République Tunisienne
Ministère de l'Enseignement Supérieur
et de la Recherche Scientifique

Université de Sfax
École Nationale d'Ingénieurs de Sfax

Ecole Doctorale
Sciences et Technologies

Mémoire de *MASTERE*

NTSID

N° d'ordre: 240-12

MEMOIRE

Présenté à

L'École Nationale d'Ingénieurs de Sfax

en vue de l'obtention du

MASTERE

Nouvelles Technologies des Systèmes Informatiques Dédiés

Par

Rahma BOUAZIZ

(***Ingénieur Informatique***)

Extension et Adaptation d'un Langage d'Aspect pour les Systèmes Temps-Réel

Soutenu le 28 juillet 2012 , devant le jury composé de :

M.	**Mohamed JMAIEL**	Président
M.	**Adel MAHFOUDHI**	Rapporteur
M.	**Bechir ZALILA**	Encadrant

À mes parents,

À mon mari,

À mes sœurs, leurs maris et leurs enfants,

À mes beaux parents,

À mes amies,

et qui ne me sont pas moins chers.

Remerciements

C'est avec un grand plaisir que je réserve cette page en signe de gratitude et de reconnaissance à tous ceux qui ont assisté ce travail.

Je remercie chaudement M. Bechir ZALILA, Maître Assistant à l'École Nationale d'Ingénieurs de Sfax (ENIS), pour la confiance qu'il m'a accordé en dirigeant mes travaux de mastère. Je lui exprime mes sincères remerciements pour la qualité de son encadrement, ses qualités humaines, sa rigueur scientifique, ses conseils toujours judicieux dont il a su me faire bénéficier, ainsi que pour le suivi et le temps qu'il m'a consacré afin de produire ce travail. Qu'il trouve dans ce mémoire le témoignage de ma profonde reconnaissance.

Je dois témoigner ma profonde gratitude à M. Mohamed JMAIEL, Professeur à l'ENIS, pour m'avoir fait l'honneur de présider la comité de l'examen.

Je remercie vivement M. Adel MAHFOUDHI, Maître Assistant Habilité à la Faculté des Sciences de Sfax, d'avoir accepté de juger mes travaux de mastère.

J'exprime mes reconnaissances envers les membres de l'unité de *Recherche en Développement et Contrôle d'Applications Distribuées* pour l'atmosphère amical et l'ambiance chaleureuse qui règne au sein de l'équipe.

Table des matières

Table des figures

Liste des tableaux

Listings

Introduction Générale

Actuellement, le développement des systèmes temps-réel ne cesse de s'accroître chaque jour dans le monde entier en vue d'être plus fiable et plus sécurisé. Les systèmes temps-réel envahissent différents domaines d'application (avioniques, spatiaux, médicaux, bancaires, etc). Cette croissance est passée au détriment d'une complexité dans les phases de développement et de maintenance. Notamment, le développement des systèmes temps-réel constitue de nos jours une tâche fastidieuse. Ceci est dû à la criticité de ces systèmes et aux nombres de contraintes qu'ils doivent respecter pour satisfaire les exigences temps-réel, dynamisme, etc.

Dans le contexte des systèmes temps-réel, quelles que soient les précautions prises, l'occurrence d'erreurs ne peut pas toujours être évitée à cause de certains facteurs internes ou externes. Nous nous intéressons aux systèmes critiques où le moindre retard implique des dégâts importants (par exemple des vies humaine sont en jeu).

Par ailleurs, la maîtrise de ces systèmes en garantissant la sûreté de fonctionnement (Eng. *Dependability*) et le respect des contraintes non fonctionnelles est devenue aujourd'hui l'un des défis du développement logiciel. Dans ce sens, des techniques destinées à assurer un haut niveau de fiabilité en temps-réel s'avèrent nécessaires.

La tolérance aux pannes (*Eng. Fault tolerance*) est l'un des moyens assurant la sûreté de fonctionnement d'un système. Elle est définie par la capacité du système de continuer à fournir ses services, même en présence d'erreurs. Elle est fondée sur la détection et le recouvrement des erreurs. Etant donné que la tolérance aux pannes est une préoccupation transversale, il vaudrait mieux l'implanter séparément. Cela veut dire ne pas intégrer son

implantation directement dans le code applicatif dans le but de le garder lisible et facilement maintenable. C'est dans ce cadre que nous allons utiliser la programmation orientée aspect (POA) [KLM+97] .

La POA est un paradigme de programmation qui permet de séparer les considérations techniques des descriptions métier dans une application. Ce type de programmation permet d'améliorer la modularité et d'assurer une meilleure réutilisation du code. La POA est fondée sur l'opération de tissage permettant d'intégrer automatiquement les bouts de code (aspects) créés séparément dans le code d'une application. Cependant, dans le contexte des systèmes temps-réel, cette opération peut compromettre le déterminisme en insérant dans le code de l'application des constructions pouvant violer les contraintes temps-réel et échappant à tout contrôle du développeur.

L'objectif principal de ce travail de mastère est l'intégration du paradigme orienté aspect dans le développement des systèmes temps-réel. Nous visons étudier un mécanisme d'Aspect existant pour pouvoir l'adapter et le modifier afin de garantir les contraintes temps-réel. Nous partirons du projet AspectAda [PC05] (extension d'aspect pour le langage Ada). Ce choix est justifié par le fait que le langage Ada [WG05] est dédié dès sa création pour le développement des systèmes temps-réel. Le langage Ada a prouvé sa puissance et sa supériorité pour le support du temps-réel par rapport aux autres technologies concurrentes définies dans ce domaine [PV98].

Nous nous intéressons à étudier le langage AspectAda vis à vis sa syntaxe, sa sémantique, sa bibliothèque standard (`Runtime`), son compilateur/tisseur et particulièrement l'opération de tissage. Ceci dans le but de l'améliorer et l'adapter pour le respect des contraintes temps-réel. L'étude de l'existant nous ramène à explorer les limites de ce langage. D'une part, le langage AspectAda présente un ensemble de lacunes. Sa syntaxe souffre de certaines anomalies et manques de certains concepts de la POA. D'autre part, la mise en œuvre d'un compilateur prototype AspectAda est encore à un stade précoce. En outre, le compilateur ne peut pas faire face à toutes les constructions du langage AspectAda dans tous les contextes possibles. Ce compilateur fait appel à un outil de génération automatique de parseur (Scaflex/Scayacc/AdaGoop) donnant un code généré illisible et pratiquement

non exploitable. En outre, le parseur généré ne detecte pas très bien les erreurs et n'affiche pas des messages d'erreurs qui aident à la réparation. De plus, les créateurs originaux de AspectAda n'ont pas attribué une importance à l'opération de tissage. Dans le contexte temps-réel, cette opération doit être soumise à des règles pertinantes parce qu'elle a un impact directe sur le fonctionnement du système. Outre ces limitations, la bibliothèque `Runtime` du langage souffre de deux problèmes : (1) d'une part, elle est tellement réduite qu'elle n'offre que peu de routines pour le développeur et (2) d'autre part, son implantation renferme des constructions qui se contredisent avec les contraintes temps-réel.

Suite à l'étude de l'existant et l'extraction des limites, nous proposons notre appproche en essayant de trouver les remèdes adéquats. L'objectif de ce travail de mastère est de :

1. Améliorer et étendre la syntaxe et la sémantique du langage AspectAda pour supporter plus de concepts de la POA et pour qu'il soit en conformité avec Ada,
2. Corriger et enrichir la bibliothèque `Runtime` par des nouvelles fonctionnalités,
3. Définir, valider et implanter les règles de tissage et de génération de code afin d'obtenir un code Ada tissé respectant les exigences temps-réel.
4. Proposer une nouvelle architecture pour le compilateur AspectAda.

Le présent rapport s'articule autour de cinq chapitres. Le chapitre 1 est dédié à mettre notre travail dans son contexte général. Dans ce chapitre, nous introduisons des concepts fondamentaux autour de la thématique de notre travail tels que les systèmes temps-réel, le développement orienté aspect, etc. Le chapitre 2 présente l'étude de l'existant : nous présentons les concepts du langage AspectAda en mettant en relief les manques et les problèmes. Les chapitres 3 et 4 présentent nos contributions : dans le premier, nous présentons en détails les extensions faites dans le langage AspectAda et la `Runtime` ainsi que la nouvelle architecture du compilateur alors que dans le second, nous définissons les règles de tissage utilisées pour générer du code Ada étendu avec les aspects. Le chapitre 5 présente la mise en œuvre de notre approche proposée ainsi que l'étude de cas "système de gestion de charge de travail" utilisée pour appliquer et valider notre travail. Ce rapport sera achevé par une

conclusion et quelques perspectives qui s'ouvrent devant le projet AspectAda.

À la fin de ce mémoire nous présentons deux annexes : (1) l'annexe A contenant des exemples pour tester et valider les règles de tissage de code proposées dans le chapitre 4 et (2) l'annexe B présente le code AspectAda appliqué sur l'étude de cas définie dans le chapitre 5.

Chapitre

1 Contexte Général

1.1 Introduction

L'évolution de la complexité des applications temps-réel engendre une difficulté dans les phases de développement et de maintenance. Cette difficulté conduit à l'apparition de failles nuisant la fiabilité de ces applications. Cependant, la fiabilité et la sûreté de fonctionnement sont deux exigences fondamentales pour ce type de systèmes. Souvent, l'occurrence d'erreurs est inévitable. Elle peut être causée par (1) des facteurs internes d'origine logiciel ou matériel tel que le vieillissement d'un équipement ou (2) des facteurs externes tels que les bugs dûs aux interventions humaines, les effets de bord, les catastrophes naturelles, etc. Dans le contexte des systèmes temps-réel, les défauts peuvent entraîner des catastrophes tels que la perte d'argent, de temps ou de vie humaine. D'où la nécessité de prévoir des techniques assurant la sûreté de fonctionnement des systèmes temps-réel distribués. La tolérance aux pannes (*Eng. Fault tolerant*) est l'un des moyens assurant la fiabilité d'un système. Elle est définie par la capacité du système de continuer à fournir ses services, même en présence d'erreurs ou des dégradations dans le comportement. Elle est basée sur la détection puis le recouvrement des erreurs. La tolérance aux pannes étant une préoccupation transversale, il est préférable d'isoler son implantation du code métier. Le paradigme de développement orienté Aspect (AOSD) [FECA05] a été proposé pour séparer l'implantation des préoccupations transversales de celles fonctionnelles.

Le contexte de recherche dans le quel s'inscrit notre travail de mastère est l'intégration du paradigme orienté aspect dans le développement des systèmes temps-réel. Nous nous

intéressons aux systèmes temps-réel implantés avec le langage Ada [WG05] puisque ce dernier a prouvé sa capacité pour un meilleur support du temps-réel. Nous voulons étendre les systèmes temps-réel développés en Ada par la notion d'aspect pour assurer une meilleur lisibilité du code et faciliter sa maintenance.

Par ailleurs, dans ce chapitre nous plaçons notre travail de mastère dans son cadre général. Nous commençons tout d'abord par la présentation des systèmes temps-réel en citant les contraintes non fonctionnelles qu'ils doivent respecter. Ensuite, nous définissons le langage Ada en tant que langage de programmation le plus adéquat pour l'implantation des systèmes temps-réel critiques avec lequel nous justifions le choix du langage AspectAda. Puis, nous introduisons le paradigme de développement orienté aspect comme solution pour développer les contraintes transversales et nous citons les langages existants supportant ce paradigme. Enfin, nous clôturons ce chapitre par la définiton du cadre spécifique de notre travail permettant de soulever notre problématique principale et nos objectifs.

1.2 Systèmes temps-réel

Dans cette section, nous donnons une définition claire des systèmes temps-réel en exposant leurs caractéristiques majeures. Puis, nous présentons les contraintes associées à ce type de système.

1.2.1 Définition

Un système temps-réel [BD99] est défini comme un système dont la réponse doit être logiquement correcte et produite avant une échéance donnée. Un résultat est considéré correct si et seulement si :

(1) Il satisfait les attentes fonctionnelles en assurant l'exactitude des traitements effectués ;

(2) Il est délivré avant le dépassement des échéances.

Un système temps-réel se compose de plusieurs sous-systèmes respectant des contraintes temporelles plus ou moins strictes.

Temps réel dûr Vs temps-réel mou

- Temps réel dûr (*Eng. Hard real-time*) : ces systèmes sont appelés critiques dont les tâches doivent absolument respecter les échéances prévues. Un retard implique des dégâts importantes (par exemple des vies humaines sont en jeu). Exemples : pilote automatique d'avion, contrôleur de vitesse pour voiture, etc.
- Temps réel mou (*Eng. Soft real-time*) : ces systèmes sont appelés non-critiques dont certains retards connus à l'avance sont acceptables dans certaines limites. Exemples : encodeur/décodeur MPEG, systèmes de visioconférence, etc.

1.2.2 Caractéristiques majeures

Il existe certaines caractéristiques indispensables pour tout système temps-réel : la prévisibilité, le déterminisme et la fiabilité.

La prévisibilité [BD99] permet de déterminer à un stade précoce les paramètres liés aux contraintes temporelles des tâches tels que leurs durées de calcul, leurs échéances, leurs périodicités, leurs taux d'utilisation du processeur, etc. Ces paramètres permettent de choisir l'algorithme d'ordonnancement le plus adéquat de façon que le comportement du système soit prévisible.

Le déterminisme [BD99] est l'objectif visé par les systèmes temps-réel. Un système temps-réel est déterministe si et seulement si les temps d'exécution des tâches sont bornés. Dans le cas de système critique, nous visons à connaître si toutes les tâches respectent leurs échéances. Dans le cas de système non-critique, nous avons intérêt à connaître le temps de réponse moyen des tâches et non pas le pire cas.

La fiabilité [ALRL04] est équivalente à la sûreté de fonctionnement. Elle est définie par la capacité à offrir un service dont nous pouvons faire confiance. Ce concept est étroitement lié aux concepts de la disponibilité, la sécurité, l'intégrité et la maintenabilité.

Toutes ces propriétés doivent être satisfaites, même en présence de menaces de sûreté de fonctionnement. Dans le but d'assurer la fiabilté, les systèmes temps-réel sont conçus de manière à être tolérant aux pannes.

Dans le contexte des systèmes temps-réel, ces exigences non fonctionnelles sont au même pied d'égalité des exigences fonctionnelles.

1.2.3 Restrictions pour les systèmes critiques

Les systèmes temps-réel sont présents dans des applications qualifiées critiques comme les applications médicales, spatiales, aéronautiques, etc. Le développement de ces systèmes requièrent plus de prudence par rapport au développement des systèmes classiques. Toute source d'indéterminisme doit être évitée très tôt dans le processus de développement. Parmi les facteurs qui engendrent des variations dans le système induisant le non-déterminisme, nous citons :

- variation des temps d'exécution des tâches : par exemple une tâche parfois s'exécute dans 10 secondes comme elle peut durer 50 secondes.
- interruptions : en cas d'interruption le système doit réagir immédiatement.
- erreurs et échecs matériels ou logiciels.

Par conséquent, le code d'une application doit respecter certaines restrictions afin d'interdire certains concepts de programmation nuisant au déterminisme du système :

1. Interdire l'allocation dynamique de mémoire

L'allocation dynamique de mémoire peut d'une part compromettre le déterminisme d'un système temps-réel [PPS02]. Et d'autre part elle peut être source de gaspillage dans les ressources en mémoire du système qui sont généralement limitées. Elle pose un problème lors des calculs des pires temps d'exécution (*Eng. Worst Case Execution Time, WCET*) des tâches. Ceci est dû à la fragmentation de la mémoire causée par la désallocation et la réallocation. En effet, l'estimation du temps de recherche d'un espace

libre dans une mémoire fragmentée est pratiquement imprévisible. De plus, comme la manipulation de la mémoire est délicate alors le programmeur peut facilement commettre des erreurs engendrant des fuites de mémoire. Ces fuites constituent les sources de gaspillage dans les ressources en mémoire.

2. Éviter certains concepts orienté-objet

La compatibilité du paradigme orienté-objet avec les systèmes temps-réel est discutée dans [FA04]. La technologie orientée objet introduit les concepts d'héritage et de polymorphisme basés sur le mécanisme d'aiguillage dynamique. La manière d'implantation de l'aiguillage dynamique dans la plupart des compilateurs orientés objet est nuisible au déterminisme et à la sûreté de fonctionnement des applications temps-réel. Afin de réaliser l'aiguillage dynamique, les compilateurs construisent automatiquement des tables appelés "tables d'aiguillage". Ces tables ne subissent aucune analyse et échappent à tout contrôle du programmeur ce qui rend le flux de contrôle n'est pas connu à l'avance. Ceci compromet l'analysabilité statique du système. Pour pallier ces problèmes, les auteurs de [FA04] citent les recommandations nécessaires pour pouvoir bénificier des avantages de l'orienté objet (héritage, encapsulation, etc.) dans les systèmes avioniques en évitant les concepts non compatibles avec ces systèmes. Ils ont interdit l'aiguillage dynamique et le polymorphisme.

L'implantation des systèmes temps-réel critiques nécessite un support fort pour assurer un haut niveau d'intégrité. Depuis sa création, le langage Ada est ciblé au développement des systèmes temps-réel. La version actuelle du langage Ada a une approche mûre pour le traitement des systèmes à base de priorité. Comparé par les technologies en temps-réel actuelles, Ada est techniquement supérieur à ses concurrents. Dans la section suivante, nous montrons les critères puissants du langage Ada pour le support du temps-réel.

1.3 Ada langage de programmation pour les systèmes temps-réel

Ada [WG05] est un langage de programmation généraliste supportant à la fois l'orienté objet et le procédural. Il synthétise les avantages des langages existants et les intègre dans un ensemble cohérent. Il s'agit d'une norme internationale. Cette norme est utilisée pour assurer une validation rigoureuse des nouveaux compilateurs. Ce langage a prouvé sa puissance dans le cadre de la réalisation des systèmes embarqués avec de fortes contraintes temps-réel. En effet, il permet de construire des programmes sûrs grâce à sa syntaxe claire et non ambiguë. Compraré à d'autres langages de programmation (C, C++, Java, etc), Ada offre certaines garanties et minimise le coût de développement et d'intégration des applications. Ce langage dispose d'un compilateur puissant capable de détecter les erreurs à la phase de compilation ce qui minimise la probabilité de propagation des erreurs à l'exécution. De plus, Il offre une forte sémantique favorisant les optimisations. Notamment, Ada offre plusieurs moyens pour un support fort du temps-réel :

- Support de la concurrence d'une manière intrinsèque (le mot clef **task**),
- Support du parallèlisme, des rendez-vous...
- Support des objets protégés et partagés,
- Support de l'horloge et gestion du temps.

Ada propose deux approches (Profil Ravenscar [BDV04], SPARK [Bar03]) dans le but de traduire les contraintes nécessaires pour la réalisation des systèmes critiques et les imposer au développeur.

1.3.1 Profil Ravenscar

Le profil Ravenscar [BDV04] a été créé afin d'imposer une suite de restrictions sur un programme Ada. Le respect de ces restrictions permet la production de programmes efficaces et prévisibles. Ces restrictions limitent l'usage du langage Ada pour garantir une analyse statique d'ordonnancement et l'absence d'interblocage et de famine par construc-

tion .

Pour garantir une analysabilité statique des tâches, Ravenscar recommande l'utilisation des tâches périodiques et sporadiques et il interdit l'utilisation des tâches qui peuvent être déclenchés à un instant inconnu. Pour garantir l'absence d'interblocage et de famine, Ravenscar recommande une communication asynchrone entre les tâches via des objets partagés. Ces objets doivent être protégés à la concurrence de manière que deux tâches ne peuvent pas accéder à l'objet simultanément. Ainsi un problème de blocage peut survenir lors de l'attente des tâches pour accéder aux objets partagés. Pour pallier ce problème, Ravenscar impose l'utilisation du protocole de plafonnement de priorité (*PCP, Priority Ceiling Protocol*) [SRL90].

1.3.2 SPARK

Ada peut être combiné avec différentes technologies de preuve formelle afin de prouver formellement certaines propriétés du code. L'approche SPARK [Bar03] est un sous-ensemble du langage Ada qui utilise les méta-informations annotées (sous la forme de commentaires Ada). Elle permet d'avoir un programme Ada correct par construction grâce à des théorèmes établis et prouvés. Elle est destinée pour les systèmes de haute intégrité comme les systèmes avioniques, spatiaux, médicaux, bancaires, nucléaires, etc. SPARK vise à exploiter les points forts d'Ada tout en essayant d'être plus sûr et plus sécurisé par l'intermédiaire des annotations qui expriment des contraintes d'intégrité telles que les droits d'accés, la concurrence, etc.

Une fois les annotations sont ajoutées au code Ada, un outil spécifique appelé "examinateur" analyse le programme SPARK/Ada et évalue la conformité du code source Ada aux annotations. S'il est bien conforme alors il est possible de passer à un compilateur Ada pour compiler le code source (les annotations seront ignorées par le compilateur puisqu'elles sont sous la forme de commentaires).

Certaines propriétés non fonctionnelles liés aux systèmes temps-réel sont transversales.

Il vaudrait mieux les implanter indépendamment du code fonctionnel du système pour favoriser la lisibilité et l'évolution du code. Le développement orienté aspect offre les moyens nécessaires pour séparer les propriètés non fonctionnelles des descriptions métier. Dans la section suivante, nous définissons le paradigme de développement orienté aspect et nous expliquons ses concepts fondamentaux.

1.4 Développement orienté aspect de logiciel

La complexité des systèmes ne cesse de s'accroître chaque jour dans le monde entier en vue d'être plus fiables et plus sécurisés. Cette complexité est dûe à l'importance attribuée aux préoccupations transversales (*Eng. Crosscutting concerns*) telles que la tolérance aux pannes, la sécurité, la journalisation, etc. Ceci est fait au détriment d'une mauvaise lisibilité du code induisant à la difficulté de la maintenance. Le paradigme de développement orienté aspect de logiciels, (Aspect Oriented Software Developement, AOSD) [FECA05] est apparu pour pallier ces problèmes. Il assure la séparation des considérations techniques des descriptions métier tout au long du cycle de développement logiciel.

Dans cette section, nous commençons par une introduction à la programmation orientée aspect (POA) [KLM+97]. Ensuite, nous présentons ses concepts de base. Enfin, nous clôturons par les avantages apportés.

1.4.1 Définition de la POA

L'AOSD est un paradigme qui vise à améliorer la modularité du code et assurer une meilleure réutilisation en séparant la programmation des deux types de préoccupations (transversales/fonctionnelles). La séparation des préoccupations est effectuée à travers l'isolation du traitement d'une préoccupation transversale dans une unité autonome appelée "Aspect" en laissant le code fonctionnel dans des modules indépendants. La figure 1.1 montre l'aspect de séparation des préoccupations.

En outre, l'AOSD offre des concepts (*pointcuts*, *joinpoints*, *advices*) et des mécanismes

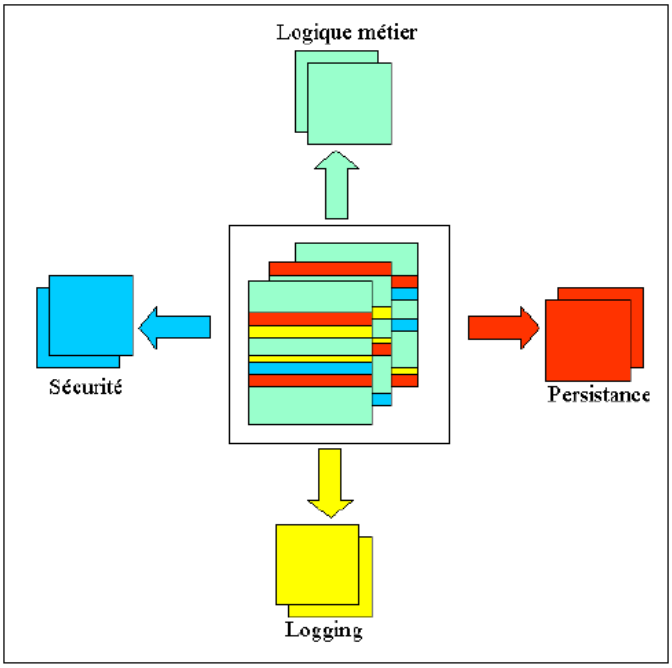


Figure 1.1 – Séparation des préoccupations

(interception, tissage) permettant de faire la liaison entre les préoccupations transversales et les descriptions métier. Nous reviendrons plus en détails sur ces notions dans la section suivante.

Dans ce contexte, différents langages de programmation d'aspect POA (Aspect Oriented Programmation, AOP [KLM+97]) ont apparu dans le but d'étendre des langages existants par le concept d'aspect. Ces langages peuvent être mis en œuvre aussi bien avec des langages orienté-objet comme java, qu'avec un langage procédural comme C. Nous citons quelques langages orienté aspect associés aux langages de programmation les plus connus :

- Java : AspectJ [KHH+01], JAC (*Java Aspect Components*) [PDFS02], JBoss [tea02], CeasarJ [MO03], ComposeJ [Wic99]
- C/C++ : AspectC [CKFS01], AspectC++ [SGSP02]
- C#/.NET : AspectC# [Kim02], Loom.NET [SP02]
- Ada : AspectAda [PC05]

1.4.2 Concepts de base de la POA

Comme nous l'avons déja montionné, si nous vonlons intégrer l'orienté aspect à une application alors nous devons diviser l'application en deux parties :

(1) L'application de base implantant les fonctionnalités métier ;

(2) Les modules orientés aspect implantant les critères transversaux.

La POA a défini différents concepts et mécanismes permettant la définition et l'intégration des propiètés non-fonctionnelles dans le code fonctionnel.

- **Points de jonction** (*joinpoints*) : appelés aussi points d'insertion permettant d'établir la liaison entre le comportement transversal et le code métier de l'application. Nous définissons les points de jonction afin de désigner des moments bien précis lors de l'exécution de l'application où nous voulons insérer le comportement transversal.
- **Point de coupure** (*Pointcut*) : il permet de regrouper un ensemble de points de jonction.
- **Code de l'advice** (*Advice*) : c'est la portion de code qui définit le comportement transversal. Un *advice* est associé à un *pointcut*. Un code *advice* est exécuté chaque fois où l'exécution atteint un *joinpoint* appartenant à l'ensemble défini par le *pointcut* correspondant. Il peut être exécuté avant (*Before*), après (*After*) ou autour (*Around*) d'un *joinpoint*.
- **Aspect** : un aspect sert pour la modularisation d'une préoccupation spécifique [FECA05].

En outre, la POA définit aussi deux mécanismes intéressants :

- **Interception** : c'est l'action d'intercepter un moment (défini par le *joinpoint*) de l'exécution de l'application pour insérer un comportement spécifique.
- **Tissage** (*weaving*) : cette opération permet d'intégrer le code des advices dans le code métier de l'application aux points de jonction définis par leurs pointcuts correspondants. Elle s'effectue par une entité spécifique appelée tisseur (*weaver*). Il existe deux types de tisseurs : les tisseurs statiques où le tissage des aspects se fait au mo-

ment de la compilation et les tisseurs dynamiques où le tissage se fait au moment de l'exécution.

La figure 1.2 présente le principe de la programmation orientée aspect.

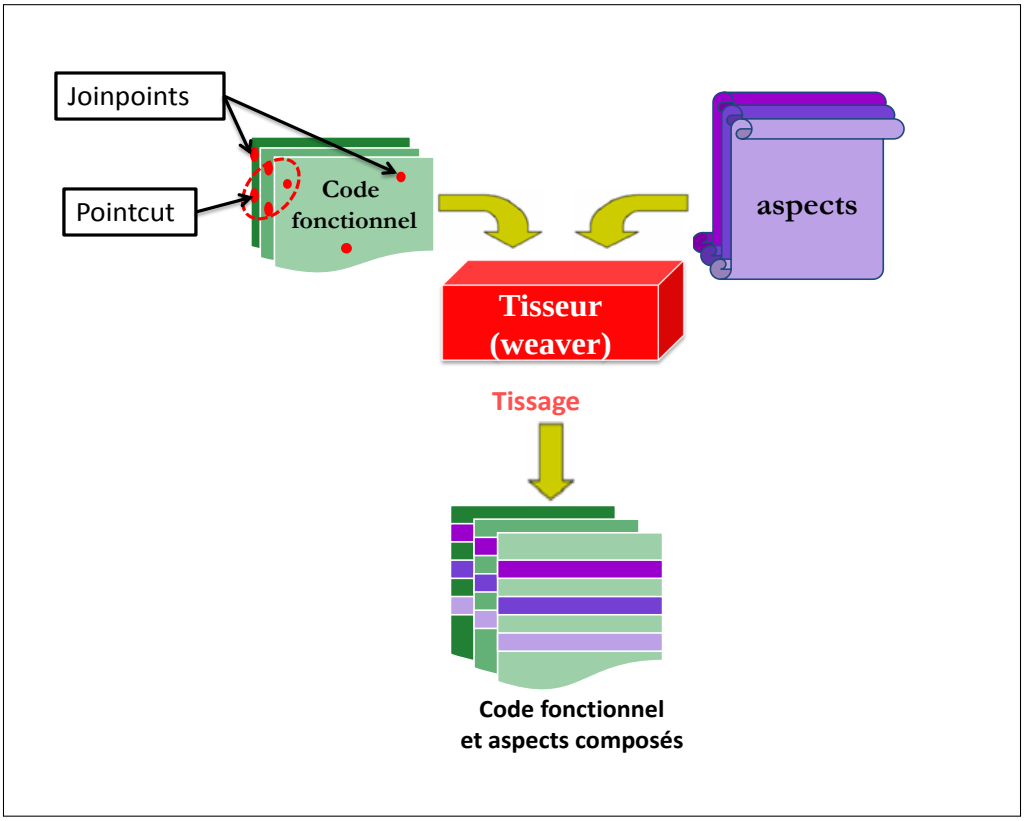


Figure 1.2 – Principe de la programmation orientée aspect

1.4.3 Apports de la POA

L'objectif principal de la POA n'est pas de remplacer les paradigmes actuellement utilisés, mais de les améliorer. L'intégration manuelle des traitements décrivant les comportements transversaux dans le code métier de l'application a posé certains problèmes parmis lesquels :

Dispersion de code (*Eng. code scattering*) : C'est la difficulté de limiter la portée d'une préoccupation transversale à un seul module. L'exemple très courant de gestion de

traces introduit bien le problème. Si, nous voulons afficher une trace après l'exécution de chaque sous-programme, nous serons obligés d'ajouter une ligne de code à la fin du traitement de chacun d'eux. Nous obtenons donc une redondance de code dans la totalité de l'application pour supporter la fonctionnalitée de gestion de trace.

Enchevêtrement du code (*Eng. code tangling*) : C'est la conséquence de traiter plusieurs préoccupations dans un même module. Le développeur du système peut traiter dans un même module plusieurs préoccupations simultanément (logique métier, journalisation, sécurité, etc).

Ces problèmes influent sur la qualité, l'évolution, la maintenabilité et la réutilisation du code. La figure 1.3 expose les deux problèmes cités.

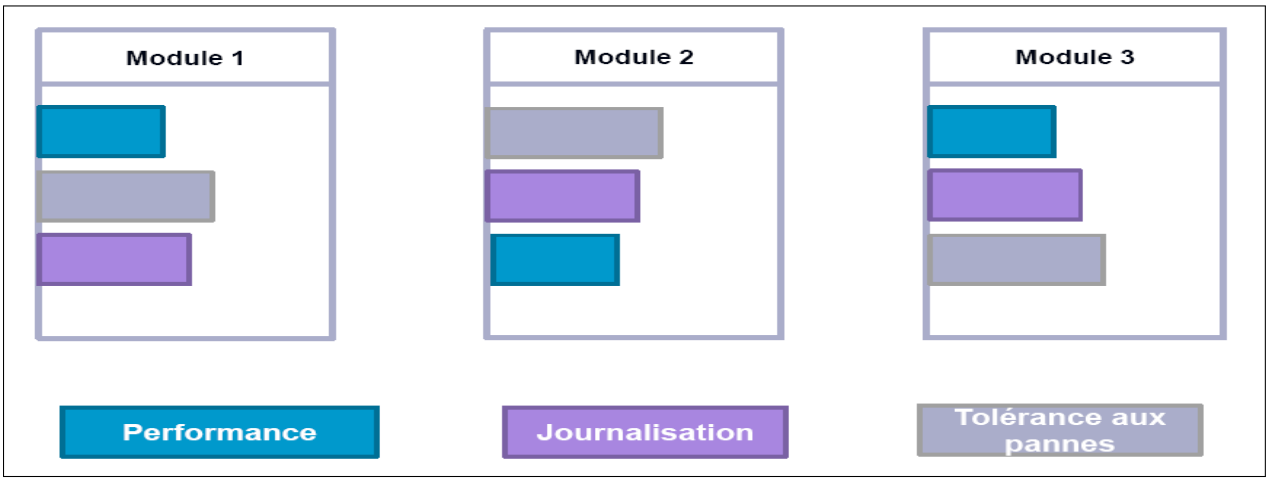


Figure 1.3 – Problèmes de dispersion et d'enchevêtrement du code

La POA a pu résoudre ces problèmes grâce à la séparation entre les préoccupations transversales et les préoccupations fonctionnelles. Elle a garanti l'amélioration de qualité de code en évitant les redondances ce qui facilite sa compréhension et son évolution. De plus, une préoccupation est encapsulée dans une unité autonome appelée "aspect" ce qui favorise la réutilisation de code. Nous pouvons par exemple développer un aspect implantant la gestion de traces qui sera réutilisé dans différentes applications sans avoir besoin de faire la moindre modification.

1.5 Cadre, problèmatique et objectif

À l'issue de l'étude des concepts fondamentaux de notre travail (les systèmes temps-réel, le langage Ada, la POA), nous avons amené à mettre notre travail dans son cadre spécifique en soulevant notre problèmatique principale permettant ensuite de déduire l'objectif que nous visons à réaliser.

