Prévention et détection des interférences inter-aspects : méthode et application à l'aspectisation de la tolérance aux fautes

Jimmy Lauret

THESE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par
Institut National Polytechnique de Toulouse (INP Toulouse)

Discipline ou spécialité :
Systèmes Informatiques Critiques

Présentée et soutenue par
Jimmy LAURET
Le *15 mai 2013*

Titre :
Prévention et détection des interférences inter-aspects :
méthode et application à l'aspectisation de la tolérance aux fautes

---

JURY

Rapporteurs :	*Laurence DUCHIEN*	*Professeur des Universités*
	François TAIANI	*Professeur des Universités*
Examinateurs:	*Emilie BALLAND*	*Chargé de recherche*
	François BELTRAND	*Ingénieur AIRBUS*
Directeurs de Thèse :	*Jean-Charles FABRE*	*Professeur des Universités*
	Hélène WAESELYNCK	*Chargé de recherche*

Ecole doctorale :
Systèmes (EDSYS)
Unité de recherche :
LAAS-CNRS
Directeur(s) de Thèse :
Jean-Charles FABRE et Hélène WAESELYNCK
Rapporteurs :
Laurence DUCHIEN et François TAIANI

Remerciements

Les travaux de thèse présentés dans ce mémoire ont été effectués au Laboratoire d'Analyse et d'Architecture des Systèmes du Centre National de la Recherche Scientifique (LAAS-CNRS).

Je remercie Madame Karama Kanoun, responsable du groupe de recherche Tolérance aux fautes et Sûreté de Fonctionnement informatique (TSF), pour m'avoir permis de réaliser mes travaux au sein de ce groupe.

J'exprime ma profonde reconnaissance à mes encadrants, dont j'ai pu apprécier les qualités tant scientifiques qu'humaines : Monsieur Jean-Charles FABRE, Directeur de recherche au LAAS-CNRS ; Madame Hélène WAESELYNCK, Chargée de recherche au LAAS-CNRS. Leur rigueur intellectuelle, leur très grande compétence, ont été déterminantes pour ces travaux.

Je remercie les personnes qui m'ont fait l'honneur de participer à mon jury de thèse :
- Laurence DUCHIEN Professeur des Universités à l'Université de Lille 1
- François TAIANI Professeur des Universités à l'Université de Rennes 1
- Emilie BALLAND Chargée de recherche INRIA au centre de Bordeaux - Sud Ouest
- François BELTRAND Ingénieur à AIRBUS
- Jean-Charles FABRE Professeur à l'Institut National Polytechnique de Toulouse
- Hélène WAESELYNCK Chargée de recherche, LAAS-CNRS, Toulouse

Madame Laurence DUCHIEN et Monsieur François TAIANI ont accepté de consacrer une partie de leur temps précieux pour juger mon travail en tant que rapporteurs. Je leur témoigne tout mon respect et ma reconnaissance.

Ces travaux ont été réalisés dans le cadre d'un work package du projet IMAP (Information Management for Avionics Platform) financé par Airbus. J'exprime mes remerciements aux responsables de ce work package, Jean-François DA ROCHA et François BELTRAND.

Ma thèse m'a aussi permis de goûter aux joies de l'enseignement. Je tiens à remercier en premier lieu Philippe ESTEBAN qui est un enseignant captivant et un collègue d'une efficacité redoutable. Un grand merci pour m'avoir épaulé lors de mes premiers pas en tant qu'enseignant et j'espère avoir fait et faire aussi bien. Merci aussi à Brahim HAMID et Nathalie HERNANDEZ avec qui j'ai eu la chance d'enseigner, d'écrire des sujets et de corriger des copies. Bien sûr, j'adresse aussi ma gratitude à toutes les personnes que j'ai côtoyées épisodiquement au fil de ces dernières années comme Sandy RHAMES, Ines MEGANEM, Bogdan ROBU, Damien FOURES.

Je remercie également Bernard CHERBONNEAU, Ileana OBER et l'ensemble du corps enseignant du département informatique de l'Université Paul Sabatier , qui m'ont fait confiance

et permis de faire de l'enseignement durant la préparation de cette thèse. Ces heures passées à préparer ou animer des cours m'ont permis de trouver un équilibre essentiel dans mon quotidien de doctorant.

Il m'aurait sans doute été impossible de mener cette thèse à son terme sans la présence des nombreuses personnes qui m'ont supporté, par sympathie ou par nécessité ; mes remerciements ou excuses donc à tous mes amis : l'ensemble des membres du groupe TSF, permanents, doctorants et stagiaires, avec lesquels j'ai partagé ces années de travail.
Je remercie également toutes les personnes qui permettent aux membres du laboratoire d'avoir un cadre de travail agréable, de la documentation au magasin en passant par la reprographie, sans oublier Sonia DE SOUSA, qui a su m'aider face aux problèmes administratifs et logistiques que j'ai rencontrés au cours de ces trois ans.

Ces remerciements seraient incomplets sans une pensée pour ma famille : mes parents, mes sœurs, mes beaux-parents. Merci à vous tous.

Finalement, merci à Océane. Tu as su me soutenir, m'encourager, me changer les idées et me remettre sur le droit chemin parfois. Ma thèse est finie et c'est bien grâce à toi.

Table des matières

Chapitre 1

Introduction

L'ÉVOLUTION des systèmes au cours de leur vie opérationnelle est incontournable. Les systèmes sûrs de fonctionnement, doivent évoluer afin de se conformer aux changements d'origines diverses. Par exemple de nouvelles exigences en matière de tolérance aux fautes, ou des changements dans leurs environnement ou dans la disponibilité de leur ressources. Ces évolutions ne doivent pas violer leurs propriétés de sûreté de fonctionnement, ce qui conduit à la notion d'informatique résiliente.

La capacité de faire facilement évoluer les systèmes pour faire face efficacement aux changements est donc une exigence de la plus haute importance. Ce problème est encore plus complexe dans le cas des systèmes critiques, qui ne peuvent être arrêtés pendant une longue période.

1.1 Problématique

Pour garantir la capacité de pouvoir reconfigurer un système logiciel et donc d'assurer la résilience à moindre coût, ce système doit posséder certaines qualités comme la modularité et la réutilisabilité de ses composants. Il est possible d'assurer ces qualités par construction, en utilisant des techniques qui permettent de modulariser le système logiciel, comme par exemple les approches orientées objet ou orientées composant. Ces paradigmes offrent un découpage des préoccupations fonctionnelles augmentant la modularité, et permettent une meilleure réutilisation des différents éléments composant les logiciels.

La mise en œuvre de mécanismes non-fonctionnels entraîne une dispersion du code des mécanismes dans le code fonctionnel. Par exemple, la journalisation, qui consiste à enregistrer les événements d'un programme. Typiquement, la journalisation nécessite de disperser à travers tout le code du programme des instructions ayant pour objectif d'enregistrer différents événements. C'est aussi le cas des mécanismes de tolérance aux fautes auxquels nous nous intéressons plus particulièrement dans cette thèse. La dispersion et l'entrelacement des différentes préoccupations rendent le code plus difficile à lire et à comprendre, augmentent le risque d'erreur, et rendent la maintenance et l'évolution du code des systèmes logiciels plus difficiles. La reconfiguration et donc la capacité de garantir la tolérance aux fautes sont difficilement réalisables dans ce contexte.

Ces deux principes fondamentaux ont guidé des recherches en génie logiciel pour faire face à ce problème : la séparation des préoccupations et la modularité. La création de logiciel selon

ces deux principes améliore sa compréhension, sa maintenabilité, sa réutilisabilité et sa configurabilité.

Au principe de modularité déjà adopté par la communauté du génie logiciel s'est ajouté le principe de séparation des préoccupations. Une solution élégante permettant d'obtenir la séparation des préoccupations pour la tolérance aux fautes consiste à utiliser une approche réflexive. Dans une approche réflexive pour la tolérance aux fautes, les architectures logicielles se composent de deux niveaux d'abstraction où le niveau de base fournit les fonctionnalités requises et le niveau supérieur contient le mécanisme de tolérance aux fautes.

De nombreux travaux ont montré l'efficacité de ces approches pour séparer le code fonctionnel et le code dédié à la tolérance aux fautes. Cela permet une réutilisation des différentes parties et une configurabilité accrue. Les concepts de ces approches ont évolué durant les deux dernières décennies et ont récemment abouti au paradigme de programmation orientée aspect (POA).

Proposé par KICZALES et al. [40], le paradigme de la programmation orientée aspect est de séparer le programme selon deux axes orthogonaux. Le premier, le programme de base, regroupe les fonctionnalités principales implémentées à l'aide de classes ou de composants, tandis que les préoccupations transversales, ne pouvant être implémentées de manière unitaire sur l'axe principal, sont encapsulées sur l'axe secondaire dans des aspects. Ces aspects se composent de bloc de code appelés greffons (advice) implémentant des actions non-fonctionnelles (trace, stockage, chiffrement, etc.)

Au moment de la compilation, les greffons contenus dans les aspects sont insérés à différents endroits du programme. Cette implémentation présente deux avantages. Le premier tient au fait que le code des préoccupations transversales est modularisé ce qui améliore la lisibilité, et facilite l'évolution ou la maintenance. Le deuxième avantage est que le code de chaque aspect n'est pas répété, ce qui diminue le risque d'erreur et facilite aussi la reconfiguration.

Les travaux s'appuyant sur la programmation orientée aspect pour l'intégration de la tolérance aux fautes montrent qu'elle permet d'obtenir la même couverture en terme de modèle de fautes, les mêmes performances avec une séparation des préoccupations améliorée. Cependant, la programmation orientée aspect ouvre également la voie à de nouvelles erreurs de programmation que peuvent commettre les développeurs et les intégrateurs d'aspects.

De nombreux travaux s'intéressent à la validation par le test des programmes orientés aspect en s'intéressant notamment au test des nouvelles constructions introduites par le paradigme. Ces travaux couvrent les modèles de fautes associées à la programmation orientée aspect mais aussi le critère de couverture, les niveaux de test et la génération de données de tests.

Ces travaux couvrent un certain nombre de problèmes mais certains d'entre eux restent peu traités par les approches de validation par le test. C'est notamment le cas des fautes liées aux interactions entre les aspects. Ces fautes peuvent compromettre le fonctionnement du système lors de la construction de la configuration de la tolérance aux fautes d'un système ou lors de son évolution. Ce sont ces fautes que nous ciblons dans cette thèse.

1.2 Contributions

Deux moyens peuvent être utilisés afin de pouvoir bénéficier des avantages de la programmation orientée aspects tout en maîtrisant les nouveaux types d'erreur qu'elle induit.

Le premier moyen est la détection à l'exécution lors du test du logiciel, de ces erreurs ; il s'agit alors d'appliquer des mesures curatives visant à éliminer du logiciel sous test ces erreurs. Le deuxième est l'évitement, il s'agit alors d'imposer de bonnes pratiques afin de pouvoir garantir, par des mesures préventives, que le système logiciel construit ne contient pas certains types d'erreurs.

Dans cette thèse nous défendons l'idée qu'il est nécessaire, pour s'assurer de la qualité des logiciels, de combiner les techniques d'évitement avec les techniques de détection. En effet, il n'est pas envisageable de surmonter le coût de la qualité des logiciels au seul prix de la détection, ou d'assurer la qualité sur la seule base de mesures préventives.

La première contribution de cette thèse [45] vise à proposer une méthode alliant évitement et détection d'interférences entre aspects. Nous avons à cet effet comparé des langages permettant de mettre en œuvre la programmation orientée aspect. Ces observations nous ont permis d'analyser comment les aspects sont concrètement composés et d'identifier un certain nombre de problèmes.

La seconde contribution de cette thèse a pour but d'étudier la faisabilité de la méthode d'instrumentation proposée pour la détection. Afin de tolérer les fautes auxquelles peut être sujet un système, une stratégie de tolérance aux fautes est mise en place. Dans ce cas il est important de pouvoir vérifier que l'ensemble des interférences entre aspects ciblés dans cette thèse sont détectées par notre méthode de détection. Nous proposons une évaluation de l'efficacité de l'instrumentation sur laquelle s'appuie notre approche de détection d'interférence. Différentes configurations d'un logiciel où sont tissés des aspects sont instrumentées. Pour chacune de ces configurations, un certain nombre d'interférences doivent être détectées. Nous vérifions dans cette étude de faisabilité que notre instrumentation détecte bien ces cas.

La troisième contribution de cette thèse est l'application de notre approche d'évitement et de détection d'interférences à l'implémentation orientée aspect d'un protocole de réplication duplex. Une stratégie de tolérance aux fautes est composée de plusieurs mécanismes de tolérance aux fautes. Ces mécanismes doivent être implémentés sous forme d'aspects à grain fin permettant la reconfiguration et la résilience. Pour que l'assemblage de ces aspects réalise le comportement attendu du protocole duplex, chaque aspect doit pouvoir réaliser sa spécification sans compromettre l'exécution des autres aspects. L'assemblage des aspects composant le protocole duplex introduit des interférences difficiles à corriger à cause du manque de traçabilité des erreurs.

1.3 Organisation de ce mémoire

Le **chapitre 2** détaille le contexte de cette thèse et analyse l'état de l'art. Le contexte est celui de l'ingénierie logicielle et plus spécifiquement le développement de logiciels sûrs de fonctionnement et résilients. Nous nous intéressons à la tolérance aux fautes au niveau logiciel et à son implémentation à l'aide de la programmations orientée aspect. Nous clôturons ce chapitre par

une discussion sur un type particulier de problèmes qui sont liés aux interactions entre aspects.

Le **chapitre 3** précise la problématique en détaillant les problèmes d'interaction entre aspects et discute les résultats et des travaux existants. À partir de ces observations, nous avons identifié plusieurs problèmes pour la détection des interférences dans les programmes orientés aspect, pour lesquels nous proposons ensuite des solutions dans le chapitre suivant.

Le **chapitre 4** présente le cœur du travail de cette thèse qui est l'implémentation d'une approche permettant la validation d'un assemblage d'aspects. La validation d'un assemblage d'aspects nécessite la vérification d'un certain nombre de propriétés de non-interférence. L'efficacité de notre approche est ensuite évaluée par une série d'expérimentations.

Enfin, dans le **chapitre 5** nous proposons d'appliquer nos travaux à un cas d'étude. Ce cas d'étude consiste à implémenter à l'aide de micro-aspects un mécanisme de réplication duplex dans une architecture distribuée. Nous détaillerons les différents scénarios où des problèmes d'interférences peuvent apparaître lors de l'intégration de ces aspects.

Enfin, la conclusion de cette thèse présente les perspectives pour les travaux futurs.

Chapitre 2

Tolérance aux fautes, séparation des préocupations et validation

Préambule

Ce chapitre présente le contexte de nos travaux et justifie la problématique abordée. Le contexte est celui de l'ingénierie logicielle et plus spécifiquement le développement de logiciels sûrs de fonctionnement et résilients. Nous nous intéressons à la tolérance aux fautes au niveau logiciel et à son implémentation à l'aide d'approches réflexives. Nous présentons un bref état de l'art de l'implémentation de la tolérance aux fautes via des approches réflexives. Nous nous concentrons ensuite sur une spécialisation de l'approche réflexive, la programmation orientée aspect (POA). Nous discutons des bénéfices de cette approche au regard de la résilience et des précautions à prendre au regard du test. Nous clôturons ce chapitre par une discussion sur un type particulier de problèmes qui sont liés aux interactions entre aspects et qui sont au centre de notre travail de thèse.

Sommaire

Construits en intégrant des éléments hétérogènes dans des systèmes multi-plateformes pour répondre aux exigences du marché, les systèmes modernes se construisent majoritairement par la réutilisation de différents composants développés indépendamment : systèmes d'exploitation, bibliothèques spécialisées, Frameworks, intergiciels.

Pour ces systèmes, il est nécessaire d'assurer un degré de sûreté de fonctionnement adapté. Une défaillance du contrôle aérien, lorsqu'elle se produit, a des conséquences humaines et économiques catastrophiques, alors que celle d'une application de géolocalisation est moins critique.

À la complexité inhérente à ces architectures logicielles hétérogènes, dans le contexte des systèmes ouverts, s'ajoute un certain nombre de problèmes liés à la sûreté de fonctionnement et à la résilience. La sûreté de fonctionnement est la propriété qui permet aux utilisateurs de placer une confiance justifiée dans la qualité du service que leur délivre un système. La résilience est la persistence de la sûreté de fonctionnement en dépit des changements. Nous allons tenter de les mettre en lumière dans ce premier chapitre.

Nous commençons tout d'abord par introduire quelques notions de sûreté de fonctionnement, plus particulièrement la tolérance aux fautes, puisque c'est sur cet aspect que nous nous focaliserons dans ce mémoire. Nous revenons ensuite sur la notion de configurabilité et son positionnement au regard de la tolérance aux fautes. Nous définissons à cette occasion la notion de *séparation des préoccupations* (SoC) qui est nécessaire pour la résilience.

Nous discutons d'une solution s'appuyant sur des techniques réflexives. Ces approches réflexives constituent une solution élégante permettant d'aboutir à la séparation des préoccupations et donc à une implémentation configurable, résiliente de la tolérance aux fautes. Nous explorons différents travaux réalisés autour de l'implémentation à l'aide d'approches réflexives de la tolérance aux fautes en particulier à l'aide de la programmation orientée aspect. L'utilisation d'une telle technologie appelle évidemment de nouvelles approches de validation que nous présentons brièvement. L'étude de ces deux thèmes, sûreté de fonctionnement et configurabilité via la programmation par aspect, nous conduira naturellement à préciser en conclusion notre problématique : la validation des architectures composées à base d'aspects.

2.1 Sûreté de fonctionnement et tolérance aux fautes

Nous allons dans un premier temps définir l'ensemble des notions permettant d'appréhender la sûreté de fonctionnement. Ces notions sont considérées indépendamment de la nature du système auquel elles s'appliquent. Pour ce faire, nous utilisons la taxonomie proposée par JEAN-CLAUDE LAPRIE et al. [9].

Ce paragraphe définit les notions de la sûreté de fonctionnement dans leurs grandes lignes, à partir des notions de faute, d'erreur et de défaillance. Une fois ces notions définies, nous pourrons aborder la problématique de cette thèse, qui porte sur l'implémentation au niveau logicielle des mécanismes de tolérance aux fautes.

Nous décrivons à présent en quelques mots les moyens proposés pour réaliser des systèmes sûrs de fonctionnement, en détaillant plus particulièrement l'un d'entre eux, la tolérance aux fautes dans la section qui suit. D'une manière générale un système rend des services à des opérateurs humains ou à d'autres systèmes.

La sûreté de fonctionnement d'un système est définie comme :

> *« La propriété qui permet à ses utilisateurs de placer une confiance justifiée dans la qualité du service qu'il leur délivre »*
>
> JEAN-CLAUDE LAPRIE, *2004*, [9]

Lorsque le service délivré par un système diverge du service attendu, le système est défaillant. La sûreté de fonctionnement cherche donc à éviter les défaillances, à prévenir les plus catastrophiques pour améliorer la survie du système.

La prévention des défaillances s'appuie sur deux notions : faute et erreur. Une faute est la cause d'une défaillance. Les fautes étant les causes des défaillances, la sûreté de fonctionnement cherche à les combattre, si possible en évitant qu'elle se produisent (prévention), ou en les éliminant (élimination).

La sûreté de fonctionnement met donc à disposition quatre moyens pour éviter les fautes ou les éliminer :

- **La prévention des fautes** dont l'objectif est d'éviter l'introduction de fautes de développement tant au niveau logiciel qu'au niveau matériel.
- **L'élimination des faute**s consiste à vérifier la présence de fautes lors du processus de développement et à réaliser de la maintenance corrective et préventive sur le système en cours d'utilisation.
- **La prévision des fautes** est réalisée par une évaluation du comportement du système en présence de faute.
- **La tolérance aux fautes** est intégrée au système et agit en ligne en détectant et recouvrant les erreurs du système.

Les trois premiers moyens ne sont cependant pas suffisants pour deux raisons. Ces raisons sont inhérentes à la nature composite d'un système complexe dont les éléments peuvent être matériels ou logiciels. La première vient du fait que pour les logiciels il est impossible de concevoir des systèmes totalement exempts de fautes. La deuxième est liée à l'usure inévitable et à la détérioration des systèmes, agressés par leurs environnements d'exploitation.

L'occurrence d'une faute reste donc possible. Vient alors la notion d'erreur, qui lie l'occurrence d'une faute à une défaillance. L'activation d'une faute amène le système dans un état erroné, un état anormal d'un système, qui peut potentiellement amener à une défaillance. Pour faire face à l'occurrence d'une faute, pour la tolérer, un quatrième moyen, la tolérance aux fautes doit être utilisé.

La tolérance aux fautes, que nous détaillons maintenant, permet de casser cette chaîne de causalité, en empêchant une erreur de se propager jusqu'à l'utilisateur. On améliore donc la sûreté de fonctionnement d'un système en tolérant ses fautes.

2.1.1 La tolérance aux fautes

Dans les systèmes composites (systèmes de systèmes) la défaillance de sous-systèmes se propage au système global. Une faute est la cause adjugée ou supposée d'une erreur. Par pro-

pagation, une erreur crée de nouvelles erreurs. Une défaillance survient lorsque, par propagation, une erreur affecte le service délivré par le système. Cette défaillance peut alors apparaître comme une faute du point de vue d'un autre composant. On obtient ainsi la chaîne fondamentale suivante : → défaillance → faute → erreur → défaillance → faute →.

Les flèches dans cette chaîne expriment la relation de causalité entre fautes, erreurs et défaillances. Elles ne doivent pas être interprétées au sens strict : par propagation plusieurs erreurs peuvent être créées avant qu'une défaillance ne survienne.

La tolérance aux fautes consiste à empêcher la propagation de l'erreur jusqu'aux frontières du système où celle-ci sera alors perçue comme une défaillance par l'utilisateur du système.

La tolérance aux fautes au niveau logiciel s'appuie essentiellement sur deux grandes idées : la détection d'erreur et le recouvrement d'erreur.

- La détection d'erreur permet d'identifier un état erroné comme tel ;
- Le recouvrement d'erreur, permet de substituer un état exempt d'erreur à l'état erroné ; la substitution peut elle même prendre trois formes :
  - La reprise, où le système est ramené dans un état survenu avant l'occurrence de l'erreur ; ceci passe par l'établissement de points de reprise, qui sont des instants de l'exécution où l'état courant peut ultérieurement nécessiter d'être restauré.
  - La poursuite, où un nouvel état est trouvé à partir duquel le système peut fonctionner,
  - La compensation d'erreur, où l'état erroné comporte suffisamment de redondance pour permettre la transformation de l'état erroné en un état exempt d'erreur.

Une stratégie de tolérance aux fautes est une combinaison de moyens permettant de couvrir le modèle de fautes souhaité. Une stratégie de tolérance aux fautes est choisie en fonction du modèle de fautes à couvrir, mais aussi en tenant compte de propriétés comme la disponibilité, fiabilité, robustesse, testabilité, maintenabilité, etc.

Dans les sections qui suivent nous illustrons les difficultés à réaliser une implémentation résiliente de la tolérance aux fautes en utilisant les paradigmes de développement logiciel conventionnels. Nous discutons ensuite des travaux ayant mis en œuvre des solutions réflexives pour palier les problèmes liés à la résilience de la tolérance aux fautes.

2.2 Séparation des préoccupations

La résilience [44] est la persistance de la sureté de fonctionnement d'un système lors de l'évolution de ce dernier. L'essence de la résilience découle d'une vision évolutionnaire d'un système informatique ; un système évolue aux regards de ses fonctionnalités, des mécanismes de tolérance aux fautes garantissant la confiance placée dans ses fonctionnalités élémentaires (ou combinaison de fonctionnalités élémentaires), des ressources dont il dispose, de son environnement.

Un système évolue selon trois axes. Un système fournit des services qui peuvent être sujets à différents types de fautes. Ces fautes peuvent être des fautes résiduelles de conception ou des fautes opérationnelles, logicielles ou matérielles. L'ensemble de ces fautes correspond au modèle de fautes à considérer. Le modèle de fautes doit être couvert par une stratégie de tolérance aux fautes. Puisque nous nous intéressons à la tolérance aux fautes, nous ne traitons pas les évolutions liées au code fonctionnel. Nous nous intéressons aux évolutions liées au change-

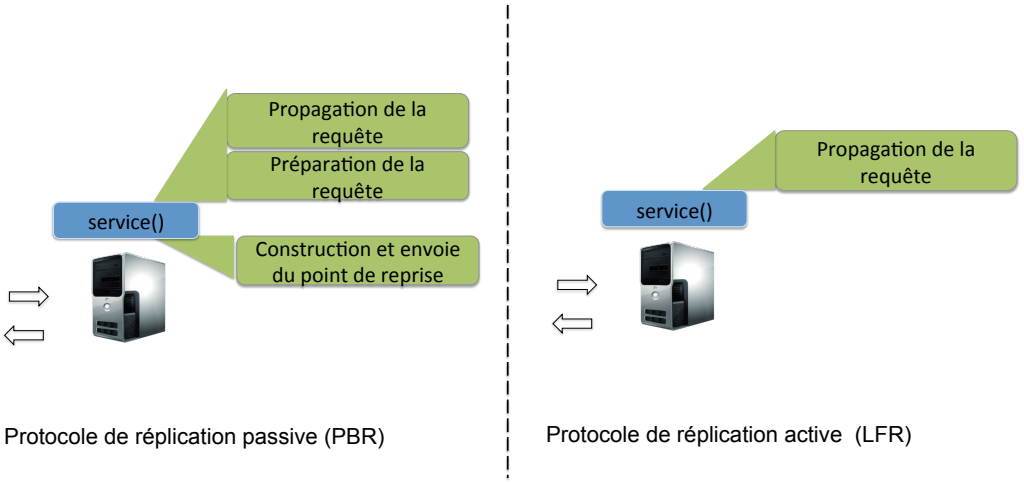


FIGURE 2.1 – Organisation du code fonctionnel et du code non-fonctionnel coté serveur primaire pour un protocole de réplication passive (partie gauche) et un protocole de réplication active (partie droite)

ment des propriétés de sûreté de fonctionnement, aux changements dans l'environnement, à la disponibilité des ressources et aux hypothèses liées aux différents composants du système.

Par exemple considérons un système distribué dans lequel un serveur fournit un service à plusieurs clients. Supposons que ce serveur soit un composant auto-testable parfait (*self checking*). Son seul mode de défaillance est donc le *crash*.

Un moyen intuitif de tolérer les fautes de type *crash* est de remplacer la réplique défaillante par une réplique non-défaillante. Plusieurs solutions sont possibles. Ces dernières s'appuient sur le principe de la réplication. Dans ce contexte, deux serveurs (approches duplex) sont capables de fournir le service, quand le premier défaille, le second prend le relais. Plusieurs stratégies de réplication peuvent être mises en œuvre :

- La réplication passive consiste à exécuter le service uniquement sur le serveur primaire, la synchronisation est alors faite par transfert d'état par des points de reprise.
- La réplication active consiste à exécuter le service sur le serveur primaire et le serveur secondaire.

L'implémentation de ces stratégies consiste à ajouter du code non fonctionnel dédié à la réplication "*avant*" et "*après*" ou "*autour*" de l'exécution du service.

La figure 2.1 illustre ce principe et présente l'organisation des implémentations simplifiées d'un protocole de réplication passif (partie gauche) et celle d'un protocole de réplication active (partie droite).

Nous considérons qu'un protocole de réplication passive est mis en place pour tolérer les fautes de type *crash*. Supposons maintenant que le trafic sur le réseau contenant les nœuds hébergeant les serveurs primaire et secondaire soit saturé par le transfert d'état par point de reprise. Il nécessaire de reconfigurer le serveur de manière à ce que cette consommation ne soit pas problématique.

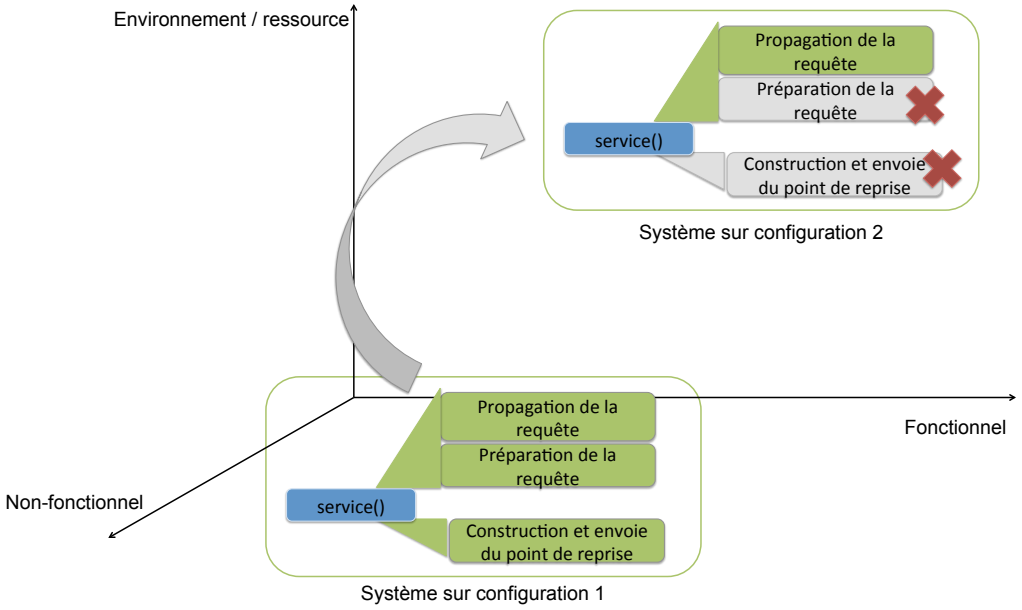


FIGURE 2.2 – Reconfiguration du protocole de réplication passive en protocole de réplication active

Une solution possible consiste à utiliser un protocole de réplication active. Dans ce dernier, les requêtes sont propagées vers le serveur secondaire, qui quand il les reçoit, exécute le service. Il n'y a donc plus de transfert d'état et donc de plus problème de congestion du réseau.

La figure 2.2 illustre le passage d'une stratégie de réplication à l'autre. Un point important à noter est que le passage d'une stratégie à l'autre est facilité par le fait que les différents éléments non-fonctionnels composant le protocole de réplication passive sont implémentés à grain fin. Ces derniers peuvent donc être réutilisés dans différentes configurations mais aussi ajoutés et supprimés facilement puisqu'ils sont modulaires. La granularité est donc un élément fondamental puisqu'elle offre la capacité de reconfigurer le logiciel non-fonctionnel du système en modifiant le moins de code possible. D'une manière générale d'une configuration à l'autre il y a des éléments communs, ici il s'agit du code fonctionnel et des éléments qui varient pour assurer ses propriétés non-fonctionnelles.

Ce scénario nous positionne dans le meilleur des cas, celui où les différents éléments du protocole sont implémentés de façon modulaire. Dans la pratique, cette implémentation à grain fin est difficile à obtenir.

L'ajout des mécanismes de tolérance aux fautes introduit une complexité supplémentaire à l'application de base. Le code des mécanismes de détection d'erreur et de recouvrement introduit une complexité supplémentaire qui peut nuire à la sûreté de fonctionnement d'un système.

Bien qu'un mécanisme puisse être défini syntaxiquement avec seulement quelques lignes de code, les techniques orientées objet exigent qu'une très grande quantité de code soit ajoutée dans le code applicatif pour les utiliser.

Par exemple, le *control flow checking* est utilisé pour détecter les exécutions de flot de

contrôle erronées causées par l'exécution des branches illégales. Le mécanisme repose sur le principe selon lequel, si un programme entre dans un bloc de code, il doit sortir du bloc lors de sa prochaine sortie. Pour pouvoir le vérifier, un identificateur unique pour chaque bloc est placé au début et à la fin du bloc afin de tracer l'exécution. Ainsi le *control flow checking* affecte toutes les fonctions de chaque entité du système et est donc dispersé dans le code de ce dernier. Même si ce mécanisme peut être défini syntaxiquement avec seulement quelques lignes de code, une très grande quantité de code source doit être ajoutée au code applicatif pour le mettre en œuvre.

La dispersion de code lié à une préoccupation aussi appelée *code scattering* est un effet du manque de modularisation de cette préoccupation. L'enchevêtrement de code ou *code tangling* en est une autre conséquence, c'est-à-dire que chaque module comporte le code associé à de nombreuses préoccupations.

Le code lié au différentes préoccupations du système se mélange au code fonctionnel. Un certain nombre de problèmes liés à la qualité du logiciel et sa reconfiguration résultent de l'enchevêtrement et de la dispersion du code non-fonctionnel. Le premier est lié à la réutilisation du code fonctionnel. L'enchevêtrement du code rend difficile la réutilisation du code fonctionnel dans un autre système, puisque le code non fonctionnel est entrelacé avec ce dernier.

La réutilisation du code non-fonctionnel, tel que du code pour la tolérance aux fautes, est encore plus difficile puisque le code est à la fois éparpillé autour du code applicatif et enchevêtré avec d'autres préoccupations. De la dispersion et de l'enchevêtrement du code non-fonctionnel résultent des systèmes complexes difficilement reconfigurables.

Pour faire face à la complexité du logiciel, les programmeurs et les concepteurs ont naturellement essayé d'appliquer le principe "diviser pour mieux régner". Cela s'est traduit par une division des applications en artefacts plus petits, plus faciles à comprendre, à maintenir et à faire évoluer qu'un système monolithique. Cette technique appelée *séparation des préoccupations* a été initialement décrite par PETER MICHAEL-SMITH MELLIAR dans [52]. Le but de la séparation des préoccupations est de pouvoir analyser, développer et raisonner sur certaines parties d'un système sans avoir à prendre en compte les autres parties .

Les paradigmes Orienté Objet (OO) [14] et de l'ingénierie logicielle à base de composants (Component-Based Software Enineering, CBSE) [83], permettent de réaliser partiellement la séparation et la modularisation des préoccupations en combinant des objets/composant simples pour construire des objets/composants plus complexes. La programmation OO et le CBSE offre les moyens nécessaires pour faire évoluer un système sur l'axe fonctionnel dans le temps. Cependant, le code des mécanismes de tolérance aux fautes implémentés à l'aide de ces paradigmes est dispersé dans le code des composants qui sont ainsi difficiles à faire évoluer.

Finalement, qu'une application soit implémentée à l'aide d'objets ou de composants, les préoccupations transversales sont difficilement encapsulées et modularisées dans des entités. Ces préoccupations appelées préoccupations transversales (*crosscutting concerns*) sont des problèmes d'implémentation qui impactent plusieurs entités à différents points de leur code.

Les préoccupations transversales constituent un vrai défit pour les paradigmes traditionnels puisque leur implémentation conduit typiquement à du code dupliqué et dispersé [52]. L'utilisation de langages à objet, ou d'approches à base de composants ne suffit pas pour sépa-

rer les préoccupations transversales comme la tolérance aux fautes du code applicatif.

Comment implémenter les mécanismes et les composants logiciels de manière à ce qu'ils puissent être reconfigurables ? Quelles sont les qualités attendues d'une intégration applicative résiliente de la tolérance aux fautes ?

En conséquence, la capacité à modifier les mécanismes de tolérance aux fautes pendant la vie opérationnelle du système impose cinq qualités : la séparation vis-à-vis de l'applicatif, la transparence, la configurabilité, la composabilité, la réutilisabilité.

- **Séparation** : si le code non fonctionnel est intégré dans le code applicatif, cette dépendance devient problématique lors de l'évolution. Les changements du code fonctionnel impactent le code non fonctionnel et vice versa. La séparation signifie l'indépendance entre le code fournissant la tolérance aux fautes et le code applicatif. Ces éléments doivent être encapsulés dans des entités logicielles distinctes.
- **Transparence** : la transparence des mécanismes de tolérance aux fautes par rapport au code applicatif signifie que l'application du mécanisme est invisible pour l'applicatif.
- **Configurabilité** : la configurabilité correspond à la capacité de choisir les mécanismes de tolérance aux fautes à appliquer à un code applicatif donné. Cette configuration est évidemment guidée par le modèle de fautes à considérer, les conditions d'exploitation, les ressources disponibles. Comme nous l'avons discuté précédemment, dans le meilleur des cas cette configurabilité doit être obtenue à grain fin. Par exemple nous devons avoir la possibilité dans une stratégie de tolérance aux fautes de choisir entre différentes politiques de sauvegarde de l'état courant du système, réplication, sauvegarde sur support stable. Chacun de ces éléments ne représente qu'une partie configurable de la stratégie de tolérance aux fautes.
- **Composabilité** : la composabilité correspond à la capacité de composer plusieurs mécanismes de tolérance aux fautes de natures différentes. C'est une qualité nécessaire lorsqu'il s'agit de composition à grain fin puisque la construction d'une stratégie de tolérance aux fautes passe par la composition de micro-mécanismes. Par exemple un mécanisme de réplication peut être composé d'un mécanisme qui capture l'arrivée d'une requête, d'un autre qui capture l'état de la requête après le traitement de la requête, un troisième qui envoie cet état à une réplique pour une éventuelle restauration du système ultérieurement.
- **Réutilisabilité** : cette propriété correspond à la capacité de réutiliser dans différentes applications un mécanisme de tolérance aux fautes sans avoir à le modifier. La réutilisabilité des mécanismes procure une certaine latitude dans la composition des stratégies de tolérance aux fautes. Par exemple, lors du passage d'une stratégie de tolérance aux fautes à une autre, la conservation de certains mécanismes présents dans l'ancienne stratégie pour les utiliser au sein de la nouvelle stratégie à mettre en œuvre est un avantage intéressant.

La séparation et la transparence permettent l'évolution du système sur un plan fonctionnel et non fonctionnel de manière isolée avec un impact maîtrisé des modifications à effectuer.

Configurabilité, composabilité, réutilisabilité permettent de construire des configurations de la couche non fonctionnelle du système avec des *points de variation* à grain fin en fonction de différents paramètres (ressources du système, hypothèses..).

Malheureusement ces propriétés ne sont pas offertes par les langages de programmation

orientée objet ou les langages à composant. Modulariser les préoccupations transversales requiert des mécanismes qui vont au-delà des concepts traditionnels comme les classes, les composants. Les travaux autour des architectures réflexives et des protocoles à méta-objet (MOP) s'attaquent à ce problème.

2.3 Approches réflexives pour la tolérance aux fautes

L'objectif de nos travaux est de fournir une méthode de validation d'un système implémentant des mécanismes de sûreté par une approche à grain fin. Dans ce but, nous nous appuyons sur une approche réflexive afin de séparer le logiciel de tolérance aux fautes, des fonctionnalités du système. Dans un premier temps, nous rappelons les principes de la réflexivité en informatique ainsi que les travaux utilisant la réflexivité pour l'implémentation de la tolérance aux fautes. Ensuite, nous présentons la notion de *programmation orientée aspects* (POA). La POA est une forme de réflexivité qui fournit des propriétés intéressantes pour l'implémentation de logiciels configurables.

2.3.1 Approche réflexives

Comme cela a été montré depuis de nombreuses années, une solution pratique pour l'implémentation de mécanismes orthogonaux à l'application est l'utilisation de la réflexivité [71, 49], particulièrement sous la forme de protocole à méta-objets, ainsi que des framework de compilation ouverte. En effet, ces notions ont prouvé qu'elles permettent non seulement d'implémenter ces mécanismes d'une façon élégante et efficace mais surtout qu'elles fournissent une grande flexibilité, tout en étant transparentes pour l'utilisateur.

Dans cette section nous rappelons la définition de réflexivité et ses différentes déclinaisons. Nous présentons ensuite différentes applications du concept de la réflexivité et, pour chaque déclinaison, nous présentons un exemple d'implémentation de la tolérance aux fautes à base de plateformes réflexives.

La réflexivité est la propriété d'un système qui est capable de raisonner à propos de lui-même [49]. Est réflexif un système capable d'appliquer à lui–même ses propres capacités d'action, c'est-à-dire selon le système considéré, des capacités de description, de calcul. Ceci introduit la notion de moyens d'analyse, de modification ou d'extension a posteriori du comportement initial du système.

Depuis plus d'une dizaine d'années, la réflexivité a été adoptée comme un outil puissant et général des langages de programmation. Dans [49], PATTIE MAES définit cette propriété comme suit :

> *« Un système informatique est dit réflexif lorsqu'il fait lui-même partie de son propre domaine. Plus précisément cela implique que (i) le système a une représentation interne de lui-même, et (ii) le système alterne parfois entre calcul «normal» à propos de son domaine externe et calcul «réflexif» à propos de lui-même. »*
>
> Pattie Maes, *1988*, [49]

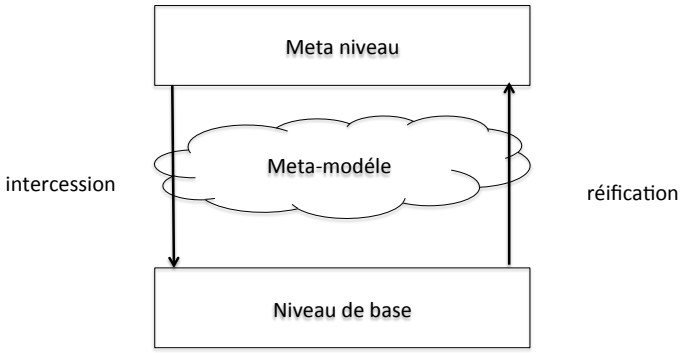


FIGURE 2.3 – Organisation d'un protocole à méta-objet (MOP)

D'un point de vue architectural, le système contient un niveau de base (partie inférieure de la figure 2.3) et un méta-niveau (partie supérieure de la figure 2.3). Le niveau de base correspond au code fonctionnel. A un plus haut niveau, au méta-niveau, sont définies un ensemble d'actions que le système s'applique à lui même. Ces actions sont réalisées sur le niveau de base à travers un méta-modèle qui constitue une interface entre le méta-niveau et le niveau de base. Ce méta-modèle décrit la connaissance qu'il a de lui-même, c'est-à-dire les notions constitutives du système. Le méta-niveau contient le modèle du niveau de base (l'auto-représentation du système) qu'il peut interpréter et modifier.

Le système et son auto-représentation sont liés de manière causale : si l'un d'entre eux est modifié, l'autre est modifié en conséquence. Un système réflexif peut donc s'auto-modifier en modifiant son auto-représentation, et son auto-représentation doit toujours rester cohérente avec son état réel.

Cette liaison causale est réalisée à travers deux processus d'interaction, *la réification* qui s'opère du niveau de base vers le méta-niveau lorsque le premier subit une modification devant être reflétée dans le second, et *l'intercession*, qui va du méta-niveau vers le niveau de base lorsqu'une modification du premier doit s'opérer sur le dernier. Ces deux notions sont en pratiques liées à l'observation et à la commande du niveau de base et définissent un protocole à méta-objet (cf. figure 2.3).

Dans cette architecture, au méta-niveau le développeur est capable d'exprimer des traitements en termes extrêmement génériques, puisque ceux-ci s'appuient sur les éléments du méta-modèle du langage. Dans un système orienté-objet, un exemple de traitement réflexif serait par exemple la création de point de reprise qui peut s'exprimer comme suit :

```
1 A chaque modification d un attribut persistant, sauvegarder sa
    nouvelle valeur.
```

Listing 2.1 – Exemple d'expression de création de point de reprise

À travers cet exemple nous pouvons noter que ce type de méta-programme est indépendant du système de base puisqu'il est exprimé en terme "d'attribut", de "valeur" pour tout programme orienté objet, et peut s'appliquer qu'il s'agisse du système d'exploitation embarqué d'une voiture ou d'une application ludique. Même si cette généricité trouve ces limites dans la complexité de l'état à traiter et de la dépendance à l'égard du système, dans le meilleur des cas un système réflexif permet donc d'atteindre les objectifs d'indépendance, de transparence, et

de réutilisation.

- **L'indépendance** : elle résulte du fait que les méta-programmes attachés à un système de base, réalisant des fonctionnalités transversales sont complètement orthogonaux aux aspects applicatifs du système. Le code du système applicatif n'appelle plus le code transversal, le code transversal est appelé à chaque fois que le code d'un élément applicatif est appelé. Cette méthode aussi appelée injection de dépendance [61] procure une capacité de découplage entre les aspects applicatifs et les aspects non-fonctionnels d'un système.
- **La réutilisabilité** : le code non-fonctionnel est également réutilisable puisque une fois implémenté, un méta-programme peut être réutilisé de manière transparente sur tout système procurant le modèle de programmation réflexive. Notre méta-programme de sauvegarde d'état par exemple pourrait s'appliquer à un système d'exploitation pour construire des points de reprise pour certaines applications ou à une application ludique permettant la reprise d'une partie après un arrêt inattendu du système.
- **La transparence** : elle est la conséquence de l'utilisation du méta-modèle du langage comme niveau d'indirection entre le code fonctionnel et le code non fonctionnel. La réflexivité apporte des capacités supplémentaires au langage pour organiser la structure interne d'un système de façon à faciliter son développement, sa réutilisation, sa reconfiguration.

