuvent être effectuées lors de l'implémentation du tissage des aspects et ceci indépendamment des différents langages d'aspect. Pour ce faire, la définition de la fonction Σ^{ψ} est détaillée tout en restant indépendant de tout langage d'aspect. Cette définition nous permet d'optimiser les tissages de la Section 2.4, page 48 où les actions vides (cas `before` et `after`) et les actions ayant uniquement `proceed` comme instruction (cas `around`) peuvent être supprimées sans modifier le résultat final.

5.3.1 Définition formelle des aspects

Dans cette section, nous définissons la sémantique de la fonction Σ^{ψ} qui représente les aspects. Nous considérons qu'un aspect est composé d'un type K (`before`, `after` ou `around`), d'un point de coupure P filtrant des points de jonction et d'une action C (*c.-à-d.,* un code à exécuter). Il a la grammaire suivante :

$$\begin{array}{lcl} A & ::= & K\ P\ C \\ K & ::= & \texttt{before} \mid \texttt{after} \mid \texttt{around} \\ P & ::= & T \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid \neg P \mid \texttt{if}(E) \\ T & ::= & c\ T_1 \ldots T_n \mid * \end{array}$$

Un point de coupure P peut être un simple motif T, une disjonction, conjonction, négation des points de coupure ou un test dynamique `if`(E). Un motif T est un terme dont la syntaxe est celle des instructions : c est un symbole d'arité $n \geq 0$ qui représente un élément syntaxique du langage de programmation ou des informations particulières des points de jonction (*c.-à-d.,* des noms, des variables, des types, des valeurs, *etc.*). Le motif $*$ filtre toute instruction.

La sémantique d'un aspect (*c.-à-d.,* la fonction Σ^{ψ}) se définie comme suit :

$$\begin{array}{ll} [\![\texttt{before}\ P\ C]\!] & = \lambda(i : C', \Sigma).\mathit{if}\ \neg \mathit{match}(P, i)\ \mathit{then}\ \mathit{nil} \\ & \quad \mathit{else}\ (\mathit{test}\ (\lambda\Sigma.(\mathit{if}\ \mathit{match}_D(P, i, \Sigma) \\ & \quad \mathit{then}\ C\ \mathit{else}\ \epsilon)) : \bar{i} : C', \Sigma) \\ [\![\texttt{after}\ P\ C]\!] & = \lambda(i : C', \Sigma).\mathit{if}\ \neg \mathit{match}(P, i)\ \mathit{then}\ \mathit{nil} \\ & \quad \mathit{else}\ (\bar{i} : \mathit{test}\ (\lambda\Sigma.(\mathit{if}\ \mathit{match}_D(P, i, \Sigma) \\ & \quad \mathit{then}\ C\ \mathit{else}\ \epsilon)) : C', \Sigma) \\ [\![\texttt{around}\ P\ C]\!] & = \lambda(i : C', X \cup \Sigma^P).\mathit{if}\ \neg \mathit{match}(P, i)\ \mathit{then}\ \mathit{nil} \\ & \quad \mathit{else}\ (\mathit{test}\ (\lambda(X \cup \Sigma^P).(\mathit{if}\ \mathit{match}_D(P, i, X \cup \Sigma^P) \\ & \quad \mathit{then}\ C\ \mathit{else}\ \texttt{proceed})) : \mathit{pop}_p : C', X \cup \bar{i} : \Sigma^P) \end{array}$$

L'aspect prend une configuration $(i : C', \Sigma)$ en argument et sélectionne l'instruction courante i en la filtrant avec un motif P. Si la partie statique de P ne filtre pas i (*c.-à-d.,* la fonction *match*

retourne $false$, voir Section 2.5, page 51), l'aspect retourne nil. Sinon, si le point de coupure filtre dynamiquement l'instruction (*c.-à-d.,* i est filtrée et `if`(E) est vrai), l'action C est insérée en accord avec les sémantiques définies à la Section 2.3, page 44. Dans le cas contraire (i est filtrée et `if`(E) est faux), une action vide ϵ est insérée dans le cas des aspects de type `before` et `after` ; pour les aspects de type `around`, l'action `proceed` est insérée. Lorsque plusieurs aspects $A_1, \ldots, A_n$, sont appliqués à $(i : C', \Sigma)$, la définition de Σ^ψ varie en fonction des types d'aspect (voir Section 2.4, page 48). Par exemple, pour les aspects de type `around`, en accord avec la sémantique décrite à la Section 2.4.1, Σ^ψ est générée comme suit :

$$
\begin{array}{l}
[\![\texttt{around}\ P_1\ C_1, \ldots, \texttt{around}\ P_n\ C_n]\!] \\
\quad = \lambda(i : C', X \cup \Sigma^P). \\
\quad \mathit{if}\ \neg \textbf{\textit{match}}(P_1, i) \wedge \ldots \wedge\ \neg \textbf{\textit{match}}(P_n, i)\ \mathit{then}\ \mathit{nil} \\
\quad \mathit{else}\ (\mathit{test}\ (\lambda(\mathcal{X} \cup \Sigma^P).(\mathit{if}\ \mathit{match}_D(P_1, i, X \cup \Sigma^P) \\
\quad \mathit{then}\ C_1\ \mathit{else}\ \texttt{proceed}))\ :\ \mathit{pop}_p\ n : C', \\
\quad X \cup f_2 : \ldots : f_n : \bar{i} : \Sigma^P) \\
\quad \text{avec}\ f_2 = \mathit{test}\ (\lambda(\mathcal{X} \cup \Sigma^P).(\mathit{if}\ \mathit{match}_D(P_2, i, X \cup \Sigma^P) \\
\qquad\qquad\qquad \mathit{then}\ C_2\ \mathit{else}\ \texttt{proceed})) \\
\quad \text{et}\ f_n = \mathit{test}\ (\lambda(\mathcal{X} \cup \Sigma^P).(\mathit{if}\ \mathit{match}_D(P_n, i, X \cup \Sigma^P) \\
\qquad\qquad\qquad\ \mathit{then}\ C_n\ \mathit{else}\ \texttt{proceed}))
\end{array}
$$

Le filtrage de motif dynamique est spécifié par la fonction $match_D$ qui prend en paramètres un point de coupure, une instruction, un état (pour évaluer le prédicat `if`(E)) et retourne un booléen.

$$match_D : P \times Instruction \times \Sigma \rightarrow Bool$$

$$
\begin{array}{lcl}
match_D(T, i) & = & \lambda\Sigma.\textbf{\textit{unify}}(T, i) \\
match_D(P_1 \wedge P_2, i) & = & \lambda\Sigma.match_D(P_1, i)\Sigma \wedge match_D(P_2, i)\Sigma \\
match_D(P_1 \vee P_2, i) & = & \lambda\Sigma.match_D(P_1, i)\Sigma \vee match_D(P_2, i)\Sigma \\
match_D(\neg P, i) & = & \lambda\Sigma.\neg match_D(P, i)\Sigma \\
match_D(\texttt{if}(E), i) & = & \lambda\Sigma.[\![E]\!]\Sigma
\end{array}
$$

Cette sémantique de filtrage a deux niveaux comme démontré par la définition abstraite de Σ^ψ qui tisse une fonction ϕ. Le premier niveau concerne uniquement les informations statiques (*c.-à-d.,* une instruction i). Le second repose sur les informations dynamiques (*c.-à-d.,* l'état Σ juste avant l'exécution de l'action). La fonction *unify*, qui n'est pas définie ici, retourne $true$ si le motif T s'associe à l'instruction i. Dans le cas contraire, le motif ne filtre pas l'instruction et la fonction retourne $false$.

La fonction $match_D$ génère des expressions booléennes qui peuvent être simplifiées avec des lois usuelles de la logique booléenne. Par exemple, dans le cas de plusieurs aspects, quand le point de coupure P ne filtre pas une instruction i, $match_D(P, i)$ est simplifié pour s'écrire $\lambda\Sigma.false$ et dans ce cas, l'action insérée est $\lambda\Sigma.\epsilon$ pour les aspects `before` et `after` et $\lambda\Sigma.$`proceed` pour les aspects `around`. Ces actions peuvent être supprimées du code à exécuter sans modifier le résultat final. Pour formaliser cette optimisation, la Définition 30 spécifie que deux aspects sont équivalents (noté $\simeq$) si pour tout programme de base, les programmes tissés calculent le même résultat final.

DÉFINITION **30.**

$$\Sigma^\psi \simeq \Sigma'^\psi \textit{ iff } \forall C.\forall(\Sigma = X \cup \Sigma^\psi \vee \Sigma = X \cup \Sigma'^\psi). \quad (C, X \cup \Sigma^\psi) \rightarrow^* (\epsilon : \bullet, X' \cup \Sigma^\psi) \Leftrightarrow (C, X \cup \Sigma'^\psi) \rightarrow^* (\epsilon : \bullet, X' \cup \Sigma'^\psi)$$

Par exemple, les aspects

$$\Sigma^\psi(i : C, \Sigma) = (\overline{i} : test\ (\lambda\Sigma.\epsilon) : test\ \phi_2 : C, \Sigma) \text{ et } \Sigma^\psi(i : C, \Sigma) = (\overline{i} : test\ \phi_2 : C, \Sigma)$$

sont équivalents. En effet,

$$(test\ (\lambda\Sigma.\epsilon) : test\ \phi_2 : C, \Sigma) \rightarrow^* (test\ \phi_2 : C, \Sigma)$$

Cela est aussi le cas pour les aspects `after`, où il suffit de supprimer l'instruction $test\ (\lambda\Sigma.\epsilon)$ du code à exécuter. Pour les aspects `around`, l'instruction $(\lambda\Sigma.\texttt{proceed})$ peut être supprimé de la pile proceed Σ^P. Ce dernier cas est formalisé par la Propriété 31.

PROPRIÉTÉ **31.** *Soient Σ^ψ et Σ'^ψ deux aspects tels que*

$$\begin{array}{l}
\Sigma^\psi(i : C, X \cup \Sigma^P) \\
\quad = (test\ \phi_1 : pop_p\ n : C, \\
\qquad X \cup test\ \phi_2 : \ldots : test\ (\lambda\Sigma.\texttt{proceed}) : test\ \phi_{j+1} : \ldots : test\ \phi_n : \overline{i} : \Sigma^P) \\
\Sigma'^\psi(i : C, X \cup \Sigma^P) \\
\quad = (test\ \phi_1 : pop_p\ n - 1 : C, X \cup test\ \phi_2 : \ldots : test\ \phi_{j+1} : \ldots : test\ \phi_n : \overline{i} : \Sigma^P) \\
\Sigma'^\psi(i' : C, X \cup \Sigma^P) = \Sigma^\psi(i' : C, X \cup \Sigma^P) \quad \textit{avec } i' \neq i \\
\textit{alors } \Sigma^\psi \simeq \Sigma'^\psi
\end{array}$$

Preuve. Immédiate par application des règles PROCEED* et POP* de la Section 2.4.1, page 49. □

Le langage des points de coupure utilisé dans cette concrétisation et optimisation, peut être étendu afin de prendre en compte les points de coupure sur le flot de contrôle (*c.-à-d., cflow* et *cflowbelow*). Le langage de base doit alors avoir la notion de procédure ou de fonction. Dans ce cas, les sémantiques de Σ^ψ données ici, doivent modifier une pile de flot qui contient les fonctions qui sont en train de s'exécuter et la fonction $match_D$ doit prendre cette pile en argument et l'explorer pendant le processus de filtrage.

5.3.2 Transformation de points de coupure dynamique

Nous montrons maintenant comment la CASB peut être utilisée afin de prouver la correction d'une transformation de programme par aspect. Nous ne détaillons pas les preuves. Notre principal objectif est de :

– montrer que la preuve de correction d'une transformation apparemment simple peut être délicate et requérir ainsi un traitement formel ;
– montrer que la CASB est un cadre suffisant pour effectuer une telle preuve.

La transformation porte sur l'implémentation des points de coupure P. Si un point de coupure est purement statique, il appartient au sous ensemble représenté par la grammaire

$$P^S ::= T \mid P_1^S \wedge P_2^S \mid P_1^S \vee P_2^S \mid \neg P^S$$

Comme décrit par la fonction $match_D$, filtrer une instruction avec P, consiste d'abord à appliquer un motif appartenant à P^S et ensuite à évaluer un prédicat $\texttt{if}(E)$ sur l'état courant Σ. Cependant, au lieu d'utiliser un test dynamique pendant un filtrage de motif, les points de coupure peuvent être implémentés en utilisant une instruction conditionnelle dans l'action. Plus précisément, un aspect de la forme

$$\texttt{around}\,(P^S \wedge \texttt{if}(E))\,C$$

est équivalent à

$$\texttt{around}\,P^S\,(\texttt{if}\,E\,\texttt{then}\,C\,\texttt{else}\,\texttt{proceed})$$

de tel sorte que si P^S ne filtre pas l'instruction courante, nil est retourné ; sinon l'action

$$\texttt{if}\,E\,\texttt{then}\,C\,\texttt{else}\,\texttt{proceed}$$

est insérée. Le test dans le point de coupure est donc remplacé par un test dans l'action. Nous supposons ainsi que les expressions booléennes E des points de coupure et du langage des actions ont la même syntaxe. Nous supposons aussi que l'instruction conditionnelle a la sémantique suivante :

$$\begin{array}{lll}
(\texttt{if}\,E\,\texttt{then}\,C_1\,\texttt{else}\,C_2 : C, \Sigma) & \rightarrow_b & ([\![E]\!]\Sigma : \{C_1\} : \{C_2\} : C, \Sigma) \\
(true : \{C_1\} : \{C_2\} : C, \Sigma) & \rightarrow_b & (C_1 : C, \Sigma) \\
(false : \{C_1\} : \{C_2\} : C, \Sigma) & \rightarrow_b & (C_2 : C, \Sigma)
\end{array}$$

Avant d'appliquer cette transformation, tout aspect doit être transformé en aspects de la forme $\texttt{around}\ (P^S \wedge \texttt{if}(E)))\ C$. Cette transformation est similaire à celle présentée à la Section 4.4, page 80. La transformation des points de coupure dynamiques en instructions conditionnelles dans l'action suit le processus ci-dessous :

1. les aspects `before` et `after` sont transformés en aspects `around` en appliquant la fonction *toaround* de la Section 2.4.2, page 51 ;
2. comme présenté à la Section 4.4, page 80, le point de coupure P de chaque aspect est mis sous la forme
$$(P_1^S \wedge \texttt{if}(E_1)) \vee \ldots \vee (P_n^S \wedge \texttt{if}(E_n))$$
Les règles de transformations utilisées ici sont les suivantes :
$$\texttt{if}(E_1) \wedge \texttt{if}(E_2) \equiv \texttt{if}(E_1 \wedge E_2) \qquad \neg\texttt{if}(E) \equiv \texttt{if}(\neg E)$$
$$\texttt{if}(E) \equiv * \wedge \texttt{if}(E) \qquad P^S \equiv P^S \wedge \texttt{if}(true)$$

Notons que, contrairement à la transformation de la Section 4.4, les points de coupure ne possèdent pas de variables de motif, et la négation d'un motif $\texttt{if}(E)$ est possible et elle consiste à évaluer si l'expression booléenne $\neg E$ est vraie. L'aspect

$$\texttt{around}\,((P_1^S \wedge \texttt{if}(E_1)) \vee \ldots \vee (P_n^S \wedge \texttt{if}(E_n)))\,C$$

peut être alors transformé en plusieurs aspects mutuellement exclusifs de la forme $\texttt{around}\,(P^S \wedge \texttt{if}(E))C$;

3. chaque aspect de la forme $\texttt{around}\,(P^S \wedge \texttt{if}(E))\,C$ est alors transformé en l'aspect $\texttt{around}\ P^S\ (\texttt{if}\ E\ \texttt{then}\ C\ \texttt{else}\ \texttt{proceed})$ dont le point de coupure est purement statique.

5.3.3 Preuve de correction

Afin de prouver la correction du processus de transformation, nous allons montrer la correction des deux dernières étapes car la correction de la première étape est directe en appliquant les règles de sémantique d'un aspect de type `around` (voir Section 2.4.1, page 49). La correction de l'étape 2 est formalisée par la Propriété 32. Par souci de simplicité, cette propriété concerne l'exemple de transformation présenté à la Section 4.4.

PROPRIÉTÉ **32.**

$$\begin{aligned}
&[\![\texttt{around}\,(P_1^S \wedge \neg P_2^S \wedge \texttt{if}(E_1))\,C,\\
&\ \texttt{around}\,(P_2^S \wedge \neg P_1^S \wedge \texttt{if}(E_2))\,C,\\
&\ \texttt{around}\,(P_1^S \wedge P_2^S \wedge \texttt{if}(E_1 \vee E_2))\,C]\!]\\
\simeq\ &[\![\texttt{around}\,(P_1^S \wedge \texttt{if}(E_1)) \vee (P_2^S \wedge \texttt{if}(E_2))\,C]\!]
\end{aligned}$$

La Propriété 32 assure que l'aspect d'origine est équivalent au sens de la Définition 30 aux aspects obtenus après transformation.

Preuve. Nous convertissons les deux fonctions sémantiques en des formes qui correspondent aux quatre cas possibles que peuvent avoir les parties statiques de l'aspect d'origine. Par exemple, pour toute configuration $(i : C', X \cup \Sigma^P)$, dans le cas où l'instruction courante peut être filtrée par P_1^S et pas par P_2^S, la fonction sémantique Σ^ψ correspondant à l'aspect d'origine retourne

$$(\mathit{test}\ (\lambda(X \cup \Sigma^P).\mathit{if}\ [\![E_1]\!](X \cup \Sigma^P)\ \mathit{then}\ C\ \mathit{else}\ \texttt{proceed}) : \mathit{pop}_p : C', X \cup \overline{i} : \Sigma^P)$$

tandis que la fonction Σ'^ψ correspondant aux trois aspects retourne

$$\begin{aligned}
(\mathit{test}\ (\lambda(X \cup \Sigma^P).\mathit{if}\ [\![E_1]\!](X \cup \Sigma^P)\ \mathit{then}\ C\ \mathit{else}\ \texttt{proceed}) : \mathit{pop}\ 3 :\\
C', X \cup \mathit{test}\ (\lambda(X \cup \Sigma^P).\texttt{proceed}) : \mathit{test}\ (\lambda(X \cup \Sigma^P).\texttt{proceed}) : \overline{i} : \Sigma^P)
\end{aligned}$$

Par application de la Propriété 31 on a $\Sigma^\psi \simeq \Sigma'^\psi$. Les autres cas sont similaires. □

La correction de l'étape 3 est formalisée par la propriété suivante

PROPRIÉTÉ **33.**

$$\begin{array}{rl} & \texttt{around}\,(P^S \wedge \texttt{if}(E))\ \overline{skip} : C \\ \simeq & \texttt{around}\,P^S\ (\texttt{if}\ E\ \texttt{then}\ \overline{skip} : C\ \texttt{else}\ \texttt{proceed}) \\ & \textit{avec}\ \ \forall\Sigma.(skip : C, \Sigma) \rightarrow_b (C, \Sigma) \end{array}$$

Rappelons qu'une étape d'exécution du programme tissé exécute d'abord l'instruction courante par $\rightarrow_b$ avant de tisser la configuration retournée par cette exécution (voir la Règle REDUCE, Section 2.3, page 44). Nous avons utilisé $\overline{skip}$ pour éviter que le code C de `if` E `then` C `else proceed` soit tissé avant celui de `around` $(P^S \wedge \texttt{if}(E))\ C$ car si tel est le cas, la dernière étape de la transformation n'est pas correcte. En effet, $(test\ \phi(\Sigma) : C', \Sigma)$ se réduisant par $\rightarrow$ en $(\phi(\Sigma) : C', \Sigma)$ (voir Règle ADVICE, page 45),
et `if` E `then` C `else proceed` se réduisant d'abord par $\rightarrow_b$, C pourrait être tissé tandis que $(\phi(\Sigma) : C', \Sigma)$ est réduite par $\rightarrow_b$. Sachant que $\phi(\Sigma)$ pourrait retourner C, l'équivalence ne serait plus correcte. Comme $\overline{skip}$ ne peut pas être tissé, lorsque E est évaluée à vraie, on a :

$$(\texttt{if}\ E\ \texttt{then}\ \overline{skip} : C\ \texttt{else}\ \texttt{proceed} : C', \Sigma) \rightarrow^2 (\overline{skip} : C : C', \Sigma)$$

De même pour

$$(test\ \phi(\Sigma) : C', \Sigma) \rightarrow (\overline{skip} : C : C', \Sigma)$$

et puisque $(\overline{skip} : C : C', \Sigma) \rightarrow_b (C : C', \Sigma)$, on a bien l'équivalence de l'étape 3.