1.5.1 Cadre spécifique du travail

Ce travail de Mastère s'inscrit dans le cadre d'un travail de thèse qui vise à concevoir un processus de développement permettant d'intégrer dès le stage de modélisation des éléments de la tolérance aux pannes pour ensuite donner lieu à une génération de code tolérant aux pannes pour les systèmes temps-réel distribués.

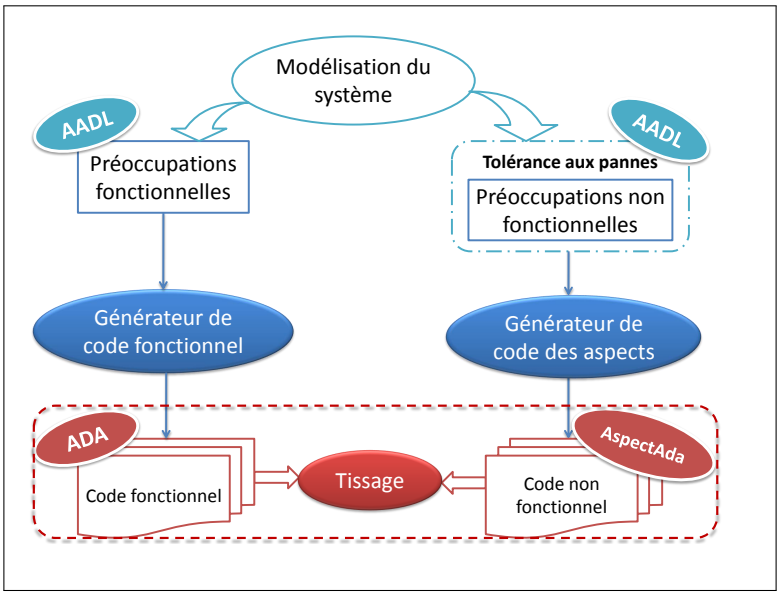


Figure 1.4 – Processus de développement proposé

Comme le montre la figure 1.4, l'approche proposée décrit un processus de développement intégrant les exigences de tolérance aux pannes à l'étape de la modélisation. Elle utilise le langage de description d'architecture AADL [SAE04] (*Architecture Analysis & Design Language*) pour modéliser les préoccupations fonctionnelles et non fonctionnelles

particulièrement les exigences de la tolérance aux pannes. Ensuite, un générateur de code est utilisé pour avoir le code correspondant au modèle AADL. Le code généré est écrit en Ada [WG05] puisque nous travaillons dans le contexte des systèmes temps-réel dont Ada est le langage de programmation le plus adéquat pour les raisons citées dans la section 1.3.

La tolérance aux pannes étant une préoccupation transversale, il vaudrait mieux ne pas intégrer le code source l'implantant directement dans le code métier pour le garder lisible et facilement maintenable. C'est dans ce cadre que la programmation orientée aspect [KLM+97] (POA) sera utilisée. Pour des raisons de conformité, le code implantant les éléments de la tolérance aux pannes doit être développé en AspectAda [PC05] afin d'être tissé automatiquement dans le code fonctionnel généré en Ada.

1.5.2 Problèmatique principale et objectif visé

La POA est fondée sur l'opération de tissage. Le tissage est une opération automatique permettant d'intégrer dans le code d'une application les aspects créés séparément. Dans le contexte des systèmes temps-réel, cette opération peut compromettre le déterminisme en insérant dans le code de l'application des constructions pouvant violer les contraintes temps-réel (détaillées à la section 1.2.3) et échappant à tout contrôle du développeur.

L'objectif principal de ce travail de mastère est l'intégration du paradigme orienté aspect dans le développement des systèmes temps-réel. Nous visons à étudier un mécanisme d'Aspect existant pour permettre de l'adapter et le modifier pour le respect des contraintes temps-réel. Nous partirons du projet AspectAda [PC05] (extension d'aspect pour le langage Ada). Ce choix est justifié par le fait que Ada et un langage très bien adapté aux systèmes temps-réel. Nous visons à étudier ce langage vis à vis sa syntaxe, sa sémantique, sa bibliothèque standard (Runtime), son compilateur/tisseur et particulièrement l'opération de tissage. En étudiant ces éléments, nous essayons d'extraire d'une manière exhaustive les limites de ce langage. Après l'étude de l'existant et l'extraction des limitations, nous essayons de trouver les remèdes adéquats.

1.6 Conclusion

Dans ce chapitre, nous avons introduit les systèmes temps-réel en nous focalisant sur les contraintes qu'ils doivent respecter. Ensuite, nous avons défini le langage de programmation Ada en tant que langage de programmation très bien adapté pour les systèmes temps-réel. Puis, nous avons présenté le paradigme de développement de logiciel orienté aspect (AOSD) permettant la séparation des considérations techniques de la logique métier de l'application. Nous avons montré les bénifices de ce paradigme au niveau de la qualité, la modularité et la réutilisation du code. Enfin, nous avons spécifié le cadre auquel s'inscrit notre travail qui nous ramène à soulever la problèmatique et l'objectif visé.

Dans le chapitre suivant, nous étudions le langage AspectAda. Nous allons explorer sa syntaxe, son tisseur, son compilateur, sa bibliothèque de routines prédéfinies, etc dans le but d'extraire les problèmes et les manques et proposer des solutions.

Chapitre

2 Étude de l'existant

2.1 Introduction

Les langages de programmation classiques n'ont pas la capacité d'assurer la modularisation des contraintes non fonctionnelles des systèmes temps-réel. Le paradigme orienté aspect peut être appliqué pour permettre une mise en œuvre modulaire de ces préoccupations. Les langages de programmation existants ont été étendus avec de nouvelles fonctionnalités pour supporter l'orienté aspect. Par exemple AspectJ [KHH+01] (extension d'aspect pour le langage Java) est le langage d'aspect le plus complet et le plus connu parmi les langages d'aspect existants. Cependant, Java est inadéquat pour tous types de systèmes et particulièrement pour les systèmes temps-réel. Notamment, la seule solution gratuite de l'RTSJ (*Real Time Specification for Java*) [Aut09] de Sun n'est plus disponible en version académique. En contre partie, le langage Ada fournit un support riche pour le développement des systèmes temps-réel qui manque aux autres langages de programmation.

Dans ce chapitre, nous présentons une étude du langage AspectAda existant (extension d'aspect pour le langage Ada). Nous entamons cette étude par introduire les différents concepts supportés par AspectAda. Puis, nous passons à l'exploration et le test de son outil compilateur et tisseur "*AspectAda*" et sa bibliothèque de fonctions "*Runtime*". Au fur et à mesure de cette étude, nous essayons d'extraire les lacunes du langage AspectAda de point de vue syntaxe, outils et bibliothèque.

2.2 Introduction au langage AspectAda

AspectAda est un projet de recherche débuté en 2003 pour étendre le langage Ada95 avec les concepts d'aspect. L'équipe productrice de ce projet (Knut H. Pedersen et al.,Concordia University) a publié la première version du langage AspectAda en 2005 [PC05]. Après sa publication, le projet n'a pas évolué suite à la dissociation des membres de l'équipe.

Le langage AspectAda est inspiré du langage AspectJ. Ce dernier est un produit open source, avec de nombreux contributeurs, qui a évolué depuis plusieurs années. AspectAda a adopté, les *joinpoints*, les *pointcuts*, les aspects et les advices. Toutefois, AspectAda a traduit ces concepts à des notions plus proches du jargon Ada pour être en conformité avec Ada. AspectAda contient toutes les constructions Ada95 en ajoutant des extensions. Il étend le langage Ada par deux nouvelles unités de compilation :

1. ***Aspect*** : est une unité de programme contenant les types des aspects et les codes des advices mélangés avec le code Ada classique. Un Aspect est paramétré par au moins un *pointcut*.
2. ***Weaver*** : est une unité de programme contenant (1) la définition des pointcuts permettant d'intercepter des collections de *joinpoints* et (2) l'instanciation des aspects pour associer les pointcuts aux aspects et particulièrement aux advices correspondants.

Dans la suite, nous allons détailler et discuter la syntaxe de chacune de ces notions en utlisant des exemples simples.

2.2.1 Points de jonction (*Joinpoints*)

Comme nous avons déja vu à la section 1.4.2 du chapitre 1, un *joinpoint* sert à désigner des moments bien précis lors de l'exécution de l'application où le comportement transversal doit être inséré. Un joinpoint peut désigner par exemple l'exécution ou l'appel d'un sous-programme. Le listing 2.1 montre un exemple d'un point de jonction dans AspectAda :

l'exécution de la procédure Somme du package Calcul.

Listing 2.1 – Exemple de joinpoint

```
1 with Ada.Text_IO;
2 package body Calcul is
3 -- ...
4   function Somme (A,B : in Integer) return Integer is
5   begin
6       return (A + B) ;
7   end Somme;
8 -- ...
9 end Calcul;
```

Disscussion

Dans [PC05], les auteurs affirment : "*Les joinpoints supportés par le langage AspectAda peuvent être l'appel ou l'exécution d'un sous-programmes ou d'une entrée, l'initialisation et la finalisation de types contrôlés (controlled types) et l'élaboration d'un package*". Cependant, Ceci se contredit avec ce que nous avons trouvé dans la grammaire qu'ils ont définie. Le listing 2.2 est un extrait de la grammaire des pointcuts. Ce listing montre que seules les exécutions "**execution**" des sous programmes (procédures ou fonctions) d'un paquetage sont supportés.

Listing 2.2 – Règle de grammaire spécifiant l'expression basique du pointcut

```
1 basic_pointcut_expr ::= (pointcut_expr)
2    | execution (method_pattern)
3    | identifier
```

La grammaire de AspectAda est fortement influencée par celle de AspectJ. La règle de la grammaire citée dans le listing 2.2, utilise l'attribut '*method_pattern*'. Toutefois, la notion de "*methode*" n'existe pas en Ada. Elle est remplacée par les sous programmes. Par ailleurs, le modèle des *joinpoints* offert par le langage AspectAda est pauvre. Contrairement à AspectJ qui présente une référence de la POA, définit un modèle riche de *joinpoints* tels que les appels ou les exécutions de méthodes, l'exécution d'un object constructeur ou destructeur, ou l'initialisation statique d'une classe. Le modèle de *joinpoints* de AspectAda peut être étendu pour supporter les appels des sous programmes en plus de leurs exécution. De plus, le langage Ada définit des constructions intéressantes pour le temps-réel (1) les entrées sur les objets protégés (*protected objects*) et sur les tâches (*tasks*) et (2) la création des tâches. Ces constructions peuvent constituer des points de jonction. Les extensions

faites sur le modèle des *joinpoints* sont communiquées dans la section 3.2.1 du chapitre 3.

2.2.2 Pointcuts

Un pointcut décrit une collection de joinpoints par l'intermédiaire de son expression très particulière. Le langage d'expression des pointcuts offre un moyen simple pour spécifier les patrons (*patterns*) des identifiants des sous programmes, des types de leurs paramètres et des packages. Pour regrouper un ensemble de *joinpoints*, le langage d'expression des pointcuts fait appel aux mécanismes de quantification [1] en introduisant les *wildcards* tel que '`*`', '`+`' ou '`..`' ainsi que les opérateurs logiques "`and`", "`or`", "`xor`" et "`not`".

Dans une expression d'un pointcut, un patron de recherche peut effectuer le filtrage de l'identifiant et des paramètres du joinpoint.

Les définitions des pointcuts sont encapsulées dans une unité de programme spécifique en AspectAda nommé ***weaver*** contenant, en plus des *pointcuts*, l'instanciation des aspects génériques (voir section 2.2.5).

Le listing 2.3 présente deux exemples de pointcuts. Le pointcut `Create_file_PC` désigne les sous-programmes du package `Text_IO` dont le nom commence par `Create` et possèdant un ou plusieurs paramètre(s) sachant que le premier paramètre est de type `File_Type`. Le wilcard '`..`' désigne n'importe quel profile de paramètres. Le pointcut `File_Management_PC` permet de réunir plusieurs pointcuts en utilisant l'opérateur '`or`'.

Listing 2.3 – Exemple d'un pointcut

```
1  weaver Text_IO_Composition_Rules is
2     Create_File_PC : Pointcut := execution Text_IO.Create* (File_Type, ..);
3
4     File_Management_PC : Pointcut := Create_File_PC or
5                                      execution Text_IO.Open (..) or
6                                      execution Text_IO.Close (..) or
7                                      execution Text_IO.Delete (..) or
8                                      execution Text_IO.Reset (..);
9     -- ...
10 end Text_IO_Composition_Rules;
```

Discussion

À l'issue de notre étude de la grammaire du langage d'expression des pointcuts de

1. La quantification est la capacité des aspects d'affecter de multiples points dans le programme.

AspectAda, nous avons constaté que celle-ci est fortement influencée par le langage d'-expression des pointcuts de AspectJ. Cette influence a créé une certaine contradiction entre AspectAda et Ada. Comme AspectJ, AspectAda considère tous les sous-programmes comme des méthodes java. Cependant, Ada distingue deux types de sous-programmes les procédures et les fonctions. Pour que les expressions des pointcuts soient en conformité avec Ada, il est préférable de spécifier la catégorie du *joinpoint* à intercepter (*procedure*, *function*, etc.).

De plus, nous avons remarqué que la grammaire des expressions des pointcuts est mal soignée. Nous citons à titre d'exemple, un extrait des règles de la grammaire (listing 2.4) permettant de spécifier le type d'un paramètre en utilisant les opérateurs logiques. D'après ces règles (lignes 2,3), le développeur peut spécifier un type d'un paramètre en faisant (T1 **and** T2) ce qui est logiquement absurde de dire un paramètre peut à la fois être de type T1 et T2 (idem pour **xor**).

Listing 2.4 – Règles des patrons d'expressions des types des paramètres

```
1  ...
2  type_pattern_expr ::= or_type_pattern_expr
3      | type_pattern_expr and or_type_pattern_expr
4
5  or_type_pattern_expr ::= unary_type_pattern_expr
6      | or_type_pattern_expr or unary_type_pattern_expr
7      | or_type_pattern_expr xor unary_type_pattern_expr
8
9  unary_type_pattern_expr ::= basic_type_pattern
10     |  not unary_type_pattern_expr
11 ...
```

En outre, il existe certains critères très intéressants pour la recherche de *joinpoints* sont négligés dans les expressions des pointcuts tels que le type de retour d'une fonction (s'il s'agit d'intercepter des fonctions) ainsi que les modes des paramètres. Dans la section 3.2.1 du chapitre 3, nous détaillons les modifications effectuées sur la syntaxe et la sémantique des expressions des pointcuts.

2.2.3 Types d'aspect et advices

Un advice sert à définir le code implantant le comportement de l'aspect pour être exécutés en atteignant les *joinpoints* désignés par le pointcut correspondant. Un pointcut peut

être vu comme un ensemble d'évènements pour lesquels certains advices seront exécutés. Il existe trois moyens pour associer un advice à un pointcut :

(1) Exécuter l'advice avant le pointcut : il s'agit d'un advice *before*,

(2) Exécuter l'advice après le pointcut : il s'agit d'un advice *after*,

(3) Exécuter l'advice au lieu d'exécuter le pointcut en offrant au pointcut la possibilité pour reprendre son exécution : il s'agit d'un advice *around*.

Selon [PC05], les advices sont déclarés pour un type d'aspect (*aspect type*) comme la déclaration des sous-programmes pour les *tagged types* dans la programmation orientée-object en Ada. Ceci dans le but de partager un état (*state*) entre les exécutions des advices liés ou bien entre les exécutions d'un même advice. La bibliothèque standard de AspectAda "`Runtime`" définit deux types parents de tous types d'aspect : **`Simple_Aspect`** et **`Detailed_Aspect`**. Le type `Detailed_Aspect` est utilisé lorqu'il y a accès aux informations du joinpoint courant. Simple_Aspect est utilisé dans le cas contraire. Un nouveau type d'aspect doit être créé à partir de l'un des deux types prédéfinis. Le listing 2.5 montre un exemple de déclaration d'un type d'aspect `Logger_A` qui dérive du type prédéfini `Detailed_Aspect`. `Logger_A` possède un état (*state*) `Log_count` qui sert à compter le nombre de fois de l'exécution de l'advice.

Listing 2.5 – Exemple de déclaration d'un type d'aspect

```
1 type Logger_A is new Aspect_Ada.Detailed_Aspect with
2    record
3        Log_Count : Natural := 0;
4    end record;
```

Syntaxiquement la déclaration et l'implantation d'un advice AspectAda sont similaires à celles d'une procédure Ada. La seule différence est au niveau du mot clé **procedure** est remplacé par **advice**. Mais de point de vue invocation, ils sont totalement différents. Une procédure est exécutée suite à un appel. Par contre, un advice est exécuté au moment où l'exécution du code de l'application atteint un joinpoint désigné par le pointcut correspondant. Les advices doivent être associés aux pointcuts. Pour ce faire, AspectAda utilise la clause "**for**...**use**" de Ada et un nouveau attribut "**pointcut**" indiquant le pointcut associé à l'advice. La clause est placée juste après la déclaration de l'advice (listing 2.6).

Listing 2.6 – Exemple de déclaration d'un advice Before

```
1 advice Before (Logger : in out Logger_A);
2 for Before'pointcut use Create_File_PC;
```

Comme une procédure Ada, l'implantation d'un advice possède une partie déclarative et un corps. Toute instruction autorisée au sein d'une procédure Ada, elle est également autorisée dans l'implantation d'un advice. Le listing 2.7 illustre un exemple d'implantation d'un advice *Before*. Cet advice incrémente le compteur et laisse une trace du *joinpoint* courant en affichant sa signature et le nombre total de son exécution.

Listing 2.7 – Exemple d'implantation d'un advice Before

```
1 advice Before (Logger : in out Logger_A) is
2 begin
3    Logger.Log_count := Logger.Log_count + 1;
4    Put_Line("execution count : " &
5             Natural'Image(Logger.Log_count) &
6            "of the joinpoint " &
7             Image(Get_Join_Point(Logger_A).all));
8 end Before;
```

Syntaxiquement, un advice *After* est similaire à un advice Before. Un advice *Around* s'exécute au lieu du *joinpoint*. Cela n'implique pas que le *joinpoint* ne s'exécute jamais, puisqu'une instruction spéciale '**proceed**' peut être utilisée pour provoquer l'exécution du *joinpoint*. Le listing 2.8 montre un exemple d'un advice around qui laisse une trace avant et après l'exécution du *joinpoint* courant.

Listing 2.8 – Exemple d'implantation d'un advice Around

```
1  advice Around (Logger : in Logger_A) is
2     begin
3        Ada.Text_IO.Put_Line
4          ("Starting execution => " &
5           Aspect_Ada.Image(Aspect_Ada.Get_Join_Point(Logger).all));
6        Proceed;
7        Ada.Text_IO.Put_Line
8          ("Finishing execution => " &
9           Aspect_Ada.Image(Aspect_Ada.Get_Join_Point(Logger).all));
10    end Around;
```

Discussion

L'étude des types d'aspects et des advices nous ramène à la discussion de certains éléments de la syntaxe. Nous discutons en premier lieu l'utilité des déclarations des types d'aspect. En second lieu, nous montrons les manques relatifs à la syntaxe des advices

around ainsi que l'instruction *proceed*.

1. Utilité des déclarations des types d'aspect

L'idée de définir les advices pour un type d'aspect est inspirée de la programmation orientée-objet de Ada. Ceci est dans le but de mettre la syntaxe de AspectAda en conformité avec celle de Ada. Cependant, il ne s'agit pas du même contexte. D'une part, les types d'aspect sont déclarés pour être utilisés comme types des paramètres des advices. Un paramètre ayant un type d'aspect qui hérite du type `Detailed_Aspect` signifie que l'advice accède aux informations du *joinpoint*. Le type `Detailed_Aspect`, comme il est défini dans la Runtime (listing 2.9), contient un seul champ qui est un pointeur sur le joinpoint.

Listing 2.9 – Extrait de la Runtime présentant les types d'aspect prédéfinis

```
1 ...
2 type Root_Aspect is abstract tagged null record;
3
4 type Simple_Aspect is new Root_Aspect with null record;
5
6 type Detailed_Aspect is new Root_Aspect with record
7       Join_Point : Join_Point_Ptr;
8 end record;
9 ...
```

La question qui se pose : pourquoi pas le développeur utilise directement le type `Join_Point` s'il veut accéder aux informations du *joinpoint* ?

L'utilisation d'un type intermédiaire (`Detailed_Aspect`) ne sert à rien. D'autre part, les types d'aspect sont définis pour partager un état (*state*) entre les advices. Toutefois, il est possible de partager un état entre les advices sans définir un type d'aspect. De plus, un paramètre d'un advice (ayant le type de l'aspect) est réellement déclaré comme variable globale dans l'aspect. Cela veut dire que les paramètres des advices ne sont pas des vrais paramètres. À travers ces arguments, nous pouvons affirmer que le passage par la déclaration des types d'aspect pour partager un état ou pour accéder au contexte du *joinpoint* est inutile puisqu'il complique la syntaxe du langage sans l'enrichir.

Nous présentons dans la section 3.2.2 du chapitre 3 la nouvelle syntaxe des aspects en omettant les types d'aspect.

2. Syntaxe d'un advice *around* et l'instruction *proceed*

Un advice around peut remplacer une fonction Ada. Cependant, sa syntaxe permet seulement de remplacer une procédure. Il faut tenir compte de ceci en ajoutant dans la syntaxe d'un advice around la possibilité de retourner une valeur (la construction '**return**' comme pour une fonction Ada).

Dans un advice *around*, l'instruction *proceed* sert à exécuter le comportement du joinpoint courant. En s'inspirant de AspectJ, les auteurs de [PC05] mentionnent que cette instruction très utile doit figurer dans le langage AspectAda. Cependant, AspectAda n'a pas spécifié une règle de sa grammaire pour cette instruction. Il n'a même pas défini *proceed* parmi les mots clefs du langage. Il la traite comme un appel d'un sous programme Ada sans paramètres appelé '**proceed**'. De plus, cette instruction ne prend pas de paramètres puisque les arguments du *joinpoint* ne sont pas disponibles dans le code de l'advice. En outre, le compilateur ne traite pas cette instruction lors de l'opération du tissage. Il la garde telle qu'elle est dans le code tissé (code Ada généré) au lieu de la remplacer par un appel au joinpoint courant.

Les extensions faites sur la grammaire pour ajouter l'instruction proceed et pour corriger la syntaxe d'un advice *around* se trouvent à la section 3.2.2 du chapitre 3.

2.2.4 Aspects

Un Aspect est l'unité de modularisation pour les types d'aspect (aspect types) et les advices. Syntaxiquement, un aspect est similaire à un paquetage Ada. Il est défini par une spécification (*aspect specification*) et une implantation (*aspect body*). Toute construction permise au sein d'un paquetage Ada est également autorisée dans un aspect. Outre ces constructions, un aspect contient la déclaration des types d'aspect et des advices.

Dans l'ancienne version de AspectAda, la spécification d'un aspect doit contenir au moins une déclaration d'un type d'aspect et une déclaration d'un advice qui prend le type de l'aspect comme paramètre. Le corps de l'aspect (*aspect body*) doit contenir les implantations des entités (advices, sous-programmes, etc) déclarées dans la spécification

de l'aspect.

Un advice défini dans un aspect donné est associé à un pointcut. AspectAda sépare les définitions des pointcuts de celles des spécifications et des implantations des advices. Afin d'assurer la liaison entre les pointcuts et les advices, un aspect est paramétré par des pointcuts. Pour ce faire, AspectAda a exploité le concept des unités génériques (*generic units*) de Ada. Par conséquent, les aspects en AspectAda sont toujours génériques par des pointcuts.

Listing 2.10 – Exemple d'une spécification d'un aspect

```
1  generic
2     Log_PC : Pointcut;
3  aspect Logger_Aspect is
4
5     type Logger_A is new Aspect_Ada.Detailed_Aspect with
6        record
7          Log_Count : Natural := 0;
8        end record;
9
10    advice Before (Logger : in out Logger_A);
11    for Before'pointcut use Log_PC;
12 end Logger_Aspect;
```

Listing 2.11 – Exemple d'une implantation d'un aspect

```
1  aspect body Logger_Aspect is
2    advice Before (Logger : in out Logger_A) is
3      begin
4         Logger.Log_count := Logger.Log_count + 1;
5         Put_Line("execution count : " &
6                  Natural'Image(Logger.Log_count) &
7                  "of the joinpoint " &
8                  Image(Get_Join_Point(Logger_A).all));
9    end Before;
10 end Logger_Aspect;
```

Discussion

La division d'un aspect en une spécification et un corps (body) provient de la vision que AspectAda doit être aussi proche que possible de la syntaxe Ada. Néanmoins, cette séparation n'ajoute pas de valeur en termes de visibilité réduite. Les corps des aspects ne peuvent pas être recompilés séparément sans refaire le tissage à tous les joinpoints désignés.

2.2.5 Règles de composition : weaver

AspectAda a spécifié une unité de programme appelée *weaver* afin de définir les pointcuts et les associer avec les aspects. Le *weaver* contient toutes les règles de composition des aspects avec les pointcuts à travers la définition des pointcuts et l'instanciation des aspects. Le listing 2.12 présente un exemple de *weaver* `Text_IO_Composition` contenant deux pointcuts et une instanciation de l'aspect `Logger_Aspect`.

Listing 2.12 – Exemple de weaver

```
1  with Logger_Aspect;
2  weaver Text_IO_Composition_Rules is
3     Create_File_PC : Pointcut := execution Text_IO.Create* (File_Type, ..);
4
5     File_Management_PC : Pointcut := Create_File_PC or
6                                      execution Text_IO.Open (..) or
7                                      execution Text_IO.Close (..) or
8                                      execution Text_IO.Delete (..) or
9                                      execution Text_IO.Reset (..);
10
11    aspect File_Management_Logger is
12      new Logger_Aspect (Log_PC => File_Management_PC);
13 end Text_IO_Composition_Rules;
```

Principe de priorité des aspects

Si deux aspects différents sont instanciés dans le même programme *weaver* alors l'aspect instancié le premier est le plus prioritaire.

La question qui se pose : Que signifie un aspect plus prioritaire qu'un autre ?

Supposons deux aspects instanciés avec le même *pointcut* alors les advices de l'aspect le plus prioritaire s'exécutent avant ceux de l'aspect le moins prioritaire.

2.3 Compilateur/Tisseur prototype AspectAda 1.0

Les créateurs de AspectAda ont mis en œuvre un compilateur/tisseur prototype. Ils ont conçu l'architecture du compilateur en spécifiant les outils utilisés. Le prototype AspectAda effectue la génération et le tissage du code. Dans cette section, nous commençons par l'explication et le critique de l'architecture du compilateur AspectAda ainsi que les outils utilisés. Ensuite, nous citons les problèmes relevés au niveau de l'opération de tissage et de génération de code après une phase de test du compilateur AspectAda.

2.3.1 Ancienne Architecture

L'architecture du compilateur prototype (figure 2.1) pour le langage AspectAda est composée de deux composants : le tisseur (*weaver*) et la *Runtime*.

Le sous-système *weaver* prend trois types d'entrées :

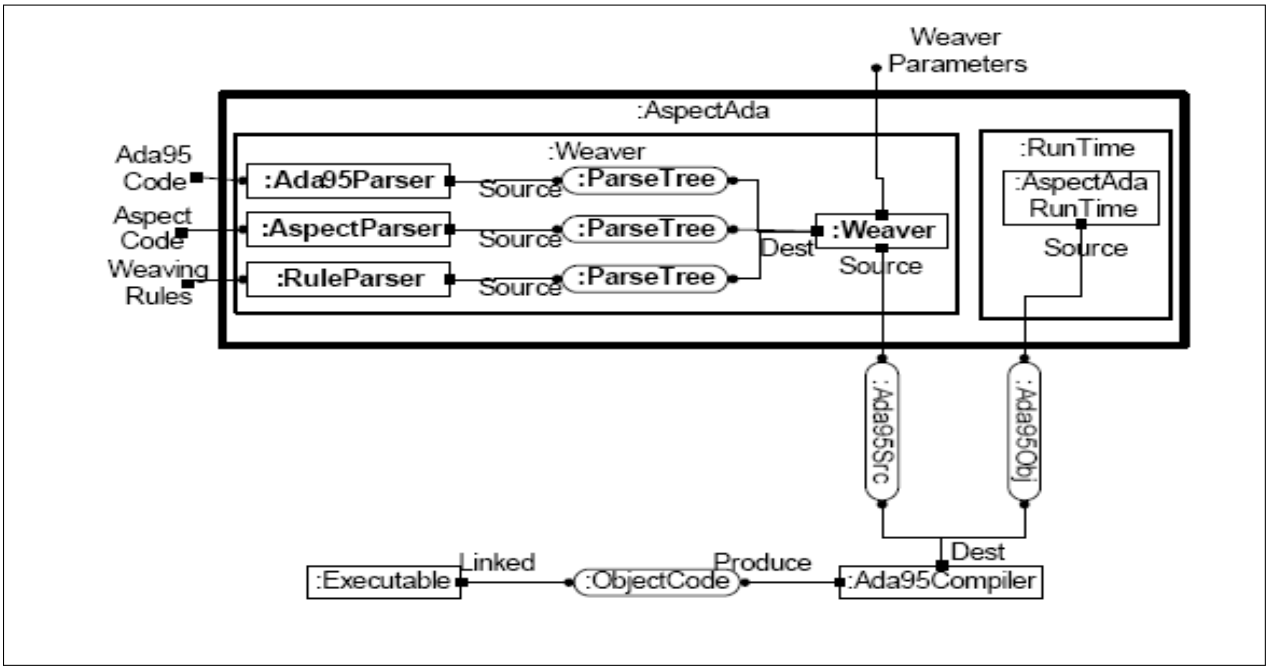


Figure 2.1 – Ancienne architecture du compilateur prototype [PC05]

- le code fonctionnel en Ada,
- le code des aspects : c'est du code Ada95 (grammaire complète) avec des extensions pour définir les advices.
- le code des règles de composition (*weaver code*) : grammaire des pointcuts.

Il produit en sortie un code Ada tissé. La *Runtime* est une bibliothèque de fonctions prédéfinies utilisées dans le code des advices ainsi que dans le code généré après tissage pour accéder à des informations sur le *joinpoint* courant.

Les trois entrées du *weaver* sont analysées syntaxiquement chacune par le parseur spécifique à elle pour donner lieu à trois arbres syntaxiques abstraits (AST) différents. Ces arbres peuvent être traversés et interrogés par le weaver lors des opérations de sélection des

joinpoints (*selection*) et de tissage de code (*merging*). La compilation, l'assemblage (*binding*) et l'edition des liens (*linking*) du code Ada généré sont effectués par le compilateur Ada GNAT (*The GNU Ada Compiler*) en donnant lieu à un exécutable.

Dans le but d'effectuer l'analyse syntaxique (*parsing*) des différentes entrées (chacune des entrées appartient à un langage indépendant), le *weaver* fait appel à deux outils Scaflex/Scayacc/AdaGOOP [Car00, CS02] (*Ada Generator of Object-Oriented Parsers*) et ASIS (*Ada Semantic Interface Specification*) [BSB91].

Scaflex/Scayacc/AdaGOOP

AdaGOOP est un outil qui prend en entrée une grammaire et génère automatiquement un analyseur lexical (*Lexer*), un analyseur syntaxique (*Parser*), et le code pour générer un arbre syntaxique orienté objet. Il génère également le code pour faire la traversée de l'arbre syntaxique en se basant sur le patron de conception visiteur (*visitor*) [GHJV95]. AdaGOOP, est utilisé pour effectuer les analyses syntaxiques du code des aspects et du code des règles de composition (l'unité *weaver*).

ASIS

ASIS est une interface standardisée ISO qui fournit des informations sémantiques et syntaxiques. ASIS fournit la base pour implanter des outils portables visant à analyser les propriétés statiques du code source Ada [RS96]. L'analyse syntaxique (*parsing*) du code fonctionnel Ada95 se fait en utilisant ASIS.

Une fois les AST sont créés, le weaver effectue en collaboration avec ASIS la sélection des joinpoints appropriés qui sont désignés par les pointcuts. Le *weaver* fait le parcours de l'AST Ada par l'intermédiaire de *ASIS iterator*. Au fur et à mesure, le *weaver* interroge chaque noeud de l'arbre (en utilisant également des fonctions fournies par ASIS) pour faire le *matching* des *joinpoints*. Après le parcours de l'AST Ada et la sélection des *joinpoints*, le *weaver* fusionne les advices avec le code Ada pour produire du code source Ada tissé.

Discussion

Suite à l'étude de l'architecture du compilateur/tisseur AspectAda, nous avons constatés un ensemble de lacunes liées aux composants de l'architecture.

1. Limites de la bibliothèque `Runtime`

D'une part, la `Runtime` offerte par AspectAda est très réduite. Elle permet l'accès à peu d'informations sur les *joinpoints* :

- Nom du joinpoint
- Noms, modes et types des arguments.

Elle néglige certaines informations très utiles dans le développement des advices telles que les valeurs des paramètres, le type et la valeur retournée si le *joinpoint* est une fonction, la catégorie du *joinpoint* (*function*, *procedure* ou *entry*), etc.

D'autre part, l'implantation de la `Runtime` renferme des constructions qui se contredisent avec les contraintes temps-réel. La `Runtime` définit un type de tableau de taille non spécifique (Unconstrained Array Type) appelé `Arg_Array` utilisé pour créer des tableaux servant au stockage des arguments des *joinpoints*. De plus, elle utilise les routines du paquetage `Ada.Containers.Vectors` pour la manipulation de ces tableaux. Or, l'utilisation d'un type de tableau de taille non spécifique et du package `Ada.Containers.Vectors` sont interdites par le profil *Ravenscar*.

