N° d'ordre : 3506
version : 7 décembre 2006

THÈSE

présentée devant

l'université de Rennes 1

pour obtenir le grade de

Docteur de l'université de Rennes 1
mention Informatique

par

Stéphane Hong Tuan Ha

Équipe d'accueil : Lande – IRISA
École doctorale : MATISSE
Composante universitaire : IFSIC

Titre de la thèse :

Programmation par aspects et tissage de propriétés

Application à l'ordonnancement et à la disponibilité

À soutenir le 30 janvier 2007 devant la commission d'examen

Composition du jury

M.	Jean-Pierre Banâtre	Examinateur
M.	Pierre Cointe	Rapporteur
M.	Frédéric Cuppens	Examinateur
M.	Pascal Fradet	Directeur
M.	Thomas Jensen	Examinateur
Mme	Florence Maraninchi	Rapporteur

Table des matières

Introduction

Pour des raisons de maintenance et de réutilisation, les applications sont décrites modulairement selon un ou plusieurs paradigmes de programmation. Cependant certaines préoccupations de l'application se décomposent mal selon les abstractions fournies par les langages de programmation et se retrouvent entrelacées dans toute l'application. La programmation par aspects [KLM+97] est un paradigme de programmation qui propose une solution à ces problèmes de décomposition. En programmation par aspects, une application est formée d'un programme de base et d'une collection d'aspects (décrivant, par exemple, les préoccupations de synchronisation, de tolérance aux fautes, ou de sécurité). Un outil, appelé tisseur, est chargé d'intégrer automatiquement les aspects à l'application. Comme les aspects sont décrits séparément du reste du programme, les programmes sont plus faciles à développer et à maintenir, leur lisibilité et leur réutilisabilité sont améliorées. La programmation par aspects est une technique récente et prometteuse qui commence à être utilisée dans l'industrie [DGB+06, Duc06]. La recherche sur la programmation par aspects porte entre autres sur des langages d'aspects plus expressifs et mieux adaptés. Les langages d'aspects généraux (comme AspectJ [LK98]) sont très expressifs et permettent de décrire n'importe quel aspect.

Un des défis de la programmation par aspects est de permettre un raisonnement modulaire. Le programmeur doit pouvoir raisonner sur l'application de base et les aspects sans avoir à regarder le code obtenu après tissage. Les langages d'aspects doivent permettre au programmeur de décrire simplement et à un haut niveau les préoccupations transverses. Leur syntaxe et leur sémantique doivent être simples et intuitives. Lorsque les aspects sont décrits dans des langages d'aspects très expressifs, il est difficile de prévoir le comportement de l'application tissée. Par exemple, un aspect d'AspectJ permet d'insérer du code java arbitraire (pouvant modifier les attributs des objets) ou de remplacer des appels de méthodes par d'autres. Il peut changer complètement la sémantique du programme de base. De plus, le tissage de deux aspects n'est généralement pas commutatif et les aspects peuvent interagir entre eux de manière non souhaitée par le programmeur. Raisonner sur le programme tissé ou résoudre les interactions entre les aspects passent par une visualisation (au moins mentale) du code tissé. Pour ces raisons, la formalisation des questions liées à ce paradigme comme la sémantique des langages d'aspects et la détection des interactions entre aspects devient une direction de recherche active [FL06].

Cette thèse propose une approche où les aspects représentent des propriétés sur les traces d'exécution. Nos aspects ont pour unique effet de sélectionner ou de couper des traces d'exécution du programme de base. A la différence des langages d'aspects généraux, l'impact sémantique des aspects est maîtrisé (pas d'insertion de code arbitraire). Notre approche garantit trois propriétés importantes qui permettent au programmeur de raisonner sur l'application et les aspects. La première propriété est que si le programme tissé termine normalement alors il retourne les mêmes résultats que le programme de base. La deuxième propriété est que le comportement d'un aspect est indépendant des autres aspects. En particulier, le tissage d'aspects est associatif et commutatif. La troisième propriété est que l'application tissée respecte la propriété décrite par l'aspect.

L'approche repose sur un formalisme simple, les automates (à états finis ou temporisés). Les programmes et les aspects sont représentés comme des automates et le tissage comme un produit d'automates. De cette manière, le tissage peut être vu comme l'intersection des traces du programme et des aspects. Nous avons appliqué cette approche à deux domaines :

- l'ordonnancement : spécifier et imposer des contraintes d'ordonnancement pour l'exécution de réseaux de composants ;
- la disponibilité : spécifier et imposer des propriétés de disponibilité à des services utilisant des ressources partagées.

Les aspects d'ordonnancement permettent de fusionner automatiquement des réseaux de composants en un programme séquentiel équivalent. Cette technique, la fusion de réseaux, transforme automatiquement un réseau de composants et un aspect d'ordonnancement en un unique programme séquentiel plus efficace. Comme une exécution parallèle se représente par un arbre des exécutions possibles, l'aspect en spécifiant un ensemble de traces autorisées décrit le choix d'un ordonnancement. Cet aspect permet de réconcilier la performance et la modularité en supprimant une bonne part du surcoût induit par la modularité.

La disponibilité assure que les demandes faites par des sujets autorisés sont toujours traitées (on dit qu'il n'y a pas de dénis de service). Notre approche peut s'appliquer à la prévention des dénis de service (*i.e.,* la disponibilité). Nos aspects spécifient des limites de temps ou de fréquence d'allocation des ressources. Ils peuvent être vus comme la spécification d'une politique de disponibilité prévenant les famines et les interblocages. Le tissage est décrit comme un *produit d'automates temporisés* (*i.e.,* une intersection des traces d'exécution temporisées) qui restreint le comportement du programme aux traces autorisées par l'aspect. Le tissage reviens alors à insérer des délais ou arrêter le programme.

Les contributions de cette thèse sont d'une part de proposer une approche originale et formelle où les aspects sont des propriétés et, d'autre part, d'appliquer cette approche à l'ordonnancement et la disponibilité. Le travail sur l'aspect d'ordonnancement a été présenté à la conférence internationale APLAS'04 [FH04]. Une version préliminaire du travail sur l'aspect de disponibilité a été présentée à la deuxième journée francophone sur les aspects JFDLPA'05 [FH05] et une seconde version étendue et profondément remaniée a été publiée dans la revue francophone L'objet [FH06].

Ce manuscrit se décompose en quatre chapitres.

- Le premier chapitre présente les principes et les fondations de la programmation par aspects. Nous commençons par présenter le paradigme des aspects et les caractéristiques principales des langages d'aspects. Nous survolons ensuite une collection de travaux visant à établir des fondations ou à formaliser la programmation par aspects. Ce premier chapitre s'attarde à montrer que si les langages d'aspect généraux sont très expressifs, ils ne permettent pas en général de raisonner modulairement.

- Le deuxième chapitre présente notre approche dans un cadre simple de programmes séquentiels et déterministes. Nous décrivons la syntaxe et la sémantique des programmes, les aspects, l'abstraction des programmes en automate, le tissage par produit d'automates, et la concrétisation. Un aspect définit une propriété de sûreté qui sélectionne les traces autorisées ou coupe les traces avant la violation de la propriété. Dans le cadre de programmes séquentiels où la sémantique est donnée par une unique trace linéaire, cela revient à arrêter le programme juste avant que la propriété soit violée. Nous illustrons ce chapitre par un exemple tiré du domaine de la sécurité. Cette partie présente les définitions et propriétés clés, les preuves de correction, ainsi que plusieurs extensions.
- Le troisième chapitre présente l'application de l'approche à l'ordonnancement. L'approche du chapitre 2 est étendue à des réseaux de processus concurrents en insistant sur les différences. Dans ce cadre de programmes parallèles où plusieurs traces d'exécution sont possibles pour un même état initial, l'aspect revient à sélectionner une ou des traces autorisées, ou à arrêter le programme quand aucune trace ne convient. Nous proposons aussi un langage d'aspects dédié plus riche qui permet à l'utilisateur de guider et contrôler l'exécution des réseaux.
- Le quatrième chapitre présente l'application de l'approche à la disponibilité. L'approche du chapitre 2 est étendue à des propriétés temporisées. Dans ce cadre, l'aspect sélectionne des traces ayant un comportement temporel adéquat. En terme d'instrumentation de code, cela revient à couper des exécutions ou à insérer des délais. En considérant que le programme de base peut avoir tous les comportements temporels possibles, l'insertion de délais, à l'instar de l'arrêt d'une exécution, peut être vue comme une sélection de trace. Nous proposons un langage d'aspects textuel de haut niveau adapté à la disponibilité. Les aspects ainsi définis sont ensuite traduits automatiquement en automates.
- Nous concluons la thèse par une synthèse et la description de plusieurs perspectives de recherche.

Chapitre 1

Programmation par aspects : principes et fondations

Sommaire

1.1 Problématique des langages de programmation

Les langages de programmation ont évolué depuis l'assembleur (un des premiers langages de programmation) et il existe maintenant différents paradigmes de programmation pour décrire une application. Par exemple, dans la paradigme impératif, le programmeur décrit son application comme une séquence d'instructions ; dans la programmation logique, comme un ensemble de formules logiques ; et dans la programmation fonctionnelle, comme un ensemble de fonctions mathématiques. Actuellement un des paradigmes dominants est la programmation orientée objet (POO) où une application est décomposée en objets qui représentent des données et les opérations pour modifier ces données. Ces langages ont l'avantage que la représentation interne des données peut être changée sans modifier tout le programme. Ce paradigme de programmation fournit aussi des méthodologies (comme UML) pour aider à la conception des applications (en particulier une méthode pour décomposer un programme en objets). On dit qu'un langage est de haut-niveau quand celui-ci permet de s'abstraire des détails du fonctionnement du matériel et de bas-niveau dans le cas contraire.

Progressivement l'informatique a été utilisée pour résoudre des problèmes de plus en plus complexes et comme les tailles des applications devenaient de plus en plus grandes, il est devenu important de faciliter leurs développements (lisibilité, développement séparé) et leurs évolutions (maintenabilité et réutilisabilité). Il est important que ces applications puissent être développées par des équipes disjointes de programmeurs et qu'un programmeur puisse comprendre les parties développées par un autre programmeur. Il est aussi nécessaire que ces applications puissent être modifiées facilement (sans devoir tout réécrire) et qu'on puisse facilement réutiliser des sous-parties de l'application (sans devoir réinventer la roue). L'augmentation de la taille des applications a montré que la décomposition (en procédures, formules, fonctions, ou objets) n'est pas toujours adaptée pour représenter certaines préoccupations de l'application. Ces *préoccupations transverses* ne peuvent pas être représentées dans les paradigmes de programmation habituels et se retrouvent séparées dans plusieurs objets. Ces préoccupations transverses ne permettent plus aux programmeurs de développer proprement des applications (code maintenable et réutilisable, *etc.*)

La programmation par aspects (*PPA* ou *AOP*[1] en anglais) est un nouveau paradigme de programmation qui fournit des solutions aux problèmes de décomposition des préoccupations traverses. Ce paradigme propose une nouvelle notion de modularité qui s'applique en complément des paradigmes de programmation habituels (impératif, fonctionnel, *etc.*). Cette thèse se place dans le domaine de la programmation par aspects et propose une approche formelle et originale de ce paradigme. Ce chapitre propose un survol des principes et fondations de la PPA.

[1] *Aspect Oriented Programming*

1.2 Le paradigme des aspects

1.2.1 Présentation générale

La programmation par aspects a été proposée par Kiczales et coll. en 1997 [KLM+97]. Les paradigmes de programmation habituels reposent sur le fait que les différentes préoccupations de l'application puissent se décomposer en une collection d'abstractions (procédures, fonctions, *etc.*). Cependant certaines préoccupations, que nous appelons transverses, ne peuvent pas respecter cette décomposition et se retrouvent séparées dans plusieurs abstractions. Un exemple simple de préoccupation transverse est la gestion du débogage : de nombreux ajouts sont nécessaires pour tester des assertions ou tracer une valeur qui se retrouvent entrelacés dans tout le code. De manière non exhaustive, la gestion de la sécurité, la mise en œuvre de transactions, la gestion de la distribution (synchronisation des accès concurrents) et la propagation de contexte sont des préoccupations transverses. Nous décrivons plusieurs de ces exemples de préoccupations transverses dans la partie suivante. La programmation par aspects se base sur l'idée qu'une application est mieux décrite en séparant toutes les préoccupations et propose de représenter de manière séparée les tâches transverses dans des aspects. De façon plus générale, le programmeur souvent poussé par le paradigme de programmation utilisé choisit une manière de décomposer l'application et certaines préoccupations ne respectent pas cette décomposition. Ce problème de la tyrannie de la décomposition dominante se produit avec tous les paradigmes de programmation. Même si la majorité des travaux sur la PPA sont appliqués pour la programmation orientée objets ; différentes recherches ont montré que ce paradigme pouvait aussi être appliqué à la programmation impérative [And01], fonctionnelle [TMY05] et réactive [AMS04].

La programmation par aspects repose sur un langage d'aspect pour décrire les préoccupations transverses et un tisseur qui intégrer automatiquement les aspects au programme de base. Il est possible de voir le tissage comme un moniteur de programme qui surveille l'exécution du programme et qui peut ajouter ou supprimer l'exécution d'instructions. Cependant, le tissage est généralement effectué de manière statique et peut se voir comme un préprocesseur syntaxique qui transforme le code en fonction de l'aspect. Le préprocesseur peut comporter des analyses afin que l'application tissée soit aussi efficace que si elle avait été développée sans aspects. Comme les aspects sont décrits séparément du reste du programme, les préoccupations transverses sont plus faciles à développer et à maintenir, augmentent la lisibilité de l'application, et sont réutilisables sur d'autres applications. La figure 1.1 illustre le problème des préoccupations transverses. L'application initiale est constituée de deux abstractions ; dans chacune de ces abstraction, une partie du code est liée à la sécurité et au débogage. Avec la programmation par aspects, l'application peut se décrire par les deux abstractions dans lesquelles les préoccupations transverses ont été enlevées et par deux aspects qui décrivent la sécurité et le débogage. Sur cet exemple, le tisseur (qui n'est pas décrit) prendrait

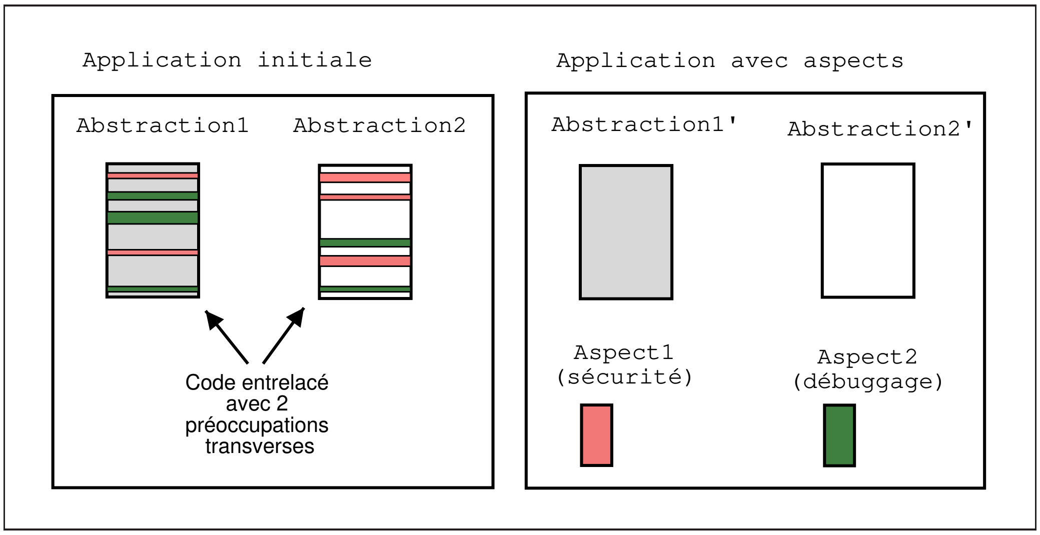


Fig. 1.1 – Séparation des préoccupations transverses avec la programmation par aspects

en entrée les objets et les aspects (partie droite de la figure), et retournerait en sortie l'application initiale où les préoccupations étaient entrelacées dans les deux objets (partie gauche de la figure).

L'objectif de la programmation par aspects est de permettre une meilleure décomposition de l'application en séparant les préoccupations transverses dans des aspects. Les aspects permettent de modifier le comportement de l'application de base et sont décrits dans des *langages d'aspects* qui décrivent où et comment modifier le programme et son exécution. Comme les langages d'aspects sont généralement basés sur des transformations syntaxiques, une des difficultés de la programmation par aspects est qu'il est généralement difficile de raisonner sur l'application avec aspects sans visualiser (au moins mentalement) le code tissée (*i.e.,* le code obtenu après tissage). De plus, l'application tissée peut avoir des comportements non souhaités dus aux interactions entre les aspects. Un des défis majeur de la programmation par aspects est donc de permettre un raisonnement modulaire. C'est à dire qu'il faut pouvoir écrire et raisonner sur l'application de base et les aspects sans étudier le code tissé.

1.2.2 Domaines d'application des aspects

La programmation par aspects permet de séparer les préoccupations transverses d'une application en les décrivant dans des aspects. Afin de donner un aperçu des utilisations de la PPA, nous donnons trois exemples d'application des aspects et listons la plupart des autres applications à notre connaissance.

Débogage

Un des exemples les plus utilisés pour illustrer la programmation par aspects est l'aspect pour le débogage. La gestion du débogage nécessite de nombreux ajouts pour tester des assertions ou tracer une valeur ou encore faire des logs des comportements potentiellement dangereux (comme les appels d'exceptions). Le débogage est une préoccupation transverse qui se retrouve entrelacée dans toute l'application. Un aspect pour le débogage [KHH$^+$01a] permet d'aider au développement en décrivant de manière séparée où ajouter les tests d'assertion et les différents affichages de valeurs.

L'utilisation d'un tel aspect permet au programmeur de décrire les comportements qu'il souhaite tester et lui évite d'ajouter de nombreuses lignes de code dans tout le programme. L'aspect de débogage a l'avantage qu'une fois les erreurs trouvées, il peut être simplement désactivé (à la place de supprimer ou commenter ces lignes). Cet aspect améliore la lisibilité du code, et permet une plus grande maintenabilité (l'aspect peut être réactivé en cas de nouvelles erreurs). Cependant, il est fortement dépendant du programme à déboguer et est difficilement utilisable sur un autre programme.

Gestion des permissions

Les politiques de gestion des permissions déterminent en fonction de l'historique de l'exécution si une permission doit être accordée [HR06]. Par exemple, la politique "*no right down*" définit qu'une fois que l'application accède à des données haut-niveau, elle n'a alors plus le droit d'écrire dans des données bas-niveau car il y a possibilité de transmettre un secret. Il existe plusieurs approches pour assurer une politique de sécurité sur la gestion des permissions. La première technique est de vérifier que l'application respecte la politique de sécurité. Cependant les analyses ne sont pas assez précises en général et peuvent refuser des programmes corrects. Une autre approche consiste à assurer la politique en utilisant un moniteur qui surveille l'exécution de l'application. Cette solution permet de définir la politique séparément de l'application mais pose des problèmes d'efficacité (dû à l'utilisation de moniteur). Une approche mixte est d'utiliser des analyses statiques pour s'assurer que le programme respecte la propriété et d'insérer manuellement des tests dans les cas où les analyses permettent pas de l'assurer que la propriété ne sera pas violée. Cette approche ne rejette aucun programme et permet d'éviter le surcoût de l'exécution en parallèle d'un moniteur comme seuls les tests nécessaires sont ajoutés. Cependant le code qui correspond à la politique de sécurité se retrouve entrelacé dans toute l'application.

Les aspects peuvent être utilisés pour imposer des politiques de gestion des permissions. Des analyses peuvent être utilisées lors du tissage [CF00] pour minimiser le nombre d'instructions ajoutées (et même laisser le programme inchangé dans le cas où il vérifie la politique de sécurité). L'utilisation d'un aspect permet de réconcilier les performances et la séparation des préoccupations. Ceci permet une meilleure maintenabilité (il est facile de modifier la politique de sécurité qui est décrite séparément) et une meilleure réutilisabilité (la politique de sécurité peut être appliquée sur d'autres applications).

Gestion des exceptions et conception par contrats

Lippert et Lopes montrent dans [LL00] que la programmation par aspects peut être utilisée pour simplifier la gestion des exceptions. Les exceptions fournissent un support pour traiter les cas d'erreurs, mais celui-ci peut devenir trop complexe dans le cadre de grosses applications et être source d'erreurs (comme le bug d'Ariane5 [Boa96]). Une façon de spécifier la gestion des exceptions est d'utiliser des méthodologies comme la conception par contrats. Les méthodes doivent alors vérifier des préconditions, des postconditions et des invariants qui dans le cas contraire déclenchent une exception. Par exemple, une méthode peut nécessiter que l'objet appelant soit d'un type donné et que les objets retournés soient instanciés (la référence est différente de `null`). L'utilisation de la PPA permet d'implémenter la conception par contrats en décrivant les préconditions, les postconditions et les invariants de manière séparée. Il en découle une meilleure compréhension, maintenabilité, et réutilisabilité de l'application (application de base et aspects). Ces aspects peuvent aussi simplifier les premières étapes de développement d'une application. Les programmeurs peuvent définir rapidement une gestion simplifiée des erreurs pour mieux se concentrer sur le fonctionnement correct du programme et plus tard décrire une gestion complète des erreurs.

Autres domaines d'application

Le guide du programmeur d'AspectJ[1] présente plusieurs utilisations possibles des aspects. Les aspects permettent de faire le profilage[2] d'une application en calculant par exemple le nombre de fois qu'une méthode a été appelée ou le nombre de fois qu'une méthode a été appelée à partir d'une autre méthode. Les aspects peuvent servir à décrire le passage de contexte d'une méthode à une autre. Cette tâche, qui nécessite d'ajouter un nouvel argument à toutes les méthodes de l'application, peut se décrire de manière séparée dans un aspect. Les aspects permettent aussi de calculer le temps de connexion d'un client en insérant le départ d'un minuteur dans la méthode de connexion et la sauvegarde de ce minuteur dans la méthode de déconnexion. [SDMML03] optimise l'utilisation du cache d'un serveur web en utilisant la PPA pour ajouter une stratégie de prélecture . Une stratégie simple consistant à mettre en cache tous les documents accessibles à partir d'un lien du document demandé (par exemple, une page web) donnera à l'utilisateur

[1] http://www.eclipse.org/aspectj/doc/released/progguide/index.html
[2] *profiling* en anglais

l'impression que les chargements sont plus rapides. [BdPK05] propose une approche pour gérer de manière réutilisable et maintenable la tolérance aux fautes dans les applications J2EE. Les auteurs supposent que la détection des fautes revient à détecter l'occurrence d'une exception et proposent une solution reposant sur la PPA pour traiter les fautes. Les aspects peuvent aussi servir pour la mise en œuvre de transactions, et la gestion de la distribution (synchronisation des accès concurrents) ; nous détaillons plus en détails ces deux exemples dans la partie 1.3.2.

La programmation par aspects est une technique récente et prometteuse. Pour preuve de son développement, elle commence à être utilisée dans l'industrie [DGB+06, Duc06]. La recherche sur la programmation par aspects porte entre autres sur des langages d'aspects plus expressifs et plus adaptés, et l'amélioration de la formalisation des questions liées à ce paradigme comme la sémantique des langages d'aspects et la détection des interactions entre aspects. Il existe aussi d'autres travaux sur le problème général du développement de logiciels avec aspects. Ce thème regroupe les méthodologies (à la UML) pour développer et maintenir des applications avec des aspects. Dans la suite de ce chapitre, nous nous focalisons sur les langages d'aspects et sur la formalisation de la programmation par aspects.

1.3 Les langages d'aspects

La programmation par aspects repose sur le langage d'aspects pour exprimer les préoccupations transverses et le tisseur pour insérer les aspects dans l'application de base. Les langages d'aspects doivent permettre au programmeur de décrire simplement et à un haut niveau les préoccupations transverses. La syntaxe doit être facile d'utilisation par le programmeur et la sémantique doit être intuitive.

Les travaux sur les langages d'aspects peuvent être séparés en deux axes. Le premier axe consiste à proposer des *langages d'aspects généraux* très expressifs permettant d'exprimer toutes les préoccupations transverses d'une application. Cependant cette grande expressivité laisse la possibilité au programmeur de casser le raisonnement modulaire ; en général, ces aspects peuvent changer complètement la sémantique de l'application et les interactions entre plusieurs aspects sont difficiles à maîtriser. Face aux difficultés rencontrées avec les langages généraux, un seconde axe vise à proposer des *langages d'aspects dédiés* permettant de décrire un unique type de préoccupation transverse. Comme ces langages sont spécifiques à un domaine d'application précis, ils ont une syntaxe plus adaptée que celle des langages d'aspects généraux et l'impact sémantique de ces aspects est généralement plus limité. Cependant, ils font aussi l'hypothèse implicite d'un cadre où il n'y a pas d'autres aspects et donc ne traitent pas des problèmes d'interactions.

Dans cette section, nous commençons par décrire les langages d'aspects généraux et en particulier le langage AspectJ qui est le premier et le plus connu des langages d'aspects. Nous présentons ensuite deux langages d'aspects dédiés qui s'appliquent sur des systèmes distribués.