Dans cette preuve, nous supposons que `if ... then...else...`, *true*, *false*, `proceed`, $\{C\}$, et $\{\texttt{proceed}\}$ ne peuvent pas être filtrées par un aspect. Si le langage d'aspect peut filtrer ces instructions, alors elles doivent être taguées (`if`, *etc.*), sinon la propriété n'est pas correcte (pour la même raison que ci-dessus). Notons que ce style de détail n'est pas facile à mettre en évidence avec une approche informelle.

Premièrement, nous définissons une relation de bissimulation $\sim$ entre les configurations. Soit $\Sigma^\psi = [\![\texttt{around}\ (P^S \wedge \texttt{if}(E))\ \overline{skip} : C]\!]$ et $\Sigma'^\psi = [\![\texttt{around}\ P^S\ (\texttt{if}\ E\ \texttt{then}\ \overline{skip} : C\ \texttt{else}\ \texttt{proceed})]\!]$ alors

$$(C_1, X \cup \Sigma^\psi \cup \Sigma_1^P) \sim (C_2, X \cup \Sigma'^\psi \cup \Sigma_2^P) \Leftrightarrow C_1 \sim C_2 \wedge \Sigma_1^P \sim \Sigma_2^P$$

$$\begin{array}{lll} \text{avec}\ \ Z_1 \sim Z_2 & \textit{iff} & Z_1 = Z_2 \vee \\ & & (Z_1 = i_1 : Z_1' \wedge Z_2 = i_2 : Z_2' \\ & & \wedge\ Z_1' \sim Z_2' \wedge [\![i_1]\!] = [\![i_2]\!]) \end{array}$$

$$\begin{array}{lll} \text{et} \quad [\![i_1]\!] = [\![i_2]\!] & \textit{iff} & i_1 = i_2 \vee \\ & & (i_1 = \texttt{test}\ (\lambda(X \cup \Sigma^\psi \cup \Sigma_1^P).\textit{if}\,[\![E]\!](X \cup \Sigma^\psi \cup \Sigma_1^P) \\ & & \qquad \textit{then}\ C\ \textit{else}\ \texttt{proceed}) \wedge \\ & & i_2 = \texttt{if}\ E\ \texttt{then}\ C\ \texttt{else}\ \texttt{proceed}) \end{array}$$

Deux configurations sont en relation si elles ont le même état et si les aspects Σ^ψ et Σ'^ψ, leurs codes et leurs piles proceed sont en relation. Deux piles sont en relation, si leurs éléments sont syntaxiquement identiques ou l'élément du premier est de la forme $test\ (\lambda(X \cup \Sigma^\psi \cup$

$\Sigma_1^P).if\,[\![E]\!](X \cup \Sigma^\psi \cup \Sigma_1^P) then\ C\ else$ `proceed`) quand celui du second est `if` E `then` C `else proceed`.

Nous montrons que si deux configurations sont en relation alors après une réduction de la première et une ou deux réductions de la seconde, les configurations résultantes sont en relation.

LEMME **34.**

$$\begin{array}{ll} & (C_1, X \cup \Sigma^\psi \cup \Sigma_1^P) \sim (C_2, X \cup \Sigma'^\psi \cup \Sigma_2^P) \wedge \\ & (C_1, X \cup \Sigma^\psi \cup \Sigma_1^P) \rightarrow (C_1', X' \cup \Sigma^\psi \cup \Sigma_1'^P) \\ \Rightarrow & (C_2, X \cup \Sigma'^\psi \cup \Sigma_2^P) \rightarrow^{1/2} (C_2', X' \cup \Sigma'^\psi \cup \Sigma_2'^P) \wedge \\ & (C_1', X' \cup \Sigma^\psi \cup \Sigma_1'^P) \sim (C_2', X' \cup \Sigma'^\psi \cup \Sigma_2'^P) \end{array}$$

Comme la relation est préservée par réduction, les états finaux (*c.-à-d.,* de la forme $(\epsilon : \bullet, X \cup \Sigma^\psi \cup \epsilon)$) sont en relation et satisfont la Définition 30. La Propriété 33 est alors prouvée et notre transformation correcte.

Preuve du Lemme 34. La preuve est faite en considérant les différentes règles de réduction

Cas REDUCE : Σ^ψ et Σ'^ψ retournent nil

$$\begin{array}{lllll} 1. & & (i : C_1, X_1 \cup \Sigma^\psi \cup \Sigma_1^P) & \sim & (i : C_2, X_2 \cup \Sigma'^\psi \cup \Sigma_2^P) \\ & \text{alors} & (i : C_1, X_1 \cup \Sigma^\psi \cup \Sigma_1^P) & \rightarrow & (C_1', X' \cup \Sigma^\psi \cup \Sigma_1'^P) \\ & \text{et} & (i : C_2, X_2 \cup \Sigma'^\psi \cup \Sigma_2^P) & \rightarrow & (C_2', X' \cup \Sigma'^\psi, \Sigma_2'^P) \end{array}$$

Notons que ce cas ne concerne que les instructions courantes qui sont syntaxiquement identiques. Intuitivement, l'exécution des instructions peut dépendre de la structure de C et Σ^P mais aussi ajouter/tester/retirer/mettre à jour des éléments de C et Σ. Cela se vérifie dans la majorité des langages de programmation qui empêche la réflexivité du code. Nous n'avons pas formalisé cela ici. Ainsi, la réduction de la même instruction i de deux configurations en relation produit deux configurations en relation car l'instruction ayant le même effet sur C et Σ, les continuations, les piles proceed et les autres environnements seront en relation. Ce raisonnement s'applique aussi aux règles PROCEED et POP.

$$\begin{array}{ll} 2. & (test\ (\lambda(X \cup \Sigma^\psi \cup \Sigma_1^P).if\,[\![E]\!](X \cup \Sigma^\psi \cup \Sigma_1^P) then\ \overline{skip} : C\ else\ \texttt{proceed}) : \\ & \quad C_1, X \cup \Sigma^\psi \cup \Sigma_1^P) \\ \sim & (\texttt{if}\ E\ \texttt{then}\ \overline{skip} : C\ \texttt{else proceed} : C_2, X \cup \Sigma'^\psi \cup \Sigma_2^P) \end{array}$$

Il est clair que les deux configurations obtenues après une réduction pour la première configuration et deux pour la seconde, sont en relation (voir les Règles REDUCE et ADVICE, Section 2.3 et 2.3.1).

Cas REDUCE : Σ^ψ, Σ'^ψ ne retournent pas nil et $(C_1, X \cup \Sigma^\psi \cup \Sigma_1^P) \sim (C_2, X \cup \Sigma'^\psi \cup \Sigma_2^P)$

Par définition, Σ^ψ et Σ'^ψ filtrent uniquement les instructions obtenues après réduction par $\rightarrow_b$ et telles que

$$
\begin{array}{ll}
 & (C_1, X \cup \Sigma^{\psi} \cup \Sigma_1^P) \sim (C_2, X \cup \Sigma'^{\psi} \cup \Sigma_2^P) \wedge \\
 & (C_1, X \cup \Sigma^{\psi} \cup \Sigma_1^P) \rightarrow_b (i : C_0, X' \cup \Sigma^{\psi} \cup \Sigma_1'^P) \\
\Rightarrow & (C_2, X \cup \Sigma'^{\psi} \cup \Sigma_2^P) \rightarrow_b (i : C_0', X' \cup \Sigma'^{\psi} \cup \Sigma_2'^P) \wedge \\
 & (i : C_0, X' \cup \Sigma^{\psi} \cup \Sigma_1'^P) \sim (i : C_0', X' \cup \Sigma'^{\psi} \cup \Sigma_2'^P)
\end{array}
$$

Dans les autres cas, soit les instructions ne peuvent pas être explicitement filtrées par les points de coupure statiques, soit elles sont taguées (Σ^{ψ} et Σ'^{ψ} retournent nil). Après le tissage, on obtient les réductions suivantes :

$$
\begin{array}{l}
(C_1, X \cup \Sigma^{\psi} \cup \Sigma_1^P) \rightarrow (test\ (\lambda(X' \cup \Sigma^{\psi} \cup \Sigma_1'^P).if\ [\![E]\!](X' \cup \Sigma^{\psi} \cup \Sigma_1'^P) \\
\qquad\qquad then\ C\ else\ \texttt{proceed}) : pop_p : C_0, X' \cup \Sigma^{\psi} \cup \bar{i} : \Sigma_1'^P) \\
(C_2, X \cup \Sigma'^{\psi} \cup \Sigma_2^P) \rightarrow (\texttt{if}\ E\ \texttt{then}\ C\ \texttt{else}\ \texttt{proceed} : pop_p : C_0', X' \cup \Sigma'^{\psi} \cup \bar{i} : \Sigma_2'^P)
\end{array}
$$

Les deux nouvelles configurations sont clairement en relation. □

5.4 Conclusion

Afin de montrer que le cadre formel du Chapitre 2 est suffisamment expressif pour décrire différents types de langage d'aspect, nous avons décrit des mécanismes relativement sophistiqués appartenant à des langages d'aspect différents. Par souci de clarté, ces primitives sont décrites de façon séparée mais elles fournissent des briques de base et des détails importants pour la description formelle d'un langage d'aspect complet (voir Appendice C). Ces primitives sont :

- les points de coupure `cflow` et `cflowbelow`, qui permettent de filtrer l'ensemble des points de jonction qui se trouvent dans le flot de contrôle d'un point de jonction donné ;
- l'action `around throws` d'AspectJ. Une action `around throws` est une action `around` où le point de jonction pourrait lever une exception donnée ;
- l'action `after throwing` d'AspectJ. Elle exécute le corps de l'action après qu'une exception soit levée ;
- le point de coupure `handler` d'AspectJ, qui sélectionne la capture d'une exception ;
- la primitive `deploy` de CaesarJ qui permet d'activer dynamiquement un aspect dans un bloc d'instruction. A la sortie du bloc, l'aspect est désactivé (*c.-à-d.,* ne peut plus être utilisé) ;
- les primitives `pertarget` et `percflow` d'AspectJ, qui permettent respectivement de créer une instance d'aspect pour chaque receveur d'une classe donnée et pour chaque entrée dans le flot de contrôle d'un point de jonction donné.

L'autre point important de ce chapitre, a été l'utilisation de notre cadre formel, pour montrer que toute action `around` avec un point de coupure `if(...)` est équivalente à une action `around` où le point de coupure se transforme en une instruction conditionnelle dans le corps de l'action. Cette transformation permet d'optimiser le tissage statique des aspects. La preuve de cette équivalence est faite en montrant que, le tissage de chacun de ces aspects sur un même programme de base, donne le même résultat final en passant par des états intermédiaires équiva-

lents. Cette transformation est montrée pour tout type de langage d'aspect possédant le point de coupure dynamique `if(...)`. Formaliser cette preuve qui paraissait facile nous a permis de mettre en évidence des subtilités et des restrictions pour qu'elle soit correcte (*par ex.,* les instructions conditionnelles `if ... then...else...` ne peuvent pas être filtrées, les programmes ne doivent pas être réflexifs, *etc.*).

D'autres travaux ont effectués des transformations d'aspect. Robert Dyer et Hridesh Rajan [DR06] propose le langage Nu qui étend Java avec deux primitives *bind* et *remove*. La primitive *bind* prend en paramètres un motif et une méthode, et associe à tout point de jonction filtré par le motif, la méthode, qui sera exécutée avant ou après le point de jonction. La primitive *remove* supprime l'association faite par *bind*. Les aspects à la AspectJ sont ensuite transformés dans ce langage. Ces aspects sont ceux dont les actions sont de type `before` ou `after` avec les points de coupure dynamiques `cflow`, `if`, `this`, `perthis`, `percflow`, *etc.* Par exemple, la transformation utilisée pour le point de coupure `if` est celle que nous avons démontrée. L'objectif de ces transformations est de pouvoir utiliser Nu comme un langage intermédiaire qui préserve la structure modulaire des aspects AspectJ à la compilation. En effet, la compilation avec AspectJ des aspects qui ont des points de coupure dynamiques introduit des instructions supplémentaires qui rompent la structure modulaire des aspects. Cependant ces transformations sont informelles et l'équivalence entre les aspects AspectJ et Nu n'est pas montrée. Dans [ACH+05], Avgustinov et coll. proposent une démarche d'optimisation du code tissé généré par AspectJ. Cette démarche est basée sur des transformations du code Java généré à la compilation des aspects ainsi que sur des analyses inter et intra procédurales. Par exemple une analyse inter procédurale qui détermine statiquement les points de jonction filtrés par le point de coupure `cflow` est présentée. Cette analyse permet d'optimiser le code généré par le compilateur et qui teste dynamiquement si le point de jonction est filtré. Malgré les études de cas qui présentent les gains de coûts qu'apportent l'implémentation de ces optimisations, elles sont informelles et l'équivalence entre le code tissé et le code optimisé n'est pas montrée.

Notre cadre pourrait être utilisé pour compléter les travaux ci-dessus en montrant leur correction. Un autre objectif, à plus long terme, serait de montrer la correction du tissage dans les langages à aspect car il est effectué par transformation de programme telle que décrite par la CASB et par des transformations supplémentaires comme celle prouvée dans ce chapitre.

Conclusion

Bilan

Ce travail a défini un cadre sémantique formel pour la programmation par aspect. Ce cadre décrit, à l'aide d'une sémantique opérationnelle petit pas, toutes les étapes d'exécution d'un programme tissé, indépendamment d'un langage de base spécifique. Le tissage d'aspects communs aux langages d'aspect généraux (*c.-à-d.,* `before`, `after` et `around`) est décrit en faisant un minimum d'hypothèses sur le langage de base. Ce tissage est très général et peut sélectionner tout type d'instruction du programme de base.

Nous avons utilisé ce cadre pour proposer une approche qui permet de raisonner modulairement sur le programme de base et les aspects (*c.-à-d.,* sans regarder le programme tissé). Dans les travaux précédents plusieurs démarches ont été utilisées pour résoudre ce problème. La vérification [KFG04, GK07] génère à partir du programme de base des informations qui permettent de vérifier uniquement les actions des aspects. Cette approche est coûteuse car toute modification d'un aspect entraîne une nouvelle vérification. La classification des aspects [CL02, RSB04, Kat06, DW06] identifie des catégories d'aspect en fonction de leurs impacts sur le programme de base. Les travaux utilisant cette démarche sont, soit incomplets car ils ne raisonnent pas sur les propriétés du programme de base, soit imprécis et informels lorsqu'ils raisonnent sur les propriétés.

En s'inspirant de l'approche par classification, nous avons identifié et défini formellement des catégories d'aspect. Ces catégories d'aspect sont les observateurs, les terminateurs, les verrouilleurs, les sélecteurs, les adaptateurs et les faiblement intrusifs.

- les observateurs ne modifient pas l'état et le flot de contrôle du programme de base. Les aspects de traçage et de profilage en sont des exemples.
- les terminateurs sont des observateurs qui peuvent arrêter l'exécution du programme de base. Les aspects de sécurité qui se contentent d'arrêter l'exécution du programme de base lorsqu'une propriété est violée sont typiquement des terminateurs.
- les verrouilleurs modifient l'état et le flot de contrôle du programme de base sans sortir de l'ensemble des états atteignables par celui-ci. Les aspects de tolérance aux fautes qui permettent de retourner vers un état précédent sûr sont des verrouilleurs si la restauration de l'état est vue comme une opération atomique.

- les faiblement intrusifs peuvent également modifier l'état et le flot de contrôle du programme de base, mais peuvent sortir de l'ensemble des états atteignables pendant l'exécution des actions. Un aspect de tolérance aux fautes avec une restauration non atomique des états précédents est faiblement intrusif.
- les sélecteurs existent uniquement dans le cas de programmes concurrents. Ils sélectionnent un sous ensemble d'exécutions dans l'ensemble des exécutions possibles. Les aspects d'ordonnancement, qui choisissent des exécutions parmi plusieurs exécutions, sont des sélecteurs.
- les adaptateurs sont des sélecteurs qui peuvent en plus arrêter l'exécution du programme.

Pour chacune de ces catégories, nous avons identifié une classe de propriétés du programme de base qui est préservée quelque soit les aspects tissés appartenant à la catégorie correspondante. Plus précisément, pour tout programme de base, pour tout aspect appartenant à une catégorie, si le programme de base satisfait une propriété appartenant à la classe de propriétés correspondant à la catégorie de l'aspect, alors le programme tissé satisfait aussi cette propriété. Nous avons exprimé les classes de propriétés comme des sous ensembles de LTL dans le cas des programmes séquentiels et CTL* dans le cas des programmes concurrents. Par exemple, hormis les propriétés qui portent sur l'opérateur "next", les observateurs permettent de préserver toutes les propriétés de LTL qui portent sur l'état du programme de base tandis qu'ils préservent uniquement les propriétés de type "eventually" ($true \cup \varphi'^{o}$) sur les événements du programme de base. Cela a été prouvé par induction sur la structure des propriétés. Pour les autres catégories, la structure de la preuve est similaire. Nos catégories d'aspect sont reliées par une relation d'inclusion, où les observateurs sont inclus dans les terminateurs et les sélecteurs qui ne peuvent pas être comparées mais qui sont tous les deux inclus dans les adaptateurs, qui sont inclus dans les verrouilleurs, qui à leur tour sont inclus dans les faiblement intrusifs. Les classes de propriétés sont reliées par la relation d'inclusion inverse. Ces relations d'inclusion donnent la catégorie d'appartenance de la composition entre aspects de nos catégories. Par exemple, la composition (ou l'interaction) entre un terminateur et un observateur appartient à la catégorie des terminateurs quelque soit l'ordre de composition. La composition entre un terminateur et un verrouilleur est dans la catégorie des verrouilleurs. Il suffit donc de connaître la catégorie des aspects, pour déterminer son impact sur le programme de base.