Pour pallier les manques et les problèmes de la `Runtime`, nous proposons dans la section 3.3 du chapitre 3 :

- une extension de la Runtime par des nouvelles fonctionnalités afin d'exporter plus d'informations sur les *joinpoints*,
- une solution pour rendre la manière de stockage des arguments des *joinpoints* statique (c-à-d nous exigeons l'utilisation d'un type de tableau de taille fixe).

2. Limites de l'outil AdaGoop

Le composant *weaver* du compilateur fait appel à un outil de génération automatique de parseur (Scaflex/Scayacc/AdaGoop) donnant un code généré illisible et pratiquement non exploitable. En outre, le parseur généré ne detecte pas très bien les erreurs et n'affiche pas des messages d'erreurs qui aident à la réparation. De plus, les programmeurs du compilateur

ont modifié le code généré par AdaGoop pour ajouter des fonctionnalités manquantes. Cependant, cette méthode d'implantation est très mauvaise du moment où la regénération du code provoque la perte des modifications qu'ils ont ajoutées.

2.3.2 Tissage et génération de code

Le tisseur prototype AspectAda effectue une simple opération de tissage. Le code de l'advice est inséré directement dans le code du *joinpoint* par l'intermédiaire des blocs *declare* du langage Ada. Pour voir mieux l'opération de tissage, nous présentons l'exemple simple '`Hello_World`' qui accompagne le code source du compilateur.

1. Code fonctionnel Ada : Il s'agit d'un code source Ada (Code métier) qui se compose de :

• Un package '`Do_Hello`' contenant la fonction '`Hello_World`'. Les listings 2.13 et 2.14 présentent respectivement la spécification et l'implantation du package Do_Hello.

Listing 2.13 – Spécification du package Do_Hello

```
1 package Do_Hello is
2    procedure Hello_World(A : Integer);
3 end Do_Hello;
```

Listing 2.14 – Implantation du package Do_Hello

```
1 with Ada.Text_IO;
2 package body Do_Hello is
3
4    procedure Hello_World(A : Integer) is
5    begin
6       Ada.Text_IO.Put_Line("Hello World");
7    end Hello_World;
8 end Do_Hello;
```

• Une procédure 'Hello' qui appelle la fonction Hello_World.

Listing 2.15 – Procédure Hello

```
1 with Ada.Text_IO;
2 with Do_Hello;
3
4 procedure Hello is
5 begin
6    Do_Hello.Hello_World(1);
7 end Hello;
```

2. Code AspectAda : Il contient le code de l'aspect et le code du `weaver` :

• Le code de l'aspect : c'est l'aspect '`Logger_Aspect`' (listings 2.10 et 2.11).

• Le code du *weaver* (les règles de composition) : le listing 2.16 présente le fichier '`hello_rules.aaw`'. Il contient :

- la définition du pointcut '`Hello_World_PC`' permettant d'intercepter l'exécution de la procédure '`Hello_World`' du package '`Do_Hello`'.
- L'instanciation de l'aspect '`Logger_Aspect`' en lui donnant en paramètre le pointcut '`Hello_World_PC`'.

Listing 2.16 – Code du *weaver* Hello_Rules

```
1  with Logger_Aspect;
2  weaver Hello_Rules is
3
4    Hello_World_PC : Pointcut := execution(Do_Hello.Hello_World(..));
5
6    -- Instansiate an aspect
7    aspect My_Simple is new Logger_Aspect(Hello_World_PC);
8
9  end Hello_Rules;
```

3. Code source Ada généré : Nous obtenons dans le code généré un ensemble de fichiers :

1. Le package '`Logger_Aspect`' c'est le package généré à partir de l'aspect '`Logger_Aspect`'. La spécification du package 2.17 contient la déclaration du type `Logger_A` et une variable `Logger` de type de l'aspect. L'implantation du package 2.18 est vide.

Listing 2.17 – Spécification du package `Logger_Aspect` projection de l'aspect

```
1  with Ada.Text_IO;
2  with Aspect_Ada;
3
4  package Logger_Aspect is
5     pragma Elaborate_Body;
6
7     type Logger_A is new Aspect_Ada.Detailed_Aspect with
8        record
9          Log_Count : Natural := 0;
10       end record;
11
12    Logger : Logger_A;
13 end Logger_Aspect;
```

Listing 2.18 – Corps du package `Logger_Aspect`

```
1  package body Logger_Aspect is
2
3  end Logger_Aspect;
```

2. Le corps du package `Do_Hello` tissé et généré : nous remarquons bien que l'opération de tissage est basée sur les blocs declare du langage Ada. Le corps de l'advice *before* est inséré dans un bloc declare avant le corps de la procédure `Hello_World` qui est également inséré dans un bloc declare. Ces deux blocs sont encapsulés à leur tour dans un bloc declare.

Listing 2.19 – Corps du package `Do_Hello` tissé et généré

```
1  -- BEGIN AspectAda
2  with Ada.Text_IO;
3  with Aspect_Ada;
4  with Ada.Text_IO;
5  with Logger_Aspect; use Logger_Aspect;
6  -- END AspectAda
7  with Ada.Text_IO;
8
9  package body Do_Hello is
10
11    procedure Hello_World(A : Integer) is
12    begin
13    -- BEGIN AspectAda
14       declare
15         Args : Aspect_Ada.Arg_Array := (
16           1 => Aspect_Ada.To_Arg("A", "Integer", Aspect_Ada.A_DEFAULT_IN_MODE));
17       begin
18         Aspect_Ada.Set_Join_Point(Logger,
19                     Aspect_Ada.Create_Join_Point("Do_Hello.Hello_World", Args));
20         declare
21         begin
22            Logger.Log_count := Logger.Log_count + 1;
23            Put_Line("execution count : " &
24                  Natural'Image(Logger.Log_count) &
25                  "of the joinpoint " &
26                  Image(Get_Join_Point(Logger_A).all));
27         end;
28    -- END AspectAda
29         begin
30            Ada.Text_IO.Put_Line("Hello World");
31         end;
32
33       end;
34    end Hello_World;
35
36 end Do_Hello;
```

Discussion

Dans le contexte des systèmes temps-réel, l'opération de tissage peut compromettre le déterminisme du système parce qu'elle a un impact directe sur son fonctionnement. De plus, il faut préciser l'emplacement exacte de l'insertion du code de l'advice dans celui du

joinpoint selon le type de l'advice (*before*, *after* ou *around*) et le type du *joinpoint*.

Après une étude approfondie du langage AspectAda, nous avons constaté que l'opération de tissage et de génération de code n'est pas régie par des règles bien précises. La documentation officielle de AspectAda ne contient pas des informations pertinentes sur l'opération de tissage de code. Dans la version publiée de AspectAda [PC05], les auteurs ont juste indiqué que le code des advices est inséré directement dans le code du *joinpoint*. Pour cela, nous avons adopté la méthode de l'ingénierie inverse pour tester le fonctionnement du compilateur prototype et particulièrement pour s'assurer du bon déroulement de l'opération de tissage en élaborant des exemples simples et examinant le code généré. Malheuresement, nous avons relevés plein d'erreurs et de défauts au niveau de l'opération de tissage. Nous reviendrons plus en détails sur ces fonctionnalités manquantes dans la section 2.3.3.

L'ancienne version de AspectAda supporte seulement des *joinpoints* de type execution. D'après le listing 2.19, les advices sont insérés entre '*begin*' et '*end*' du sous-programme intercepté. Pour garantir le bon ordre d'exécution des différents advices et le code du sous-programme intercepté, les codes des advices ainsi que le code du sous-programme sont insérés dans des blocs '*declare*'. Cependant cette solution présente deux inconvénients qui empêchent son utilisation dans le contexte des systèmes temps-réel critiques :

1. L'utilisation des blocs '*declare*' est nuisible au déterminisme et à la sûreté de fonctionnement de l'application. Dans un bloc '*declare*', les données déclarées entre '*declare*' et '*begin*' sont alloués dynamiquement à l'exécution, et sont automatiquement détruites à la fin du bloc '*declare*'.
2. Toute modification faite dans le corps d'un advice de l'aspect engendre la recompilation de tout le code source de l'application.

Une solution alternative consiste à transformer les advices en sous programmes qui seront appelés par le *joinpoint*. Avec une telle approche un changement dans le corps d'un advice conduit uniquement à recompiler le sous-programme présentant l'advice. Cependant, cette solution pourra ajouter des coûts (*overhead*) à l'exécution car un appel de sous-

programme devait être fait pour chaque advice qui doit être exécuté.

Nous consacrons le chapitre 4 de ce rapport pour définir les règles de tissage et de génération de code en respectant les contraintes temps-réel.

2.3.3 Limitations du Compilateur/Tisseur

Au début du stage, nous avons commencé le test du compilateur/tisseur de AspectAda. Nous avons trouvé tellement de fonctionnalités manquantes qu'il n'arrivait pas à compiler et générer du code Ada correct même pour des exemples simples. Le compilateur AspectAda n'arrive pas à analyser correctement le code de l'aspect '`Logger_Aspect`' donné dans le listing 2.10. Pour pouvoir le compiler, nous devons enlever certaines instructions du code de l'aspect : la clause '*generic*', les paramètres formels de l'aspect (de types Pointcut), les clauses '**for**..**use**' des advices déclarés et nous ajoutons la déclaration du paramètre de l'advice comme variable globale de l'aspect (voir listing 2.20). Dans ce cas le code compile mais il génère un code Ada erroné.

Listing 2.20 – Aspect Logger_Aspect après les modifications

```
1  with Ada.Text_IO;
2  with Aspect_Ada;
3
4  --generic
5     --My_Pointcut : Pointcut;
6  aspect Logger_Aspect is
7     type Logger_A is new Aspect_Ada.Detailed_Aspect with
8        record
9           Log_Count : Natural := 0;
10       end record;
11
12    advice Before (Logger : Logger_A);
13    --for Before'Pointcut use My_Pointcut;
14
15    Logger : Logger_A;
16 end Logger_Aspect;
```

Nous avons commencé par la correction et l'ajout des fonctionnalités manquantes du parseur ainsi que du *weaver* du compilateur AspectAda. Cependant, AspectAda est un compilateur très difficilement maintenable et souffre de plusieurs anomalies de fonctionnement. De plus, le code généré par l'outil AdaGOOP n'aide pas à la correction et l'extension. C'est vrai que la génération de code par un tel outil diminue la charge au développeur. Mais, ce

code généré souffre d'une mauvaise lisibilité ce qui empêche sa compréhension et par suite sa maintenance.

Ces caractéristiques se situent aux antipodes des propriétés attendues d'un compilateur pour faciliter son extension. La meilleure solution pour avoir un compilateur flexible et extensible était de construire un nouveau compilateur en évitant de tomber dans les mêmes pièges de conception rencontrés lors du développement de l'ancien compilateur.

Nous proposons la nouvelle architecture du compilateur AspectAda dans la section 3.4 du chapitre 3.

2.4 Conclusion

Dans ce chapitre, nous avons étudié le langage AspectAda existant. Nous avons analysé sa syntaxe en mettant l'accent sur les manques et les défauts de sa grammaire. Ensuite, nous avons examiné les composants de l'architecture de son compilateur/tisseur. Nous avons également cité les anomalies de fonctionnement du compilateur vis à vis l'analyse syntaxique, l'opération de tissage et de génération de code. Suite à l'étude de ce langage, nous avons décidé de

1. Corriger et étendre sa syntaxe et sa bibliothèque '`Runtime`' avec de nouvelles fonctionnalités,
2. Proposer une nouvelle architecture du compilateur/tisseur,
3. Définir les règles de tissage et de transformation de code de manière que le code Ada généré respecte les contraites temps-réel.

Nous proposons dans le chapitre suivant (chapitre 3) nos contributions liées aux extensions et modifications (1) de la syntaxe et la sémantique du langage AspectAda, (2) de la bibliothèque et (3) la nouvelle architecture du compilateur. Les règles de tissage et de génération de code sont définies dans le chapitre 4.

Chapitre

3 Extension du Langage AspectAda

3.1 Introduction

Dans le chapitre précédent, nous avons étudié les différents concepts du langage AspectAda en mettant en relief les manques et les problèmes rencontrés lors de son étude. Comme déjà mentionné, nous nous intéressons dans ce travail de mastère à adapter et étendre le langage AspectAda d'une part pour l'améliorer et d'autre part pour intégrer la POA dans le développement des systèmes temps réel. Dans ce chapitre, nous présentons les solutions proposées pour pallier les problèmes et les manques rencontrés avec l'ancienne version de AspectAda.

3.2 Nouvelle syntaxe et sémantique de AspectAda

Comme c'est mentionné dans le chapitre 2, la syntaxe du langage AspectAda souffre de plusieurs lacunes. Nous présentons dans cette section les modifications et les ajouts pour la grammaire des pointcuts ainsi que pour la grammaire des aspects.

3.2.1 Nouveau modèle des joinpoints

Dans les sections 2.2.1 et 2.2.2 du chapitre 2, nous avons essayé à travers les discussions d'extraire les problèmes liés au modèle des *joinpoints* et aux expressions des pointcuts. Dans le but de les améliorer, nous proposons dans cette section les solutions et les extensions adoptées. Une première amélioration consiste à ajouter la primitive '**call**' en plus

de '**execution**'. En effet, nous étendons les expressions des pointcuts de AspectAda pour supporter les appels des *joinpoints* en plus de leurs exécutions.

Le nouveau modèle des *joinpoints* distingue entre les fonctions et les procédures, contrairement à l'ancien qui les considère comme des méthodes (*method_pattern*) comme AspectJ. De plus, nous étendons le modèle des *joinpoints* de AspectAda pour :

1. Supporter les entrées (*entry*) [1] en plus des sous-programmes.
2. Spécifier les modes des arguments des *joinpoints* en plus de leurs types.
3. Définir le type de retour d'une fonction, s'il s'agit d'intercepter une ou plusieurs fonctions.

De plus, nous modifions les *patterns* des expressions des types des arguments afin d'enlever l'absurdité que nous avons relevé dans le listing 2.4 au niveau du chapitre 2. Il s'agit d'enlever les opérateurs '`and`' et '`xor`' d'une expression d'un type et garder les opérateurs '`or`' et '`not`'. Le listing 3.1 expose les règles de la grammaire après modification.

Listing 3.1 – Règle d'expression des types des arguments après correction

```
1 -- ...
2 type_pattern_expr ::= unary_type_pattern_expr
3    | type_pattern_expr or unary_type_pattern_expr
4
5 unary_type_pattern_expr ::= basic_type_pattern
6    |  not unary_type_pattern_expr
7 -- ...
```

Wildcards

Les composants d'une expression d'un pointcut tels que les identifiants (des noms des *joinpoints*, des noms des paquetages et des types des arguments), les modes des paramètres, etc, sont basés sur les wildcards ('`*`', '`..`', '`+`'). Dans la syntaxe de AspectAda nous précisons les possibilités d'utilisation de chaque type de wildcard. Mais, la sémantique de chaque utilisation reste floue. Le tableau 4.2 montre pour chaque wildcard les possiblités d'utilisation et pour chaque possiblité d'utilisation précise la signification et donne un exemple.

1. Une entrée peut être sur les tâches (*tasks*) ou sur les objets protégés (*protected object*). Les entrées sur les tâches sont utilisées pour définir les rendez-vous. Alors qu'une entrée sur un object protégé est similaire à une procédure sauf qu'elle s'exécute suite à la vérification de la condition de sa barrière.

Table 3.1 – Signification des wildcards dans les expressions des pointcuts

Wilcard	**Utilisation**	**Signification**	**Exemple**
'*'	type du *joinpoint* : function, procedure ou entry	N'importe quel type de *joinpoint*	execution (* com.TypeManager.set *) ;
	les identifiants des noms des *joinpoints*, des paquetages et des types	N'importe quel caractère sauf le '.'	com.*Manager.set_*
	les modes des paramètres	N'importe quel mode	(* Integer)
	type de retour d'une fonction	N'importe quel type de retour	return *
'..'	spécification de l'identifiant d'un paquetage	N'importe quel niveau de sous paquetages	com..*.Put_Line(..)
	profil des paramètres	**f(..)** => n'importe quel profil de paramètres. **f ..** => avec ou sans paramètres.	**f(..)** **f(.. , in Integer)** **f ..**
'+'	spécification des types	N'importe quel sous type	Ada.Containers.Indefinite_Vectors+

3.2.2 Modification des aspects

La deuxième partie de la syntaxe de AspectAda est la syntaxe des aspects : c'est la syntaxe Ada complet avec des extensions pour reconnaître les advices. Nous consacrons cette section pour détailler les modifications faites sur la syntaxe des aspects et des advices. Nous traitons en premier lieu l'omission des types d'aspects. En second lieu, nous corrigeons la syntaxe des advices *around* et nous ajoutons l'instruction *proceed* à la grammaire.

Omission des types d'aspects

Nous avons justifié dans la section 2.2.3 du chapitre 2 à travers la discussion l'inutilité des types d'aspect. Pour cela, nous choisissons d'omettre les types d'aspect dans la définition des aspects et des advices. Notamment, les types `Detailed_Aspect` et `Simple_Aspect` sont enlevés de la Runtime. Dans ce sens, deux questions se posent :

(1) Comment distinguer entre un advice qui veut accéder aux informations du *joinpoint*

courant et celui qui n'y accède pas ?

(2) Comment accéder aux informations du *joinpoint* courant ?

Dans l'ancienne version de AspectAda, tout type d'aspect défini à partir du type `Detailed_Aspect` va hériter l'attribut `Join_Point` ayant le type `Join_Point_Ptr` qui est un pointeur sur un enregistrement de type `Root_Join_Point`. Au lieu de ceci, nous utilisons directement le type `Root_Join_Point`. Par ailleurs, la nouvelle syntaxe des aspects et des advices est de la manière suivante :

– 1er cas : Le comportement d'un advice requiert l'accès aux informations du *joinpoint* courant, alors cet advice prend un paramètre de type `Join_Point` (c'est le type `Root_Join_Point` sauf le nom est modifié). L'advice peut accéder aux informations du *joinpoint* en invoquant les fonctions *getter* et *setter* des attributs du type *Join_Point*. Pour mieux voir la différence, nous présentons dans les listings 3.2 et 3.3 l'exemple `Logger_Aspect` (précédemment vu avec l'ancienne version dans les listings 2.10, 2.11) en appliquant les changements nécessaires :

Listing 3.2 – Exemple de spécification d'un aspect

```
1  with AspectAda_Types;
2  generic
3     Log_PC : Pointcut;
4  aspect Logger_Aspect is
5
6     Log_Count : Natural := 0;
7
8     advice Before (thisJoinPoint : AspectAda_Types.Join_Point);
9     for Before'pointcut use Log_PC;
10 end Logger_Aspect;
```

Listing 3.3 – Exemple d'implantation d'un aspect

```
1  with Aspect_Ada;
2  aspect body Logger_Aspect is
3    advice Before (thisJoinPoint : in out AspectAda_Types.Join_Point) is
4      begin
5         Log_count := Log_count + 1;
6         Put_Line("execution count : " &
7                  Natural'Image(Log_count) &
8                  "of the joinpoint " &
9                  Aspect_Ada.Get_Signature(thisJoinPoint));
10   end Before;
11 end Logger_Aspect;
```

– 2ème cas : un advice n'accède pas aux informations du `joinpoint`, dans ce cas il ne

prend rien en paramètre.

Correction de la syntaxe des advices

Dans l'ancienne version de AspectAda, la syntaxe des advices est similaire à celle des procédures. Néanmoins, un advice *around* pourra remplacer une fonction Ada et dans ce cas il devra retourner une valeur. Pour cela, nous ajoutons à la syntaxe des advices la possibilité de retourner une valeur (listing 3.4). Lors de l'analyse sémantique, le compilateur doit vérifier que seuls les advices *around* pouvant retourner des valeurs. Dans le cas contraire, il doit afficher une erreur.

Listing 3.4 – Extrait de la grammaire présentant la nouvelle syntaxe des advices

```
1 ...
2 advice_spec ::= advice identifier [formal_part]
3         | advice identifier formal_part_opt_and_result
4 formal_part_opt_and_result ::= [formal_part] return name
5 formal_part ::= (param_s)
6 param_s ::= param {, param}
7 ...
```

Nouvelle syntaxe et sémantique de l'instruction proceed

Dans la nouvelle version de AspectAda, la grammaire est étendue par des nouvelles règles pour supporter l'instruction *proceed*. Le listing 3.5 expose un extrait des règles de la grammaire des aspects présentant la syntaxe de *proceed*. Comme un advice *around* peut remplacer une fonction, une procédure ou une entrée alors l'appel de l'instruction *proceed* doit respecter la catégorie du *joinpoint*. Ceci est vérifiée lors de l'analyse sémantique. De plus, une instruction *proceed* ne doit figurer qu'au sein d'un advice *around*. Ceci est également le rôle de l'analyse sémantique.

Listing 3.5 – Extrait de la grammaire présentant l'instruction *proceed*

```
1 ...
2 advice_stmt  ::= statement | proceed_stmt
3 proceed_stmt ::= proceed [value_s] ;
4 value_s ::= value {,value}
5 ...
```

Dans la nouvelle version de AspectAda, les paramètres du *joinpoint* sont disponibles dans le code de l'advice. Pour cela, l'instruction *proceed* pourra être appelée avec des paramètres. Ceci engendre plusieurs cas d'utilisation de l'instruction *proceed* :

- ✦ 1er cas : le développeur veut appeler le *joinpoint* avec les mêmes paramètres passés lors de l'appel → le *proceed* est appelée sans paramètres.
- ✦ 2ème cas : le développeur veut appeler le *joinpoint* par des nouveaux paramètres → dans ce cas le *proceed* est appelée avec les nouveaux paramètres (ceci dans le cas où les paramètres sont en lecture (*in*) ou en lecture/écriture (*in out*)).

  Un sous-cas peut exister : le développeur veut modifier les valeurs de certains paramètres et garder les autres avec les anciennes valeurs. Dans ce cas, il doit récupérer les valeurs des paramètres qu'il ne veut pas les changer (la méthode de récupération des valeurs des paramètres sera communiquée à la section 3.3.1). Puis, il appelle l'instruction proceed avec les nouvelles valeurs des paramètres (pour les paramètres à changer) et les valeurs récupérées (pour les paramètres à garder).
- ✦ 3ème cas : le développeur veut récupérer les valeurs des paramètres après l'appel de *proceed* dans le cas où ces paramètres sont en mode écriture (*out*) ou lecture/écriture (*in out*) → après le *proceed*, le développeur peut utiliser ces paramètres avec leurs noms définis dans la signature du *joinpoint*.

3.3 Extension et Modification de la `Runtime`

Comme c'est expliqué dans la section 2.3.1 du chapitre 2, certaines constructions de la bibliothèque `Runtime` ne respectent pas les contraintes temps-réel :

- La déclaration du type `Arg_Array` : un type de tableau de taille non spécifique utilisé pour la création des tableaux des arguments des *joinpoints*.
- L'utilisation du paquetage `Ada.Containers.Vectors` pour la maniplulation de ces tableaux.

Pour pallier ce problème, nous proposons la solution suivante qui consiste à :

1. Rendre la manière de stockage des arguments des *joinpoints* statique en forçant l'utilisation d'un type de tableau de taille fixe. Pour ce faire, nous avons modifié le type `Arg_Array` pour qu'il soit un type de tableau de taille fixe. La question qui se pose : Quelle est la taille des tableaux de type `Arg_Array` ?

Le type `Arg_Array` est utilisé pour créer des tableaux des arguments de tous les *joinpoints* interceptés dans une application donnée. Par ailleurs, la taille des tableaux de type `Arg_Array` doit être égale au nombre maximal d'arguments d'un *joinpoint* (`Nmax`) que nous pouvons trouver dans les *joinpoints* interceptés. Ainsi la définition du type `Arg_Array` dépend des *joinpoints* interceptés et par suite de l'application. Pour cela, nous avons pensé à isoler les types définis dans la `Runtime` (fait inclus le type `Arg_Array` *Constrained Array Type* de tableau de taille limitée égale à `Nmax`) dans un paquetage qui sera généré. En d'autres termes, pour une application donnée, un paquetage nommé `AspectAda_Types` est généré contenant les déclarations des types nécessaires pour sauvegarder les informations des *joinpoints* ainsi que d'autres types et routines que nous les verrons ci-après. Les types définis dans ce paquetage sont utilisés par les routines de la `Runtime` ainsi que par le code généré après le tissage.

2. Éviter l'utilisation des routines du paquetage `Ada.Containers.Vectors`.

En outre, nous avons mentionné dans la section 2.3.1 du chapitre 2 que la bibliothèque `Runtime` de AspectAda est réduite. Elle ne permet d'exporter aux advices que peu d'informations simples sur les *joinpoints*. Pour cela, nous étendons d'une part la `Runtime` par des nouvelles routines et d'autre part, le paquetage `AspectAda_Types` par de nouveaux types afin d'exporter plus d'informations sur les *joinpoints* :

- Les valeurs des arguments du joinpoint,
- Le type et la valeur retournée si le *joinpoint* est une fonction,
- La catégorie du *joinpoint* (*function*, *procedure* ou *entry*).

3.3.1 Accès aux valeurs des arguments

Dans l'ancienne version de AspectAda, les seules informations connues sur un argument d'un *joinpoint* donné sont : son type, son nom et son mode ; à travers des routines de la `Runtime`. Il n'y a aucun moyen permettant d'accéder (et éventuellement modifier), dans le code d'un advice donné, aux valeurs des arguments du *joinpoint*.

Dans cette section, nous citons dans un premier temps les extensions nécessaires permettant d'accéder aux valeurs des arguments. Dans un second temps, nous définissons la procédure d'accès en mettant en relief le code généré pour la manipulation des types des arguments. Enfin, nous validons la procédure proposée par un exemple simple.

1. Extensions requises pour l'accès aux arguments

Nous avons étendu le type 'Arg' du paquetage `AspectAda_Types` en lui ajoutant un nouveau champ nommé 'Value'. Il permet d'accéder à la valeur d'un argument. La question qui se pose : quel type pouvons nous attribuer au champ 'Value' ?
Les valeurs des arguments peuvent avoir n'importe quel type défini par l'utilisateur ou prédéfini. Pour cela, nous avons défini deux nouveaux types dans le paquetage `AspectAda_Types` : (1) le type '`Value_Holder`' un type parent de tous les types ; (2) le type '`Value_Access`' un pointeur sur le premier type et ses fils : (listing 3.6).

Listing 3.6 – Définition des types `Value_Holder` et `Value_Access`

```
1    type Value_Holder is tagged null record;
2    type Value_Access is access all Value_Holder'Class;
```

Nous avons attribué le type '`Value_Access`' au champ Value. Le type 'Arg' aura alors la structure donnée dans le listing 3.7 :

Listing 3.7 – Extension du type 'Arg'

```
1  type Arg is record
2      Name      : Ada.Strings.Unbounded.Unbounded_String;
3      Type_Name : Ada.Strings.Unbounded.Unbounded_String;
4      Mode_Kind : Mode_Kinds;
5      Value     : Value_Access;
6  end record ;
```

Pour pouvoir récupérer et modifier le nouveau champ 'Value', nous avons ajouté à la `Runtime` deux accesseurs (un 'Getter' et un 'Setter') :

Listing 3.8 – Accesseurs au champ 'Value'

```
1  Function  Get_Value (Argument : in AspectAda_Types.Arg) return Value_Access;
2  Procedure Set_Value
3              (Argument : in out AspectAda_Types.Arg ; V : in AspectAda_Types.Value_Access);
```

- **Getter** : fonction `Get_Value` qui prend en entrée un argument de type 'Arg' et retourne

son champ 'Value'.

- **Setter** : procédure `Set_Value` qui prend deux arguments : le premier en entrée sortie de type 'Arg' et le deuxième en entrée de type '`Value_Access`'. Elle permet d'assigner le deuxième argument au champ 'Value' du premier argument.

2. Procédure de manipulation des arguments

Après avoir étendu la `Runtime`, nous montrons dans cette section comment procéder pour manipuler la valeur d'un argument donné d'un *joinpoint* selon son type.

Nous considérons un argument 'A' donné de type 'T' bien déterminé. Nous voulons manipuler la valeur de 'A' ; cependant il n'y a pas une relation entre le type 'T' et le type '`Value_Holder`' défini dans le paquetage `AspectAda_Types`. D'où vient la nécessité d'avoir une relation entre le type 'T' de l'argument donné et le type '`Value_Holder`' à travers des fonctions de conversion de types. Pour résoudre ce problème, nous avons réflèchi à une solution déterministe :

Pour un argument de type 'T', nous devons générer dans le paquetage `AspectAda_Types` :

– Un type '`T_Holder`' (listing 3.9) héritant du type '`Value_Holder`'. Il possède un champ 'V' de type 'T'.

Listing 3.9 – Type 'T_Holder'

```
1  type T_Holder is new Value_Holder with
2  record
3     V : T;
4  end record;
```

– Une fonction '`Extract_T`' qui permet d'extraire le champ de type 'T' à partir d'un enregistrement de type '`T_Holder`'.

Listing 3.10 – Signature de la fonction Extract_T

```
1  Function Extract_T (H : in T_Holder) return T;
```

– Une autre fonction '`Insert_T`' permettant de spécifier la valeur du champ 'V' d'un enregistrement de type '`T_Holder`'.

Listing 3.11 – Signature de la fonction Insert_T

```
1    Function Insert_T (V : in T) return T_Holder;
```

Lors de l'élaboration de cette solution, nous avons évité tout type de polymorphisme de la technologie orientée objet de Ada pour les raisons cités dans la section 1.2.3 du chapitre 1.

Pour valider cette solution, nous présentons un exemple simple : il s'agit d'un advice *Before* qui accède à la valeur d'un argument d'un *joinpoint* intercepté. Nous supposons que cet advice est associé, à travers les règles de composition du weaver, à un *joinpoint* possédant deux arguments de type `Integer`. Alors pour ce *joinpoint* intercepté, nous devons obtenir dans le paquetage `AspectAda_Types` ce qui est nécessaire pour manipuler le type `Integer` : le listing 3.12 est un extrait de la spécification du paquetage `AspectAda_Types` généré.

Listing 3.12 – Un extrait du paquetage `AspectAda_Types`

```
1  package AspectAda_Types is
2     -- ...
3     -- Integer_Holder is a derived type of Value_Holder
4     -- through which, we can access to values having
5     -- the type Integer.
6
7     type Integer_Holder is new Value_Holder with
8        record
9           V : Integer;
10       end record;
11
12    -- Extract_Integer is a function used to get the value
13    -- of an argument of Integer Type.
14
15    function Extract_Integer (H : Integer_Holder) return Integer;
16
17    -- Insert_Integer is a function used to set the value
18    -- of an argument of Integer Type.
19
20    function Insert_Integer (V : Integer) return Integer_Holder;
21    -- ...
22 end AspectAda_Types;
```

Le listing 3.13 présente un extrait de l'advice *Before* de cet exemple :

Listing 3.13 – Un extrait de l'advice Before

```
1     --...
2     advice Before (thisJoinPoint : AspectAda_Types.Join_Point) is
3       Args            : AspectAda_Types.Arg_Array;
4       First_Arg       : AspectAda_Types.Arg;
5       First_Arg_Value : Integer ;
6     begin
7        --...
```

```
8         Args := Aspect_Ada.Get_Args(thisJoinPoint);
9         First_Arg := Args(1);
10        Ada.Text_IO.Put_Line ("The name of first arg of the JP is " &
11                                                Aspect_Ada.Get_Name(First_Arg));
12
13        First_Arg_Value := AspectAda_Types.Extract_Integer (
14                                    AspectAda_Types.Integer_Holder (
15                                     Aspect_Ada.Get_Value_Access(First_Arg).all));
16
17        Ada.Text_IO.Put_Line ("The value of first arg of the JP is " & First_Arg_Value'Img);
18        --...
19    end Before;
```

3.3.2 Accès à la valeur retournée par un *joinpoint*

Dans AspectJ, un advice *after* peut accéder à la valeur retournée par le *joinpoint* (bien évidemment dans le cas où le *joinpoint* est une fonction). Ceci n'est pas valable dans l'ancienne version de AspectAda. Une telle fonctionnalité est aussi importante que nous devons l'ajouter dans la nouvelle version.

Nous étendons le type '`Join_Point`' de `AspectAda_Types` par trois nouveaux champs '`Return_Type`' et '`Returned_Value`' (listing 3.14).

Listing 3.14 – Extension du type `Join_Point`

```
1  --...
2  type Join_Point is
3     record
4        Args           : Arg_Array;
5        Name           : Ada.Strings.Unbounded.Unbounded_String;
6        Return_Type    : Ada.Strings.Unbounded.Unbounded_String
7                           := Ada.Strings.Unbounded.Null_Unbounded_String;
8        JP_Kind        : JP_Kinds := Procedure_Kind;
9        Returned_Value : Value_Access := null;
10    end record;
```

Le champ '`Returned_Value`' est de type `Value_Access`. Il est initialisé à *null* puisque les *joinpoints* ne sont pas tous des fonctions. L'advice after (listing A.5) de l'exemple A.1 de l'annexe A accède à la valeur retournée par la fonction interceptée. À travers cet exemple, nous pouvons voir comment le développeur peut accéder dans le code d'un advice *after* à la valeur retournée.

3.3.3 Catégorie du *joinpoint*

La spécification de la catégorie du *joinpoint* (fonction, procédure ou entrée) est ajoutée au paquetage des types de AspectAda (`AspectAda_Types`) puisque une telle information est supportée par les nouvelles expressions des pointcuts. Pour ce faire, nous ajoutons le champ '`JP_kind`' au type '`Join_Point`' (voir ligne 8 du listing 3.14). Ce nouveau champ est de type '`JP_Kinds`' (listing 3.15) qui est une énumération de trois valeurs présentant les trois catégories de *joinpoint* actuellement supportées par AspectAda.

Listing 3.15 – Type `JP_Kinds`

```
1 type JP_Kinds is (
2       Procedure_Kind,
3       Function_Kind,
4       Entry_Kind);
```

3.4 Nouvelle Architecture

La construction des compilateurs est un sujet très complexe au cœur de l'informatique. La nouvelle architecture du compilateur AspectAda est très semblable à l'architecture des compilateurs modernes [WM95]. Comme le montre la figure 3.1, la nouvelle architecture renferme trois composants principals : une bibliothèque centrale, une collection de parties frontales et une collection de parties dorsales.