1.3.1 AspectJ, un langage d'aspects général

Nous nous concentrons ici sur le langage AspectJ [LK98, KHH+01a, KHH+01b] un langage d'aspects général pour Java. Ce langage a été proposé par Kiczales et coll. pour montrer l'utilité de la programmation par aspects. Le lecteur intéressé pourra également regarder d'autres langages d'aspects généraux comme CaeserJ [AGMO06] pour le langage Java ou Arachne [DFL+05] pour le langage C.

Les langages d'aspects généraux décrivent où et comment influencer le programme de base. Par exemple, un aspect indique qu'une exception doit être lancée quand telle méthode est appelée avec tels paramètres. Les langages d'aspects généraux respectent cette séparation entre le "où" et le "comment". Ils sont construits sur trois notions : un modèle de *points de jonction* ("joint point") qui identifient les points de l'exécution où il est possible d'interagir, un langage de *points de coupure* ("point cut") qui permet de définir les points de jonctions d'intérêt pour un aspect (le où), et un langage d'*inserts* ("advice") qui décrit comment modifier l'exécution à un point de jonction (le comment).

Le modèle de points de jonction définit l'ensemble des points de l'exécution visibles par les aspects. Ces points de jonction comportent une partie statique et une partie dynamique. Dans AspectJ, la partie statique d'un point de jonction fait référence, par exemple, à l'appel d'une méthode, à l'accès à un attribut d'une classe, ou à l'exécution d'une exception. Dans le cas de l'appel d'une méthode, la partie dynamique fait référence, entre autres, aux valeurs des paramètres de la méthode, aux valeurs des attributs de l'objet appelant, ou à la pile d'appels des méthodes.

Le langage de points de coupure permet de choisir un sous-ensemble des points de jonction. AspectJ contient différents primitives et opérateurs (motifs, opérateurs logiques, *etc.*) pour définir des ensembles de points de jonction. Par exemple :

- call(void Point.setX(int))
  définit l'ensemble des appels de la méthode setX(int) de la classe Point ;
- call(void Point.*)
  définit l'ensemble des appels des méthodes de la classe Point. AspectJ permet l'utilisation de jokers comme * pour simplifier la définition des points de coupure ;

– pointcut move() :
  call(void Point.setX(int)) ||
  call(void Point.setY(int))

  définit le point de coupure move() qui est l'ensemble des appels aux méthodes setX(int) et setY(int) de la classe Point. Outre l'opérateur "ou" (||), AspectJ fournit les opérateurs logiques "et" (&&) et "non" (!).
– pointcut move(int x) :
  call(void Point.setX(int)) &&
  args(x)

  définit un point de coupure dynamique qui est paramétré avec une variable qui représente l'argument de la fonction Point.setX(int)) et qui pourra ensuite être utilisée dans le code de l'insert. Le point de coupure args(x) filtre les fonctions à un argument et lie l'argument à la variable x. Outre args(), les points de coupure target() et this() permettent de récupérer l'objet appelant et l'objet appelé ;
– cflow(call(void Point.setX(int))
  désigne l'ensemble des points de jonction sous l'appel de la fonction Point.setX(int). Ce point de coupure utilisé en conjonction avec un point de coupure call(void m(void)) permet de faire référence à l'appel de m() à partir d'une autre méthode.

Le langage des inserts offre les mécanismes pour ajouter du comportement aux points de coupure. AspectJ décrit également où ajouter précisément du code Java à l'aide des directives after, before et around. Comme le code Java n'est pas restreint, les inserts sont très expressifs et permettent par exemple de modifier les variables liées à un point de coupure. Un insert before (respectivement after) est exécuté avant (respectivement après) le point de coupure. Par exemple, l'insert suivant

```
after() : move() {
    flag = true
    }
```

met à vrai l'attribut flag après le point de coupure move() (défini précédemment). Un insert around met en attente l'exécution du point de jonction et exécute son propre code. Le point de jonction peut être exécuté par l'instruction particulière proceed() qui prend en argument le même nombre et les mêmes types d'argument que le point de coupure ; si le proceed() n'est pas présent, le point de jonction n'est pas exécuté. Par exemple, l'insert suivant

```
around(int x) :
    call(void Point.setX(int)) &&
    args(x) {
    if(x < 256) proceed(x);
    }
```

permet de ne pas exécuter la méthode Point.setX(int) quand son argument est supérieur à 256. AspectJ fournit la possibilité de changer les paramètres d'appel d'une méthode avec proceed() (en modifiant les arguments du point de coupure) ou la variable de retour d'une méthode lors d'un insert after (avec la construction after returning()).

Dans les langages d'aspects généraux, un aspect est une abstraction, au même titre qu'une classe, qui décrit une préoccupation transverse. Un aspect contient la déclaration des attributs, méthodes locales, points de coupure et inserts ; la portée de ces déclarations est localisée à l'aspect. Par exemple ; l'aspect suivant

```
aspect pointProfiling {
   int moveCall, rectangleCall, predCall ;

   boolean pred(int x) {
      return (x < 256)
   }

   pointcut move(int x ) :
      call(void Point.setX(int)) &&
      args(x)

   before(int x) : move(x) {
      moveCall++ ;
      if(x < 256) predCall++ ;
   }

   before(int x) : move(x) && cflow(call(* Rectangle.*) {
      rectangleCall++ ;
   }
}
```

est un aspect de profilage qui calcule le nombre d'appels aux méthodes de la classe Point (mémorisé dans la variable moveCall), le nombre de ces appels à partir de la classe Rectangle (variable rectangleCall)), et le nombre de ces appels dont l'argument vérifie le prédicat $pred(v)$ (variable predCall). Comme les classes, les aspects peuvent être instanciés. Par défaut, un aspect est unique et s'applique sur toute l'application, mais AspectJ offre aussi la possibilité d'instancier un aspect par objet (permettant pour l'aspect ci-dessus d'avoir le profilage pour chaque objet). AspectJ permet aussi de définir l'ordre d'application des aspects quand plusieurs inserts s'appliquent sur le même point de jonction. Par exemple, en supposant l'aspect ci-dessus et un autre aspect qui n'exécute pas la méthode si l'argument est supérieur à 256, alors l'ordre d'application détermine si l'appel d'une méthode avec un argument supérieur à 256 est comptabilisé.

Ces langages d'aspects généraux reposent sur des transformations d'ordre plus syntaxique que sémantique et donc obligent à raisonner sur le code tissé. Ceci nous semble un inconvénient majeur car l'intérêt de séparer les préoccupations transverses de l'application de base est de pouvoir raisonner indépendamment sur l'application et les aspects. Comme les inserts permettent d'insérer du code java arbitraire, (pouvant modifier les attributs des objets) ou de supprimer des appels de méthodes (avec l'utilisation de around sans proceed) les aspects peuvent changer complètement la sémantique du programme de base. Le tissage de deux aspects n'est donc pas commutatif et les aspects peuvent interagir entre eux de manière non souhaitée par le programmeur. Encore une fois, résoudre les interactions entre les aspects passe par une visualisation (au moins mentale) du code tissé. Un autre point négatif des langages généraux est qu'ils sont souvent de trop bas-niveau pour décrire simplement certaines préoccupations transverses. Par exemple, AspectJ ne fournit aucun mécanisme générique pour parler de l'historique de l'exécution. Pour préciser que l'événement X (comme l'appel d'une méthode) a déjà été rencontré il est nécessaire d'utiliser une variable mise à jour dans un insert lorsque l'événement est exécuté et ensuite de tester cette variable dans l'insert.

1.3.2 Langages d'aspects dédiés

Face à ces difficultés, les langages dédiés proposent des langages d'aspects pour un domaine spécifique qui permettent de décrire simplement la préoccupation transverse tout en maîtrisant mieux l'impact sémantique. Dans cette partie, nous présentons deux exemples de langage d'aspects dédié à la distribution et aux transactions.

Les langages COOL et RIDL

Les deux langages COOL et RIDL ont été proposés par Crista Lopes dans sa thèse [Lop97]. Dans ce travail, une application distribuée est constituée d'un ensemble d'objets distribués et concurrents qui communiquent par des appels à leurs méthodes partagées. Ces deux langages d'aspects dédiés facilitent le développement des systèmes distribués en séparant le code de base de l'application, qui est décrit dans un langage restreint sans instructions pour gérer les préoccupations transverses liées au cadre distribué.

Le langage COOL permet au programmeur d'exprimer la coordination entre les différents processus en spécifiant les contraintes pour appeler les méthodes partagées. Ce langage permet de spécifier les exclusions entre les différents appels de méthodes avec les deux constructions dédiées selfexclusif{} et mutexclusif{}. Chaque méthode inclue dans l'ensemble selfexclusif{} ne peut pas être exécutée par plusieurs processus en même temps. Les méthodes appartenant au même ensemble mutexclusif{} ne peuvent pas être exécutées en concurrence. Ce langage permet aussi de décrire des conditions qui bloquent l'entrée dans une méthode tant qu'elles ne sont pas respectées. Ces conditions sont spécifiées par la construction require et reposent sur des variables qui définissent l'état de synchronisation de l'objet. Les évolutions de l'état de synchronisation sont uniquement possible en entrée ou en sortie d'une méthode, et sont décrites par les constructions on_entry et on_exit. En supposant que nous avons une classe BoundedStack qui définit une pile bornée avec deux méthodes pop et push, l'aspect suivant

```
coordinator boundedStackCoord : BoundedStack {
   selfexlusive{pop,push} ;
   mutexlusive{pop,push} ;

   cond boolean full = false ;
   cond boolean empty = true ;

   push : requires !full ;
     on_exit {
       if (sp == MAX) full = true ;
       if (sp == 1) empty = false ;
     }

   push : requires !empty ;
     on_exit {
       if (sp == 0) empty = true ;
       if (sp == MAX-1) ful = false ;
     }
}
```

décrit la politique de synchronisation ; c'est à dire au maximum une unique méthode est exécutée à un instant donné, la méthode push nécessite que la pile ne soit pas pleine et la méthode pop nécessite que la pile ne soit pas vide.

Une autre difficulté dans les programmes distribués réside dans la transmission des paramètres lors d'un appel de méthode. Les solutions triviales de transmettre le référence de l'objet ou une copie complète de l'objet sont inefficaces. Le langage RIDL fournit les moyens de paramétrer pour chaque méthode comment les données (arguments des méthodes) sont effectivement transmises (par copie ou par référence). Par comparaison, le langage Java permet de définir pour chaque classe comment les attributs sont effectivement transmis (référence ou copie) ; il est possible d'avoir une spécification par méthode, mais de manière transverse et adhoc (en décrivant plusieurs classes héritées fictives, et en spécifiant pour chacune de ses classes la façon de la transmettre, puis en forçant le type de l'objet). En supposant que nous avons une classe Printer qui définit une imprimante avec trois méthodes print, cancel et getPrintQueue, l'aspect suivant

```
remote Printer {

   Status print(Document :
      copy (*.int, *.String, data));
   Status cancel(int);
   Document[] getPrintQueue();

  default :
     Document : (copy *.int, *.String)
}
```

décrit comment les paramètres sont transmis pour chacune des méthodes. Par exemple, lors de l'utilisation de la méthode print(Document), tous les attributs du Document sont transmis par copie. Un comportement par défaut pour les Document, où seuls les entiers et les chaines sont transmis par copie, est spécifié avec la directive default.

Le langage KALA

Un autre exemple est le langage KALA [FD06, FP06] dédié à la gestion des transactions dans un système distribué. La gestion des transactions est une préoccupation transverse à l'application. Les transactions permettent d'assurer la cohérence des variables d'un système en assurant que les variables nécessaires à un calcul n'ont pas été modifiées entre la lecture des variables (le début de la transaction) et l'écriture du résultat (le "commit" de la transaction) ; dans le cas contraire la transaction est annulée ("abort") puis relancée. Cependant les transactions simples ont été initialement conçues pour des accès brefs à des bases de données et pour des transactions plus longues, il est nécessaire d'utiliser des systèmes avancées de transactions dont l'implémentation s'entrelace dans toute l'application. Le langage Kala permet de spécifier séparément le système de transaction souhaité dans un aspect. Ce langage de haut-niveau permet de spécifier entre autres quelles sont les transactions (qui sont définies comme des méthodes), et les dépendances entre les transactions (telle transaction doit se terminer avant telle autre). Par exemple, l'aspect suivant

```
m1(parameter list){
   begin { }
   commit { }
   abort { terminate(self + "m2"); }
}

m2(parameter list){
   begin { dep(m1 cd self); }
   commit { }
   abort { terminate(self); }
}
```

spécifie un système de transactions imbriquées. La transaction m1 peut exécuter une transaction fils m2 qui a accès à ses données. Une première contrainte, le commit du père ne peut pas avoir lieu avant celui de son fils, est spécifiée par la directive dep(m1 cd self) (cd pour "commit dépendance") au début des transactions fils. La seconde contrainte, si le père effectue un "abort" alors il faut terminer ses fils, se décrit par terminate(self + "m2"). L'implémentation de l'aspect est effectuée en construisant un moniteur qui impose la politique de gestion des transactions. Le fonctionnement de ce moniteur est basé sur l'interception des appels de méthodes et par l'utilisation de barrières de synchronisation qui arrête conditionnellement le flot de contrôle d'une méthode.

Nous avons présenté dans cette partie les langages d'aspects en séparant les langages généraux des langages dédiés. D'un coté, les langages d'aspects généraux se placent dans un cadre général où de nombreux aspects peuvent s'appliquer sur l'application de base. Ces langages d'aspects sont très expressifs (comme ils visent à décrire n'importe quel aspect) mais cassent aussi le raisonnement modulaire (insertion arbitraire de code, interactions entre aspects). De l'autre coté, les langages d'aspects dédiés permettent de raisonner plus modulairement en décrivant l'application dans un langage restreint et la préoccupation transverse dans un langage dédié. Cette séparation entre les rôles de l'application de base et des aspects est orthogonale et permet de raisonner indépendamment sur la sémantique de l'application et sur l'implémentation de la distribution ou des transactions. Cependant les langages dédiés ne font pas d'hypothèse sur l'application d'autres aspects et donc ne traitent pas des problèmes d'interactions avec d'autres aspects. Par exemple, l'utilisation simultanée des deux langages dédiés n'a pas été prévue et nécessiterait de définir un langage dédié plus général pour la distribution et les transactions.

1.4 Travaux sur la formalisation de la programmation par aspects

La programmation par aspects est un paradigme récent et de nombreuses questions ne sont pas encore formalisées. Dans cette section, nous survolons une collection de travaux visant à établir des fondations ou à formaliser la programmation par aspects. En particulier, nous présentons des travaux sur les sémantiques de langages d'aspects, les classifications d'aspects, les interactions entre aspects, et des modèles formels d'aspects. Pour chacun de ces thèmes, nous introduisons les tenants et les aboutissants et présentons les articles les plus intéressants.

1.4.1 Sémantique des langages d'aspects

Les langages d'aspects généraux fournissent la syntaxe nécessaire pour décrire séparément les préoccupations transverses d'une application. Ces langages sont souvent accompagnés d'une sémantique informelle qui décrit le comportement usuel des différentes structures du langage. La sémantique formelle d'un langage est indispensable pour montrer la correction du tissage, qu'un aspect a le comportement attendu, ou qu'il n'y a pas d'interactions entre aspects. Une sémantique formelle permet aussi de clarifier les ambiguïtés de la sémantique informelle. Par exemple, la sémantique informatique d'AspectJ ne permet pas de déterminer si le point de coupure `around : call(methode()) && cflow(call(void methode()))` filtre seulement les appels récursifs de `methode`, tous les appels de `methode`, ou aucun appels de `methode`.

Les travaux sur les sémantiques des langages d'aspects peuvent être séparés selon deux approches. La première approche propose des sémantiques classiques (opérationnelle, dénotationnelle) pour les langages d'aspects [LÖ2, WKD04, DDFL06]. L'inconvénient de cette méthode est que ces sémantiques sont souvent complexes et comportent de nombreuses règles pour décrire un langage d'aspect complet. La seconde approche propose des calculs minimaux qui permettent d'interpréter les langages d'aspects [And01, BJJR04, JLZ06]. Cette méthode a l'inconvénient de nécessiter une étape de traduction du langage d'aspect vers le calcul minimal. La suite de cette partie dresse un panorama d'une sélection de travaux représentatifs de ces deux approches.

Sémantique dénotationnelle d'un langage d'aspects général [WKD04]

Wand, Kiczales et Dutchyn proposent une sémantique dénotationnelle pour un petit langage d'aspects qui contient les structures clefs de la programmation par aspects (points de jonction dynamiques, points de coupure et inserts). Cette sémantique permet de mieux comprendre le fonctionnement de la programmation par aspects et explicite certains détails techniques comme le fonctionnement des points de coupure (en particulier, comment les données sont transmises à l'insert) ou l'exécution des inserts (en particulier, l'insert de type `around`).

Le langage de base est un simple langage fonctionnel de premier ordre (sans exceptions ou processus). Un point de jonction est défini récursivement par la grammaire suivante :

$$\begin{array}{rcl} jp & ::= & \langle\rangle \mid \langle k, pname, wname, v^\star, jp\rangle \\ k & ::= & \mathsf{pcall} \mid \ldots \end{array}$$

Par exemple, le point de jonction $\langle pcall, f, g, v^\star, jp\rangle$ décrit l'appel d'une procédure f avec les arguments $v^\star$ à partir de la procédure g. Les points de coupure sont décrits par la grammaire suivante :

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

Cette sémantique décrit la fonction mathématique qui calcule le résultat du programme. Cette fonction utilise des sous-fonctions comme match-pcd $pcd\ jp$ qui prend un point de coupure et un point de jonction, et retourne $Fail$ ou les associations (de variables à des valeurs) nécessaires pour vérifier le point de coupure. La sémantique d'une expression est définie par la fonction

$$\mathcal{E}[\![e]\!] : Env \to PE \to AE \to T(\mathrm{Val})$$

qui prend un environnement Env (utilisé pour les proceed), un environnement de procédure PE (qui associe une fonction au nom des procédures) et un environnement d'inserts AE (l'ensemble des inserts pouvant s'appliquer), et retourne le calcul des variables (modélisé par une monade). L'évaluation d'une procédure est décrite par la règle suivante :

$$\begin{array}{l} \mathcal{E}[\![(pname\ e_1 \ldots e_n)]\!]\rho\phi\gamma \\ \quad = \mathsf{let} v_1 = \mathcal{E}[\![e_1]\!]; \ldots ; v_n = \mathcal{E}[\![e_n]\!] \\ \quad\quad \mathsf{in}(\mathsf{enter\text{-}joint\text{-}point}\ \gamma \\ \quad\quad\quad (\phi(pname)) \\ \quad\quad\quad (v1, \ldots, v_n) \end{array}$$

L'évaluation commence par calculer la valeur des arguments pour fonctionner selon un appel par valeurs. Ensuite, la procédure n'est pas exécutée mais le contrôle est donné à la fonction enter-joint-point (définie de manière similaire par une autre règle) qui tisse les différents inserts sur cet appel. Les problèmes de ce travail sont que les règles sont assez complexes et qu'une sémantique dénotationnelle fournit un sens aux programmes mais cache les détails d'implémentation du langage (par exemple, la sémantique du programme est donnée par un point fixe).

Sémantique opérationnelle d'un langage d'aspects général [DDFL06]

Douence et coll. proposent une sémantique opérationnelle à petits pas d'un langage d'aspects à la AspectJ. Cette sémantique a l'avantage que chaque fonctionnalité clef de la programmation par aspects est décrite de manière séparée. De plus, les hypothèses sur le langage de base sont limitées de telle sorte que ce travail peut être réutilisé pour d'autres paradigmes de programmation (impératif, orienté objet, logique, *etc.*)

Le sémantique petit pas du programme de base est décrit par une relation de transition $\to$ entre des configurations de la forme (C, Σ) où C représente le programme et Σ la mémoire (qui définit entre autres les valeurs des variables). Un programme est défini comme une suite d'instructions $i_1 : (i_2 : \ldots : (i_n : \epsilon))$

Un aspect est défini comme une fonction ψ qui prend en paramètre l'instruction du point de jonction et retourne une paire composée d'une fonction ϕ et d'un type t qui dénote le type d'insert (before, after, around). La fonction ϕ modélise les conditions dynamiques d'application d'un insert. Elle prend en paramètre la mémoire Σ et retourne le code à exécuter. La fonction ψ retourne ϵ quand l'aspect ne filtre pas l'instruction courante. La première règle modélise ce cas :

$$\textsc{NoAdvice}\frac{\psi(i) = \epsilon \quad (i : C, \Sigma) \rightarrow_b (C', \Sigma')}{(i : C, \Sigma) \rightarrow (C', \Sigma')}$$

La règle suivante montre le cas d'un insert before qui ajoute du comportement avant l'exécution de l'instruction filtrée.

$$\textsc{Before}\frac{\psi(i) = (\phi, \mathsf{before})}{(i : C, \Sigma) \rightarrow (\text{test } \phi : \bar{i} : C, \Sigma)}$$

L'instruction i est annotée $(\bar{i})$ pour éviter que cette instruction soit filtrée une seconde fois par l'aspect. La règle Advice est utilisée pour insérer le code de l'insert.

$$\textsc{Advice}\frac{}{(\text{test } \phi : C, \Sigma) \rightarrow (\phi(\Sigma) : C, \Sigma)}$$

test ϕ évalue les conditions dynamiques d'application de l'état courant (Σ). Si elles sont satisfaites, $\phi(\Sigma)$ rend le code de l'insert sinon l'instruction vide ϵ.

De la même manière, les auteurs étendent leur modèle de programme et d'aspects en décrivant des règles supplémentaires pour le tissage de plusieurs aspects, un langage de points de coupure dynamique (permettant de représenter cflow) et la majorité des constructions d'AspectJ et de CaesarJ.

Calcul minimal basé sur les algèbres de processus [And01]

Andrews a proposé le premier calcul minimal pour la programmation par aspects. Celui-ci repose sur les algèbres de processus à la CSP. Sa problématique est d'une part de montrer l'équivalence entre deux aspects en utilisant les techniques classiques d'équivalences de processus (comme la bisimulation), et d'autre part de montrer la correction du tissage.

Les programmes et les aspects sont décrits par des processus. Par exemple, le programme suivant

$$\begin{aligned} \mathsf{button}[\,] \quad &= \mathsf{lock(door)} \rightarrow \mathsf{turn(on)} \rightarrow \mathsf{button}[\,] \\ &\square\ \mathsf{turn(off)} \rightarrow \mathsf{unlock(door)} \rightarrow \mathsf{button}[\,] \end{aligned}$$

définit un bouton on/off d'un four micro-onde qui soit bloque la porte et allume le four, ou soit éteint le four et ouvre la porte. L'aspect suivant

$$\mathsf{sed}[\,] \quad = \mathsf{turn(on)} \rightarrow \mathsf{turn(off)} \rightarrow \mathsf{sed}[\,]$$

restreint les traces du programme à respecter le motif turn(on), turn(off), *etc.*. L'application d'un aspect sur un programme est spécifiée en synchronisant les processus sur un ensemble d'événements. Par exemple, root $=$ buttom[ ] | turn(_) | sed[ ]

représente le programme buttom[ ] sur lequel est appliqué l'aspect sed[ ]. Les traces infinies de celui-ci répètent le motif lock(door), turn(on), turn(off), unlock(door), *etc.*. L'auteur propose un algorithme de tissage adhoc qui supprime les événements de synchronisation. La correction de cet algorithme est montré en assurant que le programme tissé a les mêmes traces que le programme sur lequel est appliqué l'aspect. Par exemple, le tissage de l'application root (définie ci-dessus) retourne le processus

$$
\begin{array}{ll}
\mathsf{Root}[\,] & = \mathsf{lock(door)} \rightarrow \mathsf{turn(on)} \rightarrow \mathsf{Sync2}[\,] \\
\mathsf{Sync2}[\,] & = \mathsf{lock(door)} \rightarrow \mathsf{nil} \\
& \;\;\Box\; \mathsf{turn(off)} \rightarrow \mathsf{Sync3}[\,] \\
\mathsf{Sync3}[\,] & = \mathsf{unlock(door)} \rightarrow \mathsf{Root}[\,]
\end{array}
$$

L'auteur montre ensuite comment encoder un langage de programmes et un langage d'aspects dans ce calcul minimal. Leur langage d'aspects se limite à des points de coupure qui filtrent l'exécution d'un événement et ne permet pas de décrire des points de coupure tels que cflow. Un autre problème est que la transformation des programmes et des aspects en processus est assez complexe.

Un calcul minimal basé sur le lambda-calcul [JLZ06]

Ligatti, Walker et Zdancewic proposent un calcul minimal simple et complet qui permet de modéliser le comportement des applications avec aspects. Ce travail a pour objectif de donner des sémantiques aux langages d'aspects en les représentant dans un unique cadre afin de les étudier ou de les comparer. Les auteurs illustrent leur travail sur un langage d'aspects typé (proche d'AspectJ) nommé MinAML. Un résultat de ce travail, que nous ne détaillons pas, est que le typage du langage MinAML est prouvé sûr (*type safe*). Le calcul minimal proposé étend un lambda-calcul simplement typé avec booléens, chaînes de caractères et n-uplets en ajoutant des points de jonction étiquetés et des inserts de premier ordre.