Pour compléter notre démarche, nous avons défini des langages qui assurent par construction qu'un aspect appartient à une catégorie. Nous avons proposé des langages d'aspect généraux pour les observateurs et les terminateurs. Pour les verrouilleurs, il est impossible de concevoir un langage d'aspect général, qui modifie arbitrairement les variables et le flot de contrôle du programme de base tout en restant dans l'ensemble de ses états atteignables. Nous avons proposé deux langages d'aspect dédiés appartenant à cette catégorie : un langage d'aspect dédié à la tolérance aux fautes qui permet de retourner vers des états précédents sûrs et un langage d'aspect dédié à la mémoisation, une optimisation qui permet d'atteindre plus rapidement des états futurs. Nous avons montré que tout aspect écrit avec le langage des observateurs est dans la catégorie correspondante. La preuve des terminateurs a une structure semblable à celle des observateurs, et celle pour les aspects de mémoisation est immédiate par définition. Pour les programmes concurrents, nous avons indiqué comment adapter les langages ci-dessus. Nous avons mis en avant des éléments pour la conception des langages d'aspect généraux pour les sélecteurs et les adaptateurs, et un langage d'aspect dédié `rollback` pour la tolérance aux

fautes.

Enfin, afin de montrer que notre cadre sémantique est expressif, nous l'avons utilisé pour décrire des constructions complexes de langages d'aspect classiques comme AspectJ et CaesarJ. Nous avons également montré que notre cadre sémantique peut être utilisé pour prouver la correction des transformations d'aspect effectuées lors du tissage.

Ce travail complète et clarifie les travaux précédents effectués dans l'approche par classification car il définit formellement des catégories d'aspect et les propriétés du programme de base que chacune d'elle préserve. Par rapport au travail de Katz [Kat06] et celui de Dantas et Walker [DW06], que nous estimons être les deux principaux représentants de cette approche, ce travail a permis d'améliorer celui de Katz en mettant en évidence des imprécisions, et d'enrichir celui de Dantas et Walker en traitant d'autres catégories que les aspects inoffensifs (les terminateurs) et en présentant les propriétés du programme de base qu'elles permettent de préserver sous la forme de logique temporelle. Cependant il reste incomplet car il existe des aspects qui n'appartiennent pas aux catégories d'aspect traitées. Au delà, ce travail est un cadre formel pour raisonner sur la programmation par aspect en général. Il permet notamment de décrire la sémantique du tissage et l'exécution du programme tissé, de définir les catégories d'aspect et les propriétés qu'elles préservent, ainsi que de faire les preuves de corrections.

Perspectives

Concernant nos catégories d'aspect et les classes de propriétés qu'elles permettent de préserver, un certain nombre de points mériterait d'être approfondis :

- En premier lieu, il faudrait montrer que nos classes de propriétés sont maximales. Nous devons prouver que, chaque classe peut exprimer exactement toutes les propriétés qui peuvent être préservées par la catégorie correspondante. Cette tâche est difficile car montrer qu'une classe est maximale, n'est pas un critère syntaxique, mais sémantique. Par exemple, la propriété $(ep \vee \neg ep) \cup \varphi'^{o}$, qui est préservée par les observateurs, n'est pas une propriété de φ^{o}. Cependant, il est sémantiquement équivalent à $true \cup \varphi'^{o}$, qui appartient à φ^{o}.
- Un second prolongement utile serait l'extension de nos catégories, avec des nouvelles catégories d'aspect définies par rapport à un sous ensemble des variables du programme de base. Ces catégories permettraient par exemple, aux aspects observateurs et terminateurs de pouvoir modifier des variables particulières (*par ex.,* les variables temporaires) de l'état du programme de base. Pour les verrouilleurs, certaines variables du programme de base, pourraient sortir de l'ensemble des états atteignables. C'est le cas par exemple, des aspects de tolérance aux fautes qui font une restauration non atomique des états précédents sûrs.
- Enfin, notre approche se focalise sur la préservation des propriétés pour tout aspect d'une catégorie et pour tout programme de base. Il serait intéressant d'étudier une approche moins générale, en fixant soit le programme de base, soit un aspect, soit une propriété.

On pourrait, par exemple, fixer une propriété φ et chercher une catégorie d'aspect $\mathcal{A}_x$ tel que, le tissage de tout aspect $A \in \mathcal{A}_x$ préserve φ. Il est également possible de fixer le programme de base car la classe de propriétés préservée par les observateurs pour un programme spécifique est plus grande que φ^o. Il en est de même pour la classe préservée par un aspect observateur spécifique, même si c'est pour tout programme de base. On peut aussi fixer deux paramètres, par exemple le programme de base et l'aspect et étudier l'ensemble des propriétés préservées. Le cas où le programme de base, l'aspect et la propriété sont fixés correspond à la vérification de programme dont une instance est présentée à la Section 1.3.2, page 27.

Concernant nos langages d'aspect spécialisés, nous souhaitons définir des langages d'aspect spécifiques qui assurent par construction que les aspects sont faiblement intrusifs. Comme pour les verrouilleurs, il semble impossible de définir un langage général pour cette catégorie. Le travail effectué sur les verrouilleurs et les programmes concurrents doit être amélioré en définissant de nouveaux langages d'aspect spécifiques.

A plus long terme, nous pensons que l'avenir de la programmation par aspect passe par l'utilisation des méthodes formelles et des langages dédiés afin de garantir une programmation simple et sûre. En effet, la complexité des langages généraux comme AspectJ et le peu de confiance qu'ils procurent (expressivité débridée associée à sémantique informelle) ne favorise pas l'essor de la programmation par aspect. Une suite logique à apporter à ce travail serait le développement d'un projet comme AspectSandBox [1] qui pourrait s'appuyer sur notre cadre formel pour décrire et étudier les sémantiques des différents langages d'aspect. Un autre point qui pourrait être également traité dans ce cadre est l'analyse des propriétés que les aspects peuvent ajouter au programme de base. Ces propriétés variants d'un aspect à un autre, un raisonnement par catégorie semble inapproprié pour ce cas. Notre projet pourrait en plus implémenter les langages de nos différentes catégories afin de proposer un cadre concret pour l'étude des catégories d'aspect ainsi que celui des langages d'aspect spécifiques et leur interaction.

[1] http ://www.cs.ubc.ca/labs/spl/projects/asb.html

Annexe A

Preuves du Chapitre 3

A.1 Preuves pour les observateurs

Cet appendice présente la preuve du Théorème 1. Cette preuve utilise deux fonctions auxiliaires $trace_b$ et rib. La fonction $trace_b$ projette les traces tissées sur les traces de base correspondantes. Il retire des traces tissées, les étapes avec une instruction de l'action (i_a) et projette tout état Σ sur celui du programme de base Σ^b qui lui correspond.

$$\begin{array}{l} trace_b :: Traces_{\mathcal{W}} \rightarrow Traces_{\mathcal{B}} \\ trace_b(i_b, \Sigma) : S \ = \ (i_b, \Sigma^b) : trace_b \ S \\ trace_b(i_a, \Sigma) : S \ = \ trace_b \ S \end{array}$$

La fonction $rib \ \alpha \ i$ retourne le rang de la $i^{\text{ème}}$ instruction de base dans une trace tissée $\tilde{\alpha}$. Si n instructions d'action ont été introduites/exécutées avant d'atteindre la $i^{\text{ème}}$ instruction de base alors $rib \ \tilde{\alpha} \ i = i + n$. Nous utilisons la notation $\tilde{i}$ pour $rib \ \tilde{\alpha} \ i$.

La preuve du Théorème 1 utilise aussi la propriété suivante qui indique que la trace d'exécution tissée avec un observateur peut être projetée ($trace_b$) sur la trace d'exécution de base.

PROPRIÉTÉ **35.**

$$\begin{array}{l} \forall (C, \Sigma). \ \Sigma^{\psi} \in \mathcal{A}_o \ \Rightarrow \ trace_b(\tilde{\alpha}) = \alpha \\ \textit{avec} \ \ \alpha = \mathcal{B}(C, \Sigma^b) \ \ \textit{et} \ \ \tilde{\alpha} = \mathcal{W}(C, \Sigma) \end{array}$$

Preuve : Par définition

$$\begin{array}{lll} \forall (C, \Sigma). \ \Sigma^{\psi} \in \mathcal{A}_o & \Leftrightarrow & proj_b(\alpha) = proj_b(\tilde{\alpha}) \\ & & \wedge \ preserve_b(\tilde{\alpha}) \end{array}$$

L'égalité des traces en utilisant $proj_b$ assure que toutes les actions terminent tant disque $preserve_b(\tilde{\alpha})$ assure que l'état de base ne change pas pendant l'exécution de l'action. Ainsi, $trace_b$ qui retire les étapes de l'action, l'aspect et son état, projette donc la trace tissée sur la trace de base. □

Lorsqu'une trace d'exécution tissée peut être projetée sur une trace de base, la $i^{\text{ème}}$ étape de la trace de base correspond à la $\tilde{i}^{\text{ème}}$ étape de la trace tissée.

LEMME **36.**

$$trace_b(\tilde{\alpha}) = \alpha \Rightarrow$$
$$\forall(j \geq 1).\ \alpha_j = (i_b, \Sigma^b) \Leftrightarrow \tilde{\alpha}_{\tilde{j}} = (i_b, \Sigma)$$

La preuve est triviale en utilisant la définition de rib et de $trace_b$. Le lemme suivant est aussi utilisé

LEMME **37.**

$$trace_b(\tilde{\alpha}) = \alpha \Rightarrow$$
$$\forall(i \geq 1).\ \forall(\widetilde{i-1} < j \leq \tilde{i}).trace_b(\tilde{\alpha}_{j\rightarrow}) = \alpha_{i\rightarrow}$$

Il indique que pour toute trace de base et tissée reliée par projection ($trace_b$), toute sous trace de base (resp. tissée) correspond à une sous trace tissée (resp. de base).
Preuve : Par induction sur la longueur de α et $\tilde{\alpha}$.
Cas de base

– $i = 1 \wedge \alpha_1 = (i_1 : \_, \Sigma_1^b)$

$trace_b(\tilde{\alpha}) = \alpha \ \wedge\ \Sigma^{\psi}(i_1 : \_, \Sigma_1) = nil \ \Rightarrow\ \tilde{\alpha}_1 = \tilde{\alpha}_{\tilde{1}}$ avec $j = \tilde{1}$
$\Rightarrow\ trace_b(\tilde{\alpha}_{j\rightarrow}) = \alpha_{i\rightarrow}$
car $trace_b(\tilde{\alpha}_{\tilde{1}}) = trace_b(\tilde{\alpha})$ et $\alpha_{1\rightarrow} = \alpha$

Par définition de rib
$trace_b(\tilde{\alpha}) = \alpha \wedge \Sigma^{\psi}(i_1 : \_, \Sigma_1) \neq nil$
$\quad \Rightarrow\ \exists(k > 1).\tilde{\alpha}_k = \tilde{\alpha}_{\tilde{1}} \ \wedge\ \forall(k' < k).\tilde{\alpha}_{k'} = (i_a, \_)$
Par définition de $trace_b$
$\Rightarrow\ \forall(j \geq k).trace_b(\tilde{\alpha}_{j\rightarrow}) = \alpha_{1\rightarrow}$

Induction
On suppose que pour $i = n$

$$\forall(\widetilde{i-1} < j \leq \tilde{i}).trace_b(\tilde{\alpha}_{j\rightarrow}) = \alpha_{i\rightarrow}$$

et nous montrons que c'est aussi le cas pour $i = n + 1$

– $i = n+1 \wedge \Sigma^{\psi}(i_{n+1} : \_, \Sigma_{n+1})$

$trace_b(\tilde{\alpha}) = \alpha \ \wedge \ \Sigma^{\psi}(i_{n+1} : \_, \Sigma_{n+1}) = nil$
$\Rightarrow \ \exists(k \geq n+1).\tilde{\alpha}_k = \tilde{\alpha}_{\widetilde{n+1}} \ \wedge \ (\tilde{\alpha}_{k-1} = \tilde{\alpha}_{\tilde{n}} \ \vee \ \forall(\tilde{n} < k' < \widetilde{n+1}).\tilde{\alpha}_{k'} = (i_a, \_))$
Par définition de $trace_b$
$\Rightarrow \ \forall(\tilde{n} < j \leq \widetilde{n+1}).trace_b(\tilde{\alpha}_{j\rightarrow}) = \alpha_{n+1\rightarrow}$

En effet,

– si $\tilde{\alpha}_k = \tilde{\alpha}_{\widetilde{n+1}} \ \wedge \ \tilde{\alpha}_{k-1} = \tilde{\alpha}_{\tilde{n}}$
$j = n+1$ et comme l'hypothèse d'induction implique
$trace_b(\tilde{\alpha}_{\tilde{n}} : \tilde{\alpha}_{\widetilde{n+1}\rightarrow}) = \alpha_n : \alpha_{n+1\rightarrow}$
on a donc par définition de $trace_b$,
$trace_b(\tilde{\alpha}_{\tilde{n+1}\rightarrow}) = \alpha_{n+1\rightarrow}$
– si $\tilde{\alpha}_k = \tilde{\alpha}_{\widetilde{n+1}} \ \wedge \ \forall(\tilde{n} < k' < \widetilde{n+1}).\tilde{\alpha}_{k'} = (i_a, \_)$
comme l'hypothèse d'induction implique
$trace_b(\tilde{\alpha}_{\tilde{n}} : (i_a, \_) : \ldots : (i_a, \_) : \tilde{\alpha}_{\widetilde{n+1}\rightarrow}) = \alpha_n : \alpha_{n+1\rightarrow}$
on a donc par définition de $trace_b$,
$\forall(\tilde{n} < j \leq \widetilde{n+1}).trace_b(\tilde{\alpha}_{j\rightarrow}) = \alpha_{n+1\rightarrow}$

$trace_b(\tilde{\alpha}) = \alpha \ \wedge \ \Sigma^{\psi}(i_{n+1} : \_, \Sigma_{n+1}) \neq nil$
$\Rightarrow \ \exists(k \geq n+1).\tilde{\alpha}_k = \tilde{\alpha}_{\widetilde{n+1}} \ \wedge \ \forall(\tilde{n} < k' < \widetilde{n+1}).\tilde{\alpha}_{k'} = (i_a, \_)$
identique au sous cas précédent

□

Le Théorème 1 est montré en prouvant la propriété plus générale

$$\begin{array}{ll} \Sigma^{\psi} \in \mathcal{A}_o \wedge trace_b(\tilde{\alpha}) = \alpha \ \Rightarrow & \forall(p \in \varphi^o).\alpha \models p \ \Rightarrow \ \tilde{\alpha} \models p \\ \text{avec } \ \tilde{\alpha}_1 = (x, \Sigma) & \wedge \forall(p' \in \varphi'^o).\ \forall(j \geq 1). \\ & \alpha_{j\rightarrow} \models p' \ \Rightarrow \ \tilde{\alpha}_{\tilde{j}\rightarrow} \models p' \end{array}$$

Quand une trace tissée peut être projetée sur une trace de base et l'aspect initial est un observateur alors deux conséquences suivent. La première correspond au Théorème 1 tant disque la deuxième concerne les propriétés de φ'^o qui interviennent uniquement dans les formules de la forme $true \cup \varphi'^o$. Pour une telle propriété p', toute sous trace de base satisfaisant p', a une sous trace tissée satisfaisant p' qui lui correspond. La preuve du Théorème 1 découle directement de cette propriété générale.

Preuve : Par induction sur la structure des formules de φ^o et de φ'^o.
Cas de base

– $p = sp \ \in \varphi^o$

$$\begin{array}{l} \alpha \models sp \ \Rightarrow \ \alpha_1 \models sp \\ \Leftrightarrow l(\Sigma_1^b, sp) = true \ \text{ avec } \ \alpha_1 = (i_1, \Sigma_1^b) \end{array}$$

$$trace_b(\tilde{\alpha}) = \alpha \ \Rightarrow \ \tilde{\alpha}_{\tilde{1}} = (i_1, \Sigma_1) \quad \text{par le Lemme 36}$$

Notons que $\tilde{\alpha}_{\tilde{1}}$ peut ne pas être la première étape de la trace tissée. C'est uniquement l'étape avec la première instruction de base. Comme $\Sigma^{\psi} \in \mathcal{A}_o$, la première étape de la trace tissée $\tilde{\alpha}_1 = (i'_1, \Sigma'_1)$ est telle que $\Sigma'^b_1 = \Sigma^b_1$ (l'état de base n'est pas modifié par une action `before`) et, comme les propriétés sur les états ne prennent en compte que Σ^b, alors

$$\begin{array}{rl} l(\Sigma^b_1, sp) & \Rightarrow\ l(\Sigma'_1, sp) \\ & \Rightarrow\ \tilde{\alpha}_1 \models sp \\ & \Rightarrow\ \tilde{\alpha} \models sp \end{array}$$

– $p = ep \in \varphi'^o$

$\forall(j \geq 1).\ \alpha_{j\rightarrow} \models ep \Rightarrow \alpha_j \models ep \Rightarrow m(i_j, ep)$
Par le Lemme 36
$\alpha_j = (i_j, \Sigma^b) \Rightarrow \tilde{\alpha}_{\tilde{j}} = (i_j, \Sigma)$
d'où $m(i_j, ep) = m(\tilde{\alpha}_{\tilde{j}}, ep)$
et $\tilde{\alpha}_{\tilde{j}} \models ep$
donc $\tilde{\alpha}_{\tilde{j}\rightarrow} \models ep$

– $p = \neg sp \in \varphi^o$ et $p = \neg ep, sp, \neg sp \in \varphi'^o$ sont similaires aux cas précédents.

Induction

Pour toute sous formule δ de φ^o l'hypothèse d'induction est :

$$\alpha \models \delta \Rightarrow \tilde{\alpha} \models \delta$$

avec pour toute sous formule δ de φ'^o :

$$\forall(j \geq 1).\ \alpha_{j\rightarrow} \models \delta \Rightarrow \tilde{\alpha}_{\tilde{j}\rightarrow} \models \delta$$

Pour appliquer ces hypothèses, il suffit de vérifier que les traces correspondantes sont en relation (*c.-à-d.,* $trace_b(\tilde{\alpha}) = \alpha$). Pour la seconde condition, comme l'aspect initial est observateur, toutes les instances de celui-ci qui seront éventuellement créées le seront aussi.