3.4.1 Bibliothèque centrale

La bibliothèque centrale définit un ensemble de routines utiles pour la construction, le parcours et la manipulation des nœuds des arbres syntaxiques (AST). Elle offre de plus d'autres routines pour la manipulation des fichiers et des chaînes de caractères en utilisant les codes de hachage pour améliorer les performances.

Les parties frontales et dorsales ont besoin d'échanger des arbres syntaxiques des différents formalismes (Ada, code des aspects et code du weaver) supportés par le compilateur. En effet, les parties frontales sont censées construire les nœuds des AST et les parties dor-

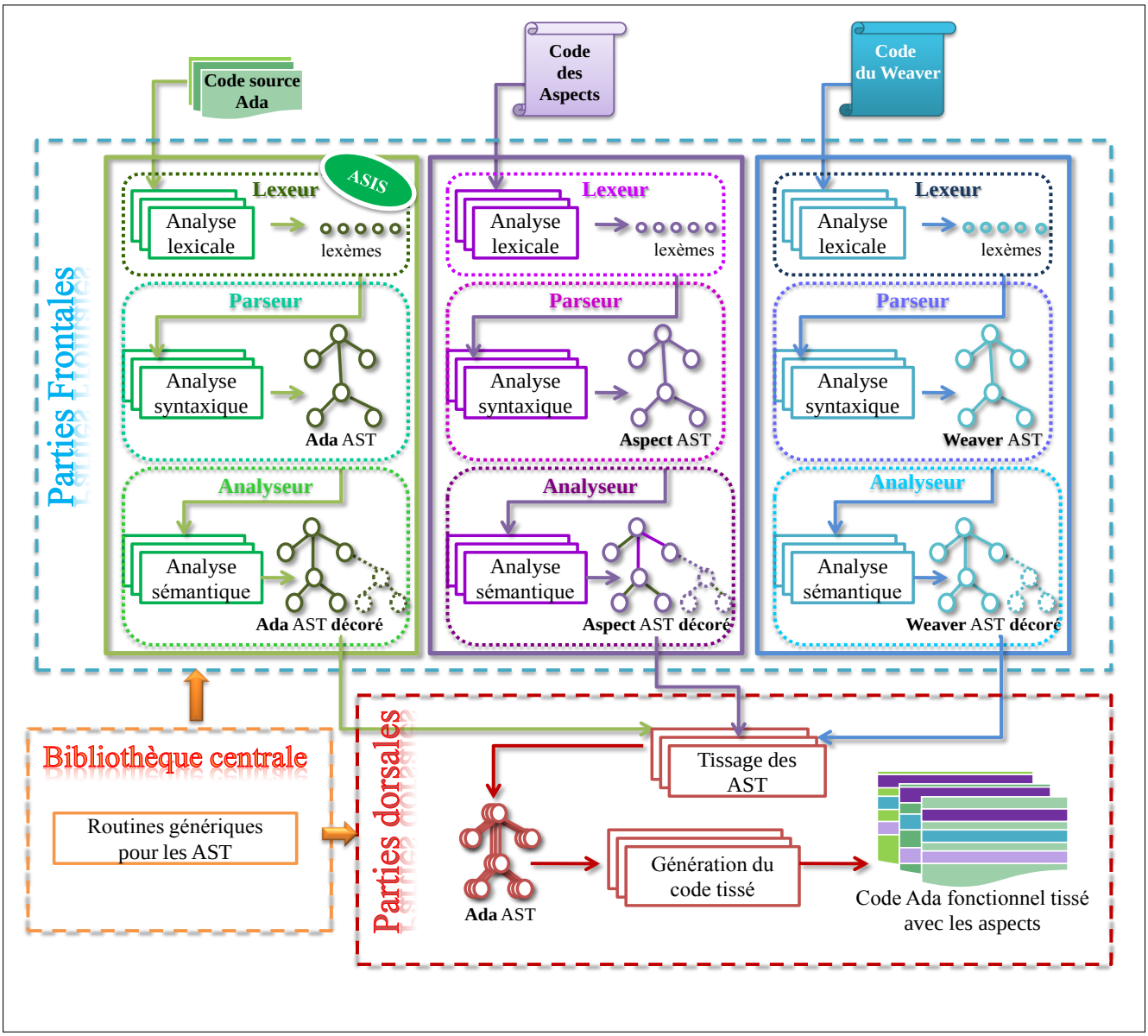


Figure 3.1 – Nouvelle architecture du compilateur AspectAda

sales sont censées parcourir et lire les mêmes nœuds. Il s'avère donc nécessaire que les routines relatives à la manipulation des AST soient situées dans un niveau supérieur à celui des parties frontales et dorsales où elle sont utilisées.

3.4.2 Partie frontale

Similairement à tout compilateur moderne, la partie frontale de AspectAda permet l'analyse lexicale, syntaxique et sémantique de trois types de code source : le code des aspects, le code du weaver et le code source Ada fonctionnel. La sortie principale de chaque partie frontale est un arbre syntaxique. Pour ce faire, les parties frontales font appel aux routines de la bibliothèque centrale. Les sorties secondaires de la partie frontale de AsepctAda consistent en d'éventuels avertissements ou messages d'erreurs. La figure 3.2 donne un aperçu global sur le fonctionnement de la partie frontale de AspectAda.

Figure 3.2 – Fonctionnement global de la partie frontale de AspectAda

Cette étape frontale aide à la vérification de la cohérence syntaxique et sémantique du programme source.

3.4.3 Partie dorsale

La partie dorsale permet principalement le tissage et la génération du code Ada. Elle reçoit les trois arbres syntaxiques résultants de la partie frontale. Puis, elle parcourt et interroge les différents arbres afin de produire un seul arbre Ada. Enfin, à partir de l'arbre Ada il y aura génération du code Ada tissé par les aspects.

3.4.4 Bénéfices de la nouvelle architecture

La conception de la nouvelle architecture de AspectAda d'une façon plus modulaire que l'ancienne offre plusieurs avantages. Elle permet de piloter plus simplement le processus de production tout en ayant la possibilité d'étendre ce processus en ajoutant des parties dorsales ou en enrichissant les parties déjà existantes. De plus, la bibliothque centrale permet la manipulation des arbres syntaxiques à un haut niveau d'abstraction. En outre, la nouvelle architecture n'introduit pas l'outil générateur de parseur (AdaGoop). Ceci permet de développer un parseur flexible, extensible et qui aide à la réparation.

3.5 Conclusion

Dans ce chapitre nous avons présenté les extensions et les corrections attribuées à la syntaxe et la sémantique de AspectAda ainsi que les fonctionnalités ajoutées à travers la `Runtime`. De plus, nous avons présenté l'architecture globale du nouveau compilateur. Dans le chapitre suivant, nous présentons la partie la plus importante de notre approche : les règles de tissage et de transformation du code.

Chapitre

4 Règles de Transformation et de Tissage de Code

4.1 Introduction

Afin de mettre en œuvre un tisseur d'aspects pour les systèmes temps-réel Ada, nous devons mettre en évidence les règles de tissage et de transformation de code AspectAda vers un code Ada. Ces règles devront être établies de manière à obtenir un code Ada tissé obéissant aux contraintes des systèmes temps-réel pour ne pas dégrader leurs fonctionnements. Dans ce chapitre, nous présentons ces règles tout en les illustrant avec des exemples explicatifs.

4.2 Règles de transformation des éléments lexicaux et des types des données

Le langage Ada est étendu avec de nouvelles constructions permettant de supporter l'orientation aspect ce qui donne naissance au langage AspectAda. Par conséquent, la transformation des étéments lexicaux de AspectAda tels que les identifiants (*identifiers*), les littéraux (*literals*), les constantes (*constants*), les opérateurs (*operators*) et la transformation des types des données (*types*) donne lieu à des éléments lexicaux et des types correspondants en Ada.

4.3 Résumé des règles de transformation de code AspectAda

Le tableau 4.1 résume les transformations des constructions du langage AspectAda vers des constructions Ada. Chacune de ces constructions sera détaillée dans les sections qui suivent.

Table 4.1 – Résumé des transformations des constructions AspectAda vers Ada

Construction AspectAda	**Construction Ada**
Aspect générique	*Package* non générique
Advice Before pour les *joinpoints call* ou *execution*	Fonction Ada
Advice After pour les *joinpoints call* ou *execution*	Procédure Ada
Advice Around pour les *joinpoints execution*	Code inséré dans chaque *joinpoint* séléctionné par le *pointcut* correspondant
Advice Around pour les *joinpoints call*	Code inséré dans le contexte de l'appelant
Pointcut et *Aspect instanciation*	Ne seront pas traduit directement, leurs valeurs vont influencer la transformation d'autres éléments.

4.4 Règles de transformation des aspects

Chaque définition d'un aspect consiste à définir sa spécification (fichier ayant l'extension **.aas**) et son corps (fichier ayant l'extension **.aab**). Dans cette section, nous expliquons les règles de transformation des spécifications et des corps des aspects.

4.4.1 Spécification de l'aspect (*Aspect Specification*)

Les aspects dans le langage AspectAda sont toujours génériques avec au moins un pointcut comme un paramètre générique formel. La spécification d'un aspect générique

(listing 4.1) est transformée en une spécification d'un paquetage Ada non générique (listing 4.2) ayant le même nom de l'aspect. La spécification du paquetage généré à partir de l'aspect contient les mêmes déclarations de l'aspect sauf les paramètres génériques (*les pointcuts*) ; les déclarations des *advices* et les déclarations de « *for... use* » clause associée à chaque déclaration d'un *advice*. La spécification du paquetage généré peut contenir d'autres déclarations : des sous-programmes par exemple, en effet, les *advices Before* et *After* sont transformés en sous-programmes (voir la section 4.5).

Listing 4.1 – Spécification d'un aspect

```
1  -- with clauses
2  generic
3      First_Pointcut : Pointcut;
4      -- .....
5  aspect My_Aspect is
6     -- toutes les déclarations
7     -- permises dans un package Ada.
8     -- Déclarations des advices
9     -- les clauses for...use
10    -- ...
11 end My_Aspect;
```

Listing 4.2 – Spécification du package généré

```
1  -- with clauses
2  package My_Aspect is
3    -- ...
4    -- toutes les déclarations projetées
5    -- du contenu de l'aspect.
6
7    -- déclaration des sous-programmes
8    -- présentant les advices
9    -- ...
10
11 end My_Aspect;
```

4.4.2 Corps de l'aspect (*Aspect Body*)

L'implantation des *advices* déclarés dans la spécification de l'aspect se fait dans son corps (listing 4.3). Le corps du paquetage généré à partir de l'aspect (listing 4.4) contient toute implantation définie dans le corps de l'aspect (des sous-programmes,...) sauf les implantations des advices. Nous verrons dans les sections suivantes les règles de transformation et tissage des advices.

Listing 4.3 – Corps d'un aspect

```
1  -- with clauses
2  aspect body My_Aspect is
3     -- Implantation des advices,
4     -- des sous-programmes, ...
5     -- ...
6  end My_Aspect;
```

Listing 4.4 – Corps du package généré

```
1  -- with clauses
2  Package body My_Aspect is
3     -- Implantation
4     -- de sous-programmes,...
5     -- ...
6  end My_Aspect;
```

4.5 Règles de transformation et de tissage des advices et des *joinpoints*

À Chaque *advice* défini dans le code de l'aspect correspond un *pointcut* défini dans le code du *weaver*. Un *pointcut* possède une expression via laquelle il sélectionne un ensemble de *joinpoints*. Ces derniers peuvent être des sous-programmes (des procédures et ou des fonctions) ou des entrées. La nouvelle version de AspectAda supporte deux types de *joinpoints* : ***execution*** *joinpoints* et ***call*** *joinpoints*. La différence entre ces deux types est expliquée dans la section 4.5.1.

Nous différencions deux catégories d'advice :

◇ Les advices dépendants des *joinpoints* : le comportement de certains advices requiert l'accès aux informations du *joinpoint* (son nom, sa catégorie, ses arguments, etc.) à travers des fonctions fournies par la '`Runtime`' de AspectAda. Dans ce cas (comme nous avons vu dans le chapitre précédent), l'advice prend en paramètre le *joinpoint* de type '`Join_Point`' défini dans le paquetage `AspectAda_Types`.

◇ Les advices indépendants des *joinpoints* : d'autres advices ne nécessitent pas l'accès aux informations des *joinpoints*. Dans ce cas, l'advice sera sans paramètre.

La transformation et le tissage des *advices* dans le code métier dépendent des types des *joinpoints* (*execution* ou *call*), de leurs catégories (*procedure*, *function* ou *entry*), du type de l'advice lui-même (*Before*, *After* et *Around*) et de son catégorie (dépendant ou indépendant des *joinpoints*).

Les entrées (*entry*) sont similaires aux procédures. Pour cela, dans la suite, nous spécifions les *joinpoints* de catégorie sous-programme (soit procédure ou fonction). Pour les *joinpoints* de catégorie entrée, ils seront traités de la même manière que les procédures.

Comme c'est mentionné dans la section 2.3.2 du chapitre 2, l'opération de tissage des advices de l'ancienne version de AspectAda utilise les blocs '`declare`' qui peuvent induire le système en non-déterminisme. Nous devons alors trouver une autre manière de tissage adéquate pour les systèmes critiques. Après réflexion et étude d'autres langages d'aspect

comme AspectJ, nous avons élaboré un ensemble de règles de tissage et de génération de code en traitant les combinaisons possibles entre les types des *joinpoints* et les types des advices. L'élaboration de ces règles est basée sur les deux critères suivants :

- Garantir le bon ordre d'exécution des *advices* et du code métier ;
- Obtenir un code tissé optimal, déterministe et avec un *overhead* optimal.

La différence entre les deux types de joinpoints (*call* et *execution*) impose une différence dans leur interception et leurs règles de tissage.

4.5.1 *Joinpoint call* Vs *execution*

Généralement, l'exécution d'un sous-programme se caractérise par deux moments clefs : (1) lorsqu'il est appelé et (2) quand il est effectivement exécuté [Bet09]. Le langage *AspectAda* supporte deux types de *joinpoint* définis dans la POA : *execution joinpoint* et *call joinpoint*.

- *Call joinpoint* : **call** (`joinpoint_pattern`) : ce type identifie tous les appels vers des *joinpoints* ayant des profils conformes à la description donnée par `joinpoint_pattern`.
- *Execution joinpoint* : **execution** (`joinpoint_pattern`) : ce type identifie toutes les exécutions des *joinpoints* ayant des profils conformes à la description donnée par `joinpoint_pattern`.

La différence entre les types *call* et *execution* d'un *joinpoint* est principalement liée au contexte de l'application lors de l'exécution du code advice associé au *joinpoint*. En effet, le code advice associé à un *joinpoint* de type *call* (par exemple l'appel d'une procédure) doit s'exécuter dans le contexte du sous-programme appelant cette procédure. Par contre, le code advice associé à un *joinpoint* de type *execution* (par exemple l'exécution d'une procédure) doit s'exécuter dans le contexte de la procédure appelée. La figure 4.1 montre l'ordre d'exécution des advices associés à l'appel d'un sous-programme et ceux associés à l'exécution du même sous-programme.

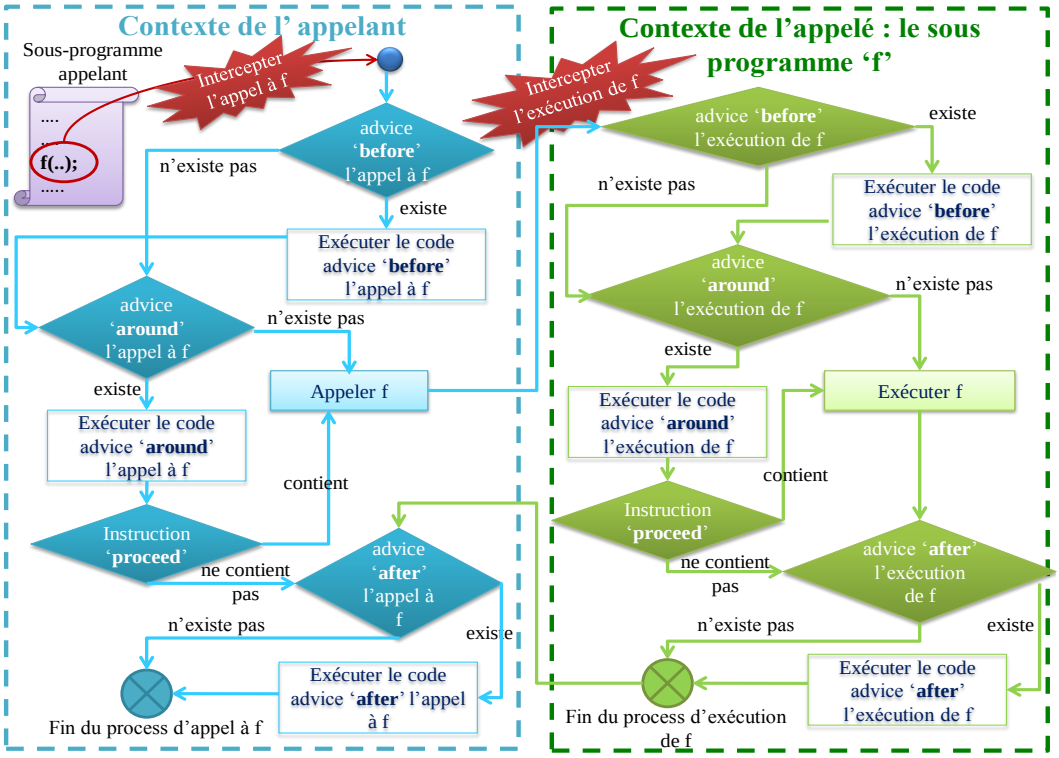


Figure 4.1 – Ordre d'exécution des advices associés à l'appel et ceux associés à l'exécution d'un sous-programme

4.5.2 Règles de transformation et de tissage des advices associés à des joinpoints de type *execution*

Comme expliqué dans la section précédente, le code *advice* associé au joinpoint *execution* doit être tissé dans le contexte de l'appelé. Le tissage des advices se fait alors dans le code du sous-programme intercepté lui-même.

Le tissage des advices diffère selon le type de l'*advice* (*before*, *after* et *around*). Dans la suite, nous traitons les *advices* type par type pour mettre en relief les règles de tissage de chaque type d'*advice* associé à des *joinpoints* de type *execution*. Les différentes règles seront accompagnées par des exemples explicatifs.

1. *Advice Before*

Un code advice *Before* associé à un *joinpoint execution* doit s'exécuter avant le com-

mencement de l'exécution du sous-programme intercepté. Puisqu'il s'agit d'un *joinpoint execution* alors l'advice *Before* est tissé dans le code du sous-programme intercepté (contexte de l'appelé).

Le corps d'un sous-programme Ada est constitué de deux parties : (1) une partie pour la déclaration et éventuellement l'initialisation des données (entre '*is*' et '*begin*') et (2) une partie pour développer le code du sous-programme (entre '*begin*' et '*end*'). Nous considérons le code métier en Ada donné par le listing 4.5. Nous supposons un advice *Before* associé à l'exécution de la fonction F.

Listing 4.5 – Code métier avant tissage

```
-- ...
function F is
   A   : Integer := G(2);
   Ret : Integer;
begin
   Ada.Text_IO.Put_Line ("I am the function F");
   Ret := A * 2;
   return Ret
end F;

function G (B : in Integer) is
begin
   Ada.Text_IO.Put_Line ("I am the function G");
   return B * B;
end G;
```

Une solution intuitive pour le tissage de l'advice *Before* dans la fonction F consiste à insérer le code de cet advice juste aprés le '*Begin*' et avant toutes les instructions de F. Nous obtenons le code suivant : listing 4.6.

Listing 4.6 – Code métier généré après tissage

```
-- ...
function F return Integer is
   A   : Integer := G(2);
   Ret : Integer;
begin
   -- Insertion du code advice Before.
   Ada.Text_IO.Put_Line ("I am the function F");
   Ret := A * 2;
   return Ret
end F;

function G (B : in Integer) return Integer is
begin
   Ada.Text_IO.Put_Line ("I am the function G");
   return B * B;
end G;
--...
```

Nous remarquons bien que la variable A, déclarée dans la fonction F, est initialisée avec la valeur de la fonction G qui effectue un certain comportement. Dans ce cas, la fonction G appelée via la variable A est exécutée avant l'advice *Before*. Par ailleurs, nous obtenons un mauvais ordre d'exécution car le code de l'advice *Before* n'est pas exécuté avant le commencement de l'exécution de la fonction F interceptée. Le code advice *Before* doit ainsi être tissé avant la déclaration des variables du sous-programme intercepté.

La solution proposée consiste à transformer l'advice *Before* en une fonction. Cette dernière est appelée dans la partie déclarative du sous-programme intercepté via une variable qui sera déclarée avant la déclaration de toutes ses variables locales. Cette solution permet non seulement de garantir le bon ordre d'exécution mais aussi de gagner en temps de compilation. Si une modification est faite dans le corps d'un advice de l'aspect alors il suffit de recompiler le code de l'aspect et non pas tout le code source de l'application. À travers cette solution, le traitement d'un *advice Before* se fait en deux étapes : une étape de transformation et une étape de tissage.

Règles de transformation des *advices Before*

D'après la solution citée ci-dessus, un advice *Before* associé à un *joinpoint* de type *execution* sera transformé en une fonction. Vu que cette fonction sera appelée dans la partie déclarative du sous-programme intercepté, alors la transformation d'un *advice before* est indépendante de la catégorie (*function* ou *procedure*) du *joinpoint*. Il sera toujours transformé en une fonction possédant le même nom de l'*advice before*. Nous n'avons pas de contrainte sur le type et la valeur de retour de cette fonction. À une raison de simplification, nous avons choisi qu'elle retourne la valeur '*True*' du type '*Boolean*'. Cette fonction prend le même paramètre de l'*advice* s'il y possède un. Les listings 4.7, 4.8, 4.9 et 4.10 montrent des squelettes de code : il s'agit de la spécification et du corps d'un aspect définissant deux advices before, l'un avec paramètre et l'autre sans paramètre en AspectAda et leurs transformations en code Ada.

Listing 4.7 – Spécification des advices before

```
with AspectAda_Types;
generic
     My_First_Pointcut : Pointcut;
     My_Second_Pointcut : Pointcut;
     --...
aspect My_Aspect is
-- déclaration des variables globales
-- de l'aspect
advice Before1;
for Before1'pointcut use
     My_First_Pointcut;
advice Before2
  (JP : in AspectAda_Types.Join_Point) ;
for Before2'pointcut use
     My_Second_Pointcut;
--...
end My_Aspect ;
```

Listing 4.8 – Projection de la spécification des advices before

```
with AspectAda_Types;
package My_Aspect is

-- déclaration des variables
-- globales de l'aspect
--...

function Before1 return Boolean;

--...

function Before2
 (JP : in AspectAda_Types.Join_Point)
       return Boolean;

--...
end My_Aspect;
```

Listing 4.9 – Implantation des advices before

```
aspect body My_Aspect is

advice Before1 is
 -- declaration des variables locales
 -- de l'advice
begin
 -- le code décrivant le comportement
 -- de l'advice.
end Before1;
advice Before2
   (JP : in AspectAda_Types.Join_Point)
is
 -- declaration des variables locales
 -- de l'advice
begin
-- le code décrivant
-- le comportement de l'advice.
end Before2;

--...

end My_Aspect;
```

Listing 4.10 – Transformation des corps des advices before

```
package body My_Aspect is

function Before1 return Boolean is
-- déclaration des variables locales
-- de l'advice
begin
-- Le code décrivant le comportement
-- de l'advice.
  Return true;
end Before1;
function Before2
  (JP : in AspectAda_Types.Join_Point)
      return Boolean is
  -- déclaration des variables locales
  -- de l'advice
begin
-- Le code décrivant
-- le comportement de l'advice.
 Return true;
end Before2;
--...
end My_Aspect;
```

Règles de tissage des *advices Before*

Après avoir transformé l'*advice before* en une fonction qui retourne un *boolean*, nous devons suivre le reste des étapes et des règles de tissage d'un *advice before* associé à un *joinpoint execution* :

1. Une nouvelle variable est déclarée dans le sous-programme intercepté à travers laquelle nous pouvons appeler la fonction générée projection de l'*advice before*.
2. La nouvelle variable doit être déclarée avant la déclaration de toutes les variables locales du sous-programme intercepté.
3. Le nom de la nouvelle variable doit être différent de tous les noms des variables de même type locales ou globales au sous-programme intercepté .
4. Les priorités des aspects doivent être respectés. Nous supposons le cas où deux aspects différents possédant chacun un *advice before* et ces deux *advices* sont associés à l'exécution du même *joinpoint*. Dans ce cas, la fonction présentant l'*advice Before* de l'aspect le plus prioritaire doit être appelée la première dans la partie déclarative du sous-programme intercepté.
5. Les règles (1,2,3 et 4) sont valables pour les trois catégories de *joinpoint* : fonction, procédure ou entrée.
6. Ce traitement se répète pour tous les *joinpoints* sélectionnés par le même *pointcut*.

Nous présentons dans l'annexe A un exemple (section A.1) illustrant la manière de transformation et de tissage des advices *before* associés à des *joinpoints execution*.

2. *Advice After*

Un *advice after* associé à l'exécution d'un *joinpoint* doit s'exécuter juste après la fin de l'exécution du sous-programme intercepté. De plus, les advices associés à des *joinpoints execution* sont tissés dans le code du sous-programme intercepté (contexte de l'appelé). Afin d'éviter les blocs '*declare*' et gagner en temps de compilation nous avons choisi de transformer un advice after en un sous-programme qui sera appelé dans le sous-programme intercepté. Comme un advice before, le traitement d'un advice after se fait en deux étapes : une étape de transformation et une étape de tissage.

Transformation des *advices After*

L'appel au sous-programme généré présentant l'*advice after* se fait à la fin du sous-programme intercepté. L'emplacement exact de l'appel dépend de la catégorie du sous-

programme intercepté (fonction ou procédure). Nous mentionnons la différence dans l'étape de tissage. Comme l'appel au sous-programme présentant l'*advice after* se fait dans le corps du sous programme intercepté, alors nous n'avons pas de contrainte dans le choix de son type (fonction ou procédure). À une raison de simplification nous avons choisi le type *procedure*. Le nom de la procédure est celui de l'*advice after*. De plus, elle doit avoir le même argument que l'*advice after* s'il y possède un. Les listings 4.11, 4.12, 4.13 et 4.14 montrent des squelettes de code : il s'agit de la spécification et du corps d'un aspect qui définit deux *advices after* en AspectAda, l'un avec paramètre et l'autre sans paramètre, et leurs transformations en code Ada.

Listing 4.11 – Spécification des advices After

```
1  with AspectAda_Types;
2  generic
3        My_First_Pointcut : Pointcut ;
4        My_Second_Pointcut : Pointcut ;
5        --...
6  Aspect My_Aspect is
7     -- déclaration des variables
8     -- globales de l'aspect
9     --...
10   advice After1 ;
11   for After1'pointcut use
         My_First_Pointcut;
12   --...
13   advice After2 (thisJoinPoint :
14      in out AspectAda_Types.Join_Point) ;
15   for After2'pointcut use
         My_Second_Pointcut;
16   --...
17 end My_Aspect ;
```

Listing 4.12 – Projection de la Spécification des advices after

```
1  with AspectAda_Types;
2  package My_Aspect is
3   -- déclaration des variables
4   -- globales de l'aspect
5   --...
6
7  procedure After1;
8
9  procedure After2
10 (thisJoinPoint : in out
11   AspectAda_Types.Join_Point);
12
13
14 --...
15
16
17
18
19 end My_Aspect;
```

Listing 4.13 – Implantation des advices after

```
1  Aspect body My_Aspect is
2
3      advice After1 is
4       -- declaration des variables
5       -- locales de l'advice
6      begin
7       -- le code décrivant
8       -- le comportement de l'advice.
9       --...
10     end After1 ;
11
12     advice After2 (thisJoinPoint :
13       in out AspectAda_Types.Join_Point)
14     is
15      -- declaration des variables
16      -- locales de l'advice
17     begin
18        -- le code décrivant
19        -- le comportement de l'advice.
20        --...
21     end After2 ;
22     --...
23 end My_Aspect;
```

Listing 4.14 – transformation des corps des advices after

```
1  Package body My_Aspect is
2
3   procedure After1 is
4      -- déclaration des variables
5      -- locales de l'advice
6   begin
7     -- le code décrivant
8     -- le comportement de l'advice.
9     --...
10  end After1;
11
12  procedure After2 (thisJoinPoint :
13    in out AspectAda_Types.Join_Point)
14  is
15    -- déclaration des variables
16    -- locales de l'advice
17  begin
18    -- le code décrivant
19    -- le comportement de l'advice.
20    --...
21  end After2 ;
22  --...
23 end My_Aspect;
```

Règles de tissage des *advices After*

Les règles de tissage d'un advice after associé à un *joinpoint* execution ne seront pas les mêmes pour les *joinpoints* de catégorie fonction et les joinpoints de catégorie procédure. Après avoir transformé l'*advice after* en une procédure, nous devons suivre le reste des étapes et des règles de tissage d'un *advice after* associé à chaque catégorie de *joinpoint execution* :

• **Règles de tissage d'un *advice after* associé à l'exécution d'une procédure :**

1. Nous considérons une procédure dont l'exécution est interceptée par un *pointcut* associé à l'*advice after* à traiter. Dans le code généré, l'appel à la procédure projetant l'*advice after* est ajouté après la dernière instruction de la procédure interceptée (juste avant le 'end').

2. Nous supposons le cas où deux aspects différents possédant chacun un *advice after* et ces deux *advices* sont associés à l'exécution de la même procédure. Dans ce cas, nous devons garantir les priorités entre les aspects. C'est-à-dire, la procédure présentant

l'advice after le plus prioritaire est appelée avant celle de l'advice after le moins prioritaire.

3. Si le pointcut considéré intercepte l'exécution de plusieurs procédures, alors les règles citées seront appliquées pour chacune d'elles.

• **Règles de tissage d'un *advice after* associé à l'exécution d'une fonction :**

Nous considérons une fonction dont l'exécution est interceptée par le *pointcut* associé à l'*advice after* à traiter.

1. Une nouvelle variable ayant le même type que celui de la valeur retournée est déclarée dans la partie déclarative de la fonction interceptée. Le nom de cette variable doit être différent de tous les noms des variables, de même type, locales ou globales de la fonction interceptée.
2. L'instruction '*return*' de la fonction intercepté est remplacée par l'assignation de la nouvelle variable avec la valeur retournée.
3. L'appel à la procédure présentant l'*advice after* est inséré à la fin du code de la fonction interceptée.
4. Nous supposons le cas où deux aspects différents possédant chacun un *advice after* et ces deux advices sont associés à l'exécution de la même fonction. Le principe de priorité implique que la procédure associée à l'*advice after* le plus prioritaire doit être appelée avant celle associée à l'advice after le moins prioritaire.
5. Après l'appel à la procédure présentant l'*advice after*, il y aura une instruction *'return'* de la variable déclarée et assignée précédemment (aux étapes 1 et 2).
6. Ce traitement doit se répéter pour toutes les fonctions dont l'exécution est interceptée par le même *pointcut* qui est associé à l'*advice after* à traiter.

Dans le but d'illustrer la manière de transformation et de tissage des advices *after* associés à l'exécution des joinpoints nous présentons dans l'annexe A un exemple d'application : A.1.

3. *Advice Around*

Un advice *around* s'exécute au lieu du *joinpoint* auquel il est associé. Il peut rendre l'exécution au *joinpoint* grâce à l'instruction **proceed**. L'opération de tissage d'un advice *around* est par conséquent plus délicate que celles des advices *before* et *after*.

Discussion autour les solutions pour la transformation et le tissage d'un advice around

Solution 1 Dans un premier temps, nous avons pensé à une solution de tissage des advices *around* similaire à celle des advices *before* et *after*. Nous présentons cette solution comme suit :

- L'advice *around* est transformé en un sous-programme dans le paquetage projection de l'aspect. Le sous-programme transformé de l'advice est de même catégorie que le sous-programme intercepté (fonction ou procédure). S'il s'agit d'une fonction donc elle aura le même type de retour de la fonction interceptée. Si l'advice *around* possède un paramètre de type Join_Point alors, le sous-programme prend le même paramètre.
- Pour pouvoir modifier le sous-programme intercepté, une copie du sous programme intercepté est générée. Le code du sous-programme généré sera remplacé par un appel au sous-programme projection de l'advice *around*.
- S'il existe une instruction **proceed** dans l'advice alors : une copie du sous-programme intercepté est insérée dans le paquetage projection de l'aspect. Puis, chaque instruction *proceed* de l'advice est remplacée après sa transformation par un appel à la copie du sous-programme intercepté. De plus, si le sous-programme intercepté possède des paramètres et l'instruction *proceed* n'a pas spécifié des paramètres donc l'appel remplaçant le *proceed* doit être accompagné par les mêmes paramètres passés au sous-programme intercepté. Par la suite, lors de l'appel du sous-programme présentant l'advice *around*, nous devons lui passer les paramètres du sous-programme intercepté.

Le problème de cette solution apparaît dans le cas où l'advice *around* est associé à l'exécution de plusieurs sous-programmes ayant des profils différents (catégorie du sous-programmes, les paramètres, le type de retour, etc) et cet advice around contient l'instruction *proceed*. Nous traitons un exemple simple pour mieux comprendre le problème rencontré. Le listing 4.15 présente l'advice *Around* qui est associé à l'exécution des deux procédures P et J données dans le listing 4.16.