Un point de jonction étiqueté $l\langle e\rangle$ explicite les endroits où un insert peut être activé. Par exemple, dans l'expression $v + l\langle e\rangle$ (v représente une valeur et e une expression), une fois que e a été évalué à la valeur v', l'évaluation du sous-terme $l\langle v'\rangle$ va transmettre le contrôle aux inserts associés à l'étiquette l.

Un insert $\{l.x \rightarrow e\}$ est activé quand un point de jonction étiqueté $l\langle v\rangle$ est évalué. La variable x est alors substituée par la valeur v et l'exécution continue avec l'évaluation de e. Un insert $\{l.x \rightarrow e\}$ impose que le type de e soit le même que le type de x.

La calcul minimal fournit la syntaxe d'un lambda-calcul étendu (définition et application de termes, instruction conditionnelle, instruction print) à laquelle est ajoutée des expressions pour définir et utiliser les points de jonctions étiquetés. Plusieurs inserts peuvent être appliqués au même point de jonction étiqueté. Les expressions $a \ll e$ et $a \gg e$ installent l'insert a avant, respectivement après, les autres inserts déjà installés, puis continue en évaluant l'expression e. Par exemple,

$$\begin{array}{l} \{l.x \to x+1\} \ll (\{l.y \to y*2\} \ll l\langle 3\rangle) \longmapsto^\star 7 \\ \{l.x \to x+1\} \ll (\{l.y \to y*2\} \gg l\langle 3\rangle) \longmapsto^\star 8 \end{array}$$

où $\longmapsto^\star$ représente l'évaluation d'une expression.

La dernière possibilité syntaxique offerte par ce calcul est de modifier le flot de contrôle par un mécanisme proche des exceptions. L'expression $\mathsf{return}\ v$ to l transmet le contrôle à un point de jonction étiqueté par l en lui passant la valeur v. S'il n'y a pas un point de jonction étiqueté par l, le programme est arrêté (comme lors d'une exception non rattrapée). Par exemple,

$$p\langle 4 + (\text{return } 3 \text{ to } p)\rangle \longmapsto^\star 3$$

L'exemple, qui suit, décrit un insert qui modifie une fonction $f = \lambda x.e$ (avec x une variable de type booléen) de telle sorte que si son paramètre est vrai alors e est évalué normalement et sinon e' est évalué à la place.

$$\begin{array}{l} \{f_{\text{pre}}.x \to \text{if } x \text{ then } x \text{ else return } e' \text{ to } f_{\text{post}}\} \\ \ll \\ \lambda x.f_{\text{post}}\langle \text{let } x = f_{\text{pre}}\langle x\rangle \text{ in } e\rangle \end{array}$$

L'insert nécessite l'instrumentation de deux points de jonction étiquetés, f_{pre} et f_{post}, qui sont activés à l'entrée et à la sortie de la fonction.

La sémantique et le système de type de ce calcul minimal sont décrits formellement dans l'article [JLZ06].

L'encodage du langage de base (un langage objet classique) dans le calcul minimal repose sur l'ajout de points de jonction f_{pre} et f_{post} pour chaque fonction (comme sur l'exemple précédent). La déclaration de la fonction

```
fun p(x) = e
```

est traduit dans le calcul minimal par

$$\lambda x.p_{\text{post}}\langle \text{let } x = p_{\text{pre}}\langle x\rangle \text{ in } e\rangle$$

Le langage MinAML est un langage d'aspects général à la AspectJ qu'il est possible de traduire dans le calcul minimal. Par exemple l'insert

```
around g(x) = e
```

qui remplace le calcul effectué par la fonction g par l'expression e est traduit dans le calcul minimal par

$$\{g_{\text{pre}}.x \to \text{return } e \text{ to } g_{\text{post}}\}$$

Cet insert `around` est moins expressif que celui d'aspectJ qui permet d'exécuter g avec l'instruction particulière `proceed`. De la même manière, les auteurs décrivent les inserts `before` et `after` et des extensions syntaxiques (pour définir des objets et des points de coupure complexes).

1.4.2 Classification des aspects

Les langages d'aspects généraux permettent de décrire séparément le code de l'application et les préoccupations transverses mais l'expressivité de ces langages peut facilement bouleverser le comportement (la sémantique) des programmes. Cependant on peut aussi remarquer que tous les aspects ne sont pas aussi intrusifs. Des travaux cherchent à caractériser ces classes particulières d'aspects qui préservent certaines propriétés de l'application de base. Par exemple, Dantas et Walker [DW06] caractérisent des aspects (appelés *harmless*) qui n'interfèrent pas avec les calculs de l'application de base. Rillard et coll. [RSB04] catégorisent les aspects selon leurs interactions avec les méthodes de l'application de base (par exemple, *orthogonal* si l'aspect et la méthode ne partagent aucune variable, ou *indépendant* si l'aspect ne peut pas écrire dans des attributs que peut accéder la méthode). Dans la suite de cette partie, nous décrivons la classification des aspects de Katz que nous considérons comme la plus exhaustive.

Classification des aspects selon les propriétés temporelles conservées par le tissage [Kat06]

La classification de Katz se base sur les propriétés de l'application conservées par le tissage d'un l'aspect. Cette classification définit trois catégories d'aspects qui sont *spectateur*, *régulatif*, *faiblement invasif* et *invasif*. Ce travail modélise les applications par leurs graphes d'états (proches des *statechart* d'UML ou plus généralement des graphes de flot de contrôle) et la sémantique des aspects par une transformation du graphe d'états. La sémantique de l'application est donnée par une trace d'exécution dont les états sont composés de l'état courant dans le graphe d'états et d'une mémoire associant des valeurs aux variables.

Un aspect spectateur peut changer les variables locales de l'insert et de l'aspect, mais ne peut ni changer les valeurs des variables de l'application, ni changer le graphe des appels de méthodes. Cela assure que le programme tissé retournent les mêmes résultats que le programme de base. Par exemple, l'aspect de débogage proposé en 1.2.2 appartient à cette catégorie. Plus formellement, un aspect est spectateur si la projection du graphe d'états de l'application tissée sur les variables de l'application de base est identique au graphe d'états de l'application de base, à l'exception que certains états de la projection sont répétés et sont connectés par des transitions qui correspondent au traitement effectué par l'aspect. Il est possible de prouver qu'un aspect (à la AspectJ) est spectateur en analysant son code et en vérifiant qu'il ne peut pas écrire dans les variables de l'application, que le code des inserts terminent toujours et que les inserts around effectuent toujours un proceed. Cette classe d'aspects garantit que les propriétés de sûreté et de vivacité des exécutions de l'application de base (avec la restriction qu'elles ne parlent pas de "l'état suivant d'un état" et les invariants sur les variables sont préservées dans les exécutions de l'application tissée (*i.e.*,, toutes les propriétés que l'on peut exprimer dans la logique LTL sans l'opérateur "next").

Un aspect régulatif peut modifier le graphe d'états de l'application mais ne modifie pas les variables de l'application de base. Ces aspects peuvent bloquer ou mettre en attente l'exécution de certaines instructions mais ne peuvent pas sauter des instructions de l'application de base puis continuer normalement. Par exemple, l'aspect pour la gestion des permissions, en 1.2.2, est de type régulatif car il arrête le programme en cas de violation de la politique sur les permissions. Plus formellement, un aspect est régulatif si la projection du graphe d'états de l'application tissée sur les variables de l'application de base est identique au graphe d'états de l'application de base, à l'exception que certains états sont dupliqués (comme pour les aspects spectateurs) et que certains états et transitions sont supprimés. Cette classe d'aspect garantit que les propriétés de sûreté des traces de l'application de base (avec la restriction qu'elles ne parlent pas de "l'état suivant d'un état") et les invariants sur les variables du programme de base sont préservées dans les traces de l'application tissée. Les propriétés de vivacité ne sont plus automatiquement préservées car les exécutions peuvent être coupées. En général, un aspect régulatif impose des contrôles supplémentaires qui restreignent l'exécution de l'application de base de telle sorte que l'application tissée respectent des propriétés de sûreté supplémentaires.

Un aspect est faiblement invasif si les transitions de l'application de base commencent dans des états qui existent dans le graphe d'état de l'application de base [GK06]. En général, un aspect faiblement invasif peut faire passer la trace d'exécution dans un état inatteignable de l'état initial de l'exécution courante mais atteignable d'une autre exécution. Cette classe d'aspects garantit que les invariants sur les variables de l'application de base sont préservés dans l'application tissée. Les propriétés de sûreté ne sont pas automatiquement maintenues, car ces aspects peuvent changer le flot de contrôle de l'application.

Un aspect invasif n'est pas contraint ; il peut modifier les variables de l'application de base et créer de nouveaux états dans le graphe d'état. En général, ces aspects ne préservent pas les propriétés de l'application de base et contrarient le raisonnement modulaire.

1.4.3 Détections des interactions entre aspects

Les préoccupations transverses ne sont pas toujours orthogonales entre elles et les aspects correspondants peuvent interagir entre eux. Il arrive fréquemment, par exemple, que des inserts d'aspects différents soient activés par le même point de jonction. Les interactions entre aspects sont difficilement maîtrisables et peuvent amener des comportements non souhaités ou incorrects. Un exemple souvent utilisé pour illustrer les interactions entre aspects est le cas d'un aspect qui enregistre les données envoyées sur le réseau et d'un aspect qui crypte les données transmises sur le réseau. Cet exemple montre que l'ordre d'application des aspects est important car il détermine si les données qui sont enregistrées seront cryptées ou non cryptées. Selon les applications, le programmeur peut préférer l'une ou l'autre des deux possibilités.

Il est donc nécessaire d'avoir des outils et des analyses pour détecter ces interactions afin de s'assurer que le programmeur explicite l'ordre d'application et supprime toute ambiguïté. Ces outils devront aussi permettre de choisir la composition adéquate entre les aspects (l'ordre d'application dans le cas d'une composition séquentielle). Les classifications d'aspects apportent aussi des solutions à ces problèmes d'interactions. Notons que selon la classification de Katz, deux aspects spectateurs ou régulatifs ne peuvent pas s'influencer mutuellement comme leur tissage est commutatif (l'ordre d'application n'importe pas). Cependant deux aspects invasifs peuvent interagir ou non. C'est pourquoi des travaux proposent des analyses pour détecter les indépendances entre aspects [DFS04, DSBA05]. Nous présentons ci-dessous un article représentatif de ce type de travaux.

Détection des interactions entre aspects à états [DFS04]

Douence, Fradet et Südholt présentent un cadre formel pour analyser les interactions entre aspects. Ce cadre repose sur une définition formelle d'un petit langage d'aspects et fournit une analyse pour tester l'indépendance entre deux aspects. Ce travail fournit aussi des outils (directives du langage d'aspects) pour résoudre les conflits détectés mais nous ne les détaillons pas ici.

Le modèle d'aspects repose sur des règles linguisitiques de la forme $C \rhd I$ où C représente un point de coupure et I représente l'insert. L'insert I est de type `after` et est exécuté si le point de coupure C filtre l'instruction du programme de base. Les aspects peuvent se voir comme un automate de règles qui est défini par la grammaire suivante :

$$\begin{array}{llll} A & ::= & \mu a.A & \text{; définition récursive} \\ & | & C \rhd I; A & \text{; séquence} \\ & | & C \rhd I; a & \text{; fin de séquence} \\ & | & A_1 \Box A_2 & \text{; choix} \end{array}$$

Les règles peuvent être composés par l'opérateur de séquence ;. L'opérateur de point fixe $\mu a.A$ définit l'étiquette a et continue avec l'aspect A qui sera exécuté de nouveau s'il atteint une étiquette a (fin de séquence). L'opérateur de choix $A_1 \Box A_2$ choisit le premier aspect qui filtre l'instruction courante et l'autre aspect est ensuite ignoré. Dans le cas où les deux aspects filtrent l'instruction courante, l'aspect A_1 est choisi arbitrairement. Ces aspects sont différents de ceux d'AspectJ, car les règles sont consommées quand elles ont réussi à filtrer une instruction et l'aspect continue avec la règle suivante. Ces *aspects à états* permettent des analyses plus précises comme ils possèdent un état qui détermine l'historique de l'exécution En AspectJ, cet état devrait être codé par des variables mises à jour dans les inserts. Par exemple, l'aspect suivant enregistre ("log") toutes les lectures (les appels à read) qui sont effectuées pendant une session (c'est à dire, entre un login et un logout).

$$\mu a_1.\mathsf{login} \rhd \mathit{skip}\ ;\ (\mu a_2.(\mathsf{logout} \rhd \mathsf{skip}\ ;\ a_1) \,\Box\, (\mathsf{read(x)} \rhd \mathsf{log(x)}\ ;\ a_2))$$

Nous reparlerons de ce langage d'aspects dans le chapitre 4, car nous nous en sommes inspirés pour proposer un langage d'aspects pour la disponibilité.

Les restrictions sur les aspects (uniquement after) permettent aux auteurs de fournir une sémantique simple du tissage de plusieurs aspects sur la même application de base. Les auteurs proposent ensuite deux analyses pour tester l'indépendance entre des aspects. Deux aspects sont dits fortement indépendant (*strongly independant*) quand un point de jonction ne peut pas activer les deux aspects en même temps. L'analyse correspondante calcule un aspect équivalent aux deux aspects (par une technique similaire à un produit d'automates). Les conflits sont représentés par l'exécution non déterministe de deux inserts $I_1 \Diamond I_2$. Si l'aspect est déterministe (ne contient pas de symbole $\Diamond$) alors les deux aspects sont indépendants. Par exemple, la composition des deux aspects A et A'

$$A \parallel A' = (\mu a.C \rhd I; a) \parallel (\mu a.C' \rhd I'; a)$$

est réécrit par l'aspect

$$\mu a.(C \wedge C' \rhd (I \Diamond I'); a) \,\Box\, (C \rhd I; a) \,\Box\, (C' \rhd I'; a)$$

Si C et C' ne filtrent jamais les mêmes points de jonction, alors l'exemple peut être réécrit par l'aspect déterministe

$$\mu a.(C \rhd I; a) \,\Box\, (C' \rhd I'; a)$$

Deux aspects sont dits indépendants pour un programme donné si aucun point de jonction de ce programme ne peut activer les deux aspects. Par exemple, les deux points de coupure (à la AspectJ) call(void *.move(*,int)) et call(* Point.*(int,*)) sont indépendants si le programme ne contient pas d'appels de méthode qui unifient ces deux motifs (*i.e.*,, call(void Point.move(int,int))). Ces propriétés d'indépendance sont fortes car elles ne dépendent pas du traitement effectué par les inserts. Deux aspects peuvent aussi être indépendants si les aspects ne s'influencent pas mutuellement (par exemple dans le cas d'aspects de type spectateur).

1.4.4 Modèles formels d'aspects

Jusqu'à présent, nous avons seulement considéré des modèles d'aspects similaires à celui d'AspectJ dans lequel les applications de base sont séquentielles et les points de jonction possèdent une partie statique représentant l'instruction courante de l'exécution. Ce modèle de point de jonction n'est pas toujours adapté pour décrire certaines préoccupations transverses et pour cette raison, des travaux proposent d'autres modèles de plus haut niveau [BH05, AR05, KHJ06, CF00]. Colcombet et Fradet une technique qui permet d'imposer des propriétés sur la trace d'exécution [CF00]. Ce travail est un des points de départ de cette thèse et est décrit plus en détail dans le chapitre 2. Ali et Rashid proposent un langage de points de jonction de haut niveau basé sur les changement d'états d'un système [AR05]. L'idée est qu'un état est modélisé par des conditions sur les attributs (par exemple "dist $> 400 \ \wedge$ speed < 50") et qu'un système peut se voir comme un automate qui passe d'un état à un autre. Dans cette modélisation, les points de jonction sont définis comme le passage d'un état q_1 à un état q_2 ou l'arrivée (resp. départ) dans (resp. vers) un état q. Ces points de jonction ont la particularité de ne dépendre que des attributs de l'application et d'être défini indépendamment des méthodes (qui modifient ces attributs). D'autres travaux ciblent les langages concurrents [AMS06, DBNS06] dans lesquels il est nécessaire de définir une nouvelle notion d'aspect. Dans la suite de cette partie, nous présentons un article représentatif de ce thème.

Programmation par aspects pour les systèmes réactifs : Larissa, une proposition dans le cadre synchrone [AMS06]

Maraninchi, Altisen et Stauch explorent la notion de préoccupations transverses dans les systèmes réactifs et proposent un langage d'aspects pour ces systèmes. Un système réactif est une application qui réagit continuellement à leur environnement (à une vitesse imposée par l'environnement). Les auteurs s'intéressent plus spécifiquement aux applications définies par les langages synchrones (comme Lustre [HLR92] et Esterel [BG92]). Ces langages représentent le temps comme une suite d'instants. A chacun de ces instants, le système lit les événements en entrée, fait évoluer son état et émet des événements en sortie. Ces langages reposent sur une description des différentes tâches de l'application qui fonctionne en parallèle et qui communiquent par des envois de signaux en *broadcast*. Cette communication par *broadcast* est asymétrique (le composant qui envoie un signal ne connait pas le nombre de composants qui attendent ce signal). Dans le cadre synchrone, un observateur qui impose une propriété de sûreté se décrit comme un composant comme les autres et n'est pas une préoccupation transverse. Les auteurs montrent que certaines préoccupations sont transverses dans les applications réactives. Par exemple, la réinitialisation de l'application à la réception d'un événement particulier est implantée par du code réinitialisation entrelacé dans toutes les tâches de l'application.

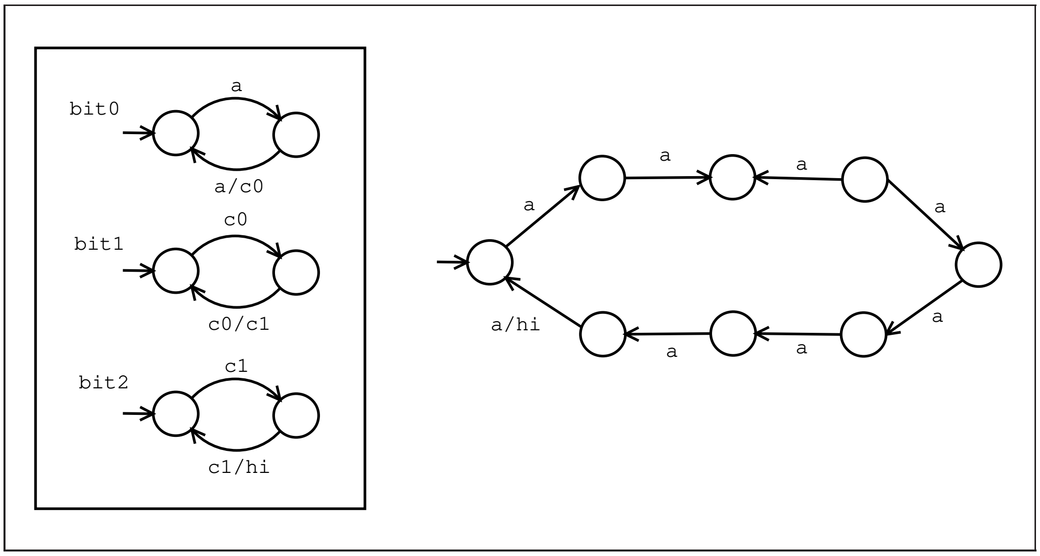


FIG. 1.2 – Description et sémantique d'un compteur modulo huit

Ce travail propose une modélisation qui repose sur des automates communiquants par *broadcast*. Les transitions sont étiquetées par condition / signaux emis où condition représente une condition sur les signaux en entrée et signaux emis définit les signaux qui sont émis. Par exemple, $a \wedge \bar{b}/c$ émets le signal c si le signal a est présent en entrée et le signal b n'est pas présent. Pour faciliter les notations, nous notons seulement a quand aucun signal est émis ($a/\emptyset$). Par exemple, la partie gauche de la figure 1.2 définit un compteur modulo huit qui toutes les huit réceptions d'un signal a envoie un signal hi. Ce compteur est défini par trois sous-compteurs modélisant chacun 1 bit. La sémantique de l'application est définie par un automate qui est la composition parallèle des automates des différentes tâches de l'application. La partie droite de la figure 1.2 décrit l'automate représentant la sémantique du compteur modulo huit. Les aspects dans ce cadre définissent des transformations de l'automate qui exprime la sémantique.

Les inserts définissent l'ajout de transitions dans l'automate. L'état destination de la transition est spécifiée par une séquence d'événements en entrée à partir de l'état initial ; cette séquence détermine un unique état car l'automate est déterministe. Par exemple, l'insert qui ramène le compteur à son état initial est spécifié par la séquence d'événements vide $\sigma = \epsilon$. Le programmeur pourrait aussi décrire un insert qui amène dans l'état atteint après quatre signaux a ($\sigma = (aaaa)$) Les points de jonction sont définis comme les états de l'automate et le point de jonction courant est l'état courant de l'automate. Les points de coupure spécifient où les inserts sont appliqués ; c'est à dire les états de l'automate qui sont les états d'origine des transitions ajoutées par les inserts. Les points de coupure sont encore une fois spécifiés par un ensemble de séquences d'événements en entrée qui spécifie un ensemble d'états de l'automate. Pour plus de facilité, ces séquences sont spécifiées par un observateur qui émet l'événement JP dans tous les états filtrés par le point de coupure. Par exemple, la partie gauche de la figure 1.3 décrit le point de coupure qui filtre tous les états, et la partie droite décrit un point de coupure qui est actif quand le nombre d'événements a reçus est pair.

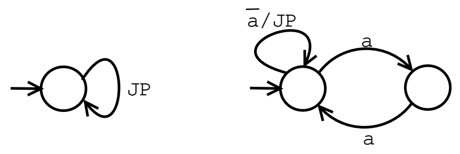


Fig. 1.3 – Exemples de points de coupure avec Larissa

Ce langage d'aspects a la particularité d'être indépendant du codage de l'application. Cette définition sémantique des aspects assure que pour deux applications sémantiquement équivalentes (pour les mêmes événements en entrée retourne les mêmes événements en sortie) l'application du même aspect préserve cette équivalence sémantique. En général, les langages d'aspects peuvent se voir comme des directives de transformation syntaxique de programme et ne permettent pas d'assurer une telle propriété sur la sémantique du programme tissé.

1.5 Conclusion

Nous avons présenté dans ce chapitre les principes de la programmation par aspect et une sélections de travaux sur la formalisation de cette approche. Les aspects sont une nouvelle notion de modularité qui facilite le développement, la réutilisation, la lisibilité et la maintenabilité des applications. Le tissage est généralement effectué de manière statique et peut se voir comme un préprocesseur syntaxique qui transforme le code en fonction de l'aspect. Un des défis majeur de la programmation par aspects est de permettre un raisonnement modulaire. C'est à dire qu'il faut pouvoir écrire et raisonner sur l'application de base et les aspects sans étudier le code tissé. Les langages d'aspects dédiés permettent de raisonner plus modulairement en décrivant l'application dans un langage restreint et la préoccupation transverse dans un langage dédié. Cette séparation entre les rôles de l'application de base et des aspects permet de raisonner plus modulairement. Cependant ils ne font pas d'hypothèse sur l'application d'autres aspects et donc ne traitent pas des problèmes d'interactions avec d'autres aspects. Les langages d'aspects généraux se placent dans un cadre général où de nombreux aspects peuvent s'appliquer sur l'application de base. Cependant ces langages d'aspects sont très expressifs et ne permettent pas en général de réfléchir indépendamment sur l'application de base ou les aspects (insertion de code arbitraire et interactions entre les aspects). Les travaux sur la classification des aspects et sur les interactions entre aspects offrent des solutions partielles en garantissant qu'il n'y a pas d'interactions entre l'application de base et les aspects.

A la différence des approches présentées, nous voyons les aspects comme des propriétés que le tissage va imposer sur l'application. Le langage d'aspects que nous proposons se situe entre les langages dédiés et les langages généraux, et pourrait se voir comme un langage général restreint. Par rapport à la classification de Katz, nos aspects sont de type régulatif et permettent de garder une maîtrise de l'impact des aspects. Les propriétés de sûreté de l'application de base sont préservées et ces aspects imposent des propriétés de sûreté supplémentaires sur l'application tissée. Cette approche a aussi l'avantage que les interactions entre les aspects sont maîtrisées comme le tissage de tels aspects est associatif et commutatif. Notre formalisation repose sur les automates pour représenter l'application de base mais aussi les aspects (à la différence de Larissa). Le tissage peut se voir comme une restriction des traces de l'application par la propriété. Nous présentons cette approche dans le chapitre 2. Nous montrons ensuite dans les chapitres 3 et 4 comment l'appliquer à l'ordonnancement et à la disponibilité.

Chapitre 2

Tissage de propriétés

Sommaire

2.1 Introduction

Nous avons montré dans le chapitre précédent que la plupart des formalisations de la programmation par aspects (calculs minimaux, sémantiques) prennent en compte des langages d'aspects généraux sur lesquels il est difficile de raisonner. Dans cette thèse, nous adoptons une approche formelle où où les aspects peuvent être vus comme des propriétés sur les traces d'exécution du programme. Notre approche a l'avantage de permettre de raisonner sur l'application de base et les aspects. Notre formalisation repose sur les automates pour représenter l'application de base et les aspects. Le tissage est défini comme une composition d'automates et peut se voir comme une restriction du comportement de l'application selon la propriété. Dit autrement, l'aspect spécifie l'ensemble des traces d'exécution (comportement) autorisées. Ce cadre pour la programmation par aspects a été appliqué à différents domaines d'applications comme la sécurité (*c.f.* l'exemple décrit dans ce chapitre), l'ordonnancement de programmes parallèles (*c.f.* chapitre 3), ou encore la disponibilité (*c.f.* chapitre 4).