– $p = \varphi^o_1 \wedge \varphi^o_2 \in \varphi^o$

$\alpha \models \varphi^o_1 \wedge \varphi^o_2$
$\Rightarrow \alpha \models \varphi^o_1 \wedge \alpha \models \varphi^o_2$
$\Rightarrow \tilde{\alpha} \models \varphi^o_1 \wedge \tilde{\alpha} \models \varphi^o_2$ par hypothèse d'induction
$\Rightarrow \tilde{\alpha} \models \varphi^o_1 \wedge \varphi^o_2$

– $p = \varphi^o_1 \vee \varphi^o_2 \in \varphi^o$ est similaire au cas précédent

– $p = \varphi_1^o \cup \varphi_2^o \ \in \varphi^o$

$$\alpha \models \varphi_1^o \cup \varphi_2^o \ \Rightarrow \ \exists(j \geq 1).\ \alpha_{j\rightarrow} \models \varphi_2^o \wedge \forall(1 \leq i < j).\ \alpha_{i\rightarrow} \models \varphi_1^o$$

Par le Lemme 37
$trace_b(\tilde{\alpha}) = \alpha \ \Rightarrow \ trace_b(\tilde{\alpha}_{\widetilde{j-1+1}\rightarrow}) = \alpha_{j\rightarrow}$
$\Rightarrow \ \tilde{\alpha}_{\widetilde{j-1+1}\rightarrow} \models \varphi_2^o$ par hypothèse d'induction
$\Rightarrow \ \exists(k \geq 1).\tilde{\alpha}_{k\rightarrow} \models \varphi_2^o$ avec $k = \widetilde{j-1}+1$

$\forall(1 \leq l < k).\ \exists(1 \leq i < j).$
$k = \widetilde{j-1}+1 \ \wedge \ \widetilde{i-1} < l \leq \tilde{i}$
d'où $trace_b(\tilde{\alpha}_{l\rightarrow}) = \alpha_{i\rightarrow}$ par le Lemme 37
et comme $\alpha_{i\rightarrow} \models \varphi_1^o$ pour de tels i
$\tilde{\alpha}_{l\rightarrow} \models \varphi_1^o$ par hypothèse d'induction

Ainsi $\tilde{\alpha} \models \varphi_1^o \cup \varphi_2^o$

– $p = true \cup \varphi'^o \ \in \varphi^o$

$$\alpha \models true \cup \varphi'^o \ \Rightarrow \ \exists(j \geq 1).\ \alpha_{j\rightarrow} \models \varphi'^o \wedge \forall(1 \leq i < j).\ \alpha_{i\rightarrow} \models true$$

par hypothèse d'induction, nous avons

$$\tilde{\alpha}_{\tilde{j}\rightarrow} \models \varphi'^o$$

d'où, en prenant $k = \tilde{j}$, nous avons $(\exists k \geq 1).\ \tilde{\alpha}_{k\rightarrow} \models \varphi'^o$
et comme $\forall(1 \leq l < \tilde{j}).\ \tilde{\alpha}_{l\rightarrow} \models true$
nous avons

$$\tilde{\alpha} \models true \cup \varphi'^o$$

– $p = \varphi_1^o W \varphi_2^o \ \in \varphi^o$ est similaire au cas précédent

– $p = \varphi_1'^o \ \wedge \ \varphi_2'^o \ \in \varphi'^o$

$\forall(j \geq 1).\ \alpha_{j\rightarrow} \models \varphi_1'^o \ \wedge \ \varphi_2'^o$
$\Rightarrow \ \alpha_{j\rightarrow} \models \varphi_1'^o \ \wedge \ \alpha_{j\rightarrow} \models \varphi_2'^o$
$\Rightarrow \ \tilde{\alpha}_{\tilde{j}\rightarrow} \models \varphi_1'^o \ \wedge \ \tilde{\alpha}_{\tilde{j}\rightarrow} \models \varphi_2'^o$ par hypothèse d'induction
$\Rightarrow \ \tilde{\alpha}_{\tilde{j}\rightarrow} \models \varphi_1'^o \ \wedge \ \varphi_2'^o$

– $p = \varphi_1'^o \ \vee \ \varphi_2'^o \ \in \varphi'^o$ est similaire au cas précédent

– $p = \varphi_1^o \cup \varphi_2^o \ \in \varphi'^o$

$$\forall(j \geq 1).\ \alpha_{j\rightarrow} \models \varphi_1^o \cup \varphi_2^o \ \Rightarrow\ \exists(k \geq j).\ \alpha_{k\rightarrow} \models \varphi_2^o \wedge \forall(j \leq l < k).\ \alpha_{l\rightarrow} \models \varphi_1^o$$

Par le Lemme 37
$trace_b(\tilde{\alpha}) = \alpha \Rightarrow\ trace_b(\tilde{\alpha}_{\widetilde{k-1}+1\rightarrow}) = \alpha_{k\rightarrow}$
$\Rightarrow\ \widetilde{\alpha}_{\widetilde{k-1}+1\rightarrow} \models \varphi_2^o$ par hypothèse d'induction
$\Rightarrow\ \exists(m \geq \tilde{j} \geq j).\tilde{\alpha}_{m\rightarrow} \models \varphi_2^o$ en prenant $m = \widetilde{k-1}+1$

$\forall(j \leq n < m).\ \exists(j \leq l < k).$
$m = \widetilde{k-1}+1\ \wedge\ \widetilde{l-1} < n \leq \tilde{l}$
d'où $trace_b(\tilde{\alpha}_{n\rightarrow}) = \alpha_{l\rightarrow}$ par le Lemme 37
et comme $\alpha_{l\rightarrow} \models \varphi_1^o$ pour de tels l
$\Rightarrow\ \tilde{\alpha}_{n\rightarrow} \models \varphi_1^o$ par hypothèse d'induction

Ainsi $\exists(m \geq \tilde{j} \geq j).\tilde{\alpha}_{m\rightarrow} \models \varphi_2^o$
$\wedge\ \ \forall(j \leq \tilde{j} \leq n < m).\tilde{\alpha}_{n\rightarrow} \models \varphi_1^o$

et donc $\tilde{\alpha}_{\tilde{j}\rightarrow} \models \varphi_1^o \cup \varphi_2^o$

– $p = true \cup \varphi'^o \ \in \varphi'^o$

$$\begin{array}{ll} \forall(j \geq 1). & \alpha_{j\rightarrow} \models true \cup \varphi'^o \\ \Rightarrow & \exists(k \geq j).\ \alpha_{k\rightarrow} \models \varphi'^o \wedge \\ & \forall(j \leq i < k).\ \alpha_{i\rightarrow} \models true \end{array}$$

par hypothèse d'induction, nous avons

$$\tilde{\alpha}_{\tilde{k}\rightarrow} \models \varphi'^o$$

d'où $k \geq j\ \Rightarrow\ \tilde{k} \geq \tilde{j}$ et comme

$$\forall(\tilde{j} \leq l < \tilde{k}).\ \tilde{\alpha}_{l\rightarrow} \models true$$

alors

$$\tilde{\alpha}_{\tilde{j}\rightarrow} \models true \cup \varphi'^o$$

– $p = \varphi_1^o W \varphi_2^o \ \in \varphi'^o$ est similaire au cas précédent

□

Annexe B

Preuves et Sémantiques du Chapitre 4

B.1 Sémantique de $Prog$

Dans cette annexe, nous présentons la sémantique du langage de base utilisé au Chapitre 4. Ce langage $Prog$, est un langage impératif dont la syntaxe est présentée à la Section 4.2, page 78. Nous décrivons l'évaluation des expressions arithmétiques (A), des expressions booléennes (B) et des commandes (S) qui sont les catégories syntaxiques de ce langage.

B.1.1 Evaluation des expressions arithmétiques : $\mathcal{E}_a$

L'évaluation des expressions arithmétiques est éffectuée par la fonction $\mathcal{E}_a$ qui prend une expression arithmétique en paramètre et retourne une fonction qui, en prenant les environnements des variables globales et locales retourne la valeur de l'expression. Elle a la signature suivante

$$\mathcal{E}_a : \textbf{Aexp} \rightarrow (\Sigma^b_g \times \Sigma^b_{l1} \rightarrow \mathbb{Z})$$

où **Aexp** est l'ensemble des expressions arithmétiques du langage, Σ^b_g et Σ^b_{l1} sont des fonctions qui associent respectivement à chaque variable globale et locale (ici, uniquement les paramètres des procédures sont utilisés comme variables locales) leurs valeurs qui est dans $\mathbb{Z}$ l'ensemble des entiers relatifs.

$$\begin{array}{lcl}
\mathcal{N} : \textbf{Num} \rightarrow \mathbb{Z} & & \\
\mathcal{E}_a[\![n]\!](\Sigma^b_g, \Sigma^b_{l1}) & = & \mathcal{N}[\![n]\!] \\
\mathcal{E}_a[\![g]\!](\Sigma^b_g, \Sigma^b_{l1}) & = & \Sigma^b_g(g) \\
\mathcal{E}_a[\![l]\!](\Sigma^b_g, \Sigma^b_{l1}) & = & \Sigma^b_{l1}(l) \\
\mathcal{E}_a[\![A_1{+}A_2]\!](\Sigma^b_g, \Sigma^b_{l1}) & = & \mathcal{E}_a[\![A_1]\!](\Sigma^b_g, \Sigma^b_{l1}) + \mathcal{E}_a[\![A_2]\!](\Sigma^b_g, \Sigma^b_{l1})
\end{array}$$

Ainsi, en supposant que la fonction $\mathcal{N}$ pour chaque numéral syntaxique (**Num**) retourne la

valeur correspondante, l'évaluation de n par $\mathcal{E}_a$ consiste à appliquer la fonction $\mathcal{N}$ à n. L'évaluation d'une variable globale g retourne sa valeur stockée dans l'environnement des variables globales. Celle d'un paramètre l retourne sa valeur qui est stockée dans l'environnement des paramètres. L'évaluation de $A_1 + A_2$ est la somme de l'évaluation A_1 et A_2.

B.1.2 Evaluation des expressions booléennes : $\mathcal{E}_b$

$\mathcal{E}_b$ est la fonction qui éffectue l'évaluation des expressions booléennes. Pour cela, elle prend une expression booléenne et retourne une fonction qui prend les environnements des variables globales et locales et retourne un booléen (élément de l'ensemble **Bool**). Sa signature est la suivante

$$\mathcal{E}_b : \textbf{Bexp} \rightarrow (\Sigma_g^b \times \Sigma_{l1}^b \rightarrow \textbf{Bool})$$

où **Bexp** est l'ensemble des expressions booléennes

$$\textbf{Bool} = \{\text{tt}, \text{ff}\}$$
$$\mathcal{E}_b[\![true]\!](\Sigma_g^b, \Sigma_{l1}^b) = \text{tt}$$
$$\mathcal{E}_b[\![A_1{=}A_2]\!](\Sigma_g^b, \Sigma_{l1}^b) = \begin{cases} \text{tt } \textit{if } \mathcal{E}_a[\![A_1]\!](\Sigma_g^b, \Sigma_{l1}^b) = \mathcal{E}_a[\![A_2]\!](\Sigma_g^b, \Sigma_{l1}^b) \\ \text{ff } \textit{if } \mathcal{E}_a[\![A_1]\!](\Sigma_g^b, \Sigma_{l1}^b) \neq \mathcal{E}_a[\![A_2]\!](\Sigma_g^b, \Sigma_{l1}^b) \end{cases}$$
$$\mathcal{E}_b[\![A_1{<}A_2]\!](\Sigma_g^b, \Sigma_{l1}^b) = \begin{cases} \text{tt } \textit{if } \mathcal{E}_a[\![A_1]\!](\Sigma_g^b, \Sigma_{l1}^b) < \mathcal{E}_a[\![A_2]\!](\Sigma_g^b, \Sigma_{l1}^b) \\ \text{ff } \textit{if } \mathcal{E}_a[\![A_1]\!](\Sigma_g^b, \Sigma_{l1}^b) \geq \mathcal{E}_a[\![A_2]\!](\Sigma_g^b, \Sigma_{l1}^b) \end{cases}$$
$$\mathcal{E}_b[\![B_1 \& B_2]\!](\Sigma_g^b, \Sigma_{l1}^b) = \begin{cases} \text{tt } \textit{if } \mathcal{E}_b[\![B_1]\!](\Sigma_g^b, \Sigma_{l1}^b) = \text{tt et } \mathcal{E}_b[\![B_2]\!](\Sigma_g^b, \Sigma_{l1}^b) = \text{tt} \\ \text{ff } \textit{if } \mathcal{E}_b[\![B_1]\!](\Sigma_g^b, \Sigma_{l1}^b) = \text{ff ou } \mathcal{E}_b[\![B_2]\!](\Sigma_g^b, \Sigma_{l1}^b) = \text{ff} \end{cases}$$
$$\mathcal{E}_b[\![\,!b]\!](\Sigma_g^b, \Sigma_{l1}^b) = \begin{cases} \text{tt } \textit{if } \mathcal{E}_b[\![b]\!](\Sigma_g^b, \Sigma_{l1}^b) = \text{ff} \\ \text{ff } \textit{if } \mathcal{E}_b[\![b]\!](\Sigma_g^b, \Sigma_{l1}^b) = \text{tt} \end{cases}$$

$\mathcal{E}_b$ utilise $\mathcal{E}_a$ pour évaluer les expressions booléennes. Notons qu'il fasse distinguer la syntaxe (*par ex.,* $<$ et $true$ à gauche) et la sémantique (*par ex.,* $<$ et tt à droite).

B.1.3 Sémantique de S

$$\text{SET}\ \frac{\mathcal{E}_a[\![A]\!](\Sigma_g^b, \Sigma_{l1}^b) = \nu}{(g := A : C, \Sigma_g^b \cup \Sigma_{l1}^b : \Sigma_{ls}^b \cup \Sigma_r^b) \rightarrow_b (C, \Sigma_g^b[g \mapsto \nu] \cup \Sigma_{l1}^b : \Sigma_{ls}^b \cup \Sigma_r^b)}$$

$$\text{SET}\ \frac{\mathcal{E}_a[\![A]\!](\Sigma_g^b, \Sigma_{l1}^b) = \nu}{(l := A : C, \Sigma_g^b \cup \Sigma_{l1}^b : \Sigma_{ls}^b \cup \Sigma_r^b) \rightarrow_b (C, \Sigma_g^b \cup \Sigma_{l1}^b[l \mapsto \nu] : \Sigma_{ls}^b \cup \Sigma_r^b)}$$

$$\text{CALL}\ \frac{pbody(p) = ((l_1, \ldots, l_n), S) \qquad \mathcal{E}_a[\![A_1]\!](\Sigma_g^b, \Sigma_{l1}^b) = \nu_1 \ldots \mathcal{E}_a[\![A_n]\!](\Sigma_g^b, \Sigma_{l1}^b) = \nu_n}{\begin{array}{c}(p(A_1, \ldots, A_n) : C, \Sigma_g^b \cup \Sigma_{l1}^b : \Sigma_{ls}^b \cup \Sigma_r^b) \rightarrow_b \\ (S : return, \Sigma_g^b \cup \{l_1 \mapsto \nu_1, \ldots, l_n \mapsto \nu_n\} : \Sigma_{l1}^b : \Sigma_{ls}^b \cup C : \Sigma_r^b)\end{array}}$$

$$\text{RETURN}\ \frac{}{(return, \Sigma_g^b \cup \Sigma_{l1}^b : \Sigma_{ls}^b \cup C : \Sigma_r^b) \rightarrow_b (C, \Sigma_g^b \cup \Sigma_{ls}^b \cup \Sigma_r^b)}$$

$$\text{ABORT}\ \frac{}{(abort : C, \Sigma_g^b \cup \Sigma_l^b \cup \Sigma_r^b) \rightarrow_b (\epsilon : \bullet, \Sigma_g^b \cup \bot : \epsilon \cup \epsilon)}$$

$$\text{SKIP}\ \frac{}{(skip : C, \Sigma_g^b \cup \Sigma_l^b \cup \Sigma_r^b) \rightarrow_b (C, \Sigma_g^b \cup \Sigma_l^b \cup \Sigma_r^b)}$$

$$\text{SEQ}\ \frac{}{(S_1; S_2 : C, \Sigma_g^b \cup \Sigma_l^b \cup \Sigma_r^b) \rightarrow_b (S_1 : S_2 : C, \Sigma_g^b \cup \Sigma_l^b \cup \Sigma_r^b)}$$

$$\text{IF1}\ \frac{\mathcal{E}_b[\![B]\!](\Sigma_g^b, \Sigma_{l1}^b) = \text{tt}}{(if(B)\ then\ S_1\ else\ S_2 : C, \Sigma_g^b \cup \Sigma_{l1}^b : \Sigma_{ls}^b \cup \Sigma_r^b) \rightarrow_b \quad (S_1 : C, \Sigma_g^b \cup \Sigma_{l1}^b : \Sigma_{ls}^b \cup \Sigma_r^b)}$$

$$\text{IF2}\ \frac{\mathcal{E}_b[\![B]\!](\Sigma_g^b, \Sigma_{l1}^b) = \text{ff}}{(if(B)\ then\ S_1\ else\ S_2 : C, \Sigma_g^b \cup \Sigma_{l1}^b : \Sigma_{ls}^b \cup \Sigma_r^b) \rightarrow_b \quad (S_2 : C, \Sigma_g^b \cup \Sigma_{l1}^b : \Sigma_{ls}^b \cup \Sigma_r^b)}$$

$$\text{WHILE1}\ \frac{\mathcal{E}_b[\![B]\!](\Sigma_g^b, \Sigma_{l1}^b) = \text{ff}}{(while(B)\ S : C, \Sigma_g^b \cup \Sigma_{l1}^b : \Sigma_{ls}^b \cup \Sigma_r^b) \rightarrow_b \quad (C, \Sigma_g^b \cup \Sigma_{l1}^b : \Sigma_{ls}^b \cup \Sigma_r^b)}$$

$$\text{WHILE2}\ \frac{\mathcal{E}_b[\![B]\!](\Sigma_g^b, \Sigma_{l1}^b) = \text{tt}}{\begin{array}{l}(while(B)\ S : C, \Sigma_g^b \cup \Sigma_{l1}^b : \Sigma_{ls}^b \cup \Sigma_r^b) \rightarrow_b \\ \quad (S : while(B)\ S : C, \Sigma_g^b \cup \Sigma_{l1}^b : \Sigma_{ls}^b \cup \Sigma_r^b)\end{array}}$$

$$\text{LOOP1}\ \frac{\mathcal{E}_a[\![A]\!](\Sigma_g^b, \Sigma_{l1}^b) \leq 0}{(loop(A)\ S : C, \Sigma_g^b \cup \Sigma_{l1}^b : \Sigma_{ls}^b \cup \Sigma_r^b) \rightarrow_b \quad (C, \Sigma_g^b \cup \Sigma_{l1}^b : \Sigma_{ls}^b \cup \Sigma_r^b)}$$

$$\text{LOOP2}\ \frac{\mathcal{E}_a[\![A]\!](\Sigma_g^b, \Sigma_{l1}^b) = n \wedge n \geq 1}{(loop(A)\ S : C, \Sigma_g^b \cup \Sigma_{l1}^b : \Sigma_{ls}^b \cup \Sigma_r^b) \rightarrow_b \quad (S^1 : \ldots : S^n : C, \Sigma_g^b \cup \Sigma_{l1}^b : \Sigma_{ls}^b \cup \Sigma_r^b)}$$

$$\text{FINAL}\ \frac{}{(\epsilon : \bullet, \Sigma_g^b \cup \bot : \epsilon \cup eps) \rightarrow_b (\epsilon : \bullet, \Sigma_g^b \cup \bot : \epsilon \cup \epsilon)}$$

B.2 Définition des fonctions utilisées par memo

$read_{\text{A}}$

$$\begin{array}{lcl} ReadA : \textbf{Aexp} \rightarrow \mathbb{P}(Var_g) & & \\ read_{\text{A}}[\![n]\!] & = & \emptyset \\ read_{\text{A}}[\![g]\!] & = & \{g\} \\ read_{\text{A}}[\![l]\!] & = & \emptyset \\ read_{\text{A}}[\![A_1 + A_2]\!] & = & read_{\text{A}}[\![A_1]\!] \cup read_{\text{A}}[\![A_2]\!] \end{array}$$

$read_{\text{B}}$

$$
\begin{array}{lcl}
read_{\text{B}}[\![true]\!] & = & \emptyset \\
read_{\text{B}}[\![A_1 = A_2]\!] & = & read_{\text{A}}[\![A_1]\!] \cup read_{\text{A}}[\![A_2]\!] \\
read_{\text{B}}[\![B_1 \& B_2]\!] & = & read_{\text{B}}[\![B_1]\!] \cup read_{\text{B}}[\![B_2]\!] \\
read_{\text{B}}[\![\ !B]\!] & = & ReadB[\![B]\!]
\end{array}
$$

$write$

$$
\begin{array}{l}
write : Decl \rightarrow Identifier \rightarrow List(\mathcal{V}ar) \\
\begin{array}{lcl}
write[\![D_1; D_2]\!] & = & write[\![D_1]\!] \uplus write[\![D_2]\!] \\
write[\![\texttt{proc}\ p\ \texttt{(...)}\ s]\!] & = & [p \mapsto mklist(write_{\text{S}}[\![s]\!]\{p\})] \\
write[\![\texttt{var}\ g\ \texttt{:=}A]\!] & = & \bot
\end{array} \\
write_{\text{S}} : Statement \rightarrow \mathbb{P}(Identifier) \rightarrow \mathbb{P}(\mathcal{V}ar) \\
\begin{array}{lcl}
write_{\text{S}}[\![g\ \texttt{:=}A]\!]ps & = & \{g\} \\
write_{\text{S}}[\![l\ \texttt{:=}A]\!]ps & = & \emptyset \\
write_{\text{S}}[\![S_1\ \texttt{;}\ S_2]\!]ps & = & write_{\text{S}}[\![S_1]\!]ps \\
 & & \cup\ write_{\text{S}}[\![S_2]\!]ps \\
write_{\text{S}}[\![\texttt{while(}B\texttt{)}\ S]\!]ps & = & write_{\text{S}}[\![S]\!]ps \\
write_{\text{S}}[\![\texttt{loop(}a\texttt{)}\ s]\!]ps & = & write_{\text{S}}[\![S]\!]ps \\
write_{\text{S}}[\![\texttt{if(}B\texttt{)}\ \texttt{then}\ S_1 & = & write_{\text{S}}[\![s_1]\!]ps \\
\quad \texttt{else}\ S_2]\!]ps & & \cup\ write_{\text{S}}[\![S_2]\!]ps \\
write_{\text{S}}[\![p(A_1, \ldots, A_n)]\!]ps & = & write_{\text{S}}[\![body(p)]\!](ps \cup \{p\})\ si\ p \notin ps \\
write_{\text{S}}[\![\_]\!]ps & = & \emptyset \quad sinon
\end{array}
\end{array}
$$

B.3 Preuves pour les observateurs

La preuve qui est faite ici, est celle de la propriété 22. Elle repose sur la propriété 38 qui implique directement la propriété 22 par définition de $\mathcal{A}_o$.