Listing 4.15 – Code de l'advice around

```
1  --...
2  advice Around (thisJoinPoint : in AspectAda_Types.Join_Point) is
3  begin
4     Ada.Text_IO.Put_Line (Aspect_Ada.Get_Signature(thisJoinPoint));
5     Proceed;
6  end Around;
7  --...
```

Listing 4.16 – Portion du code métier avant tissage

```
1  --...
2  procedure P (I : Integer) is
3  begin
4     Ada.Text_IO.Put_Line (I'Img);
5  end P;
6
7  procedure J (S : string; K : Integer) is
8  begin
9     Ada.Text_IO.Put_Line (S & K'Img);
10 end J;
11 --...
```

Dans ce cas nous avons un problème lors de la transformation de l'advice *Around* puisque l'advice *Around* contient l'instruction `proceed` non spécifiée par des paramètres. Dans ce cas, l'appel remplaçant le `proceed` doit être accompagné par les mêmes paramètres passés au *joinpoint* courant. Par conséquent le sous-programme projection de l'advice *Around* doit prendre en plus de son paramètre, les paramètres du *joinpoint*. Cependant, les deux procédures (*joinpoints*) associés à l'advice *Around* n'ont pas le même profil de paramètres.

Nous pouvons remédier à ce problème en associant à chaque sous-programme intercepté une copie du sous-programme présentant l'advice *Around* compatible avec son profil. Le code généré projection de l'advice *Around* est donné par le listing 4.17.

Listing 4.17 – code généré projection de l'advice Around

```
--...
procedure Around (thisJoinPoint : in AspectAda_Types.Join_Point ; I : Integer) is
begin
   Ada.Text_IO.Put_Line (Aspect_Ada.Get_Signature(thisJoinPoint));
   Proceed_P (I);
end Around;

procedure Proceed_P (I : Integer) is
begin
   Ada.Text_IO.Put_Line (I'Img);
end Proceed_P;

procedure Around (thisJoinPoint : in AspectAda_Types.Join_Point ;
          S : string ; K : Integer) is
begin
   Ada.Text_IO.Put_Line (Aspect_Ada.Get_Signature(thisJoinPoint));
   Proceed_J (S,K);
end Around;

procedure Proceed_J (S : string; K : Integer) is
begin
   Ada.Text_IO.Put_Line (S & K'Img);
end Proceed_J;
--...
```

Comme le montre le listing 4.18, les codes des deux procédures P et J sont remplacés par des appels à la procédure *Around* correspondante.

Listing 4.18 – Portion du code généré après le tissage

```
--...
procedure P (I : Integer) is
   Value_Access_Arg1 : aliased AspectAda_Types.Integer_Holder :=
                               AspectAda_Types.Insert_Integer(I);
   Args : AspectAda_Types.Arg_Array := (
          1 => Aspect_Ada.To_Arg ("I", "Integer",
                                  AspectAda_Types.Default_In_Mode,
                                  Value_Access_Arg1'Unchecked_Access));
   thisJoinPoint : AspectAda_Types.Join_Point :=
           Aspect_Ada.Create_Join_Point("Dummy.P",
                                        Args,
                                        AspectAda_Types.Procedure_Kind );
begin
   Log_Aspect.Around(thisJoinPoint, I);
end P;


procedure J (S : string; K : Integer) is
   Value_Access_Arg1 : aliased AspectAda_Types.String_Holder :=
                               AspectAda_Types.Insert_String(S);
   Value_Access_Arg2 : aliased AspectAda_Types.Integer_Holder :=
                               AspectAda_Types.Insert_Integer(K);
   Args : AspectAda_Types.Arg_Array := (
          1 => Aspect_Ada.To_Arg ("S", "String",
                                  AspectAda_Types.Default_In_Mode,
                                  Value_Access_Arg1'Unchecked_Access),
          2 => Aspect_Ada.To_Arg ("K", "Integer",
                                  AspectAda_Types.Default_In_Mode,
                                  Value_Access_Arg2'Unchecked_Access));
```

```
30    thisJoinPoint : AspectAda_Types.Join_Point :=
31            Aspect_Ada.Create_Join_Point("Dummy.J",
32                                         Args,
33                                         AspectAda_Types.Procedure_Kind );
34 begin
35    Log_Aspect.Around(thisJoinPoint, S, K);
36 end J;
37 --...
```

Nous remarquons bien (listing 4.17) que pour chaque *joinpoint* associé à l'advice *Around* il y aura génération de deux sous-prgrammes. Par ailleurs, cette première solution possède deux défauts :

1. Elle rend complexe l'opération de génération de code ;
2. Elle augmente énormément la taille du code généré.

Ainsi, la première solution est à rejeter.

Solution 2 Nous considérons un advice *around* associé à l'exécution d'un ensemble de sous-programmes. La deuxième solution consiste à :

- Générer une copie du sous programme intercepté pour pouvoir le modifier.
- Remplacer le code du sous-programme généré par le code de l'advice *around*.
- Dupliquer l'implantation de chaque sous-programme intercepté, en modifiant son nom dans le cas où l'advice *around* contient l'instruction *proceed*. Cette duplication sert à remplacer l'instruction *proceed* par un appel au sous-programme dupliqué.

Avec cette deuxième solution, nous évitons, d'une part, la transformation de l'advice `around` en sous-programme et l'utilisation des blocs '*declare*'. D'autre part, nous simplifions l'opération de tissage et de génération de code pour les advices *around* en garantissant un code tissé optimal et déterministe. Donc, cette solution est à conserver.

Dans la suite, nous développons cette solution afin d'explorer les règles régissant l'opération de tissage des advices *around* associés à des *joinpoints* de type *execution*.
Pour ce faire, nous considérons un sous-programme dont l'exécution est intercepté par un *pointcut* donné. Nous divisons les règles de tissage d'un advice *around* associé à des *joinpoints* de type *execution*, en deux classes selon l'utilisation de l'instruction *proceed*.

La première famille de règles concerne le tissage des *advices around* ne contenant aucune instruction *proceed*. La deuxième famille de règles concerne le tissage des *advices around* dont le code contenant une ou plusieurs instructions *proceed*.

◇ **Règles de tissage d'un *advice around* ne contenant aucune instruction *proceed***

Pour cette première famille de règles, le *pointcut* qu'on a considéré est associé à un *advice around* ne contenant aucune instruction *proceed* :

1. Dans la copie générée du sous-programme intercepté, la partie déclarative est remplacée par la partie déclarative de l'*advice around*.

2. Le corps de la copie générée du sous-programme intercepté est également remplacé par le corps de l'advice *around*. Pour cette règle, nous distinguons deux sous cas :

- □ Le *joinpoint* dont l'exécution est interceptée soit une procédure ou une entrée : dans ce cas le code de l'*advice around* est inséré entièrement dans la copie générée de la procédure interceptée sans aucune modification. Si un *advice after* est associé à l'exécution de la même procédure, alors la procédure générée à partir de l'advice after sera appelée juste après le code inséré de l'*advice around*.
- □ Le sous-programme dont l'exécution est interceptée soit une fonction : dans ce cas le code de l'*advice around* est inséré dans la copie générée de la fonction interceptée avec quelques modifications. D'abord, une nouvelle variable, ayant le même type de retour de l'*advice around*, est déclarée dans la partie déclarative de la fonction générée où nous allons effectuer le tissage. Le nom de la nouvelle variable doit être différent de tous les noms des variables de même types locales ou globales. Ensuite, l'instruction *return* de l'*advice around* est remplacée par une assignation de sa valeur retournée à la nouvelle variable. Puis, si un *advice after* est associé à l'exécution de la même fonction, alors la procédure générée à partir de l'*advice after* sera appelée juste après le code inséré de l'*advice around*. Enfin, une instruction '*return*' de la variable assignée par la valeur retournée est insérée à la fin du corps de la fonction générée.

3. Si le pointcut considéré intercepte plusieurs sous-programmes, alors les règles citées

seront appliquées pour chacun d'eux.

Dans l'annexe A, nous présentons à la section A.2.1 un exemple qui introduit un advice around associé à un joinpoint *execution* et ne contenant pas des instructions *proceed*.

◇ Règles de tissage d'un *advice around* contenant des instructions *proceed*

Le comportement d'un advice *around* associé à l'exécution d'un sous-programme va éventuellement remplacer l'exécution de ce sous-programme. L'action de tissage consiste alors à remplacer le corps du sous-programme intercepté par le corps de l'advice around. Toutefois, une instruction *proceed* dans un *advice around* traduit l'exécution du sous-programme intercepté. Par conséquent, le comportement original du sous-programme intercepté doit être sauvegardé.

Nous considérons un *pointcut* associé à un *advice around* contenant une ou plusieurs instruction *proceed*. Le tissage de cet advice *around* est de la manière suivante :

1. L'implantation de chaque sous-programme intercepté est générée pour pouvoir effectuer l'opération de tissage. Le nom du sous-programme généré est remplacé par un nom déterminé au moment de la génération de code. Le nouveau nom doit être différent des noms des autres sous-programmes appertenant au même paquetage. Nous appliquons des règles supplémentaires lors de l'implantation pour garantir que les conflits de noms ne se produisent pas.

2. Un nouveau sous-programme ayant la même signature du sous-programme intercepté est créé dans le même paquetage généré qui contient le sous-programme intercepté. L'implantation de ce nouveau sous-programme est celle de l'*advice around*.

3. Dans le nouveau sous-programme (mentionné dans l'étape 2), chaque instruction *proceed* est remplacée par un appel au sous-programme généré à partir du sous-programme intercepté et dont le nom est modifié (étape 1).

4. L'instruction *proceed* peut être utilisée avec ou sans paramètres : voir la dernière partie de la section 3.2.2 du chapitre 3.

5. Si le pointcut considéré intercepte plusieurs sous-programmes, alors les règles citées

seront appliquées pour chacun d'eux.

Pour comprendre mieux l'ensemble de ces règles, nous présentons dans l'annexe A à la section A.2.2 un exemple définissant un advice around associé à un *joinpoint execution* et contenant une ou plusieurs instructions *proceed*.

4.5.3 Règles de transformation et de tissage des advices associés à des *joinpoints* de type *call*

Comme mentionné dans la section 4.5.1, le code *advice* associé au *joinpoint* de type *call* doit être tissé dans le contexte de l'appelant. De plus, le code advice *before* doit s'exécuter juste avant l'appel effectif du sous-programme appelé. C'est-à-dire si le sous-programme appelé possède des arguments, alors l'advice before s'exécutera juste après la préparation des arguments et avant le commencement de l'exécution du sous-programme appelé.

Les règles de transformation et de tissage des advices associés à des *joinpoints* de type *call* sont inspirées des règles que nous avons élaboré pour les advices associés à des *joinpoints* de type *execution* en respectant la contrainte de tissage dans le contexte de l'appelant.

Afin de satisfaire la contraite de tissage dans le contexte de l'appelant, nous avons établi un ensemble de règles communes pour tous types d'advices (*before*, *after* et *around*) associés à des *joinpoints* de type *call* :

1. Pour chaque sous-programme f(..) appartenant à un paquetage p intercepté par un *pointcut* de type *call*, il y a génération d'un nouveau sous-programme dans un nouveau paquetage supplémentaire. Le nouveau sous-programme possède la même signature que celle du sous-programme intercepté. Le nouveau paquetage est placé dans le répertoire contenant le code généré. Son nom est de la forme : `Prefix_PackageName` avec :
   - `Prefix` est un préfixe déterminé à la génération de code de façon à garantir qu'il n'y a pas de conflit entre le nom du paquetage généré et les noms des paquetages de l'application.

• `PackageName` est le nom du paquetage où le sous-programme intercepté est défini.

Cependant il existe des cas où le sous-programme et ou le paquetage sont déjà créés :

(a) Si un même sous-programme est intercepté plus qu'une fois par des *pointcuts* de type *call* → alors la génération du nouveau sous-programme se fait une seule fois (lors de la première interception du sous-programme par un *pointcut* de type *call*).

(b) S'il y a plusieurs sous-programmes appartenant à un même paquetage et qui sont interceptés par des *pointcuts* de types *call* → alors la création du paquetage supplémentaire se fait une seule fois (lors de la création du premier sous-programme).

2. Tous fichiers du code de l'application (code métier) qui contient des appels au sous-programme intercepté f(..) seront recopiés dans le répertoire contenant le code généré. Tous les appels interceptés dans les fichiers recopiés et générés sont remplacés par des appels au nouveau sous-programme généré (décrit dans l'étape précédente 1).

3. Au niveau du sous-programme supplémentaire généré, décrit dans l'étape 1, se fait :
   – L'appel au sous-programme f(..) dont l'appel est intercepté (avec les mêmes arguments passés au sous-programme supplémentaire, s'il y a des arguments) ;
   – Le tissage des *advices* associés au *joinpoint call*.

4. Ce traitement se répète pour chaque sous-programme intercepté par un pointcut de type *call*.

Nous pouvons déduire dans le tableau 4.2 les différences entre les *joinpoints call* et les *joinpoints execution* en termes de transformations et de tissage d'advices.

La manière de tisser les advices associés à l'appel de sous-programmes est similaire à celle des advices associés à l'exécution de sous-programmes. Nous devons juste tenir compte des différences décrites dans le tableau 4.2. Cependant, il y a une petite différence au niveau de la transformation de l'instruction *proceed* lors du tissage d'un *advice around*. Puisque le tissage d'un *advice around* se fait dans le sous-programme généré supplémentaire, et le sous-programme dont les appels sont interceptés reste intact alors

Table 4.2 – Différences de tissage d'advices entre les *joinpoints call* et *execution*

Joinpoint execution	***Joinpoint call***
Pas de génération de paquetages et de sous-programmes supplémentaires.	Génération d'un sous-programme supplémentaire pour chaque sous-programme dont un ou plusieurs appels sont interceptés.
Tissage des advices dans le code du sous-programme intercepté lui-même.	Tissage des advices dans le sous-programme supplémentaire généré et associé au sous-programme dont un ou plusieurs appels sont interceptés.
– Code de l'appelant intact. – Transformations faites dans le code de l'appelé.	– Code de l'appelé intact. – Dans le code de l'appelant, remplacement des appels interceptés d'un sous-programme (ayant un profil conforme à l'expression du pointcut) par des appels au sous-programme supplémentaire généré.

nous pouvons remplacer l'instruction *proceed* par un appel à ce dernier. Pour mieux comprendre la différence, nous pouvons voir l'exemple A.5 illustrant le tissage d'un advice *around* associé à un *joinpoint call* contenant une instruction *proceed* et l'exemple A.2.2 traitant la même chose mais dans le cas de joinpoint *execution*.

Dans l'annexe A, nous illustrons avec des exemples simples la manière de tissage de chaque type d'advice associé à des *joinpoints call* : Les exemples A.3, A.4, A.5 présentent respectivement les advices *before*, les advices *after* et les advices *around*.

4.6 Conclusion

Dans ce chapitre, nous avons élaboré les règles de tissage et de génération de code Ada à partir du code AspectAda. Les règles définies sont testées et validées par des exemples simples présentés dans l'annexe A. Elles permettent d'obtenir un code Ada tissé respectant

les contraintes temps-réel. Le développeur peut introduire dans le code des advices des contructions déteruisant le déterminisme du système. Pour pallier ce problème, nous pouvons forcer le code Ada généré à respecter le profil Ravenscar ainsi que d'autres restrictions pour les systèmes critiques. Ces restrictions sont définies par le standard Ada à travers les *pragmas*[1]. Nous pouvons activer ces pragmas optionnellement à la génération du code. À l'exécution quand une assertion n'est pas satisfaite une exception système bien identifiée est levée.

Dans le chapitre suivant, nous présentons le processus d'implantation du nouveau Compilateur/Tisseur AspectAda. Puis, nous passons à la validation de nos contributions à travers une étude de cas.

1. Ada définit un ensemble de pragmas qui peuvent être utilisés pour fournir des informations supplémentaires (directives) au compilateur.

Chapitre

5 Mise en Œuvre et Validation de l'Approche

5.1 Introduction

Dans les deux chapitres précédents, nous avons décrit nos contributions. Dans ce chapitre, nous présentons la mise en œuvre du nouveau Compilateur/Tisseur AspectAda afin de pouvoir utiliser le langage AspectAda dans sa nouvelle version. Dans un premier temps, nous présentons le processus d'implantation du nouveau Compilateur/Tisseur AspectAda. Dans un second temps, nous nous intéressons à valider nos contributions à travers une étude de cas.

5.2 Processus d'implantation du Compilateur/Tisseur de AspectAda

La construction des compilateurs est une tâche très complexe au cœur de l'informatique. La nouvelle architecure du compilateur AspectAda est décrite dans la section 3.4 du chapitre 3. Certaines parties du nouveau compilateur, comme le mécanisme de construction des arbres syntaxiques (*Abstract Syntax Tree* : AST) ont été inspirées des sources du compilateur Ada GNAT. Dans cette section, nous détaillons le processus d'implantation des différents parties (parties frontales et parties dorsales) du compilateur AspectAda.

5.2.1 Implantation des parties frontales (Frontends)

La partie frontale de notre compilateur est décomposée en trois parties frontales indépendantes. Ceci est dû au fait qu'elle reçoit en entrée trois fichiers sources de différentes grammaires : le code des aspects, le code du tisseur (l'unité `weaver`) et le code fonctionnel Ada. Chacune des parties frontales fournit en sotie un arbre (AST) représentant la structure du fichier reçu en entrée. Outre les ASTs, il y aura éventuellement d'autres sorties secondaires telles que les avertissements et les messages d'erreurs.

Chaque partie frontale de AspectAda est scindée en trois composants : l'analyseur lexical (*Lexer*), l'analyseur syntaxique (*Parser*) et l'analyseur sémantique (*Analyzer*). Ces trois composants fonctionnent en collaboration. Dès lors, leurs fonctionnement n'est pas séquentiel, du moins pour l'analyseur lexical et l'analyseur syntaxique. La figure 5.1 montre la circulation du flux de données entre les trois composants de chaque partie frontale.

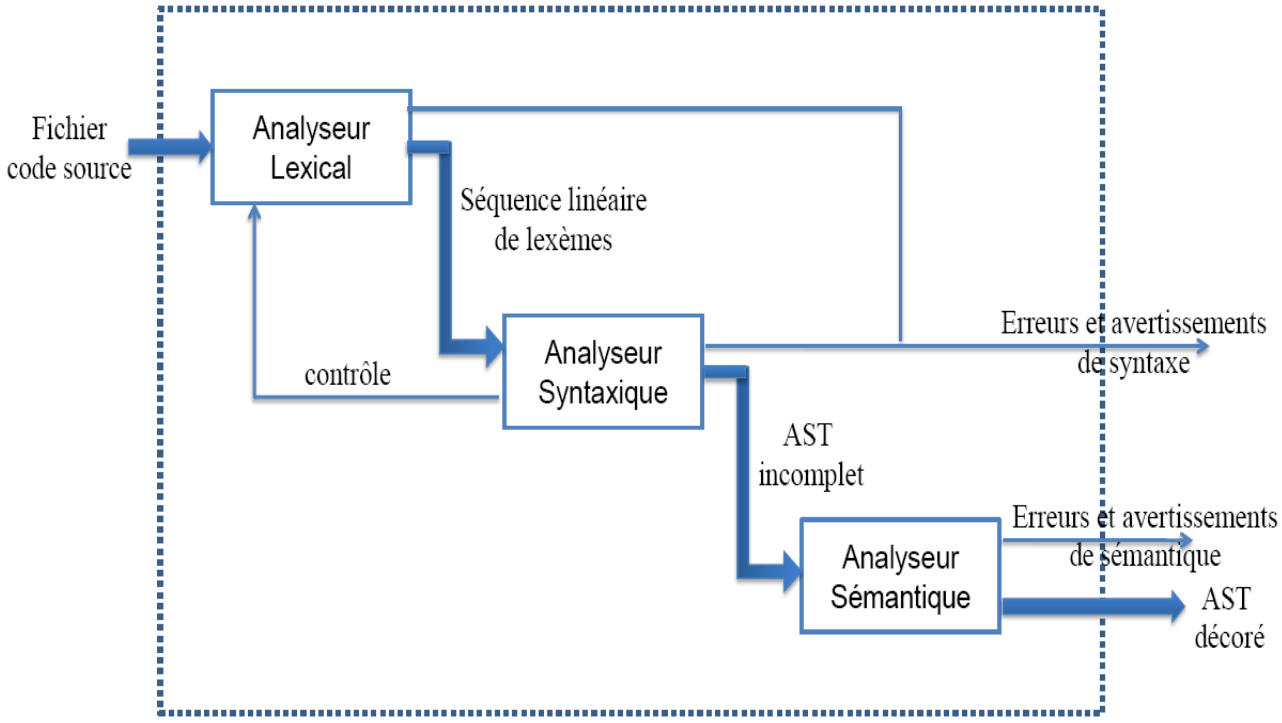


Figure 5.1 – Circulation des flux de données entre les composants d'une partie frontale

1. Analyse lexicale

Le rôle de l'analyseur lexical est de parcourir le fichier source reçu comme entrée, caractère par caractère en le transformant en une séquence d'éléments lexicaux appelés lexèmes (Eng. *Tokens*) tels que les identifiants, les opérateurs, les mots clefs, les séparateurs, etc.

Au fur et à mesure du parcours, l'analyseur lexical essaie de reconnaître les différents lexèmes rencontrés. Chaque fois qu'il traite un lexème, il met à jour un ensemble de variables indiquant le type du lexème, son identifiant et sa valeur s'il s'agit d'une valeur littérale.

Le nouveau compilateur AspectAda offre la possibilité d'enregistrer à un moment donné l'état de l'analyseur lexical et revenir ultérieurement vers cet état. Une telle propriété facilite énormément l'analyse lexicale du fichier source.

2. Analyse syntaxique

L'analyseur syntaxique reçoit en entrée la séquence de lexèmes résultante de l'analyse lexicale. Puis, il essaie de construire à partir de cette séquence, une structure arborescente formée de plusieurs nœuds appelée arbre syntaxique abstrait (AST). En outre, un analyseur syntaxique permet également de vérifier la syntaxe et détecter les erreurs syntaxiques en collaboration avec l'analyseur lexical.

Pour mieux voir le fonctionnement de l'analyseur syntaxique, nous prenons par exemple l'analyseur syntaxique relatif à la grammaire du tisseur (grammaire de l'unité `weaver`). Globalement, pour chaque règle de la grammaire du *weaver*, il existe une fonction `Parse_XXXX` qui appelle les routines de l'analyseur lexical du *weaver* et construit une partie de l'AST. La racine des sources *weaver* étant l'entité (`Weaver_Unit`), le point d'entrée vers son analyseur syntaxique est donc la fonction `Parse_Weaver_Unit` qui, selon la nature des lexèmes envoyés par l'analyseur lexical, appelle les fonctions `Parse_XXXX` correspondantes. La fonction `Parse_Weaver_Unit` retourne la racine de l'AST qui est un nœud de type `K_Weaver_Unit`.

Mécanismes mis en œuvre pour la construction des différents ASTs

Comme c'est mentionné, la partie frontale du compilateur est censée construire trois

ASTs différents pour chaque type de code source (Ada, Aspect et Weaver).

- Pour la construction de l'AST du code fonctionnel Ada, nous utilisons le compilateur Ada GNAT. Ce dernier parse le code source Ada et génère l'AST Ada correspondant qui sera utilisé dans la partie dorsale pour effectuer le tissage.
- Pour la construction de l'AST du code tisseur (l'unité `weaver`), nous implantons des mécanismes inspirés des sources du compilateur GNAT.
- Pour la construction de l'AST du code Aspect, nous adoptons les même mécanismes utilisés pour la construction de l'AST du code `weaver`.

Dans la suite, nous détaillons les mécanismes implantés pour la construction de l'AST du code `weaver`.

• **Description générale**

La structure du weaver est décrite dans un fichier pseudo-IDL où chaque interface représente une entité du weaver (unité weaver, clause with, pointcut, instanciation de l'aspect, etc). Les champs déclarés dans chaque interface correspondent aux entités qu'on peut trouver dans cette entité et aux propriétés de cette entité. Généralement, chaque interface dans le fichier pseudo-IDL correspond à une règle de la grammaire du *weaver*. Par exemple, les règles de la grammaire définissant la déclaration d'unité *weaver* sont données dans le listing 5.1.

Listing 5.1 – Règles de la grammaire définissant la déclaration de l'unité *weaver*

```
1 weaver_unit ::= {with_clause} weaver_spec
2
3 with_clause ::= with compound_name ;
4
5 weaver_spec ::= weaver ada_identifier is {decl_item} end ada_identifier;
```

Dans le fichier en pseudo-IDL, nous définissons une description très similaire pour la déclaration d'unité *weaver* (listing 5.2).

Listing 5.2 – Description en peudo-IDL de la déclaration de l'unité weaver

```
1 // weaver_unit ::= {with_clause} weaver_spec
2 interface Weaver_Unit : Node_Id {
3   List_Id    With_Clauses;   // Of Kind With_Clause
4   Node_Id    Weaver_Spec;    // Of Kind Weaver_Spec
5 };
6
7 // with_clause ::= with compound_name ;
```

```
8  interface With_Clause : Node_Id {
9    Node_Id  Withed_Entity; // Of Kind Designator
10 };
11
12 // weaver_spec ::= weaver ada_identifier is {decl_item} end ada_identifier;
13 interface Weaver_Spec : Definition {
14   List_Id   Declarations;
15 };
```

Dans le même fichier, nous trouvons l'interface `Node_Id` qui est le type parent de tous les autres types de noeuds de l'AST *weaver*. De plus, l'interface `Definition` qui est le parent de `Weaver_Spec` est définie dans le même fichier pour décrire les noms des entités du weaver.

• **Entités générées**

La grammaire est traduite en pseudo-IDL en donnant lieu au fichier `aspectada-weaver-node.idl`. Ensuite, un programme spécial nommé *mknodes*, pris des sources de la suite d'outils "Ocarina" [VZH06], est utilisé pour génèrer à partir du fichier `aspectada-weaver-node.idl` un paquetage Ada (`AspectAda.Weaver.Nodes`) permettant la construction et la manipulation des arbres syntaxiques du *weaver*. Pour chaque champ déclaré dans une interface du fichier pseudo-IDL, le paquetage `AspectAda.Weaver.Nodes` offre deux accesseurs pour pouvoir lire et modifier la valeur de ce champ. Par exemple, pour le champ `Withed_Entity` indiquant l'entité de la clause *with*, deux sous-programmes sont générés (*getter* et *setter*) :

Listing 5.3 – Accesseurs au champ *Withed_Entity*

```
1 function Withed_Entity (N : Node_Id) return Node_Id;
2 procedure Set_Withed_Entity (N : Node_Id; V : Node_Id);
```

En outre, chaque sous-programme du paquetage Ada généré par *mknodes* est fortement dynamiquement typé à travers un ensemble d'assertions (listing 5.4 : lignes 3 et 4). Ceci aide énormément à la détection des erreurs de syntaxe.

Listing 5.4 – Implantation de la fonction *Withed_Entity*

```
1   function Withed_Entity (N : Node_Id) return Node_Id is
2   begin
3      pragma Assert (False
4        or else Table (Types.Node_Id (N)).Kind = K_With_Clause);
5
6      return Node_Id (Table (Types.Node_Id (N)).L (1));
7   end Withed_Entity;
```

Le paquetage `AspectAda.Weaver.Nodes` contient de plus des sous-programmes servant à écrire chacun des nœuds si le développeur désire regarder l'AST généré à partir du code *weaver*.

De la même façon se fait la construction des ASTs du code des Aspects :

1. Traduction de la grammaire des Aspects en pseudo-IDL ;
2. Génération, par l'intermédiare du programme *mknodes*, le paquetage Ada offrant les routines nécessaires pour la construction et la manipulation des ASTs du code des Aspects.

3. Analyse sémantique

L'analyseur sémantique décore l'arbre syntaxique en ajoutant un ensemble d'opérations permettant de naviguer plus facilement dans l'AST. Certes, l'AST produit par l'analyseur syntaxique vérifie la syntaxe du langage mais la sémantique n'est pas nécessairement vérifiée. Par ailleurs, la vérification de la sémantique de l'AST est assurée par l'analyseur sémantique. Par exemple, Il se peut que certaines constantes déclarées ayant des valeurs qui dépassent l'intervalle de leur type. Ceci est détecté au niveau de cette analyse.

À l'issue des analyses effectuées par les parties frontales, on obtient trois arbres syntaxiques représentant les structures des fichiers sources. Ces arbres constituent les entrées de la partie dorsale.

5.2.2 Implantation des parties dorsales (Backends)

La partie dorsale prend comme entrées les ASTs décorés résultants de la partie frontale. Cette partie vise à générer automatiquement le code Ada tissé par les aspects. D'abord, elle parcourt les trois ASTs en interrogeant leurs nœuds. Puis, elle effectue les traitements nécessaires pour faire le tissage des ASTs et produire un seul AST Ada étendu par les aspects. Enfin l'arbre tissé est parcouru pour générer le code Ada tissé.

Nous avons accompli toute l'étude théorique nécessaire pour implanter le tisseur AspectAda en définissant les règles de tissage et de génération de code. Actuellement, le tissage et la génération se fait manuellement.

5.3 Étude de cas : Gestion de charge de travail

Pour valider nos contributions proposées dans les chapitres 3 et 4, nous choisissons le système de gestion de charge de travail (`Workload Manager`) comme étude de cas. Cet exemple est issu du document présentant les recommandations pour le profil Ravenscar [BDV04]. Elle permet d'illustrer les capacités d'expression du profil pour construire des systèmes temps-réel. L'objectif de l'élaboration de cette étude de cas est double :

1. Valider l'ensemble des règles de tissage et de génération de code élaboré précédemment avant de passer à leur automatisation dans la partie dorsale.
2. Valider l'automatisation sur cette étude de cas une fois les règles implantées dans la partie dorsale.

Nous commençons, dans une première partie, par présenter cette étude de cas en décrivant l'architecture du système de base de cette application. Ensuite, nous expliquons comment nous l'avons étendu avec la programmation aspect en spécifiant la préoccupation transversale que nous mettrons en œuvre dans cette étude de cas. Nous présentons à la fin de cette section les résultats obtenus et leurs interprétations.

5.3.1 Description du système de base

L'application que nous avons choisi présente un système de gestion de charge de travail. Il s'agit d'une tâche périodique pouvant gérer des charges de travail variables. Les travaux réguliers sont effectués par une tâche périodique de haute priorité. Les travaux supplémentaires (irréguliers) sont délégués à une tâche sporadique de moindre priorité. Par ailleurs, le système est capable de recevoir des interruptions venant de l'extérieur. Ces interruptions sont reçues par une tâche de très haute priorité et sont enregistrées dans un tampon spéci-

fique. Le traitement de ces interruptions est délégué à une tâche sporadique de très faible priorité qui est réveillée de temps en temps par la tâche périodique principale.

L'application est ainsi formée par :

- Quatre tâches (*Tasks*) citées par ordre décroissant de priorité :
  1. `External Event Server` : une tâche sporadique [1] qui effectue la réception des interruptions extérieures et leur enregistrement dans un tampon spécifique,
  2. `Regular Producer` : une tâche périodique [2] qui effectue la charge de travail régulière. Il délègue, sous des conditions spécifiques, la charge de travail supplémentaire et le traitement des interruptions extérieures à d'autres tâches,
  3. `On Call Producer` : une tâche sporadique qui effectue la charge de travail supplémentaire,
  4. `Activation Log Reader` : une tâche sporadique qui effectue une quantité de travail correspondant au traitement de la dernière interruption reçue.
- Trois données partagées et protégées :
  1. `Request Buffer` : un tampon qui reçoit les ordres de travaux supplémentaires. Il est rempli par `Regular Producer` et consulté par `On Call Producer`,
  2. `Event Queue` : une file d'attente des interruptions extérieures. Elle est remplie par le périphérique émettant les interruptions et consultée par External Event Server,
  3. `Activation Log` : un journal des interruptions à traiter. Il est rempli par `External Event Server` et consulté par `Activation Log Reader`.
- Deux entités passives :
  1. `Activation Manager` : une entité qui assure la synchronisation des activations des tâches après l'élaboration. En effet, elle prévoit l'instant auquel toutes les tâches périodiques dans le système commencent leurs exécutions.

1. Une tâche sporadique peut être activée par un évènement à un instant aléatoire mais elle se caractérise par un délai minimal entre deux activations successives.
2. Une tâche périodique est une tâche cyclique activée à des instants fixes.

Table 5.1 – Propriétés non fonctionnelles des tâches [BDV04].

Tâche	**Type**	**Période**	**Échéance**	**WCET**	**Priorité**
Regular Producer	périodique	1000 *ms*	500 *ms*	480 *ms*	7
On Call Producer	sporadique	1000 *ms*	800 *ms*	240 *ms*	5
Activation Log Reader	sporadique	1000 *ms*	1000 *ms*	120 *ms*	3
External Event Server	sporadique	5000 *ms*	100 *ms*	20 *ms*	11

2. `Production Workload` : une entité qui abrite l'opération *Small Whetstone* [CWS76] qui traite les travaux demandés par une des tâches.