Les aspects sont de type régulatif : ils ne changent pas les variables de l'application de base et peuvent seulement sélectionner ou couper une trace d'exécution de l'application. Comme décrit dans la partie 1.4.2, cette classe d'aspect garantit que les propriétés de sûreté de l'application de base (avec la restriction qu'elles ne parlent pas de "l'état suivant d'un état") et les invariants sur les états sont préservés dans l'application tissée. Ce type d'aspects garantit aussi que si le programme tissé termine normalement alors il retourne les mêmes résultats que le programme de base. Cette approche a aussi l'avantage que les interactions entre les aspects sont maîtrisées comme le tissage est associatif et commutatif. Il est possible de construire un aspect représentant les autres aspects en les combinant par produit et de le tisser de la même manière.

Cette approche est inspirée du produit instrumenté [CF00] que nous présentons en partie 2.9.2. Ce travail propose une technique, différente du produit d'automates, qui permet d'imposer des propriétés décrites par des automates sur des programme. Notre approche ne se limite pas à la technique pour imposer la propriété mais présente un cadre complet où les aspects représentent des propriétés. En particulier, nous proposons différents langages d'aspects pour décrire les automates et montrons comment utiliser cette approche pour l'ordonnancement et la disponibilité.

Les différentes étapes de notre approche sont illustrées dans la figure 2.1. Notre approche prend en entrée un programme et un aspect, et produit automatiquement un programme tissé respectant l'aspect. La première étape consiste à abstraire le programme en un automate proche de son graphe de flot de contrôle. L'aspect est exprimé sur des événements (appels de méthodes, ...) du programme. L'aspect spécifie l'ensemble des traces d'exécution autorisées. Le tissage est effectué simple-

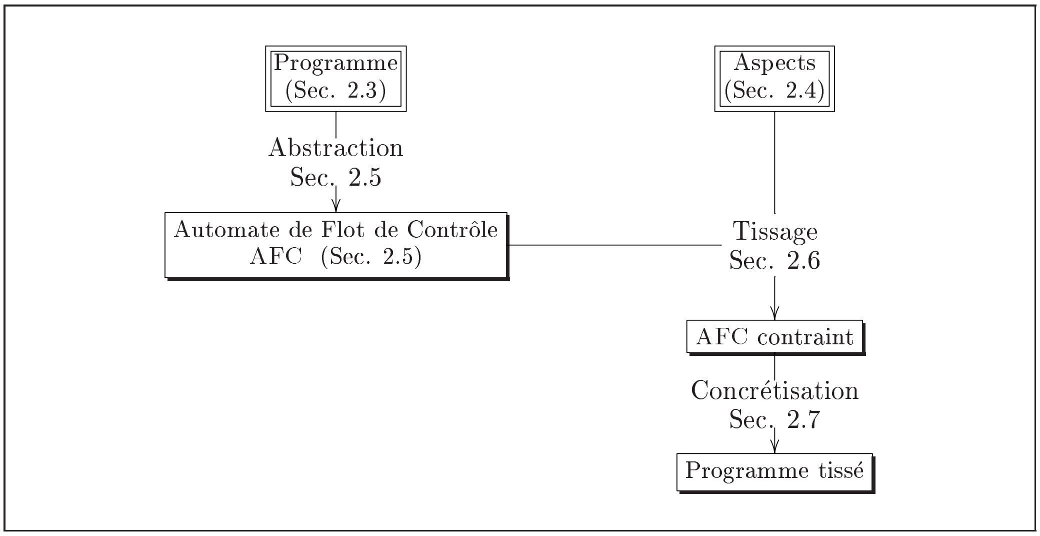


FIG. 2.1 – Les principales étapes du tissage de propriétés

ment par le produit des automates du programme et de l'aspect. La concrétisation consiste à revenir au monde des programmes en transformant l'automate produit en un programme tissé.

La section 2.2 rappelle les principales définitions et opérations des automates. Nous présentons la syntaxe et la sémantique des programmes dans la section 2.3 et les aspects sous la forme d'automates dans la section 2.4. La section 2.6 présente le tissage d'une propriété sur un programme et la section 2.7 présente la concrétisation. Nous montrons la correction de cette approche en section 2.8. Dans la section 2.9, nous présentons plusieurs extensions de cette approche. Nous illustrons les différentes étapes par des applications tirées du domaine de la sécurité.

2.2 Rappels sur les automates

Nous rappelons ici uniquement les définitions et les opérations sur les automates que nous utilisons dans la suite. Le lecteur intéressé par des informations plus complètes sur les automates pourra se reporter aux deux livres [NN92] et [HMU00]. Le formalisme des automates est simple, intuitif, facile à utiliser et à composer. Les automates sont souvent utilisés pour modéliser des systèmes qui évoluent de manière discrète (par exemple, un distributeur automatique de boissons) par opposition aux systèmes qui évoluent de manière continue (par exemple, le débit d'eau en fonction de l'ouverture d'un robinet).

2.2.1 Définitions

Un automate est un quintuplet $(\Sigma, \alpha_0, \mathcal{E}, f, \mathcal{F})$ où :

- Σ est un ensemble d'états ;
- $\alpha_0 \in \Sigma$ est l'état initial ;
- $\mathcal{E}$ est un alphabet qui définit un ensemble d'événements ;
- $f : \Sigma \times \mathcal{E} \to \Sigma$ est une relation de transition entre états selon un événement. En général, cette relation de transition est partielle sur $\mathcal{E}$;
- $\mathcal{F} \subseteq \Sigma$ est l'ensemble des états finals.

Un automate se comprend de manière intuitive comme une machine qui reconnait des suites d'événements (aussi appelées des mots). On dit qu'un automate est dans un état α (l'état courant), initialement dans l'état α_0. La notation $f(\alpha, a) = \beta$ signifie que si l'automate est dans l'état α alors il reconnait l'événement a et le nouvel état courant de l'automate devient β. L'automate se comporte comme un interprète qui effectue des transitions (*i.e.*, reconnait un événement et change d'état) et qui peut s'arrêter uniquement quand son état courant appartient à l'ensemble des états finals $\mathcal{F}$. Pour faciliter la lecture, nous notons $x \overset{a}{\to} y$ pour $f(x, a) = y$. Nous utilisons aussi la notation $\overset{w}{\to}^*$ qui définit la fermeture réflexive transitive de $\to$ étendu à un mot w. Par exemple, $\alpha \overset{ab}{\longrightarrow}^* \gamma$ signifie : $\exists \beta . f(\alpha, a) = \beta \wedge f(\beta, b) = \gamma$

La sémantique d'un automate est défini par le langage (l'ensemble des mots) qu'il *reconnait* ainsi que par le langage qu'il *accepte*. Un mot est reconnu par l'automate si cette suite d'événements correspond à une séquence de transitions à partir de l'état initial. Un mot est accepté si ce mot est reconnu et amène l'automate dans un état appartenant à l'ensemble des états finals $\mathcal{F}$. Formellement :

- Le langage reconnu par l'automate $A = (\Sigma, \alpha_0, \mathcal{E}, f, \mathcal{F})$ est défini par :
$$T(A) = \{w \in \mathcal{E}^* \mid \alpha_0 \overset{w}{\to}^* \alpha\}$$
- Le langage accepté par l'automate $A = (\Sigma, \alpha_0, \mathcal{E}, f, \mathcal{F})$ est défini par :
$$L(A) = \{w \in \mathcal{E}^* \mid \alpha_0 \overset{w}{\to}^* \alpha \wedge \alpha \in \mathcal{F}\}$$

Les automates ont l'avantage d'avoir une représentation graphique simple et suggestive. Un automate se représente graphiquement par un graphe dont les nœuds représentent les états de l'automate et les arcs représentent les transitions. Les états finals sont représentés par un double cercle et l'état initial est pointé par un arc sans état de départ.

L'automate A_1 (figure 2.2) peut reconnaitre dans l'état initial q_0 un d et rester dans ce même état, ou reconnaitre un a et passer dans l'état q_1. Dans l'état q_1, il peut de la même façon reconnaître un b et rester dans cet état, ou reconnaitre un c et revenir dans l'état q_0. Le langage reconnu par cet automate est : $T(A_1) = (ab^*cd^*)^*(a \cup \epsilon)$ [1] . Comme tous les états sont finals le langage accepté par l'automate A_1 est le même que celui reconnu ($L(A_1) = T(A_1)$).

[1] Les langages sont décrits en utilisant la syntaxe des expressions rationnelles. Un rappel de cette syntaxe peut être trouvé dans [NN92]

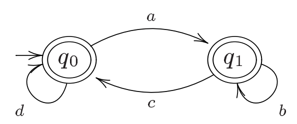


Fig. 2.2 – L'automate A_1

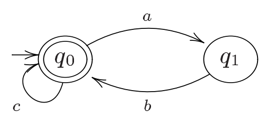


Fig. 2.3 – L'automate A_2

L'automate A_2 (figure 2.3) reconnait quant à lui le langage $T(A_2) = (c^*ab)^*(a \cup \epsilon)$ et accepte le langage $L(A_2) = (c^*ab)^*$.

Un automate est dit *déterministe* si sa relation de transition est une fonction. Dans chaque état, il n'existe pas deux transitions sortantes distinctes étiquetées par le même événement.

Définition **2.2.1.** *L'automate $A = (\Sigma, \alpha, \mathcal{E}, \rightarrow, \mathcal{F})$ est dit déterministe (et on note $Det(A)$) si et seulement si*

$$\forall \alpha_1, \alpha_2, \alpha_3 \in \Sigma, \forall a \in \mathcal{E}. \alpha_1 \xrightarrow{a} \alpha_2 \wedge \alpha_1 \xrightarrow{a} \alpha_3 \implies \alpha_2 = \alpha_3$$

2.2.2 Opérations sur les automates

Lors du tissage, nous manipulons les automates avec différentes opérations que nous présentons ici.

Accessibilité L'automate peut être simplifié sans modifier le langage qu'il reconnait et le langage qu'il accepte. Une simplification consiste à enlever tous les états qui sont inaccessibles depuis l'état initial. Nous considérons que la suppression d'un état supprime également toutes les transitions (sortantes ou sortantes) liées à cet état.

Définition **2.2.2.** *Soit l'automate $A = (\Sigma, \alpha_0, \mathcal{E}, f, \mathcal{F})$, l'automate A simplifié par l'opération d'accessibilité, noté $Ac(A)$, est dit automate accessible, et est défini par le quintuplet $(\Sigma_{ac}, \alpha_0, \mathcal{E}, f_{ac}, \mathcal{F}_{ac})$ avec :*
- $\Sigma_{ac} = \{\alpha \in \Sigma \mid \exists w \in \mathcal{E}^*. \alpha_0 \xrightarrow{w}{}^* \alpha\}$
- $\mathcal{F}_{ac} = \mathcal{F} \cap \Sigma_{ac}$

– $f_{ac}(\alpha, a) = \beta \quad si\ \mathcal{F}(\alpha, a) = \beta \wedge \alpha, \beta \in \Sigma_{ac}$
  $f_{ac}(\alpha, a) = \perp \quad sinon$

La fonction de transition f_{ac} est défini en restreignant la fonction f au domaine des états accessibles Σ_{ac}. Par la suite, nous considérons uniquement des automates A accessibles tels que $A = Ac(A)$. Par exemple, les automates A_1 et A_2 décrits précédemment sont accessibles.

Produit Un état de l'automate produit modélise l'état des deux automates comme une paire (α_a, α_b). Une transition de l'automate produit sur l'événement e est possible uniquement si les deux automates peuvent chacun prendre une transition sur l'événement e. L'alphabet du produit est défini par l'intersection $\mathcal{E}_a \cap \mathcal{E}_b$. Formellement,

Définition **2.2.3.** *Soient les automates* $A_a = (\Sigma_a, \alpha_{a0}, \mathcal{E}_a, f_a, \mathcal{F}_a)$ *et* $A_b = (\Sigma_b, \alpha_{b0}, \mathcal{E}_b, f_b, \mathcal{F}_b)$*, le produit* $A_a \times A_b$ *est l'automate* $(\Sigma_a \times \Sigma_b, (\alpha_{a0}, \alpha_{b0}), \mathcal{E}_a \cap \mathcal{E}_b, f, \mathcal{F}_a \times \mathcal{F}_b)$*, avec :*

$$\begin{array}{ll} f((\alpha_a, \alpha_b), e) = (\alpha'_a, \alpha'_b) & si\ f_a(\alpha_a, e) = \alpha'_a \wedge f_b(\alpha_b, e) = \alpha'_b \\ f((\alpha_a, \alpha_b), e) = \perp & sinon \end{array}$$

Le langage accepté (respectivement le langage reconnu) de l'automate $A_a \times A_b$ est l'intersection des langages acceptés (respectivement des langages reconnus) des deux automates A_a et A_b. Formellement,
– $T(A_a \times A_b) = T(A_a) \cap T(A_b)$
– $L(A_a \times A_b) = L(A_a) \cap L(A_b)$
Notons aussi que le produit d'automates est une opération commutative et associative.

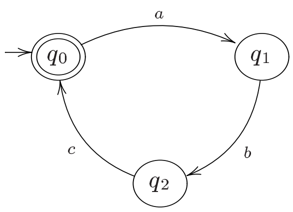


Fig. 2.4 – Produit d'automates $A_1 \times A_2$

La figure 2.4 montre le résultat du produit entre les automates A_1 et A_2. Cet automate accepte l'intersection entre les langages $L(A_1)$ et $L(A_2)$. Il accepte des séquences d'un a suivi d'un b suivi d'un c ; c'est à dire $L(A_1 \times A_2) = (abc)^*$ Cet automate reconnait le langage $T(A_1 \times A_2) = (abc)^*(a \cup ab \cup \epsilon)$.

Cette opération (et ses variantes) représente le tissage dans notre approche. En représentant le programme par un automate acceptant les traces d'exécutions du programme et un aspect par un automate sélectionnant les traces autorisées, alors l'automate produit représente un programme respectant la propriété (les traces non autorisées se sont pas dans l'intersection de l'automate produit).

2.3 Les programmes

Nous présentons maintenant la syntaxe et la sémantique du langage de programmation utilisé dans ce document. Il s'agit d'un langage simple, proche d'un code trois adresses, dont le principal avantage est qu'un programme est proche de son graphe de flot de contrôle. Cette propriété rend l'abstraction d'un programme en son graphe de flot de contrôle ainsi que la concrétisation d'un automate en programme particulièrement simples. Ce langage est restreint à des conditionnelles et des opérations qui manipulent la mémoire (par exemple, les affectation ou les opérations arithmétiques). Nous donnons la sémantique de ce langage sous la forme d'une sémantique opérationnelle petits pas exprimée classiquement par un ensemble de règles.

Nous considérons dans ce chapitre des programmes séquentiels et déterministes. Dans le chapitre 3, nous étendons ce modèle à des réseaux de programmes concurrents et ajoutons des instructions de communication. Dans le chapitre 4, nous introduisons des instructions supplémentaires pour gérer le temps et la disponibilité.

2.3.1 Syntaxe

Un programme est défini par un couple (l_0, E) où l_0 dénote une *étiquette d'entrée* et $E = \{I_1, \ldots, I_n\}$ un *ensemble d'instructions* de la forme

$$I \quad ::= \quad l_1 : c \rightsquigarrow l_2 \quad | \quad l_1 : g \rightsquigarrow l_2 \ , \ l_3$$

où l_1, l_2 et l_3 sont des *étiquettes*, c une *commande* (par exemple, $x := y * z$) et g un *test* (par exemple, $x > y + z$). Par la suite, on utilise le terme *action* pour désigner indifféremment une commande ou un test. Intuitivement, si le point de programme courant est l_1 et le programme P contient :

- l'instruction $l_1 : c \rightsquigarrow l_2$ alors la commande c est exécutée et le point de programme courant devient l_2,
- l'instruction $l_1 : g \rightsquigarrow l_2 \ , \ l_3$ alors si le test g est vrai le point de programme courant devient l_2 sinon il devient l_3.

Notre langage de programme est proche de langages bas niveaux de type bytecode, assembleur ou code trois adresses (un langage intermédiaire utilisé par les compilateurs). Ce langage est assez expressif même s'il n'est pas possible d'y représenter des instructions qui n'ont pas un flot de contrôle explicite (comme les appels de fonction via l'utilisation de pointeurs de fonction). Néanmoins de telles extensions seraient possibles en utilisant des analyses statiques de flot de contrôle [NNH99]. Les analyses peuvent cependant être trop imprécises ou conduire à une explosion d'états de la représentation par des automates.

$$P = \left\{ \begin{array}{lcll} l_0 & : & \texttt{directeur()} & \leadsto & l_1 \\ l_1 & : & \texttt{G1} & \leadsto & l_2\ ,\ l_3 \\ l_2 & : & \texttt{comptable()} & \leadsto & l_3 \\ l_3 & : & \texttt{G2} & \leadsto & l_4\ ,\ l_6 \\ l_4 & : & \texttt{achat()} & \leadsto & l_5 \\ l_5 & : & \texttt{directeur()} & \leadsto & l_6 \\ l_6 & : & \texttt{comptable()} & \leadsto & l_7 \\ l_7 & : & \texttt{achat()} & \leadsto & l_0 \end{array} \right\}$$

FIG. 2.5 – Le programme `P`

La figure 2.5 donne un exemple de programme exprimé dans cette syntaxe. Nous prendrons cet exemple tout au long de ce chapitre pour illustrer les différentes étapes de notre approche. Le programme `P` est une boucle infinie qui exécute les instructions `directeur()`, `comptable()` et `achat()` qui peuvent être vues comme des appels de procédures ou méthodes externes. Le programme commence par exécuter l'instruction `comptable()`, puis si la garde $G1$ est vraie exécute `directeur()`, ensuite si la garde $G2$ est vraie exécute `achat()` suivi de `directeur()`, et finalement exécute `comptable()` suivi de `achat()`. Nous ne décrivons pas plus ici les gardes et les instructions. Seul le graphe de flot de contrôle de ce programme est utile à nos fins.

Les programmes que nous considérons représentent des programmes déterministes et séquentiels. Pour cela nous posons les restrictions syntaxiques suivantes :
- Les étiquettes de gauche des instructions sont uniques ;
- toutes les étiquettes utilisées dans le programme sont définies dans la partie gauche d'une unique instruction.

2.3.2 Sémantique

Le déterminisme des programmes implique qu'à partir d'un état initial donné une seule exécution est possible. La sémantique des programmes est donnée par une fonction rendant pour chaque état initial la trace d'exécution du programme. Une exécution du programme $P = (l_0, E)$ à partir d'une mémoire initiale s_0 est exprimée par un LTS [1] $(\Sigma_p, (l_0, s_0), \mathcal{E}, \longrightarrow_p)$ où :
- Σ_p est un ensemble (infini) de configurations (l, s) avec l une étiquette et s une mémoire. La mémoire s associe des valeurs à l'ensemble des variables

[1] *Labeled Transition System* en anglais et système de transitions étiquetées en français

du programme ;
- (l_0, s_0) est la configuration initiale. La mémoire s_0 représente les valeurs initiales des variables du programme ; (qui incluent les paramètres fournies par l'utilisateur) ;
- $\mathcal{E}$ représente toutes les actions du programme P. Cet alphabet est constitué de symboles c, g ou $\neg g$ où c représente l'exécution d'une commande, g l'exécution d'un test vrai, et $\neg g$ l'exécution d'un test faux ;
- $\longrightarrow_p$: $\Sigma_p \times \mathcal{E} \rightarrow \Sigma_p$ est une fonction de transition sur les configurations étiquetée par l'action courante ;

La sémantique des commandes c est donnée par la fonction $\mathcal{C}[\![c]\!]$ qui prend la mémoire courante et rend la mémoire modifiée par la commande. La sémantique des tests est donnée par la fonction $\mathcal{G}[\![g]\!]$ qui prend la mémoire courante et rend un booléen. De la même façon que nous ne décrivons pas de syntaxe particulière pour les actions, nous ne décrivons pas plus ces fonctions sémantiques. Le lecteur intéressé pourra trouver dans [NN92] la syntaxe et la sémantique complète de tels langages de commandes et d'expressions booléennes.

La fonction de transition $\longrightarrow_p$ est définie par les trois règles suivantes :

$$\textsc{Comm} \quad \frac{l_1 : c \rightsquigarrow l_2 \in E \quad \mathcal{C}[\![c]\!]s_1 = s_2}{(l_1, s_1) \xrightarrow{c}_p (l_2, s_2)}$$

$$\textsc{Then} \quad \frac{l_1 : g \rightsquigarrow l_2 \ , \ l_3 \in E \quad \mathcal{G}[\![g]\!]s_1}{(l_1, s_1) \xrightarrow{g}_p (l_2, s_1)}$$

$$\textsc{Else} \quad \frac{l_1 : g \rightsquigarrow l_2 \ , \ l_3 \in E \quad \neg\mathcal{G}[\![g]\!]s_1}{(l_1, s_1) \xrightarrow{\neg g}_p (l_3, s_1)}$$

La première règle décrit l'exécution d'une commande c. L'instruction $l_1 : c \rightsquigarrow l_2$ change la mémoire initiale s_1 par $\mathcal{C}[\![c]\!]s_1$ et l'étiquette courante l_1 par l_2. Les deux règles suivantes décrivent l'exécution d'un test g. La branche du test qui est prise dépend de $\mathcal{G}[\![g]\!]s_1$ et la mémoire courante reste inchangée. Comme le service est composé d'instructions dont l'étiquette gauche est unique, une seule commande s'applique à chaque étape. Comme de plus $\mathcal{C}[\![.]\!]$ et $\mathcal{G}[\![.]\!]$ sont des fonctions, l'exécution est déterministe.

La sémantique de traces d'un programme est formalisée par la fonction Trace$[\![]\!]$ qui prend une mémoire initiale s_0 et retourne l'ensemble des traces partielles du Lts $(\Sigma_p, (l_0, s_0), \mathcal{E}, \longrightarrow_p)$. Elle est définie formellement par :

$$\text{Trace}[\![P]\!]s_0 = \{(l_0, s0) \xrightarrow{a_0}_p \ldots \alpha_i \xrightarrow{a_i}_p \ldots \xrightarrow{a_{n-1}}_p \alpha_n\}$$

La figure 2.6 donne le début d'une trace d'exécution du programme `P` pour une mémoire initiale s_0 (qui modélise les entrées du programme). Cette trace

$$(l_0, s_0) \xrightarrow{comptable()} (l_1, s_1) \xrightarrow{\neg G1} (l_3, s_1) \xrightarrow{G2} (l_4, s_1) \xrightarrow{achat()} (l_5, s_2) \dashrightarrow$$

FIG. 2.6 – Exemple d'une trace du programme P

commence par l'exécution de la commande *comptable*() qui retourne une nouvelle mémoire $s_1 = \mathcal{C}[\![comptable()]\!]s_0$. Le test G_1 est évalué à faux ($\mathcal{G}[\![G1]\!]s_1 = \textbf{false}$) et le test G_2 à vrai ($\mathcal{G}[\![G2]\!]s_1 = \textbf{true}$). Ensuite, la commande *achat*() est exécutée et la nouvelle mémoire est $s_2 = \mathcal{C}[\![achat()]\!]s_1$.

2.4 Les aspects

Nos aspects décrivent des propriétés sur les traces d'exécution des programmes. Plus précisément, les aspects sont des propriétés de sûreté qui définissent l'ensemble des traces autorisées. Pour rappel, une propriété de sûreté énonce qu'un "événement négatif" n'arrive jamais, par opposition à une propriété de vivacité qui énonce qu'un "événement positif" arrivera un jour. Les propriétés de sûreté sont telles que si une trace d'exécution ne respecte par la propriété, alors il n'est pas possible de l'étendre pour respecter de nouveau la propriété. La contraposée est que si une trace vérifie la propriété, alors tous les préfixes de cette trace vérifient aussi la propriété. Le point important est qu'au contraire des propriétés de vivacité les propriétés de sûreté peuvent être imposées par un moniteur [Sch00].

Dans notre approche, un aspect définit une propriété de sûreté qui sélectionne les traces autorisées ou coupe les traces avant la violation (l'événement négatif) de la propriété. Dans le cadre de programmes déterministes et séquentiels où la sémantique est donnée par une unique trace linéaire, cela revient à arrêter le programme juste avant que la propriété soit violée. Dans un cadre de programmes parallèles (chapitre 3) où la sémantique est donnée par l'arbre des traces possibles, cela revient à sélectionner une ou des traces autorisées, ou à arrêter le programme (quand aucune trace ne convient). Dans le chapitre 4, nous considérons des propriétés temporelles ; l'aspect peut également insérer des délais. Toutefois, en considérant que le programme de base peut avoir tous les comportements temporels possibles, l'insertion de délais peut être vue comme une sélection de trace.