PROPRIÉTÉ **38.**

$$
\begin{array}{l}
\forall(a \in Asp^o).\forall(C, \Sigma).\ \Sigma^\psi = [\![a]\!] \\
\quad \Rightarrow\ proj_b(\alpha) = proj_b(\tilde{\alpha})\ \wedge\ preserve_b(\tilde{\alpha}) \\
avec\ \ \alpha = \mathcal{B}(C, \Sigma^b)\ \ et\ \ \tilde{\alpha} = \mathcal{W}(C, \Sigma)
\end{array}
$$

La propriété 38 est prouvée en utilisant les Lemmes 39 et 41 qui montrent respectivement que les aspects ne modifient pas l'état et le flot de contrôle du programme de base. Cette preuve,

utilise les fonctions $proj$ (qui supprime les instructions i_a et les états Σ d'une trace) et $preserve_b$ (qui vérifie que les instructions i_a ne modifient pas Σ^b) définies à la Section 3.2, page 56. Aussi, nous rappelons que si α est une trace alors son $i^{\text{ème}}$ élément est noté α_i et son préfixe $\alpha_1 : \ldots : \alpha_j$ est noté $\alpha_{\rightarrow j}$.

LEMME **39.**

$$\forall(a \in Asp^o).\forall(C,\Sigma).\ \Sigma^\psi = [\![a]\!] \ \Rightarrow\ preserve_b(\tilde{\alpha})$$
$$\textit{avec}\ \ \tilde{\alpha} = \mathcal{W}(C,\Sigma)$$

Preuve : Il est facile de voir (preuve par cas) que toutes les instructions i_a de $\{S^o; \texttt{proceed}; S^o\}$ modifient uniquement Σ^a après une réduction par $\rightarrow$. En effet, les instructions de S^o modifient seulement les variables des aspects et comme la commande `proceed` modifie uniquement la pile proceed Σ^P qui est un sous ensemble de Σ^a ($\Sigma^P \subset \Sigma^a$), la preuve est immédiate. □

Pour prouver le Lemme 41, nous montrons d'abord le Lemme 40 qui exprime que pour tout préfixe de α, il existe un préfixe de $\tilde{\alpha}$ tel que les deux préfixes ont la même séquence d'instructions du programme de base.

LEMME **40.**

$$\forall(a \in Asp^o).\forall(C,\Sigma).\ \Sigma^\psi = [\![a]\!]$$
$$\Rightarrow\ \forall(l \geq 1).\exists(m \geq l).\ proj_b(\alpha_{\rightarrow l}) = proj_b(\tilde{\alpha}_{\rightarrow m})$$
$$\textit{avec}\ \ \alpha = \mathcal{B}(C,\Sigma^b)\ \ \textit{et}\ \ \tilde{\alpha} = \mathcal{W}(C,\Sigma)$$

Preuve : Par induction sur la longueur de α et $\tilde{\alpha}$ et en supposant que l'action termine (Hypothèse 1).

HYPOTÈSE **1.**

$$\forall(D^o\ \texttt{around}\ P\ \{s\} \in Asp^o).\ s\ \ termine$$

Par l'Hypothèse 1 on a
$(\forall(j \geq 1).\ \tilde{\alpha}_j = (i_a, \_)\ \Rightarrow\ \exists(k > j).\ \tilde{\alpha}_k = (i_b, \_))$

Cas de base $\quad l = 1$
$\alpha_{\rightarrow 1} = (i_1, \_)$
$\Sigma^\psi(i_1 : \_, \_) = nil\ \Rightarrow\ \tilde{\alpha}_{\rightarrow 1} = (i_1, \_)$
par définition de $\mathcal{W}(C,\Sigma)$
$\Rightarrow\ proj_b(\alpha_{\rightarrow 1}) = proj_b(\tilde{\alpha}_{\rightarrow 1})$
par définition de $proj_b$
$\alpha_{\rightarrow 1} = (i_1, \_)$
$\Sigma^\psi(i_1 : \_\,, \_\,) \neq nil\ \Rightarrow\ \tilde{\alpha}_{\rightarrow 1} = (i_a, \_\,)$
par définition de $\mathcal{W}(C,\Sigma)$
$\Rightarrow\ \exists(m > 1).\ \tilde{\alpha}_m = (i_1, \_\,) \wedge$
$\forall(m' < m).\ \tilde{\alpha}_{m'} = (i_a, \_\,)$
par l'Hypothèse 1, et définition de $\mathcal{W}(C,\Sigma)$
$\Rightarrow \exists(m > 1).\ proj_b(\alpha_{\rightarrow 1}) = proj_b(\tilde{\alpha}_{\rightarrow m})$
par définition de $proj_b$

Cas d'induction $l = n$
Nous supposons que

$$\exists(m \geq n).\ proj_b(\alpha_{\rightarrow n}) = proj_b(\tilde{\alpha}_{\rightarrow m})$$

et montrons que c'est le cas pour $l = n + 1$
$\alpha_{\rightarrow n+1} = \alpha_1 : \ldots : \alpha_n : \alpha_{n+1} \ \wedge \ \alpha_{n+1} = (i_{n+1}, \_\ )$
$\Sigma^{\psi}(i_{n+1} : \_\ , \_\ ) = nil$
$\Rightarrow \ \exists(m' = m + 1 \geq n + 1).\ \tilde{\alpha}_{m'} = (i_{n+1}, \_\ )$
$\vee(\exists(m' > m + 1).\ \tilde{\alpha}_{m'} = (i_{n+1}, \_\ )$
$\wedge \forall(m < m'' < m').\ \tilde{\alpha}_{m''} = (i_a, \_\ ))$
par l'Hypothèse 1 et définition de $\mathcal{W}(C, \Sigma)$
$\Rightarrow \ \exists(m' \geq n + 1).\ proj_b(\alpha_{\rightarrow n+1}) = proj_b(\tilde{\alpha}_{\rightarrow m'})$
par définition de $proj_b$
$\alpha_{\rightarrow n+1} = \alpha_1 : \ldots : \alpha_n : \alpha_{n+1} \ \wedge \ \alpha_{n+1} = (i_{n+1}, \_)$
$\Sigma^{\psi}(i_{n+1} : \_\ , \_\ ) \neq nil$
$\Rightarrow \ \exists(m' > m + 1).\ \tilde{\alpha}_{m'} = (i_{n+1}, \_\ )$
$\wedge \forall(m < m'' < m').\ \tilde{\alpha}_{m''} = (i_a, \_\ ))$
par l'Hypothèse 1 et définition de $\mathcal{W}(C, \Sigma)$
$\Rightarrow \ \exists(m' > n + 1).\ proj_b(\alpha_{\rightarrow n+1}) = proj_b(\tilde{\alpha}_{\rightarrow m'})$
par définition de $proj_b$ *et* $\mathcal{W}(C, \Sigma)\square$

LEMME **41.**

$$\begin{array}{l} \forall(a \in Asp^o).\forall(C, \Sigma). \\ \Sigma^{\psi} = [\![a]\!] \ \Rightarrow \ proj_b(\alpha) = proj_b(\tilde{\alpha}) \\ \textit{avec} \ \ \alpha = \mathcal{B}(C, \Sigma^b) \ \ \textit{et} \ \ \tilde{\alpha} = \mathcal{W}(C, \Sigma) \end{array}$$

Preuve : En utilisant le Lemme 40 et la relation de coninduction [GH99] ci-dessous

$$\begin{array}{l} proj_b(\alpha) = proj_b(\tilde{\alpha}) \\ \Leftrightarrow \ \forall(k \geq 1).\ \ approx\ k\ proj_b(\alpha) \ = \ approx\ k\ proj_b(\tilde{\alpha}) \end{array}$$

où $approx\ k\ \alpha$ est une fonction qui retourne les k-premiers éléments de la séquence α. $\square$

Annexe C

Aspects pour Java

Dans cette annexe, nous définissons la sémantique d'un noyau de Java et d'AspectJ. Premièrement, nous donnons une sémantique d'un sous ensemble de Java, essentiellement featherweight Java avec des affectations (AFJ). Ensuite, nous définissons la sémantique d'un sous ensemble d'AspectJ basé sur le langage AFJ et constitué des aspects `around`, du point de coupure `cflow` et des associations (`pertarget` et `percflow`).

C.1 Featherweight Java avec des affectations

Featherweight Java avec des affectations (AFJ) [MP05] c'est featherweight Java [IPW99], un sous ensemble fonctionnel pure de Java étendu avec les modifications des champs. La syntaxe de featherweight Java est étendue avec un nouveau type d'expression $e.f = e$ représentant les affectations des champs. Comparé à AFJ, notre version ne possède pas la catégorie syntaxique des constructeurs tout comme l'instruction `return` qui ne sont pas utiles pour notre objectif.

$$
\begin{array}{lll}
Prog & ::= & \overline{L};\ e \\
L & ::= & \texttt{class}\quad Y\ \texttt{extends}\quad Y\ \{\overline{Y}\,\overline{f}\ \overline{M}\} \\
M & ::= & Ym(\overline{Y}\,\overline{y})\{e\} \\
e & ::= & v \mid y \mid e.f \mid e.m(\overline{e}) \mid \texttt{new}\,Y(\overline{e}) \mid (Y)e \mid e.f = e
\end{array}
$$

Un programme est une séquence de classes ($\overline{L}$) suivie par une expression principale (e). Y représente l'identificateur d'une classe. Chaque classe a une séquence de champs associée à un type qui est l'identificateur d'une classe ($\overline{Y}\,\overline{f}$). Une classe définie aussi des méthodes ($\overline{M}$). Une méthode prend une séquence d'objets comme paramètres ($\overline{Y}\,\overline{y}$) et retourne un objet qui est le résultat du calcul du corps de la méthode (e). Une expression e peut être un objet (v), une variable (y), un accès à un champs ($e.f$), un appel à une méthode avec une séquence

d'expressions comme paramètres ($e.m(\overline{e})$), une construction d'une classe avec une séquence d'expressions comme paramètres ($\texttt{new}\ Y(\overline{e})$), une conversion automatique de type ($(Y)e$) ou une affectation ($e.f = e$). L'évaluation d'une affectation $e_1.f = e_2$ affecte au champs f de l'objet obtenu après l'évaluation de e_1, l'objet obtenu par l'évaluation de e_2. En plus de l'effet de bord, le résultat de l'affectation $e_1.f = e_2$ est celui de e_2.

Les expressions n'incluent pas la séquence. Un programme Java avec une séquence de commandes peut être transformé en un programme AFJ avec des appels de méthodes dont les arguments sont des commandes. Par exemple, le programme Java suivant

```
class Moo {
  Object o;

  Moo(Object o) {
    this.o = o;
  }
  Moo foo(Moo m) {
    this.o = m;
    return this.bar().bar();
  }
  Moo bar() {
    return (Moo)(this.o);
  }
  public static void main(String[] args) {
    new Moo(new Object()).foo(
     new Moo(new Moo(new Object()))).bar();
  }
}
```

peut être transformé dans le programme AFJ suivant

```
class Moo extends Object {
  Object o;

  Moo foo(Moo m) {
    this.foo1(this.o=m,m)
  }
  Moo foo1(Object o1, Moo m) {
     this.bar().bar()
  }
  Moo bar() {
     (Moo)(this.o)
  }
```

```
}
new Moo(new Object()).foo(new Moo(new Moo(new Object()))).bar()
```

Dans le but de prendre en compte les affectations, la sémantique de AFJ met à jour une mémoire. Elle utilise des règles de congruence [MP05] pour décrire l'exécution d'un programme. Nous modifions cette sémantique et remplaçons les règles de congruence par des règles qui séquentialisent l'exécution en utilisant l'environnement de continuation C. Notre sémantique de AFJ se définit alors comme suit. Premièrement, nous définissons certaines fonctions auxiliaires.

$$\begin{array}{lcl}
\Sigma^O & : & Object \rightarrow Y \times Fd \\
Fd & : & Identifier \rightarrow Object \\
mbody & : & Identifier \times Y \rightarrow e \\
FieldName & : & Y \rightarrow \overline{Identifier \times Identifier} \\
init & : & \overline{L} \times mbody \times FieldName \rightarrow mbody \times FieldName
\end{array}$$

$$\begin{array}{lcl}
init(\texttt{class}\ \ Y\ \texttt{extends}\ \ B\ \{\overline{T}\,\overline{f}\ \overline{M}\}\ \overline{L'},\ mbody,\ FieldName) & & \\
\qquad = \ init(\overline{L'},\ mbody[(m_0,\ Y) \mapsto e_0,\ \ldots,\ (m_n,\ Y) \mapsto e_n], & & \\
\qquad\quad FieldName[Y \mapsto \overline{(T,\ f)} \cup FieldName(B)]) & & \\
init(\epsilon,\ mbody,\ FieldName) & = & (mbody,\ FieldName)
\end{array}$$

La fonction Σ^O représente l'environnement de sauvegarde (*c.-à-d.,* le tas ou la mémoire). Il prend un $Object$ (*c.-à-d.,* une référence) comme paramètre et retourne une instance et un type. La fonction Fd prend l'identificateur d'un champ comme paramètre et retourne sa valeur courante (*c.-à-d.,* une référence). La fonction $mbody$ prend une signature d'une méthode et retourne la liste de ses paramètres et son corps. La fonction $FieldName$, retourne pour une classe donnée, la liste des champs et leurs types. La fonction $init$ construit, pour un programme donné, les environnements initiaux $mbody$ et $FieldName$. Un appel initial est de la forme $init(\overline{L}, \bot, \bot)$ avec $\overline{L}$ la liste des classes du programme. La sémantique d'un programme est donnée par un système de transition et résumée par l'équation suivante :

$$\begin{array}{c}
[\![\overline{L};\ e]\!] = \ (e : \bullet, \epsilon \cup \bot \cup \epsilon) \rightarrow_b^* (\epsilon : \bullet, v : \epsilon \cup \Sigma^o \cup \epsilon) \\
\text{avec } init(\overline{L},\ \bot,\ \bot) = \ (mbody,\ FieldName)
\end{array}$$

Le système de transition est définie par les règles d'inférences suivantes :

$$CAST1\ \ \frac{}{((Y)e : C, \Sigma^S \cup \Sigma^O \cup \Sigma^F)\ \rightarrow_b\ (e : CAST_Y : C, \Sigma^S \cup \Sigma^O \cup \Sigma^F)}$$

$$CAST2\ \ \frac{\Sigma^O(v) = (Y, Fd)\ \ Y <: D}{(CAST_D : C, v : \Sigma^S \cup \Sigma^O \cup \Sigma^F)\ \rightarrow_b\ (C, v : \Sigma^S \cup \Sigma^O \cup \Sigma^F)}$$

$$GET1\ \ \frac{}{(e.f_i : C, \Sigma^S \cup \Sigma^O \cup \Sigma^F)\ \rightarrow_b\ (e : get\ f_i : C, \Sigma^S \cup \Sigma^O \cup \Sigma^F)}$$

$$GET2\ \frac{\Sigma^O(v) = (Y, Fd)\ \ Fd(f_i) = v_2}{(get\ f_i : C, v : \Sigma^S \cup \Sigma^O \cup \Sigma^F)\ \rightarrow_b\ (C, v_2 : \Sigma^S \cup \Sigma^O \cup \Sigma^F)}$$

$$SET1\ \frac{}{(e_0.f_i = e : C, \Sigma^S \cup \Sigma^O \cup \Sigma^F)\ \rightarrow_b\ (e : e_0 : set\ f_i : C, \Sigma^S \cup \Sigma^O \cup \Sigma^F)}$$

$$SET2\ \frac{\Sigma^O(v_0) = (Y,\ Fd)}{\begin{array}{c}(set\ f_i : C, v_0 : v : \Sigma^S \cup \Sigma^O \cup \Sigma^F) \\ \rightarrow_b\ (C, v : \Sigma^S \cup \Sigma^O[v_0 \mapsto (Y,\ Fd[f_i \mapsto v])] \cup \Sigma^F)\end{array}}$$

$$CALL1\ \frac{}{\begin{array}{c}(e_0.m(e_1,\ \ldots,\ e_n) : C, \Sigma^S \cup \Sigma^O \cup \Sigma^F) \\ \rightarrow_b\ (e_1 : \ldots : e_n : e_0 : call\ m^n : C, \Sigma^S \cup \Sigma^O \cup \Sigma^F)\end{array}}$$

$$CALL2\ \frac{\Sigma^O(v_0) = (Y, Fd)\ \ mbody(m, Y) = (y_1, \ldots, y_n).e}{\begin{array}{c}(call\ m^n : C, v_0 : v_1 : \ldots : v_n : \Sigma^S \cup \Sigma^O \cup \Sigma^F) \\ \rightarrow_b\ (e[y_1/v_n,\ \ldots,\ y_n/v_1,\ this/v_0] : \{C\}, \Sigma^S \cup \Sigma^O \cup ((Y)m, v_0) : \Sigma^F)\end{array}}$$

$$NEW1\ \frac{}{\begin{array}{c}(\texttt{new}\ Y(e_1,\ \ldots,\ e_n) : C, \Sigma^S \cup \Sigma^O \cup \Sigma^F) \\ \rightarrow_b\ (e_1 : \ldots : e_n : New_Y^n : C, \Sigma^S \cup \Sigma^O \cup \Sigma^F)\end{array}}$$

$$NEW2\ \frac{\begin{array}{c}v \notin dom(\Sigma^O)\ \ FieldName(Y) = (T_1,\ f_1), \ldots, (T_n,\ f_n) \\ Fd = [f_1,\ \ldots,\ f_n \mapsto v_1,\ \ldots,\ v_n]\end{array}}{\begin{array}{c}(New_Y^n : C, v_1 : \ldots : v_n : \Sigma^S \cup \Sigma^O \cup \Sigma^F) \\ \rightarrow_b\ (v : C, \Sigma^S \cup \Sigma^O[v \mapsto (Y,\ Fd)] \cup \Sigma^F)\end{array}}$$

$$RET\ \frac{}{(\{C\}, \Sigma^S \cup \Sigma^O \cup ((Y)m, v_0) : \Sigma^F) \qquad \rightarrow_b\ (C, \Sigma^S \cup \Sigma^O \cup \Sigma^F)}$$

$$PUSHOBJ\ \frac{}{(v : C, \Sigma^S \cup \Sigma^O \cup \Sigma^F)\ \rightarrow_b\ (C, v : \Sigma^S \cup \Sigma^O \cup \Sigma^F)}$$