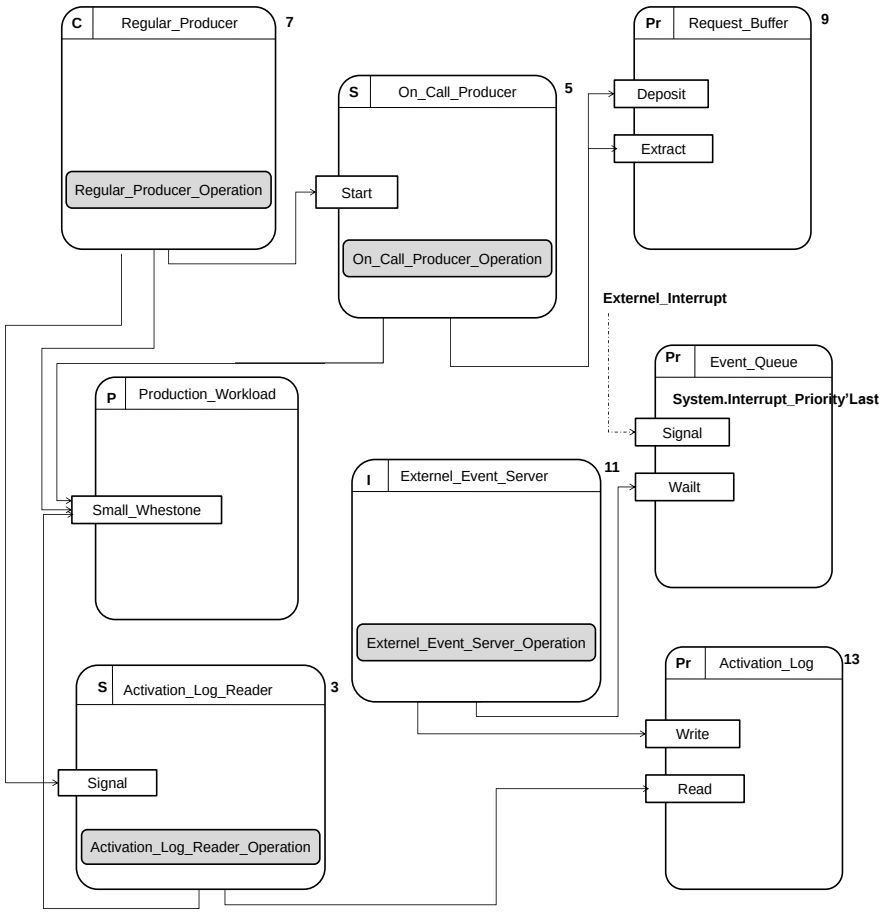
Le tableau 5.1 présente l'ensemble des propriétés non fonctionnelles de chaque tâche. Les valeurs des pires temps d'exécution (**WCET**) des différentes tâches sont déduites à partir du code Ada décrivant leurs comportements fourni par [BDV04]. La colonne **Période** indique dans le cas d'une tâche sporadique le temps minimal entre deux activations successives.

La figure 5.2 montre la description de l'architecture de l'application `Workload Manager` avec la méthode HRT-HOOD (*A Structured Design Method for Hard Real-Time Ada Systems*) [BW95], accompagnée d'une légende rappellant la signification des symboles et notations utilisés dans le schéma de cette architecture.

5.3.2 Description du système étendu avec les aspects

Nous avons présenté dans la section précédente l'architecture de base (composants métiers) de l'application gestion de charge de travail. Cependant, les propriétés transversales (ou techniques) ne sont pas prises en compte dans cette description. Nous nous intéressons dans cette partie à spécifier une propriété transversale en présentant le code AspectAda l'implantant tout en spécifiant les composants métiers sur lesquels elle agit.

La propriété transversale choisie est la gestion de la trace d'exécution : la **traçabilité** (Eng. *logging*). En effet, la traçabilité consiste à ajouter des traitements dans notre application pour permettre l'émission et le stockage de messages suite à des évènements. Pour le cas de notre application `Workload_Manager`, les évènements peuvent être :



Légende:

Étiquette	Position	Signification
n	côté droite de l'objet	Priorité de base d'une tâche ou d'un objet protégé
S	coin gauche supérieur de l'objet	Tâche sporadique (Sporadic task)
C		Tâche périodique (cyclic task)
I		Tâche d'interruption sporadique (Interrupt Sporadic task)
Pr		Objet protégé (Protected Object)
P		Entité passive (Passive Object)

Op_Name — Opération exportée par l'objet pour l'invoquer

→ Invocation d'une opération exportée par un objet

Op_Name — Opération interne d'une tâche

Figure 5.2 – Architecture de base de l'application `Workload Manager` [BDV04]

– début et fin de l'exécution d'une tâche,

– arrivée d'une interruption,

– activation d'une tâche sporadique.

Les messages emis sont très utiles pour suivre le fonctionnemment de notre application et détecter les comportements anormaux, s'ils y existent, nuisant aux exigences temps-réel. À travers la trace d'exécution, nous pouvons vérifier le respect des contraintes temps-réel :

– respect des priorités entre les tâches,
– respect des échéances des différentes tâches (en d'autres termes vérifier le déterminisme de l'application)

Pour mettre en œuvre la trace d'exécution de notre application, nous avons implanté le code AspectAda décrivant cette propriété technique en interceptant les évènements cités précédemment et en effectuant les traitements nécessaires pour chaque évènement. Le code AspectAda implanté est composé de deux parties : code de l'aspect et code du weaver. Nous présentons dans l'annexe B le code de l'aspect `Logger_Aspect` (listings B.1 et B.2) permettant de décrire le comportement des advices. Afin de garder une trace complète de l'exécution, l'aspect `Logger_Aspect` renferme cinq advices :

– **Before_Operation** : Il s'agit d'un advice *Before* associé au pointcut `Op_Pointcut` qui intercepte l'exécution des opérations des tâches `External_Event_Server`, `Regular_Producer` et `On_call_Producer`. Cet advice *Before* enregistre le début de l'exécution de l'opération interceptée selon la nature de la tâche courante.
– **After_Operation** : Il s'agit d'un advice *After* ayant un comportement très similaire à l'advice `Before_Operation` sauf que l'advice `After_Operation` enregistre la fin de l'exécution de la tâche interceptée par le pointcut `Op_Pointcut` parmi les trois tâches (`External_Event_Server`, `Regular_Producer` et `On_call_Producer`).
– **Before_Signal** : cet advice est créé dans le but d'enregistrer les activations du *thread* `Activation_Log_Reader`. En effet, ce dernier est signalé par le *thread* `Regular_Producer` qui appele la procédure 'Signal'. Pour ce faire, l'advice `Before_Signal` est associé au *pointcut* `Signal_PC` permettant d'intercepter l'appel à la procédure 'Signal'.
– **After_Start** : cet advice permet de sauvgarder les activations du *thread* `On_Call_Producer`. Le *thread* `Regular_Producer` active le *thread*

`On_Call_Producer` en lui envoyant la charge de travail supplémentaire. Ceci est fait à travers un appel à la fonction 'Start' qui prend en paramètre la charge supplémentaire. Par ailleurs, l'advice `After_Start` est associé au *pointcut* `Start_PC` permettant d'intercepter l'appel de la fonction 'Start' du *thread* `On_Call_Producer`.

– **Around** : cet advice est associé au *pointcut* `Log_Reader_PC` qui intercepte l'exécution de l'opération du *thread* `Activation_Log_Reader`. Le comportement de cet advice Around est divisé en trois parties :

  (1) Avant l'exécution de l'opération du *thread* `Activation_Log_Reader` : Enregistrer le début de l'exécution de ce *thread*,

  (2) Exécution de l'opération du *thread* `Activation_Log_Reader` (en faisant appel à l'instruction *proceed*),

  (3) Après l'exécution de l'opération du *thread* `Activation_Log_Reader` : Enregistrer la nouvelle interruption lue par ce *thread* (évidemment s'il y a une nouvelle interruption enregistrée dans le journal `Activation_Log`). Puis sauvegarder la fin de l'exécution de ce *thread*.

À travers notre étude de cas, nous avons employé les différents types d'advice et de *joinpoint*. Ceci permet d'appliquer la majorité des règles de tissage et de génération de code citées dans le chapitre 4.

Le *weaver* `Workload_Rules` définit les pointcuts permettant d'intercepter les évènements voulus. Il crée une instance de l'aspect `Logger_Aspect` permettant d'associer les pointcuts aux advices. Nous présentons dans l'annexe B le code du `Workload_Rules.aaw` (listing B.3).

5.3.3 Tissage et Génértion de code

Le tissage et la génération de code s'effectuent pour le moment à la main. À partir du code AspectAda décrit précédemment et du code fonctionnel Ada de l'application `Workload_Manager`, nous avons appliqué les règles de tissage et de génération de code pour

produire un ensemble de fichiers sources Ada. Nous donnons ci-après la liste des fichiers produits :

- `Logger_Aspect.ad[b|s]` : contiennent la spécification et le corps du paquetage `Logger_Aspect`. Ils projettent la spécification et le corps de l'aspect en des entités Ada.
- `AspectAda_Types.ad[b|s]` : contiennent la spécification et le corps du paquetage `AspectAda_Types`. Ils contiennent les types de données utilisés par la `Runtime` et le code généré. De plus, ils implantent les routines nécessaires pour les conversions des types.
- `Regular_Producer_Parameters.adb`, `On_Call_Producer.adb` et `Externel_Event_Server.adb` : contiennent respectivement les corps des paquetages `Regular_Producer_Parameters`, `On_Call_Producer` et `Externel_Event_Server` après le tissage des advices `Before_Operation` et `After_Operation`.
- `Tmp_Activation_Log_Reader.ad[b|s]` : contiennent la spécification et le corps du paquetage `Tmp_Activation_Log_Reader`. Ils implantent le tissage de l'advice `Before_Signal` associé à l'appel de la procédure `Signal` du paquetage `Activation_Log_Reader`.
- `Tmp_On_Call_Producer.ad[b|s]` : contiennent la spécification et le corps du paquetage `Tmp_On_Call_Producer`. Ils implantent le tissage de l'advice `After_Start` associé à l'appel de la fonction `Start` du paquetage `On_Call_Producer`.
- `Activation_Log_Reader_Parameters.ad[b|s]` : contiennent la spécification et le corps du paquetage `Activation_Log_Reader_Parameters`. Ils implantent le tissage de l'advice `Around` associé à l'exécution de la procédure `Activation_Log_Reader_Operation` du paquetage `Activation_Log_Reader_Parameters`.

5.3.4 Exécution et Interprétation des Résultats

Nous avons choisi d'envoyer des interruptions de manière aléatoire avec un temps d'arrivée minimal entre deux interruptions supérieur ou égal à 5 secondes. Ceci vise à respecter les contraintes de sporadicité du *thread* `External_Event_Server` qui gère la réception des interruptions. Avant d'entamer l'exécution de l'application, nous avons effectué une étude théorique de l'exécution des différents *threads* (figure 5.3) en nous basant sur les données du tableau 5.1. Ceci est dans le but de comparer les résultats obtenus après l'exécution (la pratique) avec les résultats de l'étude théorique. Nous avons utilisé l'algorithme d'ordonnancement par priorité statique (*RMS : Rate Monotonic Scheduling*) [LL73] pour faire l'étude théorique.

Par ailleurs, les instants des activations des *threads* sporadiques `Activation Log Reader` et `On Call Producer` sont déduites à partir du comportement du *thread* `Regular Producer`. En effet, le *thread* `Regular Producer` se charge de l'activation de ces deux *threads* sporadiques. Pour l'activation de chacun d'eux, `Regular Producer` définit et vérifie une condition spécifique. Une fois une condition est vérifiée, alors il y aura activation du *thread* (soit `On Call Producer` soit `Activation Log Reader`) associé à cette condition.

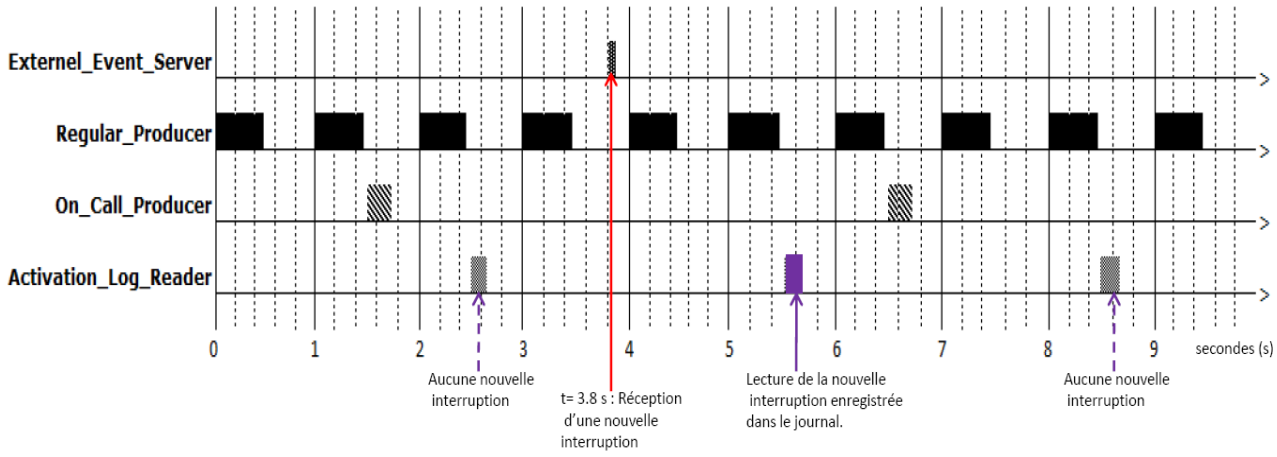


Figure 5.3 – Étude théorique de l'exécution du workload Manager

En examinant la figure 5.3, nous remarquons qu'à sa première activation (à l'instant 2.5 secondes), le *thread* `Activation Log Reader` n'a pas trouvé aucune interruption en-

registrée par le *thread* `Externel Event Sever`. Par contre, à sa deuxième activation (à l'instant 5.5 secondes), il lit l'interruption reçue par le *thread* `Externel Event Sever` à l'instant 3.8 secondes.

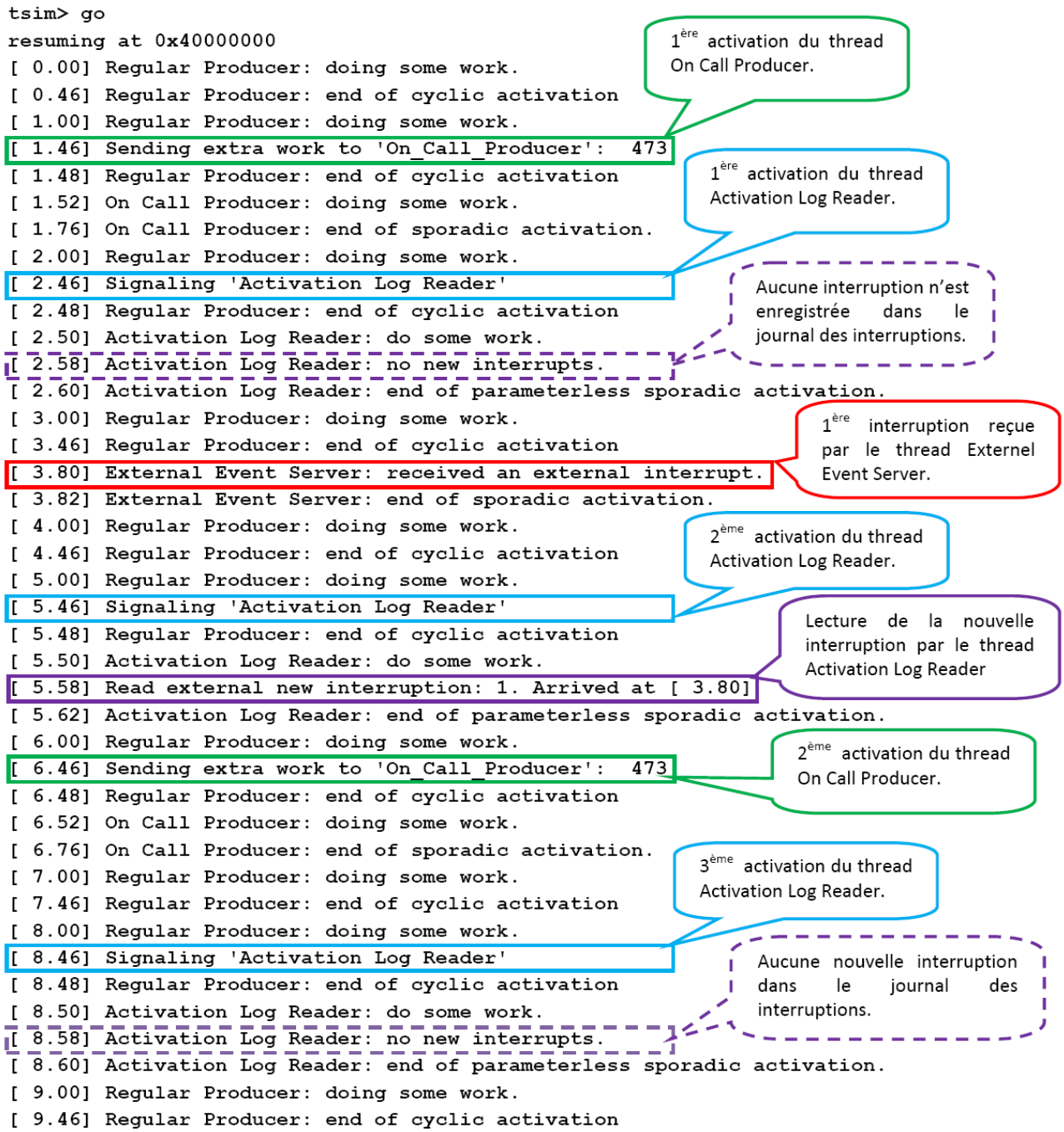


Figure 5.4 – Trace d'exécution du workload Manager sur tsim

Notre application temps-réel est compilée par le compilateur Gnat For Leon. Puis, elle a été exécutée à l'aide du simulateur du processeur Leon2 **Tsim**. Pour des limites liées au simulateur, nous avons modifié le thread `External_Event_Server` pour rendre interne l'envoi des interruptions. Une partie de la trace obtenue de l'exécution de l'application `workload_Manager` est présentée dans la figure 5.4. Nous remarquons que tous les travaux supplémentaires sont exécutés. Toutes les interruptions externes ont été traitées par la tâche `Activation_Log_Reader` ce qui montre que le dimensionnement du système utilisant une file d'attente de taille 1 pour les interruptions est correct puisqu'aucune interruption n'a été perdue.

Si nous comparons la trace d'exécution avec l'étude théorique effectuée, nous pouvons déduire que :

1. les tâches respectent bien leurs échéances ;
2. les priorités entre les tâches sont aussi respectées.

Par conséquent, l'application obtenue après le tissage des aspects est déterministe. Elle respecte les contraintes temporelles de l'application d'origine décrites dans le tableau 5.1 (avant l'introduction des aspects). Ceci montre que les règles de tissage et de génération de code élaborées n'affectent pas le déterminisme de l'application.

5.4 Synthèse

En guise de résumé, nous avons commencé l'implantation de la partie frontale relative au code du *weaver* permettant de faire l'analyse lexicale, syntaxique et sémantique.

Il nous reste à implanter la partie frontale relative au code `aspect` d'une manière similaire à la partie frontale du code `weaver`. En plus, l'implantation de la partie dorsale qui permet le tissage et la génération automatique du code n'est pas encore achevée. En effet, le tissage et la génération de code se fait, pour le moment, manuellement.

Pour atteindre tous nos objectifs fixés dès le début de ce stage, nous envisageons, à court terme, compléter la partie frontale et mettre en place la partie dorsale. Le travail

restant sur le nouveau compilateur réside essentiellement dans l'implantation de la partie dorsale qui sera relativement simple et rapide étant donnée que le plus difficile (règles de tissage et de génération de code) a été réalisé et testé.

À travers l'étude de cas, nous avons pu développer un prototype pour tester notre approche et mettre en évidence notre processus de tissage et de génération de code.

Conclusion Générale et Perspectives

Nous avons proposé dans le cadre de nos travaux de mastère l'adaptation et l'extension du langage AspectAda pour les systèmes temps-réel en agissant sur sa syntaxe, sa bibliothèque de routines `Runtime` et son compilateur/tisseur prototype.

Nous avons adapté sa syntaxe en modifiant certaines constructions pour rendre leurs utilisations plus aisées et plus flexibles. Cette adaptation est également dans le but de rendre la syntaxe de AspectAda plus en conformité avec le langage Ada. De plus, nous avons étendu la syntaxe de AspectAda pour supporter plus de concepts de la programmation aspect. Outre l'adaptation et l'extension de la syntaxe de AspectAda, nous avons introduit des modifications à la `Runtime`. D'une part, nous avons éliminé de la `Runtime` toutes les constructions qui se contredisent avec les contraintes temps-réel. Et d'autre part, nous avons ajouté des fonctionnalités à la `Runtime` en vue d'exporter le contexte des *joinpoints* aux advices. De plus, nous avons défini une nouvelle architecture pour le compilateur/tisseur en évitant de tomber dans les mêmes pièges de conception rencontrés lors du développement de l'ancien compilateur.

Avant d'entamer l'implantation du compilateur, notre approche a mis l'accent sur l'opération de tissage (*weaving*) puisque cette opération peut compremettre le déterminisme du système. Pour ce faire, nous avons défini un ensemble riche de règles régissant l'opération de tissage en tenant compte des contraintes temps-réel. Ces règles ont été validées par différents tests.

Notre approche qui s'intègre dans le cadre de développement de systèmes temps-réel avec de l'orienté aspect, permet d'une part d'améliorer la syntaxe et la sémantique du lan-

gage AspectAda et d'autre part d'assurer le respect des contraintes temps réel en présence des aspects.

Pour mettre en œuvre notre approche, nous avons commencé le développement de la partie frontale du compilateur AspectAda permettant d'analyser (lexeur et parseur) les unités de compilation '*weaver*'. Il nous reste à (1) achever l'implantation de la partie frontale et (2) implanter la partie dorsale relative a ce compilateur permettant le tissage et la génération automatique du code Ada tissé. En effet, le tissage et la génération de code se font, pour le moment, manuellement. L'implantation de la partie dorsale du compilateur sera relativement simple vu que les règles de tissage et de transformations sont déjà définies.

Nous avons validé notre solution à travers une étude de cas de gestion de charge de travail. Certes la solution que nous avons proposée répond à la majorité des objectifs fixés au début de ce mastère, toutefois différents points doivent être pris en considération. En premier temps, nous envisageons à définir des algorithmes pour l'interception des différents types *joinpoints* et poursuivre l'implantation des parties frontales et dorsales du compilateur. Nous visons de plus, à moyen terme, à trouver une solution pour faire face à l'interception des sous-programmes génériques. Finalement, nous souhaitons à long terme, intégrer le compilateur AspectAda dans le compilateur Ada GNAT.

Bibliographie

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr, *Basic concepts and taxonomy of dependable and secure computing*, IEEE Transactions on Dependable and Secure Computing **1** (2004), 11–33.

[Aut09] Thomas Autret, *Génération de code real-time java pour systèmes temps-réel*, Master's thesis, Université Pierre & Marie Curie, ParisVI, France, September 2009.

[Bar03] John Barnes, *High integrity software : The spark approach to safety and security*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[BD99] C. Bonnet and I. Demeure, *Introduction aux systèmes temps réel*, Collection pédagogique de télécommunications, Hermès Science Publications, 1999.

[BDV04] Alan Burns, Brian Dobbing, and Tullio Vardanega, *Guide for the use of the ada ravenscar profile in high integrity systems*, ACM SIGAda Ada Letters **XXIV** (2004), no. 2, 1–74.

[Bet09] Farida Bettou, *Gestion de la qualité de service lors du développement de logiciels orientés aspect*, Master's thesis, Ecole Doctorale de lEst en Informatique Pole de CONSTANTINE, 2009.

[BSB91] James B. Bladen, David Spenhoff, and Steven J. Blake, *Ada semantic interface specification (asis)*, Proceedings of the conference on TRI-Ada '91 : today's accomplishments ; tomorrow's expectations (New York, NY, USA), TRI-Ada '91, ACM, 1991, pp. 6–15.

[BW95] A. Burns and A.J. Wellings, *Hrt-hood : A structured design method for hard real-time ada systems*, Real-Time Safety Critical Systems, Elsevier, 1995.

[Car00] Martin C. Carlisle, *An automatic object-oriented parser generator for ada*, Ada Lett. **XX** (2000), no. 2, 57–62.

[CKFS01] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn, *Using aspectc to improve the modularity of path-specific customization in operating system code*, Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering (New York, NY, USA), ESEC/FSE-9, ACM, 2001, pp. 88–98.

[CS02] Martin C. Carlisle and Ricky E. Sward, *An automatic "visitor" generator for ada*, Ada Lett. **XXII** (2002), no. 3, 42–47.

[CWS76] H J Curnow, B A Wichmann, and Tij Si, *A synthetic benchmark*, The Computer Journal **19** (1976), 43–49.

[FA04] FAA, *Handbook for Object-Oriented Technology in Aviation (OOTiA)*, FAA, October 2004.

[FECA05] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit (eds.), *Aspect-oriented software development*, Addison-Wesley, Boston, 2005.

[GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides, *Design patterns : Elements of reusable object-oriented software*, Addison-Wesley, Reading, MA, 1995.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold, *An overview of aspectj*, Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science, vol. 2072, Springer, 2001, pp. 327–353.

[Kim02] H. Kim, *AspectC# : An AOSD implementation for C#*, Master's thesis, Department of Computer Science Trinity College Dublin, 2002.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin, *Aspect-oriented programming*, ECOOP, SpringerVerlag, 1997.

[LL73] Chang L. Liu and James W. Layland, *Scheduling algorithms for multiprogramming in a hard-real-time environment*, J. ACM **20** (1973), no. 1, 46–61.

[MO03] Mira Mezini and Klaus Ostermann, *Conquering aspects with caesar*, AOSD '03 : Proceedings of the 2nd international conference on Aspect-oriented software development (New York, NY, USA), ACM, 2003, pp. 90–99.

[PC05] Knut H. Pedersen and Constantinos Constantinides, *Aspectada : aspect oriented programming for ada95*, Ada Lett. **XXV** (2005), no. 4, 79–92.

[PDFS02] Renaud Pawlak, Laurence Duchien, Gerard Florin, and Lionel Seinturier, *JAC : un framework pour la programmation orientée aspect en java*, L'OBJET **8** (2002), no. 4, 145–168.

[PPS02] Isabelle Puaut, Isabelle Puaut, and Projet Solidor, *Real-time performance of dynamic memory allocation algorithms*, 14 th Euromicro Conference on Real-Time Systems (ECRTS'02, IEEE Computer Society, 2002, pp. 4–1.

[PV98] Luís Miguel Pinho and Francisco Vasques, *To ada or not to ada : Adaing vs. javaing in real-time systems*, November 1998.

[RS96] Sergey Rybin and Alfred Strohmeier, *Ada and asis : justification of differences in terminology and mechanisms*, Proceedings of the conference on TRI-Ada '96 : disciplined software development with Ada (New York, NY, USA), TRI-Ada '96, ACM, 1996, pp. 249–254.

[SAE04] SAE, *Architecture Analysis & Design Language*, September 2004, `http://www.sae.org`.

[SGSP02] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat, *AspectC++ : An aspect-oriented extension to C++*, February 2002, pp. 53–60.

[SP02] Wolfgang Schult and Andreas Polze, *Aspect-oriented programming with C# and .net*, ISORC '02 : Proceedings of the Fifth IEEE International Symposium on

Object-Oriented Real-Time Distributed Computing (Washington, DC, USA), IEEE Computer Society, 2002, p. 241.

[SRL90] L. Sha, R. Rajkumar, and J. P. Lehoczky, *Priority Inheritance Protocols : An Approach to Real-Time Synchronization*, IEEE Trans. Comput. **39** (1990), no. 9, 1175–1185.

[tea02] JBoss team., *Jboss aspect oriented programming*, 2002, available at `http://www.jboss.org/,`.

[VZH06] T. Vergnaud, B. Zalila, and J. Hugues, *Ocarina : a Compiler for the AADL*, Tech. report, École Nationale Supérieure des Télécommunications, jun 2006.

[WG05] Ada Working Group, *Ada Reference Manual*, ISO/IEC, 2005, Available at `http://www.adaic.com/standards/05rm/RM-Final.pdf`.

[Wic99] J. C. Wichman, *The development of a preprocessor to facilitate composition filters in the Java language*, Master's thesis, University of Twente, 1999.

[WM95] Reinhard Wilhelm and Dieter Maurer, *Compiler design*, International Computer Science Series, Addison-Wesley, 1995, Second Printing.

Annexe

A Application des Règles de Transformation et de Tissage de Code

Dans cette annexe nous présentons des exemples pour appliquer les règles de transformation et de tissage de code citées dans le chapitre 4.

A.1 Exemple de tissage des advices *before* et *after* associés à des *joinpoints execution*

L'exemple est constitué de trois grandes parties de code : code métier en Ada, code des aspects en AspectAda et le code tissé généré en Ada.

1. Code métier : Il s'agit d'un code source Ada qui se compose de :

• Un paquetage '`Calcul`' contenant deux fonctions '`Somme`' et '`Mult`' et une procédure '`Double`'. Les listings A.1 et A.2 présentent respectivement la spécification et le corps du paquetage `Calcul`.

Listing A.1 – Spécification du paquetage Calcul

```
1 package Calcul is
2  function  Somme  (A,B : in Integer) return Integer;
3  function  Mult   (A,B : in Integer) return Integer;
4  procedure Double (A : in out Integer);
5 end Calcul;
```

Listing A.2 – Corps du paquetage Calcul

```
1 with Ada.Text_IO;
2 package body Calcul is
3
4 function Somme (A,B : in Integer) return Integer is
5   S : Integer;
6 begin
```

```
7     S := A + B;
8     return S;
9  end Somme;
10
11 function Mult (A,B : in Integer) return Integer is
12    M : Integer ;
13 begin
14    M := A * B;
15    return M;
16 end Mult;
17
18 procedure Double (A : in out Integer) is
19 begin
20     A := Mult (A,2);
21 end Double;
22
23 end Calcul;
```

• Une procédure '`main`' qui utilise le paquetage `Calcul`.

Listing A.3 – Procédure main

```
1  with Ada.Text_IO;
2  with Calcul;
3
4  procedure main is
5     mul : Integer := Calcul.Mult(2,Calcul.Somme (3,4));
6     d   : Integer := 20;
7  begin
8     Calcul.Double(A);
9     Ada.Text_IO.Put_Line(" I'm the main procedure, I call Calcul.Somme "
10                        & Integer'Image(Calcul.Somme (3,4)));
11 end main;
```

2. Code AspectAda : Il contient le code de l'aspect et le code du *weaver* :

• Le code de l'aspect : l'aspect `Detailed_Aspect` contient un *advice Before* et un *advice After*. Grâce aux fonctions de conversion des types fournies par le paquetage généré `AspectAda_Types` (voir section 3.3.1 du chapitre 3) :

– l'*advice Before* accède au premier argument des *joinpoints* qui lui sont associés.

– l'*advice After* permet d'accéder à la valeur retournée par les *joinpoints* interceptés et de l'afficher.

Les listings A.4 et A.5 présente la description de l'aspect (spécification et corps).

Listing A.4 – Spécification de l'aspect Detailed_Aspect

```
1  with AspectAda_Types;
2  generic
3      First_Pointcut : pointcut;
4  aspect Detailed_Aspect is
5     advice Before (thisJoinPoint : AspectAda_Types.Join_Point);
6     for Before'pointcut use First_Pointcut;
7
8     advice After (thisJoinPoint : AspectAda_Types.Join_Point);
```

```
9     for After'pointcut use First_Pointcut;
10 end Detailed_Aspect;
```

Listing A.5 – Corps de l'aspect Detailed_Aspect

```
1  with Ada.Text_IO;
2  with Aspect_Ada
3  aspect body Detailed_Aspect is
4     advice Before (thisJoinPoint : AspectAda_Types.Join_Point) is
5        Args            : AspectAda_Types.Arg_Array;
6        First_Arg       : AspectAda_Types.Arg;
7        First_Arg_Value : Integer;
8     begin
9        Ada.Text_IO.Put_Line ("Entering => "
10                          & Aspect_Ada.Get_Signature(thisJoinPoint));
11       Args := Aspect_Ada.Get_Args(thisJoinPoint);
12       First_Arg := Args(1);
13       Ada.Text_IO.Put_Line ("The name of first arg of the JP is "
14                                   & Aspect_Ada.Get_Name(First_Arg));
15       First_Arg_Value := AspectAda_Types.Extract_Integer
16               (AspectAda_Types.Integer_Holder
17                         (Aspect_Ada.Get_Value_Access(First_Arg).all));
18       Ada.Text_IO.Put_Line ("The value of first arg of the JP is "
19                                                & First_Arg_Value'Img);
20    end Before;
21
22    advice After (thisJoinPoint : AspectAda_Types.Join_Point) is
23       returned_value : Integer;
24    begin
25       Ada.Text_IO.Put_Line ("Exiting => "
26                           & Aspect_Ada.Get_Signature(thisJoinPoint));
27       returned_value := AspectAda_Types.Extract_Integer
28          (AspectAda_Types.Integer_Holder
29          (Aspect_Ada.Get_Returned_Value_Access(thisJoinPoint).all));
30       Ada.Text_IO.Put_Line ("The returned value of the JP is "
31                                                & returned_value'Img);
32    end After;
33 end Detailed_Aspect;
```

• Le code du *weaver* (les règles de composition) : le listing A.6 présente le fichier '`Rules.aaw`' à travers lequel nous définissons un *pointcut* permettant d'intercepter, selon le '*joinpoint_pattern*', l'exécution des deux fonctions `Somme` et `Mult`. De plus, une instance de l'aspect appelée '`My_First_Aspect`' permet d'associer le *pointcut* créé aux *advices* de l'aspect.