Dans ce chapitre, nous considérons un modèle de points de jonction simple qui caractérise uniquement l'exécution d'une instruction et qui ne contient aucune donnée dynamique. Dans la section 2.9.3, nous étendons ce modèle pour gérer des données numériques et être ainsi plus proche des modèles usuels de points de jonction. Nous nous limitons aussi dans ce chapitre à des aspects qui arrêtent l'exécution des programmes (insert de type `abort`). Dans la section 2.9.4 nous décrivons comment tisser des aspects généraux de type spectateur (qui observent sans changer le flot de contrôle du programme), et dans la section 2.9.5 nous évoquons la modélisation dans notre cadre d'un langage d'aspects classique (avec des inserts before et after arbitraires).

2.4.1 Encodage de la propriété par un automate

Un aspect définit une propriété de sûreté sur les traces d'exécution du programme. Cette propriété de sûreté est régulière (*i.e.*, l'ensemble des traces est régulier) et peut se représenter par un automate à états finis. Cet automate accepte toutes les traces d'exécutions qui vérifient la propriété. La violation de la propriété est modélisée par une transition dans un état non final.

Un aspect est un automate $(\Sigma_\lambda, \lambda_0, \mathcal{E}, \longrightarrow_\lambda, \mathcal{F}_\lambda)$ où :
- Σ_λ est un ensemble d'états ;
- λ_0 est l'état initial ;
- $\mathcal{E}$ est le même alphabet que celui des traces des programmes ;
- $\longrightarrow_\lambda$ est une fonction de transition étiquetée entre les états.
- $\mathcal{F}_\lambda$ est l'ensemble des états finals.

Les propriétés de sûreté étant closes par préfixe, les automates (aspects) sont tels que tous les états sont finals sauf un ensemble d'états correspondant à une violation de la propriété. Ces états non finals sont des états puits : ils ne peuvent pas conduire à un état final. Pour modéliser une violation de la propriété, nous considérons un unique état non final que nous nommons λ_{err} ($\Sigma_\lambda = \mathcal{F}_\lambda \cup \{\lambda_{err}\}$). Cet état ne reconnait aucun symbole. L'alphabet de l'automate est le même que celui du programme ($\mathcal{E}$). Nous considérons que l'automate d'un aspect est complet. C'est à dire que pour chaque état final la fonction de transition est définie sur tous les événements du programme. Cette modélisation permet que toute trace du programme est soit acceptée par l'automate, soit amène dans l'état non final (une violation a été détectée).

Nous utilisons pour décrire les automates d'aspects un événement particulier $\star$ qui modélise tous les événements du programme sans intérêt pour la propriété. En règle générale, une boucle annotée par $\star$ est présente sur tous les états finals de l'automate ; elle représente l'attente d'un événement d'intérêt par l'aspect. Nous posons aussi la contrainte que l'automate d'un aspect est déterministe (comme défini dans la propriété 2.2.1). Sans cela, le tissage pourrait retourner un programme non déterministe (par exemple, qui de façon non déterministe termine normalement ou est interrompu).

Nous illustrons les automates d'aspect par une propriété de sécurité. Notre exemple exprime une politique de séparation des responsabilités qui oblige à avoir l'accord du comptable et du directeur avant de faire un achat. Cet exemple s'applique sur le programme de la figure 2.5 et impose d'appeler les procédures comptable() et directeur() avant d'appeler achat(). Notons que le programme P peut violer cette propriété si le test G_1 est évalué est à faux et le test G_2 à vrai. L'aspect de la figure 2.7 décrit cette propriété de séparation des responsabilités. Pour faciliter la lecture, les instructions comptable(), directeur(), et achat() sont

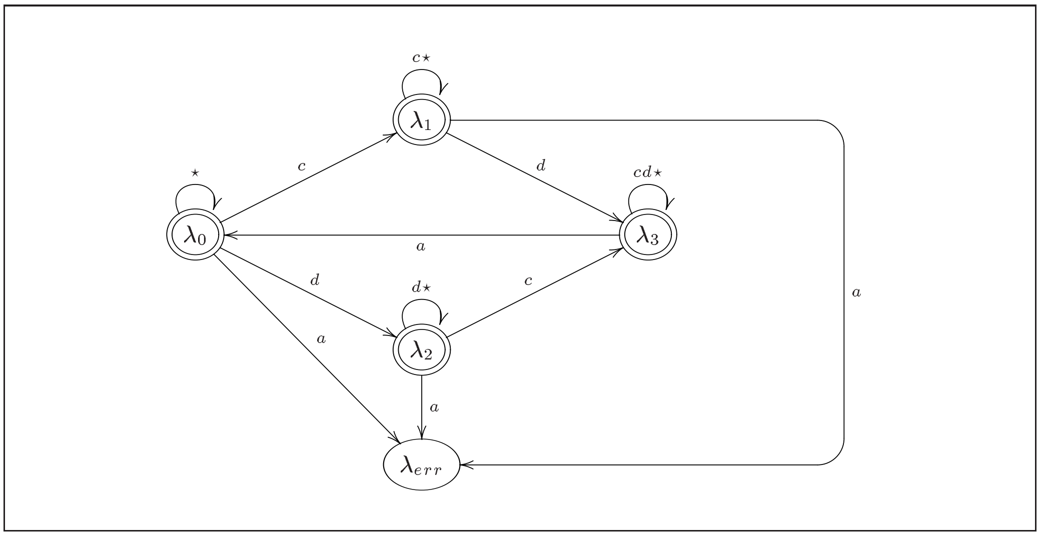


FIG. 2.7 – Automate A d'un aspect pour la séparation des responsabilités

notées respectivement c, d, et a. L'aspect commence dans l'état λ_0 et reconnait l'instruction c (resp. d) en passant à l'état λ_1 (resp. λ_2). Ensuite, dans l'état λ_1 (resp. λ_2) l'aspect reste dans le même état sur l'instruction c (resp. d) et passe dans l'état λ_3 sur l'instruction d (resp. c). L'état λ_1 (resp. λ_2) modélise que l'instruction c (resp. d) a déjà été exécutée. L'état λ_3 modélise que les instructions c et d ont été exécutées et qu'en conséquence un appel à a est autorisée. Au contraire, dans les états λ_0, λ_1 et λ_2 il est interdit d'exécuter a pour ne pas violer la politique de sécurité. En cas de violation de la propriété, l'aspect passe dans l'état puits non final λ_{err}. De manière générale, la programmation par aspect est un atout particulièrement intéressant dans le domaine de la sécurité. Il est difficile de prévoir toutes les attaques et les programmes doivent pouvoir être corrigés rapidement pour contrer les nouvelles attaques identifiées.

2.4.2 Langage d'aspects

L'aspect définit une propriété qui est représentée par un automate à états finis. En pratique, il est préférable de fournir à l'utilisateur un langage de plus haut niveau pour décrire les aspects. Le langage d'aspects peut aussi fournir une spécification de la propriété qui sera ensuite compilée par un automate. Par exemple, la propriété peut être spécifiée par une expression régulière.

La politique de la séparation des responsabilités de l'exemple précédent est décrite par l'expression régulière suivante :

$$((cc^*d|dd^*c)(c|d)^*a)^*$$

La transformation d'une expression régulière en un automate fini est une transformation bien connue [HMU00]. Une étape supplémentaire est nécessaire pour compléter l'automate et gérer les violations de la propriété. Cette étape consiste à ajouter à chaque état une boucle annotée par $\star$, puis à ajouter un état puits non final et des transitions vers cet état d'erreur lorsqu'un état n'a pas de transition sortante sur un symbole. Ce langage d'aspects basé sur les expressions régulières reste théorique. Pour le rendre utilisable, il conviendrait d'avoir un langage de points de coupure plus expressif. D'autres langages d'aspects pourraient être définis par une spécification de la propriété qui serait ensuite compilée en un automate. Par exemple, la propriété pourrait se spécifier dans un langage inspiré des formalismes logiques LTL [Eme90] ou WS1S [KS99]. Au chapitre 3, nous proposons un ensemble de directives pour décrire succinctement des ensembles de points de coupure. Nous proposons dans le chapitre 4 un langage d'aspect où l'automate est décrit de manière textuelle.

2.5 Abstraction des programmes

Le tissage nécessite d'abstraire le programme en son graphe de flot de contrôle représenté par un automate. L'abstraction permet de représenter de manière finie l'ensemble des traces d'exécution du programme. Cette abstraction produit un automate de flot de contrôle (AFC) qui accepte toutes les traces d'exécution du programme. Notre approche laisse le choix de la précision de l'abstraction tant que l'AFC respecte deux propriétés clefs.

2.5.1 Définition d'un automate de flot de contrôle (AFC)

Un AFC est un automate fini $p^\sharp = (\Sigma_{p^\sharp}, \alpha_0^\sharp, \mathcal{E}, \longrightarrow_{p^\sharp}, \mathcal{F}_{p^\sharp})$ où l'alphabet $\mathcal{E}$ est le même que celui des traces du programme P qu'il représente. Tous les états des AFC sont finals ($\mathcal{F}_{p^\sharp} = \Sigma_{p^\sharp}$). Le langage accepté correspond à l'ensemble des traces partielles du programme.

Une abstraction résume dans chaque état abstrait de $\Sigma_{p^\sharp}$ un ensemble (potentiellement infini) de configurations concrètes du programme P correspondant. Cette idée est formalisée par une relation $\approx_{abs}$ entre les configurations du programme p et les états de l'AFC correspondant $p^\sharp$ où $\alpha \approx_{abs} \alpha^\sharp$ signifie que la configuration α est représentée par l'état abstrait $\alpha^\sharp$.

Pour simplifier l'écriture des propriétés concernant des automates représentant des programmes, nous utilisons $\xrightarrow{a}$, $\xrightarrow{c}$ et $\xrightarrow{g}$ pour dénoter respectivement une transition quelconque, une transition étiquetée par une commande et une transition étiquetée par une garde.

2.5.2 Sûreté et fidélité d'un AFC

Notre approche laisse le choix de l'abstraction tant que l'AFC respecte deux propriétés clefs. La correction du tissage, qui est détaillée en section 2.8, repose sur ces deux propriétés.

Sûreté de l'abstraction

Un AFC est dit *sûr*, si toutes les configurations initiales possibles du programme sont approximées par l'état abstrait initial. De plus, si une configuration α_1 est approximée par $\alpha_1^\sharp$ et que le programme évolue de α_1 dans la configuration α_2, alors il existe une transition de $\alpha_1^\sharp$ dans $\alpha_2^\sharp$ dans l'AFC telle que α_2 est approximée par $\alpha_2^\sharp$. Cette propriété est décrite formellement par :

Définition **2.5.1.** *Soit un programme $p = (l_0, E)$ et une abstraction $p^\sharp = (\Sigma_{p^\sharp}, \alpha_0^\sharp, \mathcal{E}, \longrightarrow_{p^\sharp}, \mathcal{F}_{p^\sharp}))$, alors $p^\sharp$ est une abstraction* sûre *de p si et seulement si*

$$\begin{array}{ll} & \forall s_0, ((s_0, l_0) \xrightarrow{a_0}_p \ldots \alpha_i \xrightarrow{a_i}_p \ldots \xrightarrow{a_{n-1}}_p \alpha_n) \in \mathit{Trace}[\![p]\!]s_0. \\ & (s_0, l_0) \approx_{abs} \alpha_0^\sharp \\ \wedge & (\alpha_i \approx_{abs} \alpha_i^\sharp \wedge \alpha_i \xrightarrow{a_i}_p \alpha_j \Rightarrow \exists \alpha_j^\sharp.\ \alpha_j \approx_{abs} \alpha_j^\sharp \wedge \alpha_i^\sharp \xrightarrow{a_i}_{p^\sharp} \alpha_j^\sharp) \end{array}$$

Cette propriété formalise que les AFC sont des sur-approximations des programmes. Naturellement ils incluent aussi des faux chemins (traces abstraites ne représentant aucune trace réelle). Par exemple, si dans un programme la même garde G est testée deux fois à la suite, à l'exécution cette garde serait évaluée deux fois à **true** ou deux fois à **false**. Une trace abstraite où la transition G est suivie de $\neg G$ est un faux-chemin.

La sûreté de l'abstraction assure que toute trace d'exécution du programme p est simulée par une trace de l'AFC $p^\sharp$. Cette propriété se prouve facilement par une induction sur la taille des préfixes de la trace.

Propriété **2.5.2.** $\forall w.\alpha_0 \xrightarrow{w}_p^* \alpha \implies \alpha_0^\sharp \xrightarrow{w}_{p^\sharp}^* \alpha^\sharp$

Fidélité de l'abstraction

Un AFC est dit *fidèle* si chaque transition abstraite correspond à une transition concrète modulo la non-satisfiabilité des gardes. Formellement :

Définition **2.5.3.** *Soit un programme $p = (l_0, E)$ et une abstraction $p^\sharp = (\Sigma_{p^\sharp}, \alpha_0^\sharp, \mathcal{E}, \longrightarrow_{p^\sharp}, \mathcal{F}_{p^\sharp}))$, $p^\sharp$ est une abstraction* fidèle *de p si et seulement si*

$$\begin{array}{ll} & \forall s_0. \exists ((l_0, s_0) \xrightarrow{a_0}_p \ldots \alpha_i \xrightarrow{a_i}_p \ldots \xrightarrow{a_{n-1}}_p \alpha_n) \in \mathit{Trace}[\![p]\!]s_0. \\ & (\alpha_i^\sharp \xrightarrow{c}_{p^\sharp} \alpha_j^\sharp \ \wedge\ \alpha_i \approx_{abs} \alpha_i^\sharp \Rightarrow \alpha_j \approx_{abs} \alpha_j^\sharp \ \wedge\ \alpha_i \xrightarrow{c}_p \alpha_j) \\ \wedge & (\alpha_1^\sharp \xrightarrow{g}_{p^\sharp} \alpha_2^\sharp \ \wedge\ \alpha_1 \approx_{abs} \alpha_1^\sharp \Rightarrow (\mathcal{G}[\![g]\!]\alpha_1 \Rightarrow \alpha_2 \approx_{abs} \alpha_2^\sharp \ \wedge\ \alpha_1 \xrightarrow{g}_p \alpha_2)) \end{array}$$

La propriété de fidélité confine l'approximation aux valeurs des variables. Avec cette propriété, un faux chemin est une trace abstraite avec une transition dont l'image concrète correspond à une garde fausse. La fidélité interdit, par exemple, l'abstraction (très imprécise mais sûre) constituée d'un unique état abstrait représentant toutes les configurations concrètes et où toutes les transitions partent et arrivent à cet état. En pratique, la fidélité interdit qu'un même état abstrait représente des points de programme différents.

Les approximations sûres et fidèles sont des approximations où les nœuds des AFCs représentent les étiquettes des programmes et les étiquettes des arcs représentent l'exécution d'une action.

Automate bien formé

L'abstraction d'un programme séquentiel et déterministe et produit des AFCs avec une certaine structure. Formellement :

DÉFINITION **2.5.4.** *L'automate $A = (\Sigma, \alpha, \mathcal{E}, \rightarrow, \mathcal{F})$ est dit bien formé, et on note WF(A), si et seulement si*

$$\begin{array}{ll} & \forall \alpha_1, \alpha_2, \alpha_3 \in \Sigma. \\ & \alpha_1 \xrightarrow{a_1} \alpha_2 \wedge \alpha_1 \xrightarrow{a_2} \alpha_3 \implies ((a_1 = a_2 \wedge \alpha_2 = \alpha_3) \vee (a_1 = g \wedge a_2 = \neg g)) \\ \wedge & \alpha_1 \xrightarrow{g} \alpha_2 \implies \exists \alpha_3 . \alpha_1 \xrightarrow{\neg g} \alpha_3 \end{array}$$

où $\neg g$ dénote l'alternative au test g ($\neg(\neg g) = g$).

Chaque état a soit une transition sortante étiquetée par une commande c, soit deux transitions sortantes étiquetées par g et $\neg g$, soit aucune transition sortante. Cette propriété est utile pour revenir à un programme (concrétisation).

2.5.3 Exemple d'abstraction

Nous proposons maintenant une abstraction qui respecte les trois propriétés précédentes. Cette abstraction ne considère pas les valeurs des variables et abstrait simplement le programme par un automate qui correspond à son graphe de flot de contrôle.

Un programme $P = (l_0, E)$ est *abstrait* par l'AFC $(\Sigma_{p^\sharp}, \alpha_0^\sharp, \mathcal{E}, \longrightarrow_{p^\sharp}, \mathcal{F}_{p^\sharp})$ où :

- les états abstraits $\Sigma_{p^\sharp}$. représentent les étiquettes du service. Formellement : $\Sigma_{p^\sharp} = \{\alpha_i^\sharp \mid l_i$ est une étiquette de $P\}$;
- l'état initial abstrait $\alpha_0^\sharp$ représente l'étiquette initiale l_0 ;
- l'ensemble des actions $\mathcal{E}$ est l'ensemble des actions de P ;
- la relation de transition $\longrightarrow_{p^\sharp}$ est définie par :

$$\begin{array}{lll} \alpha_1^\sharp \xrightarrow{c}_{p^\sharp} \alpha_2^\sharp & ssi & l_1 : c \rightsquigarrow l_2 \in E \\ \alpha_1^\sharp \xrightarrow{g}_{p^\sharp} \alpha_2^\sharp & ssi & l_1 : g \rightsquigarrow l_2\ ,\ l_3 \in E \\ \alpha_1^\sharp \xrightarrow{\neg g}_{p^\sharp} \alpha_3^\sharp & ssi & l_1 : g \rightsquigarrow l_2\ ,\ l_3 \in E \end{array}$$

– $\mathcal{F}_{p^\sharp} = \Sigma_{p^\sharp}$.

La relation $\approx_{abs}$ est définie par $(l_i, s) \approx_{abs} \alpha_i^\sharp$.

La figure 2.8 illustre cette abstraction sur le programme P de la figure 2.5 Nous prouvons dans l'annexe **??** que cette fonction d'abstraction respecte les trois

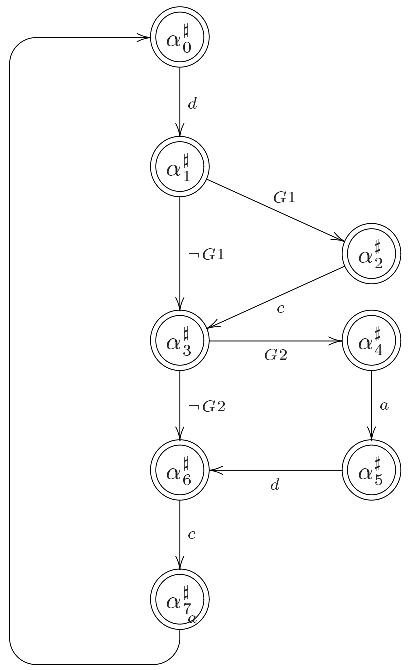


FIG. 2.8 – Abstraction du programme P en l'automate $P^\sharp$

propriétés précédentes. Ces preuves, très simple, utilisent les définitions de l'abstraction et de la sémantique d'un programme, et le fait qu'une action est soit une commande soit un test.

Bien d'autres abstractions que celle proposée sont possibles. L'intérêt d'utiliser une autre abstraction est d'optimiser le tissage et de produire un programme tissé plus performant. Par exemple, il peut être intéressant que l'abstraction déplie les boucles quand le nombre d'itération est connu afin de contenir moins de tests et faux-chemins. De manière générale, des abstraction plus précises produisent des AFCs de plus grandes tailles mais qui pourront être tissés plus efficacement.

2.6 Tissage de propriétés

Dans le vocabulaire de la programmation par aspects, le tissage est l'opération qui ajoute le code de l'aspect au programme. Dans notre approche, le tissage consiste à faire le produit (comme décrit en section 2.2.2) de l'AFC représentant le programme et de l'automate représentant l'aspect. L'impact sémantique de ce tissage est de restreindre le comportement du service en sélectionnant les traces autorisées par l'aspect. La simplicité pratique et théorique de l'opération de tissage est l'un des principaux avantages de notre approche.

Le produit de l'AFC et de l'aspect fait l'intersection des traces d'exécution et impose les contraintes spécifiées par l'aspect. En pratique, le produit d'automates (*i.e.,* le tissage) modifie l'AFC en coupant (resp. sélectionnant) les traces interdites (resp. autorisées) par l'aspect. Soit l'AFC $P^\sharp$ qui correspond au programme et l'automate A qui correspond à l'aspect, le tissage de l'aspect sur le programme est obtenu par le produit $P^\sharp \times A$.

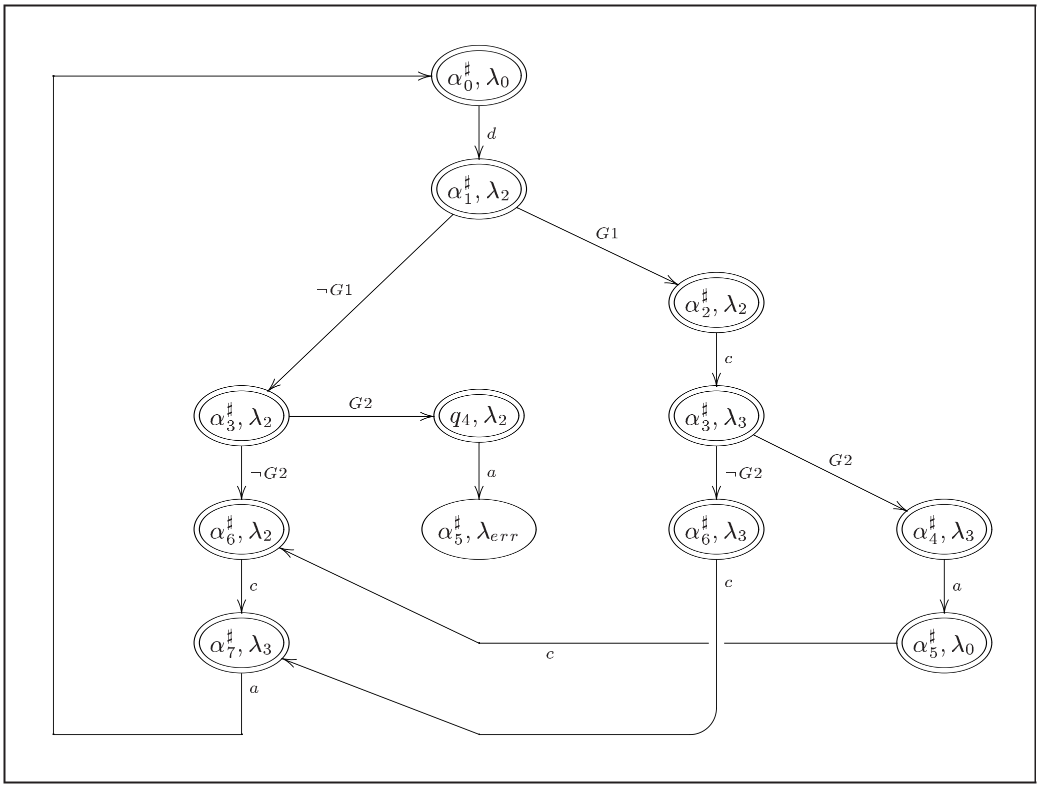


FIG. 2.9 – Tissage entre l'AFC du programme `P` et de l'aspect A

La figure 2.9 montre le produit de l'abstraction du programme $P^\sharp$ (figure 2.8) et de l'aspect A (figure 2.7). Comme tous les états des AFC sont finals, l'état non final de l'automate résultant (*i.e.,* le programme tissé) correspond à une violation de la propriété comme dans l'aspect. Notons aussi que certaines transitions de l'AFC se retrouvent dupliquées ; l'automate résultat possède 14 transitions contre 10 initialement. En général, cette duplication peut poser problème car le nombre d'états de l'automate produit est de l'ordre de grandeur $o(n * m)$ avec n la taille du programme initial et m la taille de l'aspect. En fonction de la taille des automates, ceci peut causer une explosion du nombre d'états. Ce produit peut également s'implémenter sans duplication d'états comme décrit dans l'extension 2.9.2. L'idée est d'instrumenter l'AFC avec une variable supplémentaire (qui représentera l'état de de l'aspect) et des affectations (qui implémentent les transitions d'état de l'aspect). On évite ainsi une augmentation quadratique de la taille des programmes en acceptant de payer un léger surcoût en temps.

Comparé à un tissage plus classique à la AspectJ, le résultat final est similaire : le programme est instrumenté et est interrompu quand certaines conditions sont remplies. Les approches sont en revanche totalement différentes. Dans AspectJ, la conception et le raisonnement sont principalement de nature syntaxique. Les aspects spécifient des ensembles de point de programmes et le code à insérer à ces points. Le programmeur raisonne sur l'impact des aspects en visualisant l'instrumentation attendue et le texte du programme tissé. Dans notre approche, l'aspect est vu comme une propriété sur les traces d'exécution. L'aspect spécifie un ensemble de traces autorisées qui peut être imposé au programme par produit d'automates. L'avantage principal de cette approche est évidemment de faire apparaître explicitement l'impact sémantique du tissage. Dans un langage d'aspects général ne faisant aucune restriction sur le code inséré, il est impossible en général de caractériser précisément l'effet des aspects sur la sémantique du programme.

Le tissage possède des propriétés importantes qui nous serviront pour montrer la correction de l'approche. Tout d'abord, une trace de l'AFC respecte la propriété si et seulement si l'automate produit accepte cette trace.