Dans toutes les règles, Σ^S est le sous ensemble de Σ qui représente la pile d'évaluation qui contient le résultat de l'évaluation d'une expression. La pile Σ^F contient les signatures des

appels de méthode qui sont en train de s'exécuter (comme dans les Règles CALL et RET de la page 96). Ici, les éléments de Σ^F ne sont pas seulement des signatures mais des paires $((t)id, r)$ qui contiennent aussi un receveur r qui sera utilisé pour lier la variable du point de coupure `target`. La majorité des règles de sémantique est présentée comme une paire. Les règles indexées par 1 transforme l'expression courante en une séquence d'expressions dans la continuation. Les règles indexées par 2 calculent l'expression courante. Par exemple,

- la Règle CAST1 construit une continuation qui premièrement évalue l'expression e, et ensuite effectue la conversion automatique de type. La Règle CAST2 accompli effectivement la conversion automatique de type en vérifiant les contraintes de sous type sur le type dynamique de la valeur v.
- Les Règles GET évaluent le receveur et accèdent à un de ses champs.
- Les Règles SET évaluent la partie droite, la partie gauche et exécutent l'affectation.
- Les Règles CALL évaluent les arguments d'une méthode de la gauche vers la droite, et ensuite le receveur. L'appel de la méthode lui-même place au sommet de Σ^F la signature et le receveur, substitue les paramètres par leurs valeurs et évalue son corps.
- Les Règles NEW évaluent les arguments d'un constructeur de la gauche vers la droite et construisent une instance en utilisant une référence fraîche.
- La Règle RET qui représente l'instruction de retour (la fin d'un bloc) retire une paire du sommet de Σ^F. La Règle PUSHOBJ retire la référence qui est au sommet de la continuation pour la placer au sommet de la pile des valeurs Σ^S.

EXEMPLE **42.** *Dans cette exemple, nous considérons le programme AFJ précédent (page 132) que nous appelons $Prog$. Pour être claire, une pile de plus d'un élément $x_1, \ldots, x_n$ est notée $x_1 : \ldots : x_n$, une pile d'un élément x est notée $x : \epsilon$ ou $x : \bullet$ et la pile vide est notée ϵ ou $\epsilon : \bullet$.*

L'exécution de $Prog$ en accord avec nos règles de sémantique, commence par l'exécution de la fonction $init$ qui initialise les environnements du programme :

$$init(\texttt{class}\ Moo\ \texttt{extends}\quad Object\ \{\ldots\}\epsilon,\ \bot,\ \bot)$$
$$= (\epsilon,\ mbody[(foo, Moo) \mapsto this.foo1(this.o = m, m), (foo1, Moo) \mapsto this.bar().bar(),$$
$$(bar, Moo) \mapsto (Moo)(this.o)],\ FieldName[Moo \mapsto (Object, o)])$$

$$= (mbody[(foo, Moo) \mapsto this.foo1(this.o = m, m), (foo1, Moo) \mapsto this.bar().bar(),$$
$$(bar, Moo) \mapsto (Moo)(this.o)],\ FieldName[Moo \mapsto (Object, o)])$$

$[\![Prog]\!]$
$= (start : new\ Moo(new\ Object()).foo(new\ Moo(new\ Moo(new\ Object()))).call\ bar(),$
$\quad \epsilon \cup \bot \cup \epsilon)$
$\rightarrow_b (new\ Moo(new\ Object()).foo(new\ Moo(new\ Moo(new\ Object()))).call\ bar() : \bullet,$
$\quad \epsilon \cup \bot \cup \epsilon)$
$\rightarrow_b (new\ Moo(new\ Object()).foo(new\ Moo(new\ Moo(new\ Object())))) : call\ bar^0,$
$\quad \epsilon \cup \bot \cup \epsilon)$
$\rightarrow_b (new\ Moo(new\ Moo(new\ Object()))) : new\ Moo(new\ Object()) : call\ foo^1 : call\ bar^0,$
$\quad \epsilon \cup \bot \cup \epsilon)$
$\rightarrow_b (new\ Moo(new\ Object()) : New^1_{Moo} : new\ Moo(new\ Object()) : call\ foo^1 : call\ bar^0,$
$\quad \epsilon \cup \bot \cup \epsilon)$
$\rightarrow_b (new\ Object() : New^1_{Moo} : New^1_{Moo} : new\ Moo(new\ Object()) : call\ foo^1 : call\ bar^0,$
$\quad \epsilon \cup \bot \cup \epsilon)$
$\rightarrow_b (New^0_{Object} : New^1_{Moo} : New^1_{Moo} : new\ Moo(new\ Object()) : call\ foo^1 : call\ bar^0,$
$\quad \epsilon \cup \bot \cup \epsilon)$

$\rightarrow_b (v_0 : New^1_{Moo} : New^1_{Moo} : new\ Moo(new\ Object()) : call\ foo^1 : call\ bar^0, \epsilon \cup \Sigma^O \cup \epsilon)$
$avec\ \Sigma^O = [v_0 \mapsto (Object, \bot)]$
$\rightarrow_b (New^1_{Moo} : New^1_{Moo} : new\ Moo(new\ Object()) : call\ foo^1 : call\ bar^0,$
$\quad v_0 : \epsilon \cup \Sigma^O \cup \epsilon)$
$\rightarrow_b (v_1 : New^1_{Moo} : new\ Moo(new\ Object()) : call\ foo^1 : call\ bar^0,$
$\quad \epsilon \cup \Sigma^O[v_1 \mapsto (Moo,\ o \mapsto v_0)] \cup \epsilon)$
$\rightarrow_b (New^1_{Moo} : new\ Moo(new\ Object()) : call\ foo^1 : call\ bar^0,$
$\quad v_1 : \epsilon \cup \Sigma^O[v_1 \mapsto (Moo,\ o \mapsto v_0)] \cup \epsilon)$
$\rightarrow_b (v_2 : new\ Moo(new\ Object()) : call\ foo^1 : call\ bar^0, \epsilon \cup \Sigma^O[v_2 \mapsto (Moo,\ o \mapsto v_1)] \cup \epsilon)$

$\rightarrow_b (new\ Moo(new\ Object()) : call\ foo^1 : call\ bar^0, v_2 : \epsilon \cup \Sigma^O[v_2 \mapsto (Moo,\ o \mapsto v_1)] \cup \epsilon)$
$\rightarrow_b (new\ Object() : New^1_{Moo} : call\ foo^1 : call\ bar^0, v_2 : \epsilon \cup \Sigma^O[v_2 \mapsto (Moo,\ o \mapsto v_1)] \cup \epsilon)$
$\rightarrow_b (New^0_{Object} : New^1_{Moo} : call\ foo^1 : call\ bar^0, v_2 : \epsilon \cup \Sigma^O[v_2 \mapsto (Moo,\ o \mapsto v_1)] \cup \epsilon)$
$\rightarrow_b (v_3 : New^1_{Moo} : call\ foo^1 : call\ bar^0, v_2 : \epsilon \cup \Sigma^O[v_3 \mapsto (Object,\ \bot)] \cup \epsilon)$
$\rightarrow_b (New^1_{Moo} : call\ foo^1 : call\ bar^0, v_3 : v_2 \cup \Sigma^O[v_3 \mapsto (Object,\ \bot] \cup \epsilon)$
$\rightarrow_b (v_4 : call\ foo1 : call\ bar^0, v_2 : \epsilon \cup \Sigma^O[v_4 \mapsto (Moo,\ o \mapsto v_3] \cup \epsilon)$

$\rightarrow_b (call\ foo^1 : call\ bar^0, v_4 : v_2 \cup \Sigma^O \cup \epsilon)$
$avec\ \Sigma^O = [v_0 \mapsto (Object, \bot), v_1 \mapsto (Moo, o \mapsto v_0), v_2 \mapsto (Moo, o \mapsto v_1),$
$v_3 \mapsto (Object, \bot), v_4 \mapsto (Moo, o \mapsto v_3)]$
$\rightarrow_b (v_4.foo1(v_4.o\ =\ v_2,\ v_2) : \{call\ bar^0\}, \epsilon \cup \Sigma^O \cup ((Moo)foo, v_4) : \epsilon)$
$\rightarrow_b (v_4.o\ =\ v_2 : v_2 : v_4 : call\ foo1^2 : \{call\ bar^0\}, \epsilon \cup \Sigma^O \cup ((Moo)foo, v_4) : \epsilon)$
$\rightarrow_b (v_2 : v_4 : set\ o : v_2 : v_4 : call\ foo1^2 : \{call\ bar^0\}, \epsilon \cup \Sigma^O \cup ((Moo)foo, v_4) : \epsilon)$
$\rightarrow_b (v_4 : set\ o : v_2 : v_4 : call\ foo1^2 : \{call\ bar^0\}, v_2 : \epsilon \cup \Sigma^O \cup ((Moo)foo, v_4) : \epsilon)$
$\rightarrow_b (set\ o : v_2 : v_4 : call\ foo1^2 : \{call\ bar^0\}, v_4 : v_2 \cup \Sigma^O \cup ((Moo)foo, v_4) : \epsilon)$
$\rightarrow_b (v_2 : v_4 : call\ foo1^2 : \{call\ bar^0\}, v_2 : \epsilon \cup \Sigma^O[v_4 \mapsto (Moo,\ o \mapsto v_2] \cup ((Moo)foo, v_4) : \epsilon)$
$\rightarrow_b (v_4 : call\ foo1^2 : \{call\ bar^0\}, v_2 : v_2 \cup \Sigma^O[v_4 \mapsto (Moo,\ o \mapsto v_2] \cup ((Moo)foo, v_4) : \epsilon)$
$\rightarrow_b (call\ foo1^2 : \{call\ bar^0\}, v_4 : v_2 : v_2 \cup \Sigma^O[v_4 \mapsto (Moo,\ o \mapsto v_2] \cup ((Moo)foo, v_4) : \epsilon)$

$\rightarrow_b\ (v_4.bar().bar() : \{\{call\ bar^0\}\}, \epsilon \cup \Sigma^O \cup ((Moo)foo1, v_4) : \Sigma^F)$
$avec\ \Sigma^O = [v_0 \mapsto (Object, \bot), v_1 \mapsto (Moo, o \mapsto v_0), v_2 \mapsto (Moo, o \mapsto v_1),$
$v_3 \mapsto (Object, \bot), v_4 \mapsto (Moo, o \mapsto v_2)]$
$\Sigma^F = ((Moo)foo, v_4) : \epsilon$
$\rightarrow_b\ (v_4.bar() : call\ bar^0 : \{\{call\ bar^0\}\}, \epsilon \cup \Sigma^O \cup ((Moo)foo1, v_4) : \Sigma^F)$
$\rightarrow_b\ (v_4 : call\ bar^0 : call\ bar^0 : \{\{call\ bar^0\}\}, \epsilon \cup \Sigma^O \cup ((Moo)foo1, v_4) : \Sigma^F)$

$\rightarrow_b\ (call\ bar^0 : call\ bar^0 : \{\{call\ bar^0\}\}, v_4 : \epsilon \cup \Sigma^O \cup ((Moo)foo1, v_4) : \Sigma^F)$
$\rightarrow_b\ ((Moo)v_4.o : \{call\ bar^0 : \{\{call\ bar^0\}\}\}, \epsilon \cup \Sigma^O \cup ((Moo)bar, v_4) : \Sigma^F)$
$with\ \Sigma^F = ((Moo)foo1, v_4) : ((Moo)foo, v_4)$
$\rightarrow_b\ (v_4.o : CAST_{Moo} : \{call\ bar^0 : \{\{call\ bar^0\}\}\}, \epsilon \cup \Sigma^O \cup ((Moo)bar, v_4) : \Sigma^F)$
$\rightarrow_b\ (v_4 : get\ o : CAST_{Moo} : \{call\ bar^0 : \{\{call\ bar^0\}\}\}, \epsilon \cup \Sigma^O \cup ((Moo)bar, v_4) : \Sigma^F)$
$\rightarrow_b\ (get\ o : CAST_{Moo} : \{call\ bar^0 : \{\{call\ bar^0\}\}\}, v_4 : \epsilon \cup \Sigma^O \cup ((Moo)bar, v_4) : \Sigma^F)$
$\rightarrow_b\ (CAST_{Moo} : \{call\ bar^0 : \{\{call\ bar^0\}\}\}, v_2 : \epsilon \cup \Sigma^O \cup ((Moo)bar, v_4) : \Sigma^F)$
$avec\ \Sigma^O(v_4) = (\_,\ o \mapsto v_2)$

$\rightarrow_b\ (\{call\ bar^0 : \{\{call\ bar^0 :\}\}\}, v_2 : \epsilon \cup \Sigma^O \cup ((Moo)bar, v_4) : \Sigma^F)$
$avec\ \Sigma^O(v_2) = (Moo,\ \_) \wedge Moo <: Moo$
$\rightarrow_b\ (call\ bar^0 : \{\{call\ bar^0\}\}, v_2 : \epsilon \cup \Sigma^O \cup \Sigma^F)$
$\rightarrow_b\ ((Moo)v_2.o : \{\{\{call\ bar^0\}\}\}, \epsilon \cup \Sigma^O \cup ((Moo)bar, v_2) : \Sigma^F)$
$\rightarrow_b\ (v_2.o : CAST_{Moo} : \{\{\{call\ bar^0\}\}\}, \epsilon \cup \Sigma^O \cup ((Moo)bar, v_2) : \Sigma^F)$
$\rightarrow_b\ (v_2 : get\ o : CAST_{Moo} : \{\{\{call\ bar^0\}\}\}, \epsilon \cup \Sigma^O \cup ((Moo)bar, v_2) : \Sigma^F)$
$\rightarrow_b\ (get\ o : CAST_{Moo} : \{\{\{call\ bar^0\}\}\}, v_2 : \epsilon \cup \Sigma^O \cup ((Moo)bar, v_2) : \Sigma^F)$
$\rightarrow_b\ (CAST_{Moo} : \{\{\{call\ bar^0\}\}\}, v_1 : \epsilon \cup \Sigma^O \cup ((Moo)bar, v_2) : \Sigma^F)$
$avec\ \Sigma^O(v_2) = (\_,\ o \mapsto v_1)$

$\rightarrow_b\ (\{\{\{call\ bar^0 : \bullet\}\}\}, v_1 : \epsilon \cup \Sigma^O \cup ((Moo)bar, v_2) : \Sigma^F)$
$avec\ \Sigma^O(v_1) = (Moo,\ \_)\ \wedge\ Moo <: Moo$
$\rightarrow_b\ (\{\{call\ bar^0 : \bullet\}\}, v_1 : \epsilon \cup \Sigma^O \cup \Sigma^F)$
$with\ \Sigma^F = ((Moo)foo1, v_4) : ((Moo)foo, v_4)$
$\rightarrow_b\ (\{call\ bar^0 : \bullet\}, v_1 : \epsilon \cup \Sigma^O \cup ((Moo)foo, v_4) : \epsilon)$
$\rightarrow_b\ (call\ bar^0 : \bullet, v_1 : \epsilon \cup \Sigma^O \cup \epsilon)$
$\rightarrow_b\ ((Moo)v_1.o : \{\epsilon : \bullet\}, \epsilon \cup \Sigma^O \cup ((Moo)bar, v_1) : \epsilon)$
$\rightarrow_b\ (v_1.o : CAST_{Moo} : \{\epsilon : \bullet\}, \epsilon \cup \Sigma^O \cup ((Moo)bar, v_1) : \epsilon)$
$\rightarrow_b\ (v_1 : get\ o : CAST_{Moo} : \{\epsilon : \bullet\}, \epsilon \cup \Sigma^O \cup ((Moo)bar, v_1) : \epsilon)$
$\rightarrow_b\ (get\ o : CAST_{Moo} : \{\epsilon : \bullet\}, v_1 : \epsilon \cup \Sigma^O \cup ((Moo)bar, v_1) : \epsilon)$
$\rightarrow_b\ (CAST_{Moo} : \{\epsilon : \bullet\}, v_0 : \epsilon \cup \Sigma^O \cup ((Moo)bar, v_1) : \epsilon)\ with\ \Sigma^O(v_1) = (\_,\ o \mapsto v_0)$
$\rightarrow_b\ (\{\epsilon : \bullet\}, v_0 : \epsilon \cup \Sigma^O \cup ((Moo)bar, v_1) : \epsilon)\ with\ \Sigma^O(v_0) = (Object,\ \_) \wedge Object <: Moo$
$\rightarrow_b\ (\epsilon : \bullet,\ v_0 : \epsilon \cup \Sigma^O \cup \epsilon)$
$avec\ \Sigma^O = [v_0 \mapsto (Object, \bot), v_1 \mapsto (Moo, o \mapsto v_0), v_2 \mapsto (Moo, o \mapsto v_1),$
$v_3 \mapsto (Object, \bot), v_4 \mapsto (Moo, o \mapsto v_2)]$

C.2 Featherweight AspectJ

La grammaire ci-dessous représente un sous ensemble d'AspectJ. Dans cette grammaire, un aspect est représenté par un ensemble de champs, un point de coupure P, une action Ad et son identificateur id. L'action, de type `around`, prend une séquence d'objets comme paramètres et retourne un objet. Le corps d'une action est une expression AFJ e dans laquelle `proceed` peut être une sous expression. Le point de coupure P est un motif qui peut représenter soit

- un appel de méthode $(P_T)P_I$ où P_T ou P_I sont soit des identificateurs, soit $*$;
- la référence du receveur $\texttt{target}(y)$ où y est une variable qui est associée à celle-ci ;
- le point de coupure `cflow` ;
- une disjonction, conjonction ou négation de ces motifs.