Listing A.6 – Code du *weaver*

```
1  with Detailed_Aspect;
2  weaver Rules is
3    Calcul_PC : pointcut :=
4                execution(function Calcul. (in Integer , in Integer) return Integer );
5    aspect My_First_Aspect is new Detailed_Aspect(Calcul_PC);
6  end Rules;
```

3. Code source Ada généré : Nous obtenons après tissage un ensemble de fichiers source Ada :

1. Le paquetage '`Detailed_Aspect`' est le paquetage généré à partir de l'aspect '`Detailed_Aspect`' :

Listing A.7 – Spécification du paquetage projection de l'aspect

```
with AspectAda_Types;
package Detailed_Aspect is
   function Before (thisJoinPoint : AspectAda_Types.Join_Point) return Boolean;
   procedure After (thisJoinPoint : AspectAda_Types.Join_Point);
end Detailed_Aspect;
```

Listing A.8 – Corps du paquetage Detailed_Aspect

```
with Ada.Text_IO;
with Aspect_Ada;

package body Detailed_Aspect is

   function Before (thisJoinPoint : AspectAda_Types.Join_Point) return Boolean
   is
      Args            : AspectAda_Types.Arg_Array;
      First_Arg       : AspectAda_Types.Arg;
      First_Arg_Value : Integer;
   begin
      Ada.Text_IO.Put_Line ("Entering => "
                            & Aspect_Ada.Get_Signature(thisJoinPoint));
      Args := Aspect_Ada.Get_Args(thisJoinPoint);
      First_Arg := Args(1);
      Ada.Text_IO.Put_Line ("The name of first arg of the JP is "
                                  & Aspect_Ada.Get_Name(First_Arg));
      First_Arg_Value := AspectAda_Types.Extract_Integer
              (AspectAda_Types.Integer_Holder
                          (Aspect_Ada.Get_Value_Access(First_Arg).all));
      Ada.Text_IO.Put_Line ("The value of first arg of the JP is "
                                             & First_Arg_Value'Img);
      return True;
   end Before;

   procedure After (thisJoinPoint : AspectAda_Types.Join_Point) is
      returned_value : Integer;
   begin
      Ada.Text_IO.Put_Line ("Exiting => "
                           & Aspect_Ada.Get_Signature(thisJoinPoint));
      returned_value := AspectAda_Types.Extract_Integer
         (AspectAda_Types.Integer_Holder
         (Aspect_Ada.Get_Returned_Value_Access(thisJoinPoint).all));
      Ada.Text_IO.Put_Line ("The returned value of the JP is "
                                             & returned_value'Img);
   end After;
end Detailed_Aspect;
```

2. Un paquetage nommé '`AspectAda_Types`' comme nous avons vu précédemment dans le chapitre 3 à la section 3.3. Il contient tous les types nécessaires pour stocker les in-

formations des *joinpoints* ainsi que les routines permettant de manipuler les valeurs des paramètres des *joinpoints* et leurs valeurs retournées. Dans cet exemple, les paramètres des fonctions interceptées ainsi que leurs valeurs retournées sont de type `Integer`. Donc, '`AspectAda_Types`' contient le type '`Integer_Holder`' et les deux fonctions "accesseurs" '`Extract_Integer`' et '`Insert_Integer`'. Ci-dessous, un extrait de la spécification (listing A.9) et le corps (listing A.10) du paquetage '`AspectAda_Types`'.

Listing A.9 – Extrait de la spécification du paquetage `AspectAda_Types`

```
1  package AspectAda_Types is
2    Nmax : Positive := 2;
3    type Arg_Array is array (Positive range 1 .. Nmax) of Arg;
4    --...
5    type Integer_Holder is new Aspect_Ada.Value_Holder with
6       record
7          V : Integer;
8       end record;
9    function Extract_Integer (H : in Integer_Holder) return Integer;
10   function Insert_Integer (V : in Integer) return Integer_Holder;
11   --...
12 end AspectAda_Types;
```

Listing A.10 – Corps du paquetage `AspectAda_Types`

```
1  package body AspectAda_Types is
2     function Extract_Integer (H : Integer_Holder) return Integer is
3     begin
4        return H.V;
5     end Extract_Integer;
6
7     function Insert_Integer (V : Integer) return Integer_Holder is
8        H : Integer_Holder;
9     begin
10       H.V := V;
11       return H;
12    end Insert_Integer;
13 end AspectAda_Types;
```

3. Le *pointcut* `Calcul_PC` intercepte l'exécution des deux fonctions `Somme` et `Mult` du paquetage `Calcul`. Alors, le corps de ce paquetage sera recopié, modifié et généré dans le code résultant (listing A.11). Les deux fonctions interceptées seront tissées dans le paquetage généré. Les deux fonctions `Somme` et `Mult` sont interceptées par les advices *Before* et *After* de l'aspect `Detailed_Aspect`. Or, les deux advices prennent en entrée un paramètre nommé *thisJoinPoint* de type `Join_Point`. C'est-à-dire ils accèdent aux informations des *joinpoints* interceptés. Le paramètre `thisJoinPoint` doit être créé dans chacun des deux sous programmes `Somme` et `Mult` à travers la fonction `Create_Join_Point` de la `Runtime`.

Après sa création, le paramètre `thisJoinPoint` sera passé aux sous programmes présentant les advices *Before* et *After*. (les modifications par rapport au fichier corps du paquetage `Calcul` original sont colorées en bleu) :

Listing A.11 – Corps du paquetage Calcul tissé et généré

```
1  with Ada.Text_IO;
2
3  with Aspect_Ada;
4  with Detailed_Aspect;
5  with AspectAda_Types;
6
7  package body Calcul is
8
9     function Somme (A,B : Integer) return Integer is
10
11    -- Creation of Args of the JP
12    Value_Access_Arg1 : aliased AspectAda_Types.Integer_Holder :=
13                                   AspectAda_Types.Insert_Integer(A);
14    Value_Access_Arg2 : aliased AspectAda_Types.Integer_Holder :=
15                                   AspectAda_Types.Insert_Integer(B);
16    Args : AspectAda_Types.Arg_Array := (
17    1 => Aspect_Ada.To_Arg ("A", "Integer",
18                            AspectAda_Types.Default_In_Mode,
19                            Value_Access_Arg1'Unchecked_Access),
20     2 => Aspect_Ada.To_Arg ("B", "Integer",
21                            AspectAda_Types.Default_In_Mode,
22                            Value_Access_Arg2'Unchecked_Access));
23
24    -- Creation of the joinpoint
25    thisJoinPoint  :  AspectAda_Types.Join_Point :=
26              Aspect_Ada.Create_Join_Point("Calcul.Somme",
27                                           Args,
28                                           AspectAda_Types.Function_Kind,
29                                           "Integer");
30
31    -- Call of the function Before similing the before advice
32    -- the used variable must have a name different from the names
33    -- of local and global variables and arguments of the joinpoint
34    x : Boolean := Detailed_Aspect.Before(thisJoinPoint);
35
36    -- returned_value is variable in which we stock
37    -- the Access to the returned value.
38    returned_value : aliased AspectAda_Types.Integer_Holder;
39
40    -- declaring local variables
41     S : integer;
42   begin
43       -- Body of the JP
44       S := A + B;
45
46       --  storing of the returned value into returned_value
47       --  in order to recover it in the after advice
48       returned_value := AspectAda_Types.Insert_Integer(S);
49       Aspect_Ada.Set_Returned_Value_Access
50                (thisJoinPoint, returned_value'Unchecked_Access);
51
52       -- Call of the procedure After similing the after advice.
53      Detailed_Aspect.After (thisJoinPoint);
54
55       -- The return of the JP
```

```
56          return S;
57      end Somme;
58
59  function Mult (A,B : in Integer) return Integer is
60
61  Value_Access_Arg1 : aliased AspectAda_Types.Integer_Holder :=
62                                      AspectAda_Types.Insert_Integer(A);
63      Value_Access_Arg2 : aliased AspectAda_Types.Integer_Holder :=
64                                      AspectAda_Types.Insert_Integer(B);
65      Args : AspectAda_Types.Arg_Array := (
66      1 => Aspect_Ada.To_Arg("A", "Integer",
67                             AspectAda_Types.Default_In_Mode,
68                             Value_Access_Arg1'Unchecked_Access),
69     2 => Aspect_Ada.To_Arg("B", "Integer",
70                              AspectAda_Types.Default_In_Mode,
71                               Value_Access_Arg2'Unchecked_Access));
72
73      thisJoinPoint  :  AspectAda_Types.Join_Point :=
74                Aspect_Ada.Create_Join_Point("Calcul.Somme",
75                                             Args,
76                                             AspectAda_Types.Function_Kind,
77                                             "Integer");
78
79      x : Boolean := Detailed_Aspect.Before(thisJoinPoint);
80
81      returned_value : aliased AspectAda_Types.Integer_Holder;
82
83     M : Integer ;
84  begin
85
86     M := A * B;
87
88      returned_value := AspectAda_Types.Insert_Integer(M);
89      Aspect_Ada.Set_Returned_Value_Access
90                (thisJoinPoint, returned_value'Unchecked_Access);
91
92      Detailed_Aspect.After (thisJoinPoint);
93
94      return M;
95  end Mult;
96
97  procedure Double (A : in out Integer) is
98  begin
99     A := Mult (A,2);
100 end Double;
101
102 end Calcul;
```

A.2 Exemples de tissage des advices *around* associés à des *joinpoints execution*

Les règles de tissage d'un *advice around* associé à des *joinpoints* de type *execution* sont divisés en deux classes selon l'utilisation de l'instruction '*proceed*'. Pour cela, nous présentons dans cette section un exemple d'application pour chaque classe de règles.

A.2.1 Exemple de tissage d'un advice *around* sans l'instruction *proceed*

L'exemple est constitué de trois grandes parties de code : code métier en Ada, code des aspects en AspectAda et le code tissé généré en Ada.

1. Code métier : nous prenons le même code fonctionnel utilisé dans l'exemple A.1.

2. Code AspectAda : Il contient le code de l'aspect et le code du *weaver* :

• Le code de l'aspect : l'aspect `Short_Aspect` contient un *advice after* et un *advice around*. Ce dernier possède un argument de type `Join_Point` et il retourne un `Integer`. Les listings A.12 et A.13 présente la description de l'aspect (spécification et corps) en AspectAda.

Listing A.12 – Spécification de l'aspect Short_Aspect

```
with AspectAda_Types;

generic
    Pc : Pointcut;
aspect Short_Aspect is

   advice Around (thisJoinPoint : AspectAda_Types.Join_Point) return Integer;
   for Around'Pointcut use Pc;

   advice After (thisJoinPoint : AspectAda_Types.Join_Point);
   for After'Pointcut use Pc;

end Short_Aspect;
```

Listing A.13 – Corps de l'aspect Short_Aspect

```
with Ada.Text_IO;
with Aspect_Ada;

aspect body Short_Aspect is

   advice Around (thisJoinPoint : AspectAda_Types.Join_Point) return Integer is
      var : Integer :=122;
   begin
      Ada.Text_IO.Put_Line ("I replace the following joinpoint "
                            & Aspect_Ada.Get_Signature(thisJoinPoint));
      return var;
   end Around;

   advice After (thisJoinPoint : AspectAda_Types.Join_Point) is
      returned_value : Integer;
   begin
      Ada.Text_IO.Put_Line ("Exiting => "
                          & Aspect_Ada.Get_Signature(thisJoinPoint));
      returned_value := AspectAda_Types.Extract_Integer
         (AspectAda_Types.Integer_Holder
          (Aspect_Ada.Get_Returned_Value_Access(thisJoinPoint).all));
      Ada.Text_IO.Put_Line ("The returned value of the JP is "
                                          & returned_value'Img);
```

```
24    end After;
25
26 end Short_Aspect;
```

• Le code du *weaver* (les règles de composition) : le listing A.14 présente le fichier '`Rules.aaw`'.

Listing A.14 – Code du *weaver*

```
1 with Short_Aspect;
2 weaver Rules is
3   Calcul_PC : Pointcut := execution(function Calcul.Somme (..) return Integer);
4   aspect My_Short_Aspect is new Short_Aspect(Calcul_PC);
5 end Rules;
```

3. Code source Ada généré : Nous obtenons dans le code généré un ensemble de fichiers :

1. Le paquetage '`Short_Aspect`' (listings A.15 A.16) c'est le paquetage généré à partir de l'aspect '`Short_Aspect`' : Il contient seulement la procédure présentant l'*advice after*.

Listing A.15 – Spécification du paquetage Short_Aspect

```
1 with AspectAda_Types;
2
3 package Short_Aspect is
4    procedure After (thisJoinPoint : AspectAda_Types.Join_Point);
5 end Short_Aspect;
```

Listing A.16 – Corps du paquetage Short_Aspect

```
1  with Ada.Text_IO;
2  with AspectAda;
3
4  package body Short_Aspect is
5
6     procedure After (thisJoinPoint : AspectAda_Types.Join_Point) is
7        returned_value : Integer;
8     begin
9        Ada.Text_IO.Put_Line ("Exiting => "
10                             & Aspect_Ada.Get_Signature(thisJoinPoint));
11       returned_value := AspectAda_Types.Extract_Integer
12          (AspectAda_Types.Integer_Holder
13           (Aspect_Ada.Get_Returned_Value_Access(thisJoinPoint).all));
14       Ada.Text_IO.Put_Line ("The returned value of the JP is "
15                                                & returned_value'Img);
16    end After;
17
18 end Short_Aspect;
```

2. Le paquetage '`AspectAda_Types`' : le même paquetage généré (les listings A.9 et A.10) dans le premier exemple de cette annexe à la section A.1.

3. Puisque le *pointcut* `Calcul_PC` intercepte l'exécution de la fonction `Somme` du paquetage `Calcul`, alors le corps de ce paquetage sera recopié, modifié et généré dans le code résultant (listing A.17). (les modifications par rapport au fichier corps du paquetage `Calcul` original sont colorées en bleu) :

Listing A.17 – Corps du paquetage Calcul tissé et généré

```
1  with Ada.Text_IO;
2
3  with Aspect_Ada;
4  with Short_Aspect;
5  with AspectAda_Types;
6
7  package body Calcul is
8
9     function Somme (A,B : Integer) return Integer is
10
11    -- Creation of Args of the JP
12    Value_Access_Arg1 : aliased AspectAda_Types.Integer_Holder :=
13                                  AspectAda_Types.Insert_Integer(A);
14    Value_Access_Arg2 : aliased AspectAda_Types.Integer_Holder :=
15                                 AspectAda_Types.Insert_Integer(B);
16    Args : AspectAda_Types.Arg_Array := (
17    1 => Aspect_Ada.To_Arg ("A", "Integer",
18                            AspectAda_Types.Default_In_Mode,
19                            Value_Access_Arg1'Unchecked_Access),
20    2 => Aspect_Ada.To_Arg ("B", "Integer",
21                            AspectAda_Types.Default_In_Mode,
22                            Value_Access_Arg2'Unchecked_Access));
23
24    -- Creation of the joinpoint
25    thisJoinPoint  :  AspectAda_Types.Join_Point :=
26             Aspect_Ada.Create_Join_Point("Calcul.Somme",
27                                          Args,
28                                          AspectAda_Types.Function_Kind,
29                                          "Integer");
30
31    -- returned_value_Access is variable in which we stock
32    -- the Access to the returned value.
33    returned_value : aliased AspectAda_Types.Integer_Holder;
34
35    -- Replacing locals variables of the joinpoint
36    -- by locals variables of the advice around
37    var : integer;
38   begin
39       -- Body of the around advice
40        Ada.Text_IO.Put_Line ("I replace the following joinpoint "
41                              & Aspect_Ada.Get_Signature(thisJoinPoint));
42
43       --  storing of the returned value into returned_value
44       --  in order to recover it in the after advice
45       returned_value := AspectAda_Types.Insert_Integer(var);
46       Aspect_Ada.Set_Returned_Value_Access
47                (thisJoinPoint, returned_value'Unchecked_Access);
48
49       -- Call of the procedure After similing the after advice.
50      Short_Aspect.After (thisJoinPoint);
51
52       -- The return of the advice around
53       return var;
```

```
54    end Somme;
55
56    function Mult (A,B : in Integer) return Integer is
57       M : Integer ;
58    begin
59       M := A * B;
60        return M;
61    end Mult;
62
63    procedure Double (A : in out Integer) is
64    begin
65       A := Mult (A,2);
66    end Double;
67
68 end Calcul;
```

A.2.2 Exemple de tissage d'un advice *around* contenant l'instruction *proceed*

L'exemple est constitué de trois grandes parties de code : code métier en Ada, code des aspects en AspectAda et le code tissé généré en Ada.

1. Code métier : nous prenons le même code fonctionnel utilisé dans l'exemple A.1.

2. Code AspectAda : Il contient le code de l'aspect et le code du *weaver* :

• Le code de l'aspect : l'aspect `Second_Aspect` contient un *advice before* et un *advice around*. Ce dernier possède un argument de type `Join_Point` et il retourne un `Integer`. De plus, il fait appel à l'instruction *proceed*. Les listings A.18 et A.19 présente la description de l'aspect (spécification et corps) en AspectAda.

Listing A.18 – Spécification de l'aspect Second_Aspect

```
1  with AspectAda_Types;
2
3  generic
4      Pc : Pointcut;
5  aspect Second_Aspect is
6
7     advice Around (thisJoinPoint : AspectAda_Types.Join_Point) return Integer;
8     for Around'Pointcut use Pc;
9
10    advice Before (thisJoinPoint : AspectAda_Types.Join_Point);
11    for Before'Pointcut use Pc;
12
13 end Second_Aspect;
```

Listing A.19 – Corps de l'aspect Second_Aspect

```
with Ada.Text_IO;
with Aspect_Ada;

aspect body Second_Aspect is

   advice Around (thisJoinPoint : AspectAda_Types.Join_Point) return Integer
   is
     var : Integer;
   begin
      Ada.Text_IO.Put_Line ("I replace the following joinpoint "
                            & Aspect_Ada.Get_Signature(thisJoinPoint));
      var := proceed;
      return var;
   end Around;

   advice Before (thisJoinPoint : AspectAda_Types.Join_Point) is
     Args            : AspectAda_Types.Arg_Array;
     First_Arg       : AspectAda_Types.Arg;
     First_Arg_Value : Integer;
   begin
      Ada.Text_IO.Put_Line ("Entering => "
                         &  Aspect_Ada.Get_Signature(thisJoinPoint));
      Args := Aspect_Ada.Get_Args(thisJoinPoint);
      First_Arg := Args(1);
      Ada.Text_IO.Put_Line ("The name of first arg of the JP is "
                                  & Aspect_Ada.Get_Name(First_Arg));
      First_Arg_Value := AspectAda_Types.Extract_Integer
              (AspectAda_Types.Integer_Holder
                        (Aspect_Ada.Get_Value_Access(First_Arg).all));
      Ada.Text_IO.Put_Line ("The value of first arg of the JP is "
                                             & First_Arg_Value'Img);
   end Before;

end Second_Aspect;
```

- Les règles de composition (*weaver*) : le listing A.20 présente le fichier '`Rules.aaw`'.

Listing A.20 – Code du *weaver*

```
with Second_Aspect;
weaver Rules is
   Calcul_PC : Pointcut := execution(function Calcul.Somme (..) return Integer);
   aspect My_Second_Aspect is new Second_Aspect(Calcul_PC);
end Rules;
```

3. Code source Ada généré : Nous obtenons dans le code généré un ensemble de fichiers :

1. Le paquetage `Second_Aspect` (listings A.21 A.22) c'est le paquetage généré à partir de l'aspect `Second_Aspect` : Il contient seulement la fonction présentant l'*advice before*.

Listing A.21 – Spécification du paquetage Second_Aspect

```
with AspectAda_Types;
package Second_Aspect is
   function Before (thisJoinPoint : AspectAda_Types.Join_Point) return Boolean;
end Second_Aspect;
```

Listing A.22 – Corps du paquetage Second_Aspect

```
1  with Ada.Text_IO;
2  with Aspect_Ada;
3
4  package body Second_Aspect is
5
6     function Before (thisJoinPoint : AspectAda_Types.Join_Point) return Boolean is
7        Args            : AspectAda_Types.Arg_Array;
8        First_Arg       : AspectAda_Types.Arg;
9        First_Arg_Value : Integer;
10    begin
11       Ada.Text_IO.Put_Line ("Entering => "
12                             &  Aspect_Ada.Get_Signature(thisJoinPoint));
13       Args := Aspect_Ada.Get_Args(thisJoinPoint);
14       First_Arg := Args(1);
15       Ada.Text_IO.Put_Line ("The name of first arg of the JP is "
16                                    & Aspect_Ada.Get_Name(First_Arg));
17       First_Arg_Value := AspectAda_Types.Extract_Integer
18               (AspectAda_Types.Integer_Holder
19                        (Aspect_Ada.Get_Value_Access(First_Arg).all));
20       Ada.Text_IO.Put_Line ("The value of first arg of the JP is "
21                                              & First_Arg_Value'Img);
22       return True;
23    end Before;
24
25 end Second_Aspect;
```

2. Le paquetage '`AspectAda_Types`' : le même paquetage généré (les listings A.9 et A.10) dans le premier exemple de cette annexe à la section A.1.

3. Puisque le *pointcut* `Calcul_PC` intercepte l'exécution de la fonctions `Somme` du paquetage `Calcul`. De plus, L'advice around contient une instruction *proceed*, il y aura alors génération d'un nouveau sous-programme. Donc, le paquetage `Calcul` sera recopié, modifié et généré dans le code résultant. Les listings A.23 et A.24 présentent respectivement la spécification et le corps générés du paquetage `Calcul`. (les modifications par rapport au fichier corps du paquetage `Calcul` original sont colorées en bleu) :

Listing A.23 – Spécification du paquetage Calcul généré

```
1  package Calcul is
2
3   -- The name of the function 'Somme'
4   -- is replaced by 'Proceed_Somme' that is
5   -- a unique name in the package 'Calcul'.
6   function Proceed_Somme (A,B : Integer) return Integer;
7
8   -- Declaration of the new subprogram
9   -- having the same signature of the function 'Somme'
10  function Somme  (A,B : in Integer) return Integer;
11
12  -- the rest of declaration of the package
13  function Mult   (A,B : in Integer) return integer;
14  function Double (A : in Integer)   return integer;
```

```
15 end Calcul;
```

Listing A.24 – Corps du paquetage Calcul tissé et généré

```
1  with Ada.Text_IO;
2
3  with Aspect_Ada;
4  with Second_Aspect;
5  with AspectAda_Types;
6
7  package body Calcul is
8
9     function Proceed_Somme (A,B : Integer) return Integer is
10      S : Integer;
11    begin
12       S := A + B;
13       return S;
14    end Proceed_Somme;
15
16    function Somme (A,B : Integer) return Integer is
17
18    -- Creation of Args of the JP
19    Value_Access_Arg1 : aliased AspectAda_Types.Integer_Holder :=
20                                    AspectAda_Types.Insert_Integer(A);
21    Value_Access_Arg2 : aliased AspectAda_Types.Integer_Holder :=
22                                   AspectAda_Types.Insert_Integer(B);
23    Args : AspectAda_Types.Arg_Array := (
24    1 => Aspect_Ada.To_Arg("A", "Integer",
25                           AspectAda_Types.Default_In_Mode,
26                          Value_Access_Arg1'Unchecked_Access),
27   2 => Aspect_Ada.To_Arg("B", "Integer",
28                            AspectAda_Types.Default_In_Mode,
29                            Value_Access_Arg2'Unchecked_Access));
30
31    -- Creation of the joinpoint
32    thisJoinPoint : AspectAda_Types.Join_Point :=
33              Aspect_Ada.Create_Join_Point("Calcul.Somme",
34                                           Args,
35                                          AspectAda_Types.Function_Kind,
36                                           "Integer");
37
38    -- Call of the function Before similing the before advice
39    -- the used variable must have a name different from the names
40    -- of local and global variables and arguments of the joinpoint
41    x : Boolean := Second_Aspect.Before(thisJoinPoint);
42
43    -- Replacing locals variables of the joinpoint
44    -- by locals variables of the advice around
45    var : integer;
46   begin
47       -- Body of the around advice
48        Ada.Text_IO.Put_Line ("I replace the following joinpoint "
49                              & Aspect_Ada.Get_Signature(thisJoinPoint));
50
51       -- The proceed statement is
52       -- replaced by calling the renamed function Proceed_Somme
53       var := Proceed_Somme(A,B);
54       return var;
55   end Somme;
56
57 function Mult (A,B : in Integer) return Integer is
58    M : Integer ;
59 begin
```

```
60    M := A * B;
61    return M;
62 end Mult;
63
64 function Double (A: in Integer) return Integer is
65    d : Integer ;
66 begin
67    d := Mult (A,2);
68    return d;
69 end Double;
70
71 end Calcul;
```

A.3 Exemple de tissage des advices *before* associés à des *joinpoints call*

Dans cette section, nous présentons un exemple permettant d'appliquer les règles de transformation et de tissage d'un *advice before* associé à des *joinpoints call*. Comme pour les autres exemples cités, cet exemple est constitué de trois grandes parties de code : code métier en Ada, code des aspects en AspectAda et le code tissé généré en Ada.

1. Code métier : Il s'agit d'un code source Ada qui se compose de :

• Un paquetage `Do_Hello` contenant deux procédures `Hello_World` et `Good_Bye` et une fonction `G` qui retourne un `Integer`. Les listings A.25 et A.26 présente respectivement la spécification et le corps du paquetage `Do_Hello`.

Listing A.25 – Spécification du paquetage Do_Hello

```
1 package Do_Hello is
2     procedure Hello_World;
3     procedure Good_Bye;
4     function G (A : in Integer) return Integer;
5 end Do_Hello;
```

Listing A.26 – Corps du paquetage Do_Hello

```
1  with Ada.Text_IO;
2  package body Do_Hello is
3
4    procedure Hello_World is
5        A : Integer := G(2);
6    begin
7        for i in 1..A loop
8            Ada.Text_IO.Put_Line("Hello World");
9        end loop;
10   end Hello_World;
```

```
11
12    procedure Good_Bye is
13        A : Integer := G(3);
14    begin
15        for i in 1..A loop
16            Ada.Text_IO.Put_Line("Good Bye");
17        end loop;
18    end Good_Bye;
19
20    function G (A : in Integer) return Integer is
21        Ret : Integer := A * 2;
22    begin
23        Ada.Text_IO.Put_Line("Say hello " & Ret'Img &  " time : ");
24        Return Ret;
25    end G;
26 end Do_Hello;
```

- Une procédure `Hello` qui appelle les procédures `Hello_World` et `Good_Bye`.

Listing A.27 – Procédure Hello

```
1 with Do_Hello;
2 procedure Hello is
3 begin
4       Do_Hello.Hello_World;
5       Do_Hello.Good_Bye;
6 end Hello;
```

2. Code AspectAda : Il contient le code de l'aspect et le code du *weaver* :

- Code de l'aspect : Nous définissons deux exemples d'aspect :
  - Le premier aspect `My_Aspect1` contient un advice before ayant un traitement indépendant des informations des *joinpoints* interceptés. Les listing suivants A.28 et A.29 présentent l'aspect en AspectAda.

Listing A.28 – Spécification de l'aspect My_Aspect1

```
1 generic
2    My_First_Pointcut : Pointcut;
3 aspect My_Aspect1 is
4    advice Before;
5    for Before'pointcut use My_First_Pointcut;
6 end My_Aspect1;
```

Listing A.29 – Corps de l'aspect My_Aspect1

```
1 aspect body My_Aspect1 is
2    advice Before is
3    begin
4      Ada.Text_IO.Put_Line ("Before advice for call JP ");
5    end Before;
6 end My_Aspect1;
```

  - Le deuxième aspect `My_Aspect2` contient un advice before qui accède à certaines

informations des *joinpoints* interceptés.

Listing A.30 – Spécification de l'aspect My_Aspect2

```
1 with AspectAda_Types;
2 generic
3    My_Second_Pointcut : pointcut;
4 aspect My_Aspect2 is
5    Advice Before (thisJoinPoint : AspectAda_Types.Join_Point);
6    for Before'pointcut use My_Second_Pointcut;
7 end My_Aspect2 ;
```

Listing A.31 – Corps de l'aspect My_Aspect2

```
1 with Aspect_Ada;
2 aspect body My_Aspect2 is
3   advice Before (thisJoinPoint : AspectAda_Types.Join_Point) is
4    begin
5      Ada.Text_IO.Put_Line
6       ("Entering => " &  Aspect_Ada.Get_Signature(thisJoinPoint));
7    end Before;
8 end My_Aspect2;
```

• Le code du *weaver* (les règles de composition) : le listing A.6 présente le fichier `hello_rules.aaw` qui permet d'associer les advices des aspects aux *joinpoints* correspondants grâce à la déclaration des pointcuts et l'instanciation des aspects.

Le *weaver* de cet exemple contient la déclaration de deux pointcuts : `PC1` et `PC2`. Le pointcut `PC1` permet d'intercepter tous les appels de toutes les procédures nommées `Hello_World` avec ou sans arguments et appartenant au paquetage `Do_Hello`. De même, le *pointcut* `PC2` permet d'intercepter les appels de toute fonction nommée `G`, appartenant au paquetage `Do_Hello`, ayant des arguments (non spécifiés par le *pointcut*) et retournant un `Integer`. De plus, dans ce weaver nous créons plusieurs instances à partir des deux aspects créés auparavant :

- `My_First_Aspect` est une instance de l'aspect `My_aspect1` qui prend en paramètre le *pointcut* `PC1` ; ce qui permet d'associer l'*advice before* de `My_aspect1` avec les *joinpoints* sélectionnés par `PC1` (ce sont les appels à la procédure `Hello_World`).
- `My_Second_Aspect` est une instance de l'aspect `My_aspect1` qui prend en paramètre le pointcut `PC2` ; ce qui permet d'associer l'advice before de `My_aspect1` avec les *joinpoints* sélectionnés par `PC2` (ce sont les appels à la fonction `G`).
- `My_Third_Aspect` est une instance de l'aspect `My_aspect2` qui prend en paramètre

le pointcut `PC1` ; ce qui permet d'associer l'advice before `My_aspect2` avec les *joinpoints* sélectionnés par `PC1` (ce sont les appels à la procédure `Hello_World`).

Dans cet exemple, l'aspect `My_Aspect1` est plus prioritaire que `My_Aspect2` puisqu'il est instancié le premier. Comme ces deux aspects possèdent un *joinpoint* en commun (la procédure `Hello_World`), alors nous devons tenir compte de ces priorités dans le tissage des deux advices *before*.

Listing A.32 – Code du weaver Hello_Rules

```
1 with My_Aspect;
2 weaver Hello_Rules is
3     PC1 : Pointcut := call(procedure Do_Hello.Hello_World .. );
4     PC2 : Pointcut := call(function Do_Hello.G(..) return Integer);
5
6     aspect My_First_Aspect  is new My_Aspect1(PC1);
7     aspect My_Second_Aspect is new My_Aspect1(PC2);
8     aspect My_Third_Aspect  is new My_Aspect2(PC1);
9 end Hello_Rules;
```

3. Code source Ada généré : après avoir accomplir les traitements nécessaires (interception, compilation, transformation, tissage et génération) nous obtenons dans le code généré un ensemble de fichiers :

1. Le paquetage `My_Aspect1` c'est le paquetage généré à partir de l'aspect `My_Aspect1` :

Listing A.33 – Spécification du paquetage My_Aspect1

```
1 package My_Aspect1 is
2   function Before return Boolean;
3 end My_Aspect1;
```

Listing A.34 – Corps du paquetage My_Aspect1

```
1 package body My_Aspect1 is
2   function Before return Boolean is
3   begin
4       Ada.Text_IO.Put_Line ("Before advice for call JP ");
5       Return true;
6   end Before;
7 end My_Aspect1;
```

2. Le paquetage `My_Aspect2` c'est le paquetage généré à partir de l'aspect `My_Aspect2` :

Listing A.35 – Specification du paquetage My_Aspect2

```
1 with AspectAda_Types;
2 package My_Aspect2 is
```

```
3     function Before (thisJoinPoint : AspectAda_Types.Join_Point) return Boolean;
4  end My_Aspect2;
```

Listing A.36 – Corps du paquetage My_Aspect2

```
1  with Aspect_Ada;
2  package body My_Aspect2 is
3     function Before (thisJoinPoint : AspectAda_Types.Join_Point) return Boolean is
4     begin
5        Ada.Text_IO.Put_Line
6         ("Entering => " &  Aspect_Ada.Get_Signature(thisJoinPoint));
7         Return true;
8     end Before;
9  end My_Aspect2;
```

3. Un paquetage supplémentaire généré associé au paquetage `Do_Hello` nommé `Tmp_Do_Hello`. La génération de ce paquetage supplémentaire est dûe à l'interception des appels aux sous programmes `Hello_World` et G du paquetage `Do_Hello`. Il contient alors une procédure et une fonction ayant respectivement les mêmes signatures que la procédure `Hello_World` et la fonction `G` du paquetage `Do_Hello`. Sa spécification est donnée par le listing A.37.

Listing A.37 – Specification du paquetage Tmp_Do_Hello

```
1  package Tmp_Do_Hello is
2        procedure Hello_World ;
3        function G (A : in Integer) return Integer ;
4  end Tmp_Do_Hello;
```

Son implantation est présentée par le listing A.38. La procédure `Hello_World` est interceptée par l'advice *before* de l'aspect `My_aspect2`. Or, cet advice *before* prend en entrée un paramètre `thisJoinPoint` de type `Join_Point`. C'est-à-dire il accède à des informations des *joinpoints* interceptés. Le paramètre thisJoinPoint doit être créé dans la procédure `Hello_World` à travers la fonction `Create_Join_Point` de la `Runtime`. Après sa création, le paramètre thisJoinPoint sera passé à la fonction présentant l'advice Before. Par contre, la fonction `G` est interceptée seulement par l'advice before de l'aspect `My_aspect1`. Cet advice est indépendant des *joinpoints* qu'ils lui sont associés. Pour cela, il n'y aura pas de création du *joinpoint* dans la fonction `G` du paquetage supplémentaire.