Dans les paragraphes qui suivent nous détaillons un peu plus comment la réflexivité se réalise dans les langages interprétés, les langages compilés ainsi qu'au niveau des intergiciels.

2.3.2 Implémentation réflexive de la tolérance aux fautes

La programmation orientée objet et les approches à composant offrent les outils nécessaires pour gérer la modularisation des préoccupations fonctionnelles d'un système. Aussi, nous nous focalisons sur les approches réflexives proposant une sur-couche à un système orienté objet. Il existe plusieurs sortes de réflexivité : la réflexivité au niveau des langages de programmation dont nous avons détaillé les principes, la réflexivité à la compilation qui est celle mise en œuvre dans la plus grande part des travaux relatifs à la tolérance aux fautes.

Un exemple représentatif d'utilisation de la réflexivité à la compilation est le Framework FRIENDS [29]. FRIENDS présente une utilisation complète de la réflexivité à la compilation avec un support à l'exécution illustrant l'apport de l'utilisation de méta-objets dans un système distribué tolérant les fautes. La réflexivité est utilisée de deux manières. Tout d'abord, à la compilation, elle permet de modifier le programme fonctionnel afin d'introduire des mécanismes réflexifs. Ensuite, à l'exécution, lorsque la classe est instanciée en objets, les mécanismes réflexifs introduits permettent à un intergiciel de dynamiquement munir ces objets de méta-objets qui fournissent des services de tolérance aux fautes à l'exécution. Le programme en charge de cette transformation est un compilateur. Un langage de programmation réflexif compilé s'appuie sur un compilateur réflexif autrement appelé compilateur « ouvert ». Dans le cas de FRIENDS [29] il s'agit de OPENC++ [19]. Ce compilateur réflexif possède une connaissance du langage qui lui permet de raisonner sur l'objet de la compilation (une classe), et d'interpréter cette compilation selon un méta-objet de compilation (une méta-classe). La figure 2.4 illustre ces notions. Le processus de compilation est rendu réflexif, et il permet de munir le langage cible d'un protocole à méta-objet spécialisé.

Les qualités apportées par cette utilisation sont, par exemple, une très bonne séparation des concepts entre application et tolérance aux fautes, sécurité et une grande facilité de composition des différents mécanismes implémentés dans des méta-objets distincts et indépendants.

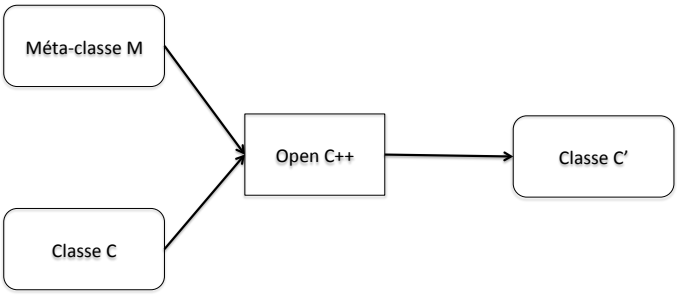


FIGURE 2.4 – Protocole à méta-objet à la compilation

2.3.3 Discussion

La réflexivité des langages fournit de nombreux avantages permettant d'implémenter des systèmes résilients, puisque les mécanismes de tolérance aux fautes sont indépendants, réutilisables, transparents. Des efforts ont été réalisés pour fournir ces mêmes avantages aux langages compilés. Cependant, les implémentations réflexives de langages compilés comme OPENC++ présentent un certain nombre de limitations. La plus importante est celle qui concerne les capacités d'intercession et de réification. Par exemple dans OPENC++ version 1, le seul évènement qui peut être capturé est l'invocation de méthode. Cette capacité limitée de réification et d'intercession est une entrave au développement de certains mécanismes qui requièrent davantage de capacités d'action et d'observation. Par exemple la capture du changement de valeur d'un attribut d'état est nécessaire dans un mécanisme de construction de point de reprise.

Dans la lignée des langages réflexifs à la compilation, les travaux de GREGOR KICZALES et JOHN LAMPING [39] étendent les capacités d'intercession et de réification de JAVA dans une extension appelée ASPECTJ. À ces améliorations déjà notables, ces travaux apportent également des clarifications sur un certain nombre de points autour de la gestion des préoccupations transversales. Un paradigme à part entière voit le jour sous le nom de programmation par aspect (POA).

Dans la section suivante nous présentons la POA et nous fournissons quelques exemples d'application de le POA à l'implémentation de la tolérance aux fautes.

2.4 Programmation orientée aspect

Dans cette section nous présentons les concepts de la programmation orientée aspect. Nous nous appuyons sur ASPECTJ pour illustrer ces concepts. Nous présentons également un exemple d'application de la programmation orientée aspect sur un petit système.

2.4.1 Concepts et terminologie

> « *Aspect-Oriented Programming is Quantification and Obliviousness.* »
>
> ROBERT E. FILMAN , *[31]* , 2000

Deux caractéristiques de la programmation orientée aspect nous intéressent. Il s'agit de la capacité de désigner des éléments du code fonctionnel par quantification et de la capacité de

pouvoir ajouter des éléments comportementaux et structuraux à un système sans que ce dernier en soit conscient (OBLIVIOUSNESS).

La programmation orientée aspect permet l'implémentation des préoccupations transversales en les encapsulant dans des modules séparés appelés aspects.

Un langage de programmation orienté aspect généraliste propose quatre caractéristiques :

1. un modèle de points de jonction qui décrit les points dans le code fonctionnel où un comportement non-fonctionnel peut être ajouté ;
2. un mécanisme pour sélectionner ces points de jonction dans le code sous la forme d'un langage compact ;
3. des modules qui encapsulent la spécification des points de jonction sélectionnés et le comportement à ajouter ;
4. un processus de tissage pour combiner le code de base et les aspects [35].

ASPECTJ [39], est le langage de POA le plus populaire à ce jour. C'est une extension du langage JAVA ayant pour objectif de généraliser la POA. Dans ASPECTJ, les aspects sont des modules qui combinent la spécification des points de jonction - les expressions de coupe, des greffons, qui implémentent le comportement que l'on souhaite ajouter à un point de jonction et les structures régulières de la POO telles que les méthodes et les attributs.

2.4.1.1 Langage de point de coupe

Dans ASPECTJ, on trouve une composition de différentes expressions de point de coupe, soit une expression de point de coupe primitive, soit une référence vers une autre expression de point de coupe. Les expressions de point de coupe primitives permettent de sélectionner des points de jonction et sont la base des expressions de point de coupe. Un élément intéressant d'ASPECTJ est son langage d'expression de coupe. Ce langage s'appuie sur un modèle de point de jonction qui définit les éléments qu'une expression de coupe peut sélectionner dans le code de base. La sélection des points de jonction est réalisée grâce à des expressions de coupe primitive et des motifs que l'on peut écrire de manière plus ou moins générique grâce à l'utilisation de *jokers* (méta-caractères). Les expressions de coupe peuvent par ailleurs être contraintes de manière statique ou dynamique.

Types de points de jonction statiques

En ASPECTJ, les points de jonction sont situés dans le flot d'exécution et peuvent donc dépendre de l'état du système à l'exécution. Cependant, à chaque point de jonction correspond un point de jonction statique qui identifie une partie du code source. La partie de code source auquel correspond un point de jonction est aussi appelée "*ombre*" du point de jonction (*joinpoint shadow*). Quelques exemples de points de jonction statiques en ASPECTJ sont présentés dans le tableau 2.1.

Motifs et jokers

Nous avons présenté ci-dessus des exemples d'éléments du programme de base qui peuvent être sélectionnés par les expressions de coupe.

Afin de désigner ces éléments dans un programme, il est nécessaire d'utiliser des motifs ou *pattern* dans les expressions de point de coupe. Dans le langage de point de jonction d'ASPECTJ, il existe quatre types de motif :

- les motifs de classe,
- les motifs de méthode,
- les motifs de constructeur,
- les motifs d'attribut.

Chaque motif possède une syntaxe propre qui permet de désigner un élément. Lorsqu'il est nécessaire de désigner plusieurs éléments, il est possible d'utiliser des jokers (*wildcard*). Il existe deux types de jokers :

- "*****" peut être utilisé dès qu'un nom est attendu (nom de méthode, de classe, etc.), il permet de désigner n'importe quelle suite de caractères.
- "**..**" peut être utilisé dès qu'un ou plusieurs paramètres sont attendus (paramètres d'une méthode ou d'un constructeur), il permet de désigner n'importe quelle suite de paramètres.

TABLE 2.1 – Types de points de jonction statiques en ASPECTJ

Type	Correspondance	Ombre
Appel	l'appel d'une méthode ou d'un constructeur	l'appel en lui-même
Exécution	l'exécution d'une méthode ou d'un constructeur	le corps complet de la méthode ou du constructeur
Lecture	l'accès en lecture à un attribut	l'accès lui-même

Expressions de point de coupe primitives

Pour sélectionner les éléments du programme de base, ASPECTJ offre des expressions de coupe primitives. Elle peuvent appartenir à trois catégories. La première catégorie sélectionne des points de jonction statiques. Les deux autres catégories permettent d'exprimer des restrictions sur la sélection réalisée par les expressions de coupe primitive. Par exemple la seconde catégorie restreint statiquement l'ensemble des points de jonction sélectionnés, tandis que la dernière catégorie restreint dynamiquement l'ensemble des points de jonction sélectionnés. Finalement les différentes expressions de coupe peuvent être combinées par des opérateurs logiques afin de construire des expressions de coupe complexe.

Sélection de points de jonction statiques.

Ces expressions de point de coupe primitives sélectionnent des points de jonction statiques. Par exemple, *call(Motif de méthode)* sélectionne les appels aux méthodes décrites par le motif de méthode (points de jonction statiques de type appel).

Contraintes statiques et contraintes dynamiques

Des expressions de point de coupe primitives peuvent être utilisées pour restreindre statiquement ou dynamiquement les points de jonctions sélectionnés. Les expressions de point

de coupe primitives statiques permettent d'exprimer des contraintes statique sur la sélection des points de jonction qui peuvent être résolues à la compilation. Par exemple *within(Motif de type)* restreint la sélection aux points de jonction dont l'ombre se trouve dans le code des classes décrites par le motif de type.

Les expressions de point de coupe primitives dynamiques restreignent dynamiquement les points de jonction sélectionnés, c'est-à-dire que les contraintes ne peuvent être résolues qu'à l'exécution. Ces expressions de point de coupe ne peuvent pas utiliser de joker car ASPECTJ ne peut résoudre les jokers qu'à la compilation. Un exemple de primitive dynamique est *this(Classe)* qui restreint la sélection aux points de jonction où l'objet courant est une instance de la classe passée en paramètre.

Une expression de point de coupe statique est une expression de point de coupe qui peut être calculée statiquement car elle ne contient pas d'expression de point de coupe primitive dynamique. Au contraire une expression de point de coupe dynamique ne peut être calculée qu'à l'exécution car elle contient une contrainte dynamique. À la compilation, seule la partie statique d'une expression de point de coupe dynamique est calculée (ombre des points de jonction). Lors de l'exécution les contraintes dynamiques sont résolues afin de déterminer si le greffon doit être exécuté ou pas.

Composition des expressions de point de coupe

Pour construire des expressions de point de coupe complexes, il est possible de composer différentes expressions de point de coupe. L'opérateur binaire de conjonction (&&) restreint les points de jonction sélectionnés aux points de jonction sélectionnés par les deux expressions de point de coupe. L'opérateur binaire de disjonction (||) permet de sélectionner les points de jonction sélectionnés par une des deux expressions de point de coupe. L'opérateur unaire de négation (!) sélectionne les points de jonction qui ne sont pas sélectionnés par l'expression de point de coupe.

En ASPECTJ, chaque greffon est associé à une expression de point de coupe qui lui est propre.

2.4.1.2 Greffons

Les greffons peuvent être exécutés "*avant*", "*après*", ou "*autour*" des points de jonction sélectionnés par l'expression de point coupe correspondante. Les greffons peuvent également accéder et manipuler les informations contextuelles à partir du point de jonction qui provoque leur exécution.

2.4.1.3 Définitions inter-types

En ASPECTJ, les définitions "inter-types" aussi appelées "introduction", sont des définitions qui ont lieu dans des aspects. Elles permettent d'ajouter des méthodes ou des attributs à des classes JAVA existantes.

Les définitions inter-types ne peuvent pas utiliser les jokers, et chaque définition est donc spécifique à une classe.

Les définitions inter-types possèdent une visibilité. Celle-ci est définie du point de vue de l'aspect déclarant et peut être déclarée comme privée ou publique. Ainsi un membre inter-type privé n'est visible que par l'aspect qui le définit.

Par défaut un membre inter-type est visible par tout le paquet où se trouve l'aspect déclarant.

Nous illustrons maintenant les concepts de la POA sur un petit exemple que nous réutiliserons tout au long des chapitres qui suivent.

2.4.2 Exemple

Le listing 2.2 illustre la notion de dispersion code. Il s'agit d'une classe *Client* et sa méthode *msgToSend* qui envoie des messages qui représentent des ordres bancaires par l'intermédiaire du réseau.

```
1  public class Client {
2  String msg_content , msg_time ;// state
3
4  public void msgToSend (){
5  msg_time = DateUtils.now ();
6  msg_content = encrypt ( msg_content );
7  log( msg_content );
8  if( currentUser.authen == true ){
9  networkInterface.send ( msg_content +" "+ msg_time )}
10 else { throw new ExceptionAuth () ;}} }
```

Listing 2.2 – Exemple de dispersion de code

Dans le Listing 2.2, le corps de la méthode contient non seulement une instruction pour envoyer des messages (ligne 9), mais aussi d'autres instructions portant sur les préoccupations non fonctionnelles (lignes 6, 7, 8, 10). La ligne 6 chiffre les messages avant l'envoi. La ligne 7 enregistre le message avant de l'envoyer. Les lignes 8 et 10 vérifient l'identité de l'utilisateur pour permettre ou interdire l'envoi.

```
1  public class Client {
2  String msg_content , msg_time ;
3
4  public void msgToSend (){
5  msg_time = DateUtils . now ();
6  networkInterface.send ( msg_content +" "+ msg_time );} }
```

Listing 2.3 – Code fonctionnel sans code non fonctionnel

Le listing 2.3 montre le code correspondant uniquement au code de base. Il est plus lisible que le listing 2.2, puisqu'il traite uniquement une préoccupation. De même, considérer séparément le chiffrement, la journalisation et l'authentification rend le code plus lisible. En outre, une telle conception privilégiant la séparation des préoccupations est plus facile à réutiliser et à maintenir.

Les langages orientés aspect offrent des mécanismes pour insérer le code correspondant dans l'application de base.

Dans l'exemple du listing 2.3 il y aurait un développement séparé des fonctionnalités de base (l'envoi de messages) et des autres préoccupations. Par exemple, nous aurions un aspect dédié à la journalisation, un autre pour le chiffrement des messages et un troisième pour l'authentification des utilisateurs.

```
1 public aspect Alog {
2 pointcut logMsgToSend ( Client clt): call ( * Client
3 .*Send (..) ) && target (clt );
4
5 before ( Client clt ) : logMsg_to_send (clt)
6 { String msg_to_log =clt . getMsg_content ();
7 log( msg_to_log ); }
8
9 public void log( String msg_to_log ) {// do logging } }
```

Listing 2.4 – Code d'un aspect de journalisation

L'intégrateur de chaque aspect précise que le code correspondant doit être insérée à chaque appel de la méthode *msgToSend*. La combinaison du code de base et du code de l'aspect, c'est-à-dire, l'insertion du code des greffons aux points spécifiés, est appelé le *processus de tissage*. Le tissage se fait souvent au moment de la compilation.

Le listing 2.4 montre un aspect de journalisation écrit en AspectJ. Il offre de puissantes constructions d'expression de coupe et de manipulation des données contextuelles des points de jonction sélectionnés. Dans le Listing 2.4, l'expression de coupe *logMsgToSend* sélectionne tous les appels à une méthode de la classe *Client* dont le nom se termine par la chaîne *Send*. Il recueille également un pointeur sur l'objet ciblé par l'appel. Le greffon implémenté en lignes 5-7 est attaché à l'expression de coupe *logMsgToSend* et effectue la journalisation. Le code d'un greffon peut être exécuté "*avant*", "*après*" ou "*autour*" des points de jonction sélectionnés. Ici, nous avons un greffon "*avant*", qui est exécuté avant l'envoi du message.

Lorsque plusieurs expressions de coupe sélectionnent le même point de jonction, un ordre de priorité d'application des différents aspects peut être défini. Ceci est illustré dans le listing 2.5. Le séquencement peut également être laissé indéfini, ce qui donne un ordre arbitraire choisi par le compilateur.

```
1 public aspect AspectPrecedence {
2 declare precedence : AEncrypt , Alog , AAuth ;}
```

Listing 2.5 – Déclaration du séquencement des aspects

2.4.3 La programmation orientée aspect pour la tolérance aux fautes

Plusieurs travaux autour de l'implémentation logicielle de la tolérance aux fautes par la POA ont été réalisés. Nous présentons ici des travaux récents réalisés par RUBEN ALEXANDERSSON et al. [3, 5, 2] et par KASHIF HAMEED [32].

2.4.3.1 Faisabilité et évaluation de la performance

Les travaux effectuent une étude de faisabilité et des analyses de performance. Ils comprennent notamment :

– une évaluation de la faisabilité de l'implémentation des mécanismes de tolérance aux fautes dans un langage de programmation orienté aspect [3],
– une évaluation du coût de la POA en termes de performance [5],
– les différents mécanismes utilisés dans ces travaux voient finalement leur efficacité en termes de couverture évaluée dans une étude s'appuyant sur des injections de fautes [2].

Dans [3], l'étude de faisabilité porte sur l'implémentation des techniques de tolérance aux fautes à travers un ensemble de mécanismes. Cet ensemble se compose de cinq mécanismes, comprenant :
– le *recovery cache* [64],
– le *time redundant execution* [23],
– le *recovery blocks* [62],
– le *runtime checks* et *control flow checking* [56]

L'étude vise à déterminer si un langage propose les constructions nécessaires pour l'implémentation de certains mécanismes de tolérance aux fautes. Ces mécanismes sont choisis de manière à être représentatifs des besoins en termes de détection et de recouvrement des fautes. Ainsi, si cet ensemble de mécanismes peut être implémenté dans un langage orienté aspect alors ce langage est suffisant pour mettre en œuvre la tolérance aux fautes. L'étude porte sur AspectC++ [67] qui est une extension dédiée à la POA de C++.

Les résultats de cette étude montrent un certain nombre de limitations d'AspectC++. L'étude a montré que seulement deux des cinq mécanismes peuvent être implémentés en l'utilisant. Afin de résoudre ce problème, AspectC++ a été étendu. Avec cette extension, l'ensemble complet des mécanismes peut être implémenté.

Le bilan de ces travaux montre finalement qu'une implémentation orientée aspect offre la même capacité de mise en œuvre de la tolérance aux fautes, la même empreinte mémoire et les mêmes performances qu'une implémentation en C++.

Ces travaux donnent par ailleurs quelques indications intéressantes au regard de la résilience.
– Par exemple, l'auteur souligne que l'implémentation du *recovery cache* peut être réutilisée dans l'implémentation du *recovery block* afin d'implémenter un recouvrement d'erreur de type *backward error recovery*. Il s'agit ici d'un exemple où, lorsqu'un mécanisme est implémenté à grain fin, il peut être réutilisé dans l'implémentation d'un autre mécanisme.
– Une autre propriété intéressante illustrée par ces travaux est l'indépendance et la réutilisabilité de certains mécanismes. Par exemple le mécanisme *time redundancy execution* est implémenté de manière générique et permet à l'intégrateur de l'appliquer à n'importe quelle méthode tout en bénéficiant de la séparation des préoccupations.
– Un deuxième exemple illustrant l'indépendance des aspects de tolérance aux fautes proposé est celui du mécanisme *Incremental Recovery cache* qui, grâce à la capacité d'expression de quantification de la POA, ne requiert pas la spécification des variables qui doivent être stockées dans le cache. Le mécanisme s'appuie sur une quantification indiquant qu'à tout changement de valeur d'une variable celle-ci doit être stockée dans un cache pour pouvoir être restaurée.

Ces résultats sont pertinents du point de vue de la résilience puisque lorsqu'un système évolue sur un plan non-fonctionnel certaines parties de son ancienne stratégie de tolérance aux fautes peuvent être réutilisées pour implémenter la nouvelle stratégie. Dans un autre scénario où il s'agit de construire une politique, il est intéressant d'avoir des mécanismes indépendants et réutilisables. Cependant ALEXANDERSON et al. étudie la faisabilité de l'implémentation de mécanisme de tolérance aux fautes élémentaires. La faisabilité de chaque mécanisme est considérée de manière isolée. Cependant la composabilité des différents mécanismes et la validation de cette composition n'est pas abordée.

2.4.3.2 Évaluation de la séparation des préoccupations

Une deuxième série de travaux est réalisée par KASHIF HAMEED et al. [32]. Ces travaux comprennent notamment une évaluation de la séparation des préoccupations dans l'implémentation des mécanismes de tolérance aux fautes via la POA. Les différents mécanismes utilisés dans ces études voient finalement leur capacité de mise en œuvre de la tolérance aux fautes évaluée dans une étude s'appuyant sur des injections de fautes.

Les travaux de KASHIF HAMEED et al. [32] proposent une implémentation orientée aspect des patrons de conception de tolérance aux fautes représentant :

- la détection d'erreur,
- le recouvrement,
- la gestion des exceptions,
- les points de reprise,
- le *watchdog*.

Aux métriques logicielles telles que le couplage, la cohésion et la taille qui ont été appliquées avec succès pour évaluer la qualité des systèmes logiciels OO [20, 12], des métriques supplémentaires dédiées à la POA sont utilisées pour effectuer une évaluation qualitative [66, 22]. Ces mesures supplémentaires telles que la diffusion des préoccupations dans les composants (CDC), la diffusion des préoccupations dans les opérations (CDO), diffusion des préoccupations dans les lignes de code (CDLOC) sont passées en revue. Ces métriques permettent de mesurer l'efficacité de la POA au regard de la séparation des préoccupations.

Le bilan de ces travaux montre que la POA permet de réaliser la séparation des préoccupations liées à la tolérance aux fautes. Cependant des dépendances à l'égard d'éléments du code de base peuvent être créées lors de l'écriture des expressions de coupe. De bonnes pratiques d'implémentation peuvent être utilisées pour pallier ce problème.

2.4.3.3 Discussion

La majorité des travaux réalisés sur l'implémentation de la tolérance aux fautes par la POA porte sur de petits systèmes. De plus, la validation de chaque mécanisme par des tests manuels ou par injection de fautes est faite de manière isolée.

Dans cette thèse, nous nous intéressons à la composition de mécanismes non fonctionnels. Nous pensons que la validation individuelle des mécanismes non-fonctionnels par des tests et par injection de fautes sont des étapes nécessaires dans un processus de validation, mais qu'ils ne sont pas suffisants. L'un des arguments défendus dans cette thèse est qu'il est également nécessaire de valider la composition des aspects. Nous nous focalisons donc sur la validation

de systèmes orientés aspect dans lesquels de nombreux aspects sont composés à un même point de jonction. Nous détaillons à présent les méthodes de test des systèmes orientés aspect existantes.

2.5 Problèmes du point de vue de la validation

La POA permet la séparation des préoccupations, et vise à faciliter la configuration et la reconfiguration. Cependant, elle peut également introduire des fautes qui n'existent pas d'ordinaire dans la programmation orientée objet et qui sont par ailleurs difficiles à révéler. Par exemple, comme mentionné précédemment, les expressions de point coupe peuvent être très complexes. Toute erreur dans ces expressions peut avoir des conséquences non désirées lorsque les aspects sont tissés dans le code de base. Le développement de systèmes sûrs de fonctionnement utilisant la POA requiert des méthodes de validation appropriées.

De nombreux travaux sur le test des applications orientées aspect ont émergé ces dernières années, ciblant différentes facettes du problème. Cette section donne un aperçu des questions de recherche qui ont été étudiées, et des contributions relatives à ces questions :

- Quels types de fautes sont les plus susceptibles de survenir dans un programme orienté aspect ? Cette question a donné lieu à l'apparition de différents modèles de fautes, ces derniers peuvent être utilisés pour orienter les activités de test et de debugging.
- Comment pouvons-nous définir une stratégie de test d'intégration ? Les niveaux de test, unitaires et d'intégration, doivent être revisités.
- Comment sélectionner les données d'entrée adéquates pour les programmes orientés aspect ? De nouveaux critères de couverture des tests ont été proposés.
- Comment décider si un test a échoué ou réussi ? Les travaux connexes ont porté sur la façon d'observer et de vérifier à grain fin des interactions entre les aspects et le code de base.
- Qu'est-ce qui doit être re-testé quand le code des aspects ou le code de base est modifié ? Les tests de non-régression ont été étudiés dans le contexte de la POA.

Pour chaque question de recherche, le paragraphe correspondant pose également certains problèmes ouverts.

2.5.1 Modèle de fautes pour la POA

Un modèle de fautes pour la POA identifie les concepts de la POA et leurs relations qui sont susceptibles de contenir des fautes. Le modèle de fautes candidat le plus général pour les programmes AspectJ a été proposé par Mortensen et Alexander [53]. Ceccato et al. [57] étendent ensuite ce modèle. Un modèle général décrit des fautes telles que : "la sélection incorrecte des expressions de coupe" qui correspond au cas où la spécification d'une expression de coupe manque des points de jonction ou sélectionne des points de jonction non pertinents. Les modèles de fautes plus fins raffinent les fautes générales pour tenir compte des détails de la programmation. Par exemple, une faute dans la force d'une expression de coupe peut provenir d'une mauvaise utilisation des opérateurs logiques (&& , !) ou des wildcards (∗). Un exemple de travaux portant sur ces modèles est décrit dans [16].

Dans le modèle de fautes proposé par Alexander et al. [1] les fautes dues aux interactions impliquant des aspects sont désignées comme suit :

– Une faute peut être une propriété émergente créée par une interaction entre le programme de base et un aspect.
– Une faute peut être une propriété émergente créée par l'interaction de plusieurs aspects.

Les modèles de fautes pour la POA sont très utiles. Ils peuvent être utilisés pour élaborer des critères de couverture des tests ou de conduire les activités de *debugging*. Ils peuvent aussi servir de base pour l'introduction de fautes dans le cadre des analyses de mutation. Dans ce cas, les fautes sont introduites de manière automatique dans un programme, ce qui donne un ensemble de mutants du programme permettant d'évaluer l'efficacité des jeux de test expérimentalement. Des exemples d'environnement de mutation pour les programmes ASPECTJ peuvent être consultés dans [30, 7]. Ces méthodes d'analyse de mutation sont en général liées à des modèles de fautes de bas niveau.

Les travaux sur les modèles de fautes donnent de précieuses indications sur les constructions "dangereuses" du langage. Il reste à exploiter pleinement ces données pour en déduire des règles de programmation pour les développeurs et pour soutenir les activités de vérification telles que la revue du code, les analyses statiques de code, ou les activités de tests.

2.5.2 Niveaux de test pour la POA

La POA impacte considérablement l'architecture globale d'un système logiciel. Du point de vue du test, nous devons identifier ce qui devrait être testé au niveau unitaire et ce que devrait être la stratégie d'intégration, depuis le test unitaire jusqu'au test système.

2.5.2.1 Les tests unitaires

Comme les aspects sont des entités de première classe des programmes orientés aspects, il est souhaitable qu'ils soient traités comme des unités à part entière et qu'ils soient testés spécifiquement. Ceci soulève la question de la construction d'un contexte d'exécution pour les greffons sous test.

Ce contexte peut être facilement construit dans les dernières versions de langages orientés aspect comme ASPECTJ qui permettent d'écrire les aspects avec la même syntaxe que des classes standard. Les informations relatives aux aspects figurent dans le code sous forme d'annotations. Le contexte peut être facilement construit dans ce cas.

2.5.2.2 Les tests d'intégration

Les tests d'intégration des classes et des aspects d'une application peuvent être effectués selon deux perspectives [84] :
– Une perspective centrée sur les aspects, qui teste un aspect avec les classes impactées par cet aspect [84, 46].
– Une perspective centrée sur les classes, qui teste une classe ainsi que les aspects qui peuvent influer sur elle [84, 47].

Ces perspectives sont complémentaires et devraient être utilisées pour déterminer une stratégie d'intégration complète. Dans la littérature, les travaux sur les tests d'intégration considèrent des petits exemples de programmes orientés aspect. Déterminer les niveaux de test pour les applications de plus grande taille exige des travaux plus approfondis.

2.5.3 Génération et sélection de données de test

Le test exhaustif est impossible même pour les programmes les plus simples. Nous devons donc avoir une façon de concevoir nos suites de tests de manière à exercer suffisamment le programme pour trouver la plupart des fautes, tout en restant assez petites pour être utiles dans la pratique. Le choix du test est généralement basé sur des critères de couverture qui déterminent un ensemble d'éléments structurels ou fonctionnels qui doivent être exercés au cours des tests.

Dans le test des programmes orientés aspect, les approches étudiées peuvent être classées en deux catégories :

- Les approches qui exploitent les outils existants de génération de test JAVA pour générer des données de test pour les programmes ASPECTJ.
- Les approches prospectives qui cherchent des critères de couverture spécifiques à la POA.

Les travaux de la première catégorie [80, 79] tirent profit du fait que le compilateur ASPECTJ fournit du byte code JAVA. Cela permet l'utilisation d'outils de génération de test pour des programme JAVA qui s'appuient sur le byte code. Ces outils cherchent généralement la couverture de branches structurelles à l'aide de la génération de données aléatoires. Par conséquent, ces approches génèrent un grand nombre de données de test et ne ciblent pas les fautes qui sont spécifiques à la POA.

Les travaux de la deuxième catégorie sont plus prospectifs et généralement pas encore pris en charge par des outils. De nouveaux critères de couverture sont étudiés, dans le but de prendre en compte les fautes spécifiques à la POA. La plupart des approches se concentrent sur des critères structurels [53, 84]. Classiquement, les critères se basent sur des représentations de flot de contrôle et de flot de données du programme cible, l'essentiel étant ici de construire des représentations adéquates pour des programmes orientés aspects.

Une approche différente adopte un point de vue fonctionnel, basé sur des modèles à état aspectisés [82, 81]. Le modèle est construit dans une perspective centrée sur la classe impactée par un ou plusieurs aspects, et l'accent est mis sur la façon dont un aspect modifie le comportement d'une instance de classe.

Il serait très utile d'avoir des outils de mesure de la couverture structurelle des programmes orientés aspect. Se pose alors la question de disposer de la possibilité de construire des représentations des flots de contrôle et des flots de données pour des programmes complexes. Par exemple, aucune des approches citées ne considèrent l'héritage et le polymorphisme.

2.5.4 Oracles de test

Après la génération et l'exécution des données de test, nous avons besoin d'un moyen (appelé un oracle de test) de déterminer si l'exécution des cas de test est correcte. Les travaux sur le test des aspects étudient différentes manières de vérifier le comportement des aspects par rapport à des attentes précises. Cela nécessite une augmentation de l'observabilité, par instrumentation du programme sous test. En règle générale, l'instrumentation est automatiquement réalisée selon une spécification des oracles.

Dans [24], DELAMARE et al. présentent un outil appelé ADVICETRACER. Cet outil fournit une API pour les testeurs afin qu'ils puissent spécifier explicitement si un greffon doit ou ne doit être activé par un cas de test, et quel est le nombre d'activations attendu. Un tel outil est tout

à fait pertinent pour la détection des fautes relevant de la catégorie "sélection incorrecte d'une expression de coupe".

D'autres approches ont étudié la notion de contrat pour les aspects [63]. Un contrat de classe peut impliquer une propriété d'invariant d'état et des pré / post conditions pour les méthodes. En attachant des contrats à des aspects et à des classes de base, il devient possible d'effectuer à grains fins, en ligne, des contrôles, par exemple vérifier que le greffon exécuté à la place d'une méthode de base conserve la sémantique pré / post de cette méthode.

Ces approches de conception par contrat couvrent les fautes liées aux interactions entre code de base et aspects. On peut ainsi vérifier qu'une expression de coupe sélectionne bien les points de jonctions requis, que le comportement qui y est appliqué est bien celui qui est attendu. Les oracles de DELAMARE et al. [24] et RINARD et al. [63] constituent des éléments essentiels pour outiller le test des interactions aspect-base. Cependant, les interactions entre les aspects ne sont pas couvertes par ces travaux.

2.5.5 Discussion

Les différents critères de test proposés par les travaux existants permettent d'exercer les nouvelles constructions qui peuvent potentiellement introduire de nouvelles fautes. De nombreux travaux, autant en matière de test que d'outillage, visent à la compréhension, la traçabilité des programmes orientés aspect. Les travaux existants permettent le test des greffons, des expressions de coupes, les introductions de manière isolée. Cependant la nature transversale de certaines préoccupations aboutit à des points dans le code où plusieurs préoccupations transversales sont appliquées. Dans ce cas, le test des expressions de coupe, des greffons et des introductions qui interagissent à ces points de jonction ne suffit pas à garantir que le système fonctionne correctement. Le modèle de fautes proposé par ALEXANDER [1] n'est que partiellement couvert par les approches de test proposées. Les interactions entre aspects sont une source de fautes, peu ciblées par les travaux de test, et c'est la problématique que nous abordons dans nos travaux.

2.6 Conclusion

La configurabilité des systèmes se trouve au cœur des préoccupations industrielles. En effet, la réutilisation de composant logiciels pour la construction de systèmes complexes requiert que ces composants soient configurés en vertu de leur environnement de déploiement et des autres composants du système. Ensuite, lorsque ces systèmes évoluent ou font face à des changement dans leur environnement, il est nécessaire de les reconfigurer. Dans ce contexte, il est nécessaire de maintenir la sûreté de fonctionnement du système, de garantir sa résilience.

Une évolution importante des technologies logicielles a été constituée par l'utilisation de la notion de réflexivité. La réflexivité est la propriété d'un système qui lui permet de raisonner et d'agir sur lui-même. L'application de ce concept aux technologies orientées-objet mène à la notion de protocole à méta-objets. Un méta-objet peut contrôler et agir sur un objet en utilisant le protocole à méta-objets. Nous avons présenté en section I.3 diverses utilisations de ces concepts, qu'ils soient utilisés au moment de la compilation ou de l'exécution de l'application. L'avantage majeur des protocoles à méta-objets est de permettre le développement séparé et indépendant de l'application et de mécanismes non-fonctionnels, de tolérance aux fautes par

exemple pour ce qui nous concerne.

Dans la même lignée, des techniques d'implémentation évoluées comme la POA permettent d'aboutir au même résultat avec cependant des capacités supplémentaires d'observation et d'action sur le système. La POA permet de gérer la variabilité des caractéristiques non fonctionnelles d'un système de manière transparente, configurable en encapsulant le code de gestion de la tolérance aux fautes dans des aspects qui sont déployés dans le système en fonction des besoins.

Le code des aspects implémentant la tolérance aux fautes doit être "zéro défaut". De nombreuses approches de validation par le test s'intéressent à différents problèmes posés par la validation de la POA.

Les travaux sur les modèles de fautes donnent de précieuses indications sur les constructions "dangereuses" du langage. Certains travaux exploitent ces données afin de fournir des critères de couverture. Cependant, la majeure partie des travaux de test de systèmes orientés aspect porte sur les interactions entre aspects et code de base. Le traitement des interactions entre aspects lors de la validation nous semble incontournable puisqu'il s'agit d'implémenter des mécanismes de tolérance aux fautes qui sont par hypothèse exempts de fautes (test et validation par injection de fautes). Ces travaux de thèse visent donc à couvrir une partie du modèle de fautes de la POA, liée à la composition des aspects, peu traitée d'un point de vue de la validation par le test.

Les travaux sur le test d'intégration considèrent des petits exemples de programmes orientés aspect. Déterminer les niveaux de test pour les applications de plus grande taille, dans lesquelles les points de jonction impactés par des aspects sont plus nombreux et les aspects plus nombreux, exige des travaux plus approfondis. L'implémentation de mécanismes de tolérance aux fautes à grain fin nécessite de prendre en considération à la fois les dépendances (plusieurs micro-mécanismes qui composent une stratégie) entre aspects et les éventuels conflits (certains mécanismes sont exclusifs) ou les effets de bord.

Les différentes approches de couverture de critères de test raisonnent sur des graphes de flot de contrôle et de flot de données. Se pose alors la question de disposer de la possibilité de construire des représentations des flots de contrôle et de données pour des programmes complexes. Plus précisément, au regard du traitement des interactions entre aspects, quel est le modèle logique qui permet la couverture des interactions, quels sont les critères de couverture de ces modèles que l'on doit réaliser ?

En termes d'oracle, les travaux existants permettent de vérifier les propriétés nécessaires pour valider les interactions aspect-base. Les interactions entre les aspects ne sont pas couvertes par ces travaux. Se pose alors la question de disposer d'un oracle permettant de valider les interactions entre aspects, et de l'identification des propriétés que doit vérifier cet oracle.

Le chapitre suivant aborde donc le problème des interactions entre aspects. Il s'agit de définir plus finement les fautes potentielles contenues dans les interactions entre aspects. Une fois ces fautes identifiées, nous pourrons alors identifier les propriétés permettant de détecter ces fautes pendant l'activité de test. Il s'agit de définir les propriétés qui serviront de base à la construction d'un oracle pour la validation des interactions. Les propriétés liées à l'absence d'interférences indésirables entre aspects constituent le sujet du prochain chapitre.

Chapitre 3

Interactions et interférence entre aspects

Préambule

Ce chapitre précise la problématique et les objectifs de ce travail de thèse. La problématique est celle de la validation des interactions entre aspects lors d'un développement à l'aide de la programmation orientée aspect. L'implémentation à grain fin des aspects de tolérance aux fautes conduit à des interactions entre ces aspects, ce qui constitue des sources potentielles d'erreur. Afin de valider les applications assemblées à partir de ces aspects, il est nécessaire de traiter ces interactions à travers des méthodes de prévention et d'élimination de fautes. Après un bref état de l'art des travaux autour de la détection, de la résolution et de la validation des interactions entre aspects, la problématique et les objectifs de cette thèse sont présentés à travers un ensemble de questions que nous nous sommes posées tout au long de ce travail. Enfin, nous terminons ce chapitre en précisant l'organisation de ce mémoire.

Sommaire

3.1 Contexte : Interactions, interférences entre aspects

À FIN de comprendre les enjeux et les principes de la construction de logiciels reconfigurables par composition d'aspects, dans ce chapitre, nous présentons les travaux les plus importants réalisés autour des interactions et des interférences entre aspects.

La section 3.2 présente une classification des interactions dans un système orienté aspect. La section 3.3 se concentre sur un sous-ensemble de ces interactions, les interactions entre aspects, et présente les différents problèmes liés au traitement de ces dernières durant l'activité de test.

Une synthèse des différentes approches est présentée dans la section 3.4 d'une part au regard des propriétés qui sont attendues pour l'implémentation de mécanismes de tolérance aux fautes à grain fin et d'autre part au regard des facilités qu'elles fournissent pour la validation des interactions.

Les résultats de cette synthèse nous permettent de présenter les motivations de cette thèse en section 3.5.

3.2 Interactions et Interférences entre aspects à un point de jonction

La littérature autour des interactions dans les applications orientées aspect foisonne de termes tels que *interactions, collaboration, interférence, interférence potentielle*. Une étude approfondie et une classification des problèmes de composition d'aspects est souhaitée pour plusieurs raisons :

- D'abord, afin de disposer d'une vision organisée sur les problèmes de composition d'aspects ;
- Ensuite pour disposer d'une cartographie des approches proposées permettant de détecter, résoudre et valider les différents types d'interaction ;
- Enfin, afin de contribuer à la création de solutions généralisées.

Les approches proposées ne s'appliquent souvent qu'à un sous-ensemble des problèmes de composition, de même d'ailleurs que nos propres travaux.

Dans cette section, les différents types d'interaction entre aspects sont présentés, afin de mieux comprendre les conflits qui peuvent apparaître dans la programmation orientée aspect. Les interactions entre les différents éléments d'un programme orienté aspect ont été identifiées et étudiées à travers plusieurs classifications [17, 34]. Nous nous appuyons sur une synthèse de ces classifications afin de définir le paysage des interactions existantes dans un système orienté aspect. Par la suite, nous nous focaliserons sur une partie de ce paysage.

Il est difficile de raisonner d'une manière générale sur des programmes orientés aspects qui utilisent des expressions de coupe et des greffons en raison de deux problèmes fondamentaux soulignés dans [10] :

- Les points de jonctions, les points dans le code où les greffons peuvent s'appliquer, sont très nombreux. À chaque point de jonction, le raisonnement sur le code doit prendre en compte les effets de tous les greffons applicables. Il s'agit du *problème de densité des points de jonction* (*Density of Join Point Shadows problem*).
- Les effets des greffons doivent être compris afin de raisonner sur le flot de contrôle et de données du programme et comprendre comment les greffons peuvent interférer avec

l'exécution d'autres greffons. Lorsque ce raisonnement n'est pas maîtrisé, des conflits peuvent apparaitre dans les interactions entre aspects (*Control Effects Reasoning problem*).

La densité des points de jonction et le caractère transparent de l'application des greffons fait du traitement des interférences une difficulté en soi. Une façon de maîtriser la densité des points de jonction est de limiter les points ou les greffons peuvent s'appliquer, par exemple, en utilisant une certaine forme d'interface explicite entre le code de base et les greffons comme indiqué dans [68, 73] et plus récemment dans [13].

Le raisonnement sur les effets des greffons reste une difficulté. Les approches à base de contrats présentées dans le chapitre 1 permettent de raisonner sur les effets des greffons sur le code de base. Cependant, les effets des greffons appliqués à un point de jonction partagé sur les autres greffons ne sont pas traités par ces approches.

Pour mieux comprendre les interactions entre aspects à ces points de jonction partagés, il est nécessaire d'avoir une vision globale des interactions possibles dans un système orienté aspect. Pour ce faire nous nous appuyons sur une classification des interactions possibles, ce qui nous permet de décrire les interactions que nous traitons dans cette thèse.

3.2.1 Définition est classification des interactions

Les interactions entre aspects apparaissent lors de la composition de plusieurs aspects dans un système. Les questions soulevées par la composition d'aspects sont discutées dans [17] et dans [34], où une classification des conflits entre aspects est proposée. Les interactions décrites par BUSSARD et al. [17] concernent les greffons. HAVINGA et al. [34] s'intéressent aux interactions liées aux modifications structurelles introduites par les aspects. Trois classes de conflits ressortent de ces travaux :

1. les conflits inhérents liés à l'incompatibilité de deux aspects [17] (a) ;
2. les conflits structurels accidentels, lorsqu'un aspect capture accidentellement une introduction faites par un autre aspect (problème de visibilité) ou lorsqu'un aspect ne capture pas une introduction qui a dû être réalisée par un autre aspect (problème d'ordre) (b) ;
3. les conflits comportementaux accidentels, lorsque deux aspects s'appliquent à un même point d'un programme où ils ont des conflits sémantiques [17], [34] (c).

Tout d'abord, lorsque deux aspects s'appliquent à un même point du programme, ils sont dits en interaction, l'intersection de leur coupe n'est pas vide. Il y a alors deux possibilités : soit ils sont incompatibles (a), dans ce cas une exclusion mutuelle doit être précisée [15, 26, 42], de manière à interdire l'utilisation de l'un des aspects. Dans le cas contraire (b) et (c), si les deux aspects doivent être appliquées, leur ordre d'application doit être précisé [15, 26, 76]. Cette procédure est communément appelée *résolution de l'interaction*. Par ailleurs des modifications structurelles qui peuvent être faites à un programme de base (b), par exemple l'ajout de membres à une classe, la visibilité de ces modifications à d'autres aspects doivent être contrôlées [34]. Des modifications de flots de contrôles et de flot de données peuvent être réalisés par les différents aspects appliqués au même point de jonction (c) selon l'ordre d'application des aspects le comportement obtenu n'est pas le même. Il est nécessaire de valider le fait que l'ordre spécifié lors de la résolution de l'interaction réalise bien le comportement attendu.

Dans cette thèse nous nous intéressons uniquement au troisième cas de figure (c). Ce dernier peut à son tour être décliné en plusieurs situations. On peut avoir besoin de définir qu'un aspect doit s'appliquer chaque fois qu'un autre s'applique [15, 26] on parle alors de *dépendance* entre aspects (c.1), ou qu'un aspect s'applique sur un autre aspect (*aspect d'aspect*)[15, 26] (c.2). La troisième déclinaison de correspond au cas où un aspect modifie le flot de contrôle ou le flot de données d'un autre aspect. Il s'agit, lorsque ces interactions ne sont pas désirées, d'interférence de flot de données et d'interférence de flot de contrôle (c.3).