Un aspect peut être optionnellement, soit un aspect `percflow` ou un aspect `pertarget`. Dans le cas de `percflow`, une instance de l'aspect est créée chaque fois que le programme entre dans le flot de contrôle de $(P_T)P_I$ et pour `pertarget`, une instance de l'aspect est créée chaque fois que $(P_T)P_I$ est accédée par un objet

$$
\begin{array}{lcl}
A & ::= & \texttt{aspect} \quad [\texttt{percflow}((P_T)P_I) \mid \texttt{pertarget}((P_T)P_I)] \ id\ \{\overline{Y}\,\overline{f}\ P\ Ad\} \mid \\
Ad & ::= & Y\ \texttt{around}(\overline{Y}\,\overline{y})\{e\} \\
P & ::= & (P_T)P_I \mid \texttt{target}(y) \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid \neg\, P \mid \texttt{cflow}(P) \\
P_T & ::= & id \mid * \\
P_I & ::= & id \mid *
\end{array}
$$

Comme présenté à la Section 5.2.4, page 103, les aspects et leurs instances sont représentés par l'environnement Σ^ψ. Il associe les identificateurs des aspects aux aspects élémentaires Σ^ψ_{id}. L'application de Σ^ψ à une configuration (C, Σ) est définie comme à la Section 5.2.4, page 103 avec des aspects élémentaires `around` qui utilisent la fonction $match_G$ pour filtrer un point de coupure P. La fonction $priority$ utilisée par cette définition est définie comme ci-dessous en supposant que nous avons les fonctions suivantes :

$$
\begin{array}{l}
Advice :\ id \rightarrow e \\
Pointcut :\ id \rightarrow P
\end{array}
$$

qui retournent respectivement pour chaque identificateur son action et son point de coupure. Avec l'état global

$$\Sigma = \Sigma^S \cup \Sigma^O \cup \Sigma^F \cup \Sigma^P \cup \Sigma^\psi$$

$$\text{où } \Sigma^S \cup \Sigma^O \cup \Sigma^F \subset \Sigma^b \cup \Sigma^a$$

$$\text{et } \Sigma^P \subset \Sigma^a$$

$$priority(\Sigma^\psi_{id1}, \ldots, \Sigma^\psi_{idn})(i : C, X \cup v : \Sigma^S \cup \Sigma^F \cup \Sigma^P) = (\sigma_{j_1}(\phi_{idj_1}) : pop_p\ j_n : C, X \cup v :$$

$$\Sigma^S \cup \Sigma^F \cup \sigma_{j_2}(\phi_{idj_2}) : \ldots : \sigma_{j_n}(\phi_{idj_n}) : \overline{i} : \Sigma^P)$$

$$\text{avec } 1 \leq j_k \leq n \ \wedge\ \forall(\sigma_{j_k}, \phi_{idj_k}).\exists 1 \leq k \leq n.\ match_G(Pointcut(idk), i, v, \Sigma^F) =$$

$$\sigma_{j_k} \wedge Advice(idk) = \phi_{idj_k}$$

Comme dans le cas de plusieurs aspects `around` présenté à la Section 2.4.1, page 49, chaque fonction Σ^{ψ}_{idk}, insère une action $\sigma_{j_k}(\phi_{idj_k})$ sur C ou Σ^P si le point de coupure de l'aspect dont l'identificateur est idk filtre l'instruction courante i. Rappelons que l'ordre d'exécution des actions est définie par la fonction $priority$ qui ordonne les aspects élémentaires de Σ^{ψ}. Chaque σ_{j_k} associe chaque variable d'un point de coupure à un objet. Ici, il associe la variable dans le point de coupure `target` à l'objet qui lui correspond, cet objet pourrait être le sommet de la pile d'évaluation ou un receveur de la pile du flot de contrôle. Parce que nous n'avons pas un point de coupure comme le point de coupure dynamique `if` d'AspectJ, chaque ϕ_{idj_k} ne prend pas Σ en paramètre comme à la Section 2.3, et retourne l'action de l'aspect idk. Donc $\sigma_{j_k}(\phi_{idj_k})$ retourne une action dans laquelle les variables du point de coupure utilisées dans le corps de l'action sont remplacées par leurs valeurs stockées dans Σ. σ_{j_k} est retournée par la fonction $match_G$ qui représente le processus de filtrage d'une instruction en prenant en compte la pile de flot de contrôle et le sommet de la pile d'évaluation qui vont permettre de lier les variables de motif à leurs valeurs. Cette fonction a la signature suivante

$$match_G :\ P \times Instruction \times \Sigma^F \times \Sigma^O \rightarrow \sigma \cup \{Fail\}$$

où, Σ^O est l'ensemble des objets, $Instruction$ est l'ensemble des instructions et Σ^F est la pile de flot de contrôle. $match_G$ retourne une substitution σ si l'instruction courante est filtrée et $Fail$ dans le cas contraire. $match_G$ est définie comme suit :

$$
\begin{array}{lcl}
match_G(target(y),\ i,\ \Sigma^F,\ v) & = & \{y \rightarrow v\} \\
\\
match_G((P_T)P_I,\ i,\ \Sigma^F,\ v) & = & if\ i = (t, id) \wedge \\
 & & (P_T = t \vee P_T = *) \wedge (P_I = id \vee P_I = *) \\
 & & then\ \emptyset \\
 & & else\ Fail \\
\\
match_G(P_1 \wedge P_2,\ i,\ \Sigma^F,\ v) & = & if\ match_G(P_1,\ i,\ \Sigma^F,\ v)\ =\ \sigma_1 \\
 & & \wedge\ match_G(P_2,\ i,\ \Sigma^F,\ v)\ =\ \sigma_2 \\
 & & then\ \sigma_1 \cup \sigma_2 \\
 & & else\ Fail \\
\\
match_G(\neg P,\ i,\ \Sigma^F,\ v) & = & if\ match_G(P,\ i,\ F,\ t)\ =\ \sigma\ then\ Fail \\
 & & else\ \emptyset \\
\\
match_G(P_1 \vee P_2,\ i,\ \Sigma^F,\ v) & = & if\ match_G(P_1,\ i,\ \Sigma^F,\ v)\ =\ \sigma\ then\ \sigma \\
 & & else\ \ match_G(P_2,\ i,\ \Sigma^F,\ v) \\
\\
match_G(\texttt{cflow}(P),\ i,\ (i^{'}, r) : \Sigma^{'F},\ v) & = & if\ match_G(P,\ i, \Sigma^F,\ v)\ =\ \sigma\ then\ \sigma \\
 & & else\ if\ \Sigma^F = (i^{'}, r) : \Sigma^{'F} \\
 & & then\ match_G(\texttt{cflow}(P),\ i^{'},\ \Sigma^{'F},\ r) \\
 & & else\ Fail
\end{array}
$$

Filtrer une instruction avec le point de coupure `target`(y), consiste à associer y avec l'objet passé à $match_G$ (*c.-à-d.,* le receveur lié à cette instruction). Filtrer une instruction avec $(P_T)P_I$

retourne une substitution $\emptyset$ en cas de succès (le point de coupure ne contient pas de variables) ou $Fail$ en cas d'échec. Comme expliqué à la Section 2.5, page 51, les variables de motifs sous la portée d'une négation peuvent conduire à plusieurs substitutions. Dans ce cas aucune variable de motif ne doit apparaître dans une action. Alors, si $\neg P$ filtre une instruction, nous retournons une substitution vide. Pour le point de coupure `cflow`(P), nous essayons premièrement de filtrer l'instruction courante avec P, si cela échoue alors nous essayons de filtrer l'appel de méthode et son receveur (r) au sommet de Σ^F.

C.2.1 Aspects `around`

Comme dans AspectJ, les aspects `around` de Featherweight AspectJ sont appliqués uniquement à des appels de méthode. A l'exécution de `proceed`, comme la pile Σ^S est partagée par l'évaluation des expressions, les valeurs au sommet de Σ^S pourraient ne pas correspondre aux arguments de l'appel de méthode courant. Pour éviter ce problème, nous modifions les règles des aspects `around` comme suit. Afin de simplifier la présentation, nous considérons uniquement le cas où un seul aspect filtre l'instruction courante.

$$\Sigma^{\psi}(call\ m^n : C, v_0 : v_1 : \ldots : v_n : \Sigma^S \cup \Sigma^O \cup \Sigma^F \cup \Sigma^P \cup \Sigma^{\psi})$$

$$= (\sigma(\phi) : pop_p : C, \Sigma^S \cup \Sigma'^O \cup \Sigma^F \cup (e[y_1/v_n,\ \ldots,\ y_n/v_1,\ this/v_0], ((Y)m, v_0)) : \Sigma^P \cup \Sigma'^{\psi})$$

$$avec\ \ \Sigma^O(v_0) = (Y, Fd)\ \ \wedge\ \ mbody(m, Y) = (y_1, \ldots, y_n).e$$

PROCEED
$$\frac{}{\begin{array}{c}(\texttt{proceed} : C, \Sigma^S \cup \Sigma^O \cup \Sigma^F \cup (e, ((Y)m, v_0)) : \Sigma^P \cup \Sigma^{\psi}) \\ \rightarrow (e : \{push_p\ (e, ((Y)m, v_0)) : C\}, \Sigma^S \cup \Sigma^O \cup ((Y)m, v_0) : \Sigma^F \cup \Sigma^P \cup \Sigma^{\psi})\end{array}}$$

POP
$$\frac{}{(pop_p : C, \Sigma^S \cup \Sigma^O \cup \Sigma^F \cup x : \Sigma^P \cup \Sigma^{\psi}) \rightarrow (C, \Sigma^S \cup \Sigma^O \cup \Sigma^F \cup \Sigma^P \cup \Sigma^{\psi})}$$

Dans ces règles, ce n'est pas l'instruction courante ($call\ m^n$) qui est placée au sommet de la pile proceed, mais une paire constituée par le corps de cet appel de méthode dans lequel ses arguments sont substitués par leurs valeurs, et sa signature. Cela permet de résoudre le problème des valeurs qui ne correspondent pas aux arguments de l'appel de méthode courant à l'exécution de `proceed`. Comme dans le cas des règles générales des aspects `around` (voir Section 2.3.3, page 46), la Règle PROCEED exécute l'expression au sommet de la pile Σ^P. Parce que cette expression est le corps de l'appel de méthode dont la signature est au sommet de Σ^P, PROCEED place aussi cette signature au sommet de Σ^F et insère un bloc représentant l'adresse de retour de cette méthode. Aussi, nous n'avons pas besoin de tagger $e[y_1/v_n,\ \ldots,\ y_n/v_1,\ this/v_0]$ car

elle pourrait être filtrée par un aspect. Une autre modification est le remplacement de $test\ \phi$ par ϕ car dans featherweight AspectJ, ϕ ne dépend pas de Σ. Notons que, comme présenté à la Section 5.2.4, page 103, et comme on verra plus loin, une instance d'aspect peut être créée (Σ'^{ψ}) et son état sauvegardé dans Σ'^{O}.

Considérons maintenant un aspect `around` d'AspectJ `A1` qui affecte au champ `o` la référence à laquelle il appartient avant de continuer avec l'exécution de la méthode `bar`.

```
aspect A1 {

  Moo around(Moo r): call(Moo Moo.bar()) && target(r) {
    r.o = r;
    proceed();
  }
}
```

Une manière d'exprimer cet aspect comme un aspect featherweight AspectJ est :

- ajouter la classe suivante au programme de base

```
class A1 {

  Moo advice(Moo r) {
    this.advice2(r.o = r,proceed());
  }
  Moo advice2(Object o, Moo e) {
    e;
  }
}
```

  les méthodes `advice` et `advice2` permettent de transformer les commandes de l'action en une expression similairement à la transformation d'un programme Java en un programme AFJ.
- l'aspect `A1` est ensuite transformé en un aspect featherweight AspectJ `A1'` comme suit

```
aspect A1' {

   ((Moo)bar)  ∧  target(r)
   Moo around(Moo r)  {
     (new A1()).advice(r)
   }
}
```

Dans cette transformation, nous créons un objet de `A1` pour accéder à la méthode `advice` de cette classe. Nous appliquons les règles des aspects `around` ci-dessus dans l'exemple suivant

EXEMPLE **43.** *Dans cet exemple, nous tissons l'aspect `A1'` et exécutons le programme tissé qui*

en résulte et qui est basé sur le programme de l'Exemple 42, page 135. Après la construction des environnements du programme de base et de l'aspect, nous avons :

$$
\begin{array}{lcl}
mbody & = & [(foo, Moo) \mapsto this.foo1(this.o = m, m), \\
 & & (foo1, Moo) \mapsto this.bar().bar(), \\
 & & (advice, A1) \mapsto this.advice2(r.o = r, proceed()), \\
 & & (bar, Moo) \mapsto (Moo)(this.o), (advice2, A1) \mapsto e] \\
FieldName & = & [Moo \mapsto (Object, o)] \\
\Sigma^{\psi} & = & \{A1' \rightarrow \Sigma^{\psi}_{A1'}\} \\
Pointcut & = & [A1' \mapsto ((Moo)bar) \wedge target(r)] \\
Advice & = & [A1' \mapsto (new\ A1()).advice(r)] \\
Field & = & \bot
\end{array}
$$

Dans le cas où l'aspect ne filtre pas une instruction, l'exécution du programme tissé est équivalente à celle du programme de base en appliquant la Règle REDUCE *avec* $\Sigma^{\psi}(C, \Sigma) = nil$. *Alors,*

$$
\begin{array}{l}
(start : new\ Moo(new\ Object()).foo(new\ Moo(new\ Moo(new\ Object()))).call\ bar(), \\
\qquad \epsilon \cup \bot \cup \epsilon \cup \epsilon \cup \Sigma^{\psi}) \\
\rightarrow\ (new\ Moo(new\ Object()).foo(new\ Moo(new\ Moo(new\ Object()))).call\ bar() : \bullet, \\
\qquad \epsilon \cup \bot \cup \epsilon \cup \epsilon \cup \Sigma^{\psi}) \\
\rightarrow^{*}\ (v_4 : call\ bar^0 : call\ bar^0 : \{\{call\ bar^0\}\}, \epsilon \cup \Sigma^O \cup \Sigma^F \cup \epsilon \cup \Sigma^{\psi}) \\
\rightarrow\ (\sigma(\phi_{A1'}) : pop_p\ 1 : call\ bar^0 : \{\{call\ bar^0\}\}, \epsilon \cup \Sigma^O \cup \Sigma^F \cup \Sigma^P \cup \Sigma^{\psi}) \\
wiht\ \Sigma^O = [v_0 \mapsto (Object, \bot), v_1 \mapsto (Moo, o \mapsto v_0), v_2 \mapsto (Moo, o \mapsto v_1), \\
v_3 \mapsto (Object, \bot), v_4 \mapsto (Moo, o \mapsto v_2)] \\
\qquad \Sigma^F = ((Moo)foo1, v_4) : ((Moo)foo, v_4) \\
\qquad \Sigma^{\psi} = \{A1' \rightarrow \Sigma^{\psi}_{A1'}\} \\
\qquad \Sigma^P = ((Moo)v_4.o, ((Moo)bar, v_4)) : \epsilon \\
\qquad \sigma = \{r \rightarrow v_4\}\ \wedge\ \phi_{A1'} = (newA1()).advice(r) \\
\rightarrow\ (v_4 : new\ A1() : call\ advice^1 : pop_p\ 1 : call\ bar^0 : \{\{call\ bar^0\}\}, \\
\qquad \epsilon \cup \Sigma^O \cup \Sigma^F \cup \Sigma^P \cup \Sigma^{\psi}) \\
\rightarrow\ (new\ A1() : call\ advice^1 : pop_p\ 1 : call\ bar^0 : \{\{call\ bar^0\}\}, \\
\qquad v_4 : \epsilon \cup \Sigma^O \cup \Sigma^F \cup \Sigma^P \cup \Sigma^{\psi}) \\
\rightarrow\ (New^0_{A1} : call\ advice^1 : pop_p\ 1 : call\ bar^0 : \{\{call\ bar^0\}\}, \\
\qquad v_4 : \epsilon \cup \Sigma^O \cup \Sigma^F \cup \Sigma^P \cup \Sigma^{\psi}) \\
\rightarrow\ (v_5 : call\ advice^1 : pop_p\ 1 : call\ bar^0 : \{\{call\ bar^0\}\}, \\
\qquad v_4 : \epsilon \cup \Sigma^O[v_5 \mapsto (A1, \bot)] \cup \Sigma^F \cup \Sigma^P \cup \Sigma^{\psi}) \\
\rightarrow\ (call\ advice^1 : pop_p\ 1 : call\ bar^0 : \{\{call\ bar^0\}\}, \\
\qquad v_5 : v_4 \cup \Sigma^O[v_5 \mapsto (A1, \bot)] \cup \Sigma^F \cup \Sigma^P \cup \Sigma^{\psi})
\end{array}
$$

$\rightarrow\ (v_5.advice2(v_4.o = v_4, proceed()) : \{pop_p\ 1 : call\ bar^0 : \{\{call\ bar^0\}\}\},$
$\quad \epsilon \cup \Sigma^O \cup \Sigma^F \cup \Sigma^P \cup \Sigma^\psi)$
$with\ \Sigma^F = ((Moo)advice1, v_5) : ((Moo)foo1, v_4) : ((Moo)foo, v_4)$
$\quad \Sigma^O = [v_0 \mapsto (Object, \bot), v_1 \mapsto (Moo, o \mapsto v_0), v_2 \mapsto (Moo, o \mapsto v_1),$
$v_3 \mapsto (Object, \bot), v_4 \mapsto (Moo, o \mapsto v_2), v_5 \mapsto (A1, \bot)]$
$\rightarrow\ (v_4.o = v_4 : proceed() : v_5 : call\ advice2^2 : \{pop_p\ 1 : call\ bar^0 : \{\{call\ bar^0\}\}\},$
$\quad \epsilon \cup \Sigma^O \cup \Sigma^F \cup \Sigma^P \cup \Sigma^\psi)$
$\rightarrow^*\ (proceed() : v_5 : call\ advice2^2 : \{pop_p\ 1 : call\ bar^0 : \{\{call\ bar^0\}\}\},$
$\quad v_4 : \epsilon \cup \Sigma^O \cup \Sigma^F \cup \Sigma^P \cup \Sigma^\psi)$
$with\ \Sigma^O = [v_0 \mapsto (Object, \bot), v_1 \mapsto (Moo, o \mapsto v_0), v_2 \mapsto (Moo, o \mapsto v_1),$
$v_3 \mapsto (Object, \bot), v_4 \mapsto (Moo, o \mapsto v_4), v_5 \mapsto (A1, \bot)]$