Listing A.38 – Corps du paquetage Tmp_Do_Hello

```
1  with Do_Hello;
2  with My_Aspect1;
3  with My_Aspect2;
4
5  package body Tmp_Do_Hello is
6
7  procedure Hello_World is
8     Args : AspectAda_Types.Arg_Array :=  AspectAda_Types.Nil_Arg_Array;
9     thisJoinPoint : AspectAda_Types.Join_Point :=
10                Aspect_Ada.Create_Join_Point ("Do_Hello.Hello_World",
11                                              Args,
12                                              AspectAda_Types.Procedure_Kind);
13    X  : Boolean := My_Aspect1.Before;
14    X1 : Boolean := My_Aspect2.Before(thisJoinPoint);
15 begin
16    Do_Hello.Hello_World;
17 end Hello_World;
18
19 function G (A : in Integer) return integer is
20    X : Boolean := My_Aspect1.Before;
21 begin
22    Do_hello.G(A);
23 end G;
24 end Tmp_Do_Hello;
```

4. Enfin, nous trouvons dans le code généré des fichiers du code de l'application qui contiennent des appels aux *joinpoints* interceptés. Ils seront recopiés et modifiés (les modifications par rapport aux fichiers originaux sont colorées en bleu) :

• Le corps du paquetage `Do_Hello` sera recopié puisqu'il contient des appels à la fonction `G`. Dans le fichier généré ces appels seront remplacés par des appels à la fonction `G` du paquetage supplémentaire `Tmp_Do_Hello`.

Listing A.39 – Corps du paquetage Do_Hello généré

```
1  with Ada.Text_IO;
2  with Tmp_Do_Hello;
3
4  package body Do_Hello is
5
6    procedure Hello_World is
7        a : Integer := Tmp_Do_Hello.G(2);
8    begin
9        for i in 1..A loop
10           Ada.Text_IO.Put_Line("Hello World");
11       end loop;
12   end Hello_World;
13
14   procedure Good_Bye is
15       A : Integer := Tmp_Do_Hello.G(3);
16   begin
17       for i in 1..A loop
18           Ada.Text_IO.Put_Line("Good Bye");
19       end loop;
20   end Hello_World;
```

```
21
22    function G (A : in Integer) return integer is
23        Ret : Integer := A * 2;
24    begin
25          Ada.Text_IO.Put_Line("Say hello " & Ret'Img & " time : ");
26          Return Ret;
27    end G;
28 end Do_Hello;
```

• La procédure Hello puisqu'elle appelle la procédure Hello_World :

Listing A.40 – Corps du paquetage Do_Hello généré

```
1 with Do_Hello;
2 with Tmp_Do_Hello;
3
4 procedure Hello is
5 begin
6    Tmp_Do_Hello.Hello_World;
7    Do_Hello.Good_Bye;
8 end Hello;
```

A.4 Exemple de tissage des advices *after* associés à des *joinpoints call*

Dans cette section, nous présentons un exemple permettant d'appliquer les règles de transformation et de tissage d'un *advice after* associé à des points de jonction *call*. Comme pour les autres exemples cités, cet exemple est constitué de trois grandes parties de code : code métier en Ada, code des aspects en AspectAda et le code tissé généré en Ada.

1. Code métier : nous prenons le même code fonctionnel présenté dans l'exemple d'application des règles de tissage des *advices before* des points de jonction *call* A.3.

2. Code AspectAda : Il contient le code de l'aspect et le code du *weaver* :

• Code de l'aspect : Nous définissons deux exemples d'aspect :

– Le premier aspect `My_Aspect1` contient un advice after ayant un traitement indépendant des informations des *joinpoints* interceptés. Les listings A.41 et A.42 suivants présentent une description de l'aspect en AspectAda.

Listing A.41 – Spécification de l'aspect My_Aspect1

```
1 generic
2    My_First_Pointcut : pointcut;
```

```
3  aspect My_Aspect1 is
4     advice After;
5     for After'pointcut use My_First_Pointcut;
6  end My_Aspect1 ;
```

Listing A.42 – Corps de l'aspect My_Aspect1

```
1  aspect body My_Aspect1 is
2     advice After  is
3     begin
4       Ada.Text_IO.Put_Line ("After advice for call JP ");
5     end After;
6  end My_Aspect1;
```

– Le deuxième aspect `My_Aspect2` contient un advice after qui accède à certaines informations des points de jonction interceptés.

Listing A.43 – Spécification de l'aspect My_Aspect2

```
1  with AspectAda_Types;
2  generic
3     My_Second_Pointcut : pointcut;
4  aspect My_Aspect2 is
5     Advice After (thisJoinPoint : AspectAda_Types.Join_Point);
6     for After'pointcut use My_Second_Pointcut;
7  end My_Aspect2 ;
```

Listing A.44 – Corps de l'aspect My_Aspect2

```
1  with Aspect_Ada;
2  aspect body My_Aspect2 is
3    advice After (thisJoinPoint : AspectAda_Types.Join_Point) is
4     begin
5       Ada.Text_IO.Put_Line
6        ("Exiting from => " &
7             Aspect_Ada.Get_Signature(thisJoinPoint));
8     end After;
9  end My_Aspect2;
```

• Code du *Weaver* `hello_rules.aaw` : nous prenons le même code du weaver présenté dans l'exemple d'application des règles de tissage des *advices before* pour un point de jonction *call* A.3.

3. Code source Ada généré : après avoir accomplir les traitements nécessaires (interception, compilation, transformation, tissage et génération) nous obtenons dans le code généré un ensemble de fichiers :

1. Le paquetage `My_Aspect1` c'est le paquetage généré à partir de l'aspect `My_Aspect1`

Listing A.45 – Spécification du paquetage My_Aspect1

```
1 package My_Aspect1 is
2     procedure After;
3 end My_Aspect1;
```

Listing A.46 – Corps du paquetage My_Aspect1

```
1 package body My_Aspect1 is
2    procedure After is
3   begin
4       Ada.Text_IO.Put_Line ("After advice for call JP ");
5   end After;
6 end My_Aspect1;
```

2. Le paquetage `My_Aspect2` c'est le paquetage généré à partir de l'aspect `My_Aspect2` :

Listing A.47 – Spécification du paquetage My_Aspect2

```
1 with AspectAda_Types;
2 package My_Aspect2 is
3    procedure After (thisJoinPoint : AspectAda_Types.Join_Point) ;
4 end My_Aspect2;
```

Listing A.48 – Corps du paquetage My_Aspect2

```
1 with Aspect_Ada;
2 package body My_Aspect2 is
3    procedure After (thisJoinPoint : AspectAda_Types.Join_Point)    is
4     begin
5      Ada.Text_IO.Put_Line
6       (" Exiting from => " &
7           Aspect_Ada.Get_Signature(thisJoinPoint));
8     end After;
9 end My_Aspect2;
```

3. Un paquetage supplémentaire généré `Tmp_Do_Hello` (listings A.49 et A.50) qui contient une procédure et une fonction ayant respectivement les mêmes signatures que la procédure `Hello_World` et la fonction `G` du paquetage `Do_Hello`. Le tissage des advices se fait au niveau de ces sous-programmes.

Listing A.49 – Spécification du paquetage Tmp_Do_Hello

```
1 package Tmp_Do_Hello is
2     procedure Hello_World;
3     function G (A : in Integer) return Integer ;
4 end Tmp_Do_Hello;
```

Listing A.50 – Corps du paquetage Tmp_Do_Hello

```
1  with Do_Hello;
2  with My_Aspect1;
3  with My_Aspect2;
4  with Aspect_Ada;
5  with AspectAda_Types;
6
7  package body Tmp_Do_Hello is
8
9  procedure Hello_World is
10    Args : AspectAda_Types.Arg_Array :=  AspectAda_Types.Nil_Arg_Array;
11    thisJoinPoint : AspectAda_Types.Join_Point :=
12        Aspect_Ada.Create_Join_Point ("Do_Hello.Hello_World",
13                                      Args,
14                                      AspectAda_Types.Procedure_Kind);
15 Begin
16     -- Appel du procedure intercepté
17     Do_Hello.Hello_World;
18     -- Appel de l'advice after de l'aspect le plus prioritaire
19     My_Aspect1.After;
20     -- Appel de l'advice after de l'aspect le moins prioritaire
21     My_Aspect2.After (thisJoinPoint);
22 end Hello_World;
23
24 function G (A : in Integer) return Integer is
25    -- Déclaration d'une nouvelle variable
26    -- pour stocker la valeur retourné par
27    -- la fonction interceptée G
28     v : Integer;
29 begin
30    -- appel de la fonction interceptée
31     v := Do_hello.G(A);
32    -- appel de l'advice after de l'aspect My_Aspect1
33     My_Aspect1.After;
34    -- Retour de la valeur stockée dans la variable v
35     return v;
36 end G;
37
38 end Tmp_Do_Hello;
```

4. Enfin, nous trouvons dans le code généré des fichiers du code de l'application qui contiennent des appels aux *joinpoints* interceptés. Ils seront recopiés et modifiés (les modifications par rapport aux fichiers originaux sont colorées en bleu) :

• Le corps du paquetage `Do_Hello` sera recopié puisqu'il contient des appels à la fonction `G`. Dans le fichier généré ces appels seront remplacés par des appels à la fonction G du paquetage supplémentaire `Tmp_Do_Hello`.

Listing A.51 – Corps du paquetage Do_Hello généré

```
1  with Ada.Text_IO;
2  with Tmp_Do_Hello;
3
4  package body Do_Hello is
5
6    procedure Hello_World is
```

```
7         A : Integer := Tmp_Do_Hello.G(2);
8     begin
9         for i in 1..A loop
10            Ada.Text_IO.Put_Line("Hello World");
11        end loop;
12    end Hello_World;
13
14    procedure Good_Bye is
15        A : Integer := Tmp_Do_Hello.G(3);
16    begin
17        for i in 1..A loop
18            Ada.Text_IO.Put_Line("Good Bye");
19        end loop;
20    end Hello_World;
21
22    function G (A : in Integer) return Integer is
23        Ret : Integer := A * 2;
24    begin
25         Ada.Text_IO.Put_Line("Say hello " & Ret'Img & " time : ");
26         Return ret;
27    end G;
28 end Do_Hello;
```

- La procédure `Hello` puisqu'elle appelle la procédure `Hello_World` :

Listing A.52 – Corps du paquetage Do_Hello généré

```
1 with Do_Hello;
2 with Tmp_Do_Hello;
3
4 procedure Hello is
5 begin
6     Tmp_Do_Hello.Hello_World;
7     Do_Hello.Good_Bye;
8 end Hello;
```

A.5 Exemples de tissage des advices *around* associés à des *joinpoints call*

Pour mettre en évidence la manière de tissage d'un *advice around* associé à des *joinpoints call* et pour mieux comprendre le traitement de l'instruction *proceed*, nous allons montrer un exemple simple et explicatif. L'exemple est constitué de trois grandes parties de code : code métier en Ada, code des aspects en AspectAda et le code tissé généré en Ada.

1. Code métier : nous prenons le même code fonctionnel utilisé dans l'exemple A.1.

2. Code AspectAda : Il contient le code de l'aspect et le code du *weaver* :

• Le code de l'aspect : dans le but de tracer les intéractions entre les sous-programmes d'une application, nous définissons l'aspect `Logger_Aspect`. Dans le fichier de la spécification de l'aspect, `Logger_Aspect` déclare :

- Deux variables globales : `Log_Count` afin de garder une trace de l'ordre des appels aux sous-programmes, `Exec_Depth` afin de garder une trace de niveaux d'imbrication des appels ;
- Un *advice around* prend un paramètre de type `Join_Point` afin de pouvoir accéder aux informations des *joinpoints* interceptés par le *pointcut* associé. Il retourne un `Integer`.

Le listing A.53 présente la spécification de l'aspect.

Listing A.53 – Spécification de l'aspect Logger_Aspect

```
1  with AspectAda_Types;
2
3  generic
4      First_Pointcut : pointcut;
5  aspect Logger_Aspect is
6
7     Log_Count  : Integer := 0;
8     Exec_Depth : Integer := 0;
9
10    advice Around (thisJoinPoint : AspectAda_Types.Join_Point) return Integer;
11    for Around'pointcut use First_Pointcut;
12
13 end Logger_Aspect;
```

Le fichier corps `Logger_Aspect.aab` (listing A.54) fournit l'implantation pour l'*advice around* de la manière suivante : Avant de procéder à l'exécution du *joinpoint* intercepté, les valeurs `Log_Count` et `Exec_Depth` sont incrémentées. En outre, les informations de journalisation sont affichées sur l'écran. Par la suite, le flux de contrôle retourne au *joinpoint* intercepté et la fonction est appelée. Afin de former une hiérarchie des appels imbriqués aux fonctions de l'application, après avoir procéder à l'appel et l'exécution et de chaque fonction interceptée, la valeur de `Exec_Depth` devrait être décrémentée. Enfin, l'*advice* retourne la valeur retournée par la fonction interceptée. Il est à noter que `Log_Count` et `Exec_Depth` ne sont pas de type '*thread safe*' [1]. Si le développeur veut les rendre '*thread*

1. Un type thread-safe est un type qui fonctionne correctement lorsqu'il est exécuté par plusieurs threads.

safe', il doit utiliser les mécanismes de concurrence comme les objets protégés etc.

Listing A.54 – Corps de l'aspect Logger_Aspect

```
with Ada.Text_IO;
with Aspect_Ada;

aspect body Logger_Aspect is
  advice Around (thisJoinPoint : AspectAda_Types.Join_Point) return Integer
  is
      Var : Integer ;
  begin
      Log_count := Log_count + 1;
      Exec_Depth := Exec_Depth + 1;

      for i in 0 .. Exec_Depth loop
          Ada.Text_IO.Put_Line (" ");
      end loop;
      Ada.Text_IO.Put_Line
         (Aspect_Ada.Get_Signature(thisJoinPoint) &
           "; log_count = " & Log_Count'Img &
           "; Exec_Depth = " & Exec_Depth'Img );
      Var := proceed;
      Exec_Depth := Exec_Depth - 1;
      return Var;
   end Around;
end Logger_Aspect;
```

• Code du *weaver* `Logger_rules.aaw` (listing A.55) : `Logger_Aspect` devrait être capable d'intercepter l'appel de chaque sous-programme de l'application (code métier). L'application de cet exemple possède un seul paquetage `Calcul` définissant seulement deux fonctions et une procédure. Le modèle de *joinpoints* correspondant est réalisé grâce à la définition de deux *pointcuts*. L'association entre l'*advice around* et les *joinpoints* interceptés se fait grâce à l'instanciation de l'aspect.

Listing A.55 – Règles de composition (*weaver*) : Logger_Rules

```
with Logger_Aspect;
weaver Logger_Rules is
  Calcul_PC_func : Pointcut := call(function  *.* (..) return * );
  Calcul_PC_proc : Pointcut := call(procedure *.* (..) );
  aspect My_First_Aspect is new Logger_Aspect(Calcul_PC_func);
  aspect My_second_Aspect is new Logger_Aspect(Calcul_PC_proc);
end Logger_Rules;
```

3. Code source Ada généré : après avoir accompli les traitements nécessaires (interception, compilation, transformation, tissage et génération) nous obtenons dans le code généré un ensemble de fichiers :

1. Le paquetage `Logger_Aspect` c'est le paquetage généré à partir de l'aspect. Ce dernier contient des déclarations de variables globales et un *advice around*. Et comme un *advice*

around ne se transforme pas en un sous-programme alors le paquetage généré contient seulement les déclarations des variables globales.

Listing A.56 – Spécification du paquetage Logger_Aspect généré à partir de l'aspect

```
1 package Logger_Aspect is
2
3    Log_Count  : Integer := 0;
4    Exec_Depth : Integer := 0;
5
6 end Logger_Aspect;
```

Listing A.57 – Corps du paquetage Logger_Aspect généré à partir de l'aspect

```
1 Package body Logger_aspect is
2   -- Empty Body
3 end Logger_Aspect;
```

2. Le paquetage '`AspectAda_Types`' : le même paquetage généré (les listings A.9 et A.10) dans le premier exemple de cette annexe à la section A.1.

3. Un paquetage supplémentaire est généré nommé `Tmp_Calcul`. En effet, les règles de composition de cet exemple permet d'intercepter les appels de tous les sous-programmes du paquetage `Calcul`. Pour cela, pour chacune de ses fonctions, une nouvelle fonction sera définie dans le paquetage `Tmp_Calcul`.

Le tissage de l'*advice around* se fait au niveau de ces fonctions du paquetage supplémentaire. Dans le listing A.58 nous proposons un exemple d'implantation du paquetage généré `Tmp_Calcul`.

Listing A.58 – Spécification du paquetage Tmp_Calcul

```
1 package tmp_Calcul is
2    function Somme  (A,B : in Integer) return Integer ;
3    function Mult   (A,B : in Integer) return Integer ;
4    procedure Double (A : in out Integer) ;
5 end tmp_Calcul;
```

Listing A.59 – Corps du paquetage Tmp_Calcul

```
1 with Ada.Text_IO;
2
3 with Calcul;
4 with Logger_Aspect;
5 with Aspect_Ada;
6 with AspectAda_Types;
7
8 package body Tmp_Calcul is
9    function Somme (A,B : Integer) return Integer is
```

```
10      -- Creation of arguments of the joinpoint
11      Value_Access_Arg1 : aliased AspectAda_Types.Integer_Holder :=
12                  AspectAda_Types.Insert_Integer(A);
13      Value_Access_Arg2 : aliased AspectAda_Types.Integer_Holder :=
14                  AspectAda_Types.Insert_Integer(B);
15      Args : AspectAda_Types.Arg_Array := (
16          1 => Aspect_Ada.To_Arg ("A", "Integer", AspectAda_Types.IN_Mode,
17                                  Value_Access_Arg1'Unchecked_Access),
18          2 => Aspect_Ada.To_Arg ("B", "Integer", AspectAda_Types.In_Mode,
19                                  Value_Access_Arg2'Unchecked_Access));
20
21      -- Creation of the joinpoint
22      thisJoinPoint : AspectAda_Types.Join_Point :=  Aspect_Ada.Create_Join_Point
23                  ("Calcul.Somme", Args, AspectAda_Types.Function_Kind, "Integer");
24
25      -- declaration of local variables of the around advice
26      Var : Integer ;
27
28      -- Declaring a variable with the same return type
29      -- of the advice around. This variable will be used
30      -- for storing the return value.
31      returned_Value : Integer ;
32   begin
33
34      -- The body of the advice around
35      Logger_aspect.Log_count := Logger_aspect.Log_count + 1;
36      Logger_aspect.Exec_Depth := Logger_aspect.Exec_Depth + 1;
37
38      for i in 0 .. Logger_aspect.Exec_Depth loop
39          Ada.Text_IO.Put ("    ");
40      end loop;
41
42      Ada.Text_IO.Put_Line (Aspect_Ada.Get_Signature(thisJoinPoint)
43                          & "; log_count = " & Logger_aspect.Log_Count'Img
44                          & "; Exec_Depth = " & Logger_aspect.Exec_Depth'Img);
45
46      -- The proceed statement is replaced by the call of the joinpoint
47      -- that is the function 'Somme' of the package 'Calcul'.
48      -- The proceed statement in the around advice is used without
49      -- parameters, although the function 'Somme' must have parameters
50      -- then, it will be called with the same arguments used
51      -- during the intercepted call in the business code.
52      Var := Calcul.Somme (A,B);
53
54      Logger_aspect.Exec_Depth := Logger_aspect.Exec_Depth - 1;
55
56      -- The end of the advice around.
57      -- His return statement is removed, it will be replaced
58
59      -- Replacement of the return instruction by storing
60      -- the return value of around advice in the new declared
61      -- variable 'returned_value'.
62      returned_Value := Var;
63
64      -- There isn't any advice After for the function Somme.
65      -- In this case, there is no weaving of advices after
66      -- and the return statement will be inserted directly
67      -- after the weaving of the advice around.
68      return returned_Value;
69   end Somme;
70
71   function Mult (A,B : in Integer) return integer is
72      Value_Access_Arg1 : aliased AspectAda_Types.Integer_Holder :=
73                  AspectAda_Types.Insert_Integer(A);
74      Value_Access_Arg2 : aliased AspectAda_Types.Integer_Holder :=
```

```
75                  AspectAda_Types.Insert_Integer(B);
76       Args : Aspect_Ada.Arg_Array := (
77           1 => Aspect_Ada.To_Arg ("A", "Integer", AspectAda_Types.IN_Mode,
78                                   Value_Access_Arg1'Unchecked_Access),
79           2 => Aspect_Ada.To_Arg ("B", "Integer", AspectAda_Types.In_Mode,
80                                   Value_Access_Arg2'Unchecked_Access));
81
82       -- Creation of the joinpoint
83       thisJoinPoint : Aspect_Ada.Join_Point :=  AspectAda_Types.Create_Join_Point
84                     ("Calcul.Mult", Args, AspectAda_Types.Function_Kind, "Integer");
85
86       Var : Integer ;
87       returned_Value : Integer ;
88    begin
89       Logger_aspect.Log_count := Logger_aspect.Log_count + 1;
90       Logger_aspect.Exec_Depth := Logger_aspect.Exec_Depth + 1;
91
92       for i in 0 .. Logger_aspect.Exec_Depth loop
93           Ada.Text_IO.Put("    ");
94       end loop;
95
96       Ada.Text_IO.Put_Line (AspectAda_Types.Get_Signature(thisJoinPoint)
97                             & "; log_count = " & Logger_aspect.Log_Count'Img
98                             & "; Exec_Depth = " & Logger_aspect.Exec_Depth'Img);
99
100      Var := Calcul.Mult (A,B);
101      returned_Value := Var;
102      return returned_Value;
103   end Mult;
104
105   procedure Double (A: in out Integer) is
106      Value_Access_Arg1 : aliased AspectAda_Types.Integer_Holder :=
107                  AspectAda_Types.Insert_Integer(A);
108      Args : AspectAda_Types.Arg_Array := (
109           1 => AspectAda_Types.To_Arg("A", "Integer", AspectAda_Types.In_Out_Mode,
110                                       Value_Access_Arg1'Unchecked_Access),
111           2 => AspectAda_Types.Nil_Arg);
112      thisJoinPoint : AspectAda_Types.Join_Point := Aspect_Ada.Create_Join_Point
113                    ("Calcul.Double", Args, AspectAda_Types.Procedure_Kind);
114
115   begin
116      Logger_aspect.Log_count := Logger_aspect.Log_count + 1;
117      Logger_aspect.Exec_Depth := Logger_aspect.Exec_Depth + 1;
118
119      for i in 0 .. Logger_aspect.Exec_Depth loop
120          Ada.Text_IO.Put ("    ");
121      end loop;
122
123      Ada.Text_IO.Put_Line (Aspect_Ada.Get_Signature(thisJoinPoint)
124                            & "; log_count = " & Logger_aspect.Log_Count'Img
125                            & "; Exec_Depth = " & Logger_aspect.Exec_Depth'Img);
126
127      Calcul.Double (A);
128      Logger_aspect.Exec_Depth := Logger_aspect.Exec_Depth - 1;
129
130   end Double;
131
132 end Tmp_Calcul;
```

4. Enfin, nous trouvons dans le code généré des fichiers du code de l'application qui contiennent des appels aux *joinpoints* interceptés. Ils seront recopiés et modifiés (les mod-

ifications par rapport aux fichiers originaux sont colorées en bleu) :

• Le corps du paquetage `Calcul` sera recopié puisque un appel à la fonction `Mult` est intercepté au sein de la procedure `Double`. Dans le fichier généré cet appel sera remplacé par un appel à la fonction `Mult` du paquetage `Tmp_Calcul`.

Listing A.60 – Corps du paquetage Calcul généré

```
1  with Ada.Text_IO;
2  with Tmp_Calcul;
3
4  package body Calcul is
5  function Somme (A,B : in Integer) return Integer is
6    S : Integer;
7  begin
8    S := A + B;
9    return S;
10 end Somme;
11
12 function Mult (A,B : in Integer) return Integer is
13   M : Integer ;
14 begin
15   M := A * B;
16   return M;
17 end Mult;
18
19 procedure Double (A: in out Integer) is
20 begin
21     -- Each call of a selected joinpoint must be replaced
22     -- by calling of the corresponding generated function
23     A := Tmp_Calcul.Mult (A,2);
24 end Double;
25
26 end Calcul;
```

• La procédure main car elle contient des appels aux différentes fonctions du paquetage `Calcul`. Ces appels devront être remplacés par des appels aux fonctions convenables du paquetage `Tmp_Calcul` :

Listing A.61 – Procédure main généré

```
1  with Ada.Text_IO;
2  with Calcul;
3  with Tmp_Calcul;
4
5  procedure main is
6     mul : Integer := Tmp_Calcul.Mult(2,Tmp_Calcul.Somme (3,4));
7     double : Integer := Tmp_Calcul.Double(20);
8  begin
9     Ada.Text_IO.Put_Line(" I'm the main procedure, I call Calcul.Somme "
10                         & Integer'Image(Tmp_Calcul.Somme (3,4)));
11 end main;
```

Annexe

B Code AspectAda Implantant la Traçabilité de l'Étude de Cas

Dans cette annexe nous présentons le code AspectAda implantant la traçabilité de l'application `Workload Manager` décrite dans le chapitre 5. Le code AspectAda implanté est composé de deux parties : code de l'aspect et code du weaver.

B.1 Code de l'aspect : `Logger_Aspect`

Listing B.1 – Spécification de l'aspect `Logger_Aspect` implanté pour `Workload_Manager`

```
1  with AspectAda_Types;
2  with Activation_Log;
3
4  generic
5     Op_Pointcut    : pointcut;
6     Signal_PC      : pointcut;
7     Start_PC       : pointcut;
8     Log_Reader_PC  : pointcut;
9  aspect Logger_Aspect is
10
11    Old_Activation_Counter : Activation_Log.Range_Counter := 0;
12
13    procedure Display_Time;
14
15    advice Before_Operation (thisJoinPoint : AspectAda_Types.Join_Point);
16    for Before_Operation'pointcut use Op_Pointcut;
17
18    advice After_Operation (thisJoinPoint : AspectAda_Types.Join_Point);
19    for After_Operation'pointcut use Op_Pointcut;
20
21    advice Before_Signal;
22    for Before_Signal'pointcut use Signal_PC;
23
24    advice After_Start (thisJoinPoint : AspectAda_Types.Join_Point);
25    for After_Start'pointcut use Start_PC;
26
27    advice Around;
28    for Around'pointcut use Log_Reader_PC;
29
30 end Logger_Aspect;
```

Listing B.2 – Corps de l'aspect `Logger_Aspect` implanté pour `Workload_Manager`

```
with System.IO;
with Ada.Real_Time;
with Activation_Manager;
with Aspect_Ada;

aspect body Logger_Aspect is

   procedure Display_Time is
     use Ada.Real_Time;
   begin
     System.IO.Put( "["
                    & Duration'Image
                    (To_Duration
                     (Clock - Activation_Manager.System_Start_Time))
                    & "] ");
   end Display_Time;

   advice Before_Operation (thisJoinPoint : AspectAda_Types.Join_Point) is
   begin
      Display_Time;
      if Aspect_Ada.Get_Name(thisJoinPoint) =
                    "External_Event_Server_Parameters.Server_Operation"  then
         System.IO.Put_Line ("External Event Server: received an external interrupt");
      elsif Aspect_Ada.Get_Name(thisJoinPoint) =
                    "On_Call_Producer_Parameters.On_Call_Producer_Operation" then
         System.IO.Put_Line ("On Call Producer: doing some work.");
      elsif Aspect_Ada.Get_Name(thisJoinPoint) =
                    "Regular_Producer_Parameters.Regular_Producer_Operation" then
         System.IO.Put_Line ("Regular Producer: doing some work.");
      end if;
   end Before_Operation;

   advice After_Operation (thisJoinPoint : AspectAda_Types.Join_Point) is
   begin
      Display_Time;
      if Aspect_Ada.Get_Name(thisJoinPoint) =
                    "External_Event_Server_Parameters.Server_Operation"  then
         System.IO.Put_Line ("External Event Server: end of sporadic activation.");
      elsif Aspect_Ada.Get_Name(thisJoinPoint) =
                    "Regular_Producer_Parameters.Regular_Producer_Operation" then
         System.IO.Put_Line ("Regular Producer: end of cyclic activation");
      elsif Aspect_Ada.Get_Name(thisJoinPoint) =
                    "On_Call_Producer_Parameters.On_Call_Producer_Operation" then
         System.IO.Put_Line ("On Call Producer: end of sporadic activation.");
      end if;
   end After_Operation;

   advice Before_Signal is
   begin
     Display_Time;
     Ada.Text_IO.Put_Line ("Signaling 'Activation Log Reader'");
   end Before_Signal;

   advice After_Start (thisJoinPoint : AspectAda_Types.Join_Point) is
     Returned_Value           : Boolean;
     Args                     : AspectAda_Types.Arg_Array;
     Activation_Parameter_Arg : AspectAda_Types.Arg;
     Activation_Parameter     : Positive;
   begin
     Returned_Value := AspectAda_Types.Extract_Boolean
                 (AspectAda_Types.Boolean_Holder(
                             Aspect_Ada.Get_Returned_Value_Access(thisJoinPoint).all));
```

```
63       Args := Aspect_Ada.Get_Args(thisJoinPoint);
64       Activation_Parameter_Arg := Args(1);
65       Activation_Parameter := AspectAda_Types.Extract_Positive
66                 (AspectAda_Types.Positive_Holder(
67                              Aspect_Ada.Get_Value_Access(Activation_Parameter_Arg).all));
68       Display_Time;
69       if Returned_Value then
70          Ada.Text_IO.Put_Line ("Sending extra work to 'On_Call_Producer': "
71                        & Activation_Parameter'Img);
72       else
73         Ada.Text_IO.Put_Line ("Failed sporadic activation.");
74       end if;
75    end After_Start;
76
77    advice Around is
78      use Ada.Real_Time;
79    begin
80      Display_Time;
81      Ada.Text_IO.Put_Line ("Activation Log Reader: do some work.");
82      Proceed;
83      Display_Time;
84      if Interrupt_Arrival_Counter /= Old_Activation_Counter then
85         Ada.Text_IO.Put_Line ("Read external new interruption:"
86                        & Activation_Log.Range_Counter'Image (Interrupt_Arrival_Counter)
87                        & ". Arrived at ["
88                        & Duration'Image
89                        (To_Duration
90                         (Interrupt_Arrival_Time - Activation_Manager.System_Start_Time))
91                        & "]");
92         Old_Activation_Counter := Interrupt_Arrival_Counter;
93      else
94        Ada.Text_IO.Put_Line ("Activation Log Reader: no new interrupts.");
95      end if;
96      --And finally we report nominal completion of current
97      --activation.
98      Display_Time;
99      Ada.Text_IO.Put_Line ("Activation Log Reader: end of parameterless sporadic"
100                       & " activation.");
101   end Around;
102 end Logger_Aspect;
```

B.2 Code Weaver : `Workload_Rules`

Listing B.3 – Code du Weaver `Workload_Rules`

```
1  with Logger_Aspect;
2
3  weaver Workload_Rules is
4
5   Operation_PC : pointcut := execution (procedure *_Parameters.Regular_Producer_Operation)
6                                          or
7                              execution (procedure *_Parameters.Server_Operation)
8                                          or
9                              execution (procedure *_Parameters.On_Call_Producer_Operation);
10
11  Signal_PC    : pointcut := call (procedure Activation_Log_Reader.Signal);
12
13  Start_PC     : pointcut := call (function On_Call_Producer.Start (..) return Boolean);
14
```

```
15    Log_Reader_PC: pointcut := execution
16            (procedure Activation_Log_Reader_Parameters.Activation_Log_Reader_Operation(..));
17
18    aspect My_Logger_Aspect is new Logger_Aspect(Operation_PC,
19                                                 Signal_PC,
20                                                 Start_PC,
21                                                 Log_Reader_PC);
22
23  end Workload_Rules;
```

République Tunisienne
Ministère de l'Enseignement Supérieur
et de la Recherche Scientifique

Université de Sfax
École Nationale d'Ingénieurs de Sfax

Ecole Doctorale
Sciences et Technologies

Mémoire de *MASTERE*
NTSID
N° d'ordre: 240-12

Extension et Adaptation d'un Langage d'Aspect pour les Systèmes Temps-Réel

Rahma BOUAZIZ

الخلاصة:
تندرج هذه الدراسة في إطار إدماج البرمجة جانبيّة المنحى في تطوير برمجة الأنظمة الآنيّة. تعتمد هذه البرمجة جانبيّة المنحى على نسج الجوانب الّتي تمّ إنشاؤها على حدة في تطبيق النّظام. و لكن عمليّة النّسيج هذه بإمكانها الإخلال بحتميّة النّظام من خلال إدخال بعض التّركيبات الّتي قد تخرق القيود الخاضعة لها الأنظمة الآنيّة والّتي قد تخرج عن إرادة المبرمج. و بالتّالي، الهدف من هذا العمل هو دراسة آليّة جانبيّة موجودة (لغة AspectAda) من أجل تكييفها و تعديلها للامتثال لقيود الأنظمة الآنيّة.

Résumé
Ce mémoire s'inscrit dans le cadre de l'intégration du paradigme orienté aspect (POA) dans le développement des systèmes temps-réel. La programmation orientée aspect est fondée sur l'opération de tissage permettant d'intégrer automatiquement les bouts de code créés séparément dans le code d'une application. Dans le contexte des systèmes temps-réel, cette opération peut compromettre le déterminisme en insérant dans le code de l'application des constructions pouvant violer les contraintes temps-réel et échappant à tout contrôle de la part du développeur. L'objectif de ce travail est l'étude d'un mécanisme d'aspect existant (le langage AspectAda) pour permettre de l'adapter et le modifier afin de garantir les contraintes temps réel.

Abstract:
This report is part of the integration of aspect-oriented paradigm (AOP) in the development of real-time systems. Aspect-oriented programming is based on the weaving process to integrate automatically segments of code created separately into the application code. In the context of real-time systems, this can endanger determinism by including in the application code constructs that may violate real-time constraints and beyond control of the developer. The objective of this work is the study of an existing aspect mechanism (the AspectAda language) to adapt and modify it to ensure real-time constraints.

المفاتيح: الأنظمة الآنيّة، البرمجة جانبيّة المنحى، لغة AspectAda، النّسيج، المحوّل البرمجي.

Mots clés: Systèmes Temps-Réel, Programmation Orientée Aspect, Langage AspectAda, Tissage, Compilateur.

Key-words: Real-Time Systems, Aspect Oriented Programming, AspectAda Language, Weaving, Compiler.