Le premier cas, le cas des dépendances, est actuellement traité dans des librairies d'aspects comme ASPECTOPTIMA [41], dédié à la gestion des transactions, dans la documentation où un libellé *dépendance* explicite les autres aspects de la librairie requis pour chaque aspect. Cette pratique est par ailleurs généralisée dans [65] en tant que bonne pratique pour la documentation des librairies d'aspects et concerne aussi bien les dépendances que les exclusions mutuelles. La principale limitation de cette approche est qu'elle vise essentiellement les aspects d'une librairie identifiés exhaustivement. Aussi les interactions impliquant des aspects d'autres librairies, *les interactions non anticipées*, dans un système ouvert comme ceux considérés dans cette thèse ne sont pas pris en compte.

Le deuxième cas qui consiste par exemple à utiliser un aspect de journalisation pour mesurer l'efficacité d'un aspect de mise en cache (aspects d'aspects) est possible dans la plupart des langages orientés aspect dont ASPECTJ. Cependant, cette pratique rend le raisonnement sur les interactions très complexe. De plus, on retrouve les problèmes liés au séquencement des aspects sur des aspects. Notre recommandation est de limiter l'utilisation des aspects d'aspects.

Le troisième et dernier cas possible s'attache à traiter le cas où les aspects s'appliquent à la chaine. C'est sur ce cas de figure que nous nous focalisons dans cette thèse.

En résumé nous nous intéressons aux interactions non anticipées entre plusieurs aspects exécutés séquentiellement à un point de jonction partagé. Ces interactions qui peuvent être réalisées à travers le flot de données ou le flot de contrôle sont décrites ci-dessous.

3.2.2 Interactions et interférences aux points de jonction partagés

À l'arrivée à un point de jonction partagé les aspects sont exécutés de manière séquentielle. Des effets de bord se produisent alors en raison d'accès en lecture / écriture aux données partagées (interaction de flot de données) ou en raison d'actions affectant le passage du flot de contrôle d'un greffon à un autre ou d'un greffon au code de base (interaction de flot de contrôles).

Afin de définir les interactions de flot de données et de contrôle nous définissons au préalable les données manipulées par un aspect. Nous notons $V_{in}(A)$ un ensemble de variables du code de base que A utilise en entrée dans son traitement et $V_{out}(A)$ un ensemble de variables dans lesquels A stocke les résultats de ces traitements.

Par exemple, les auteurs de [38] considèrent quatre cas d'interférence, les deux premiers étant des interférences de flot de données les deux autres des interférences de flot de contrôles :

– **Changement avant (CB) :**
  Un aspect A accède à une variable $v \in V_{in}(A)$ du code de base, la valeur de cette dernière étant changée par d'autres aspects exécutés avant A. Il s'agit d'une interférence, si

le comportement de A diffère de celui que nous aurions obtenu si la variable avait gardé sa valeur d'origine (début de la chaine d'aspect).

– **Changement après (CA) :**
Un aspect A accède à une variable $v \in V_{out}(A)$ du code de base, la valeur de cette dernière est changée plus tard dans le flot d'exécution par d'autres aspects exécutés après A. En raison de la nouvelle valeur de la variable v, le comportement d'un greffon peut se révéler inadéquat, ou peut être partiellement ou complètement annulé.
– **Invalidation avant (IB) :**
Un aspect exécuté avant A met le système dans un état qui n'est plus un point de jonction pour A. De ce fait, cet aspect exécuté avant A empêche l'exécution de A.
– **Invalidation après (IB) :**
Un aspect exécuté après A amène le système dans un état qui n'est plus un point de jonction pour A, il supprime l'exécution du point de jonction de A après qu'il se soit exécuté à ce dernier.

Dans cette thèse, nous nous en tenons à ces quatre cas. Ils sont suffisants pour notre discussion et ils illustrent les caractéristiques générales et des interférences de flot de données et des interactions de flot de contrôles à un même point de jonction. Il est à noter que ces interactions ne sont pas nécessairement des interférences. Cela dépend du comportement attendu des aspects, c'est-à-dire le comportement attendu par l'utilisateur.

3.2.3 Exemple d'interactions et d'interférences aux points de jonction partagés

Nous considérons trois aspects : *Alog*(aspect de journalisation), *ACrypt* (chiffrement / déchiffrement) et *AAuth* (authentification). Les points de jonction partagés sont atteints chaque fois que le code de base tente d'envoyer un message par un appel à la méthode *send*. Nous supposons que les trois aspects possèdent des greffons de type *before*. À l'arrivée à une instruction *send* dans le code de base, le flot contrôle est transféré du code de base à la chaine d'aspects.

Ils sont exécutées de façon séquentielle, selon un certain ordre par exemple *ACrypt* <*Alog* <*AAuth*< *send*.

Selon l'ordre précédent, et supposant que l'authentification d'*AAuth* réussisse aucun aspect n'est invalidé. Tous sont exécutés. Puis, le contrôle est rendu au code de base pour effectuer l'envoi. L'échec de l'authentification, ainsi que un ordre différent des aspects, produirait un comportement différent. Nous allons maintenant discuter des différentes possibilités et les utiliser pour illustrer les quatre cas d'interférence impactant *ALog*.

Alog peut enregistrer les messages chiffrés (*ACrypt* <*Alog* <*AAuth* <*send*) ou bien en clair. Dans le premier cas, *ACrypt* fait une interférence CB sur le contenu du message. Dans le second cas, *ACrypt* vient après *Alog* nous avons donc une interférence CA. Selon le comportement désiré par l'intégrateur pour *Alog*, nous pouvons interdire un cas ou l'autre.

Les interférences de flot de contrôle sont illustrées en considérant l'ordre de $Alog$ et $AAuth$. Supposons que $Alog$ soit exécuté en premier. Si l'authentification échoue, nous avons une interférence IA : un envoi de message est enregistré, mais l'envoi effectif n'est finalement pas exécuté. Ce comportement est acceptable si le but de $Alog$ est d'enregistrer toutes les tentatives d'émission, il n'est pas acceptable si l'objectif est d'enregistrer uniquement les messages

envoyés. Dans ce dernier cas, il est préférable d'exécuter $AAuth$ avant $Alog$, ce qui induit une interférence de l'IB (l'exécution de $Alog$ est annulée lorsque l'authentification échoue).

Dans le paysage des interactions entre aspects nous nous intéressons aux interactions aux points de jonction partagés ; plus spécifiquement aux interactions de flot de données et de flot de contrôle autour de ces points de jonction. Se pose alors la question de la gestion de ces interactions. Plus spécifiquement au regard des arguments sur les difficultés à raisonner sur les programmes orientés aspect avancés dans [10] comment maîtriser la complexité des interactions aux points de jonction partagés ? Comment les détecter ? Comment spécifier le comportement attendu ? Comment valider le comportement ?

RÉMI DOUENCE et al. [26] proposent à cet effet un Framework pour la détection et la résolution des interactions entre aspects que nous détaillons maintenant.

3.2.4 Cadre pour la détection et la résolution des interactions

Au niveau du code une procédure de traitement des interactions est celle proposée par DOUENCE, FRADET et SUDHOLT dans [26]. Ce framework dédié au traitement des interactions a été adopté de manière consensuelle dans plusieurs travaux [26]. Les auteurs de [26] soulignent la nécessité de séparer le traitement des interactions entre aspects et la définition des aspects. Aussi ils proposent un modèle à trois phases auquel nous ajoutons une quatrième phase qui est la validation de la résolution :

1. **Programmation** : Les aspects qui font partie de l'application sont écrits de façon indépendante, par des programmeurs différents.
2. **L'analyse des conflits** : Un outil automatique détecte les interactions entre les aspects et retourne des résultats informatifs pour l'intégrateur.
3. **La résolution des conflits** : L'intégrateur résout les interactions en utilisant un langage de composition dédié. Il est généralement admis que la résolution automatique n'est pas possible.
4. **Validation de la résolution** : La résolution d'une interaction consistant à fournir un ordre de séquencement des aspects en interaction, il est nécessaire de valider l'ordre défini.

Nous avons présenté dans le chapitre 1 les différents concepts de la programmation orientée aspect utilisés à la première étape de ce processus. Nous nous attaquons aux trois étapes suivantes. Nous détaillons ci-dessous les différentes approches visant à apporter une solution pour les étapes 2 à 4 du processus proposé ci-dessus.

3.3 Interférences entre aspects au niveau du code

Dans le cycle de vie d'un système orienté aspect ces interférences peuvent être détectées, résolues, validées à plusieurs niveaux, au niveau des exigences, des modèles, du code. Nous nous plaçons au niveau du code, et nous considérons les approches pour détecter, résoudre et valider les interactions.

3.3.1 Détection

Différents types d'approches de détection peuvent être utilisés. La première catégorie correspond aux approches syntaxiques. La deuxième catégorie d'approches correspond aux approches sémantiques.

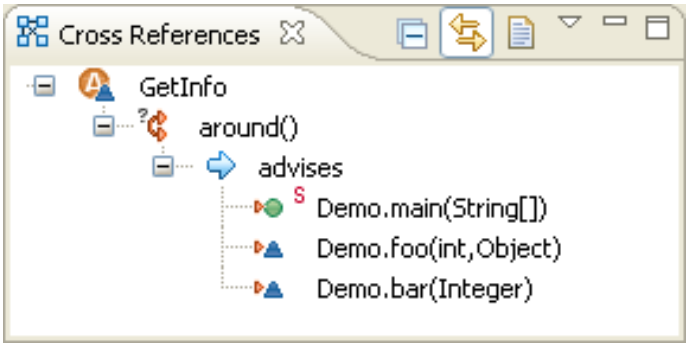


FIGURE 3.1 – Interface de l'outil de détection d'ombre de point de jonction XREF

3.3.1.1 Détection par approches syntaxiques

La détection des interactions au niveau du code peut-être réalisée par différents outils. Chacun offre un certain degré de précision plus ou moins fin en ce qui concerne l'environnement de l'interaction.

Construit sur AJDT [21], XREF [1] est fourni comme une extension pour ECLIPSE [2]. Il permet aux programmeurs de recenser les ombres des points de jonction autour desquels des aspects sont tissés. Il ne facilite que très peu le raisonnement sur les interactions entre aspects. L'identification de points de jonction partagés est obtenue par le calcul de l'intersection des expressions de coupe des aspects. Le calcul de cette intersection n'est pas intégré à XREF et reste à la charge de l'intégrateur d'aspects. De la même manière l'examen de ces interactions potentiellement dangereuses ainsi que l'examen des aspects incriminés et de leur incidence sur les autres aspects incombent à l'intégrateur.

D'autres travaux utilisant des techniques d'analyse de flot de données ont également été réalisés autour de l'identification des influences potentielles des greffons sur des variables du code de base. Ces travaux illustrent le fait qu'un greffon change (ou peut changer) la valeur de certaines variables qui sont utilisées dans le calcul réalisé par des greffons d'un autre aspect et affecte potentiellement le calcul réalisé par ces greffons [78]. Notons que les techniques de SLICING pour les aspects [11] peuvent également être utilisées pour la détection des modifications de variables par des aspects.

Les résultats de cette analyse permettent de classer les interactions entre chaque paire d'aspects comme suit :

- **Indépendant** : les deux aspects n'interagissent pas,
- **Flot de contrôles dépendant** : la présence d'un aspect crée un nouveau flot de contrôle qui conduit à l'application d'un autre aspect,
- **Flot de données dépendant** : un aspect change les valeurs des variables utilisées par l'autre aspect,
- **Co-dépendant** : les aspects s'influencent mutuellement, à la fois par leur flot de données et par leur flot de contrôle.

Cette approche est implémentée comme une extension du compilateur ASPECTBENCH COM-

1. http ://www.eclipse.org/ajdt/xref/
2. http ://www.eclipse.org/

PILER [3] [8] appelé AIDA [4], des informations précises sur les interactions trouvées sont incorporées sous forme d'annotations dans le code intermédiaire tissé [28].

L'utilisateur peut alors parcourir ce code tissé afin d'examiner les liens entre les greffons en interaction et déterminer quelles interactions peuvent conduire à une interférence. Les interactions détectées par cet outil sont des interactions potentiellement dangereuses. Les interactions détectées ne représentent que des sur-approximations des interférences et ne conduisent pas nécessairement à des interférences.

Les approches proposées par [21] et [78] classent les interactions entre paires d'aspects. Du point de vue du test, ces informations sont utiles dans la mesure où elles peuvent servir de base à la détermination de critères de test [77, 46], au calcul de couverture des tests [77, 46]. Cependant, ces informations ne fournissent aucune information utile pour la production d'un oracle de test. En d'autres termes elles ne permettent pas de répondre à la question : est que le comportement observé à ce point du code est une interférence ? Il faut pour cela confronter le comportement obtenu avec celui qui est attendu de la composition des aspects en interaction. Il est donc nécessaire d'introduire des éléments sémantiques pour répondre à cette question.

Nous résumons ici d'autres approches qui se sont intéressées à la détection des interférences sémantiques.

3.3.1.2 Détection par approches sémantiques

La plupart des approches sémantiques s'appuient sur des approches syntaxiques et cherchent à déterminer si une interaction syntaxique est ou n'est pas une interférence sémantique par le biais d'une confrontation du comportement attendu par l'intégrateur d'aspect et le comportement courant obtenu par la résolution courante spécifiée aux points de jonction partagés.

Une façon de différencier une interaction souhaitée d'une interaction non souhaitée, c'est-à-dire une interférence, est d'annoter les aspects avec des informations sémantiques. Un certain nombre de travaux ont déjà exploré cette voie.

Model checking

Dans les travaux de KATZ et KATZ [37], les aspects sont annotés avec des spécifications *assume-guarantee* qui expriment (en logique temporelle linéaire) les propriétés sémantiques qu'un aspect suppose vraies respectivement avant et après son tissage. La détection des interférences consiste alors à vérifier le modèle formé par les formules LTL de chaque paire d'aspects.

Les auteurs soulignent également le fait qu'il est difficile d'utiliser cette méthode en raison de la nécessité de modéliser avec précision les aspects. L'utilisateur doit écrire la spécification des propriétés en logique temporelle, et le modèle à vérifier est dans décrit le langage SMV [51]. L'utilisation de ces langages constituent des obstacles importants pour de nombreux utilisateurs. Ceci a motivé des travaux sur l'aide à l'expression des propriétés [38]. L'approche repose sur une procédure facilement automatisable sous la forme d'un *workflow* où l'utilisateur répond de façon interactive à une série de questions. Ces questions sont guidées par le

3. http ://www.sable.mcgill.ca/abc/
4. http ://www.cs.technion.ac.il/ ssdl/research/cape/aida/aida.html

modèle de faute possible lors d'une interaction (présenté en section 3.2.2). L'utilisateur précise alors ce qu'il attend du comportement de la composition plus précisément dans le cadre de la réutilisation d'aspects dans un système donné. Sur la base des réponses fournies aux questions articulant la procédure, la spécification LTL est automatiquement augmentée. Cette approche permet alors de vérifier si une interaction est une interférence.

Dans notre exemple présenté en section 3.2.3 une interaction de type CB se produit si le chiffrement est appliqué avant la journalisation. La procédure semi-automatique proposée par KATZ et KATZ cible les interférences de type CB à travers la question suivante :

– Existe-il des variables d'entrée de *Alog* pour lesquelles la valeur à l'arrivée au point de jonction doit être préservée jusqu'à ce que *Alog* commence son exécution ?

Une réponse positive à cette question entraine une augmentation de la spécification de *Alog*. Une réponse négative à cette question n'entraîne pas une augmentation de la spécification de *Alog* et une interaction de type CB n'est pas considérée comme un problème. Cette approche où l'utilisateur est guidé nous parait très pertinente.

Dans les travaux de DURR, BERGMANS et AKSIT [27], les propriétés d'un système sont représentés par une bibliothèque de ressources partagées. Les aspects sont autorisés à effectuer des actions sémantiques sur les ressources et sont annotés avec une séquence explicite des actions qu'ils effectuent réellement. Des motifs d'interférences sont ensuite déclarés globalement pour déterminer quelles sont les séquences d'actions autorisées sur les ressources et quelles sont celles qui ne le sont pas. Les interférences sont détectées autour des points de jonction partagés en déterminant des correspondances entre la trace des actions effectuées par les greffons exécutés autour du point de jonction et les motifs d'interférences.

Les deux approches ont en commun le fait qu'elles utilisent les propriétés globales pour annoter les aspects et détecter les interférences. En conséquence, l'annotation des aspects avec ces propriétés globales revient à supposer une connaissance de tous les aspects qui vont être utilisés dans la construction d'un système à construire, alors qu'ils sont censés être implémentés indépendamment. Par ailleurs, l'annotation des aspects à l'aide de propriétés globales rompt la séparation des préoccupations, car elle réintroduit des dépendances. Par conséquent, ces solutions peuvent être difficiles à maintenir lorsque le système évolue puisque des propriétés peuvent être modifiées, ajoutées ou même supprimées.

Déclaration d'intention de composition

Les travaux présentés dans [50] s'attaquent aux limitations évoquées ci-dessus en remplaçant les informations globales sur les interactions par des hypothèses locales appelées *compositional intentions*. Il s'agit d'un mécanisme d'annotation des greffons, qui respecte le principe de séparation des préoccupations et permet la détection d'interférences sémantiques lors de l'exécution. Leur proposition consiste en une méthode simple pour annoter sémantiquement les greffons avec les propriétés attendues de leur composition tout en préservant les bénéfices de la séparation des préoccupations. Ces informations supplémentaires sont appelées *compositional intentions* et elles sont utilisées pour détecter des interférences sémantiques à l'exécution.

Les *compositional intentions* sont définies pour chaque greffon et décrivent les hypothèses

faites sur les actions des autres greffons à un même point de jonction partagé. Chaque *compositional intention* a un type et une description du comportement. Le type d'une *compositional intention* décrit le moment où le greffon doit être exécuté par rapport à d'autres greffons. En plus de avant (*before*) et après (*after*), ignoré (*ignored*) est utilisé pour indiquer que les greffons devraient être ignorés lorsque la partie de description comportement de sa *compositional intention* correspond à un autre greffon. Par exemple, dans notre exemple, le greffon *logArg* de l'aspect *Alog* peut être associé à la *compositional intention* du listing 3.1. Celle-ci est de type *before* et indique donc que le greffon *logArg* doit être exécuté avant tous les greffons effectuant les actions *Write(thisJp.args(1), f)* et *Write(thisJp.args(2), f)* au point de jonction partagé où *logArg* est appliqué.

Les actions *Write(thisJp.args(1), f)* et *Write(thisJp.args(2), f)* correspondent à une description de comportement. Celle-ci est une conjonction et / ou une disjonction de prédicats d'action. Un prédicat d'action désigne une action élémentaire d'un greffon. Par exemple dans le listing 3.1, *Write(thisJp.args(1), f)* est un prédicat d'action. Deux types de prédicat d'action sont disponibles, les prédicats sur le flot de contrôle et ceux sur le flot de données. En ce qui concerne le flot de contrôle, un greffon peut exécuter un point de jonction (*Proceed*), passer outre ce dernier (*NoProceed*), en faire plusieurs appels (*MultiProceed*), ou remplacer les propriétés du point de jonction (*proceedWithSubst*). Les prédicats sur le flot de données, sont disponibles sous une forme abstraite, ils désignent la lecture (*Read*) et l'écriture (*Write*) de variables.

Finalement une *compositional intention* d'un greffon statue que, si la description du comportement de sa *compositional intention* correspond à un autre greffon, en fonction du type de cette *compositional intention* (*before*, *after*, *ignored*), le greffon en question doit être exécuté avant, après ce greffon ou son exécution est complètement ignorée. La figure reprend la compositional intention du listing 3.1 et illustre son utilisation pour la détection d'une interférence de flot de données sur le greffon *logArg*. À l'exécution la *compositional intention* est comparée avec le comportement des greffons de la chaîne d'aspects. Ici lors de cette comparaison une interférence est détectée puisque la *compositional intention* ❶ indique que *logArg* doit être exécuté avant toute ecriture sur *args(1)* et que *ACrypt* effectue une écriture avant l'exécution de *logArg* ❷.

```
1 Before: Write(thisJp.args(1), f)
2 Write(thisJp.args(2), f)
```

Listing 3.1 – Spécification d'une compositional intention pour *Alog*

Dans cette approche, les conflits sont détectés lors de l'exécution : les hypothèses faites pour un greffon sont comparées avec tout le comportement de tous les autres greffons appliqués au point de jonction partagé, si l'hypothèse du greffon incompatible avec le comportement d'un greffon exécuté avant ou après ce dernier, une erreur est signalée à l'utilisateur.

COMPAR

Les approches avancées exigent des spécifications de comportement supplémentaires de l'intégrateur pour chaque greffon. Les auteurs de [58] présentent une façon de valider formellement l'ordre de séquencement des aspects à un point de jonction partagé.

A travers un langage appelé COMPAR, l'intégrateur définit de manière abstraite un domaine d'exécution, la sémantique des greffons et leurs contraintes d'exécution implicites (des post-

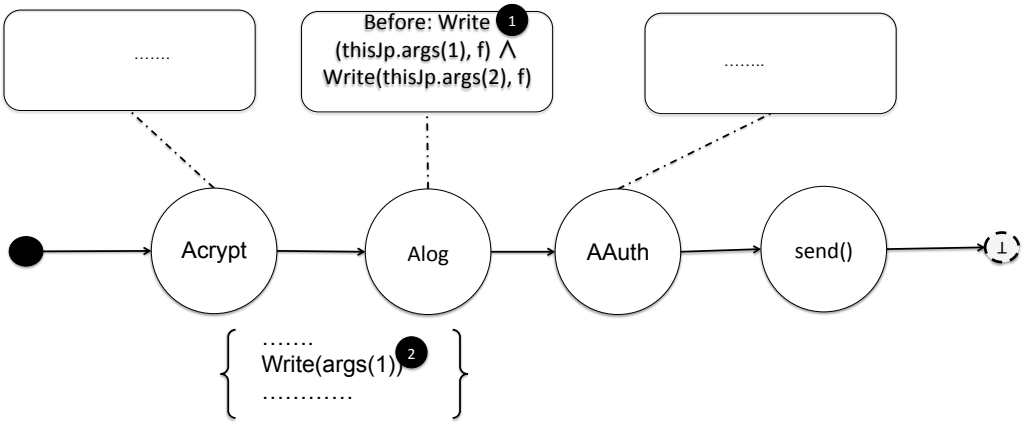


FIGURE 3.2 – Vue schématique de la détection d'interactions à l'aide des *compositional intentions*

conditions) d'un aspect afin de vérifier si les contraintes d'exécution sont remplies lorsque les aspects partagent un point de jonction. Le compilateur COMPAR peut alors vérifier qu'un ordre donné d'application des greffons n'invalide pas une propriété d'un greffon.

Les conflits sémantiques entre greffons peuvent alors être détectés automatiquement en fonction des contraintes spécifiées, ce qui permet donc la détection automatique des interférences. Une contribution majeure des travaux dans [58] est le haut niveau d'abstraction du langage COMPAR qui permet la définition de propriétés génériques sur les aspects.

Afin d'illustrer le fonctionnement de COMPAR, nous montrons comment spécifier le problème de la composition qui a été présenté impliquant les aspects *Alog*, *ACrypt* et *AAuth* présenté précédemment. Pour ce faire, nous écrivons la spécification décrite dans le listing 3.2.

```
1 choices: authenticated;
2 advice Alog:loggingEnter {loggingEnter+loggingExit}
3 advice ACrypt:{ACryptEnter+ACryptExit}
4 advice AAuth { [authenticated]?-+-:throw NotAuthenticated }
5 advised a { Alog, AAuth, ACrypt; }
```

Listing 3.2 – Spécification du comportement attendu dans notre exemple

L'instruction *choices* de la ligne 1 définit les différentes variables booléennes du domaine d'exécution, ici *authenticated* signifie que l'autentification réussie. L'instruction *advice* définit le code abstrait du greffon. La ligne 2 décrit le comportement du greffon *Alog* en indiquant qu'il exécute sa partie "*avant*" (*loggingEnter*), puis exécute le point de jonction (+), enfin exécute sa partie "*après*" (*loggingExit*). L'instruction *advised* figurant à la ligne 5 définit un ordre composition a tester par le compilateur. Lors de son exécution, le compilateur exécute l'ordre a tester dans le domaine (toutes les combinaisons possibles des valeurs choices dans le cas où l'ordre n'est pas défini) et vérifie que les post-conditions de chaque greffon sont réalisées.

Une post-condition figure dans une définition *advice*, celle-ci qui contient deux parties : une contrainte de post-condition optionnelle, définie après le nom des greffons et séparée par un « : » , et un corps, entre accolades, qui représente la définition abstraite du code greffons. Par exemple le greffon *Alog* la contrainte à vérifie est que l'action *loggingEnter* a été exécutée

comme l'indique la ligne 2.

3.3.1.3 Discussion

Trois remarques peuvent être faites sur les approches de détection d'interférences. La première remarque relève du cheminement qui s'est dessiné au fil des travaux autour de la détection. Les premières approches comme XREF se limitent à la détection d'interactions aspect-base. Puis vient l'approche de WESTON qui examine les interactions et leur source en terme de flot de contrôles et de flot de données. Malgré l'efficacité et la précision de ces outils, il reste difficile de se prononcer sur le fait qu'une interaction soit une interférence, ou pas, sans information sur la sémantique. Un certain nombre d'informations sémantiques, sur les flots de données et les flots de contrôles sont nécessaires pour discriminer les interférences. Ces informations sémantiques sont étroitement attachées au contexte de réutilisation et de composition de l'aspect et collectées lors de la composition. C'est ce que propose KATZ et KATZ dans [38].

La deuxième remarque relève de la détection à l'exécution. L'approche proposée par MAROT [50], à base d'intention de composition permet de détecter les interferences à l'exécution. L'implémentation des *compositional intentions* exploite les capacités de SMALLTALK, ces capacités d'intercession et de reification pour observer les propriétés ciblés par les *compositional intention*. En d'autres termes pour MAROT les *compositional intentions* sont un DSAL (Domain Specific Aspect Language) dedié à la détection des interférences.

La troisième remarque vise à souligner le fait que même si des approches comme celles de MAROT sont efficaces pour la détection d'interférence il faut garder à l'esprit que le code de l'intégrateur d'aspect peut introduire des erreurs dans la spécification des propriétés attendues de la composition. Par exemple la détection d'une interférence liée à une interaction de type CB sur l'aspect *Alog* de l'exemple présenté en section 3.2.3 peut être exprimé via des *compositional intentions*. L'intégrateur peut cependant, par manque de compréhension de l'aspect *AAuth*, ne pas spécifier qu'il ne veut enregistrer que les messages effectivement envoyés (interférence IA). Il faut donc appliquer une méthode de spécification systématique, couvrant toutes les interactions possibles comme explicité dans [38].

Les leçons apprises qui découlent de ces remarques et des propriétés attendues de la programmation orientée aspect pour l'implémentation de la tolérance aux fautes suivent. La détection syntaxique des interactions est une première étape nécessaire pour identifier les interactions à couvrir dans une approche de validation. Un enrichissement sémantique est ensuite nécessaire pour indiquer si les interactions détectées sont désirées ou ne sont pas désirées.

Les approches de détection décrites dans ce chapitre s'appuient sur des représentations abstraites des aspects qu'ils soient représentés sous forme de modèle ou d'annotation. Le principal inconvénient de ces approches est qu'elles ne représentent pas forcément le système qu'elles sont censées représenter. Ainsi, il est nécessaire de s'appuyer sur le système réel afin de valider le système et non une abstraction de ce dernier.

3.3.2 Résolution des interactions

Une fois les interactions détectées il faut les résoudre. La résolution consiste à définir un ordre d'application des aspects. Plusieurs travaux proposent des approches pour résoudre les interactions. Nous proposons maintenant trois approches qui représentent trois archétypes

des méthodes de résolution d'interaction et trois points aux extrêmes et au centre du spectre des méthodes de résolution. Cet ensemble d'approches comprend :

- ASPECTJ et les constructions proposées pour la gestion de la précédence entre greffons,
- REFLEX un cadre réflexif possédant les mêmes caractéristiques qu'un protocole à méta-objet,
- une approche hybride appelée AIRIA.

3.3.2.1 Résolution des interactions dans ASPECTJ

D'une manière générale la résolution consiste à permettre aux programmeurs de contrôler la priorité des greffons en interaction. Par exemple, ASPECTJ [21] fournit l'instruction *declare precedence* à cet effet. Dans une déclaration de priorité si un aspect A est défini comme précédent un autre aspect B alors tous les greffons de A sont toujours exécuté avant les greffons de B.

L'inconvénient de cette approche est qu'il est difficile de spécifier des priorités spécifiques. Par exemple dans notre exemple impliquant les aspects *Alog*, *Acrypt* et *AAuth*. Ces trois greffons sont appliqués au point de jonction correspondant à l'appel à la méthode *send()* qui envoie un message à un serveur. Pour notre exemple, puisque nous voulons que lors de l'envoi les messages soient journalisés non chiffrés, puis envoyés chiffrés. La priorité permettant de réaliser ce comportement est donnée dans le listing 3.3. Elle spécifie qu'à tout point de jonction partagé les greffons de l'aspect *ACrypt* sont appliqués avant ceux de l'aspect *Alog* et *AAuth*.

Considérons maintenant que deux de ces aspects, *Alog* et *Acrypt* soient également appliqués à un second point de jonction, correspondant à l'appel à la méthode *receive()* effectuant la réception des messages provenant du serveur. À ce point de jonction les messages reçus doivent être déchiffrés puis journalisés. Au point de jonction *receive()* les greffons *Alog.log* et *Acrypt.decrypt* sont doivent donc être appliqués dans l'ordre suivant : *Acrypt.decrypt* > *Alog.log*. Comme le montre la figure 3.3 la priorité déclarée dans le listing 3.3 ne permet pas de réaliser le séquencement adéquat au point de jonction correspondant à l'appel à la méthode *receive()*.

D'une manière générale pour certaines combinaisons de greffon, il n'y a pas de séquencement correct avec lequel le comportement obtenu à l'issue de la composition est acceptable en utilisant un mécanisme de résolution comme *declare precedence*. Cela est d'une part dû au fait que la déclaration de priorité est faite au niveau des aspects et non des greffons et d'autre part que la déclaration de priorité est faite pour tous les points de jonction et non par point de jonction.

```
1 declare precedence : ACrypt , Alog , AAuth ;
```

Listing 3.3 – Déclaration du séquemcement des aspects

3.3.2.2 Résolution des interactions dans REFLEX

Les travaux de ERIC TANTER [75] ont pour but de permettre l'utilisation de différents langages de programmation orientée aspect ciblant différentes préoccupations d'un système logiciel de manière unifiée dans un même framework. À cet effet un noyau d'architecture multi-langages appelé REFLEX est proposé. REFLEX offre un cadre expérimental qui cible l'ensemble des problèmes liés à la composition que nous avons décrit dans la section précédente. Nous nous focalisons ici uniquement sur ses capacités en termes de résolution des interactions entre

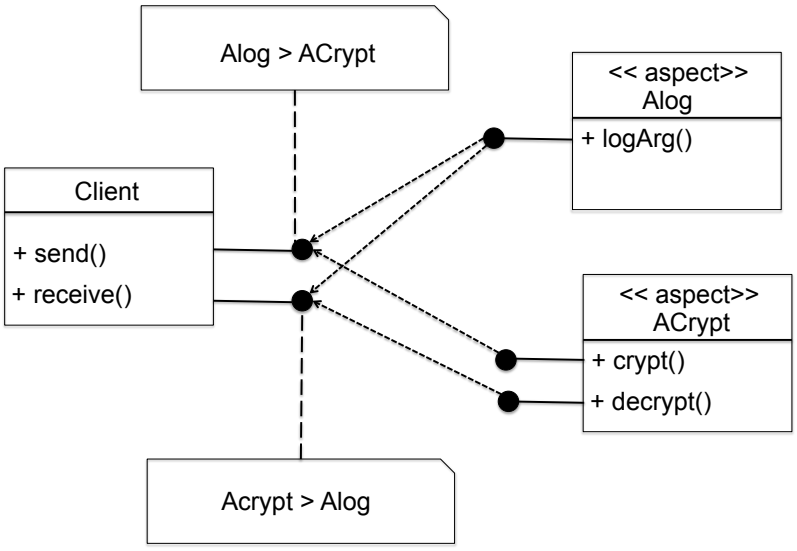


FIGURE 3.3 – Définition de la priorité dans notre exemple

aspects.

REFLEX est une bibliothèque qui étend JAVA avec des capacités de réflexivité structurelles et comportementales. La notion centrale est celle de liens explicites (*link*) liant un ensemble de points du programme (un *hookset*) à un méta-objet. Un lien est caractérisé par un certain nombre d'attributs, parmi lesquels le moment ou les méta-objets agissent ("*avant*", "*après*", "*autour*"), et une condition d'activation évaluée dynamiquement. La figure 3.4 représente ces concepts ainsi que la correspondance avec les concepts de la programmation orientée aspect expression de coupe / greffon.

Notez que la programmation orientée aspect mise en oeuvre dans REFLEX est intrinsèquement liée à la notion de protocole à méta-objet : une coupe est réalisée par l'introspection d'un programme et son action se compose des modifications comportementales / structurelles (intercessions). Cependant REFLEX n'impose pas un protocole à méta-objet spécifique (MOP), mais rend plus facile la spécification d'un MOP sur mesure. Comme dans ASPECTJ aux points d'interactions entre méta-objets, une résolution doit être spécifiée.

Dans REFLEX, les règles de composition de liens sont spécifiées à l'aide des *opérateurs de composition*. Par exemple supposons qu'un point de jonction soit lié à deux méta-objets via des liens *l1* et *l2*. La règle *seq(l1, l2)* utilise l'opérateur *seq* pour spécifier que *l1* doit être appliqué avant-être *l2*, à la fois "*avant*" et "*après*" l'occurrence de l'opération impactée.

Cet opérateur va définir un ordre total *ord()* entre les parties "*avant*" *before(l1) > before(l2)*, l'appel à *proceed around(l1) > around(l2)* et les parties "après" *after(l1) > after(l2)*.

De façon similaire d'autres opérateurs de composition que *seq* sont offerts par le noyau. Ils peuvent être utilisés pour définir des opérateurs de composition personnalisé.

Ord est un opérateur du noyau, qui exprime l'ordre total. Les méthodes de *b* ("*avant*"), *r* ("*autour*"), *a* ("*après*"), sont fournis par l'opérateurs de compostions *ord*. La méthode *expand*, est exécuté à chaque fois que l'interaction se produit entre deux liens.

Les opérateurs de compositions extensibles proposés par REFLEX permettent de spécifier la

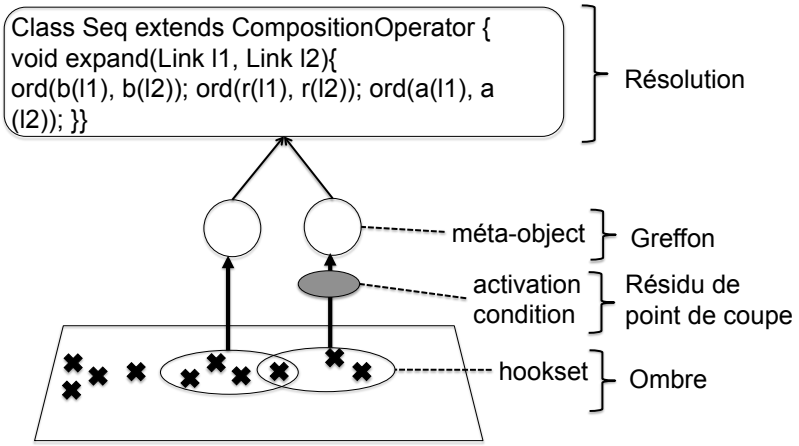


FIGURE 3.4 – Le modèle Link et ces correspondances avec les concepts de la POA

résolution des interactions de manière plus flexible que ce qui est proposé dans ASPECTJ.

```
1 class Seq extends CompositionOperator { void expand(Link l1, Link
      l2){
2 ord(b(l1), b(l2)); ord(r(l1), r(l2)); ord(a(l1), a(l2));
3 }}
```

Listing 3.4 – Exemple d'arbre de resolver généré

3.3.2.3 Résolution des interactions dans AIRIA

FUMINOBU et CHIBA [74] proposent une extension du langage ASPECTJ nommé AIRIA pour la résolution des interactions. Cette solution est à mi-chemin entre ASPECTJ et REFLEX. AIRIA qui fournit un nouveau type de greffon around pour résoudre les interactions entre greffons. Ce type de greffon nommé *resolver* est invoqué uniquement quand un certain ensemble de greffons est en interaction à un point de jonction partagé. Les resolvers peuvent appeler une version étendue de *proceed*, ce qui prend comme argument le séquencement des greffons en interaction au point de jonction.
Un resolver est utilisé pour implémenter la composition des greffons en interaction. AIRIA propose les mêmes constructions que ASPECTJ c'est-à-dire les expressions de coupe et introduction qui peuvent également être utilisées dans AIRIA.

```
1 aspect LogEncryptAuth {
2 void resolver sender():
3 and(ALog.logMsg, ACrypt.encrypt, AAuth.auth){
4  [AAuth.auth, ALog.logMsg, ACrypt.encrypt].
5 proceed();}}
```

Listing 3.5 – Exemple d'arbre de resolver généré

AIRIA de la même manière que REFLEX permet de résoudre les interactions de manière flexible. Le resolver peut être vue comme un opérateur de composition personnalisable. L'un des avantages intéressants de AIRIA est sa capacité à utiliser la quantification pour appliquer la résolution contenue dans les resolvers à différents points du code où cette interaction a lieu.

3.3.2.4 Discussion

La résolution est un mécanisme essentiel des langages de programmation orientée aspect d'une manière générale, mais aussi particulièrement important lorsqu'il s'agit de construire des logiciels reconfigurables sur le plan non fonctionnel. En effet comme explicité dans le chapitre 1, la pérennité de la tolérance aux fautes lors de l'évolution d'un système nécessite de pouvoir composer de manière flexible les aspects composant les protocoles de tolérance aux fautes. En ASPECTJ le mécanisme de résolution, l'instruction *declare precedence* souffre de deux handicaps : il n'est pas flexible (un seul ordre pour tous les points de jonction) et défini à trop gros grain (précédence au niveau des aspects).

REFLEX offre une plus large latitude dans la définition de la résolution des interactions. Indépendamment de ces caractéristiques fortement inspirées des protocoles à méta-objet, cela tient au fait qu'il utilise des opérateurs de composition spécialisables. Nous remarquons que le fait de pouvoir spécialiser les opérateurs de composition au cas par cas est un avantage non négligeable pour la composabilité des mécanismes de tolérance aux fautes.

AIRIA est à mi-chemin entre ces deux solutions. Il propose à la fois un mécanisme de composition à grain fin (au niveau des greffons), la possibilité de composer les aspects aux cas par cas, de manière flexible. L'avantage de AIRIA est l'introduction d'un modèle de points de jonction dédié à la composition des aspects. Aussi, contrairement à REFLEX il n'est pas nécessaire de définir une résolution pour chaque interaction. Il est possible de définir une résolution pour plusieurs interactions dans un resolver et de l'appliquer à chaque point du code pour la résolution doit être appliquée par le mécanisme de quantification. Par ailleurs AIRIA dispose du mécanisme d'expression de coupe qui est aussi intéressant pour une application transparente de la tolérance aux fautes.

Les leçons apprises au regard de cette analyse et des propriétés attendues de la programmation orientée aspect pour l'implémentation reconfigurable de la tolérance aux fautes peuvent se résumer ainsi.

La composition requiert des opérateurs de composition permettant la réalisation de différentes configurations de manière flexible. Idéalement ces opérateurs sont personnalisables pour répondre à une large gamme de variation possible dans la composition des aspects. ASPECTJ se voit donc écarté de la liste des candidats potentiellement utilisables pour l'implémentation de systèmes reconfigurables.

3.3.3 Validation des interactions entre aspects

Dans un programme orienté aspect, les aspects et les classes interagissent de plusieurs manières, à savoir :

- par le biais des paramètres passés à partir des méthodes impactées dans une classe aux greffons dans l'aspect,
- par la lecture ou l'écriture de variables d'état des classes dans un greffon.

Un problème clé avec les critères de test de flot de données existantes est qu'ils ne considèrent pas tous les types d'interactions de flot de données dans un programme orienté aspect. Dans [77], WEDYAN et GHOSH proposent des critéres de flot de données couvrant les interactions qui sont basées sur des variables d'état. Une approche de test basée sur des critères de flot de données pour la programmation orientée aspect est proposée. Les auteurs identifient les différents types de DUAs (*Def-Use Associations*) pour ces variables et propose un ensemble

de critères de flot de données qui nécessitent l'exécution de ces DUAs. L'approche est outillée par un outil appelé DCT-AJ, qui identifie les DUAs dans un programme de ASPECTJ et calcule la couverture obtenue par une suite de test donnée.

Les informations fournies par DCT-AJ constituent un précieux guide pour couvrir de manière exhaustive les interactions d'un système orienté aspect dans une approche de validation. Cependant, en termes de validation il n'existe pas de travaux proposant un oracle de test pour confronter le comportement obtenu lors de l'exécution de ces interactions et le comportement attendu.

3.4 Bilan

Nous reprenons les travaux présentés précédemment afin d'en présenter un bilan. Une procédure à suivre a été dégagée au gré des différents travaux autour du traitement des interférences. Elle consiste à détecter les conflits potentiels, les résoudre, puis valider les interactions. Nous présentons une analyse des approches au niveau du code, leurs bénéfices et les lacunes au regard des différentes étapes de cette procédure. Notons que cette synthèse s'articule autour de la réutilisabilité, de la composabilité des aspects procurée par chaque solution.

DOUENCE et al. [26] soulignent la nécessité de séparer le traitement des interactions entre aspects et la définition des aspects. Puisque réutilisables, ces aspects ne peuvent pas être attachés à une spécification particulière (liée à un contexte précis). Pour des aspects réutilisables la spécification de leurs effets attendus est donc faite à l'intégration. À l'intégration, la détection vise à recenser les interactions, la résolution consiste à spécifier comment doivent interagir les aspects, la validation vise à vérifier que la résolution spécifiée pour une interaction réalise le comportement attendu.

La détection est facilitée par des fonctionnalités du tisseur d'aspects. La détection syntaxique des interactions est une première étape nécessaire pour cartographier de manière plus ou moins précise les interactions à couvrir dans une approche de validation. Un enrichissement sémantique est ensuite nécessaire pour indiquer si les interactions détectées sont désirées ou ne sont pas désirées. En ce qui concerne la détection syntaxique, il faut cependant adopter un bon niveau de raisonnement pour détection ; une détection à grains fins basée sur les flots de données donne des informations sur les variables affectées par des modifications du flot de données. Cependant, l'absence de variables partagées par des aspects ne signifie pas l'absence d'interférences (une interférence de flot de contrôle peut se produire). Par ailleurs nous considérons des systèmes ouverts et évolutifs. Aussi une interaction sans données partagées à une étape du cycle de vie d'un système peut devenir au fil des évolutions une interaction ou le flot de données est partagé par plusieurs aspects. Nous préconisons à cet effet l'application d'un principe de prudence dans lequel la détection des interférences de flot de données et de flot de contrôle est appliquée systématiquement. Finalement le niveau de la simple intersection entre deux expressions de coupe est suffisant pour discriminer une interférence potentielle candidate à un enrichissement sémantique.

La détection des interférences au niveau du code, comme illustré dans les travaux de MAROT [50] nécessite l'instrumentation du code à l'aide d'assertions exécutables. Ces assertions exécutables observent, dans ce cas, des propriétés de non-interférences. Quand une propriété est violée une erreur est signalée pendant la phase de test. Dans le cas de MAROT l'observation de ces propriétés ne présente aucune difficulté puisque ce dernier s'appuie sur SMALLTALK qui est un protocole à méta-objet. Dans ce cas l'observabilité des propriétés sur le code fonctionnel

par le méta-niveau et non-fonctionnel par un méta-méta-niveau peut être réalisée. L'observation de ces propriétés ne semble pas aussi facile dans des langages comme ASPECTJ.

La résolution requiert des opérateurs de composition permettant la réalisation de différentes configurations de manière flexible. Idéalement ces opérateurs sont personnalisables pour répondre à une large gamme de variation possible dans la composition des aspects. ASPECTJ se voit donc écarté de la liste des candidats potentiellement utilisables pour l'implémentation de systèmes reconfigurables. Des Frameworks comme REFLEX permettent de définir des compositions flexibles à travers l'utilisation d'opérateurs de composition configurables. Cependant, ce Framework ne dispose pas de langage d'expression de coupe et est majoritairement destiné au prototypage de langage plus qu'à l'implémentation d'applications industrielles. Un compromis intéressant est proposé par FUMINOBU et CHIBA avec AIRIA qui propose un langage de point de coupe et offre de plus la possibilité de composer les préoccupations transversales, de raisonner à un niveau de granularité fine et d'exprimer la composition de manière flexible.