Propriété **2.6.1.** *Soit un* AFC $P^\sharp = (\Sigma_{p^\sharp}, \alpha_0^\sharp, \mathcal{E}, \longrightarrow_{p^\sharp}, \mathcal{F}_{p^\sharp})$*, un aspect* A*, et l'automate* $I = P^\sharp \times A = (\Sigma_I, q_0, \mathcal{E}, \longrightarrow_I, \mathcal{F}_I)$*, alors*

$$\alpha_0^\sharp \xrightarrow{w}_{p^\sharp}{}^* \alpha^\sharp \wedge w \in L(A) \iff w \in L(I)$$

Ensuite, si une trace de l'AFC ne respecte pas l'aspect, alors il existe une instruction de cette trace qui viole la propriété et à cette même instruction l'automate produit va en état puits non final (*i.e.,* on coupe juste avant).

Propriété **2.6.2.** *Soit un* AFC $P^\sharp = (\Sigma_{p^\sharp}, \alpha_0^\sharp, \mathcal{E}, \longrightarrow_{p^\sharp}, \mathcal{F}_{p^\sharp})$*, l'automate de l'aspect* A*, et l'automate* $I = P^\sharp \times A = (\Sigma_I, q_0, \mathcal{E}, \longrightarrow_I, \mathcal{F}_I)$*, alors*

$$\begin{gathered}\alpha_0^\sharp \xrightarrow{w}_{p^\sharp}{}^* \alpha^\sharp \wedge w \notin L(A) \\ \Downarrow \\ \exists w_1, a, w_2, q_1, q_a . w = w_1 a w_2 \wedge w1 \in L(A) \wedge w_1 a \notin L(A) \\ \wedge\ q_0 \xrightarrow{w_1}{}_I^* q_1 \xrightarrow{a}_I q_a \wedge q_a \notin \mathcal{F}_I\end{gathered}$$

Finalement, le produit d'un AFC bien formé (représentant un programme séquentiel et déterministe) et de l'automate d'un aspect (par définition déterministe) retourne un automate bien formé.

Propriété **2.6.3.** *Soit un* AFC $P^\sharp$ *et un aspect* A*, alors*

$$WF(P^\sharp) \land Det(A) \implies WF(P^\sharp \times A)$$

Ces propriétés se prouvent facilement en utilisant la définition du produit d'automates, la définition de l'abstraction (tous les états d'un AFC sont finals), et la définition des aspects (tous les états sont finals sauf un état puits qui symbolise la violation de la propriété). Les preuves sont décrites en annexe **??**.

2.7 Concrétisation

La génération d'un programme à partir d'un automate est effectuée par une fonction de concrétisation. Dans notre approche, cette fonction prend l'automate tissé (le produit d'un AFC et d'un aspect) et retourne le programme correspondant. Comme l'abstraction, la concrétisation doit être sûre et fidèle. Nous décrivons la fonction de concrétisation et montrons qu'elle respecte ces deux propriétés.

2.7.1 Fonction de concrétisation

La concrétisation utilise la commande spéciale abort. Cette commande est insérée dans le programme tissé juste avant de violer la propriété. La commande abort a pour effet de conclure la trace d'exécution par une transition vers une configuration particulière notée $\perp$. Plus formellement, le comportement de cette commande est décrit par la règle sémantique suivante :

$$\textsc{Abort} \quad \frac{l_1 : \mathsf{abort} \rightsquigarrow l_2 \in P}{(l_1, s) \xrightarrow{\mathsf{abort}}_p \perp}$$

Un automate fini bien formé $I = (\Sigma_I, q_0, \mathcal{E}, \longrightarrow_I, \mathcal{F}_I)$ est concrétisé dans le programme conc(I) avec :

$$\begin{aligned} Conc(I) = (l_{q_0}, \quad & \{l_{q_1} : c \rightsquigarrow l_{q_2} \mid q_1 \xrightarrow{c}_I q_2 \land q_2 \in \mathcal{F}_I\} \\ & \cup \{l_{q_1} : \mathsf{abort} \rightsquigarrow l_{q_2} \mid q_1 \xrightarrow{c}_I q_2 \land q_2 \notin \mathcal{F}_I\} \\ & \cup \{l_{q_1} : g \rightsquigarrow l_{q_2}\ ,\ l_{q_3} \mid q_1 \xrightarrow{g}_I q_2 \land q_1 \xrightarrow{\neg g}_I q_3\}) \end{aligned}$$

Une étiquette l_q est crée pour chaque état $q \in \Sigma_I$. La fonction *Conc* est appliquée sur tous les états pour obtenir l'ensemble des instructions du programme.

Le fait que la fonction de concrétisation s'applique uniquement à des automates bien formés (propriété 2.5.4) assure que le programme retourné est séquentiel et déterministe. Ceci est toujours le cas lorsque l'on tisse un aspect déterministe sur un AFC bien formé (*i.e.*, qui correspond à un programme déterministe) comme garanti par la propriété 2.6.3. Notons aussi que nous considérons pour simplifier qu'une transition étiquetée par un test amène toujours dans un état final.

De même que pour l'abstraction, chaque état de l'automate produit représente un ensemble (potentiellement infini) de configurations concrètes du programme tissé. Cette idée est formalisée par une relation $\approx_{conc}$ entre les états de l'automate et les configurations concrètes où $\alpha \approx_{conc} q$ signifie que la configuration α est représentée par l'état q. Par définition de la fonction de concrétisation, nous avons :

$$\forall s.(l_q, s) \approx_{conc} q$$

Une exécution du programme $z = Conc(I)$ à partir d'une mémoire initiale s_0 est définie par le LTS $(\Sigma_Z, (l_{q0}, s_0), \mathcal{E} \cup \{abort\}, \longrightarrow_Z)$. Les seules différences entre les exécutions du programme de base et celles du programme tissé concerne la gestion de l'événement abort.

La figure 2.10 montre le programme retourné par la concrétisation de l'automate 2.9. Nous avons aussi effectué un renommage des étiquettes pour simplifier les notations. Le programme tissé respecte la propriété car si la garde $G1$ est fausse

$$P = \left\{ \begin{array}{lcll} l_0 & : & \texttt{directeur()} & \rightsquigarrow & l_1 \\ l_1 & : & \texttt{G1} & \rightsquigarrow & l_2\ ,\ l_6 \\ l_2 & : & \texttt{comptable()} & \rightsquigarrow & l_3 \\ l_3 & : & \texttt{G2} & \rightsquigarrow & l_4\ ,\ l_6 \\ l_4 & : & \texttt{achat()} & \rightsquigarrow & l_5 \\ l_5 & : & \texttt{comptable()} & \rightsquigarrow & l_9 \\ l_6 & : & \texttt{comptable()} & \rightsquigarrow & l_{10} \\ l_7 & : & \texttt{G2} & \rightsquigarrow & l_8\ ,\ l_9 \\ l_8 & : & \texttt{abort()} & \rightsquigarrow & l_{erreur} \\ l_9 & : & \texttt{comptable()} & \rightsquigarrow & l_{10} \\ l_{10} & : & \texttt{achat()} & \rightsquigarrow & l_0 \end{array} \right\}$$

FIG. 2.10 – Concrétisation de l'automate produit $P \times A$

et la garde $G2$ est vraie, alors l'instruction abort est exécutée et le programme est arrêté au lieu d'effectuer achat() sans avoir exécuter comptable() comme dans le programme initial.

2.7.2 Propriétés de la concrétisation

Nous montrons maintenant que la fonction de concrétisation respecte deux propriétés qui mettent en relation l'automate produit et la sémantique du programme tissé.

Sûreté de la concrétisation

La fonction de concrétisation est dite sûre, si la configuration concrète initiale du programme tissé est approximée par la configuration abstraite initiale. De plus, les transitions de l'automate correspondent à des transition du programme tissé modulo la non-satisfiabilité des gardes et la non-violation de la propriété définie par l'aspect. Dans le cas d'une violation de l'aspect (repérée par une transition dans un état non terminal), l'instruction abort est exécutée. Formellement :

PROPRIÉTÉ **2.7.1.** *Soit l'automate $I = (\Sigma_I, q_0, \mathcal{E}, \longrightarrow_I, \mathcal{F}_I)$ et un programme $z = Conc(I)$, alors*

$$\begin{array}{ll} & \forall s_0, \exists ((l_{q0}, s_0) \xrightarrow{a_0}_Z \ldots \alpha_i \xrightarrow{a_i}_Z \ldots \xrightarrow{a_{n-1}}_Z \alpha_n) \in Trace[\![z]\!]s_0. \\ \wedge & (l_{q0}, s_0) \approx_{conc} q_0 \\ \wedge & (\alpha_i \approx_{conc} q_i \wedge q_i \xrightarrow{c}_I q_j \wedge q_j \notin \mathcal{F}_I \perp \implies \alpha_i \xrightarrow{\mathsf{abort}}_Z) \\ \wedge & (\alpha_i \approx_{conc} q_i \wedge q_i \xrightarrow{a}_I q_j \wedge q_j \in \mathcal{F}_I \wedge (a = c \vee (a = g \wedge \mathcal{G}[\![g]\!]\alpha_i)) \\ & \quad \implies \alpha_j \approx_{conc} q_j \wedge \alpha_i \xrightarrow{a}_Z \alpha_j) \end{array}$$

Cette propriété assure que le comportement de l'automate est modélisé par la sémantique du programme obtenu après la concrétisation. Cette propriété de la concrétisation met en relation les transitions de l'automate et les traces correspondant à l'exécution du programme tissé. La fonction de concrétisation transforme chaque transition de l'automate en une instruction. La preuve de cette propriété est fournie en annexe **??**.

Fidélité de la concrétisation

Une concrétisation d'un automate est dite fidèle si toutes les traces du programme tissé correspondent à des traces de l'automate. De plus si le programme tissé effectue un abort, cela implique que la propriété a été violée (l'automate va dans un état non final). Formellement :

PROPRIÉTÉ **2.7.2.** *Soit l'automate $I = (\Sigma_I, q_0, \mathcal{E}, \longrightarrow_I, \mathcal{F}_I)$ et Soit un programme $z = Conc(I)$, alors*

$$\begin{array}{ll} & \forall s_0, ((l_{q0}, s_0) \xrightarrow{a_0}_Z \ldots \alpha_i \xrightarrow{a_i}_Z \ldots \xrightarrow{a_{n-1}}_Z \alpha_n) \in Trace[\![z]\!]s_0. \\ & (\alpha_i \xrightarrow{a}_Z \alpha_j \wedge \alpha_i \approx_{conc} q_i \wedge a \neq \mathsf{abort} \implies \exists q_j \in \mathcal{F}_I.\alpha_j \approx_{conc} q_j \wedge q_i \xrightarrow{a}_I q_j) \\ \wedge & (\alpha_i \xrightarrow{\mathsf{abort}}_Z \perp \wedge \alpha_i \approx_{conc} q_i \implies \exists a \in \mathcal{E}, q_j \notin \mathcal{F}_I.q_i \xrightarrow{a}_I q_j) \end{array}$$

Les détails de la preuve de cette propriété se trouvent en annexe **??**.

2.8 Correction du tissage

Notre approche prend un programme et un aspect qui définit une propriété de sûreté et retourne un programme tissé qui respecte cette propriété. Nous montrons dans cette partie que l'abstraction, le tissage et la concrétisation tels que définis précédemment permettent d'imposer la propriété au programme. Nous commençons par définir la propriété de correction et nous en donnons ensuite la preuve en utilisant deux lemmes intermédiaires.

2.8.1 Propriété de correction du tissage

La sémantique fonctionnelle d'un programme $P = (l_0, E)$ est une fonction prenant en argument la mémoire initiale s_0 et rendant la mémoire finale. Elle est définie par :

$$SOS[\![P]\!]s_0 = \begin{cases} s & \text{si } (l_0, s_0) \xrightarrow{w}{}^{*}_{p} (l, s) \wedge \nexists \sigma, a.(l, s) \xrightarrow{a}_{p} \sigma \\ \text{err} & \text{si } (l_0, s_0) \xrightarrow{w}{}^{*}_{p} \bot \\ undef & \text{sinon} \end{cases}$$

où $\xrightarrow{w}{}^{*}_{p}$ représente la fermeture réflexive et transitive de $\rightarrow_p$ étendu à un mot w représentant la trace d'exécution. Cette sémantique donne un sens aux programmes qui terminent et est moins précise que la sémantique de traces (identifie des programmes différents pour autant qu'ils donnent le même résultat final).

Informellement le programme tissé doit avoir le comportement suivant :
- si le programme de base respecte la propriété de l'aspect, alors le programme tissé doit avoir le même comportement ;
- si le programme ne respecte pas la propriété, le programme tissé a le même comportement mais s'arrête (`abort`) juste avant de violer la propriété.

Formellement, la sémantique d'un programme P qui respecte la propriété spécifiée par l'automate A est définie par :

$$SOS^A[\![P]\!]s_0 = \begin{cases} s & \text{si } (l_0, s_0) \xrightarrow{w}{}^{*}_{p} (l, s) \wedge \nexists \sigma, a.(l, s) \xrightarrow{a}_{p} \sigma \wedge w \in L(A) \\ \text{err} & \text{si } (l_0, s_0) \xrightarrow{w}{}^{*}_{p} (l, s) \xrightarrow{a}_{p} \sigma \wedge w \in L(A) \wedge wa \notin L(A) \\ \text{undef} & \text{sinon} \end{cases}$$

Nous pouvons maintenant exprimer la propriété de correction de l'approche.

Propriété **2.8.1.** [**Correction du tissage**] *Soit le programme de base P, $P^\sharp$ une abstraction* sûre *et* fidèle *de P, et l'automate de l'aspect A, le tissage est correct si et seulement si :*

$$\forall s_0 . SOS[\![Conc(P^\sharp \times A)]\!]s_0 = SOS^A[\![P]\!]s_0$$

La preuve de cette propriété repose sur deux lemmes intermédiaires, décrit ci-dessous, que nous appelons les propriétés de sûreté et de fidélité du tissage. Les différentes preuves sont fournies en annexe **??**.

Propriété **2.8.2.** [**Sûreté du tissage**] *Soit le programme de base P, $P^\sharp$ une abstraction* sûre *et* fidèle *de P, et l'automate de l'aspect A, on dit que le tissage est* sûr *si et seulement si :*

$$\begin{array}{c} \forall s_0, ((l_0, s_0) \xrightarrow{w}{}^{*}_{p} (l, s)) \in Trace[\![p]\!]s_0 . w \in L(A) \\ \Downarrow \\ \exists ((l'_0, s_0) \xrightarrow{w}{}^{*}_{Z} (l', s)) \in Trace[\![Conc(P^\sharp \times A)]\!]s_0 \end{array}$$

$$\begin{array}{c}\forall s_0, ((l_0, s_0) \xrightarrow{w}{}^{*}_{p} \alpha) \in \mathit{Trace}[\![p]\!] s_0 . w \notin L(A))\\ \Downarrow \\ (\exists ((l'_0, s_0) \xrightarrow{w_1}{}^{*}_{Z} \alpha' \xrightarrow{\mathsf{abort}}{}_{Z} \bot) \in \mathit{Trace}[\![\mathit{Conc}(P^\sharp \times A)]\!] s_0, w_1, a, w_2 . \\ w = w_1 a w_2 \wedge w_1 \in L(A) \wedge w_1 a \notin L(A))\end{array}$$

Cette propriété se prouve par une induction sur la taille de w en utilisant les propriétés de sûreté du tissage et de l'abstraction.

PROPRIÉTÉ **2.8.3.** **[Fidélité du tissage]** *Soit le programme de base P, $P^\sharp$ une abstraction* sûre *et* fidèle *de P, et l'automate de l'aspect A, on dit que le tissage est* fidèle *si et seulement si :*

$$\begin{array}{c}\forall s_0, ((l'_0, s_0) \xrightarrow{w}{}^{*}_{Z} (l', s)) \in \mathit{Trace}[\![\mathit{Conc}(P^\sharp \times A)]\!] s_0 \\ \Downarrow \\ (\exists ((l_0, s_0) \xrightarrow{w}{}^{*}_{p} (l, s)) \in \mathit{Trace}[\![p]\!] s_0 . (l_0, s_0) \xrightarrow{w}{}^{*}_{p} (l, s) \wedge w \in L(A))\end{array}$$

$$\begin{array}{c}\forall s_0, ((l'_0, s_0) \xrightarrow{w}{}^{*}_{Z} \alpha' \xrightarrow{\mathsf{abort}}{}_{Z} \bot) \in \mathit{Trace}[\![\mathit{Conc}(P^\sharp \times A)]\!] s_0 \\ \Downarrow \\ (\exists ((l_0, s_0) \xrightarrow{w}{}^{*}_{p} \alpha_1 \xrightarrow{a}{}_{p} \alpha_2) \in \mathit{Trace}[\![p]\!] s_0 . \\ w \in L(A) \wedge wa \notin L(A))\end{array}$$

Cette propriété se prouve encore une fois par une induction sur la taille de w et repose sur les propriétés de fidélité de la concrétisation et de l'abstraction.

La propriété de correction de notre approche est formulée sur des sémantiques fonctionnelles qui ne sont définies que pour les programmes séquentiels qui terminent. La correction devrait pouvoir s'étendre à des programmes non terminants comme les propriétés intermédiaires de sûreté et fidélité du tissage restent valables. Il faut évidemment effectuer quelques adaptations de la preuve. Par exemple cela nécessite d'utiliser la sémantique de traces des programmes. Nous avons aussi l'intuition que cette approche reste correcte dans un cadre de programmes parallèles comme les preuves des propriétés de sûreté et de fidélité du tissage n'utilisent pas le fait que les programmes sont séquentiels et déterministes.

2.9 Extensions

L'approche que nous avons décrite dans les sections précédentes, a l'avantage d'être simple et compréhensible, mais elle a aussi certaines limitations ou inconvénients. Par exemple, cette approche ne décrit le tissage que pour un unique aspect. Nous avons aussi remarqué que le produit d'automates pouvait conduire à une explosion d'états de la taille du programme tissé. Nous avons aussi limiter l'expressivité du langage d'aspects en nous restreignant au seul effet d'arrêt du programme. Dans cette section, nous présentons des solutions à ces différentes questions :

- nous expliquons comment étendre cette approche au tissage de plusieurs aspects ;
- nous proposons une alternative au produit d'automates qui permet de retourner des programmes tissés de tailles similaires à celles des programmes initiaux ;
- nous proposons une extension pour gérer des événements qui ne seraient plus uniquement syntaxiques mais aussi sémantiques ;
- nous discutons également comment étendre notre approche à des aspects de type spectateur, et à des langages d'aspects généraux (de type invasif).

2.9.1 Tissage de plusieurs aspects

Intuitivement, le tissage de plusieurs aspects consiste à d'effectuer un tissage séquentiel en tissant le premier aspect sur le programme, puis le deuxième aspect sur le programme tissé obtenu, etc. Le tissage, qui est modélisé par une composition parallèle, s'étend à plusieurs aspects simplement de la manière suivante :

$$(\ldots((P \times A_1) \times A_2) \times A_3) \ldots)$$

où P représente l'AFC du programme et les A_i les aspects.

Comme l'opération $\times$ de produit d'automates est associative, il est aussi possible de construire un aspect $A = (A_1 \times A_2 \times A_3 \ldots)$ représentant les autres aspects. De plus comme l'opération $\times$ est commutative, il n'y a pas de conflits entre les aspects (par exemple $A_1 \times A_2 = A_2 \times A_1$). Cette propriété s'explique par le fait que les aspects ont pour unique comportement de couper et sélectionner des traces. Ces propriétés ne seraient plus satisfaites dans le cas d'aspects généraux qui peuvent insérer du code arbitraire. L'ordre de tissage de ces aspects détermine l'ordre d'exécution des inserts quand plusieurs inserts s'appliquent sur le même point de coupure.

2.9.2 Tissage par un produit instrumenté

L'opération de produit peut conduire à une explosion d'états de l'automate représentant le programme tissé. Une solution possible pour résoudre ce problème est d'utiliser à la place du produit standard le produit instrumenté proposé par Fradet et Colcombet dans [CF00]. L'idée de ce produit instrumenté est de coder l'état de l'automate par une variable et ainsi éviter de dupliquer le code du programme. L'une augmentation quadratique de la taille des programmes tissé est remplacé par un léger surcoût en temps (mise à jour et test de la variable représentant l'état de l'automate).

Nous illustrons cette technique en gardant le même exemple que celui utilisé dans ce chapitre. La figure 2.11 montre le graphe instrumenté de P par A. Les nœuds sont annotés par l'ensemble des états de l'automate (auxquels on at-

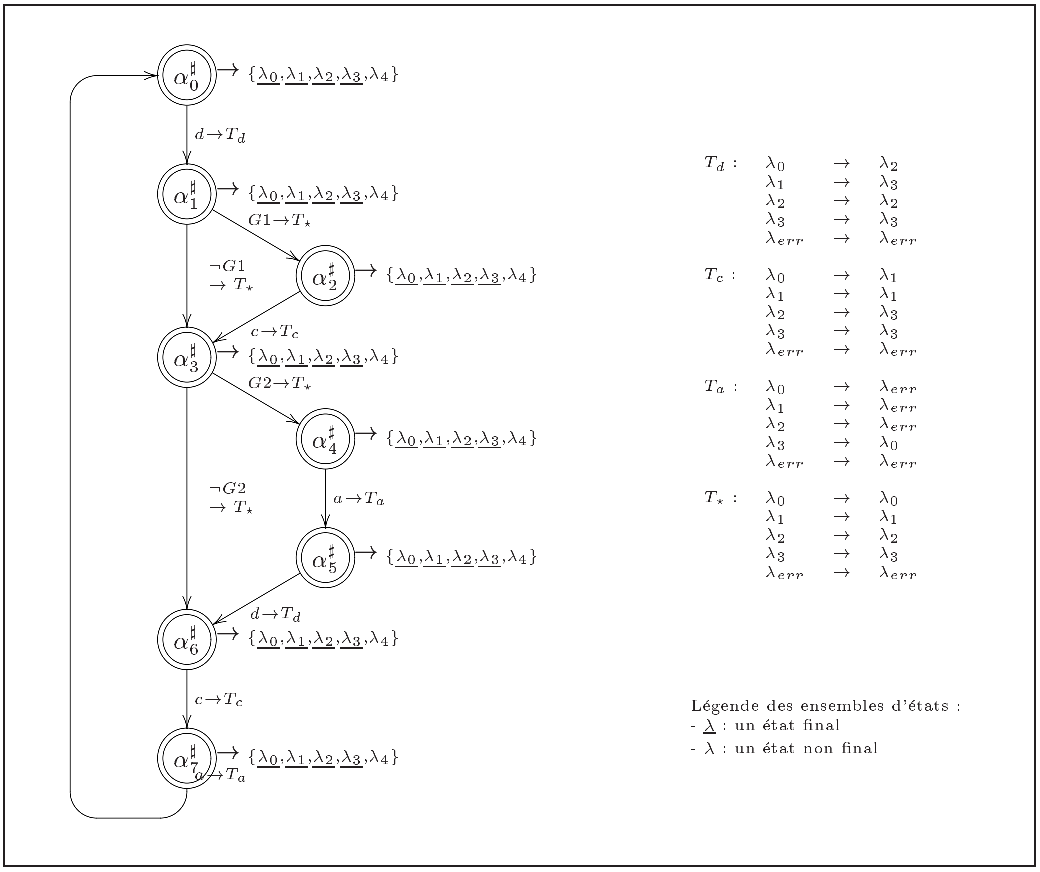


FIG. 2.11 – Instrumentation directe du programme P par l'aspect A

tache l'information d'être final ou non-final) et les arcs sont annotées par une table de transitions qui représente l'effet de l'événement sur l'état de l'aspect. La sémantique du graphe instrumentée est donnée par une trace dont les états sont des couples de l'état du programme et de l'état de l'aspect et qui commence dans les états initiaux du programme et de l'aspect. Une transition de cette trace est étiquetée par l'événement du programme et fait évoluer l'état courant de l'aspect ; en particulier, l'état de l'aspect évolue en fonction de la table de transition associée à l'événement du programme. Nous pouvons noter que ce graphe reconnait le même langage que celui reconnu par l'automate $P \times A$; (l'intersection des langages de l'automate du programme P et de celui de l'aspect A).

Le graphe instrumenté peut se voir comme une abstraction d'un programme où :

- l'état courant de l'aspect est codée par une variable ;
- les tables de transitions sont traduites par des tests et des affectations qui mettent à jour la variable de l'aspect ;
- des tests sont ajoutés pour arrêter le programme si l'état courant de l'aspect

arrive dans un état non final.
Avant d'être concrétisé en un programme, le graphe instrumenté est optimisé afin d'effectuer le minimum de tests et d'affectations.