$\rightarrow\ ((Moo)v_4.o : \{push\ ((Moo)v_4.o, ((Moo)bar, v_4)) : v_5 : call\ advice2^2$
$: \{pop_p : call\ bar^0 : \{\{call\ bar^0\}\}\}\}, v_4 : \epsilon \cup \Sigma^O \cup \Sigma^F \cup \epsilon \cup \Sigma^\psi)$
$with\ \Sigma^F = ((Moo)bar, v_4) : ((Moo)advice1, v_5) : ((Moo)foo1, v_4) : ((Moo)foo, v_4)$
$\rightarrow^*\ (call\ advice2^2 : \{pop_p : call\ bar^0 : \{\{call\ bar^0\}\}\}, v_5 : v_4 : v_4 \cup \Sigma^O \cup \Sigma^F \cup \Sigma^P \cup \Sigma^\psi)$
$with\ \Sigma^P = ((Moo)v_4.o, ((Moo)bar, v_4)) : \epsilon$
$\quad \Sigma^F = ((Moo)advice1, v_5) : ((Moo)foo1, v_4) : ((Moo)foo, v_4)$
$\rightarrow\ (v_4 : \{\{pop_p : call\ bar^0 : \{\{call\ bar^0\}\}\}\}, \epsilon \cup \Sigma^O \cup \Sigma^F \cup \Sigma^P \cup \Sigma^\psi)$
$with\ \Sigma^F = ((Moo)advice2, v_5) : ((Moo)advice1, v_5) : ((Moo)foo1, v_4) : ((Moo)foo, v_4)$
$\rightarrow\ (\{\{pop_p : call\ bar^0 : \{\{call\ bar^0\}\}\}\}, v_4 : \epsilon \cup \Sigma^O \cup \Sigma^F \cup \Sigma^P \cup \Sigma^\psi)$
$\rightarrow^*\ (pop_p : call\ bar^0 : \{\{call\ bar^0\}, v_4 : \epsilon \cup \Sigma^O \cup \Sigma^F \cup \Sigma^P \cup \Sigma^\psi)$
$with\ \Sigma^F = ((Moo)foo1, v_4) : ((Moo)foo, v_4)$
$\rightarrow\ (call\ bar^0 : \{\{call\ bar^0\}\}, v_4 : \epsilon \cup \Sigma^O \cup \Sigma^F \cup \epsilon \cup \Sigma^\psi)$
$\rightarrow\ ((Moo)v_4.o : \{\{\{call\ bar^0\}\}\}, \epsilon \cup \Sigma^O \cup \Sigma^F \cup \epsilon \cup \Sigma^\psi)$
$with\ \Sigma^F = ((Moo)bar, v_4)((Moo)foo1, v_4) : ((Moo)foo, v_4)$
$\rightarrow^*\ (\{\{\{call\ bar^0\}\}\}, v_4 : \epsilon \cup \Sigma^O \cup \Sigma^F \cup \epsilon \cup \Sigma^\psi)$
$\rightarrow^*\ (call\ bar^0 : \bullet, v_4 : \epsilon \cup \Sigma^O \cup \epsilon \cup \epsilon \cup \Sigma^\psi)$
$\rightarrow^*\ (\epsilon : \bullet, v_4 : \epsilon \cup \Sigma^O \cup \epsilon \cup \epsilon \cup \Sigma^\psi)$

C.2.2 Point de coupure `cflow`

Ici, nous voulons illustrer le tissage d'un aspect avec le point de coupure `cflow`. Pour cela, nous considérons l'aspect `A2'` ci-dessous. Cet aspect filtre les appels de la méthode `bar` qui se trouve dans le flot de contrôle de `foo1`. Il affecte au champ `o` du receveur de `foo1` la valeur de ce receveur avant de continuer avec l'exécution de `bar` (voir la méthode `advice` de la classe `A1`).

```
aspect A2'{

  ((Moo)bar)  ∧  cflow((Moo)foo1  ∧ target(r))
  Moo around(Moo r){
   (new A1()).advice(r)
```

```
    }
  }
```

Le tissage de cet aspect et l'exécution du programme tissé est comme suit

EXEMPLE **44.** *Les environnements* $mbody$ *et* $FieldName$ *du programme de base sont identiques à ceux de l'Exemple 43. Les environnements des aspects sont :*

$$\begin{array}{lcl} \Sigma^{\psi} & = & \{A2' \rightarrow \Sigma^{\psi}_{A2'}\} \\ Pointcut & = & [A2' \mapsto ((Moo)bar) \wedge cflow((Moo)foo1 \wedge target(r))] \\ Advice & = & [A2' \mapsto (newA1()).advice(r)] \\ Field & = & \bot \end{array}$$

$alors$,
$(start : new\ Moo(new\ Object()).foo(new\ Moo(new\ Moo(new\ Object()))).call\ bar(),$
$\quad \epsilon \cup \bot \cup \epsilon \cup \epsilon \cup \Sigma^{\psi})$
$\rightarrow^*\ (v_4 : call\ bar^0 : call\ bar^0 : \{\{call\ bar^0\}\}, \epsilon \cup \Sigma^O \cup \Sigma^F \cup \epsilon \cup \Sigma^{\psi})$
$\rightarrow\ (\sigma(\phi_{A2'}) : pop_p\ 1 : call\ bar^0 : \{\{call\ bar^0\}\}, \epsilon \cup \Sigma^O \cup \Sigma^F \cup \Sigma^P \cup \Sigma^{\psi})$
$wiht\ \Sigma^O = [v_0 \mapsto (Object, \bot), v_1 \mapsto (Moo, o \mapsto v_0), v_2 \mapsto (Moo, o \mapsto v_1),$
$v_3 \mapsto (Object, \bot), v_4 \mapsto (Moo, o \mapsto v_2)]$
$\quad \Sigma^F = ((Moo)foo1, v_4) : ((Moo)foo, v_4)$
$\quad \Sigma^{\psi} = \{A2' \rightarrow \Sigma^{\psi}_{A2'}\}$
$\quad \Sigma^P = ((Moo)v_4.o, ((Moo)bar, v_4)) : \epsilon$
$\quad \sigma = \{r \rightarrow v_4\}\ \wedge\ \phi_{A2'} = (newA1()).advice(r)$
$\rightarrow^*\ (call\ bar^0 : \{\{call\ bar^0\}\}, v_4 : \epsilon \cup \Sigma^O \cup \Sigma^F \cup \epsilon \cup \Sigma^{\psi})$
$with\ \Sigma^F = ((Moo)foo1, v_4) : ((Moo)foo, v_4)$
$\quad \Sigma^O = [v_0 \mapsto (Object, \bot), v_1 \mapsto (Moo, o \mapsto v_0), v_2 \mapsto (Moo, o \mapsto v_1),$
$v_3 \mapsto (Object, \bot), v_4 \mapsto (Moo, o \mapsto v_4), v_5 \mapsto (A1, \bot)]$

C.2.3 Association

Comme présenté à la Section 5.2.4, page 102, le traitement des associations est éffectué par la définition de la fonction *update*. Lorsqu'aucune instance ne doit être générée (l'aspect n'est pas déclaré `pertarget` ou `percflow`) *update* retourne à chaque instruction l'état Σ qu'il prend en paramètre, signifiant qu'aucune instance n'est créée (*update*$(i, \Sigma) = \Sigma$). Lorsque des aspects doivent générés dynamiquement des instances d'aspect (`pertarget` et `percflow`), ils modifient *update* de manière à ce qu'elle crée une instance à chaque fois qu'il le faut. Regardons un exemple.

```
aspect A3 pertarget((Moo)bar) {
   ((Moo)bar) ∧ cflow((Moo)foo1 ∧ target(r))
   Moo around(Moo r){
```

```
      (new A1()).advice(r)
    }

}
```

Cet aspect génère une instance de `A3`, à chaque fois qu'un objet accède à une méthode `bar` qui retourne un objet de type `Moo` (`pertarget((Moo)bar)`). La fonction *update* est donc modifiée comme suit :

$$\begin{array}{lcll} \textit{update}(\texttt{call bar}, \Sigma) & \triangleq & \Sigma' & \text{si } \Sigma^O(v) = (Moo, \_) \wedge \texttt{A3\_}v \notin \texttt{dom}(\Sigma^\psi) \\ & & & \text{avec } G_3(v, \Sigma) = (\Sigma^\psi_{\texttt{A3\_}v}, \Sigma') \\ & & & \text{et } \Sigma'^\psi = \Sigma^\psi\{\texttt{A3\_}v \rightarrow \Sigma^\psi_{\texttt{A3\_}v}\} \\ & & & \text{et } \Sigma = X \cup \Sigma^\psi \cup v : \Sigma^S \cup \Sigma^O \\ & & & \text{et } \Sigma' = X \cup \Sigma'^\psi \cup v : \Sigma^S \cup \Sigma'^O \\ & & & \text{et } \Sigma'^O = \Sigma^O[a_v \mapsto (A3, \bot)]] \end{array}$$

A chaque appel de la méthode `bar` dont le type de retour est `Moo`, *update* par l'intermédiaire de $G_3(v, \Sigma)$, génère l'instance d'aspect si elle n'existe pas déjà ($\texttt{A3\_}v \notin \texttt{dom}(\Sigma^\psi)$). Pour un objet donné v qui génère une instance d'aspect, et l'état courant Σ, G crée une nouvelle instance d'aspect, qui est une nouvelle fonction $\Sigma^\psi_{\texttt{A3\_}v}$ et un nouvel objet a_v est sauvegardé dans Σ'^O. De plus, l'état de cette instance est sauvegardé par l'intermédiaire d'un objet a_v de Σ'^O. Cet état peut être modifié pendant l'exécution de l'action. Dans cet exemple, cet état est vide ($\bot$).

Ce raisonnement s'applique aussi à des aspects `percflow` mais en modifiant la fonction *update* de la page 105.

Comparaison avec le cas d'une instance unique En résumé, par rapport au cas par défaut ou une instance d'aspect est créée une fois pour toute, les différences sont les suivantes :

- interpréter un aspect `pertarget` ou `percflow` ne modifie pas directement l'environnement global Σ, mais la fonction *update* comme présenté ci-dessus.
- La fonction $\Sigma^\psi_{\texttt{A3\_}v}$ associée à un aspect est similaire à la fonction par défaut Σ^ψ, excepté qu'elle fait reférence à une instance a_v qui dépend du receveur courant v.
- l'aspect n'est pas instancié uniquement à l'initialisation (par la fonction $init$) mais aussi dynamiquement par la fonction *update*.

Bibliographie

[AC96] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[ACH+05] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Optimising AspectJ. In Mary Hall, editor, *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 2005.

[AGMO06] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of caesarj. In *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer, 2006.

[ASB+08] Francisco Afonso, Carlos Silva, Nuno Brito, Sergio Montenegro, and Adriano Tavares. Aspect-oriented fault tolerance for real-time embedded systems. In *ACP4IS '08 : Proceedings of the 2008 AOSD workshop on Aspects, components, and patterns for infrastructure software*, pages 1–8, New York, NY, USA, 2008. ACM.

[BCD+06] Sara Bouchenak, Alan Cox, Steven Dropsho, Sumit Mittal, and Willy Zwaenepoel. Caching Dynamic Web Content : Designing and Analysing an Aspect-Oriented Solution. In *ACM/IFIP/USENIX 7th International Middleware Conference (Middleware-2006)*, Melbourne, Australia, November 2006.

[BdPK05] Sara Bouchenak, Noël de Palma, and Sacha Krakowiak. Tolérance aux fautes dans les grappes d'applications Internet. In *4ème Conférence Française sur les Systèmes d'Exploitation (CFSE 2005)*, Le Croisic, France, April 2005.

[BJJR04] Glenn Bruns, Radha Jagadeesan, Alan Jeffrey, and James Riely. μabc : A minimal aspect calculus. In *CONCUR 2004*, pages 209–224. Springer-Verlag, 2004.

[CC77] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CEP00] O. Grumberg Clarke E. and D. Peled. Model checking. *MIT Press*, 2000.

[CES83] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications : a practical approach.

In *POPL '83 : Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 117–126, New York, NY, USA, 1983. ACM Press.

[CF00] Thomas Colcombet and Pascal Fradet. Enforcing trace properties by program transformation. In *Symposium on Principles of Programming Languages (POPL'00)*, pages 54–66, 2000.

[CKFS01] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using aspectc to improve the modularity of path-specific customization in operating system code. *SIGSOFT Softw. Eng. Notes*, 26(5) :88–98, 2001.

[CL02] C. Clifton and G. Leavens. Observers and assistants : A proposal for modular aspect-oriented reasoning. In *FOAL Workshop*, 2002.

[CL06] Curtis Clifton and Gary T. Leavens. MiniMAO1 : An imperative core language for studying aspect-oriented reasoning. *Science of Computer Programming*, 63 :321–374, 2006.

[CLN07] Curtis Clifton, Gary T. Leavens, and James Noble. MAO : Ownership and effects for more effective reasoning about aspects. In *ECOOP*, volume 4609 of *LNCS*, pages 451–475, 2007.

[CLW03] Curtis Clifton, Gary T. Leavens, and Mitchell Wand. Parameterized aspect calculus : A core calculus for the direct study of aspect-oriented languages. Technical Report 03-13, Iowa State University, Department of Computer Science, October 2003. Submitted for publication.

[CVK06] Matt Chapman, Alexandre Vasseur, and Günter Kniesel, editors. *AOSD 2006 - Industry Track Proceedings, Bonn, Germany, March 20-24, 2006*, volume IAI-TR-2006-3. Computer Science Department III, University of Bonn, 2006.

[DDDF06] Simplice Djoko Djoko, Rémi Douence, and Pascal Fradet. CASB : Common aspect semantics base. Technical Report AOSD-Europe Deliverable D54, Inria, August 2006.

[DDDF07] Simplice Djoko Djoko, Rémi Douence, and Pascal Fradet. Proof of correctness of aspect transformations in the CASB. Technical Report AOSD-Europe Deliverable D88, Inria, July 2007.

[DDDF08a] Simplice Djoko Djoko, Rémi Douence, and Pascal Fradet. Aspects preserving properties. In *PEPM'08*, pages 135–145. ACM, 2008.

[DDDF08b] Simplice Djoko Djoko, Rémi Douence, and Pascal Fradet. Specialized aspect languages preserving classes of properties. In *SEFM'08*, pages 227–236. IEEE, 2008.

[DDDF08c] Simplice Djoko Djoko, Rémi Douence, and Pascal Fradet. A common aspect semantics base and some applications. Technical Report AOSD-Europe Deliverable D135, August 2008.

[DFL+05] Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean M. Menaud, Marc S. Devillechaise, and Mario Südholt. An expressive aspect language for system applications with arachne. In *AOSD '05 : Proceedings of the 4th international*

conference on Aspect-oriented software development, pages 27–38, New York, NY, USA, 2005. ACM.

[DFS02] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*, pages 173–188, October 2002. preprint version is ftp ://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-4435.pdf.

[DR06] Robert Dyer and Hridesh Rajan. Modular program transformations for aspect-oriented constructs. 2006.

[DW06] Daniel S. Dantas and David Walker. Harmless advice. *SIGPLAN Not.*, 41(1) :383–396, 2006.

[DWWW08] Daniel S. Dantas, David Walker, Geoffrey Washburn, and Stephanie Weirich. Aspectml : A polymorphic aspect-oriented functional programming language. *ACM Trans. Program. Lang. Syst.*, 30(3) :1–60, 2008.

[FETD08] Johan Fabry, Éric Tanter, and Theo DHondt. Kala : Kernel aspect language for advanced transactions. *Sci. Comput. Program.*, 71(3) :165–180, 2008.

[FHTH04] Pascal Fradet and Stéphane Hong Tuan Ha. Network fusion. In *Proc. of Asian Symposium on Programming Languages and Systems (APLAS'04)*, pages 21–40. Springer-Verlag, LNCS, Vol. 3302, November 2004.

[FHTH07] Pascal Fradet and Stéphane Hong Tuan Ha. Aspects of availability. In *GPCE'07*, pages 165–174. ACM, October 2007.

[FL06] Pascal Fradet and Ralf Lämmel. Special issue on foundations of aspect-oriented programming. *Sci. Comput. Program.*, 63(3) :203–206, 2006.

[GH99] Jeremy Gibbons and Graham Hutton. Proof Methods for Structured Corecursive Programs. In *Proceedings of the 1st Scottish Functional Programming Workshop*, August 1999.

[GK07] Max Goldman and Shmuel Katz. Maven : Modular aspect verification. In Orna Grumberg and Michael Huth, editors, *TACAS*, volume 4424 of *LNCS*, pages 308–322. Springer, 2007.

[IPW99] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight java : A minimal core calculus for java and gj. In *ACM Transactions on Programming Languages and Systems*, pages 132–146, 1999.

[JJR03] Radha Jagadeesan, Alan Jeffrey, and James Riely. A calculus of untyped aspect-oriented programs. In *In European Conference on Object-Oriented Programming*, pages 54–73. Springer-Verlag, 2003.

[Kat06] Shmuel Katz. Aspect categories and classes of temporal properties. *TAOSD*, 1, 2006.

[KFG04] Shriram Krishnamurthi, Kathi Fisler, and Michael Greenberg. Verifying aspect advice modularly. In *SIGSOFT '04/FSE-12*, pages 137–146. ACM Press, November 2004.

[KHH+01a] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with aspectj. *Commun. ACM*, 44(10) :59–65, 2001.

[KHH+01b] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP '01 : Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[KKM07] Claude Kirchner, Radu Kopetz, and Pierre-Etienne Moreau. Anti-pattern matching. In *ESOP*, pages 110–124, 2007.

[KL07] Sergei Kojarski and David H. Lorenz. Awesome : an aspect co-weaving system for composing multiple aspect-oriented extensions. In *OOPSLA*, pages 515–534, 2007.

[KLM+97] Gregor Kiczales, J. Lamping, A. Mendhekar, Chris Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, June 1997.

[LJ06] Bert Lagaisse and Wouter Joosen. True and transparent distributed composition of aspect-components. pages 42–61. 2006.

[LL00] Martin Lippert and Cristina Videira Lopes. A study on exception detection and handling using aspect-oriented programming. In *ICSE'00*, pages 418–427. ACM, 2000.

[LOO01] Karl Lieberherr, Doug Orleans, and Johan Ovlinger. Aspect-oriented programming with adaptive methods. *Commun. ACM*, 44(10) :39–41, 2001.

[MKL97] A. Mendhekar, G. Kiczales, and J. Lamping. RG : A case-study for aspect-oriented programming. Technical Report SPL97-009 P9710044, Palo Alto, CA, USA, February 1997.

[MP92] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag, 1992.

[MP05] Thomas Molhave and Lars H. Petersen. Assignment featherweight java : Bringing mutable state to featherweight java. 2005.

[NN92] F. Nielson and H. R. Nielson. *Semantics with Applications - A Formal Introduction*. John Wiley and Sons, 1992.

[OT01] Harold Ossher and Peri Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Commun. ACM*, 44(10) :43–50, 2001.

[PSD+04] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, Gérard Florin, Fabrice Legond-Aubry, and Laurent Martelli. Jac : an aspect-based distributed dynamic framework. *Softw. Pract. Exper.*, 34(12) :1119–1148, 2004.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[RSB04] Martin Rinard, Alexandru Salcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. In *SIGSOFT '04/FSE-12*, pages 147–158. ACM Press, 2004.

[SLS03] Macneil Shonle, Karl Lieberherr, and Ankit Shah. Xaspects : an extensible system for domain-specific aspect languages. In *OOPSLA '03 : Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 28–37, New York, NY, USA, 2003. ACM.

[SLU05] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban. AspectC++ : An AOP Extension for C++. *Software Developer's Journal*, (5) :68–76, 2005.

[TN05] Éric Tanter and Jacques Noyé. A versatile kernel for multi-language aop. In *GPCE*, pages 173–188, 2005.

[VL97] Cristina Videira Lopes. *D : A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, Boston, 1997.

[WKD04] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *TOPLAS*, 26(5) :890–910, September 2004.

[WZL03] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *ICFP '03 : Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 127–139, New York, NY, USA, 2003. ACM.