L'approche proposée par PAWALK [58] et MAROT [50] couvre les phases de résolution et de validation de la composition. Cependant, la résolution demande une certaine expertise du problème d'interférence ce qui peut entrainer une spécification incomplète de la résolution et donc une vérification incomplète. Afin d'éviter ces écueils, l'approche proposée par KATZ et KATZ [38] peut être utilisée afin de diriger la spécification par le modèle de faute des interférences.

La validation du comportement des aspects composés à un point de jonction partagé se fait en exerçant des clusters d'aspects comme décrit dans [84]. Un *cluster d'aspects* est un point de jonction autour duquel plusieurs aspects sont tissés. La génération de données de test à cet effet est décrite par LEMOS dans [46]. Les critères de couverture sont définis dans [77] par WEDYAN et GHOSH. Bien que ne couvrant pas toutes les possibilités de faute, cette approche cible a minima l'activation d'un chemin entre une définition d'une variable et son utilisation en prenant en compte l'application de plusieurs aspects. Cependant, cette approche ne propose pas d'oracle de test permettant de vérifier qu'aucune interférence ne se produit aux différents points d'interactions entre les aspects.

Finalement une approche générale se dessine à l'issue de cette discussion. Dans cette approche, la détection sémantique des interactions est réalisée par une détection syntaxique puis par une instrumentation du code des aspects à l'aide d'informations sémantiques permettant de discriminer une interférence. Cette approche laisse cependant plusieurs questions ouvertes. Comment instrumenter le code des aspects ? Comment garantir que l'instrumentation mise en place cible correctement les interférences potentielles à un point de jonction partagé ? Ces questions et les éléments de réponse issus de la discussion précédente sont maintenant discutés afin de définir plus précisément les objectifs de nos travaux.

3.5 Motivations et objectifs de la thèse

Dans cette section, à travers un ensemble de questions et les éléments de réponse issus des différentes discussions de ce chapitre, nous présentons tout d'abord les motivations de cette thèse ; ce qui nous permet ensuite de définir précisément notre problématique et d'énoncer brièvement nos contributions.

Comment instrumenter le code des aspects ?

Au regard des différents travaux autour des interférences, il est difficile d'énoncer clairement comment instrumenter le code tissé pour observer les interférences à l'exécution comme on peut le faire pour les architectures réflexives comme REFLEX ou les protocoles à méta-objet comme SMALLTALK [50]. On peut toutefois affirmer que le coeur de la détection d'interférences au niveau du code est certainement le mécanisme de séquencement des aspects. Toutefois, ce mécanisme ne présente pas la même flexibilité dans toutes les propositions, par exemple il ne permet la résolution que de manière restreinte dans ASPECTJ.

Actuellement, les approches comme celle de MAROT décrites dans [50] et discutées ci-dessus émettent l'idée que pour une propriété donnée un certain nombre de points d'observations sont nécessaires et leur instrumentation permet d'observer cette propriété. Se pose cependant la question concernant l'emplacement du code servant à l'instrumentation. Nous savons que l'instrumentation ne doit pas être implémenté dans les aspects. Nous savons également que le code d'instrumentation, tout comme le code de composition, est une préoccupation transversale. Apporter une réponse concrète à cette question est actuellement difficile car la majorité des langages de programmation orientée aspect sont des extensions de langages à objets qui permettent l'instrumentation du code des objets mais pas celui des aspects. Le code de la composition des aspects peut être encapsulé avec succès dans des aspects dédiés comme le *resolver* [74]. Dans ce contexte, peut-on imaginer la même chose pour le code d'instrumentation des aspects ? De plus, cela laisse supposer que tous les mécanismes d'instrumentation ciblant les différentes propriétés de non interférence font partie intégrante de l'approche d'instrumentation. Finalement, répondre à cette question revient à identifier tous les mécanismes nécessaires et suffisants pour instrumenter la détection des interférences que nous ciblons.

Comment garantir que l'instrumentation mise en place cible correctement les interférences potentielles à un point de jonction partagée ?

Comment garantir que l'instrumentation couvre les différents types de fautes potentielles aux points de jonction partagés. Les travaux sur la détection des interactions et les critères de couverture existants permettent de couvrir tous les points de jonction partagés et toutes les interactions de flot de données et de flot de contrôle à ces points de jonction. La couverture de points de jonctions peut donc être garantie.

La couverture des différentes fautes possibles à ces interactions requiert cependant d'avantage de rigueur. Nous pensons en effet que l'instrumentation doit être, d'une part, dirigée par les hypothèses faites lors de la réutilisation d'un aspect dans un contexte donné et, d'autre part, par le modèle de faute des interférences entre aspect : les interférences sont issues de la violation d'une hypothèse faite par l'intégrateur des aspects nécessaires à la réalisation d'une exigence non fonctionnelle.

Objectif .

En résumé, dans cette thèse nous défendons que l'assemblage d'applications résilientes constituées d'aspects implémentés à grain fin requiert que le code du protocole assemblé soit exempt de défaut. La validation d'un tel assemblage nécessite la mise en place d'une méthode et d'un outillage dédié à la validation des interactions entre aspects. Cette méthode de validation nécessite l'observation à l'exécution de propriétés de non-interférence et donc une instrumentation rigoureuse de ces propriétés. Notre problématique est donc la suivante :

- **Identifier les points d'observation et les mécanismes d'instrumentation nécessaires pour la détection d'interférences à l'exécution,**
- **outiller le processus d'instrumentation afin qu'un assemblage d'aspects puisse être validé.**

Dans le chapitre 1 nous avons présenté les différents travaux sur la validation des systèmes construit à l'aide de la programmation orientée aspect. Nous avons mis en évidence le manque de travaux autour de la validation des interactions entre aspect.

Dans ce chapitre nous avons proposé une synthèse des travaux autour de la détection à l'exécution des interférences entre aspects. Nous concluons que le cœur d'une approche de détection d'interférence à l'exécution passe par une instrumentation adéquate.

L'instrumentation est le fer de lance d'une approche de détection d'interférences à l'exécution. C'est donc ce point qui sera traité en premier lieu. Le chapitre 3 présente la méthode d'instrumentation visant la détection des interférences dans un assemblage d'aspects. Ce chapitre commence par présenter les points d'observation requis pour l'instrumentation. Ensuite, chaque section présente l'instrumentation nécessaire pour un type d'interférence, flot de données, flot de contrôle. Avant de conclure ce chapitre, une validation de l'approche d'instrumentation est présentée.

Chapitre 4

Évitement et détection des interférences entre aspects

Préambule

Ce chapitre présente le cœur du travail de cette thèse qui est l'implémentation d'une approche permettant la validation d'un assemblage d'aspects.

La validation d'un assemblage d'aspects nécessite la vérification d'un certain nombre de propriétés de non-interférence. L'instrumentation du code tissé est donc l'élément le plus important de notre approche de validation. Dans ce chapitre nous commençons par présenter les propriétés de non-interférence que nous devons vérifier pour garantir qu'un assemblage d'aspects réalise bien le comportement attendu par l'intégrateur. Nous présentons ensuite les motivations et les objectifs qui ont guidé nos choix techniques pour l'instrumentation. L'efficacité de notre approche est ensuite évaluée par une série d'expérimentations.

Sommaire

4.1 Introduction

Les systèmes informatiques évoluent dans des environnements dynamiques, leurs fonctionnalités, leurs modèles de faute et les ressources évoluent durant le cycle de vie.

En effet, une implémentation de ces préoccupations dans un langage basé sur un paradigme conventionnel ne permet pas d'obtenir une implémentation facilement reconfigurable de la tolérance aux fautes. La programmation orientée aspect permet d'obtenir une implémentation reconfigurable du logiciel de tolérance aux fautes. La capacité de reconfiguration est obtenue par une implémentation à grain fin d'aspects réutilisables. Ces aspects sont utilisés pour configurer la tolérance aux fautes des systèmes par composition et reconfigurer ces systèmes par ajout ou suppression de sous-ensemble d'aspects. Cependant, la composition des aspects implémentant une stratégie de tolérance aux fautes peut contenir des interférences.

Ces interférences sont décrites dans le chapitre précédent. Nous nous concentrons sur les interférences liées à l'application de plusieurs aspects à un même point de jonction. Dans ce chapitre, nous présentons une approche combinant à la fois l'évitement et la détection des interférences. L'évitement des interférences est réalisé par l'utilisation de AIRIA qui permet de contrôler la composition des aspects. La détection des interférences est réalisée grâce à l'utilisation d'assertions exécutables qui exploitent les points d'observations fournis par AIRIA.

Dans ce contexte, nous devons :

- d'abord définir pour chaque type d'interférence de flot de données et de flot de contrôle, les points d'observation à observer à l'exécution pour détecter une interférence.
- Puis pour l'ensemble de ces propriétés nous définissons une stratégie d'instrumentation permettant de les vérifier à l'exécution en utilisant des assertions exécutables.
- Enfin nous validons notre approche d'instrumentation. Notre méthode d'instrumentation s'inscrivant dans une approche générale, nous présentons l'organisation de cette approche et nous décrivons l'implémentation d'un prototype permettant de l'outiller.

Structure du chapitre

La section 4.2 définit les propriétés de non interférence à observer pour être capable de garantir qu'un assemblage d'aspects ne contient pas d'interférence. Dans cette même section nous présentons une étude comparative des langages de programmation orientés aspect présentés au chapitre 2 au regard de l'observabilité des propriétés de non interférence. La section 4.3 détaille le fonctionnement de AIRIA dans lequel les propriétés de non-interférence sont observables. Les sections 4.5.2 et 4.5 proposent une étude de faisabilité de l'instrumentation des propriétés de non interférence identifiées en utilisant AIRIA. La section 4.6 présente le processus de validation de notre approche d'instrumentation.

4.2 Observabilité des propriétés de non-interférence

Le premier objectif de cette section est de discuter des propriétés de non interférence à observer. Nous discutons également des caractéristiques d'observabilité des langages nécessaires pour détecter les problèmes d'interférence à l'aide d'assertions exécutables. Nous montrons que les opérateurs de composition fournis par un langage de programmation orienté aspects constitue la pierre angulaire de l'observabilité et donc de notre capacité à instrumenter notre code afin d'observer des propriétés de non interférence.

Dans la section 4.2.1, nous définissons un ensemble de propriétés non-interférence de base

et, pour chaque propriété, nous identifions l'ensemble des points d'observation dans le cycle de vie d'un greffon, permettant la vérification de cette propriété.

Nous discutons ensuite de l'observabilité de ces propriétés de non-interférence dans les langages de programmation orientés aspect.

4.2.1 Propriétés de non-interférence

Suivant la classification des interférences donnée dans le chapitre 2, nous avons identifié quatre propriétés représentant des interférences de type flot de données ou flot de contrôle, à un point de jonction jp_i et pour un greffon A_i. Ces propriétés peuvent être décrites comme suit :

- détection des changements avant (*Change Before* CB) : Il n'y a pas de modification d'une variable v du code de base utilisée par A_i, par un autre aspect exécuté avant A_i.
- détection des changements après (*Change After* CA) : Après l'exécution de A_i, il n'y a pas de changement d'une variable v utilisée par A_i, par un autre aspect suivant A_i.
- détection des invalidations avant (*Invalidation Before* IB) : Tous les aspects exécutés avant A_i, n'empêchent pas l'exécution de A_i.
- détection des invalidations après (*Invalidation After* IA) : Le point de jonction jp_i doit toujours être exécuté après A_i.

Notons que ces propriétés peuvent être vérifiées pour les parties avant et après des greffons. Cependant, la détection d'une interférence de type IA n'a pas de sens sur la partie après des greffons d'une chaine d'aspects.

La figure 4.1 représente le cycle de vie d'un greffon de type before A_i appliqué avant un point de jonction jp_i, où plusieurs aspects sont attachés. Depuis l'exécution du niveau de base, l' événement α déclenche une série de greffons avant A_i. Lorsque A_i se termine, la transition γ déclenche l'exécution d'un ensemble de greffons après A_i. La transition δ représente le déclenchement de l'exécution du point de jonction dans le code de base, et la transition δ' l'exécution de la séquence de greffons après l'exécution du point de jonction au niveau de base. Nous considérons aussi deux transitions, notées ϕ qui représentent deux comportements possibles lorsque le flot de contrôle revient à son niveau de base :

- l'ensemble des aspects exécutés avant A_i empêchent l'exécution de A_i et du point de jonction ;
- l'ensemble des aspects exécutés après A_i empêchent l'exécution du point de jonction ;

Cet automate générique représente les points de contrôle qui doivent être observés pour la vérification des propriétés mentionnées ci-dessus dans la partie avant d'une chaine de greffons. Il est à noter que nous supposons que la variable v est n'importe quelle variable accessible depuis A_i.

- La détection de CB commence lorsque la chaîne d'aspects est déclenchée (la transition α est observée et v est stockée). La vérification que la variable v n'est pas modifiée s'effectue en β, sauf si ϕ est observée au lieu de β.
- La détection de CA est déclenchée en γ et la vérification que la variable v n'est pas modifiée se fait à δ, à moins que l'on n'observe ϕ.
- La détection de IB commence avec α, et β doit être observée. Si ϕ est observée au lieu de β alors la vérification se termine et une erreur est générée.
- La détection de la IA commence lorsque γ est observée, le comportement correct est déterminé par l'observation de δ. Si ϕ est observée au lieu de δ alors la vérification se

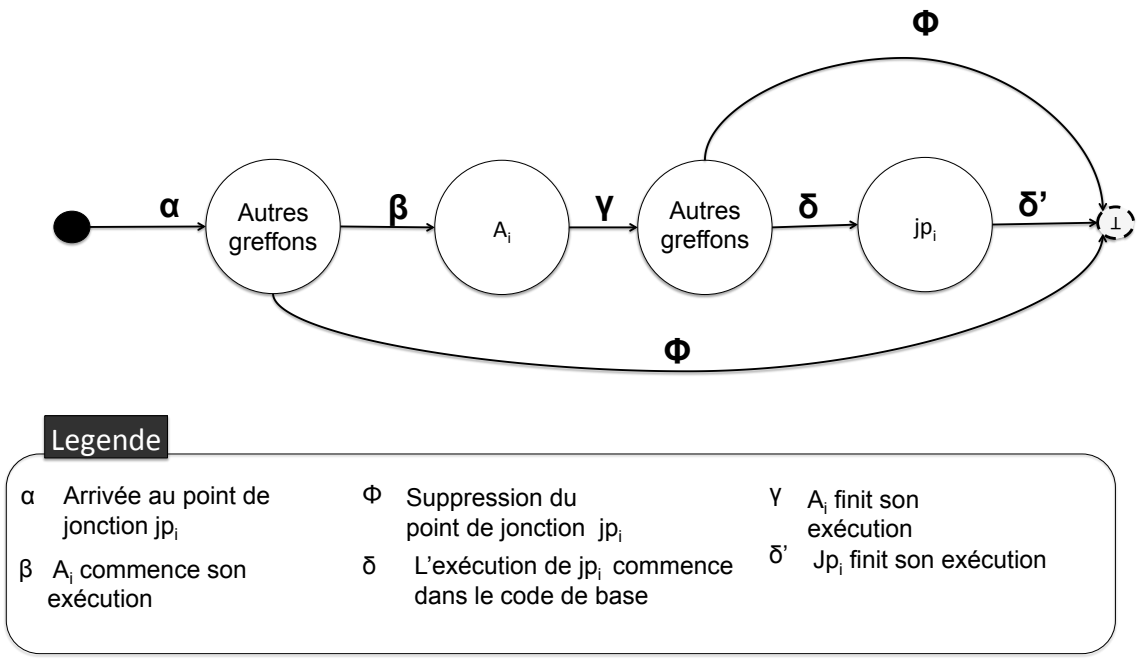


FIGURE 4.1 – Cycle de vie d'un greffon de type *before*

termine et une erreur est générée.

Le tableau 4.1 identifie les observations nécessaires pour la vérification des quatre propriétés de non-interférence portant sur la partie avant de A_i.

TABLE 4.1 – Point d'observation requis pour la vérification de propriétés de non interférence

Propriété \ Transition	α	β	γ	δ	ϕ
CB	X	X			X
CA			X	X	X
IB	X	X			X
IA			X	X	X

De façon similaire on peut faire la même analyse sur la partie après des greffons d'une chaîne d'aspects.

Ces propriétés de non interférence ne sont pas observables dans tous les langages de programmation orientés aspect. De toute évidence, lorsque des événements définis ci-dessus ne peuvent pas être observés, cela a un impact fort sur la vérification des propriétés de non-interférence.

Dans ASPECTJ, β et γ peuvent être observés, mais il n'y a pas de point explicite dans le flot de contrôle qui représente le début ou la fin d'une chaîne d'aspects. En conséquence, ni α ni ϕ et δ ne sont observables, aucune des propriétés précédentes ne peut être observée.

Dans AIRIA ou dans REFLEX les différentes transitions sont observables ce qui permet de les instrumenter. La vérification des propriétés de non-interférence que nous considérons nécessite un accès explicite à certains points dans le flot de contrôle de la chaîne d'aspects, en particulier la transition entre le code de base et de la chaîne d'aspects.

Nous avons montré par un graphe simple représentant une chaîne de greffons de type "*avant*", que tous les points d'observation peuvent être facilement identifiés. La définition des assertions exécutables peut être directement dérivée de l'expression des propriétés du flot de données et de flot de contrôle attendues. Nous avons constaté que ces propriétés ne sont pas observables dans tous les langages de programmation orientée aspect, mais que AIRIA et REFLEX permettent leur observation.

Finalement AIRIA présente de nombreux avantages : la capacité de composer les greffons de manière flexible (discutée au chapitre 2) et l'observabilité nécessaire pour l'observation des propriétés de non interférence à l'exécution.

4.3 Composition des greffons à l'aide d'AIRIA

Les resolvers définissent la politique de séquencement au niveau des greffons et non pas au niveau des aspects. De plus, le compilateur d'AIRIA signale une erreur pour chaque interaction entre greffon non résolu ce qui répond à notre besoin d'appliquer une résolution explicite et systématique des interférences potentielles. Pour la mise en œuvre de la détection d'interférences à base d'assertion, il n'existe pas d'extension ASPECTJ spécifique.

La contribution principale de cette thèse est de démontrer que l'extension AIRIA - ciblant les questions de séquencement - est également suffisante pour rendre la détection d'interférences possible. Cela est dû au fait que la construction de resolver ajoute davantage d'observabilité à ASPECTJ. Les points d'observation manquants deviennent disponibles, ce qui nous permet de mettre en œuvre la détection des interférences à l'aide d'assertions.

Un resolver est une sorte de greffon *around* utilisé pour contrôler la composition des greffons à des points de jonction partagés, c'est-à-dire un greffon pour composer des greffons. Un exemple est donné dans le listing 4.1. Les caractéristiques du resolver sont les suivantes. Un resolver est défini par un mot-clé suivi d'un nom de resolver et d'une liste de paramètres (*sender* et une liste vide dans le listing 4.1, ligne 2). De plus, le resolver précise quand il contrôle la composition (par l'intermédiaire de la clause *and* / *or*) et comment le séquencement des greffons est réalisé (par le biais de la clause *proceed*). La clause *and* / *or* du resolver fonctionne comme une expression de coupe, sauf qu'elle spécifie une liste de greffons potentiellement en conflit (listing 4.1, ligne 3). Lorsque les greffons spécifiés dans la clause *and* / *or* sont tissés à un point de jonction partagé, le resolver est tissé en *premier* à ce point de jonction, afin de gérer le séquencement des greffons. En d'autres termes, un resolver est un greffon possédant une priorité plus élevée que les greffons qu'il gère.

Un resolver ayant également le statut de greffon, il peut être manipulé par un autre resolver, ce qui donne des hiérarchies de resolvers. Il est important de noter que tous les greffons dans AIRIA sont des greffons de type *"autour"*. Ainsi, pour mettre en œuvre un greffon de type "*avant*", nous utilisons simplement un greffon de type "*autour*" muni d'une partie "*après*" vide. De même, nous mettons en œuvre un greffon "*après*" par un greffon de type "*autour*" muni d'une partie "*avant*" vide. Ces greffons sont identiques à ceux de ASPECTJ, mis à part le fait

qu'ils possèdent un nom unique. Par exemple, dans le listing 4.1, la ligne 3 se réfère à un greffon nommé *logMsg* appartenant à l'aspect *Alog*.

Un resolver spécifie une précédence à grain fin entre les greffons. Dans le listing 4.1, le resolver *sender* est appliqué à chaque point de jonction où les greffons *ALog.logMsg, ACrypt.encrypt, AAuth.auth* sont tissés (comme spécifié dans la clause *and*). La liste entre crochets détermine l'ordre d'exécution de la clause *proceed* (Listing 4.1, ligne 4). Dans notre exemple, *AAuth.auth* est appliqué en premier, puis *Alog.logMsg* est appliqué et enfin *ACrypt.encrypt* chiffre le message avant de l'envoyer.

```
1 aspect LogEncryptAuth {
2 void resolver sender():
3 and(ALog.logMsg, ACrypt.encrypt, AAuth.auth){
4 [AAuth.auth, ALog.logMsg, ACrypt.encrypt].proceed();}}
```

Listing 4.1 – Utilisation de resolver dans notre exemple

La liste entre crochets peut être considérée comme représentant la chaîne d'exécution des greffons sur la figure 4.1. Cette représentation s'avère pratique pour la détection d'interférences, parce que toutes les transitions dans le cycle de vie sont désormais rendues explicites. Nous pouvons insérer des greffons supplémentaires dans le séquencement pour la capture des variables d'état, l'observation des modifications de leur valeur, la capture des événements, lever ou abaisser des drapeaux, etc. Notez cependant que le listing 4.1 représente un cas simple où tous les greffons sont gérés par un seul resolver. Dans le cas général, le séquencement peut être défini en utilisant des resolvers de resolvers, comme l'illustrent les listings 4.2 et 4.3. Dans cet exemple, nous avons un resolver pour la journalisation et le chiffrement (Listing 4.3). Il est invoqué par un resolver racine (Listing 4.2) qui gère aussi le chiffrement. L'ordre qui en résulte pour les trois greffons est le même que dans le listing 4.1, même si cela n'est pas intuitif.

En effet le listing 4.2 indique que pour les greffons gérés par le resolver *LogEncryptAuth.sender* l'ordre de précédence est celui figurant dans la clause *proceed* de ce dernier, c'est-à-dire *LogCrypt. logCrypt*< *AAuth.auth* < *ALog.logMsg* comme l'indique la partie supérieure de l'arbre de resolvers décrit dans la figure 4.2.

L'ordre de précédence défini par le greffon *LogCrypt.logCrypt* est *ALog.logMsg* < *ACrypt.encrypt* ce qui correspond à la partie inférieure de l'arbre décrit dans la figure figure 4.2. Finalement l'ordre de précédence définie par l'arbre figure dans la partie droite de la figure 4.2 et correspond à l'ordre total suivant *LogEncryptAuth.sender* < *LogCrypt. logCrypt* < *AAuth.auth* < *ALog.logMsg* < *ACrypt.encrypt*.

```
1 aspect LogEncryptAuth{
2 void resolver sender():
3 and(LogCrypt.logCrypt, AAuth.auth, ALog.logMsg){
4 [LogCrypt.logCrypt, AAuth.auth, ALog.logMsg].proceed();}}
```

Listing 4.2 – Utilisation de resolver de resolvers (partie 1 : resolver racine)

Le compilateur d'AIRIA vérifie que, quel que soit le point de jonction partagé, un resolver de racine unique gère les conflits à ce point de jonction. Un ordre d'exécution total doit être obtenu à partir de l'arbre de resolvers à partir de la racine (voir [74] pour plus de détails). On peut ainsi contrôler précisément l'ordre des greffons. L'insertion des greffons d'instrumentation doit être adaptée à de telles structures arborescentes.

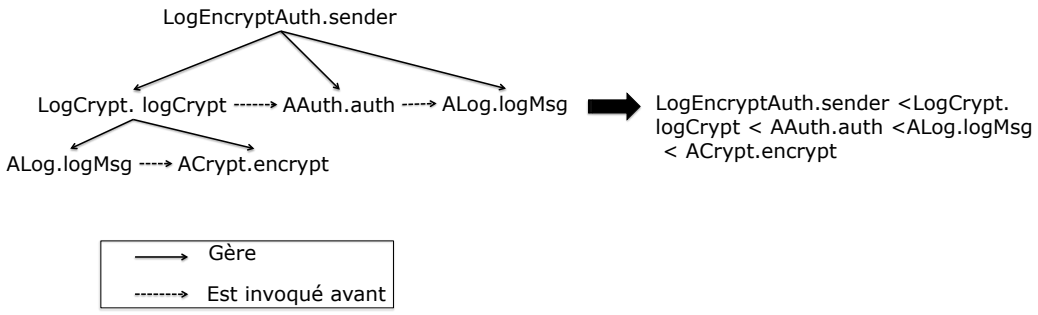


FIGURE 4.2 – Ordre total induit par les listing 4.2 et 4.3

```
1 aspect LogCrypt {
2 void resolver logCrypt():
3 and(ALog.logMsg, ACrypt.encrypt){
4 [ALog.logMsg, ACrypt.encrypt].proceed();}}
```

Listing 4.3 – Utilisation de resolver de resolvers (partie 2 : resolver auxiliaire)

4.4 Détection des interférences

Cette section détaille comment cette spécification est utilisée pour séquencer les aspects et instrumenter le séquencement. Nous commençons par poser les hypothèses restrictives nécessaires au bon fonctionnement de notre approche d'instrumentation. Puis nous détaillons l'instrumentation de la chaine d'aspects.

Notre approche suppose que les aspects de vérification ne sont pas sélectionnés comme points de jonction par d'autres méta-aspects. Les seuls méta-aspects sont les resolvers avec éventuellement des resolvers de resolvers.

Tous les problèmes de composition d'aspects sont résolus par les resolvers, avec éventuellement des resolvers de resolvers. Nous expliquons d'abord comment les transitions α , β,... de la figure 4.3 peuvent être rendues visibles, étant donné un arbre arbitraire de resolvers. Nous présentons ensuite les greffons d'instrumentation à placer au niveau des transitions appropriées pour détecter les interférences de flot de données et de flot de contrôle. Un arbre de resolver induit un ordre total entre les entités exécutées qui peut être déterminé à la compilation. La figure 4.3 illustre le cycle de vie d'un greffon de type *around* A_i pour un ordre arbitraire. Elle étend la figure 4.1 en tenant compte des parties avant et après des greffons. Après le point de jonction, les greffons en interaction sont dépilés dans le sens inverse de la précédence. Les transitions α, ..., δ ont comme contreparties δ', ..., α'.

Le principe de l'instrumentation est le suivant. Une assertion est décomposée en un ensemble d'actions de surveillance destinées à être exécutées à des transitions spécifiques. Les actions de surveillance sont assurées par des greffons, qui viennent s'ajouter aux greffons en conflit déjà gérés par l'arbre de resolvers. Nous avons donc besoin d'identifier des points spécifiques ou *placeholders* dans le code de resolvers (en particulier, dans les clauses *proceed*) de sorte que si un greffon de surveillance est inséré à cet endroit, alors la logique de surveillance sera exécutée à la transition souhaitée. L'approche doit fonctionner pour des arbres de resolvers construits de manière arbitraire. Nous identifions d'abord les *placeholders* pour exposer

les transitions de la figure 4.3. Nous présentons ensuite les greffons de surveillance qui doivent être exécutés au niveau des transitions appropriées pour la détection des interférences de flot de données et des interférences de flot de contrôles.

4.4.1 *Placeholders* pour exposer les transitions au moment de l'exécution

Nous utilisons l'exemple précédent avec des resolvers de resolvers (listings 4.2 et 4.3) pour illustrer les *placeholders*, ce qui aboutit aux listings 4.4 et 4.5. Les symboles ℓ_α, ℓ_β ... désignent les *placeholders* pour insérer des greffons de surveillance qui exposent les transitions désirées. Plus précisément, chaque *placeholders* permet à une paire de transitions "*avant*" / "*après*" d'être exposée. Par exemple, un greffon de type "*autour*" inséré à ℓ_α expose à la fois α et α' : sa partie "*avant*" est exécutée à α et sa partie "*après*" à α'. Du code supplémentaire est également nécessaire pour détecter les transitions ϕ, quand un greffon soulève une exception ou n'appelle pas l'instruction *proceed* (voir le Listing 4.4 , lignes 7 et 9). Comme le flot de contrôle sort de la chaîne de greffons, ϕ ne peut pas être exposée par des greffons insérés. Les greffons insérés peuvent toutefois lever des drapeaux pour indiquer si un point spécifique est atteint. Nous pouvons ensuite exposer ϕ au niveau du resolver racine, par du code dédié à l'inspection de ces drapeaux.

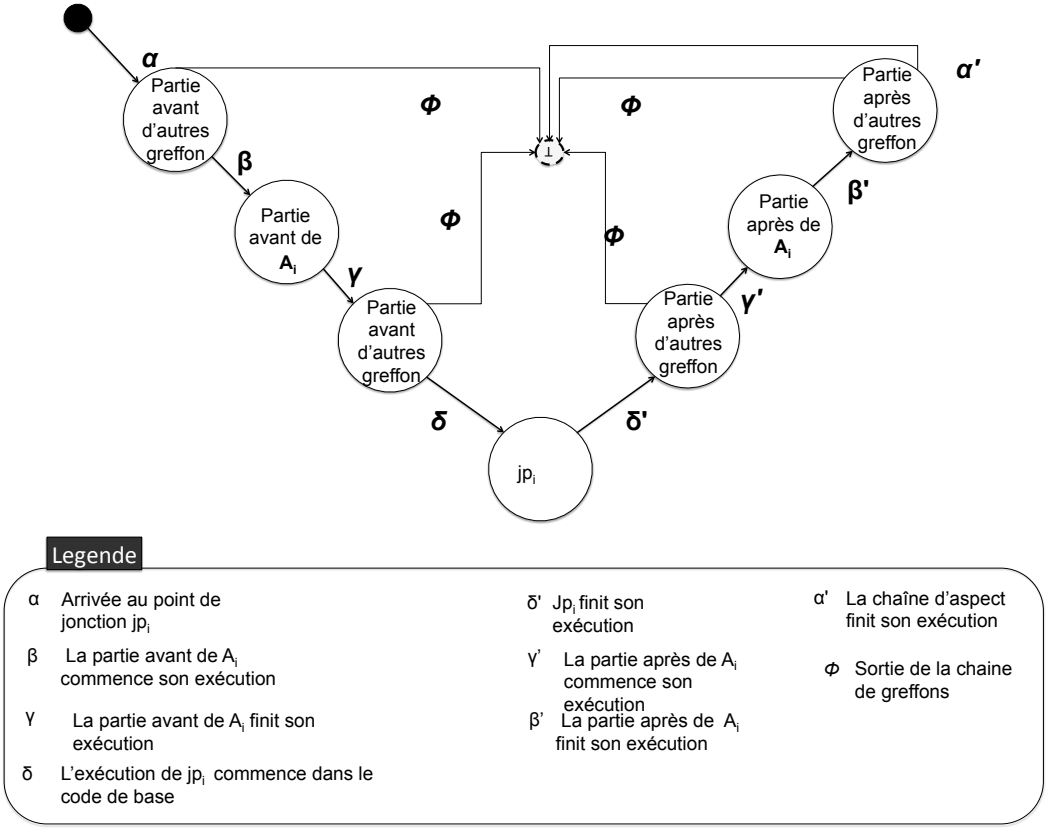


FIGURE 4.3 – Cycle de vie d'un greffon de type around

Comme expliqué plus haut, le passage du contrôle du code de base au premier greffon est exposé par l'insertion d'un greffon de surveillance au *placeholder* ℓ_α du resolver racine (listing 4.4, ligne 5). Il posséde la plus haute précédence par rapport à toutes les entités manipulées. Sa partie "*avant*" est exécuté en premier et sa partie "*après*" la dernière.

Le début et la fin d'un greffon A_i peuvent être exposés en insérant des greffons immédiatement "*avant*" et "*après*" ces dernières, à tous les endroits où il est invoqué dans l'arbre de resolvers. Les listings 4.4 et 4.5 illustrent les emplacements ℓ_β et ℓ_γ pour le greffon *ALog.logMsg*. Selon l'algorithme de séquencement d'AIRIA, les greffons de surveillance insérés satisfont : $.. < A_{i-1} < \ell_\beta < A_i < \ell_\gamma < A_{i+1} < ...$

En appliquant cette logique à la partie "*avant*" de A_i, nous enveloppons l'exécution de la partie "*avant*" (transitions β et γ). De même, nous pouvons appliquer la même logique sur la partie "*après*" et exposer l'exécution de la partie "*après*" de A_i (transitions γ' et β').

Enfin, les transitions δ et δ' sont exposées par un greffon de surveillance ayant la plus faible priorité dans la chaîne : sa partie "*avant*" est la dernière à être exécutée avant le point de jonction, sa partie "*après*" est la première après le point de jonction. Ce greffon doit être inséré dans les clauses $proceed$ de tous les resolvers de l'arbre de resolver (voir les points ℓ_δ dans les listings 4.4 et 4.5), y compris les resolvers qui ne gèrent pas A_i. De cette façon, nous nous assurons que toutes les entités exécutées dans l'arbre sont prioritaires sur les greffons à la position ℓ_δ.

Nous venons d'expliquer comment exposer les transitions dans une chaîne d'exécution de greffons. En plaçant la logique appropriée aux endroits appropriés, nous sommes maintenant en mesure de mettre en œuvre la détection d'interférences.

```
1 aspect LogEncryptAuth{
2 void resolver sender():
3 and(ℓα, ℓβ, ℓγ, ℓδ, LogCrypt.logCrypt, AAuth.auth, ALog.logMsg){
4 throws RuntimeException{
5 try {[ℓα, LogCrypt.logCrypt, AAuth.auth, ℓβ, ALog.logMsg, ℓγ, ℓδ].proceed
      ();}
6 catch (Exception e) {
7 // code to expose a φ transition due to an exception raised
8 }
9 // code to expose a φ transition due to not calling a proceed }}
```

Listing 4.4 – Instrumentation du resolver racine

4.4.2 Détection des interferences de flot de données

La détection des interférences CB et CA utilise deux aspects. *AStorer* fournit un greffon qui stocke les valeurs des variables sélectionnées à un certain point de la chaîne d'aspects. *AChecker* vérifie que les valeurs sont inchangées à un point ultérieur de la chaîne. Dans cette section, pour des raisons de simplicité, nous centrerons la discussion sur la partie avant de ces greffons. Il est simple d'appliquer une approche similaire à la vérification des propriétés attachées à la partie "*après*" des greffons.

```
1 aspect LogCrypt {
2 void resolver logCrypt():
3 and(ℓβ, ℓγ, ℓδ, ALog.logMsg, ACrypt.encrypt){
4 [ℓβ, ALog.logMsg, ℓγ, ACrypt.encrypt, ℓδ].proceed();}}
```

Listing 4.5 – Instrumentation du resolver auxiliare

4.4.2.1 Détection des interférences "changement avant" (CB)

AStorer.store est placé en α et *AChecker.check* en β. La chaîne d'exécution induite est montrée dans la figure 4.4. L'aspect *AStorer* possède une structure de données *variablesToStore*, en spécifiant les variables $(v_1, ...v_n)$ à stocker pour la détection CB. Les tuples de cette structure de données sont composés d'un ensemble de paires *(description du point de jonction, variables)*. La description du point de jonction est l'identifiant (techniquement, le nom complet) du point de jonction, et *les variables* sont des variables d'état de ce point de jonction. *AStorer* analyse *variablesToStore* afin de connaître les variables à stocker ❶. Puis, à travers l'interface réflexive de JAVA [2] , on accède à la structure des points de jonction pour récupérer la valeur courante des variables ❷.

Les valeurs sont stockées dans une structure de données appelée *storedVariables* ❸. Cette structure est partagée par les aspects *AStorer* et *AChecker*. Le greffon *AChecker.check* est inséré juste avant l'exécution du greffon cible A_i. La liste des variables à contrôler est récupérée à partir de sa structure de données *variablesToCheck* ❹. Leurs valeurs sont lues via la structure des points de jonction ❺. Enfin, *AChecker* compare ces valeurs à celles de *storedVariables* ❻. Si elles sont différentes, une interférence est signalée.

4.4.2.2 Détection des interférences "changements après" (CA)

Dans le cas d'une interférence CA, nous devons observer les changements de valeur de certaines variables après l'exécution du greffon cible, et avant l'exécution du point de jonction. Nous appliquons la même démarche que précédemment, sauf que le *storer* est placé en γ et le *checker* en δ.

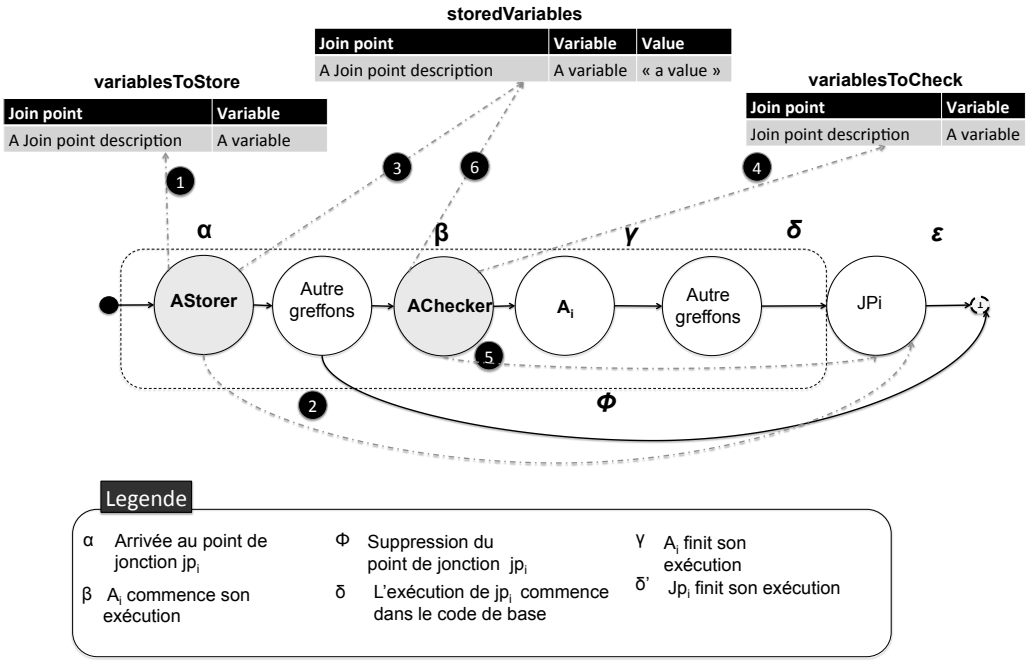


FIGURE 4.4 – Instrumentation pour la détection d'interférences de type CB

2. http ://java.sun.com/developer/technicalArticles/ALT/Reflection/

4.4.3 Détection de interférences de flot de contrôle

La détection des interférences IB et IA utilise des drapeaux pour représenter l'occurrence d'événements attendus dans la chaîne de greffons. La valeur initiale d'un drapeau indique que l'événement ne s'est pas encore produit. Lorsque l'événement se produit, la valeur du drapeau est changée. À la fin de la chaîne des greffons, le resolver racine est en mesure de déterminer si un événement attendu s'est produit ou non.

4.4.3.1 Détection des interférences "invalidation avant" (IB)

La détection de l'IB nous oblige à vérifier si *le greffon cible* est toujours exécuté après α. Nous utilisons deux greffons insérés dans la chaîne d'exécution pour gérer des drapeaux, ainsi que des fragments de code placés dans le resolver racine pour exposer une transition ϕ. Le premier greffon, *EventInitializer.init*, est placé en α. Le second, *AspectObserver.set* est placé en β. La figure 4.5 montre la chaîne d'exécution.

EventInitializer.init possède une structure de données *eventToObserve* contenant l'ensemble des événements qui doivent être observés ❶. Il l'utilise pour initialiser une structure de données partagée appelée *observedEvent* ❷. La structure est initialisée avec des paires d'évènements et de drapeaux. Chaque drapeau est initialement fixé à false (0). *AspectObserver.set* est placé à β, juste avant l'exécution du *greffon cible* ❸. Il bascule le drapeau correspondant à *observedEvent* à *true* ❹. Le contrôle final est effectué dans le corps du resolver racine. Le code de contrôle est à la fois dans le gestionnaire d'exceptions (levée d'une exception IB) et dans le code exécuté après la chaîne de greffons (IB due à l'omission d'une instruction $proceed$).

Le listing 4.6 montre un exemple où le greffon de journalisation doit toujours être exécuté. Le resolver racine fournit un ordre incorrect de l'exécution (ligne 4). Chaque fois que l'authentification échoue, une interférence de type IB est détectée en observant un drapeau *false* pour l'exécution de *ALog.logMsg*.

4.4.3.2 Détection des interférences "invalidation après" (IA)

La détection d'interférence de type IA nous oblige à vérifier qu'aucun greffon exécuté après A_i n'interrompt la chaîne d'exécution. *EventInitializer.init* est désormais exécuté à la transition γ, et *AspectObserver.set* en δ. Comme précédemment, le resolver racine réalise la vérification finale.

```
1 aspect LogEncryptAuth {
2 void resolver senderCA():
3 and(EventInitializer.init, AspectObserver.set, ACrypt.encrypt, ALog.
      logMsg, AAuth.auth) throws RuntimeException{
4 try { [ EventInitializer.init, AAuth.auth, AspectObserver.set, ALog.
      logMsg, ACrypt.encrypt ].proceed();
5  } catch (Exception e) { // check whether all flags are set, report an
       IB detection otherwise }
6 // check whether all flags are set, report an IB detection otherwise }}
```

Listing 4.6 – Exemple de détection d'une interférence de type IB

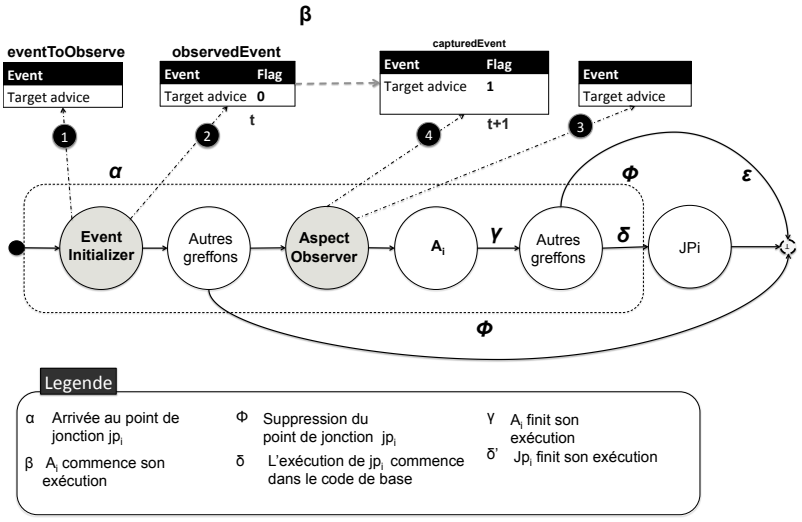


FIGURE 4.5 – Instrumentation pour la détection des interférences de type IB

4.5 Etude de faisabilité

Dans la section précédente, nous avons présenté des propriétés de non-interférence à vérifier pour un assemblage d'aspects donné. Cela nous permet de garantir qu'une configuration d'aspects ne contient pas les comportements non désirés correspondants. Nous avons également présenté une méthode d'instrumentation permettant de vérifier ces propriétés lors de l'exécution. Nous avons implémenté cette méthode dans un outil appelé AIFACTORY pour ***A**dvice **I**nstrumentation Factory*. *AIFactory* réalise l'instrumentation de hiérarchies de resolvers afin de valider la composition des aspects. Nous devons maintenant valider le fonctionnement de cet outil. L'objectif de cette section est de montrer que l'instrumentation produite par AIFACTORY permet de détecter toutes les interférences liées à l'application de greffons multiples autour d'un même point de jonction.

La figure 4.6 donne une vision globale du processus de validation que nous suivons.

À partir des objectifs ❶ de test, nous fournissons en entrée des données de test ❷. Puis à partir de ces données, une instrumentation d'un système simplifié est produite ❸ et exécutée ❹. Nous enregistrons la trace de l'exécution de ce système et le résultat produit par le biais de notre instrumentation ❺. Dans cette trace, un certain nombre d'interférences doivent apparaître. Si toutes les interférences à observer sont observées dans la trace et seulement celles-ci, alors le test réussit (pass) sinon il échoue (faux positif, faux négatif)❻. Ce résultat, aussi appelé verdict, est rendu par un programme appelé l'oracle de test. Dans les sections qui suivent nous détaillons :

- les différents objectifs de test pour ce composant (section 4.5.1),
- les données d'entrée générées et la méthode de génération de ces données (section 4.5.2)
- ainsi que notre oracle (section 4.5.3).