Le graphe est tout d'abord minimisé en calculant pour chaque nœud les états atteignables de l'aspect. Il est ensuite simplifié pour minimiser le nombre de transitions nécessaires afin de supprimer des affectations dans le programme tissé. La figure 2.12 montre un exemple de simplification des transitions. L'idée est que

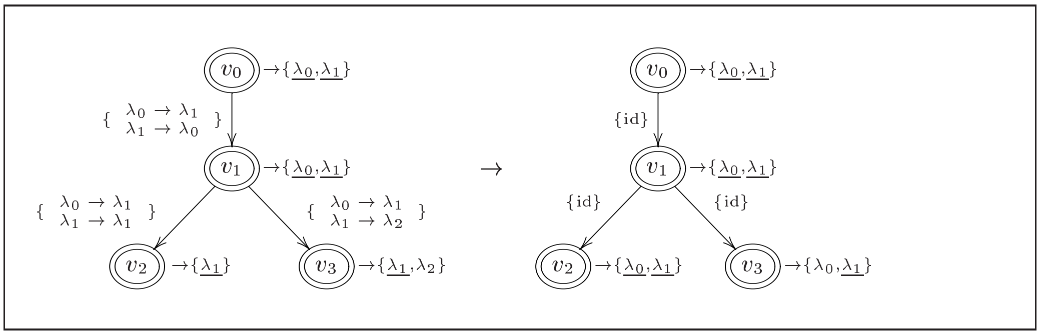


FIG. 2.12 – Exemple de simplification d'un graphe

si le nœud courant du graphe est v_3, il est clair que le graphe est passé par le nœud v_0 et qu'il a ensuite pris deux arcs. On peut supprimer ces transitions en notant que si le graphe est dans le nœud (v_3, λ_0), cela correspond au nœud (v_3, λ_2) pour le graphe non optimisé. De manière générale les tables de transitions d'un sous-graphe qui forme un arbre peuvent être supprimées sans perte d'information (comme l'exemple). Il est uniquement nécessaire de garder des tables de transitions quand il y a un nœud dans lequel au moins deux transitions sont entrantes (pour savoir quelle transition a été prise) ; cela correspond à des boucles et des conditionnelles. La figure 2.14 montre le graphe instrumenté de la figure 2.11 après simplifications.

Le graphe obtenu est finalement concrétisé. Pour notre exemple, le code du programme tissé est donnée en figure 2.14. Nous pouvons remarquer que le code du programme initial n'a pas été dupliqué. Seuls deux affectations et un seul test ont été ajoutés. L'avantage de cette technique est qu'elle évite l'explosion de la taille du programme : la taille des programmes tissés est proportionnelle au programme de base. Dans le pire cas, il faut ajouter une affectation pour chaque conditionnelle et boucle dans le programme ainsi que un test à chaque point de programme où la propriété peut être violée. Sur cet exemple, le programme instrumenté (11 instructions) est cependant plus grand que le programme tissé par produit (10 instructions). Ce résultat s'explique par la petite taille du programme de base.

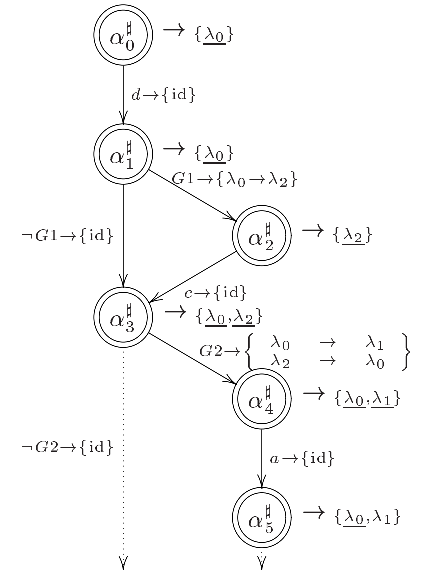


FIG. 2.13 – Simplification du graphe de la figure 2.11

$$P = \left\{ \begin{array}{lclcl} l_0 & : & \mathtt{state} := \mathtt{0} & \leadsto & l_1 \\ l_1 & : & \mathtt{directeur}() & \leadsto & l_2 \\ l_2 & : & \mathtt{G1} & \leadsto & l_3 \ , \ l_5 \\ l_3 & : & \mathtt{state} := \mathtt{1} & \leadsto & l_4 \\ l_4 & : & \mathtt{comptable}() & \leadsto & l_5 \\ l_5 & : & \mathtt{G2} & \leadsto & l_6 \ , \ l_{10} \\ l_6 & : & \mathtt{state} == \mathtt{0} & \leadsto & l_7 \ , \ l_8 \\ l_7 & : & \mathtt{abort}() & \leadsto & l_{erreur} \\ l_8 & : & \mathtt{achat}() & \leadsto & l_9 \\ l_9 & : & \mathtt{directeur}() & \leadsto & l_{10} \\ l_{10} & : & \mathtt{comptable}() & \leadsto & l_{11} \\ l_{11} & : & \mathtt{achat}() & \leadsto & l_0 \end{array} \right\}$$

FIG. 2.14 – Concrétisation du graphe de la figure

2.9.3 Gestion d'événements sémantiques

Nous avons présenté jusqu'à présent des aspects qui reposent sur des points de jonction syntaxique (définis uniquement par le nom de l'instruction). Dans les langages d'aspects classiques comme AspectJ, le point de jonction peut aussi être d'ordre sémantique ; c'est à dire qu'il est défini non seulement par le nom de l'instruction exécutée, mais aussi par des conditions sur les paramètres de cette instruction.

Notre approche peut gérer de tels événements sémantiques. Comme le tissage est effectué statiquement et les valeurs dynamiques des variables sont complètement abstraites, des tests correspondant à la partie dynamique du point de jonction doivent être ajoutés. Il faut noter que si l'aspect était imposé par un moniteur, ces mêmes tests seraient exécutés par le moniteur pour tester les conditions sur les valeurs du point de jonction. L'alphabet des automates contient des événements de la forme (t, a) où t est un test sur la mémoire courante (la partie sémantique de l'événement), et a est l'instruction du programme qui est filtrée (la partie syntaxique de l'événement). Nous supposons aussi que les aspects respectent des contraintes de structure afin que l'automate obtenu par produit puisse être concrétisé en un programme. En particulier, l'aspect doit être déterministe et si dans un état l'événement sémantique (t, a) est filtré, alors l'événement sémantique $(\neg t, a)$ est également filtré dans cet état.

L'ajout des événements sémantiques dans les aspects nécessite d'adapter l'opération de tissage. A la place du produit classique, nous utilisons un produit dédié $(\times_d)$ qui est défini par les deux règles suivantes :

$$\frac{\alpha \xrightarrow{a}_{p^\sharp} \alpha' \quad \lambda \xrightarrow{a}_{\lambda} \lambda'}{(\alpha, \lambda) \xrightarrow{a}_{I} (\alpha', \lambda')}$$

$$\frac{\alpha \xrightarrow{a}_{p^\sharp} \alpha' \quad \lambda \xrightarrow{(t,a)}_{\lambda} \lambda'}{(\alpha, \lambda) \xrightarrow{t}_{I} q \xrightarrow{a}_{I} (\alpha', \lambda')}$$

La première règle correspond à la gestion d'événements purement statiques comme précédemment (produit standard). La deuxième règle considère un événement sémantique (t, a) et ajoute un test t avant l'exécution de l'instruction a.

Nous reprenons l'exemple utilisé dans ce chapitre pour illustrer l'expressivité et le tissage d'événements sémantiques. Nous modifions l'aspect tel qu'il n'y a plus de violation de la propriété si le coût de l'achat envisagé est inférieur à 50. La figure 2.16 montre l'automate de cette politique de sécurité. Nous considérons que le coût d'un achat est défini par la variable globale cout. La figure 2.16 donne le programme P tissé avec ce nouvel aspect. Comparé à précédemment, un test est ajouté (cout¿ `50`) avant de conclure à la violation de la propriété.

L'ajout des tests rend le produit dédié non commutatif. L'ordre de tissage de plusieurs aspects peut changer le programme tissé si les deux aspects ajoutent des tests avant l'exécution de la même instruction. Cependant comme les tests ne sont pas bloquants et ne modifient pas la mémoire, la sémantique fonctionnelle du programme tissé reste la même quelque soit l'ordre de tissage des aspects. Le produit dédié reste associatif et commutatif en terme sémantique. Les aspects peuvent

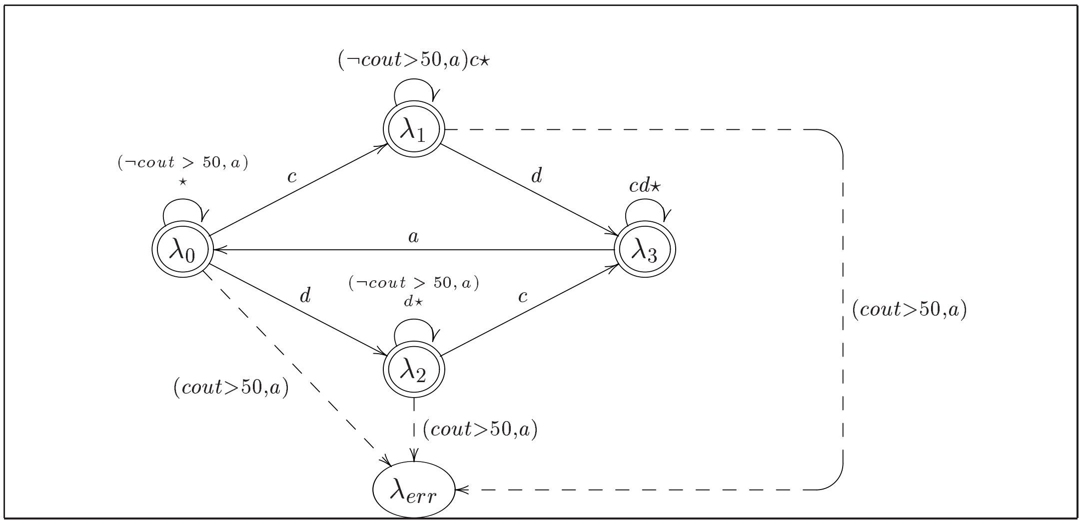


FIG. 2.15 – Aspect avec des événements sémantiques

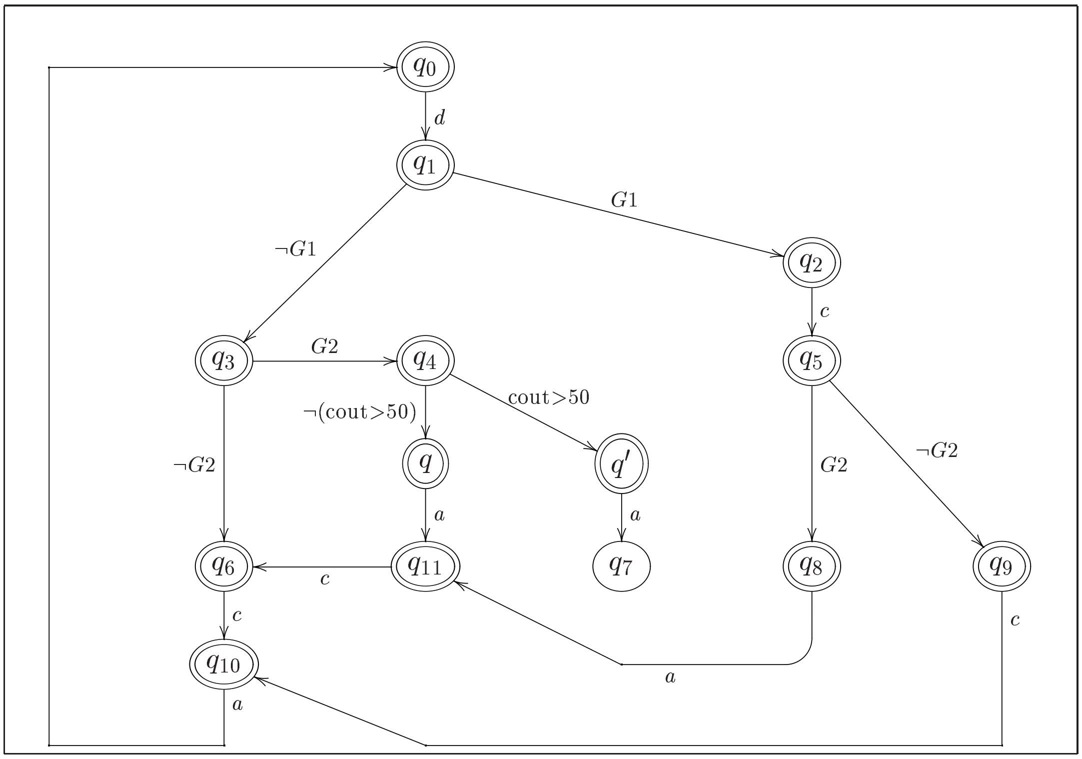


FIG. 2.16 – Résultat du produit dédié entre le programme P (en bas) et l'aspect de la figure

toujours être vus comme des propriétés qui déterminent les traces valides et qui arrêtent le programme en cas de violation de la propriété.

2.9.4 Modéliser des aspects de type spectateur

Nos aspects qui décrivent des propriétés de sûreté ont pour unique effet de sélectionner et de couper des traces. L'aspect pourrait effectuer d'autres traitements quand la propriété n'est plus respectée. Afin de toujours maîtriser l'impact sémantique des aspects, nous nous intéressons à aspects de type spectateur. Ces aspects peuvent insérer des instructions, indépendantes du programme de base, avant ou après une instruction du programme ; cela correspond aux inserts de type `before` ou `after` d'AspectJ. Par exemple, l'aspect pourrait, au lieu d'interrompre le programme, avertir l'utilisateur lorsqu'une propriété de sécurité est violée. On peut vouloir également tissé un aspect qui compte le nombre d'appels d'une méthode.

Il est possible d'adapter notre technique afin de modéliser ces aspects. Il suffit de modifier le codage des automates d'aspects et d'utiliser la composition parallèle d'automates à la place du produit. La composition parallèle est une opération qui étend le produit en permettant des transitions sur les symboles n'appartenant pas à l'alphabet des deux automates. Cette opération est décrite formellement dans l'annexe B.Comme les aspects introduisent de nouvelles instructions, le langage de l'automate correspondant inclut, en plus des événements du programme, les événements ajoutés par l'aspect.

L'insertion de l'événement I *après* l'événement A est modélisé par une transition étiquetée par A suivie d'une transition étiquetée par I. Cette insertion est décrite par la figure 2.17. Lors de la composition parallèle, toutes les transitions

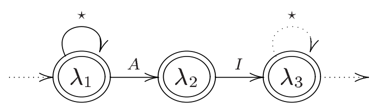


FIG. 2.17 – Aspect insèrant l'instruction I *après* l'instruction A

étiquetées par A deviendront la transition A suivie par la transition I. Comme le symbole I n'appartient pas au langage du programme. la composition parallèle l'ajoute après synchronisation sur A.

L'insertion de l'événement I *avant* l'événement A est modélisé par une transition étiquetée par I suivie d'une transition étiquetée par A. Cependant la composition parallèle de cet aspect ajoute la transition I dans tous les états de l'automate. Ce problème est résolu en s'assurant que l'état entre les deux transitions (I et A)

n'est pas final. L'opération de coaccessibilité permet ensuite de supprimer toutes les transitions I qui amènent dans un état puits non final (*i.e.,* toutes les transitions I non suivies immédiatement de A). La figure 2.18 décrit l'insertion de l'instruction I avant l'instruction A.

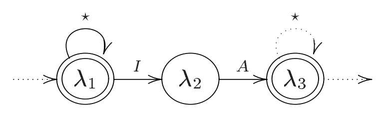


FIG. 2.18 – Aspect insèrant l'instruction I *avant* l'instruction A

La figure 2.19 montre un aspect de type spectateur qui peut être tissé par composition parallèle d'automates sur notre programme exemple P. Cet aspect

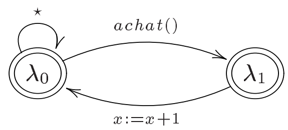


FIG. 2.19 – Aspect comptant le nombre d'achats

compte le nombre d'achats effectué par un utilisateur. A chaque fois que la fonction `achat` est appelée, la variable x (supposée locale à l'aspect et initialisée à 0) est incrémentée.

2.9.5 Modéliser un langage d'aspect classique

Même en utilisant les extensions proposées, nos aspects restent moins expressifs que les langages d'aspects classiques (comme AspectJ) qui offrent des mécanismes pour insérer des inserts arbitraires avant, après, ou à la place d'un point de jonction (respectivement appelé `before`, `after` et `around` dans AspectJ). Dans cette partie, nous envisageons les modifications nécessaires pour représenter dans notre cadre des langages d'aspects classiques.

La technique précédente (section 2.9.4) peut être utilisée pour tisser des inserts arbitraires de type `before` ou `after`. Cependant le fait d'avoir des inserts arbitraires ne permet pas de garder la maîtrise de l'impact sémantique du tissage. Un insert modifiant une variable du programme de base peut changer complètement le comportement (flot de contrôle) et invalide potentiellement toutes les propriétés que vérifiait le programme. Le tissage n'est plus associatif, ni commutatif ; l'ordre d'application des aspects détermine la sémantique du programme tissé. Il est très difficile de déterminer la propriété imposée par l'aspect comme celle-ci dépend des autres aspects.

Cette technique ne permet pas de représenter des inserts de type `around` sans `proceed`, c'est à dire de remplacer une instruction par d'autres instructions. Ce type d'insert n'est pas modélisable dans notre cadre qui repose sur les automates car la composition parallèle d'automates ne permet pas de coder la suppression d'instructions. Les inserts de type `around` pourraient être modélisés avec d'autres formalismes proche comme les "Edit Automata [LBW05]". De manière générale, les langages d'aspects classiques sont trop expressifs et sortent du cadre de notre approche (où les aspects définissent des propriétés).

2.10 Conclusion

Nous avons présenté une approche formelle de la programmation par aspects qui repose sur les automates finis déterministes. Un aspect est une propriété sur les traces d'exécution du programme qui est imposée par un tissage modélisé par un produit d'automates. La contribution de ce chapitre tient dans le travail de formalisation de l'approche. Une originalité de ce travail est de se restreindre à des aspects qui expriment une propriété de sûreté. L'effet du tissage se résume à sélectionner les traces d'exécution ou à couper les traces interdites. Cette restriction sur les aspects permet de raisonner modulairement sur le programme et les aspects comme les propriétés suivantes sont garanties :

- si le programme tissé termine normalement alors il retourne les mêmes résultats que le programme de base ;
- le comportement d'un aspect est indépendant des autres aspects (*i.e.*, le tissage est associatif et commutatif) ;
- les propriétés de sûreté de l'application de base (avec la restriction qu'elles ne parlent pas de "l'état suivant d'un état") et les invariants sur les états sont préservés sur le programme tissé.
- le tissage de l'aspect impose des contrôles supplémentaires de telle sorte que l'application tissée respectent des propriétés de sûreté supplémentaires.

Nous avons considéré dans ce chapitre des programmes déterministes et séquentiels. Dans ce contexte, les aspects arrêtent le programme quand la propriété est violée. Dans le chapitre 3, nous utilisons cette approche dans le contexte de programmes parallèles ; les aspects expriment en décrivant l'ensemble des traces souhaitées le choix d'un ordonnancement. La sémantique d'un réseaux de composants est donnée de manière similaire par l'ensemble de ses traces d'exécution. Nous décrivons toujours les aspects par des automates, mais ceux-ci sont spécifiés en utilisant des ensembles d'événements qui facilitent la description des automates. L'abstraction, le tissage et la concrétisation sont similaires (modulo les contraintes pour lire des données sur les canaux de communication) ce qui nous fait penser que la preuve de correction est une simple adaptation de la preuve de correction de ce chapitre. Dans le chapitre 4, nous étendons l'approche pour gérer des propriétés temporisées. Les aspects décrivent l'ensemble des traces temporelles souhaitées. Nous proposons un langage d'aspects textuel et dédié à la disponibilité. Le tissage revient à insérer des délais, ou arrêter le programme lorsque un délai ou une instruction viole la propriété. La preuve de correction pour des propriétés temporisées reste à faire. Nous pensons néanmoins que cette preuve suit la même structure que la preuve de correction de l'approche de base.

Chapitre 3

Aspects d'ordonnancement

Sommaire

3.1 Introduction

Nous avons proposé dans le chapitre 2 un cadre formel pour la programmation par aspects dans lequel les aspects sont des propriétés qui spécifient l'ensemble des traces d'exécution autorisées. Le tissage de l'aspect restreint les traces du programme aux seules traces autorisées ; ceci est équivalent à considérer que le tissage coupe les traces interdites. Nous avons mis l'accent sur des programmes séquentiels et déterministes pour lesquels l'aspect se limite à arrêter le programme quand la propriété est violée. Dans ce chapitre, nous appliquons cette approche à des réseaux de composants concurrents. Comme une exécution parallèle se représente par un arbre des exécutions possibles, la restriction de trace imposée par l'aspect peut se voir comme le choix d'un ordonnancement. L'utilisation de cette approche dans un cadre parallèle permet de réconcilier les avantages de la programmation par composants et les performances de la programmation séquentielle.

En effet, la programmation modulaire possède de nombreux avantages : lisibilité, maintenabilité, développement et compilation séparés. Cependant, la composition des unités modulaires (composants) pose des problèmes d'efficacité. La composition séquentielle pose un problème d'espace mémoire. Par exemple, composer séquentiellement un producteur et un consommateur implique que le producteur délivre toutes ses sorties avant que le consommateur puisse commencer. La composition parallèle utilise des tâches ("threads") où les synchronisations et les changements de contexte dégradent les temps d'exécution.

Nous proposons une technique, la *fusion de réseaux*, pour transformer automatiquement un réseau de composants en un unique programme séquentiel. Cette technique prend en entrées un réseau de composants et un aspect d'ordonnancement. Les réseaux sont formalisés par des *réseaux de Kahn* (désignés par KPNs pour "Kahn Process Networks"), un modèle déterministe du parallélisme à la fois simple et expressif [Kah74]. L'aspect d'ordonnancement permet à l'utilisateur de guider et contrôler la stratégie d'ordonnancement. L'aspect se présente sous la forme de contraintes qui spécifient le sous-ensemble des exécutions permises. Comme dans le chapitre 2, la sémantique opérationnelle des KPNs et l'aspect d'ordonnancement sont modélisés par des automates à états finis. En revanche, les aspects d'ordonnancement sont décrits par des automates et un langage permettant de décrire plus facilement des ensembles d'événements sémantiques, et la fusion comporte une étape supplémentaire avant la concrétisation pour séquentialiser totalement le réseau.

La fusion est un processus automatique qui prend un réseau de composants et un aspect et produit (par tissage et transformation) un programme séquentiel équivalent qui respecte les contraintes de l'aspect. Le programme obtenu rend les mêmes résultats que le KPN modulo les blocages artificiels ("deadlocks") éventuellement introduits par les contraintes (auquel cas, l'utilisateur est averti). Le fusion diminue le surcoût introduit par l'assemblage de composants en permettant la suppression des changements de contexte, le remplacement des communications internes par des affectations à des variables locales et de nombreuses optimisations standards sur le programme séquentiel résultant.

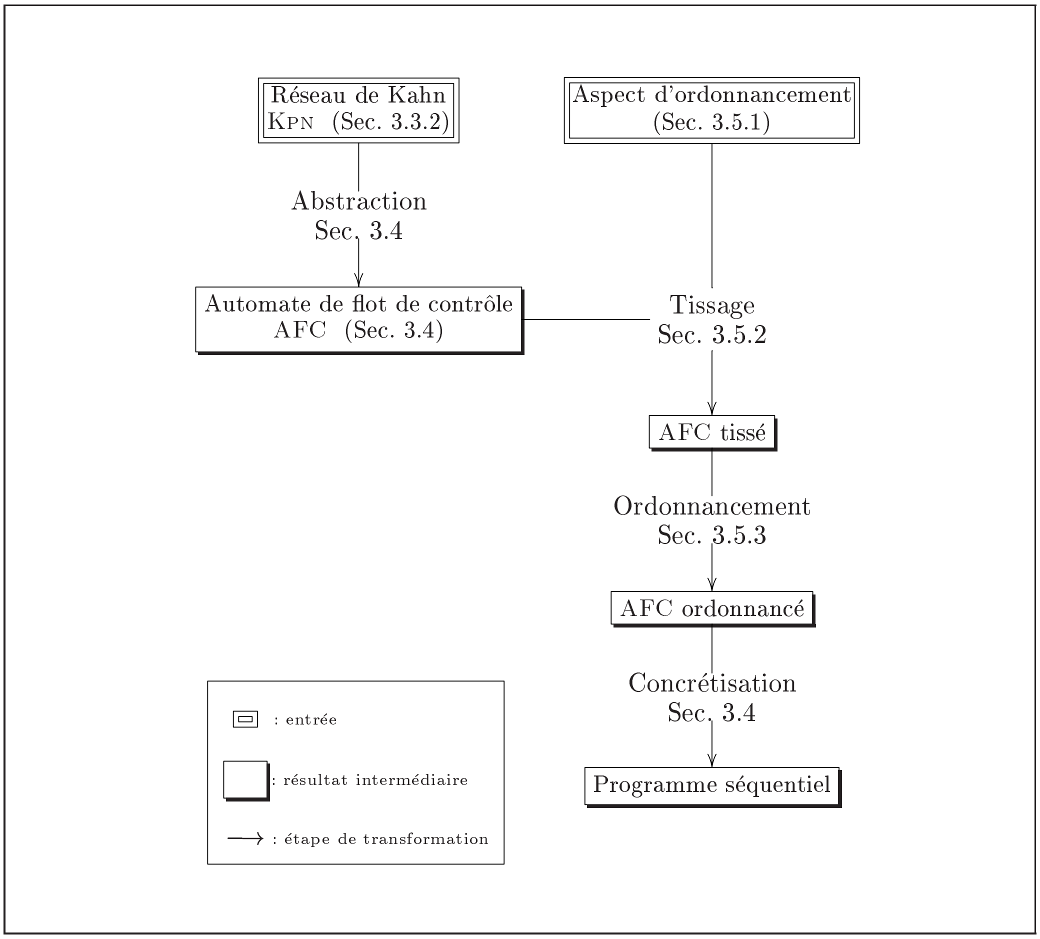


Fig. 3.1 – Les principales étapes de la fusion de réseau

Les étapes principales du processus de fusion sont représentées dans la figure 3.1. Cette figure montre aussi que la fusion est une simple extension de l'approche de base décrite au chapitre 2.