4.5.1 Les objectifs de test

Nous visons deux objectifs de test. Le premier est relatif à l'applicabilité de notre approche sur les différents types de greffons existants. Même si dans AIRIA tous les greffons sont de type "*autour*" il est possible de simuler des greffons de type "*avant*" et de type "*après*" en implémen-

tant un greffon "*autour*" sans partie "*avant*" ou "*après*". De ce fait, d'une manière générale, les greffons sont composés autour d'un point de jonction. Il nous faut donc évaluer l'efficacité de notre approche pour des propriétés portant aussi bien sur les parties "*avant*" et "*après*" des greffons, aux points de jonction partagés.

Par ailleurs sur les parties "*avant*" et "*après*" d'un point de jonction partagé, plusieurs interférences peuvent se produire. Par exemple une interférence de type CA et une interférence de type IB peuvent se produire sur la partie "*avant*" d'un point de jonction. Aussi, notre deuxième objectif de test consiste à vérifier que les aspects visant la détection des différents types d'interférence (CA, CB, IA, IB) peuvent être composés. Un aspect d'instrumentation ne doit avoir aucun effet secondaire sur le flot de contrôle et le flot de données des autres aspects ciblant la détection des différents types d'interférences. Par exemple, un *AStorer* (ciblant la détection de CB) et un *EventInitializer* (ciblant de détection de l'IA) doivent pouvoir être placés à α dans n'importe quel ordre.

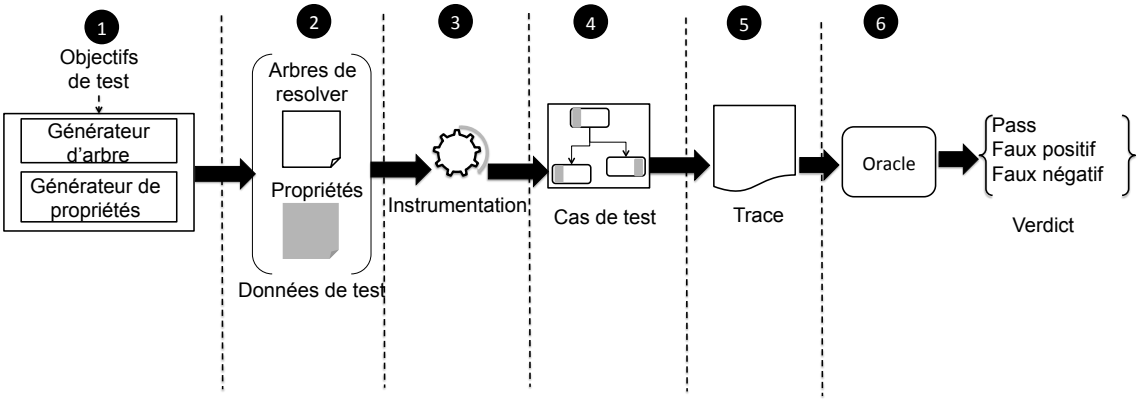


FIGURE 4.6 – Organisation du processus de validation de l'approche d'instrumentation

Pour valider ces deux objectifs de test, nous avons réalisé plusieurs expérimentations. Nous détaillons à présent comme ont été organisées ces expérimentations d'une manière générale. Plus loin dans ce manuscrit, nous présentons les différents paramètres des expérimentations que nous avons réalisées et les résultats. Afin de donner une idée générale du déroulement d'une expérimentation nous considérons cinq greffons A_1, A_2, A_3, A_4 et A_5 :

- A_1 et A_2 sont des greffons neutres du point de vue des interférences, effectuant uniquement des accès en lecture seule sur une variable v du code base ;
- A_3 et A_4 écrivent v et peuvent donc induire des interférences de type flot de données ;
- A_5 peut interrompre la chaîne d'exécution des greffons.

Une expérience est caractérisée par trois paramètres d'entrée :

- Une arborescence de resolvers, correspondant à la détermination de l'ordre d'exécution des greffons.
- Un ensemble de propriétés de non-interférence à vérifier (CA, CB, IA, IB).
- La version sélectionnée de A_5 (entre 4 versions possibles : pas d'interruption, levée d'exception dans la partie "*avant*", levée d'exception dans la partie "*après*", pas d'appel à $proceed$).

Nous présente dans la section suivante la génération des arbres de resolver qui constituent la première donnée d'entrée de nos cas de test.

4.5.2 Les données d'entrée

Cette section détaille la méthode de génération de données de test utilisée pour la validation de notre méthode d'instrumentation. Le programme de génération de données a été conçu et implémenté par HÉLÈNE WAESELYNCK.

Les données de test sont composées de hiérarchies de resolvers et de propriétés à vérifier sur ces hiérarchies. Le but, est de vérifier que pour une configuration des greffons où une interférence se produit au regard d'une propriété donnée, celle-ci est détectée. Les hiérarchies et les propriétés sont générées de manière aléatoire. Nous détaillons ici la génération de hiérarchies de resolvers.

La génération des hiérarchies de resolvers commence à partir d'un cas de base où tous les greffons sont gérés par un resolver racine unique. La figure 4.7 montre un cas exemple de cas de base avec quatre greffons A_1, A_2, A_3, A_4. La représentation sous forme d'arborescence visualise deux types de contraintes d'ordre :

- (1) entre un resolver et les entités qu'il gère
- (2) entre des entités invoquées (*proceeded*) par le resolver.

De l'ensemble des contraintes, il résulte un ordre d'exécution total du resolver R_0 et des quatre greffons A_1, A_2, A_3, A_4. Par exemple cet ordre d'exécution est visible au-dessus de la figure 4.7.

L'algorithme de génération utilise des structures de données à la fois pour la représentation de l'arbre et de l'ordre total des entités. Le principe est de construire une arborescence de façon inductive à partir du cas de base par l'insertion de resolvers auxiliaires dans l'arbre. Chaque insertion produit un ordre total qui préserve toutes les contraintes d'ordre vérifiées avant l'insertion.

Les figures 4.7 ,4.8 ,4.9 illustrent un processus de génération avec deux étapes d'insertion.

L'opération d'insertion ci-dessus est dédiée aux resolvers qui contribuent à la détermination d'un ordre total d'exécution. Notre algorithme de génération donne aussi la possibilité d'insérer de façon aléatoire des resolvers inutiles qui sont gérés, mais pas exécutés (*proceeded*), par leur parent. L'opération d'insertion est simple, car il n'est pas nécessaire d'assurer la cohérence avec un ordre total. Un exemple est donné dans la figure 4.10. Comme le resolver R_3 n'est pas appelé, il n'apparaît pas dans l'ordre total de la trace d'exécution. En particulier, la contrainte $R_3 < A_2 < A_1$ de la sous-arborescence insérée est tout simplement ignorée.

Avant de commencer à automatiser l'instrumentation avec des assertions, des expériences préliminaires ont confirmé que tous les cas sont compilés sans erreurs et que leur exécution produit l'ordre prévu des greffons.

4.5.3 L'oracle de test

Notre objectif est de tester AIFACTORY qui est notre logiciel d'instrumentation. Les sections précédentes ont présenté respectivement les objectifs de test et la génération des données de test.

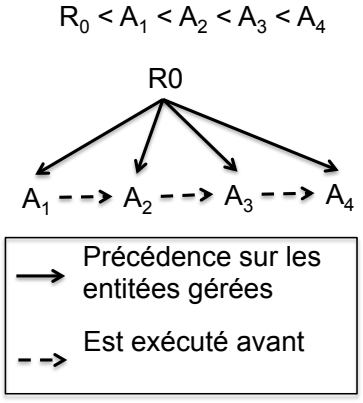


FIGURE 4.7 – Cas de base

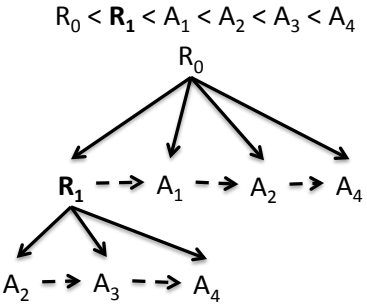


FIGURE 4.8 – Insertion de **R1**

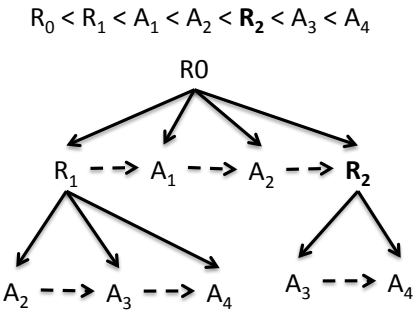


FIGURE 4.9 – Insertion de R2

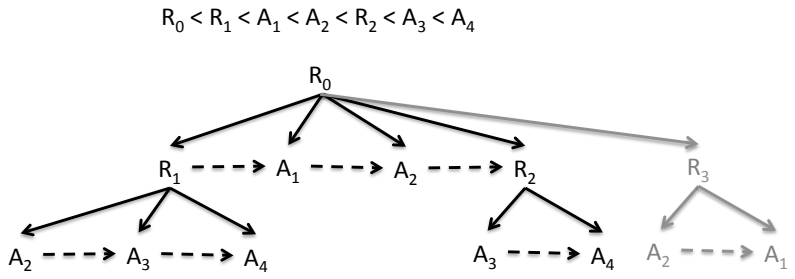


FIGURE 4.10 – Insertion d'un resolver qui n'est pas exécuté

Nous présentons maintenant l'oracle de test. L'oracle de test est un programme qui prend en entrée trois données :

1. un arbre de resolvers obtenu par la méthode de génération décrite ci-dessus,
2. un ensemble de propriétés de non-interférence à vérifier,
3. une trace d'exécution de l'arbre de resolver (1) instrumenté en fonction des propriétés à vérifier (2).

Le rôle de l'oracle dans le cadre de notre évaluation consiste à fournir un verdict indiquant si oui ou non les interférences qui devaient être observées sont bien observées dans la trace d'exécution.

Pour pouvoir prononcer ce verdict l'oracle s'appuie sur un modèle. Ce modèle doit dans un premier temps être initialisé, puis plusieurs traitements sont effectués sur ce dernier et enfin le verdict est calculé. Nous détaillons dans ce qui suit :

– le modèle et son initialisation
– les différents traitements effectués sur ce dernier
– le calcul du verdict

```
1 WN1AbN2
2 R0.r0 = ( A1, A2, A3, A4, A5) = proceed = (A3, A1, A5, A4 ,A2 )
3 FIN WN1AbN2
```

Listing 4.7 – Exemple d'arbre de resolver généré

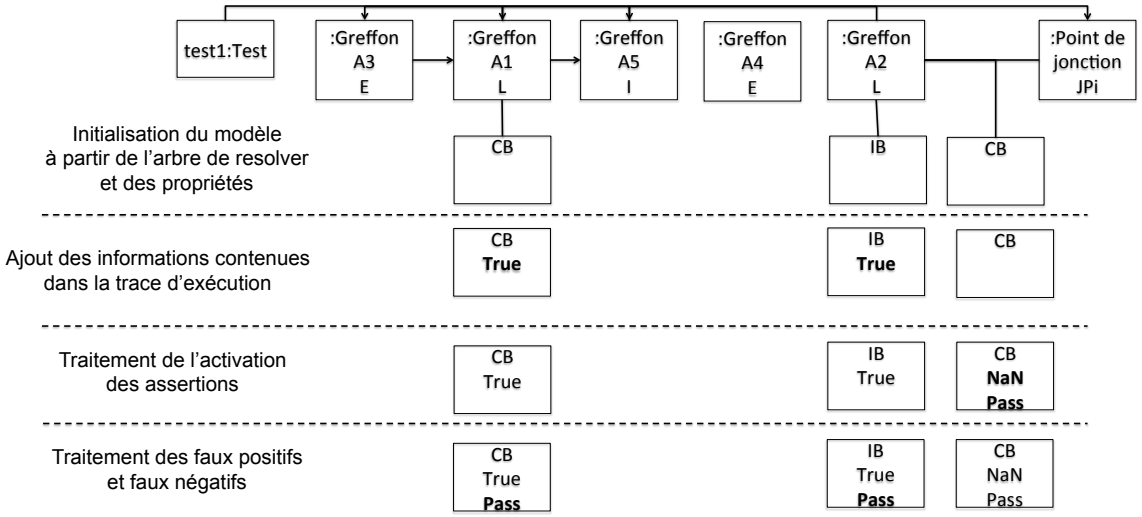


FIGURE 4.11 – Principe de l'analyse d'un cas de test instance du méta-modèle cas de test de la figure 4.12

4.5.3.1 Initialisation du modèle des propriétés

```
1 BEFOREO
2 before:  CB(A1), IB(A2), CB(A2)
3 FIN BEFOREO
```

Listing 4.8 – Exemple de propriété générée

La figure 4.12 montre l'organisation d'un cas de test. Cette figure présente le méta-modèle d'un cas de test. Ce dernier spécifie qu'un cas de test porte sur une chaine de greffons. Plus précisément chaque élément de la chaine est une partie "*avant*" ou une partie "*après*" d'un greffon. Pour faciliter le raisonnement ici la distinction est faite entre ces deux parties. Cette chaine de greffons commence par un greffon qui est la tête de la chaine. Chaque greffon a un successeur immédiat et un prédécesseur immédiat. L'association *vérifie* entre la méta-classe *greffon* et la méta-classe *propriété* indique que plusieurs propriétés peuvent être attachées à un greffon. Une propriété possède un *nom*, une *valeur*, un *verdict local*.

Ce méta-modèle est implémenté en utilisant EMF (Eclipse Modeling Framework) [1] [69]. Chaque arbre de resolvers est parcouru par un *parser* appelé TREEPARSER que nous avons implémenté. Cet outil fait appel à la fabrique du modèle EMF pour construire un modèle de cas de test.

Nous nous appuyons maintenant sur les listings 4.7 et 4.8 pour illustrer l'initialisation du modèle de test.

L'initialisation comprend deux étapes :
- la construction du modèle de cas de test grâce aux informations contenues dans l'arbre de resolvers et dans la liste des propriétés de non-interférence,
- l'affectation d'une valeur (avant, après) à l'attribut type de chaque partie de greffon.

1. http ://www.eclipse.org/modeling/emf/

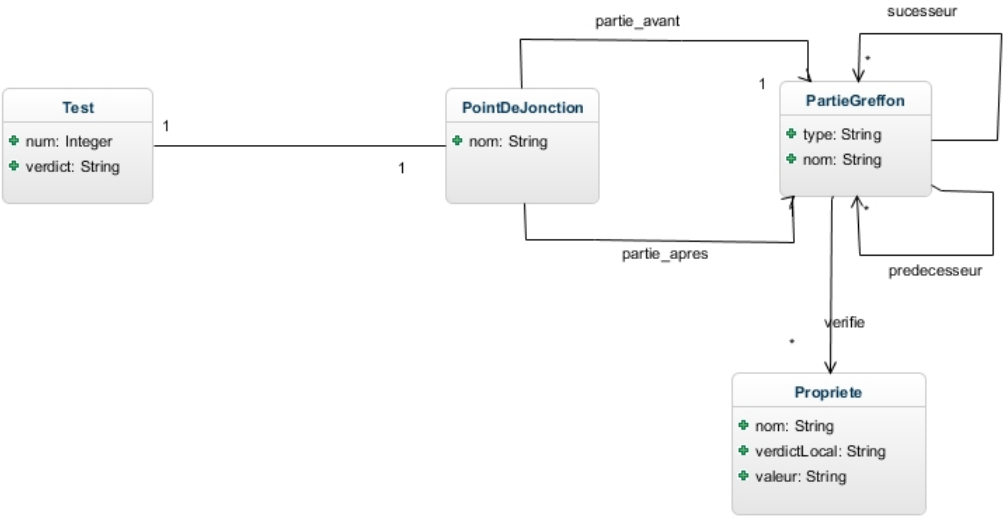


FIGURE 4.12 – Organisation d'une configuration de test

Construction du modèle de cas de test

Le listing 4.7 décrit un arbre de resolvers de niveau 0, c'est-à-dire que tous les greffons sont gérés par un resolver unique. Dans cet arbre un seul resolver, le resolver *r0* gère l'ensemble des greffons A_1, A_2, A_3, A_4, A_5. L'ordre de précédence entre les greffons est le suivant $A_3 < A_1 < A_5 < A_4 < A_2$. La version de A_5 que nous utilisons est celle qui soulève une exception de manière systématique.

Afin de donner une vision générale du déroulement de l'initialisation, nous nous appuyons sur la partie avant de la chaîne de greffons. Notons cependant que l'initialisation, ainsi que toutes les étapes de traitement du modèle de test détaillées par la suite, s'appliquent à la fois à la partie avant et à la partie après d'une chaîne de greffon.

Le listing 4.8 contient la spécification de l'ensemble des interférences qui ne doivent pas se produire à l'égard des greffons du listing 4.7. Le mot clé *before* indique que la spécification porte sur la partie "*avant*" des greffons concernés. Les différentes interférences redoutées suivent le mot clé *before*. Chaque élément indique respectivement l'interférence redoutée et le greffon concerné entre parenthèses. Par exemple CB(A_1) signifie qu'une interaction de type CB sur le greffon A_1 est une interférence. Notons cependant qu'en pratique cette clause s'écrit CB(A_1, v) où v est la variable sur laquelle l'interaction de type CB se produit. Ici afin d'être plus concis nous utiliserons CB(A_1) puisqu'une seule variable existe dans le code de base considéré.

À partir de l'arbre de resolver du listing 4.7 et des propriétés du listing 4.8, le modèle présenté dans la figure 4.11 est construit. Notons qu'à cette étape nous avons utilisé uniquement les paramètres d'entrée du cas de test pour construire le modèle, c'est-à-dire l'arbre de resolver et la liste des propriétés à observer. Le modèle instancié est alors celui représenté par la partie la plus haute de la figure 4.11.

Typage des greffons

À cette étape, l'attribut type du greffon est initialisé à l'aide d'un outil de classification auto-

matique des greffons appelé ABIS. Cet outil fournit par FREDDY MUNOZ et décrit dans [54] permet d'associer à un greffon, un type par analyse statique. Dans notre cas, les greffons peuvent être de trois types :

- Interruption (I) : le greffon remplace le comportement existant par un nouveau comportement.
- Écriture (E) : le greffon réalise des écritures affectant les valeurs d'une ou plusieurs variables du programme de base.
- Lecture (L) : le greffon réalise des lectures des valeurs d'un ou plusieurs attributs du programme de base.

Le modèle après son initialisation peut être décrit comme suit. Ce modèle représente le cas de test *test1*. Ce cas de test porte sur les parties "*avant*" des greffons A_1, A_2, A_3, A_4, A_5 . A_3 a pour successeur immédiat A_1 qui a lui-même pour successeur A_5 puis A_4 et A_2. Les informations qui ont servi à construire cette partie du modèle proviennent du listing 4.7. De la même manière les propriétés spécifiées dans le listing 4.8 ont servi à instancier les propriétés attachées à A_1, A_2, A_3, A_4, A_5. Par exemple une propriété CB est attachée au greffon A_1. Cette étape est l'initialisation du cas de test sous forme de modèle afin de permettre son traitement par l'oracle.

Comme nous l'avons vu précédemment notre oracle prend trois entrées, un cas de test composé d'un arbre de resolvers et d'un ensemble de propriétés de non-interférence, et une trace d'exécution. À partir de leur analyse, il prononce un verdict sur le succès ou l'échec de ce cas de test. Nous venons de voir comment initialiser le modèle de cas de test interprétable par notre oracle à partir de l'arbre de resolvers et des propriétés de non interférence et d'une classification des greffons. Voyons maintenant comment inclure dans ce modèle les informations issues de la trace d'exécution.

Après l'initialisation, la construction du modèle de cas de test est effectué en trois étapes :

- la prise en compte des informations contenues dans la trace d'exécution
- le traitement de l'activation des assertions.
- le traitement des faux positifs et faux négatifs.

```
1  CB(A1,v), IB(A2)
```

Listing 4.9 – Trace d'exécution de l'arbre de resolver du listing 4.7 instrumenté pour observer les propriétés spécifiées dans le listing 4.8

4.5.3.2 Prise en compte des informations contenues dans la trace d'exécution

L'étape suivante consiste donc à alimenter le modèle avec des informations issues de la trace d'exécution. Dans cette étape, la trace d'exécution du cas de test est parcourue par un *parser* nommé TRACEPARSER. Cet outil s'appuie tout comme le précédent sur la fabrique EMF pour ajouter au modèle des informations disponibles dans la trace.

Chaque élément de la trace représente la violation d'une propriété de non-interférence. Ces violations sont utilisées pour renseigner la valeur de l'attribut *valeur* d'une instance de propriété. Par exemple la trace figurant dans le listing 4.9 est celle qui résulte de l'exécution de l'arbre de resolvers décrit dans le listing 4.7. On a observé deux interférences de type CB(A_1, v) et IB(A_2).Les différents éléments de la trace ont été produit par nos assertions exécutables. Les

assertions ont été construites à partir des propriétés spécifiées dans le listing 4.8.

Par exemple à la lecture de l'élément CB(A_1, v), cette donnée est utilisée pour affecter l'attribut valeur de la propriété CB attaché à A_1 à vrai (*true*). Finalement le parcours de cette trace aboutit à un modèle dont les instances des propriétés possèdent les valeurs telles quelles sont décrites dans la seconde partie du diagramme d'objets de la figure 4.11. Ce dernier reprend les informations présentées dans la partie correspondant à l'initialisation et présente également le contenu de l'attribut *valeur* des instances de propriété attachées à chaque greffon.

4.5.3.3 Traitement de l'activation des assertions

L'étape suivante consiste à distinguer les cas où une interférence n'a pas pu être tracé par les greffons de vérification, des cas où l'interférence n'a pas été détectée par défaut d'instrumentation.

Nous illustrons ce cas de figure en nous appuyant sur le listing 4.7 et le listing 4.8. Dans ce dernier la partie avant du greffon A_4 précède la partie avant du greffon A_2. Puisque A_4 écrit dans la variable v utilisée par A_2 et que cette action est considérée en vertu de la spécification du listing 4.8 comme non désirée ; on pourrait penser qu'il y a une interférence de type CB. Or l'interférence CB(A_2, v) ne figure pas dans la trace d'exécution.

Rappelons que la version de A_5 utilisée soulève une exception et interrompt donc la chaîne de greffon. Rappelons que l'instrumentation d'une propriété de type CB pour un greffon A_i consiste à insérer une greffon stockant les valeurs à préserver au début de la chaine et un greffon vérifiant que la valeur concernée n'a pas été changée avant le greffon cible A_i.

Ici, le greffon *AChecker* placé avant A_2 dans la chaîne d'aspects n'est pas activé, de même que A_2. De ce fait, même si la valeur de v à été modifiée aucun élément dans la trace ne fait état d'une interférence CB(A_2, v).

Afin que l'oracle de test soit en mesure de prononcer un verdict sur le succès ou l'échec d'un cas de test, nous devons dans un premier temps discriminer les cas où l'instrumentation dédiée à la détection d'une interférence n'est pas activée. Nous nous appuyons pour cela sur des "motifs de séquence de greffons" dans la chaîne d'aspects. Lorsque ces motifs sont présents, ils impliquent que certaines propriétés ne sont pas tracées et donc ne figurent pas dans la trace d'exécution.

Nous avons exhiber trois motifs de non-activation. Chaque motif implique qu'une partie spécifique de l'instrumentation d'une propriété de non-interférence n'est pas activée. Ces motifs sont utilisés par un *parser* appelé NOTRACEPARSER que nous avons implémenté. Nous présentons maintenant ces motifs d'application de greffons ainsi que les règles de traçabilité des interférences CB, CA, IB, IA. Ces motifs sont formés à partir des informations contenues dans l'attribut "type" de chaque greffon du modèle de test.

- **Motif de non-activation pour CB** : le premier motif permet de déterminer si une interférence de type CB portant sur un greffon A_i est tracée ou pas. Une interférence de type CB est instrumentée en utilisant un greffon qui récupère la valeur initiale d'une variable v du code de base (*storer*) et un greffon qui vérifie que cette valeur n'a pas été modifiée jusqu'à l'exécution de A_i (*checker*). D'une manière générale l'instrumentation d'une pro-

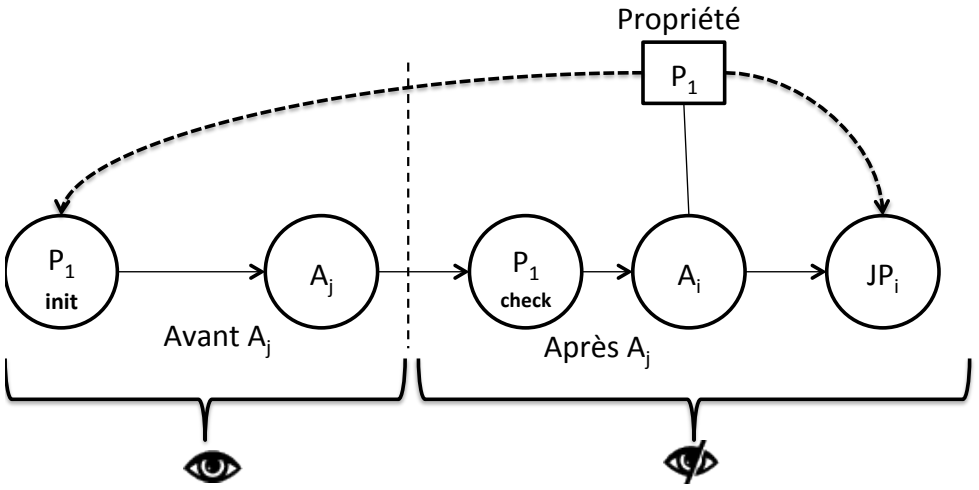


FIGURE 4.13 – Traçabilité des interférence de type CB

priété de non-interférence portant sur le flot de données requiert une initialisation P_{init} et une vérification P_{check}. Ici l'initialisation P_{init} correspond au greffon *storer* et la vérification P_{check} correspond au greffon *checker*.

Le raisonnement est le suivant : pour les interférences de type CB attachées à un greffon A_i, si un greffon A_j placé avant A_i est de type I, $(I < A_i)$ alors P_{check} dédiée à la détection de CB sur A_i n'est pas activé. La figure 4.13 illustre ce cas.

– **Motif de non-activation pour CA** : le second motif permet de déterminer si une interférence de type CA portant sur un greffon A_i est tracée ou pas. Le raisonnement est similaire en ce qui concerne la traçabilité des interférences de type CA excepté le fait qu'ici l'initialisation et la vérification ne sont pas activées. Dans le cas où une propriété de non-interférence de type CA est attachée à un greffon A_i, si un greffon A_j placé avant A_i est de type I alors P_{check} dédiée à la détection de CA n'est pas activé. Ce cas de figure est illustré dans la figure 4.14. Il peut aussi y avoir une interruption entre P_{init} et P_{check}, auquel cas la détection n'est pas non plus activée

– **Motif de non-activation pour IB** : Les interférences de flot de contrôle constituent une exception en ce qui concerne la traçabilité. D'une manière générale l'instrumentation dédiée à la détection des interférences de type IB et des interférences de type IA se décompose en trois parties : P_{init} qui initialise les propriétés, P_{check1} , P_{check2}. P_{check2} est réalisée par du code situé dans le resolver racine. De ce fait, toutes les interférences dont la détection est initialisée sont détectées grâce au code situé dans le resolver racine. Le seul cas où une interférence de type IB où IA n'est pas tracée est lorsque l'initialisation n'est pas réalisée.

Le tableau 4.2 récapitule les différents motifs de non-activation des assertions exécutables pour chaque type d'interférence.

Grâce à ces motifs, nous pouvons effectuer un traitement sur les instances de propriétés du modèle de test afin de discriminer les interférences non tracées. Le modèle de test est parcouru et pour chaque greffon A_i et pour chaque instance de propriété attachée à ce greffon, l'attribut

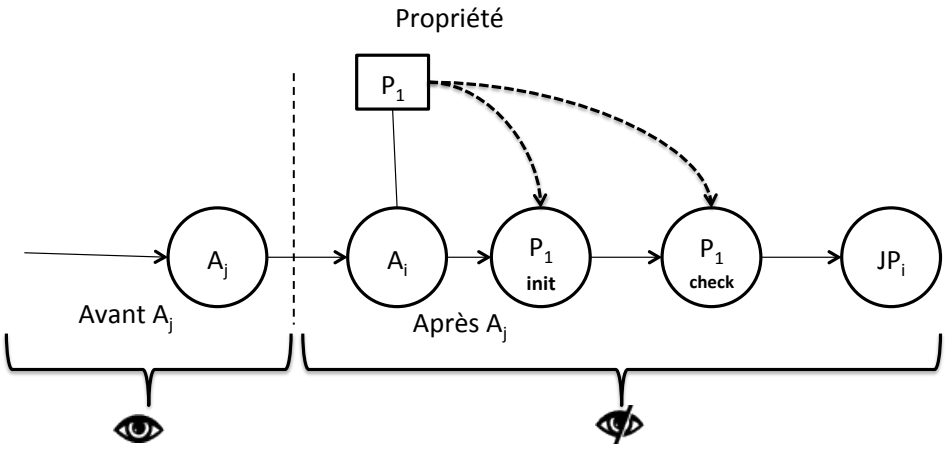


FIGURE 4.14 – Traçabilité des interférence de type CA

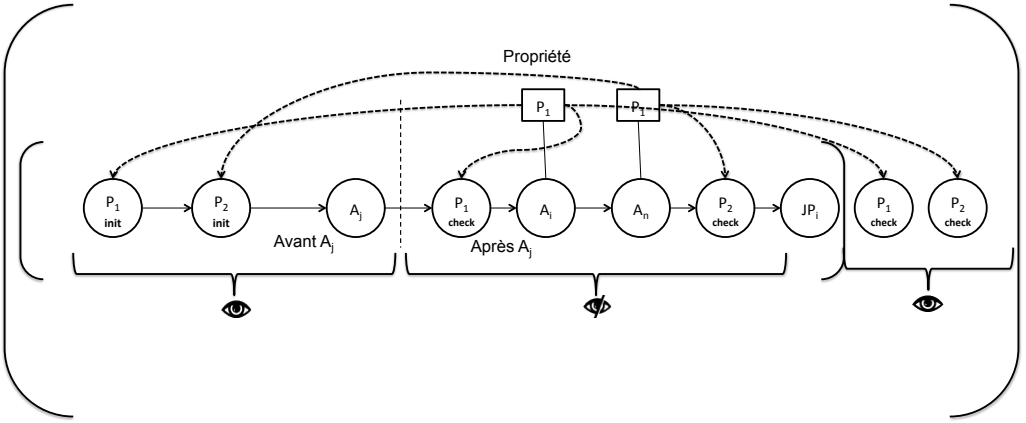


FIGURE 4.15 – Traçabilité des interférence de flot de contôle

valeur de ce dernier est affecté à *NaN* si le greffon appartient à un des motifs de non-activation. *NaN* signifie que l'interférence n'est pas observée dans la trace d'exécution non pas par défaut de l'instrumentation, mais par défaut d'activation de l'instrumentation. Le modèle résultant de ce traitement est visible dans la troisième partie de la figure 4.11. À ce stade, un verdict partiel peut être émis : tous les cas *NaN* qui n'avaient pas été observés dans la trace ont un verdict pass ; par contre l'observation d'une interférence dans un cas *NaN* indique un défaut d'instrumentation, d'où un verdict "faux positifs" du test.

4.5.3.4 Faux positifs et faux négatifs

L'étape suivante consiste à vérifier que les éléments de la trace d'exécution sont d'une part des interférences qui ont été observées à juste titre (pas de faux positifs) et d'autre part que tous les éléments qui doivent être observés sont observés (pas de faux négatifs).

Dans cette étape nous nous appuyons sur la troisième partie de la figure 4.11. Le but est de discriminer les cas différents de *NaN* où l'interférence est tracée alors qu'elle ne devrait pas l'être (faux positifs) et les cas où l'interférence n'est pas tracée alors qu'elle se produit (faux né-

TABLE 4.2 – Motifs de non-activation des assertions exécutables

Propriété	Motifs de non-activation
CB	$P_{init} < I < P_{check}$
CA	$I < A_i < P_{init} < P_{check}$
CA	$A_i < P_{init} < I < P_{check}$
IA	$I < A_i < P_{init} < P_{check1} < P_{check2}$

gatifs). Le modèle résultant du traitement que nous allons réaliser est visible dans la quatrième partie de la figure 4.11.

Nous nous appuyons sur le type des greffons. Il s'agit ici de dégager des "motifs de séquence de greffons" qui impliquent l'apparition d'une interférence. Puis nous comparons ces motifs avec le modèle de test. Trois cas sont alors possibles :

- Si le motif impliquant une interférence est observé mais que l'interférence n'apparaît pas dans le modèle de test alors il s'agit d'un faux négatif.
- Si le motif impliquant l'interférence n'apparaît pas dans le modèle de test mais que l'interférence apparait dans la trace il s'agit d'un faux positif.
- Si le motif impliquant l'interférence apparaît dans le modèle de test et que l'interférence apparaît dans la trace alors la détection a réussi (verdict Pass).

Nous listons maintenant les différents motifs de séquence de greffon pour les différents types d'interférence. Nous détaillons les motifs pour la partie "*avant*" de la chaine de greffons. Les motifs portant sur la partie "*après*" ne sont pas détaillés ici. Ils sont construits sur le même principe et figurent dans le tableau 4.4.

- **Motif de détection pour CB** : une interférence de type CB se produit lorsqu'un greffon A_i accède à une variable $v \in V_{in}(A_i)$ du code de base, la valeur de cette dernière étant changée par d'autres greffons exécutés avant A_i. Le motif de séquence correspondant à cette interférence est $E < L$. Ce qui correspond au cas où un greffon de type "écriture" (E) est appliqué avant un greffon de type "lecture" (L).

- **Motif de détection pour CA** : une interférence de type CA se produit lorsque un greffon A_i accède à une variable $v \in V_{out}(A_i)$ du code de base, la valeur de cette dernière est changée plus tard dans le flot d'exécution par d'autres greffons exécutés après A_i. Le motif de séquence correspondant à cette interférence est $E < E$. Ce motif correspond à l'application d'un greffon de type "écriture" (E) après un autre greffon de type "écriture" (E).

- **Motif de détection pour IB** : une interférence de type IB se produit lorsqu'un greffon exécuté avant A_i met le système dans un état qui n'est plus un point de jonction pour A_i. De ce fait, cet aspect exécuté avant A_i empêche l'exécution de A_i. Le motif de séquence correspondant à cette interférence est $I < *$. Ce motif correspond à l'application d'un greffon de type "interruption" (I) avant un greffon de type quelconque.

- **Motif de détection pour IA** : une interférence de type IA se produit lorsqu'un aspect exécuté après A_i amène le système dans un état qui n'est plus un point de jonction pour A_i, il supprime l'exécution du point de jonction de A_i après qu'il se soit exécuté à ce dernier.

Le motif de séquence correspondant à cette interférence est $* < I$. Ce cas correspond à l'application d'un greffon de type interruption après un autre greffon.

TABLE 4.3 – Motifs de détection des interférences. E = écriture, L= lecture, I= interruption

Propriété	Motifs de détection
CB sur la partie "*avant*"	$écriture < A_i < jp_i$
CB sur la partie "*après*"	$jp_i < écriture < A_i$
CA sur la partie "*avant*"	$A_i < écriture < jp_i$
CA sur la partie "*après*"	$jp_i < A_i < écriture$
IB sur la partie "*avant*"	$interruption < A_i < jp_i$
IB sur la partie "*après*"	$jp_i < interruption < A_i$
IA sur la partie "*avant*"	$A_i < interruption < jp_i$
IA sur la partie "*après*"	$\emptyset$

En utilisant ces motifs le modèle de la figure 4.11 est parcouru. Pour chaque propriété attachée à chaque greffon, nous affectons une valeur à l'attribut valeur de la propriété (faux positif, faux négatif, Pass).

À chaque cas de test est associé un attribut verdict (cf. figure 4.11). Le verdict pour un cas de test est rendu en parcourant toutes les propriétés de chaque greffon du modèle de test. Si pour toutes les propriétés l'attribut *verdict local* vaut "Pass", l'attribut verdict du cas de test est affecté à "Pass". Cela signifie que le cas de test réussit et donc que l'instrumentation détecte bien les interférences.

4.6 Expérimentations et résultats

Nous présentons maintenant les expérimentations et les résultats qui ont permis de valider notre méthode d'instrumentation. Nous rappelons brièvement les objectifs de test visés puis nous détaillons les différentes expérimentations.

4.6.1 Vue globale de expérimentations

Afin d'évaluer notre méthode d'instrumentation et son implémentation, nous avons réalisé deux séries d'expériences :

- La première se concentre sur la partie "*avant*" d'une chaîne de greffons et considère chacun des cas interférences CB, CA, IA, IB de manière isolée.
- La deuxième série est une généralisation de l'utilisation des assertions aux interférences multiples dans les parties avant et après des greffons.

4.6.1.1 Les premières expériences avec des greffons *before*

Dans la première série d'expériences, nous avons développé le petit exemple impliquant les trois aspects *Alog*, *ACrypt*, *AAuth*. Les trois aspects sont tissés dans une application de base simple à un point de jonction partagé (un appel à la méthode *Send*). Nous avons produit 60 arbres de resolvers. Il y a 6 ordres possibles (par exemple LCA pour *A**L**og*, *A**C**rypt*, *AA**uth***), et

10 implémentations alternatives d'arbres de resolvers pour chaque ordre. Le premier arbre correspond à un cas de base avec un resolver unique. Les 9 autres correspondent à des arbres de resolvers générés aléatoirement, ajoutant un ou deux niveaux de resolvers au cas de base.

Dans les différents arbres de resolvers générés, le fait qu'une interaction soit considérée comme une interférence dépendra de l'utilisation désirée des aspects et donc des propriétés de non-interférence formulées par leur intégrateur. Plusieurs scénarios d'utilisation sont considérés dans le tableau 4.4.

TABLE 4.4 – Scenarios d'interférence

Scenarios	ordres incorrects
Alog doit enregistrer des messages en clair	ACL, CAL, CLA
Alog enregistre des messages chiffrés	ALC, LAC, LCA
Alog enregistre toutes les tentatives d'envoi	ACL, ALC, CAL
Alog enregistre uniquement les messages qui sont effectivement envoyés au serveur	CLA, LAC, LCA
Alog enregistre uniquement les messages en clair qui qui sont effectivement envoyés au serveur	tout sauf ALC

Les quatre premiers scénarios contiennent une seule interférence (respectivement CB, CA, IB et IA). Ils nous permettent de démontrer notre capacité à détecter chacune d'elles. Bien que les assertions multiples ne soient pas la principale cible de ces expérimentations, le dernier scénario introduit un cas avec deux interférences interdites (CB et IA). Pour chaque scénario, l'instrumentation appropriée est automatiquement insérée dans l'arbre de resolvers (pour les 60 arbres de resolvers). Les interférences de flot de données sont testés avec une authentification réussie. Pour les interférences de flot de contrôle, nous considérons deux variantes de *AAuth*, l'une où *AAuth* n'appelle pas $proceed$ et l'autre où elle soulève une exception.

Dans tous les cas, nous obtenons le résultat attendu : la détection des interférences se produit chaque fois que l'ordre d'exécution viole les propriétés attendues.

4.6.1.2 Généralisation : greffons *around* et assertions multiples

Les expérimentations précédentes se concentrent sur la partie "*avant*" d'une chaîne de greffons et considèrent chacun des cas d'interférences CB, CA, IA, IB de manière isolée. Une deuxième série d'expériences vérifie que :

- Nous sommes en mesure de détecter des interférences dans les différentes parties d'une chaîne de greffons (dans les parties "*avant*" et "*après*").
- Les greffons de vérification n'ont pas d'effet de bord, c'est-à-dire que les assertions ciblant différentes propriétés de non-interférence peuvent être composées en toute sécurité.

Puisque nous considérons maintenant une instrumentation généralisée à toute la chaîne de greffons, c'est-à-dire à la partie "*avant*" et à la partie "*après*", nous devons considérer un autre exemple. En effet si nous réutilisons les aspects *Alog*, *Acrypt*, *AAuth* précédemment utilisés, certains scénarios sont dénués de sens. Puisque nous utilisons un générateur aléatoire pour la construction des arbres de resolvers, nous pouvons obtenir des arbres où par exemple

le greffon *AAuth.auth* est appliqué après le point de jonction, en dernière position dans la partie "*après*" de la chaîne de greffon. Du fait de la génération aléatoire en considérant *Alog, Acrypt, AAuth* de nombreux scénarios correspondant à des arbres de resolvers générés ne sont pas réalistes.

Les expériences impliquent donc un nouvel exemple artificiel avec quatre greffons en interaction :

- N_1, N_2 sont des greffons neutres du point de vue des interférences, effectuant uniquement des accès en lecture seule sur une variable v du code base ;
- W écrit v et peut donc induire des interférences de type flot de données ;
- *Ab* peut interrompre la chaîne d'exécution des greffons.

Toutes les propriétés portent sur des interférences affectant les greffons neutres, par exemple la partie après de N_1 doit toujours être exécutée si le point de jonction est exécuté.

L'exemple artificiel est en fait une généralisation de l'exemple précédent où le greffon de journalisation était neutre, le chiffrement jouait le rôle d'écrivain et l'authentification pouvait interrompre la chaine d'exécution. La généralisation tient au fait que les accès lecture / écriture sont désormais réalisés à la fois dans les parties avant et après des greffons et que *Ab* est disponible en 4 versions :

- (1) n'interrompt pas la chaine d'exécution,
- (2) soulève une exception dans la partie avant,
- (3) *proceed* n'est pas appelé,
- (4) soulève une exception dans la partie après.

Notons que l'utilisation de deux greffons neutres N_1, N_2 est liée au deuxième objectif de cette série d'expériences : vérifier que les greffons de vérification utilisés (*AStorer*...) n'ont pas d'effet secondaire sur les autres greffons de vérification. Grâce à N_1, N_2 nous pouvons par exemple vérifier que nous pouvons détecter plusieurs interférences de flot de données sur N_1 et N_2. Le fait que N_1 et N_2 lisent la même variable v donne lieu au cas où plusieurs greffons *AChecker.check* existent pour la variable v. Ces derniers doivent rester indépendants car ils portent sur la valeur de v à différents points d'exécution.

Le tableau 4.5 donne les paramètres de la fonction de génération qui ont été utilisés pour notre étude de faisabilité. Nous avons obtenu un échantillon de 10 arbres pour chaque ordre possible des greffons cibles (10 * 24 = 240 arbres pour l'étude de cas avec 4 greffons) : un arbre pour le scénario de base, ainsi que 9 arbres dérivés aléatoirement du cas de base.

TABLE 4.5 – Paramètres utilisés pour la génération

Nombre de cas de test généré à partir du cas de base = 9
Profondeur maximale de l'arbre en dessous de la racine = 3
Nombre maximum de resolvers exécutés à une profondeur donnée = 2
Probabilité de supprimer des entités redondantes du champ d'application du parent = 0.75
Probabilité d'insérer un resolver inutile à une profondeur donnée = 0.25

4.6.1.3 Discussion des résultats

Les résultats de la première série d'expérimentations confirment le fait que les cas simples d'interférence sont tous détectés. Rappelons qu'on se limite ici à la partie avant de la chaîne de greffons. Les interférences sont de type CB, CA, IA, IB.

La deuxième série d'expérimentations offre deux résultats.

D'une part elle montre que les assertions exécutables permettent de détecter les interférences dans les parties avant et après, ou avant et après conjointement, des points de jonction partagés.

D'autre part, sur chaque partie avant et après, plusieurs instrumentations visant différentes propriétés peuvent être combinées. Nous avons généré un échantillon de 14.400 configurations de paramètres de cas de test. Les structures d'arbres couvrent tous les ordres possibles. Les ensembles de propriétés générées comprennent le cas maximal avec toutes les propriétés à vérifier.

4.7 Bilan

Dans ce chapitre, nous avons présenté une méthode d'instrumentation d'une configuration d'aspects. Nous avons ensuite détaillé les propriétés de non interférences qui doivent être observées afin de garantir qu'un assemblage d'aspect ne contient pas d'interférences CB, CA, IA, IB. Pour chaque propriété, nous avons décrit des points d'observation dans le code permettant de les observer. Nous avons ensuite défini une méthode permettant d'instrumenter ces points d'observation. Par rapport à d'autres approches décrites au chapitre 2, l'utilisation d'assertions permet une observation à l'exécution du système implémenté.

Nous avons également validé notre approche d'instrumentation de manière intensive. Tous les cas d'interférence ciblés dans cette thèse peuvent être détectés et cela indépendamment de la complexité de leur séquencement.

Chapitre 5

Etude de cas : Implémentation orientée aspect d'un protocole PBR

Préambule

Dans ce chapitre, nous proposons d'appliquer nos travaux à un cas d'étude. Ce cas d'étude consiste à implémenter un mécanisme de réplication duplex dans une architecture distribuée. Nous commençons par expliquer le fonctionnement du protocole duplex et de l'implémentation que nous considérons. Puis nous détaillons les différentes préoccupations transversales qui le composent et nous montrons comment ce mécanisme peut être conçu en utilisant des aspects implémentés à grain fin. Nous montrons comment les différents aspects peuvent être composés pour réaliser le comportement du protocole duplex. Nous présentons la spécification des aspects au regard des propriétés de non interférences qui doivent être réalisées lors de leur composition. Nous détaillerons les différents scénarios où des problèmes d'interférences peuvent apparaître lors de l'intégration de ces aspects.

Sommaire

5.1 Introduction

NOUS avons présenté dans le chapitre précédent une approche basée sur les resolvers, visant à éviter et détecter des problèmes de composition d'aspects.

La construction de systèmes résilients nécessite la capacité de reconfigurer le logiciel de tolérance aux fautes d'un système en fonction de son environnement, de son modèle de fautes et de ses ressources. Une implémentation traditionnelle de la tolérance aux fautes aboutie à un code difficilement reconfigurable à moindre effort, étant donné la dispersion (*code scattering*) et l'entrelacement du code de tolérance aux fautes (*code tangling*) et du code fonctionnel. Contrairement aux approches traditionnelles, la programmation orientée aspect permet d'implémenter des systèmes disposant d'un logiciel de tolérance aux fautes reconfigurable. Cependant, lors de la configuration ou la reconfiguration d'un logiciel de tolérance aux fautes implémenté à l'aide d'aspects, des problèmes issus de l'interaction entre plusieurs aspects peuvent apparaître.

Dans ce chapitre, nous présentons un scénario d'implémentation de la tolérance aux fautes d'un système par la composition d'aspects. Nous présentons notre cas d'étude et le mécanisme de tolérance aux fautes qui y est mis en œuvre. Il s'agit en l'occurrence d'un mécanisme de réplication duplex. Nous étudions ensuite les différents aspects composant le protocole duplex. Nous illustrons la nécessité d'une implémentation à grain fin de ces mécanismes et la facilité de reconfiguration qu'une telle implémentation procure.

Nous illustrons comment ces aspects sont assemblés de manière à réaliser le comportement du protocole duplex, ce qui nous permet d'aborder la problématique de l'assemblage non anticipé d'aspects et des interférences qui peuvent en découler. Nous présentons une application du processus de spécification qui permet d'exhiber les propriétés de non-interférences à vérifier. Puis nous appliquons alors notre approche d'instrumentation de l'implémentation orientée aspects d'un protocole de réplication duplex. Nous présentons également un scénario illustrant l'application de notre méthode de détection lors de l'évolution des conditions d'exploitation du système considéré dans notre cas d'étude.

Ce scénario met en évidence les principaux défis qui ont été ciblés par notre travail : i) l'implémentation de mécanismes de tolérance aux fautes à grain fin ii) la reconfiguration de la tolérance aux fautes et iii) la validation de la composition des mécanismes.

Motivations

Ce chapitre vise deux objectifs :
- Illustrer la construction d'un protocole de tolérance aux fautes à l'aide d'aspects réutilisables permettant de transformer n'importe quel serveur fonctionnant comme un singleton en un serveur fonctionnant de manière répliquée.
- illustrer comment notre approche d'instrumentation permet de détecter les différents cas d'interférences qui peuvent apparaitre lors des différents cas de réutilisation de ces aspects de ces aspects.

Structure du chapitre

Ce chapitre est organisé comme suit : dans la section 5.2, nous présentons l'étude de cas

que nous utilisons pour la validation de nos travaux à travers une brève description du système considéré, de son modèle de fautes et les mécanismes de tolérance aux fautes associés. La section 5.3 met en évidence les différentes préoccupations transversales qui composent le protocole duplex, le mécanisme utilisé dans notre étude. Elle présente également une décomposition sous forme d'aspects des différentes préoccupations.

La section 5.4 illustre la composition des aspects pour réaliser le comportement du protocole duplex. La section 5.5 présente la spécification des propriétés de non-interférence qui doivent être réalisées pour que le protocole duplex assemblé réalise le comportement attendu. La section 5.6 montre comment la spécification obtenue est utilisée pour instrumenter l'assemblage des aspects composant le protocole duplex et comment cette instrumentation permet de détecter les interférences qui peuvent être introduites dans l'assemblage.

La section 5.7 présente deux scénarios de reconfiguration qui serviront à illustrer l'application de notre approche et illustre l'application de notre approche de détection d'interférences lors de l'évolution du logiciel de tolérance aux fautes. La section 5.8 conclut ce chapitre.

5.2 Cas d'étude et modèle de fautes

Cette section présente le cas d'étude auquel nous appliquons notre approche, le modèle de faute lié au système considéré et les mécanismes de tolérance aux fautes associés.

5.2.1 Cas d'étude

Dans ce chapitre nous considérons un système volontairement simplifié du point de vue fonctionnel afin de mettre l'accent sur les difficultés à réaliser une implémentation de la tolérance aux fautes clairement séparée du code fonctionnel, réutilisable et reconfigurable comme discuté au chapitre 1.

Le système est composé d'un client et d'un serveur qui rend un service de calcul simple. D'un point de vue fonctionnel, il n'est pas nécessaire de traiter un cas complexe pour effectuer notre démonstration, puisque nous ciblons les mécanismes non-fonctionnels de tolérance aux fautes.

Les propriétés des composants du système et des liens de communication entre ces composants sont les suivants :

- Le serveur est considéré comme un composant auto-testable (*self-checking*) ce qui signifie que des mécanismes internes permettent de garantir que le serveur ne fournit aucune valeur incorrecte message corrompu ; ici on fait l'hypothèse qu'il n'y a qu'un seul mode de défaillance pour le serveur, le *crash*. Le serveur a ainsi un comportement à silence sur défaillance (*fail-silent*).
- Le lien de communication entre le serveur et le client est fiable et FIFO, ce qui signifie qu'un message arrive toujours au serveur en l'absence de *crash*. Nous nous plaçons dans un modèle synchrone, tout message émis est reçu par un récepteur au bout d'un temps borné.

En cas de *crash* le service rendu par notre système n'est donc plus fourni. Nous nous intéressons à l'ajout de mécanismes permettant de continuer à fournir ce service même en cas de

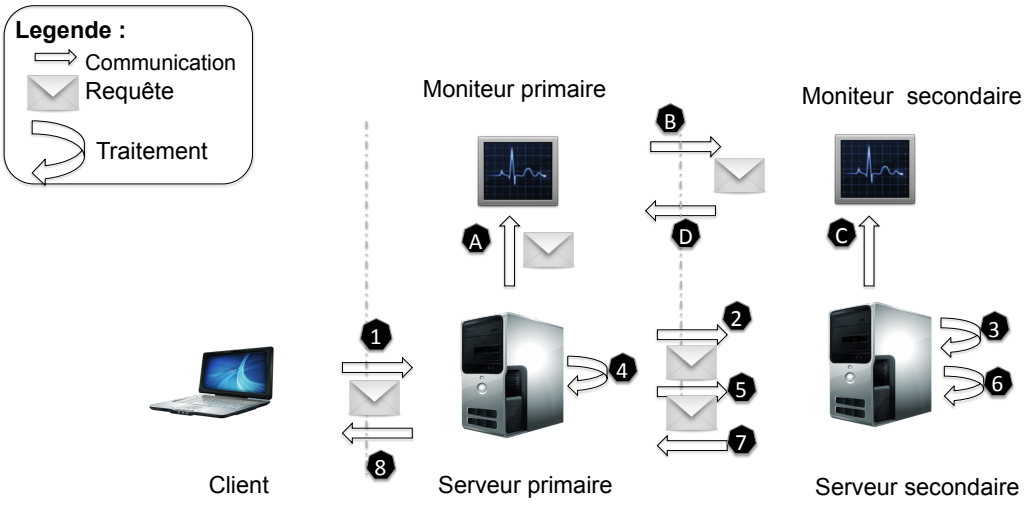


FIGURE 5.1 – Schéma de fonctionnement général du protocole de réplication duplex (réplication passive)

crash. En d'autres termes, le rendre capable de tolérer les fautes de type *crash*.

5.2.2 Modèle de fautes et mécanismes de tolérance aux fautes

Une solution intuitive pour continuer à fournir un service malgré la défaillance du serveur consiste à remplacer le serveur défaillant par un serveur non défaillant. Ce remplacement doit être fait sans interruption du service et c'est l'idée de la réplication de service selon un protocole duplex. Plusieurs variantes du protocole de synchronisation des répliques peuvent être utilisées. Nous décrivons maintenant le fonctionnement d'une instanciation du protocole duplex appelée *Primary Backup Replication*.

Dans le protocole duplex, deux serveurs sont en mesure de fournir le service au client, grâce à une synchronisation des répliques par transfert d'état. Chaque serveur fonctionne sur un nœud du réseau. Un seul serveur dit primaire (*primary*) exécute les requêtes du client, le secondaire (*backups*) met à jour son propre état à partir de points de reprise (informations d'état) propagés par le serveur primaire. Lorsque le serveur primaire défaille (*crash*), le serveur secondaire prend le relais et commence à traiter les requêtes à partir du dernier état reçu. En pratique, les deux serveurs possèdent le même code mais fonctionnent selon des modes différents. Les différentes copies du serveur fonctionnant sur des nœuds physiquement séparés sont appelées des répliques.

Un service de surveillance sert de détecteur de *crash* sur chaque nœud. Dans la figure , ces services sont implémentés par les moniteurs primaire et secondaire. Les nœuds exécutant les répliques envoient périodiquement un message *"I am alive"* à leur moniteur, ce qui permet de détecter la défaillance de n'importe quel serveur (i.e. l'absence de message *"I am alive"*).

Le principe de base de ce mécanisme simplifié dans notre étude de cas est résumé dans la figure 5.2.2. Le client envoie sa requête au serveur principal ❶. Le serveur propage la requête vers le serveur secondaire ❷. Le secondaire sauvegarde la requête ❸. Le primaire effectue le calcul correspondant au service ❹ puis, après la réalisation du service, propage le résultat et son état au secondaire ❺. Le serveur secondaire met à jour son état ❻ puis accuse réception

des données de mise à jour au serveur primaire ❼. Pour finir le serveur primaire envoie le résultat de la requête au client ❽. Lorsqu'un *crash* de la réplique primaire est détecté, la réplique de sauvegarde change de mode pour devenir la nouvelle réplique primaire.

Il existe principalement deux façons de réaliser la synchronisation de l'état :
- L'état peut être stocké dans un support de stockage fiable. Lors de l'élection d'une nouvelle réplique primaire, après la détection d'un crash, l'état stocké dans le support de stockage est rechargé dans la nouvelle réplique primaire, puis cette réplique est chargée et démarrée. La stratégie est qualifiée de *Cold Primary Backup Replication*.
- L'état est pré-chargé dans la réplique secondaire qui reste inactive d'un point de vue purement fonctionnel mais réceptionne et traite les points de reprise. Sur détection du *crash*, la réplique est simplement démarrée. La stratégie est qualifiée de *Warm Primary Backup Replication*.

Nous proposons de mettre en place la stratégie *Warm Primary Backup Replication* avec une sauvegarde systématique de l'état de reprise de la réplique par le secondaire.

5.3 Protocole de réplication duplex

Dans cette section nous décrivons en détail le fonctionnement du protocole duplex afin de déterminer les aspects qui seront nécessaires à sa mise en œuvre. Afin de faciliter la compréhension de ce dernier nous le décrivons en deux temps. Nous détaillons dans un premier temps le protocole client-serveur qui structure les interactions entre le client et le serveur en section 5.3.1. Puis nous décrivons le protocole inter-répliques, c'est-à-dire l'ensemble des interactions entre le serveur primaire et le serveur secondaire dans la section 5.3.2.

5.3.1 Protocole client-serveur

Dans ce protocole duplex simple, nous ne faisons pas l'hypothèse d'un service de communication de groupe fournissant la diffusion atomique de message. Ainsi le client s'adresse au serveur primaire et en cas de défaillance de celui-ci s'adresse au serveur secondaire. Nous faisons donc l'hypothèse que les ports de communication du primaire et du secondaire sont connus a priori du client. Rappelons aussi que l'on se place dans un système synchrone, à savoir que tout message est reçu dans un intervalle de temps borné en l'absence de *crash*.

5.3.1.1 Préoccupations transversales coté client

Dans le protocole duplex *primary-backup* lorsque le serveur primaire défaille, le serveur secondaire prend le relais. Le client qui n'a pas reçu de réponse à sa requête la renvoie. Le secondaire la reçoit. Le serveur secondaire ne doit alors pas exécuter la requête si elle a déjà été exécutée afin de préserver l'intégrité du système. La requête ne doit être exécutée qu'une seule fois (*only once semantics*). Pour ce faire, il est nécessaire d'ajouter un numéro de séquence à la requête et un identifiant de client. Grâce à ces informations le serveur secondaire sait si pour un client donné (identifiant client) une requête donnée (numéro de séquence de requête) a déjà été traitée. Du côté client, il est donc nécessaire d'ajouter aux requêtes ces informations supplémentaires en plus du nom du service demandé. Ces identifiants sont uniques. De la même manière à la réception de la réponse les informations ajoutées à l'envoie doivent être retirés de la réponse afin que celle-ci soit traitée. Le client doit également gérer la réception des réponses et prendre des décisions en ce qui concerne la disponibilité du serveur. En effet lorsque le client

ne reçoit pas de réponse à une requête après un certain délai, le serveur est considéré comme défaillant. Les requêtes sont alors envoyées au serveur secondaire.

```
1 Variables du protocole:
2 adresse primaire
3 adresse secondaire
4
5 Boucle client {
6 ajouter numero de sequence de requete
7 ajouter identifiant client
8 envoi de la requete au serveur
9 enregistrement de la requete dans le cache
10 armement du  minuteur
11   tant que reponse=null {
12   si( minuteur≤ 0) { exception}
13   si( minuteur> 0) { reception de la reponse
14     si( reponse≠ null) {
15       arret du minuteur,
16       retirer numero de sequence de reponse
17       }
18   }
19   }
20 traitement de la reponse
21 }
```

Listing 5.1 – Algorithme exécuté coté client

Toutes les lignes excepté les lignes 8 et 20 du listing 5.5 sont des préoccupations transversales. Il convient donc de les encapsuler dans des aspects afin de les séparer des préoccupations fonctionnelles implémentées par les lignes 8 et 20.

5.3.1.2 Aspects côté client

Les différentes préoccupations transversales identifiées dans le listing 5.5 peuvent être encapsulées dans différents aspects.

Les préoccupations des lignes 7 et 16 sont encapsulées dans un aspect dédié à la gestion de l'ajout et de la suppression du numéro de séquence de requête. Cet aspect est nommé *ANumbering* contient deux greffons *insert* et *remove* dont les rôles sont les suivants :

- **ANumbering.insert** : ajoute un numéro de séquence à la requête ;
- **ANumbering.remove** : supprime le numéro de séquence de la réponse ;

Celle de la ligne 7 est encapsulée dans un aspect dédié à la gestion des identifiants de client (*AIdentifier*). Cet aspect contient un greffon dont le rôle est le suivant :

- **AIdentifier.insert** : ce greffon ajoute l'identifiant du client à la requête avant qu'elle ne soit envoyée.

Un troisième aspect nommé *ACacheManager.addIn* encapsule les préoccupations transversales de la ligne 9. Cet aspect contient un greffon nommé *ACacheManager.addIn* dont le rôle est le suivant :

- **ACacheManager.addIn** : ce greffon est chargé de sauvegarder les requêtes envoyées dans une structure de données afin de permettre le renvoi des requêtes pour lesquelles aucune réponse n'a été reçue. Il ajoute dans une structure de données l'identifiant de la requête,

son contenu et un minuteur.

Ce greffon permet le renvoi au serveur secondaire d'une requête pour laquelle il n'y a pas eu de réponse de la part du serveur primaire pour cause de défaillance. Si l'on se réfère au comportement décrit par les lignes 11 à 19 lors de l'attente de la réponse à une requête, le client vérifie que le temps d'attente imparti pour une requête n'est pas dépassé par le biais du minuteur affecté à la requête ; si ce denier est dépassé une exception est soulevée. Le client passe alors au traitement de la requête suivante dans sa liste de requêtes à traiter. Or il faut renvoyer la requête pour laquelle aucune réponse n'a été reçue du serveur primaire au serveur secondaire. Ce greffon et la structure de données utilisée contribuent à réaliser ce comportement.

Le renvoi de la requête est géré par un aspect nommé *ARequestResend*. Il encapsule cette préoccupation transversale. Cet aspect contient un greffon nommé *ARequestResend.reload* dont le rôle est le suivant :

– *ARequestResend.reload* : est chargé de recharger en tête de liste des requêtes à envoyer la requête envoyée pour laquelle qui n'a pas reçu de réponse.

L'expression de coupe de *ARequestResend* capture la levée de l'exception liée au dépassement du temps imparti par le minuteur. Le comportement du greffon **ARequestResend.reload** est décrit dans le listing 5.2.

Le greffon *ACacheManager.addIn* détaillé ci-dessus ajoute dans une structure de données les informations liées à une requête et un objet minuteur. Le minuteur fournit une interface proposant deux opérations :

– *start()* le minuteur commence le décompte à partir de la valeur avec laquelle il a été initialisé ;
– *stop()* le décompte du minuteur est arrêté.

Un quatrième aspect (*ATimer*) est donc dédié la gestion de ce minuteur. Il qui contient deux greffons *start* et *stop*. Il encapsule les préoccupations des lignes 10 et 16. Les rôles respectifs de ces greffons sont les suivants :

– **ATimer.start** : appelle l'interface *start()* du minuteur référencé dans la structure de données, qui décompte un certain temps au bout duquel la réponse liée à une requête envoyée au primaire doit être reçue.
– **ATimer.stop** : arrête le minuteur lorsqu'une réponse pour une requête donnée est reçue.

Ces différents aspects sont appliqués autour de deux points de jonction, les méthodes *send()* et *receive()*. Chaque aspect effectue une action permettant de composer le comportement attendu du client dans le protocole de réplication duplex. Nous reviendrons plus en détail sur la composition de ces aspects dans la section 5.4.

```
1 exception{
2 recuperation de la requete dans le cache
3 changement d adresse serveur
4 retour a la boucle client
5 }
```

Listing 5.2 – Algorithme exécuté coté client lors de la levée d'une exception

5.3.2 Protocole inter-répliques

Nous détaillons maintenant le protocole inter-répliques. Nous présentons les différentes préoccupations transversales qui le composent et les aspects qui sont utilisés pour les encapsuler.

La première action réalisée par le serveur consiste à déterminer le mode fonctionnement dans lequel il se trouve, soit en duplex primaire, soit duplex secondaire, soit encore en primaire seul (*single*) dans le cas où le secondaire n'est plus opérationnel. Selon le mode, différents comportements sont possibles. Nous détaillons dans cette section le comportement du serveur dans chacun des modes. Pour chaque mode nous identifions les préoccupations transversales dispersées dans ce comportement et nous proposons une implémentation de ces dernières sous forme d'aspects.

5.3.2.1 Description générale des modes de fonctionnement du serveur

Le serveur peut fonctionner dans trois modes : duplex primaire, duplex secondaire, *single* primaire. Dans chacun de ces modes, le comportement du serveur diffère dans la mesure où les actions qui sont effectuées à la réception d'une requête ne sont pas les mêmes.

La gestion des différents modes est détaillée plus loin dans ce chapitre. Dans cette section nous détaillons les préoccupations transversales mises en œuvre dans chacun des modes. La section suivante illustre comment ces préoccupations sont ensuite encapsulées dans des aspects afin d'implémenter le comportement de chaque mode.

5.3.2.2 Comportement en mode duplex primaire

En mode duplex primaire, le comportement est le suivant. Le serveur est en attente de requêtes des clients :

1. Le client envoie une requête vers le serveur principal, la requête est traitée, l'état actuel du calcul est capturé et emballé dans un message (point de reprise). Le point de reprise est composé des données nécessaires pour relancer le calcul lorsque le serveur principal est défaillant. Le point de reprise se compose dans notre cas des éléments suivants :
   – L'état du système après le traitement d'une demande ;
   – Une copie de la réponse obtenue par le serveur primaire ;
2. Le serveur primaire envoie au serveur secondaire le point de reprise après le traitement de la requête.
3. Le serveur secondaire met à jour son état à partir du point de reprise.
4. Le secondaire envoie au serveur primaire un accusé de réception lorsque son état est enfin mis à jour.
5. Le serveur primaire envoie la réponse au client.

5.3.2.3 Comportement en mode secondaire

En mode secondaire, le comportement est le suivant. Le serveur est en attente de messages de synchronisation du primaire :

1. Le primaire propage une requête vers le serveur secondaire, la requête est stockée par le secondaire mais n'est pas traitée.

2. Le serveur principal envoie au serveur secondaire le point de reprise après le traitement de la requête.
3. Le serveur secondaire met à jour son état à partir du point de reprise.
4. Le secondaire envoie au serveur primaire un accusé de réception lorsque son état est enfin mis à jour.

Lorsque le serveur secondaire reçoit un message du serveur primaire il doit décider des actions à effectuer pour le traitement de cette requête puisque deux cas sont possibles :

– Réception de la requête : la requête est reçue pour la première fois, alors la seule action à entreprendre est de la stocker ;
– Réception du point de reprise : la requête a déjà été reçue, il s'agit donc de traiter le point de reprise lié à cette requête puis d'envoyer un accusé réception au serveur primaire.

```
1 variable du protocole:
2 mode
3
4 Boucle primaire {
5 reception de la requete
6 si (mode= primaire) {
7   envoi de la requete au secondaire
8   retirer numero de sequence de requete
9   retirer identifiant client
10  calcul du resulat
11  envoi du point de reprise au secondaire
12  inserer le numero de sequence de la requete dans la reponse
13  envoi de la reponse au client}
14 }
```

Listing 5.3 – Algorithme du client dans le protocole de réplication duplex *primary backup*

5.3.2.4 Comportement en mode single

Lorsque le serveur primaire tombe en panne, la requête du client est transmise au serveur secondaire. Le serveur secondaire est informé de l'échec du primaire par son moniteur. Par conséquent, il passe du statut de secondaire à primaire et commence le traitement de la demande du client à partir du dernier point de reprise reçu.

Son fonctionnement est similaire au cas standard non répliqué et donc ce mode fonctionnement ne sera pas considéré dans la suite de ce chapitre. En mode primaire unique (single), on peut considérer que le protocole est en cours d'exécution dans un mode dégradé. Dans ce mode, le service est livré sans tolérance aux fautes.

5.3.3 Aspects du protocole inter-répliques

Les lignes 6-9 et 11 sont des préoccupations transversales que nous encapsulons dans des aspects comme décrit ci-dessous.

5.3.3.1 Aspects côté serveur primaire

La gestion de la propagation de la requête reçue par le serveur primaire vers le serveur secondaire est également une préoccupation transversale. Elle est encapsulée dans un aspect appelé *ACheckPoint*. Cet aspect contient deux greffons dont les rôles respectifs sont le suivant :

– **ACheckPoint.forwardRequest** : transmet la requête reçue par le primaire au serveur secondaire.
– **ACheckPoint.buildCheckPoint** : construit un point de reprise à partir de l'état de l'application après l'exécution du service et une copie de la réponse obtenue. Une fois le point de reprise construit, il est envoyé au serveur secondaire.

La chaîne d'octets contenue dans chaque message délivré au serveur directement utilisée pour invoquer les commandes sur le serveur en projetant cette chaîne sur le format de requête attendu. Les informations supplémentaires contenues dans le message peuvent compromettre cette interprétation. Deux greffons *ANumbering.remove* et *AIdentifier.remove* sont chargés de supprimer le numéro de séquence de la requête et l'identifiant de client et d'insérer à nouveau le numéro de séquence de la requête avant l'envoi de la réponse.

L'aspect *ANumbering* utilisé côté client est réutilisé côté serveur. Il s'agit plus précisément de réutiliser le greffon *ANumbering.remove* qui supprime le numéro de séquence de la requête arrivant du côté du serveur avant que la chaîne d'octets reçue ne soit interprétée pour exécuter le service correspondant.

À l'aspect *AIdentifier* nous ajoutons alors un greffon nommé *AIdentifier.remove* qui prend en charge la tâche suivante :
– **AIdentifier.remove** : supprime l'identifiant du client de la requête reçue.

Du coté serveur primaire et secondaire un aspect nommé *AServeurMode* prend en charge les préoccupations liées à l'initialisation du serveur. Il contient un greffon nommé *AServeurMode.init* dont la tâche est la suivante :
– **AServeurMode.init** : lit les informations de configuration dans un fichier et initialise une variable correspondant au mode.

Dans cette section le mode du serveur est initialisé à "Duplex primaire".

```
1 variable du protocole:
2 mode =single
3
4 Boucle primaire single {
5 reception de la requete
6 si( duplication){renvoyer le resultat stocke}
7  sinon {retirer identifiant de requete
8 retirer identifiant client
9 calcul du resulat
10 envoi de la reponse}
11 }
```

Listing 5.4 – Algorithme du serveur en mode single

5.3.3.2 Aspects côté serveur secondaire

Plusieurs préoccupations transversales sont dispersées dans le comportement du serveur en mode secondaire. Elles concernent essentiellement la gestion de la synchronisation entre le serveur primaire et secondaire.

En ce qui concerne la gestion de la synchronisation entre le serveur primaire et le serveur secondaire, à l'arrivée d'un message provenant du primaire deux cas sont possibles :

– il s'agit de la propagation de la requête sans le résultat associé.
– il s'agit de la propagation d'un point de reprise.

La gestion de ces deux comportements relatifs à la synchronisation est encapsulé dans un aspect appelé *ACheckPoint*. Il s'agit du même aspect utilisé dans le cadre du serveur primaire. Nous ajoutons deux greffons à ce dernier *ACheckPoint.putInCache* et *ACheckPoint.updateCache*. Ces deux greffons ont les rôles suivants :

– **ACheckPoint.putInCache** : stocke la requête reçue dans une structure de données.
– **ACheckPoint.updateCache** : stocke le point de reprise dans une structure de données.
– **ACheckPoint.updateState** : met à jour l'état du serveur grâce aux informations contenues dans le point de reprise.

La structure de données considérée est décrite dans la figure 5.4. Il s'agit d'un tableau. Ce tableau contient trois colonnes qui servent à stocker les informations concernant des requêtes reçues :

1. l'identifiant du client qui a envoyé la requête,
2. le numéro de séquence de la requête,
3. la réponse associée à la requête.

Le greffon *ACheckPoint.putInCache* est exécuté dans le cas où il s'agit de la propagation de la requête et non du point de reprise. Dans ce cas, seules les colonnes correspondant à l'identifiant du client et à l'identifiant de la requête sont remplies après lors de la création d'une nouvelle ligne dans la table.

Le greffon *ACheckPoint.updateCache* est appliqué aussi lorsqu'il s'agit de la propagation d'un point de reprise. Dans ce cas, seule la colonne résultat est mise à jour et les différents éléments composant l'état du serveur sont mis à jour.

5.3.3.3 Aspects côté serveur en mode *single*

Plusieurs préoccupations transversales composent le comportement du secondaire basculant en mode primaire (*crash* du serveur primaire).

Les greffons de suppression d'identifiant client et de numéro de séquence de requête ont déjà été présentés dans le cadre du serveur primaire et secondaire. Ces greffons, *ANumbering.insert* et *ANumbering.remove* sont réutilisés ici. Ils sont appliqués dans le deuxième cas de figure abordé ci-dessus.

Dans le premier cas de figure lorsque la requête a déjà été exécutée, la récupération du résultat correspondant et son envoi sont des préoccupations transversales. Ces dernières sont encapsulées dans un aspect appelé *ACheckDuplicate* contenant un greffon nommé *ACheckDuplicate.check*. Le rôle de ce greffon est le suivant :

– **ACheckDuplicate.reply** : gère le cas où la requête est dupliquée. Si la requête est dupliquée et si le résultat de la requête est retrouvé dans le cache, alors la méthode *send* est invoquée avec en paramètre le résultat récupéré dans le cache. Si la réponse n'est pas dans le cache, alors cette requête doit être traitée.

5.4 Composition du protocole de réplication

Découper un logiciel en modules indépendants qui sont conçus et développés séparément, implique une phase de composition. C'est cette phase que nous décrivons ici. Dans les sections précédentes nous avons présenté les différents aspects qui composent le protocole duplex. Nous présentons maintenant leur intégration en vue d'obtenir par composition des comportements de chaque aspect, le comportement du protocole duplex.

Nous détaillons dans un premier temps les propriétés et les bénéfices de la décomposition en aspects à grain fin du protocole duplex. Nous discutons notamment de la séparation des préoccupations qui résulte du découpage à grain fin, et de la capacité à composer ces aspects de manière transparente avec notre système. Nous discutons également trois points importants dans le cadre de cette thèse, la réutilisabilité et la configurabilité, la composabilité.

Nous illustrons ensuite ces propriétés en présentant l'intégration des différents aspects développés à un système de base considéré dans le cadre de notre étude de cas.

5.4.1 Propriétés de l'implémentation

L'implémentation par des aspects à grain fin des différentes préoccupations transversales composant le protocole duplex offre plusieurs bénéfices.

Le premier est celui d'une séparation des préoccupations fonctionnelles et non fonctionnelles mais également des différentes préoccupations entre elles. Ainsi nous disposons d'un aspect pour la gestion de point de reprise, d'un autre pour la vérification de la duplication des messages, etc. Comparée à une implémentation traditionnelle, nous sommes maintenant en mesure d'effectuer des changements du code fonctionnel sans que ces derniers impactent le code non fonctionnel et vice versa. Nous avons donc une indépendance entre le code fournissant la tolérance aux fautes et le code applicatif.

La séparation des préoccupations à une influence directe sur d'autres propriétés. Le code fonctionnel et non fonctionnel sont clairement séparés, ces éléments étant encapsulés dans des entités logicielles distinctes. Ils sont tous les deux réutilisables.

Nous pouvons constater que certains aspects sont réutilisables à la fois pour l'implémentation du logiciel du serveur primaire, du secondaire et du client, mais aussi pour la gestion d'un mode de fonctionnement à l'autre. Les deux logiciels serveur sont déployés sur deux sites physiques distincts. Dans chacun de ces logiciels les aspects que nous avons présentés ci-dessus sont tissés afin de réaliser le comportement du serveur primaire et secondaire du protocole duplex. Il y a donc un réel avantage en terme d'effort de développement puisque les préoccupations liées au protocole duplex sont factorisées dans des aspects et réutilisés pour implémenter les logiciels du primaire et du secondaire. Nous pouvons également constater que certains aspects sont réutilisés dans différents modes. Par exemple les aspects *ANumbering.remove* et *AIdentifier.remove* sont utilisés pour implémenter le fonctionnement du serveur en mode primaire et réutilisés pour implémenter le fonctionnement du serveur en mode secondaire.

La capacité de pouvoir réutiliser des aspects permet de disposer d'une certaine latitude dans la composition des stratégies de tolérance aux fautes. Par exemple lors du passage d'une stratégie de tolérance aux fautes à une autre, la conservation de certains mécanismes présents dans l'ancienne stratégie pour les utiliser au sein de la nouvelle stratégie à mettre en œuvre

est un avantage intéressant. La reutilisabilité a donc une conséquence directe sur deux points cruciaux en terme de résilience, la configurabilité et la composabilité, comme cela est montré dans [70].

La configurabilité consiste en la capacité de choisir les mécanismes de tolérance aux fautes à appliquer à un code applicatif donné. Dans le meilleur des cas, cette configurabilité doit être obtenue à grain fin. La configurabilité permet de faire face à la variabilité. Ainsi au lieu d'avoir un mécanisme de type duplex *primary backup* nous pouvons à moindre frais obtenir un mécanisme de réplication de type *leader follower* en n'utilisant pas l'aspect *ACheckPoint* et en modifiant la synchronisation des répliques.

Grâce à notre librairie d'aspects, le protocole duplex peut être composé à partir de différents mécanismes. La composabilité est une qualité nécessaire à la construction d'une stratégie de tolérance aux fautes. Une approche à grain fin favorise la composabilité. Par exemple un mécanisme de réplication côté serveur primaire est composé d'un greffon (*ACheckPoint.forward*) qui propage la requête, de deux autres greffons qui préparent la requête pour l'interprétation côté serveur et enfin d'un greffon construisant et envoyant un point de reprise.

Tous les aspects développés peuvent être ajoutés au système de manière transparente. Grâce à la construction de "point de coupe" aucune modification du système n'est nécessaire. Notons cependant que de bonnes pratiques sont requises pour l'écriture des points de coupe afin conserver l'indépendance entre les aspects et le système dans lequel ils sont tissés. Nous ne les détaillons pas ici nous renvoyons le lecteur au travaux décrit dans [33].

5.4.2 Composition du protocole duplex

Nous allons maintenant illustrer ces différentes propriétés à travers la description de l'intégration des aspects implémentant le protocole duplex. Ces aspects sont intégrés à une application fonctionnant comme un singleton. De la même manière le client que nous utilisons au départ est un client conçu pour communiquer avec un serveur unique non répliqué.

Nous supposons donc que pour construire notre système nous partons d'un client et d'un serveur standard. Le comportement du client est décrit en pseudo code dans le listing 5.5. Le client connait l'adresse d'un serveur (ligne 2) auquel il envoie ses requêtes (ligne 5). Puis il reçoit les réponses du serveur (ligne 6) et traite les réponses.

Le comportement du serveur est décrit dans le listing 5.6. Le serveur reçoit les requêtes des clients (ligne 2). Puis il exécute le service (ligne 3) et envoie la réponse au client (ligne 4).

En composant les différents aspects présentés dans la section 5.3 nous pouvons configurer les clients et le serveur décrit dans les listings 5.5 et 5.6 de manière à ce qu'ils réalisent le comportement requis par le protocole duplex.

Dans cette section nous discutons de la composition des aspects implémentant le protocole duplex avec un client et un serveur standard. Nous illustrons comment le comportement et la structure d'un serveur peuvent être modifiés par l'application des aspects et aboutir au comportement du protocole duplex. Nous terminons en mettant en avant le séquencement choisi pour composer le protocole duplex à partir des différents aspects. Nous mettons en évidence les erreurs qui peuvent être introduites dans ce séquencement.

```
variable du protocole:
adresse serveur

Boucle client {
envoi de la requete au serveur
reception de la  reponse
traitement de la reponse
}
```

Listing 5.5 – Algorithme du client

```
Boucle serveur {
reception de la  requete
execution de la  requete
envoi de la reponse au client
}
```

Listing 5.6 – Algorithme du serveur

5.4.2.1 Introduction inter-type

Les aspects que nous avons identifiés dans la section précédente modifient le comportement des classes où ils sont tissés. Dans le cadre du protocole duplex il est également nécessaire de modifier la structure des classes du système. Par exemple, le client ne possède qu'une seule adresse de serveur. Il est nécessaire d'ajouter une variable d'adresse secondaire en utilisant le mécanisme d'introduction, c'est-à-dire un attribut qui correspond à l'adresse du serveur secondaire.

Du côté serveur, l'attribut mode n'existe pas dans le listing 5.6. La gestion du mode du côté du serveur primaire et du côté serveur secondaire nécessite que ce serveur possède un attribut *mode* sur lequel le serveur peut raisonner afin de savoir quel comportement à adopter. Puisque cet attribut n'existe pas par défaut dans le code du serveur il faut le créer. Ce dernier peut prendre plusieurs valeurs : *PRIMARYDUPLEX*, *SECONDARYDUPLEX*, *PRIMARYSINGLE*. Ces trois constantes sont visibles dans le listing 5.7 ligne 3. Pour chaque réplique de serveur le mode est diffère. L'attribut mode du serveur primaire voit sa valeur initialisée à *PRIMARYDUPLEX*. Le serveur secondaire sera initialisé à *SECONDARYDUPLEX*. La valeur *PRIMARYSINGLE* est utilisée lorsque le serveur passe en mode single c'est-à-dire que le serveur primaire est défaillant et que le serveur secondaire traite les requêtes.

```
1 public aspect AIMode {
2       enum Mode {
3            PRIMARYDUPLEX, SECONDARYDUPLEX, PRIMARYSINGLE
4       }
5        public Mode Serveur.mode=Mode.PRIMARY_DUPLEX;
6 }
```

Listing 5.7 – Introduction pour la gestion du mode

5.4.2.2 Aspects du protocole duplex et resolver

A cette étape, les dépendances entre les aspects et les données que leur utilisation requiert sont résolues. Nous pouvons maintenant les intégrer à l'application. Nous détaillons l'intégration des aspects en deux étapes : l'intégration côté client, puis côté serveur, le logiciel du serveur

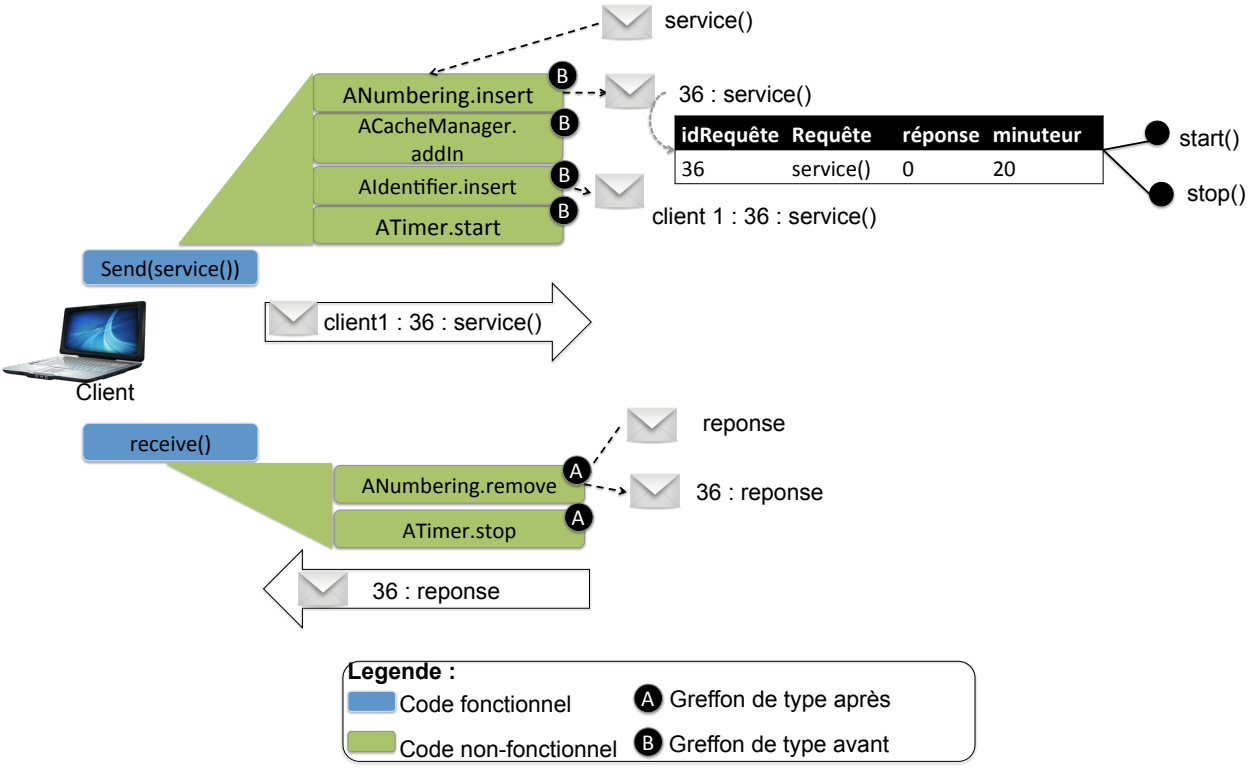


FIGURE 5.2 – Organisation des aspects coté client

primaire et secondaire étant identique.

Intégration des aspects au client

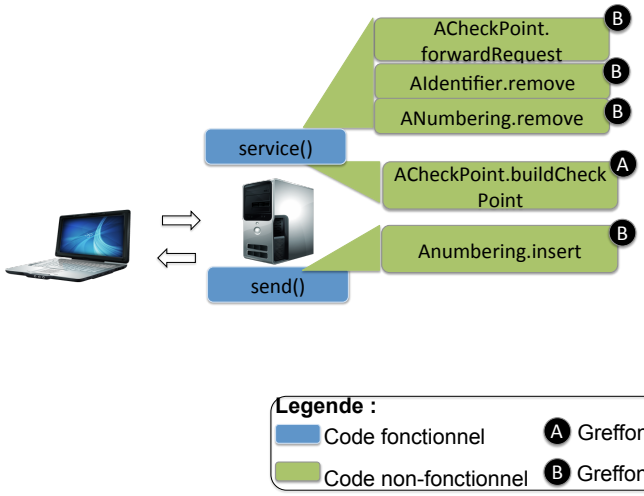
Du coté client les aspects *ANumbering.insert, ACacheManager.addIn, ATimer.start, AIdentifier.insert*, sont appliqués au même point de jonction, c'est-à-dire avant l'appel à la méthode *send()*. Puisque nous avons une interaction, il est nécessaire de déterminer un ordre de séquencement de ces aspects.

Ces greffons sont tous de type "*avant*". L'ordre d'application des greffons est le suivant : *AIdentifier.insert* > *ACacheManager.addIn* > *ANumbering.insert* > *ATimer.start*. Cette précédence est définie par un resolver dont le code est présenté dans le listing 5.8.

```
1 aspect sendResolution {
2 void resolver sender():
3 and(AIdentifier.insert, ACacheManager.addIn, ATimer.start, ANumbering.
      insert){
4 [ANumbering.insert,  ACacheManager.addIn, AIdentifier.insert, ATimer.
      start, ].proceed();}}
```

Listing 5.8 – Résolution des interactions autour du point de jonction *send()*

Une résolution doit également être proposée pour le point de jonction *receive()*. À ce point de jonction les greffons *ATimer.stop, ANumbering.remove* sont appliqués. L'ordre de précédence de ces greffons est le suivant : *receive()* < *ANumbering.remove* < *ATimer.stop*. Notons que puisqu'il s'agit de greffon de type "après" la partie après de *ATimer.stop* est appliquée suivie de la



Identifiant client	Numéro de requête	résultat
client1	36	15

FIGURE 5.3 – Organisation des aspects côté serveur en mode primaire

partie après de *ANumbering.remove*. Un resolver est utilisé pour implémenter cette résolution. Ce dernier est détaillé dans le listing 5.9.

```
1 aspect receiveResolution {
2 void resolver receive():
3 and(ANumbering.remove, ATimer.stop){
4 [ANumbering.remove, ATimer.stop].proceed();}}
```

Listing 5.9 – Résolution des interactions autour du point de jonction *send()*

Le comportement obtenu est décrit dans la figure 5.2.

Intégration des aspects au serveur

Du côté du serveur l'intégration des aspects est plus complexe que du côté client. Un serveur fonctionnant sans réplication ne possède qu'un seul état. Cet état correspond à l'état *running* du diagramme d'état présenté dans la figure 5.8. L'intégration des aspects revient à décomposer l'état *running* en sous-états. Dans chaque état le serveur exécute un certain nombre d'activités. Ces activités sont décrites dans le diagramme de la figure 5.9.

Grâce à l'introduction montrée dans le listing 5.7, le serveur peut fonctionner dans l'un des trois modes, primaire, secondaire, single primaire. Chacun de ces modes correspond à un état du serveur. Dans un état donné une ou plusieurs activités sont exécutées. Dans la figure 5.9 le diagramme d'activité représente l'ensemble des actions réalisées par le système, avec tous les branchements conditionnels et toutes les boucles possibles. Il s'agit d'un graphe orienté dont les nœuds représentent des actions et les arcs les transitions entre les actions. Les transitions sont franchies lors de la fin des actions ; ici les étapes sont réalisées en séquence. Chaque branche du diagramme d'activité représente les activités exécutées dans un état. Par exemple dans l'état duplex primaire le serveur exécute les activités suivantes : propagation de la requête, suppression du numéro de séquence de requête, suppression de l'identifiant client, envoi traitement de la requête, construction et envoi du point de reprise et enfin envoi de la réponse. Chacune de ces activités correspond à du code fonctionnel ou à du code non-fonctionnel. Par exemple le traitement de la requête est un code fonctionnel. Au contraire, la construction et l'envoi du point de reprise est une préoccupation non-fonctionnelle encapsulée dans l'un de

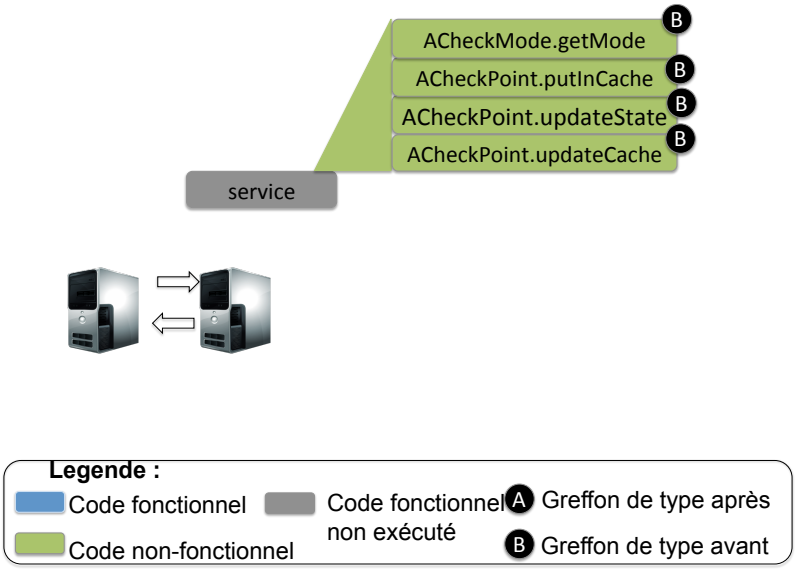


FIGURE 5.4 – Organisation des aspects dans le mode duplex secondaire

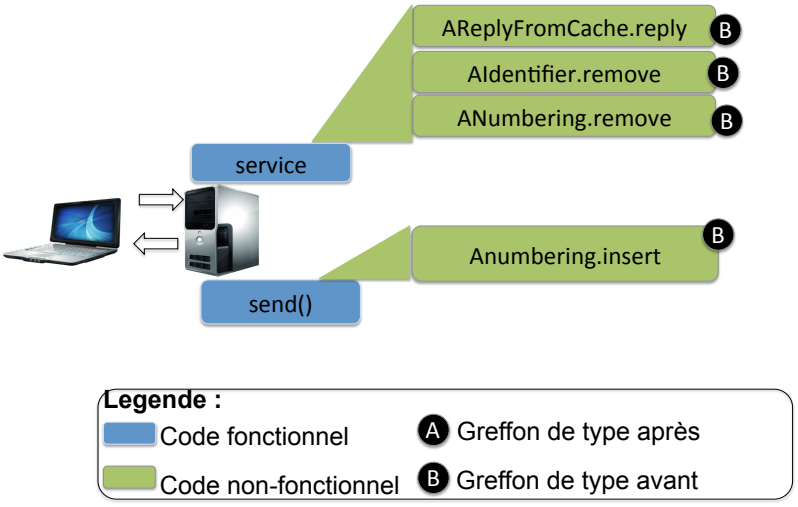


FIGURE 5.5 – Organisation des aspects dans le mode *single* primaire

nos aspects.

Afin d'obtenir ce comportement du côté serveur il est nécessaire d'utiliser une hiérarchie de resolver. Comme nous pouvons le voir sur la figure 5.9, une première décision concerne le mode de fonctionnement du serveur. Suite à cette décision plusieurs séquences d'activités peuvent être exécutées. Cette décision et l'invocation des différentes séquences d'activité peuvent être réalisées par un premier resolver. Ce resolver décrit dans la figure 5.6 est le resolver racine d'une arborescence. Selon la valeur du mode la chaîne de greffon invoquée diffère.

Tous les greffons à gérer partagent le même point de jonction et sont appliqués avant l'exécution du service. Le resolver *RCheckMode.getMode* s'applique. Selon la valeur de la variable mode la chaine de greffon à appliquer est définie dans l'un des resolvers fils. Les différents greffons gérant les greffons dans les mode primaire, secondaire et *single* primaire sont gérés par

des resolvers dédiés à chacun des modes. Par exemple les greffons exécutés dans le mode primaire sont gérés par un resolver appelé *RduplexPrimary.run*. Ces trois resolvers sont gérés par le resolver racine *RCheckMode.getMode*. Ceux du mode secondaire par le resolver *RduplexSecondary.run*. Enfin les greffons invoqués dans le mode *single* primaire sont gérés par le resolver *RsinglePrimary.run*.

Le greffon *getMode* à la vocation suivante :

- **RCheckMode.getMode** : est chargé de déterminer en lisant la variable *mode* (ligne 2 du listing 5.3) du code de base quel est le mode de fonctionnement actuel du serveur. Cette variable étant multi-valuée, le mode peut être : duplex primaire, duplex secondaire, *single* primaire. Une fois le mode déterminé, différentes séquences de greffons peuvent être appliquées. La première correspond à la séquence de greffons réalisant le comportement du serveur en mode primaire, la deuxième en mode secondaire et le troisième en mode *single* primaire.

RCheckMode.getMode est le resolver racine de l'arborescence qui gère les différents greffons appliqués au serveur. Il se positionne donc au niveau 0 de l'arbre comme indiqué dans la figure 5.6. Nous détaillons à présent successivement les fils de ce resolver. Il s'agit des resolvers de niveau 1 gérant les modes duplex primaire, duplex secondaire et *single*.

Resolver pour la gestion du mode primaire

Le mode primaire est géré par un seul resolver. Ce dernier défini la précédence suivante entre les greffons qui doivent être appliqués en mode primaire. La précédence suivante est défini entre ces greffons : *ACheckPoint.forwardRequest* < *AIdentifier.remove* < *ANumbering.remove*< service < *ACheckPoint.buildCheckPoint*. Cette chaîne de greffon correspond au sous-arbre gauche de la figure 5.6.

Resolver pour la gestion du mode secondaire

Dans le mode secondaire deux séries d'actions différentes qui peuvent être invoquées en fonction de l'état du traitement de la requête en terme de synchronisation avec le serveur primaire. Ces dernières sont encapsulées dans un aspect appelé *RduplexSecondary*. Il contient un resolver nommé *RduplexSecondary.run* contenant à la fois la logique du choix de la séquence d'action à invoquer en fonction de l'état de la synchronisation avec le primaire, et l'invocation de ces actions. Le rôle du greffon *RduplexSecondary.run* est le suivant :

- **RduplexSecondary.run** : détermine si la requête reçue est un point de reprise ou une requête propagé par le serveur primaire. S'il s'agit d'un point de reprise, les greffons *ACheckPoint.updateCache* et *ACheckPoint.updateState* sont appelés. S'il s'agit d'une propagation de requête, le greffon *ACheckPoint.putInCache* est appelé.

Resolver pour la gestion du mode *single*

En mode *single*, lors de l'appel du service, la gestion de la sémantique *only once* nécessite de pouvoir fournir deux comportements qui correspondent à :

- la gestion de la duplication des requêtes reçues et la récupération de la réponse à partir du cache.
- la gestion nominale d'une requête qui comprend la suppression de données dans la requête reçue déjà présentée dans le cas du serveur primaire.

Ces deux séquences d'actions sont gérées par un aspect nommé *RsinglePrimary*. Le greffon *run* qu'il contient vérifie que la requête est reçue pour la première fois. Il s'appuie sur la structure de cache présentée précédemment. La structure de cache est implémentée sous forme de table. La requête est retrouvée grâce à son numéro de séquence. Deux cas sont alors possibles :

- Si celle-ci est présente dans la table et que la réponse à cette requête est également présente alors le service ne doit pas être exécuté une seconde fois (*only once*) la réponse est récupérée dans la structure de cache et renvoyée.
- Dans le cas contraire, celui où la requête est absente de la structure de cache ou bien elle est présente mais pas la réponse, le service est exécuté et la réponse est envoyée au client.

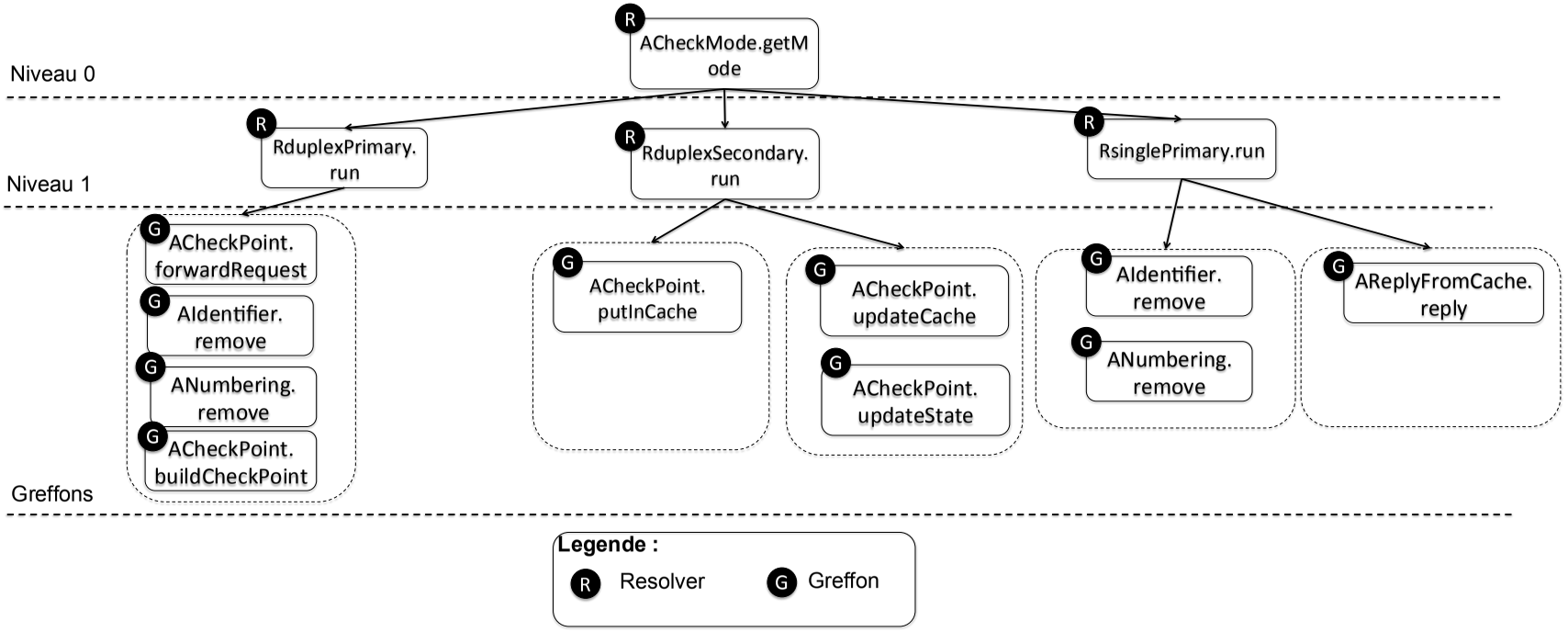


FIGURE 5.6 – Arborescence de resolvers utilisée pour la résolution des interactions coté serveur.

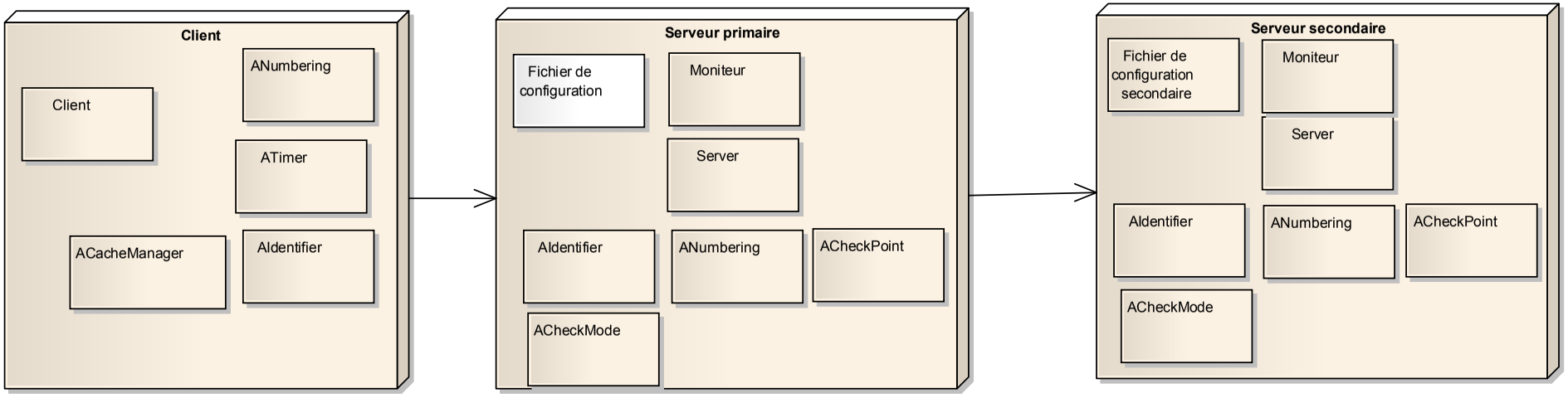


FIGURE 5.7 – Diagrame de déploiement du système munie du protocole duplex

Les greffons qui s'appliquent dans chacun des modes sont gérés par les resolvers de niveau 1.

D'un point de plus macroscopique le déploiement des aspects peut être représenté par un diagramme de déploiement. Le diagramme de déploiement décrit l'implantation physique de notre système. D'une manière générale, un diagramme de déploiement montre la configuration physique des différents composants qui participent à l'exécution du système, ainsi que les artefacts qu'ils supportent. Ce diagramme est constitué de nœuds connectés par des liens physiques. Les nœuds peuvent contenir des artefacts.

Dans notre cas, les différents nœuds représentent différentes machines virtuelles composés d'artefacts qui sont des ensembles de classes et des aspects. Le diagramme de déploiement de la figure 5.7 représente le déploiement de notre système. Nous pouvons distinguer trois nœuds : un nœud client, deux nœuds où sont localisés un serveur et un moniteur. Le nœud client contient l'application client et les aspects *ANumbering*, *AIdentifier*, *ATimer*, *ACacheManager*.

Les nœuds serveur sont composés du logiciel serveur et des aspects permettant de réaliser le comportement du protocole duplex ainsi que d'un fichier de configuration permettant d'initialiser le serveur en mode primaire ou secondaire. Enfin deux moniteurs sont utilisés pour la surveillance de l'activité du nœud serveur et la détection des *crashs*. Les nœuds client et serveur communiquent entre eux pour le traitement des requêtes. Les nœuds serveur communiquent via un lien pour se synchroniser. Les moniteurs se communiquent mutuellement des données concernant l'activité des serveurs dont ils ont la charge.

La compilation et le déploiement des artefacts sont réalisés par un script ANT. Ce script contient deux tâches *compilePrimary* et *compileSecondary*. Ces deux tâches réalisent la même commande de compilation. La première ajoute à l'archive produite à la compilation un fichier de configuration spécifiant que l'archive produite est celle du serveur primaire. La deuxième indique que l'archive est celle du secondaire. Ces fichiers sont lus au démarrage du serveur pour déterminer son mode.

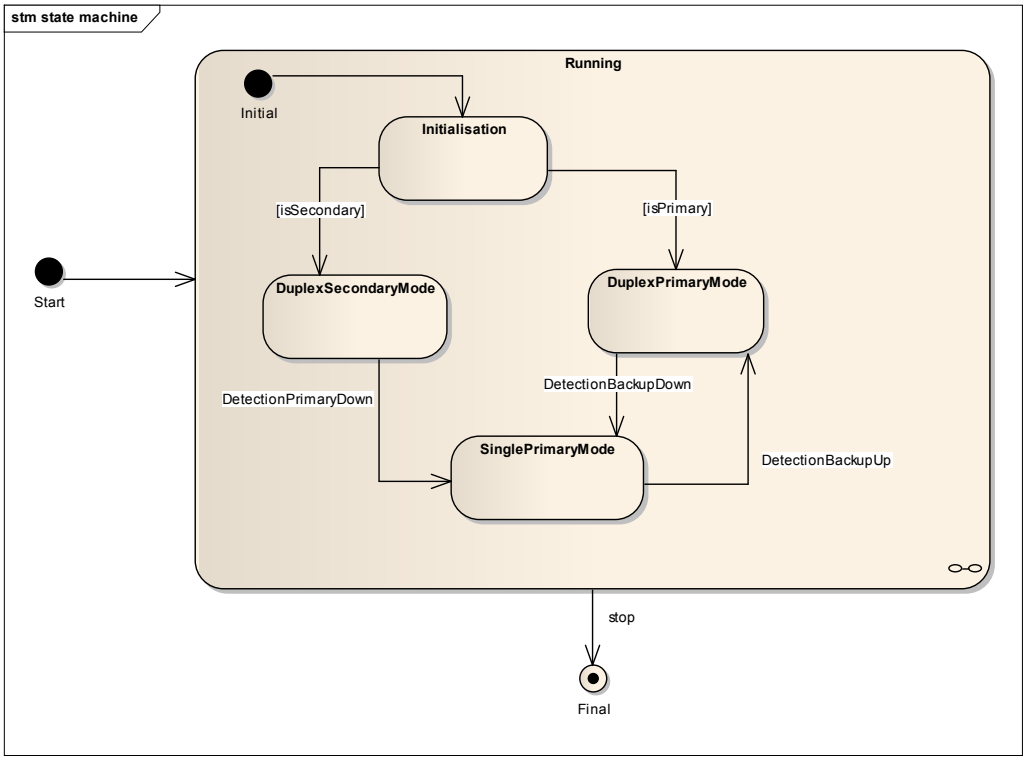


FIGURE 5.8 – Diagramme d'états du système tissé

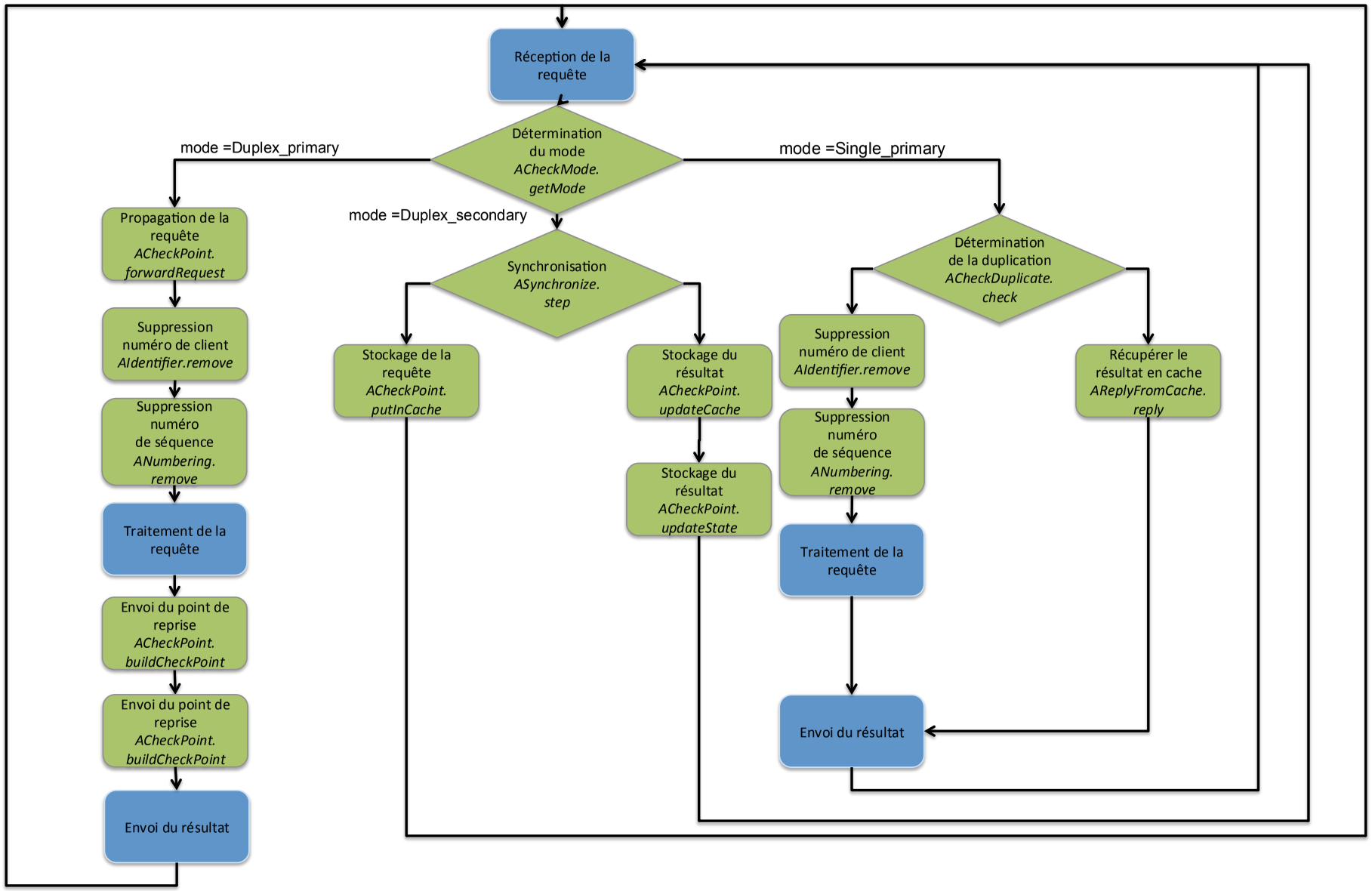


FIGURE 5.9 – Diagramme d'activité du serveur en mode primaire, secondaire et unique

5.4.3 Discussion

Dans cette section nous avons présenté l'intégration des différents aspects composant le protocole duplex. Cette intégration se fait en insérant ces aspects à différents points de jonction. Plusieurs points de jonction s'avèrent être partagés par plusieurs greffons. La résolution de ces interactions est réalisée à l'aide de resolvers, voire de resolvers de resolvers. Nous avons donc recours à un séquencement complexe des greffons lors de leur intégration au système. De nombreuses erreurs liées à des interférences sont apparues lors de l'intégration des greffons. Pour la plupart d'entre elles, il a été difficile de les repérer et de les corriger puisqu'elles sont très difficilement traçables. Dans la section suivante, nous présentons une application de notre approche de détection d'interférences. Nous illustrons son application à notre étude de cas dans plusieurs scénarios et nous montrons comment elle permet de tracer facilement les interférences qui se produisent.

5.5 Spécification des propriétés attendues

Lors de l'intégration des aspects composant le protocole duplex, des hypothèses sur les propriétés de la composition sont faites de manière implicite. Conformément à notre discussion au chapitre 3, il est nécessaire que l'intégrateur rende ces propriétés explicites. Puis ces propriétés sont transformées en assertions exécutables afin de pouvoir les vérifier à l'exécution. Dans cette section nous présentons un exemple d'application de la procédure de spécification abordée dans le chapitre précédent, puis la spécification de chacun des aspects pour le nœud client, le serveur primaire et le serveur secondaire.

5.5.1 Processus de spécification

Nous présentons maintenant le processus permettant de spécifier les propriétés en termes d'évitement des interférences CB, CA, IB, IA pour les aspects composant le protocole duplex. La spécification est produite en suivant le processus interactif décrit dans [38].

Nous ne détaillons pas systématiquement les différentes étapes du processus de spécification interactive pour chacun des greffons composant le protocole duplex. Nous présentons ici un exemple d'application de ce processus. Notons que la spécification des autres greffons est dérivée de la même manière. Nous utilisons le greffon *ANumbering.insert* côté client pour illustrer la procédure de spécification. Les questions ci-dessous sont posées à l'intégrateur dans le cadre de la spécification des propriétés attendues de la composition des greffons autour d'un point de jonction. Ici il s'agit du point de jonction correspondant à l'appel à la méthode *send()*. Pour chacune des questions nous fournissons la réponse correspondant aux propriétés que nous attendons de *ANumbering.insert*. Si la réponse à une question est positive, nous présentons également la propriété qui doit être attachée *ANumbering.insert*. Il s'agit finalement de construire pour chaque greffon un triplet (Q,R,P) ou Q est la question posée, R est la réponse donnée et P la propriété associée au greffon si la réponse est positive.

Q. 1 : Y a t-il des variables d'entrée de *ANumbering.insert* dont la valeur doit être préservée depuis l'arrivée au point de jonction jusqu'à l'exécution de *ANumbering.insert* ?

R. 1 : Non.

Q. 2 : Y a t-il des variables d'état du système dont la valeur doit être préservée après l'exécution de *ANumbering.insert* ?

R. 2 : Oui. La variable *msgContent* doit être préservée après l'exécution de *ANumbering.insert*.

P. 2 : L'absence d'interférence CA doit être vérifiée pour la variable *msgContent* utilisée par le greffon *ANumbering.insert*.

Q. 3 : Est-ce une erreur si les greffons exécutées avant *ANumbering.insert* invalident la condition d'application de *ANumbering.insert* ?

R. 3 : Non.

Q. 4 : Est-ce une erreur si le greffon *ANumbering.insert* est exécuté, mais que l'envoi du message supposé suivre son exécution ne se produit pas ? Par exemple, parce que le flot de contrôle a été modifié par d'autres greffons.

R. 4 : Non.

Dans le cas du greffon *ANumbering.insert* une seule propriété de non interférence doit être vérifiée. Puisque la réponse à la question *Q. 2* est positive (*R. 2*). Une détection de CA est attachée au greffon (*P. 2*).

Ce processus de spécification est appliqué à chacun des aspects appliqués à un point de jonction partagé. Nous détaillons maintenant les différentes propriétés spécifiées pour chacun des greffons composant le protocole duplex. La vérification des propriétés spécifiées pour chaque greffon est nécessaire pour la réalisation du comportement du protocole duplex.

5.5.2 Spécification des aspects côté client

Nous présentons maintenant les propriétés en termes d'évitement des interférences CB, CA, IB, IA pour les aspects côté client. La vérification des propriétés spécifiées pour chaque greffon est nécessaire pour que le comportement du client soit correctement réalisé.

TABLE 5.1 – Spécification des interférences interdites pour les greffons coté client

Greffon	Interférence interdites	Variables
AIdentifier.insert	CA	*msgContent*
ANumbering.insert	CA	*msgContent*
ACacheManager.addIn	IA	$\emptyset$
ATimer.start	IA	$\emptyset$

Le greffon *ANumbering.insert* ajoute un numéro de séquence dans la requête envoyée au serveur. Nous devons garantir qu'après l'application de *ANumbering.insert* aucun greffon ne modifie la chaîne de caractères *msgContent* représentant la requête (interférence de type CA, *Change After*). Une détection de CA doit donc être visée pour le greffon *ANumbering.insert* par rapport à la variable *msgContent*. Cette propriété peut être énoncée comme suit :

Détection de CA : il n'y a pas de modification de la variable du code de base *msgContent* utilisée par *ANumbering.insert*, par un autre aspect exécuté après *ANumbering.insert*

Le greffon *ACacheManager.addIn* ajoute dans une structure de données la requête à envoyer et une référence sur le minuteur associé. Le minuteur est initialisé à la durée après laquelle le serveur primaire est considéré comme défaillant après l'envoi d'une requête. Si le minuteur est lancé sans que la requête soit envoyée, par exemple à cause d'une interférence de type IA (*Invalidation After*), le comportement obtenu n'est pas correct. Nous devons vérifier une propriété de type IA sur le greffon *ACacheManager.addIn*. Celle-ci peut être énoncé comme suit :

Détection de IA : Le point de jonction jp_i doit toujours être exécuté après *ACacheManager.addIn*.

Le tableau 5.1 récapitule l'ensemble des propriétés de non interférence attendues pour chaque greffon appliqués côté client.

5.5.3 Spécification des aspects côté serveur primaire

Nous allons à maintenant présenter la spécification de chacun de ces aspects dans le cadre de leur utilisation du côté du serveur primaire. Le greffon *AForwardRequest.forward* doit vérifier une propriété CB :

Détection de CB : Il n'y a pas de modification de la variable du code de base *msgContent* utilisé par *AForwardRequest.forward*, par un autre aspect exécuté avant *forward*.

Le greffon *ACheckPoint.buildCheckPoint* doit vérifier une propriété CB :

Détection de CB : Il n'y a pas de modification de la variable du code de base *result* utilisé par *buildCheckPoint*, par un autre aspect exécuté avant *buildCheckPoint*.

Le greffon *ACheckPoint.buildCheckPoint* doit vérifier une propriété CA :

Détection de CA : Il n'y a pas de modification de la variable du code de base *result* utilisé par *buildCheckPoint*, par un autre aspect exécuté aprés *buildCheckPoint*.

Le greffon *ANumbering.remove* doit vérifier une propriété CA :

Détection de CA : Il n'y a pas de modification de la variable du code de base *msgContent* utilisé par *remove*, par un autre aspect exécuté après *remove*.

Le greffon *AIdentifier.remove* doit vérifier une propriété CB :

Détection de CB : Il n'y a pas de modification de la variable du code de base *msgContent* utilisé par *remove*, par un autre aspect exécuté avant *remove*.

TABLE 5.2 – Propriétés de non-interférence pour les aspects côté primaire

Greffon	Interférence interdites	Variables
AForwardRequest.forward	CB, IA	*msgContent*
ACheckPoint.buildCheckPoint	CB, CA	*result*
ANumbering.remove	CA	*msgContent*
AIdentifier.remove	CB	*msgContent*

5.5.4 Spécification des aspects côté serveur secondaire

Nous présentons maintenant les propriétés en termes d'évitement des interférences CB, CA, IB, IA pour les aspects côté serveur secondaire.

Le greffon *ACheckPoint.putInCache* ajoute les informations contenues dans une requête reçue dans une structure de données. Nous devons garantir qu'avant l'application de *ACheckPoint.putInCache* aucun greffon ne modifie la chaîne de caractères *msgContent* représentant la requête (interférence de type CB, *Change Before*). Une propriété CB (*Avoidance of Change Before*) doit donc être vérifiée pour le greffon *ACheckPoint.putInCache* par rapport à la variable *msgContent*. Ces propriétés peuvent être énoncées comme suit :

Détection de CB : il n'y a pas de modification de la variable du code de base *msgContent* utilisée par *ACheckPoint.putInCache*, par un autre aspect exécuté avant *ACheckPoint.putInCache*.

Le greffon *ACheckPoint.updateCache* met à jour les informations liées à une requête grâce aux informations contenues dans le point de reprise. Nous devons garantir qu'avant l'application de *ACheckPoint.updateCache* aucun greffon ne modifie la chaîne de caractères *msgContent* représentant la requête (interférence de type CB, *Change Before*). Une propriété CB (*Avoidance of Change Before*) doit donc être vérifiée pour le greffon *ACheckPoint.updateCache* par rapport à la variable *msgContent*. Cette propriété peut être énoncée comme suit :

Détection de CB : il n'y a pas de modification de la variable du code de base *msgContent* utilisée par *ACheckPoint.putInCache*, par un autre aspect exécuté avant *ACheckPoint.putInCache*.

Le greffon *ACheckPoint.updateState* met à jour l'état du serveur grâce aux informations contenues dans le point de reprise. Nous devons garantir qu'avant l'application de *ACheckPoint.updateState* aucun greffon ne modifie la chaîne de caractères *msgContent* représentant

la requête (interférence de type CB, *Change Before*). Une propriété CB (*Change Before*) doit donc être vérifiée pour le greffon *ACheckPoint.updateState* par rapport à la variable *msgContent*. Cette propriété peut être énoncée comme suit :

Détection de CB : il n'y a pas de modification de la variable du code de base *msgContent* utilisée par *ACheckPoint.putInCache*, par un autre aspect exécuté avant *ACheckPoint.putInCache*.

TABLE 5.3 – Propriétés de non-interférence pour les aspects côté secondaire

Greffon	Interférence interdites	Variables
ACheckPoint.putInCache	CB	*msgContent*
ACheckPoint.updateCache	CB	*msgContent*
ACheckPoint.updateState	CB	*msgContent*

5.6 Détection des interférences à l'assemblage

Cette section présente une liste des différents cas d'interférences détectés à l'intégration des aspects. Chaque interférence a été introduite dans la chaine d'aspects du côté client, serveur primaire, serveur secondaire par inadvertance. Elle relève toutes du "vécu" et m'ont causé quelques soucis.

5.6.1 Détection d'interférences côté serveur primaire

Du côté serveur, dans la gestion du mode primaire plusieurs aspects sont appliqués avant le point de jonction correspondant à l'exécution du service. Dans la section précédente nous avons présenté la spécification de chacun de ces aspects. Nous détaillons maintenant les interférences qui se sont produites lors de l'intégration de ces aspects dans un ordre qui viole la spécification d'un aspect.

Les greffons *ACheckMode.getMode*, *AForwardRequest.forward*, *ACheckPoint.buildCheckPoint*, *ANumbering.remove*, *AIdentifier.remove* sont appliqués avant l'exécution du service. L'ordre d'application permettant de réaliser le comportement attendu du protocole duplex et de réaliser la spécification de chacun des aspects est *ACheckMode.getMode* < *AForwardRequest.forward* < *ANumbering.remove* < *AIdentifier.remove* < *service* <*ACheckPoint.buildCheckPoint*. Nous supposons ici que le serveur est en mode primaire. Cette séquence d'aspects a pour conséquence de transmettre la requête au serveur secondaire, de préparer le traitement de la requête, d'exécuter le service. Enfin une fois le service exécuté, un point de reprise est construit et envoyé au serveur secondaire.

Cependant, l'intégrateur peut introduire une faute en appliquant les aspects dans l'ordre suivant : *ACheckMode.getMode* < *ANumbering.remove* < *AIdentifier.remove* < *AForwardRequest.forward* < *service* <*ACheckPoint.buildCheckPoint*.

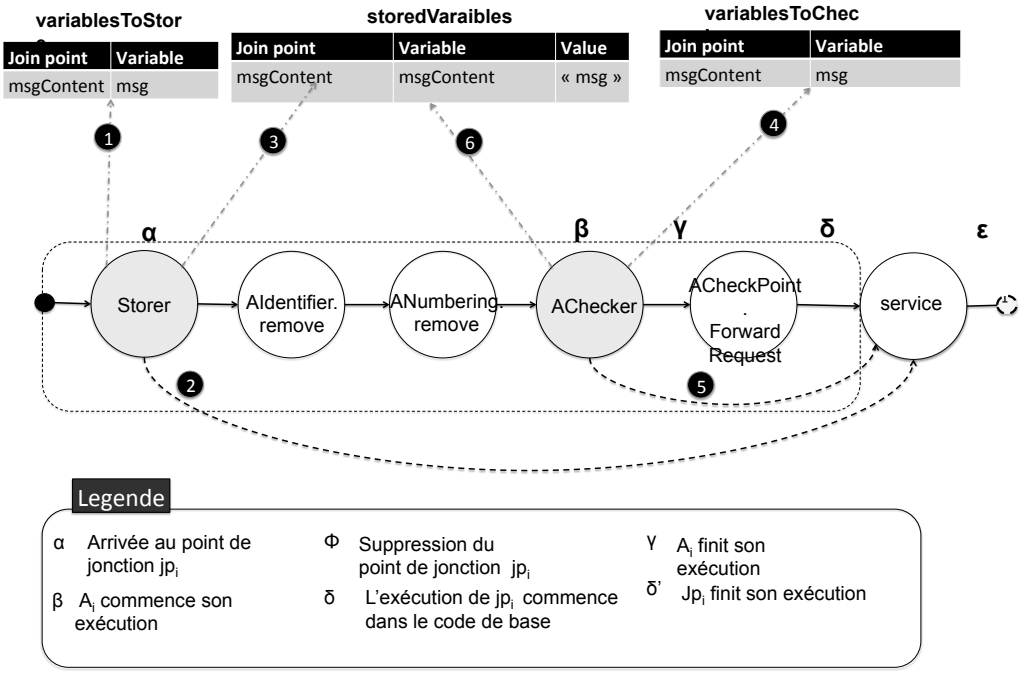


FIGURE 5.10 – Interférence de type CB entre le greffon *Anumbering.remove* et le greffon *ACheck-Point.forward*

Il y a une modification de la variable du code de base *msgContent* utilisé par *forward*, par un autre aspect exécuté avant *forward*. Cette interaction viole la spécification du greffon. Le comportement induit par l'ordre d'exécution des aspects est problématique dans la mesure où le contenu de la requête est modifié avant l'envoi au serveur par les aspects *ANumbering.remove* et *AIdentifier.remove*. Du côté du serveur secondaire la requête arrive sans identifiant de requête ni de client, ce qui induit des erreurs dans la récupération de la requête.

Du côté du secondaire l'aspect *ACheckPoint.buildCheckPoint* s'appuie sur ces identifiants pour stocker dans une structure de données l'identifiant client, le numéro de séquence de requête, requête et le résultat et mettre à jour l'état. En l'absence d'identifiant dans la requête *ACheckDuplicate.check* ne peut pas décider si la requête est dupliquée ou s'il s'agit d'une mise à jour de l'état par un point de reprise. En l'absence d'identifiant aucune action correcte n'est entreprise du côté du serveur secondaire.

La figure 5.10 illustre cette interférence. Lors de l'activation de la chaîne d'aspect, lors de la transition α la valeur de la requête est capturée et sauvegardée dans une structure de données par un greffon dédié. Avant l'exécution de *ACheckPoint.buildCheckPoint*, les greffons *ANumbering.remove* et *AIdentifier.remove* sont exécutés. Ces deux greffons modifient la valeur de la requête. Lors de la transition β le greffon *AChecker* compare la valeur au moment de l'activation de la chaine d'aspect (α) et la valeur actuelle à β. Le contenu de la requête ayant été modifiée le greffon *AChecker* écrit alors dans le fichier de trace *CB (core.server.service().java, msgContent)*, ce qui signifie qu'une interférence de type CB a été détectée autour de l'appel à la méthode *service* sur la classe *server*.

5.6.2 Détection d'interférences côté client

Du côté client, la gestion du recouvrement est géré par des greffons appliqués avant les points de jonction *send()* et *receive()*. Nous détaillons maintenant une interférence qui s'est produite lors de l'intégration de ces aspects dans un ordre qui viole la spécification des greffons présentée dans la section précédente et qui a requis d'élargir le spectre des interférences auquel nous nous sommes initialement intéressé.

Les greffons *ANumbering.insert*, *AIdentifier.insert*, *ACacheManager.addIn* sont appliqués avant l'exécution de la méthode *send()*. L'ordre d'application permettant de réaliser le comportement attendu est *ANumbering.insert* > *ACacheManager.addIn* > *AIdentifier.insert*. Cette séquence d'aspects ajoute à la requête un numéro de séquence, la sauvegarde à des fins de recouvrement, ajoute un numéro de client à la requête.

L'intégrateur peut introduire une faute en appliquant les aspects dans l'ordre suivant : *ANumbering.insert* > *AIdentifier.insert* > *ACacheManager.addIn*.

Dans ce cas il y a une modification de la variable du code de base *msgContent* utilisé par *ACacheManager.addIn*, par *AIdentifier.insert* exécuté avant *ACacheManager.addIn*. Cette interaction viole la spécification du greffon. Le comportement induit par l'ordre d'exécution des aspects est problématique dans la mesure où le contenu de la requête est modifié avant *ACacheManager.addIn*.

Le problème vient du fait que la requête en s'appuie sur le délimiteur inséré par le greffon *ANumbering.insert* pour la stoker dans une structure de données. Par exemple, comme l'illustre la partie supérieure de la figure 5.11, pour invoquer une commande appelée *service()* sur le serveur, la requête initiale est la suivante : "service()". Le greffon *ANumbering.insert* insère un numéro de séquence de requête ce qui transforme la requête comme suit : "36 : service()". Le greffon *ACacheManager.addIn* récupère ensuite le premier élément à gauche du caractère " :" pour initialiser un champ d'une structure de données correspondant au numéro de séquence de la requête. L'élément à droite du caractère " :" est utilisé pour initialiser un champ correspondant au contenu de la requête.

Analysons maintenant le cas ou le greffon *AIdentifier.insert* est appliqué après *ANumbering.insert* comme illustré dans la partie inférieure de la figure 5.11 ; le contenu de la requête est alors : "client1 : 36 : service()". Le numéro de séquence de requête stockée est "client1" et le contenu de la requête stocké est "36".

Les conséquences de cette erreur se propagent alors comme suit :

- Du côté du serveur primaire, la requête "36" ne peut pas être interprétée, car elle ne correspond pas au nom d'un service fournit par la serveur.
- Du côté du serveur primaire, pour la requête "36", le serveur ne fournit pas de réponse, il est considéré comme défaillant. Le recouvrement est lancé. Le client renvoie le requête au secondaire devenu primaire. Ici encore la requête n'est pas interprétée.

La figure 5.12 illustre cette interférence. Après l'activation de *ANumbering.insert*, lors de la transition α la valeur de la requête est capturée et sauvegardée dans une structure de données par un greffon dédié. Avant l'exécution de *ACheckPoint.buildCheckPoint*, les greffons *ANumbering.remove* et *AIdentifier.remove* sont exécutés. Ces deux greffons modifient la valeur de la requête. Lors de la transition β le greffon *AChecker* compare la valeur au moment de l'activation de la chaine d'aspect (α) et la valeur actuelle à β. Le contenu de la requête ayant été modifiée

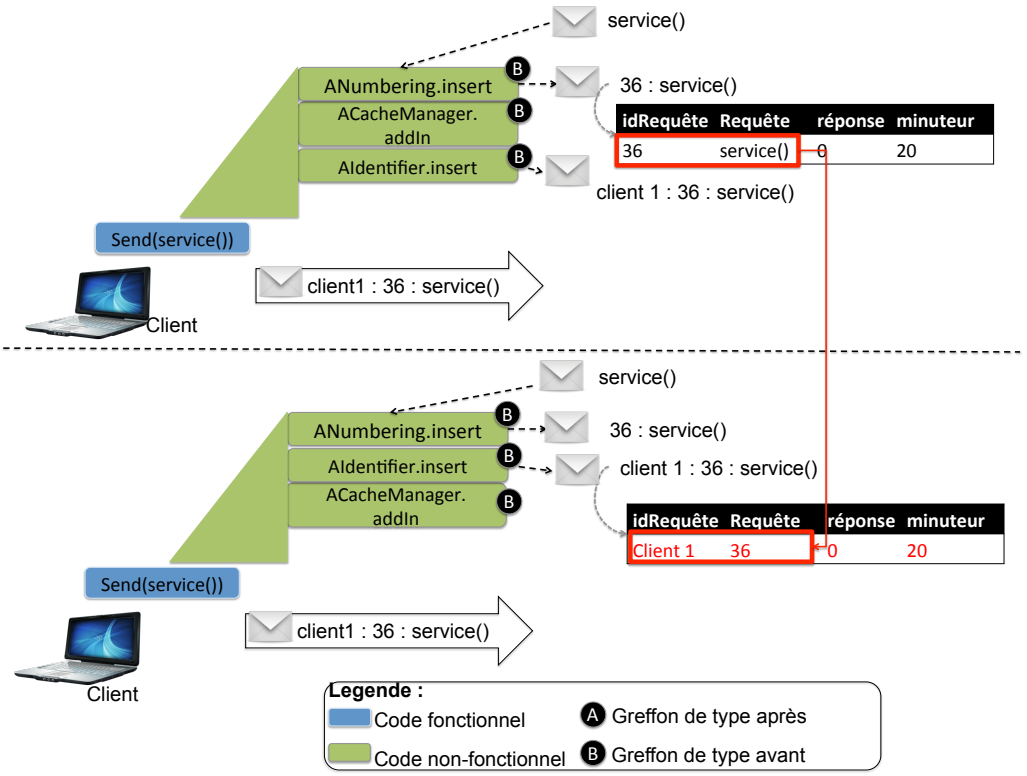


FIGURE 5.11 – Illustration de l'interaction entre les greffons *ANumbering.insert, ACacheManager.addIn, AIdentifier.insert* deux ordres de prècèdence

le greffon *AChecker* écrit alors dans le fichier de trace *CB (service, msgContent)*, ce qui signifie qu'une interférence de type CB a été détectée autour de l'appel à la méthode *service* sur la classe *server*.

Cependant, même en implémentant un resolver réalisant l'ordre de précédence escompté il s'avère que la vérification de l'évitement d'une interférence de type CA sur le greffon *ANumbering.insert* est problématique. En appliquant les greffons dans l'ordre *ANumbering.insert* > *ACacheManager.addIn* > *AIdentifier.insert* l'instrumentation pour la détection de CA est activée alors qu'il s'agit bien du comportement attendu de la composition.

Nous avons dû spécifier plus finement le comportement désirée de la composition de ces greffons. Deux éléments sont à distinguer dans le comportement attendu :

- le premier est la vérification d'une dépendance entre *ANumbering.insert* et *ACacheManager.addIn* en effet afin que *ACacheManager.addIn* sauvegarde la requête envoyée elle doit comporter des informations préalablement insérer par *ANumbering.insert*.
- le deuxième élément est une spécialisation d'une interférence de type CB tel que nous l'avons défini. Il s'agit cependant d'une interférence de type CB entre de point de la chaine d'aspect et non entre le début de la chaine.

Les propriétés qui sont attendues de la composition peut donc se résumer comme suit :

- pas de changement de la variable *msgContent* entre l'exécution de *ANumbering.insert* et *ACacheManager.addIn*.
- *ANumbering.insert* est requis pour le bon fonctionnement de *ACacheManager.addIn*

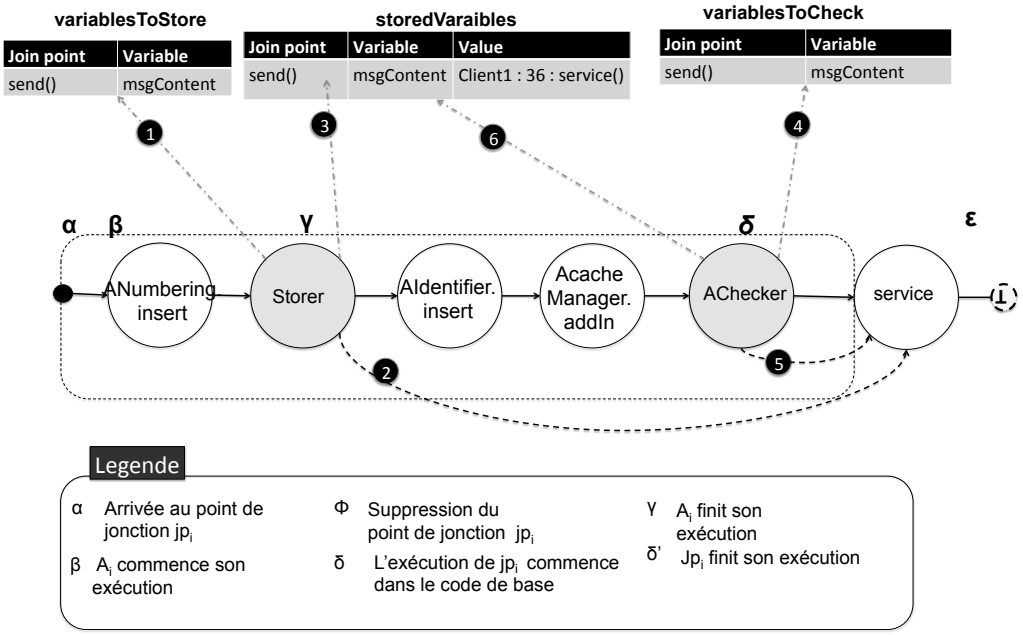


FIGURE 5.12 – Interférence de type CA entre le greffon *ANumbering.insert* et le greffon *AIdentifier.insert*

En nous appuyant sur l'approche d'instrumentation présentée précédemment nous avons instrumenté l'observation de cette propriété de la manière suivante :

- La détection d'une interférence de type CB entre *ANumbering.insert* et *ACacheManager.addIn* est instrumentée en insérant deux greffons dans la chaine d'exécution. Le premier greffon est inséré après l'exécution de *ANumbering.insert* qui capture la valeur de la variable *msgContent*. Le second greffon est inséré avant l'exécution de *ACacheManager.addIn* et vérifie que la valeur de la variable *msgContent* n'a pas été modifiée.
- La vérification que la dépendance entre *ANumbering.insert* et *ACacheManager.addIn* réalisé est instrumenté en insérant deux greffons dans la chaine d'exécution. Un premier greffon est inséré après l'exécution de *ANumbering.insert,* un deuxième greffon est inséré avant l'exécution de *ACacheManager.addIn.*

Cet exemple montre que notre approche d'instrumentation peut être étendue pour vérifier d'autres propriétés. Rappelons que cette thèse se concentre sur un sous ensemble des interférences en aspects.

5.7 Exemple de reconfiguration

Dans cette section nous présentons un scénario où le code non-fonctionnel de notre système évolue. L'objectif premier est d'illustrer le fait que, lors des évolutions du code non fonctionnel, mais aussi bien lors de la conception initiale du système, des interférences peuvent être introduites. Le deuxième objectif est d'illustrer l'efficacité de notre approche pour la détection des interférences introduites lors de l'évolution du système. Ces dernières sont, par ailleurs, différentes de celles qui ont été mises en évidence lors de la conception initiale du système. Cette section présente un scénario de reconfiguration : il s'agit de la reconfiguration du protocole de réplication duplex présenté précédemment en un protocole sécurisé. Dans ce premier

cas nous introduisons des préoccupations liées à la sécurité dans le protocole client-serveur.

Nous voulons assurer que tous les messages reçus par le serveur sont corrects et qu'une falsification par un intrus sera détectée. Des mécanismes de signature des messages sont utilisés pour garantir leur intégrité. Dans ce scénario nous faisons l'hypothèse que la liaison entre le serveur primaire et le serveur secondaire est sûre et ne peut pas être attaquée. Cette hypothèse fait qu'aucun mécanisme de sécurité n'est ajouté entre le serveur primaire et le serveur secondaire.

La figure 5.13 illustre l'utilisation de mécanismes de signature de messages. Du côté client avant l'envoi de la requête, le message est signé. Puis la requête est envoyée au serveur primaire. Arrivée au serveur, la signature de la requête est vérifiée. Ensuite la séquence d'actions consistant à la propagation de la requête, à l'exécution du service et l'envoi du point de reprise est exécutée.

5.7.1 Aspects côté serveur primaire sécurisé

Nous ajoutons un aspect dont le rôle est de signer les requêtes avant leur envoi et de vérifier la signature à la réception. Cet aspect est nommé *ASecure* et contient deux greffons :

- *ASecure.sign* dont le rôle est de signer les requêtes.
- *ASecure.check* qui est chargé de vérifier la signature des requêtes et de soulever une exception dans le cas où la signature n'est pas correcte.

5.7.2 Spécification des aspects de sécurité

La spécification des aspects utilisés pour implémenter le PBR reste la même. La spécification du greffon *ASecure.sign* et du greffon *ASecure.check* est récapitulée dans le tableau 5.4.

TABLE 5.4 – Spécification des propriétés de non-interférence des aspects de sécurité

ASecure.sign	CA
ASecure.check	CB

Une signature est ajoutée au message avant son envoi par le client. Nous devons nous assurer qu'aucun aspect appliqué après *ASecure.sign* ne modifie cette signature côté client. Le greffon *ASecure.sign* doit vérifier la propriété suivante :

Détection de CA : Il n'y a pas de modification de la variable du code de base *msgContent* utilisée par *ASecure.sign*, par un autre aspect exécuté avant *ASecure.sign*.

À l'arrivée du message côté serveur le message ne doit pas être modifié afin de ne pas altérer la signature des messages. Le greffon *ASecure.check* doit vérifier la propriété suivante :

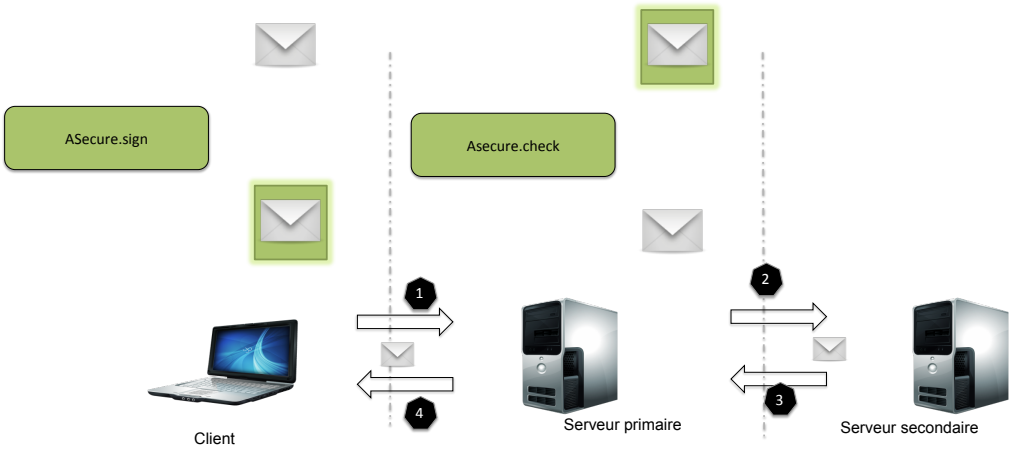


FIGURE 5.13 – Organisation des aspects de sécurité coté client et coté serveur primaire

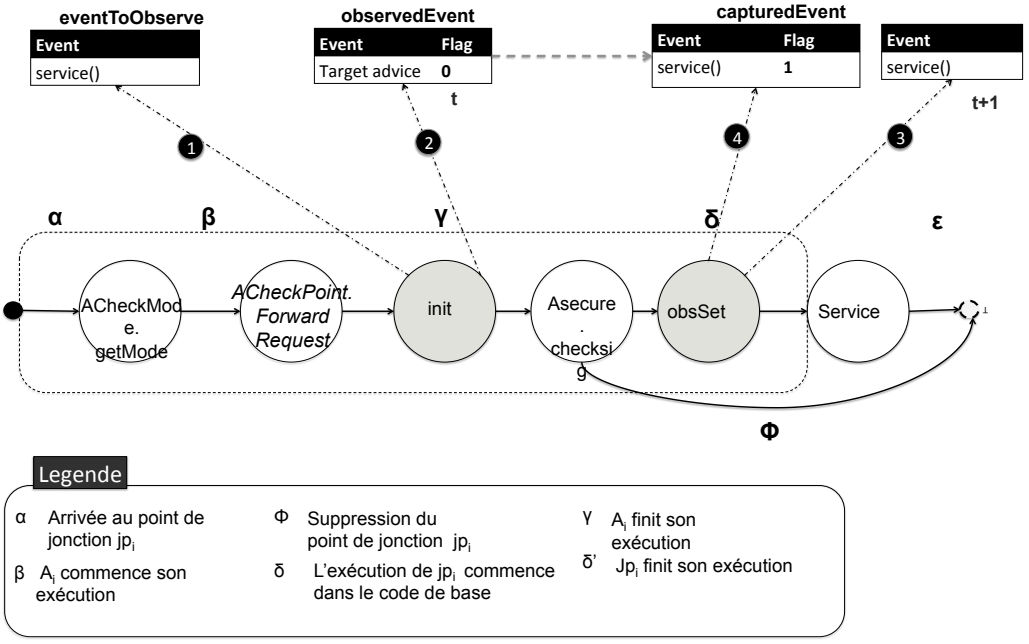


FIGURE 5.14 – Interférence de type IA entre le greffon *Anumbering.remove* et le greffon *ACheckPoint.forward*

Détection de CB : Il n'y a pas de modification de la variable du code de base *msgContent* utilisé par *ASecure.check*, par un autre aspect exécuté avant *ASecure.check*.

5.7.3 Détection des interférences

Dans ce scénario, des aspects de sécurité sont tissés en plus de ceux implémentant le protocole duplex. Les messages sont signés par le client avant l'envoi. À l'arrivée des messages au serveur, la signature est vérifiée avant le traitement de la requête. Deux erreurs ont été faites lors de l'intégration des aspects dédiés à la signature des messages et ont donné lieu à des in-

terférences. La première est de type IA et concerne les greffons *AForwardRequest.forward* et *ASecure.check*. La deuxième et de type CB et concerne les greffons *ANumbering.remove*, *AIdentifier.remove* et *ASecure.check*.

Les mêmes greffons du scénario précédent sont appliqués. Le greffon dédié à la vérification de la signature est ajouté à la chaine d'exécution qui devient alors : *ACheckMode.getMode* > *ASecure.check* > *AForwardRequest.forward* > *ACheckPoint.buildCheckPoint* > *ANumbering.remove*> *AIdentifier.remove*. Cette chaine est appliquée avant l'exécution du service. Nous supposons ici que le serveur est en mode primaire.

Cependant, l'intégrateur peut introduire une faute en appliquant les aspects dans l'ordre suivant : *ACheckMode.getMode* > *AForwardRequest.forward* > *ASecure.check* >*ANumbering.remove* > *AIdentifier.remove* > *service*.

Ce comportement est une interférence de type IA. Rappelons que la spécification du greffon requiert d'interdire les interactions de type IA soit vérifiée sur le greffon *AForwardRequest.forward*. Néanmoins, l'application du greffon *ASecure.check* après le greffon *AForwardRequest.forward* peut empêcher l'exécution point de jonction. Le comportement induit par ordre d'exécution des aspects est problématique dans la mesure où l'envoi au serveur par le greffon *AForwardRequest.forward* propage la requête au serveur secondaire avant d'avoir vérifié la signature.

La deuxième erreur introduite lors de l'intégration concerne les greffons *ANumbering.remove*, *AIdentifier.remove* et *ASecure.check*. L'ordre d'application réalisant le comportement attendu côté serveur est le suivant : *ACheckMode.getMode* > *ASecure.check* > *AForwardRequest.forward* > *ANumbering.remove* > *AIdentifier.remove* > *service*.

L'intégrateur peut introduire une faute en appliquant les aspects dans l'ordre suivant : *ACheckMode.getMode* > *ANumbering.remove* > *AIdentifier.remove* > *ASecure.check* > *AForwardRequest.forward* > *service*.

Ce comportement est une interférence de type CB. Rappelons que la spécification du greffon *ASecure.check* requiert d'interdire les interactions de type CB. L'application des greffons *ANumbering.remove* et *AIdentifier.remove* avant le greffon *ASecure.check* modifie la valeur de la variable *msgContent* sur laquelle *ASecure.check* vérifie la signature de messages. Le comportement induit par ordre d'exécution des aspects est problématique dans la mesure où la signature doit être vérifiée, coté serveur, sur la totalité du message envoyé. Ce message contient alors les informations suivantes : "*numéro de client : numéro de séquence : requête*". Néanmoins, le fait d'appliquer les greffons *ANumbering.remove* et *AIdentifier.remove* supprime les parties "*numéro de client*" et "*numéro de séquence*" du message. La vérification de la signature s'effectue alors sur un message tronqué.

5.7.4 Discussion

Dans cette section nous avons constaté que lors de l'évolution du système d'un point de vue non-fonctionnel des interférences peuvent se produire du fait de l'insertion de nouveaux aspects. Notre approche permet de détecter les interférences introduites lors de cette évolution. Dans le scénario d'évolution du système que nous avons considéré, deux nouvelles interférences ont été détectées par notre méthode d'instrumentation : la première de type IA, la deuxième de type CB.

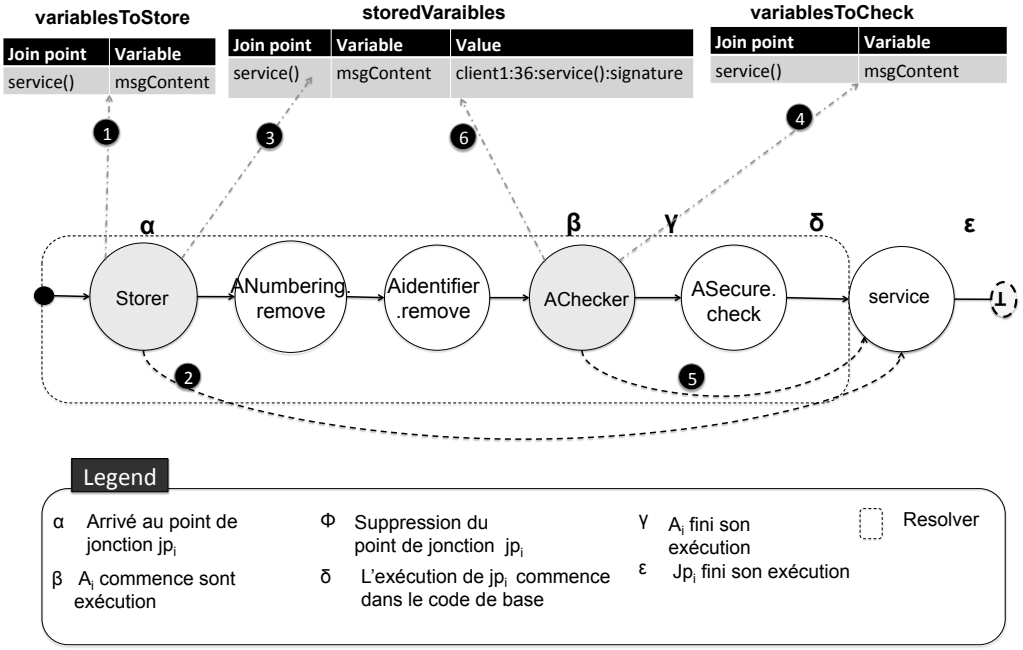


FIGURE 5.15 – Interférence de type CB entre les greffons *ANumbering.remove* et *AIdentifier.remove* sur le greffon *ASecure.check*

5.8 Conclusion

Dans une conception classique d'une application tolérante aux fautes, il existe un couplage fort entre le logiciel de tolérance aux fautes et le logiciel fonctionnel. Reconfigurer les mécanismes de tolérance aux fautes nécessite alors de modifier le système de manière invasive pour atteindre une configuration correspondant au logiciel de tolérance aux fautes souhaité. Ces modifications requièrent de gros efforts en terme de développement et de test.

Pour pallier cette problématique, ces travaux se sont appuyés sur le principe de séparation entre le logiciel de tolérance aux fautes et le logiciel applicatif pour permettre sa reconfiguration. Ainsi, il n'est pas nécessaire de re-développer complètement le service pour modifier une partie du logiciel de tolérance aux fautes.

Dans ce chapitre, nous avons conçu le protocole de tolérance aux fautes en utilisant une implémentation orientée aspect. Les mécanismes de tolérance aux fautes proposés ici ne sont pas les seuls avec lesquels une telle implémentation du protocole est possible, mais uniquement un exemple illustratif. Le premier résultat présenté dans ce chapitre est l'illustration qu'une implémentation à grain fin rend possible la modification du logiciel alors qu'une conception plus classique ne le permettait pas aussi aisément.

Cependant, l'assemblage des différents aspects développés pour implémenter le protocole de tolérance aux fautes pose de nombreuses difficultés. Des erreurs émanent d'interactions non désirées entre les différents aspects qui composent le protocole. Ces erreurs sont difficiles à corriger puisqu'il est difficile de retrouver leur source dans l'enchevêtrement de code produit

par le tisseur d'aspects. Nous avons ensuite illustré nos travaux autour de la détection d'interférences entre aspects sur cette étude de cas.

Le deuxième résultat est l'application de la méthode de détection des interférences entre différents aspects à l'exécution dans notre étude de cas. Notre approche permet d'une part de détecter les interférences dans un assemblage d'aspects et facilite d'autre part la localisation de la source de l'erreur. Ainsi l'assemblage de différents mécanismes de tolérance aux fautes visant à construire une stratégie de tolérance aux fautes est facilité et plus sûr.

Chapitre 6

Bilan et perspectives

Préambule

Ce chapitre dresse le bilan du travail réalisé au cours de cette thèse et présente cinq perspectives qui nous intéressent suite à ce travail. La première concerne la considération d'autres interférences. La seconde propose l'étude de l'application de notre approche d'instrumentation à d'autres langages de programmation orientée aspect. La troisième, plus pragmatique, soulève le besoin de concevoir des outils de développements adaptés permettant de guider l'utilisateur dans le processus d'intégration des aspects. Dans la quatrième perspective que nous présentons, nous détaillons notre vision en ce qui concerne la vérification et la validation d'une librairie d'aspect permettant de couvrir des modèles de fautes divers et variés. Enfin, dans la cinquième et dernière partie, nous présentons notre vision à long terme de ce travail qui est d'aller vers un traitement des interférences depuis les exigences jusqu'au code.

Sommaire

6.1 Bilan

CETTE thèse s'inscrit dans les domaines de recherche de la résilience et du développement par aspect, actuellement très actifs à cause de la difficulté de concevoir, de maintenir et de faire évoluer des applications de plus en plus complexes tout en garantissant leur sureté de fonctionnement. En effet, les approches par aspect promettent une meilleure modularisation des logiciels de grande taille facilitant ainsi leur conception, leur maintenance et leur évolution. Ces aspects peuvent être développés séparément, puis être utilisés pour configurer des caractéristiques non-fonctionnelles d'un système. Cependant, lors de la composition des aspects, des problèmes d'interférences peuvent apparaitre. Malgré de nombreuses propositions au niveau des exigences, des modèles, le traitement des interférences au niveau du code est une alternative peu explorée.

Dans cette thèse, nous avons abordé cette problématique en étudiant les principales approches dédiées au traitement des interférences au niveau du code (cf. chapitre 2) ce qui nous a permis d'isoler les deux notions qui nous semblent fondamentales pour le traitement d'interférences à l'exécution : l'évitement et la détection.

L'évitement consiste à forcer l'intégrateur à spécifier un ordre d'application des aspects à chaque point de jonction partagé. Cette méthode préventive nous semble nécessaire puisque par défaut des langages comme ASPECTJ tissent de façon aléatoire les aspects aux points de jonction partagés. En forçant la définition d'un ordre nous évitons ce comportement qui peut introduire des fautes difficiles à corriger. La détection consiste à vérifier des propriétés de non-interférence à l'exécution. L'observabilité, rarement mise en avant dans les travaux sur la détection des interférences, nous semble primordiale. Par observabilité, nous désignons le fait que les intégrateurs d'aspect ont besoin de certains points d'observation utilisables pour instrumenter des propriétés de non interférence.

Ces deux objectifs nous ont conduit à proposer, dans le chapitre 3, l'utilisation d'une extension d'ASPECTJ appelée AIRIA. Avec AIRIA, notre objectif est de permettre la détection d'interférence dans les assemblages d'aspects en nous appuyant sur une approche d'instrumentation utilisant l'observabilité et la composabilité qu'il fournit. La spécification de cette méthode d'instrumentation a été faite en adoptant une démarche constructive, c'est-à-dire que nous avons identifié un à un les besoins en termes de points d'observation pour un ensemble d'interférences. Puis pour chaque interférence nous avons fourni une méthode d'instrumentation.

Nous avons évalué la faisabilité de notre méthode d'instrumentation par un ensemble de 14400 expérimentations. Chaque expérimentation s'appuie sur une application comprenant des interactions entre aspects générées aléatoirement. Ces expérimentations ont montré que notre approche d'instrumentation permet de détecter des interférences dans des séquencements d'aspects complexes.

Nous avons également appliqué notre approche d'instrumentation à un cas d'étude. Celuici consiste à assembler un protocole de réplication à l'aide d'aspects implémentés à grain fin. L'assemblage des aspects est réalisé dans un premier temps sans l'aide de notre méthode d'instrumentation. Dans ce premier cas l'intégration des aspects a été fastidieuse. Dans un second temps notre approche d'instrumentation a été utilisée en suivant le même schéma d'intégration. Cette expérience a permis de mettre en évidence le réel soutient apporté par notre approche pour la détection et la correction d'interférences.

6.2 Perspectives

Les perspectives de ce travail sont multiples tant sur le plan conceptuel qu'opérationnel. Nous détaillons ici celles qui nous intéressent le plus et témoignent de notre vision de l'avenir pour notre approche d'instrumentation.

Vers d'autres types d'interférences

Tout d'abord, nous souhaitons élargir l'ensemble d'interférences ciblé par notre approche d'instrumentation. Les premiers résultats de ce travail ont été abordés dans le chapitre 5 où dans le cadre d'une étude de cas nous avons dû considérer un cas d'interférence qui n'appartenait pas à l'ensemble d'interférences considéré initialement par nos travaux. La prochaine étape consiste à prendre en compte d'autres types d'interférences comme celles décrites dans [50], voire intégrer dans notre approche les interférences entre aspects et code de base décrites dans [63] et les collaborations attendues entre aspects.

Vers d'autres langages de programmation orientée aspect

Les travaux de [31] montrent que la programmation orientée aspect peut être mise en oeuvre dans d'autres langages. Nous envisageons la possibilité d'utiliser la même approche d'instrumentation à d'autres langages permettant de réaliser la programmation par aspect. Notre volonté d'unifier le niveau code en considérant plusieurs cibles possibles notamment des langages à évènement [10].

Outils d'assistance à l'intégration de développement

Au-delà de notre approche d'instrumentation, nous souhaitons concevoir des outils spécifiques pour l'intégration des aspects. Nous avons montré comment une approche interactive de spécification comme celle proposée par KATZ [38] peut être utilisée pour assister l'intégrateur dans la spécification des propriétés attendues de la composition d'aspects. Toutefois, il existe en pratique peu d'outils pour ces nouvelles tâches et encore moins dans un environnement intégré. Nous pensons que l'intégration des aspects nécessite des outils spécifiques. Il existe d'ailleurs actuellement des propositions d'environnements dédiés à l'assemblage d'artefacts non fonctionnels, mais ces derniers sont orientés vers des langages de programmation évènementiels [36].

Vers une librairie de mécanismes de tolérance aux fautes

Nous avons appliqué notre approche d'instrumentation à une librairie d'aspects permettant de composer le comportement d'un protocole duplex. Notre approche d'instrumentation aide à l'assemblage non anticipé d'aspects. Nous pensons qu'une documentation rigoureuse des aspects d'une librairie est aussi utile et peut constituer une aide pour l'intégrateur. Nous souhaitons élargir notre librairie d'aspects en considérant d'autres mécanismes de tolérance aux fautes.

Des modèles au code

De nombreuses approches s'attaquent à la détection des interférences au niveau des modèles. Ces modèles peuvent être des modèles d'exigences [55], des modèles structurels [43],

ou comportementaux. Il nous semble intéressant d'avoir une approche *bottom up* en considérant notre approche d'instrumentation comme un moyen de vérifier des propriétés de non-interférence, dans un premier temps au niveau de modèles, dans un second temps au niveau du code, en assurant une traçabilité entre les éléments des modèles et les éléments du code. Des travaux comme [43] sur la traçabilité des modèles orientés aspect vers le code ont déjà été réalisés dans sens et nous semble de bons candidats pour explorer cette voie.

Bibliographie

[1] R. T. Alexander, J. M. Bieman, and A. A. Andrews. Towards the Systematic Testing of Aspect-Oriented Programs. In *Aspect-Oriented Software Development*, 2005.

[2] R. Alexandersson and J. Karlsson. Fault injection-based assessment of aspect-oriented implementation of fault tolerance. In *DSN*, pages 303–314, 2011.

[3] R. Alexandersson and P. Öhman. Implementing fault tolerance using aspect oriented programming. In *LADC*, pages 57–74, 2007.

[4] R. Alexandersson and P. Öhman. On hardware resource consumption for aspect-oriented implementation of fault tolerance. In *EDCC*, pages 61–66, 2010.

[5] R. Alexandersson, P. Öhman, and J. Karlsson. Aspect-oriented implementation of fault tolerance : An assessment of overhead. In *SAFECOMP*, pages 466–479, 2010.

[6] P. Anbalagan and T. Xie. Apte : automated pointcut testing for aspectj programs. In *Proceedings of the 2nd workshop on Testing aspect-oriented programs*, WTAOP '06, pages 27–32, New York, NY, USA, 2006. ACM.

[7] P. Anbalagan and T. Xie. Automated Generation of Pointcut Mutants for Testing Pointcuts in AspectJ Programs. In *International Symposium on Software Reliability Engineering*, pages 239–248, 2008.

[8] P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. *abc* : An extensible aspectj compiler. pages 293–334, 2006.

[9] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1) :11–33, Jan. 2004.

[10] M. Bagherzadeh, H. Rajan, G. T. Leavens, and S. Mooney. Translucid contracts : expressive specification and modular verification for aspect-oriented interfaces. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 141–152, New York, NY, USA, 2011. ACM.

[11] D. Balzarotti, A. C. D'Ursi, L. Cavallaro, and M. Monga. Slicing AspectJ Woven Code. In *Foundations of Aspect-Oriented Languages*, 2005.

[12] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10) :751–761, Oct. 1996.

[13] E. Bodden. Towards typesafe weaving for modular reasoning in aspect-oriented programs. In *FOAL '12 : International Workshop on the Foundations of Aspect-Oriented Languages*, Mar. 2012. Keynote abstract.

[14] G. Booch. Object-oriented development. *IEEE Trans. Software Eng.*, 12(2) :211–221, 1986.

[15] J. Brichau, K. Mens, and K. D. Volder. Building composable aspect-specific languages with logic metaprogramming. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on*

Generative Programming and Component Engineering, GPCE '02, pages 110–127, London, UK, UK, 2002. Springer-Verlag.

[16] J. Bsekken and R. Alexander. A candidate fault model for aspectj pointcuts. In *Software Reliability Engineering, 2006. ISSRE '06. 17th International Symposium on*, pages 169 –178, nov. 2006.

[17] Bussard, Carver, Ernst, Jung, Robillard, and Speck. Safe aspect composition. In *Oriented Technology : ECOOP 2000 Workshop Reader, volume 1964 of Lecture Notes in Computer Science*, page 205–210, 2000.

[18] M. Chen. Towards smart city : M2m communications with software agent intelligence. *Multimedia Tools and Applications*, pages 1–12, 2012. 10.1007/s11042-012-1013-4.

[19] S. Chiba and T. Masuda. Designing an extensible distributed language with a meta-level architecture. In *ECOOP*, pages 482–501, 1993.

[20] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6) :476–493, June 1994.

[21] A. M. Colyer and A. Clement. Aspect-oriented programming with aspectj. *IBM Systems Journal*, 44(2) :301–308, 2005.

[22] V. T. da Silva, A. F. Garcia, A. Brandão, C. Chavez, C. J. P. de Lucena, and P. S. C. Alencar. Taming agents and objects in software engineering. In *SELMAS*, pages 1–26, 2002.

[23] A. Damm. The effectiveness of software error-detection mechanisms in real-time operating systems. In *Symposium on Fault-Tolerant Computing*, 1986.

[24] R. Delamare, B. Baudry, S. Ghosh, and Y. L. Traon. A test-driven approach to developing pointcut descriptors in aspectj. In *ICST*, pages 376–385. IEEE Computer Society, 2009.

[25] R. Delamare, B. Baudry, and Y. Le Traon. Regression test selection when evolving software with aspects. In *Proceedings of the 2008 AOSD workshop on Linking aspect technology and evolution*, LATE '08, pages 7 :1–7 :5, New York, NY, USA, 2008. ACM.

[26] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, GPCE '02, pages 173–188, London, UK, UK, 2002. Springer-Verlag.

[27] P. Durr, L. Bergmans, and M. Aksit. Reasoning about Semantic Conflicts between Aspects. 2005.

[28] T. Ekman and G. Hedin. jastadd extensible compilateur java. In *OOPSLA*, pages 1–18, 2007.

[29] J.-C. Fabre and T. Pérennou. A metaobject architecture for fault-tolerant distributed systems : The friends approach. *IEEE Trans. Computers*, 47(1) :78–95, 1998.

[30] F. C. Ferrari, J. C. Maldonado, and A. Rashid. Mutation Testing for Aspect-Oriented Programs. In *International Conference on Software Testing, Verification, and Validation*, pages 52–61, 2008.

[31] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical report, 2000.

[32] K. Hameed, R. Williams, and J. Smith. Separation of fault tolerance and non-functional concerns : Aspect oriented patterns and evaluation. *JSEA*, 3(4) :303–311, 2010.

[33] S. Hanenberg and R. Unland. Parametric introductions. In *AOSD*, pages 80–89, 2003.

[34] W. Havinga, I. Nagy, L. Bergmans, and M. Aksit. Detecting and resolving ambiguities caused by inter-dependent introductions. In *AOSD*, pages 214–225, 2006.

[35] E. Hilsdale and J. Hugunin. Advice weaving in aspectj. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, AOSD '04, pages 26–35, New York, NY, USA, 2004. ACM.

[36] M. Hiltunen. Configuration management for highly customisable software. *Software, IEE Proceedings* -, 145(5) :180 –188, oct 1998.

[37] E. Katz and S. Katz. Incremental analysis of interference among aspects. In *Proceedings of the 7th workshop on Foundations of aspect-oriented languages*, FOAL '08, pages 29–38, New York, NY, USA, 2008. ACM.

[38] E. Katz and S. Katz. User queries for specification refinement treating shared aspect join points. In *SEFM*, pages 73–82, 2010.

[39] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[40] G. Kiczales and J. D. Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[41] J. Kienzle, E. Duala-Ekoko, and S. Gélineau. Aspectoptima : A case study on aspect dependencies and interactions. In A. Rashid and H. Ossher, editors, *Transactions on Aspect-Oriented Software Development V*, volume 5490 of *Lecture Notes in Computer Science*, pages 187–234. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-02059-96.

[42] H. Klaeren, E. Pulvermüller, A. Rashid, and A. Speck. *Aspect Composition Applying the Design by Contract Principle*. 2000.

[43] M. Kramer and J. Kienzle. Mapping aspect-oriented models to aspect-oriented code. In J. Dingel and A. Solberg, editors, *Models in Software Engineering*, volume 6627 of *Lecture Notes in Computer Science*, pages 125–139. Springer Berlin Heidelberg, 2011.

[44] J.-C. Laprie. From dependability to resilience. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks(DSN)*, page Fast Abstracts, Anchorage, AK, June 2008.

[45] J. Lauret, H. Waeselynck, and J.-C. Fabre. Detection of interferences in aspect-oriented programs using executable assertions. In *ISSRE Workshops*, pages 165–170, 2012.

[46] O. A. L. Lemos, I. G. Franchin, and P. C. Masiero. Integration testing of object-oriented and aspect-oriented programs : A structural pairwise approach for java. *Science of Computer Programming*, 74(10) :861 – 878, 2009.

[47] O. A. L. Lemos and P. C. Masiero. A pointcut-based coverage analysis approach for aspect-oriented programs. *Information Sciences*, 181(13) :2721 – 2746, 2011.

[48] C. V. Lopes and T. C. Ngo. Unit-Testing Aspectual Behavior. 2005.

[49] P. Maes and D. Nardi, editors. *Meta-Level Architectures and Reflection*. Elsevier Science Inc., New York, NY, USA, 1988.

[50] A. Marot and R. Wuyts. Detecting unanticipated aspect interferences at runtime with compositional intentions. In *Proceedings of the Workshop on AOP and Meta-Data for Software Evolution*, RAM-SE '09, pages 3 :1–3 :5, New York, NY, USA, 2009. ACM.

[51] K. L. McMillan. Getting started with smv. *Cadence Berkeley Laboratories*, 1999.

[52] P. M. Melliar-Smith, L. E. Moser, and P. Narasimhan. Separation of concerns : Functionality vs. quality of service. In *Proceedings of the 3rd Workshop on Object-Oriented Real-Time Dependable Systems - (WORDS '97)*, WORDS '97, pages 272–, Washington, DC, USA, 1997. IEEE Computer Society.

[53] M. Mortensen. An approach for adequate testing of aspectj programs. In *In 2005 Workshop on Testing Aspect-Oriented Programs (held in conjunction with AOSD*, 2005.

[54] F. Munoz, B. Baudry, and O. Barais. Improving maintenance in aop through an interaction specification framework. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 77–86. IEEE, 2008.

[55] G. Mussbacher, D. Amyot, and J. Whittle. Semantic-based aspect interaction detection with goal models - (position paper). In *ICFI*, pages 176–182, 2009.

[56] N. Oh, P. P. Shirvani, and E. J. McCluskey. Control-flow checking by software signatures. *IEEE TRANSACTIONS ON RELIABILITY*, 51 :111–122, 2002.

[57] M. C. Paolo Tonella. Is aop code easier to test than oop code ? In *In Workshop on Testing Aspect-Oriented Programs, 4th International Conference on Aspect-Oriented Software Development*, 2005.

[58] R. Pawlak, L. Duchien, and L. Seinturier. Compar : Ensuring safe around advice composition. In M. Steffen and G. Zavattaro, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 3535 of *Lecture Notes in Computer Science*, pages 155–178. Springer Berlin / Heidelberg, 2005. 10.1007/1149488111.

[59] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, F. Legond-aubry, and L. Martelli. JAC : an aspect-based distributed dynamic framework. *Software - Practice and Experience*, 34 :1119–1148, 2004.

[60] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt : Transparent checkpointing under unix, 1995.

[61] D. R. Prasanna. *Dependency Injection*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2009.

[62] B. Randell. System structure for software fault tolerance. *SIGPLAN Not.*, 10(6) :437–449, Apr. 1975.

[63] M. C. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In R. N. Taylor and M. B. Dwyer, editors, *SIGSOFT FSE*, pages 147–158. ACM, 2004.

[64] P. Rogers and A. Wellings. An incremental recovery cache supporting software fault tolerance. In M. Gonzalez Harbour and J. de la Puente, editors, *Reliable Software Technologies Ada Europe 99*, volume 1622 of *Lecture Notes in Computer Science*, pages 665–665. Springer Berlin Heidelberg, 1999.

[65] F. Sanen, E. Truyen, W. Joosen, A. Jackson, A. Nedos, S. Clarke, N. Loughran, and A. Rashid. Classifying and documenting aspect interactions. pages 23–26, 2006.

[66] C. Sant'anna, A. Garcia, C. Chavez, C. Lucena, and A. v. von Staa. On the reuse and maintenance of aspect-oriented software : An assessment framework. In *Proceedings XVII Brazilian Symposium on Software Engineering*, 2003.

[67] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. Aspectc++ : an aspect-oriented extension to the c++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific : Objects for internet, mobile and embedded applications*, CRPIT '02, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[68] F. Steimann, T. Pawlitzki, S. Apel, and C. Kastner. Types and modularity for implicit invocation with implicit announcement. *ACM Trans. Softw. Eng. Methodol.*, 20(1) :1 :1–1 :43, July 2010.

[69] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF : Eclipse Modeling Framework*. Addison-Wesley, 2. edition, 2009.

[70] M. Stoicescu, J.-C. Fabre, and M. Roy. From Design for Adaptation to Component-Based Resilient Computing. In *Proceedings of the IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2012)*, page 10p., Niigata, Japon, Nov. 2012.

[71] R. Stroud. Transparency and reflection in distributed systems. *SIGOPS Oper. Syst. Rev.*, 27(2) :99–103, Apr. 1993.

[72] M. Störzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM*, pages 653–656. IEEE Computer Society, 2005.

[73] K. Sullivan, W. G. Griswold, H. Rajan, Y. Song, Y. Cai, M. Shonle, and N. Tewari. Modular aspect-oriented design with xpis. *ACM Trans. Softw. Eng. Methodol.*, 20(2) :5 :1–5 :42, Sept. 2010.

[74] F. Takeyama and S. Chiba. An advice for advice composition in aspectj. In *Proceedings of the 9th international conference on Software composition*, SC'10, pages 122–137, Berlin, Heidelberg, 2010. Springer-Verlag.

[75] E. Tanter. Aspects of composition in the reflex aop kernel. In *Software Composition*, pages 98–113, 2006.

[76] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Program. Lang. Syst.*, 26(5) :890–910, Sept. 2004.

[77] F. Wedyan and S. Ghosh. A dataflow testing approach for aspect-oriented programs. In *Proceedings of the 2010 IEEE 12th International Symposium on High-Assurance Systems Engineering*, HASE '10, pages 64–73, Washington, DC, USA, 2010. IEEE Computer Society.

[78] N. Weston, F. Taiani, and A. Rashid. Interaction Analysis for Fault-Tolerance in Aspect-Oriented Programming ?

[79] T. Xie and J. Zhao. A framework and tool supports for generating test inputs of aspectj programs. In *Proceedings of the 5th international conference on Aspect-oriented software development*, AOSD '06, pages 190–201, New York, NY, USA, 2006. ACM.

[80] T. Xie and J. Zhao. Perspectives on automated testing of aspect-oriented programs. pages 7–12, 2007.

[81] D. Xu and W. Xu. State-based incremental testing of aspect-oriented programs. In *Proceedings of the 5th international conference on Aspect-oriented software development*, AOSD '06, pages 180–189, New York, NY, USA, 2006. ACM.

[82] D. Xu, W. Xu, and K. E. Nygard. A state-based approach to testing aspect-oriented programs. In W. C. Chu, N. J. Juzgado, and W. E. Wong, editors, *SEKE*, pages 366–371, 2005.

[83] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2) :292–333, Mar. 1997.

[84] J. Zhao. Data-Flow-Based Unit Testing of Aspect-Oriented Programs. In *International Computer Software and Applications Conference*, pages 188–197, 2003.

[85] J. Zhao and M. C. Rinard. Pipa : A behavioral interface specification language for aspectj. In M. Pezzè, editor, *FASE*, volume 2621 of *Lecture Notes in Computer Science*, pages 150–165. Springer, 2003.

Résumé

La programmation orientée aspects (POA) sépare les différentes préoccupations composant un système informatique pour améliorer la modularité. La POA offre de nombreux bénéfices puisqu'elle permet de séparer le code fonctionnel du code non-fonctionnel améliorant ainsi leur réutilisation et la configurabilitè des systèmes informatiques. La configurabilité est un élément essentiel pour assurer la résilience des systèmes informatiques, puisqu'elle permet de modifier les mécanismes de sûreté de fonctionnement. Cependant le paradigme de programmation orientée aspect introduit de nouveaux défis pour le test. Dans les systèmes de grande taille où plusieurs préoccupations non fonctionnelles cohabitent, une implémentation à l'aide d'aspects de ces préoccupations peut être problématique. Partageant le même flot de données et le même flot de contrôle les aspects implémentant les différentes préoccupations peuvent écrire dans des variables lues par d'autres aspects ou interrompre le flot de contrôle commun aux différents aspects empêchant ainsi l'exécution de certains d'entre eux. Dans cette thèse nous nous intéressons plus spécifiquement aux interférences entre aspects dans le cadre du développement de mécanismes de tolérance aux fautes implémentés sous forme d'aspects. Ces interférences sont dues à une absence de déclaration de précédence entre les aspects ou à une déclaration de précédence erronée.

Afin de mieux maîtriser l'assemblage des différents aspects composant un mécanisme de tolérance aux fautes, nous avons développé une méthode alliant l'évitement à la détection des interférences au niveau du code. Le but de l'évitement est d'empêcher l'introduction d'interférences en imposant une déclaration de précédence entre les aspects lors de l'intégration des aspects. La détection permet d'exhiber lors du test les erreurs introduites dans la déclaration des précédences. Ces deux facettes de notre approche sont réalisées grâce à l'utilisation d'une extension d'AspectJ appelée AIRIA. Les constructions d'AIRIA permettent l'instrumentation et donc la détection des interférences entre aspects, avec des facilités de compilation permettant de mettre en œuvre l'évitement d'interférences. Notre approche est outillée et vise à limiter le temps de déboguage : le testeur peut se concentrer directement sur les points où une interférence se produit.

Nous illustrons notre approche sur une étude de cas : un protocole de réplication duplex. Dans ce contexte le protocole est implémenté en utilisant des aspects à grain fin permettant ainsi une meilleure configurabilité de la politique de réplication. Nous montrons que l'assemblage de ces aspects à grain fin donne lieu à des interférences de flot de données et flot de contrôle qui sont détectées par notre approche d'instrumentation. Nous définissons un ensemble d'aspects interférant pour l'exemple, et nous montrons comment notre approche permet la détection d'interférences.

Abstract

Aspect-oriented programming (AOP) separates the different concerns of a computer software system to improve modularity. AOP offers many benefits since it allows separating the functional code from the non-functional code, thus improving reuse and configurability of computer systems. Configurability is essential to ensure the resilience of computer systems, since it allows modifying the dependability mechanisms. However, the paradigm of aspect-oriented programming introduces new challenges regarding testing. In large systems where multiple non-functional concerns coexist, an AOP implementation of these concerns can be problematic. Sharing the same data flow and the same control flow, aspects implementing different concerns can write into variables read by other aspects, or interrupt the control flow involving various aspects, and thus preventing the execution of some aspects in the chain. In this work we focus more specifically on interference between aspects implementing fault tolerance mechanisms. This interference is due to a lack of declaration of fine-grain precedence between aspects or an incorrect precedence declaration.

To better control the assembly of the various aspects composing fault tolerance mechanisms, we have developed a method combining avoidance of interferences with runtime detection interferences at code level. The purpose of avoidance is to prevent the introduction of interference by requiring a statement of precedence between aspects during the aspects integration. Detection allows exhibiting during the test, errors introduced in the precedence statement. These two aspects of our approach are performed through the use of an extension called AspectJ AIRIA. AIRIA 's constructs allow instrumentation and therefore the detection of interference between aspects, with facilities compilation to implement the interference avoidance. Our approach is designed and equipped to limit the debugging time : the tester can focus directly on the points where an interference occurs.

Finaly, we illustrate our approach on a case study : a duplex replication protocol. In this context, the protocol is implemented using fine grained aspects allowing a better configurability of the replication policy. We show that the assembly of these fine-grained aspects gives rise to interference data flow and control flow that are detected by our instrumentation approach. We define a set of interfering aspects in this example, and show how our approach allows the detection of interferences.