– La première étape est l'*abstraction* du réseau en un AFC qui sur-approxime l'ensemble des traces d'exécution possibles. Cette étape repose sur des techniques standards d'analyse (*e.g.,* interprétation abstraite). De nombreuses abstractions sont possibles en fonction de la taille et de la précision souhaitée de l'AFC.
– La seconde étape est le *tissage* de l'aspect dans l'AFC. Nous formalisons cette transformation par un produit entre l'AFC et l'automate représentant l'aspect. En général, cette étape ne séquentialise pas complètement l'exécution

et laisse des choix d'ordonnancement.

- La troisième étape finalise l'*ordonnancement* de l'AFC tissé. Plusieurs stratégies peuvent être utilisées pourvu qu'elle soient équitables. De nouveau, ces stratégies s'expriment par des automates et l'ordonnancement par un produit synchronisé.
- La quatrième étape, la *concrétisation*, traduit l'AFC séquentialisé en un programme séquentiel. Des optimisations standards peuvent ensuite être appliquées sur le programme obtenu.

Ce chapitre est organisé comme suit. La section 3.2 présente l'algorithme de fusion de filtres [PW96] qui est à l'origine de nos travaux sur les aspects d'ordonnancement. La section 3.3 présente la syntaxe et la sémantique des Kpns. La section 3.4 décrit les AFCs ainsi que les étapes d'abstraction et de concrétisation qui relient le modèle concret à celui abstrait. La section 3.5 présente les aspects d'ordonnancement et les deux étapes principales de la fusion : le tissage et l'ordonnancement. Nous discutons des travaux connexes et des extensions possibles en conclusion dans la section 3.6.

3.2 La fusion de filtres

Proebsting et Watterson ont proposé un algorithme qui permet de fusionner des filtres composés en pipeline [PW96]. Les filtres sont des composants prenant un flot de donnée en entrée et retournant un flot de donnée en sortie. Typiquement un filtre transforme et transmet les données au fur et à mesure de leur arrivée. Certaines applications peuvent se modéliser comme une composition de filtres où les données sont transformées par les filtres successifs. Le domaine d'application particulièrement visé est celui des protocoles réseaux. Les couches des protocoles réseaux sont souvent organisées en filtres. Chaque couche du protocole transforme (filtre) les messages qui la traversent. Une composition linéaire de filtres spécifie le chemin des données de la source à la destination.

$$\text{Source} \rightarrow \text{Filtre}_1 \rightarrow \ldots \rightarrow \text{Filtre}_N \rightarrow \text{Destination}$$

Dans une implémentation séquentielle modulaire, la source produit toutes les données avant de les transmettre au premier filtre. Les filtres fonctionnent de la même façon en opérant sur toutes les données avant de les transmettre au filtre suivant. Une telle implémentation est évidemment peu performante car elle a besoin d'une grande quantité de mémoire pour stocker les résultats intermédiaires. Il est plus efficace de fusionner les filtres pour obtenir un programme de la forme

$$Source \rightarrow Filtre_{global} \rightarrow Destination$$

qui produit ces résultats au fur et à mesure sans résultat intermédiaire. De façon classique, les programmeurs fusionnent les filtres manuellement pour obtenir un unique programme plus efficace. L'intégration manuelle est un processus coûteux en temps et générateur d'erreurs. De plus, une intégration manuelle produit des programmes difficiles à maintenir et à modifier : un changement dans un filtre a souvent des répercutions importantes dans le programme fusionné. L'algorithme proposé, Filter Fusion (FF), automatise l'intégration arbitraire de filtres qui ont été conçus indépendamment et permet en éliminant des accès mémoires d'augmenter les performances. La fusion de filtres automatique permet de réconcilier modularité et efficacité. Notre approche, la fusion de réseaux, peut se voir comme une généralisation à des stratégies de fusion (ordonnancement) spécifiées par l'utilisateur et à des réseaux arbitraires.

Un filtre est décrit par une procédure sans paramètre dans le langage C qui est étendu par trois opérations : put, get et filter. L'opération put produit une donnée pour le filtre suivant (ou pour la destination s'il n'y a pas de filtre suivant) et get consomme une donnée du filtre précédent (ou de la source s'il n'y a pas de filtre précédent). Le prédicat filter est vrai si des données sont en entrée, faux si le filtre précédent a terminé et non défini sinon.

```
Filter Evener
Decls
  int c, k ;
Code
  k=0 ;
  while filter
    get c ;
    put c ;
    k++ ;
  end-while
  if(k%2)
    put 0
End-Filter
```

```
Filter 2ByteSwap
Decls
  int x, y ;
Code
  while filter
    get x ;
    get y ;
    put y ;
    put x ;
  end-while
End-Filter
```

FIG. 3.2 – Code source des filtres Evener (à gauche) et 2ByteSwap (à droite)

La figure 3.2 donne un exemple de deux filtres. Le filtre Evener est un filtre qui envoie toujours un nombre pair de bits en complétant si nécessaire par un 0. Le filtre 2ByteSwap est un filtre qui échange les bits adjacents ; il requiert de recevoir un nombre pair de bits. Nous considérons la composition de filtres suivantes :

$$\ldots \rightarrow (Evener \rightarrow 2ByteSwap) \rightarrow \ldots$$

L'algorithme FF fusionne les filtres deux à deux, le filtre qui produit des données est désigné comme le producteur et celui qui consomme des données comme le consommateur. Le but est de faire correspondre les put du producteur avec les get du consommateur et de les remplacer par des affectations. L'algorithme FF compose les graphes de flot de contrôle des deux filtres en un unique graphe de contrôle représentant la fusion des deux filtres. Le principe de cet algorithme est d'exécuter chaque filtre symboliquement en alternant entre le producteur et le consommateur. Le contrôle, initialement donné au producteur, passe du producteur au consommateur après chaque put et du consommateur au producteur après chaque get. Chaque instruction exécutée symboliquement est ajoutée dans le nouveau graphe de contrôle représentant le programme fusionné. Chaque nœud ajouté est annoté par trois informations : la dernière instruction exécutée dans le producteur, la dernière instruction exécutée dans le consommateur et le filtre origine de ce nœud. Deux nœuds du graphe sont identiques si leurs annotations sont identiques. L'algorithme itère jusqu'à ce que le graphe de flot de contrôle n'évolue plus. Le prédicat filter représente un noeud conditionnel du graphe de contrôle du producteur ou du consommateur, qui guide le filtre en informant si le producteur a terminé ou non. L'état du producteur permet de déterminer statiquement la valeur de ce prédicat pour le consommateur ; si le producteur est suspendu en attente de reprendre le contrôle après un get, il est évalué à vrai et sinon à faux. Il n'est pas possible d'évaluer statiquement ce prédicat pour le producteur.

Nous illustrons cette technique en décrivant la fusion des deux filtres *Evener* et *2ByteSwap* dont les graphes de flot de contrôle sont donnés par la figure 3.3. Le graphe de contrôle final est composé d'instructions des deux filtres, exceptés

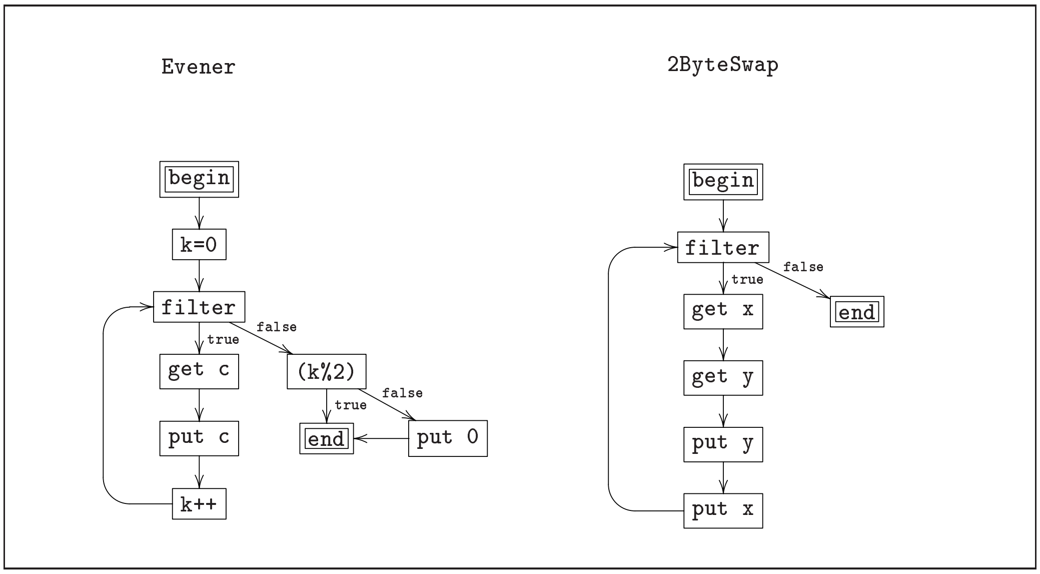


Fig. 3.3 – Graphe de contrôle des filtres Evener (à gauche) et 2ByteSwap (à droite)

les opérations put et get qui sont remplacées par des affectations. La figure 3.4 montre le résultat de la première itération de l'algorithme FF après avoir déroulé toutes les branches du producteur jusqu'à une instruction put.

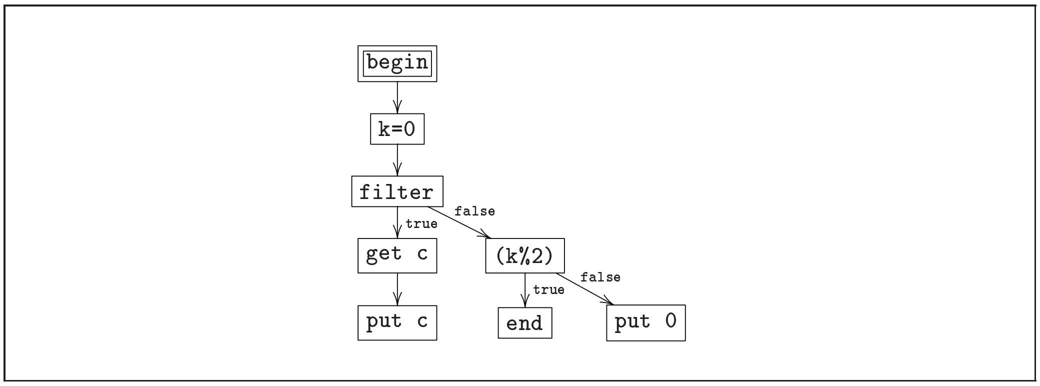


Fig. 3.4 – Résultat de la première itération de l'algorithme

La figure 3.5 montre le résultat de la seconde itération de l'algorithme en exécutant le consommateur jusqu'à une instruction get à partir des nœuds où le producteur est bloqué sur un put.

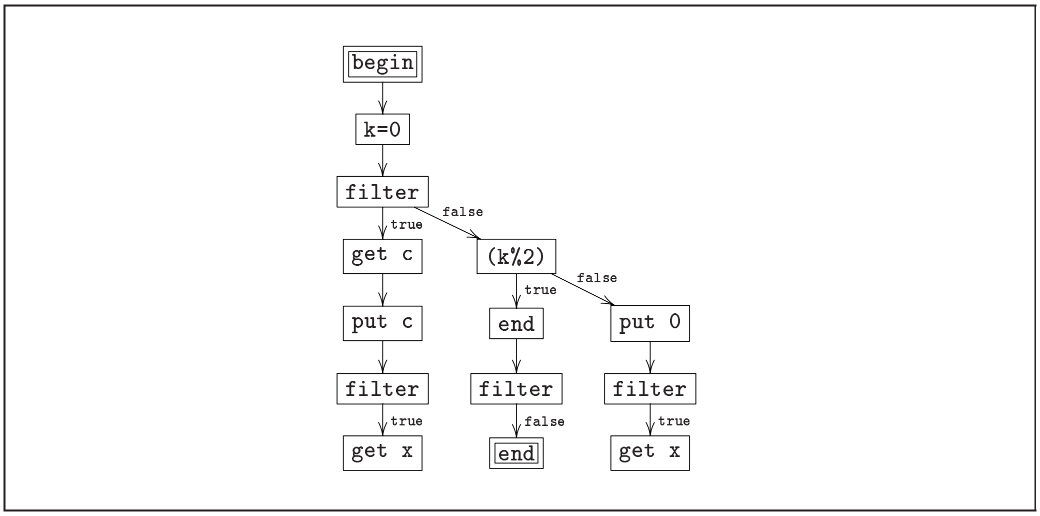


Fig. 3.5 – Résultat de la seconde itération de l'algorithme

Le graphe de contrôle obtenu n'est pas encore un programme. Certaines branches mènent à des états sans transitions de sortie où le producteur a terminé et le consommateur est bloqué sur une instruction get. Ces cas de blocages sont gérés simplement en remplaçant ces branches par une instruction générant une erreur.

Les expérimentations effectuées par les auteurs montrent que l'algorithme FF permet d'obtenir des performances presque équivalentes à du code fusionné à la main et toujours supérieures à la version exécutée séquentiellement. Cet algorithme a pour inconvénient de dupliquer du code, ce qui peut conduite dans certains cas à une explosion de la taille du programme obtenu. L'exemple le plus intéressant que Proebsting et Watterson présentent est une composition de six filtres de manipulation de bits (dont un encodage et un décodage PES [LM91]). Les temps d'exécution des différentes implémentations sont : 50 secondes pour la version exécutée séquentiellement, 25 secondes pour la version fusionnée manuellement et 30 secondes pour la version fusionnée automatiquement avec leur algorithme. Du point de vue de la taille du code source, les tailles du source (en lignes) et de l'exécutable (en octets) sont : 36 et 1955 pour la version exécutée séquentiellement, 28 et 1898 pour la version fusionnée manuellement, 197 et 2127 pour la version fusionnée automatiquement. Pour le domaine d'application considéré, la fusion de filtres permet de maintenir une conception et une implémentation modulaires sans sacrifier la performance.

Clayton et Calvert ont proposé une modification de l'algorithme FF pour diminuer le nombre de commutations entre le producteur et le consommateur [CC97]. Ils partent de l'observation que l'algorithme FF déroule une fois le producteur jusqu'à un `put` puis une fois le consommateur jusqu'à un `get`. Ils généralisent l'algorithme de sorte de dérouler n fois le producteur jusqu'à un `put` puis n fois le consommateur jusqu'à un `get`. Les affectations dans une variable sont changées en affectations dans une file bornée de taille n.

La technique que nous proposons dans ce chapitre, la fusion de réseaux, reprend le principe de la fusion de filtres mais la généralise à des réseaux arbitraires de composants et d'autres stratégies d'ordonnancement. Elle permet également de formaliser les algorithmes de fusion de filtres (Filter Fusion et l'extension de Clayton et Calvert) en des opération de produit d'automates. Nos aspects permettent de décrire d'autres stratégies comme une stratégie dirigée par la demande où le producteur est exécuté uniquement quand le consommateur a exécuté l'opération `get`.

3.3 Réseaux de composants

Nous décrivons dans cette section la syntaxe des composants et des réseaux et illustrons leur sémantique par un exemple. Le langage des composants étend le langage de programme proposé au chapitre 2 en ajoutant des instructions pour communiquer avec les autres composants. La sémantique d'un réseau de composants n'est pas donnée par une trace linéaire mais par un ensemble de traces (ou un arbre des exécutions possibles). Une sémantique opérationnelle complète des KPNs peut être trouvée dans [GB03].

3.3.1 Composants

Un composant (ou processus) p est défini, comme au chapitre 2, par un ensemble d'instructions $\{I_1, \ldots, I_n\}$ de la forme :

$$I \quad ::= \quad l_1 : c \rightsquigarrow l_2 \quad | \quad l_1 : g \rightsquigarrow l_2\ ,\ l_3$$

Une commande est soit une opération de lecture sur un canal d'entrée $f?x$, soit une opération de lecture sur un canal de sortie $f!x$, soit une opération interne i (par exemple, des affectations) que nous ne spécifions pas. Ces commandes s'interprètent de la même manière qu'au chapitre 2.

Les composants que nous considérons représentent des programmes séquentiels et déterministes. Ils satisfont les restrictions suivantes :
- un composant a un unique point d'entrée ;
- toutes les étiquettes utilisées dans le programme sont définies dans la partie gauche d'une unique instruction ;
- les commandes $f?x$ et $f!y$ ne peuvent pas apparaître dans le même composant (*i.e.,* un canal d'entrée ne peut pas être un canal de sortie et vice et versa).

Le programme P de la figure 3.6 écrit l'ensemble $\mathbb{N}$ en ordre croissant sur le canal f. Le programme C affecte à a la valeur lue sur le canal f si $a < b$ ou affecte à b la valeur lu sur le canal ι sinon ; il écrit ensuite $a + b$ sur le canal o et réitère. Nous assumons que l'étiquette initiale est toujours indexée par 0 et que la mémoire initiale initialise les variables entières à la valeur 0 ; Les étiquettes initiales des programmes P et C (figure 3.6) sont p_0 et c_0 respectivement et les variables x, a et b sont initialisées à 0.

Une trace d'exécution d'un composant p est exprimée par un LTS $(\Sigma_p, (l_0, s_0), \mathcal{E}_p, \longrightarrow_p)$ comme décrit à la section 4.6.2. Pour simplifier la présentation, nous considérons seulement des programmes qui ne terminent jamais. La terminaison peut toujours être représentée par une commande finale qui boucle de la forme

$$l_{end} : skip \rightsquigarrow l_{end}$$

où skip est la commande qui ne fait rien.

3.3.2 Réseaux

Un KPN k est constitué d'un ensemble de processus $\{p_1, \ldots, p_n\}$ exécutés de façon concurrente. Un réseau connecte deux à deux les canaux de sortie aux canaux d'entrée des composants. Ces canaux sont appelés canaux internes ; les canaux restants (non connectés) sont les canaux d'entrée et de sortie du réseau. Les communications sur les canaux internes sont asynchrones et sont modélisées par des files FIFO non bornées (l'écriture n'est jamais bloquante alors que la lecture

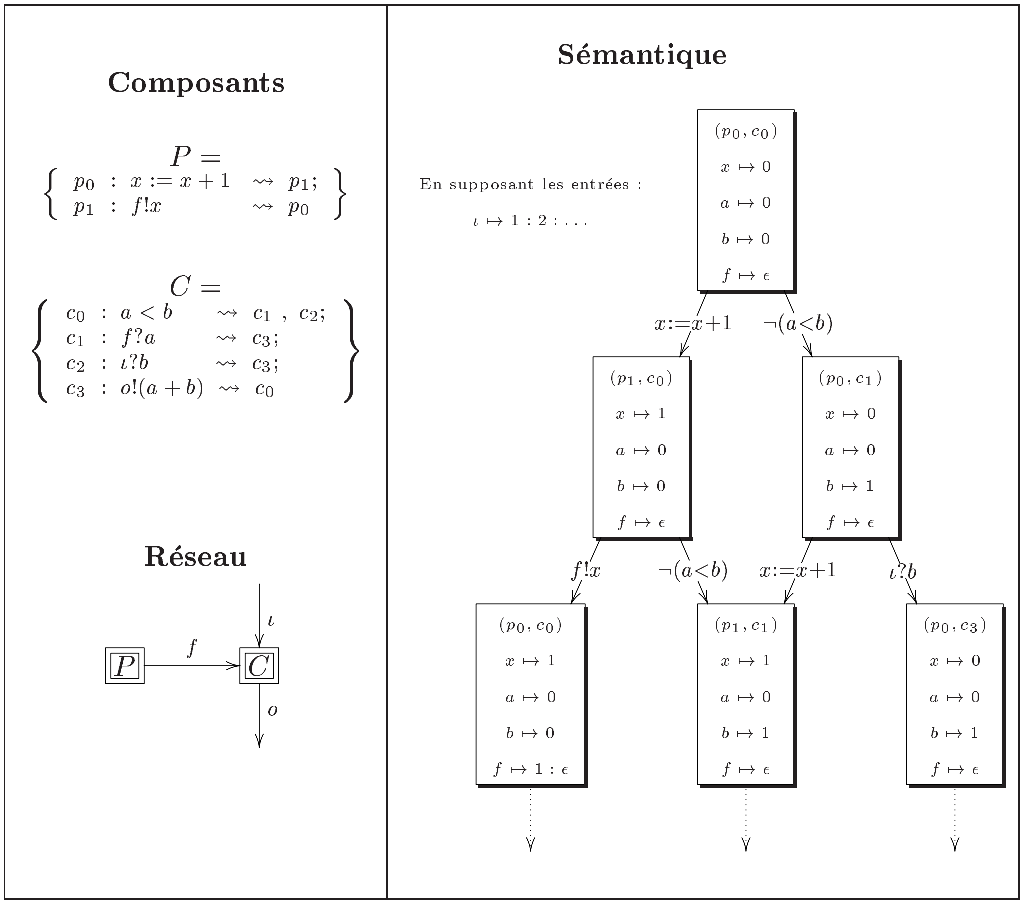


FIG. 3.6 – Un exemple simple de KPN et un extrait de sa sémantique de trace

sur file vide est bloquante). Dans le but de garantir un comportement déterministe, les KPNs vérifient les conditions syntaxiques suivantes [Kah74] :

- un canal interne est écrit par exactement un processus et lu par exactement un processus ;
- un canal d'entrée est lu par exactement un composant et écrit par aucun ;
- un canal de sortie est écrit par exactement un composant et lu par aucun ;
- un composant ne peut pas tester l'absence (ou la présence) d'une valeur sur un canal.

Dans le but de simplifier les développements techniques, nous considérons des réseaux avec un unique canal d'entrée ι et un unique canal de sortie o et supposons que le canal d'entrée n'est jamais définitivement vide.

La sémantique opérationnelle d'un KPN est donnée par l'ensemble des traces d'exécution. Une trace est exprimée par un LTS $(\Sigma_k, \alpha_0, \mathcal{E}_k, \longrightarrow_k)$ où :

- Σ_k est un ensemble (infini) de configurations. Une *configuration* est constituée

des états de chacun des composants et de l'état de chacun des canaux internes (*i.e.*, une suite finie de valeurs $v_1 : \ldots : v_n : \epsilon$) ;
- la configuration initiale α_0 est telle que chaque composant est dans son état initial et que chaque canal interne est vide ;
- $\mathcal{E}_k$ est l'union des ensembles de commandes des composants ; ces ensembles sont supposés être disjoints ;
- la relation de transition $\longrightarrow_k$ est définie comme exécutant (d'une manière non déterministe) les instructions de chaque processus. La sémantique d'un KPN ajoute la contrainte qu'une opération de lecture est bloquante sur un canal vide.

La relation de transition définit un graphe infini représentant toutes les exécutions parallèles possibles pour un état initial donné. Le début de la relation de transition $\longrightarrow_p$ pour notre exemple est donné dans la figure 3.6. Ici, aucun blocage artificiel n'est possible et toutes les traces sont infinies. Nous renvoyons le lecteur à [GB03] pour une formalisation d'une telle relation de transition.

Une trace d'exécution infinie est dite *équitable* si une action exécutable à un point de l'exécution sera finalement exécutée. Ce critère est nécessaire afin de faire progresser tous les composants et ainsi d'éviter les famines. La sémantique dénotationnelle d'un KPN est donnée par la fonction des valeurs d'entrées (le canal d'entrée) dans les valeurs de sortie (le canal de sortie) générée par les exécutions équitables. Nous écrivons $\mathit{Traces}(k)$ et $IO(k)$ pour dénoter respectivement l'ensemble des traces et la sémantique dénotationnelle du KPN k. Les KPNs de composants déterministes sont déterministes [Kah74]. Aussi, toutes les exécutions équitables avec les mêmes entrées génèrent les mêmes sorties [GB03]. Un corollaire important pour nous est que les KPNs toujours être implémentés séquentiellement par un programme séquentiel déterministe.

3.4 Automate de flot de contrôle (AFC)

La fusion est une transformation statique opérant sur l'automate de flot de contrôle (AFC). Nous rappelons dans cette section les propriétés clés qu'un AFC doit satisfaire et présentons un exemple. Les définitions et propriétés de cette section sont une simple adaptation de celles du chapitre 2 pour un KPN.

Un AFC $k^\sharp$ est un automate *fini* $(\Sigma_{k^\sharp}, \alpha_0^\sharp, \mathcal{E}_{k^\sharp}, \longrightarrow_{k^\sharp}, \mathcal{F}_{k^\sharp})$ avec :
- $\Sigma_{k^\sharp}$ un ensemble fini de configurations abstraites ;
- $\alpha_0^\sharp$ la configuration abstraite initiale ;
- $\mathcal{E}_{k^\sharp}$ un ensemble (fini) de commandes ;
- $\longrightarrow_{k^\sharp}$ une relation de transition étiquetée ;
- $\mathcal{F}_{k^\sharp}$ l'ensemble des états finals. Nous considérons que tous les états sont finals, $\mathcal{F}_{k^\sharp} = \Sigma_{k^\sharp}$.

De nombreuses abstractions peuvent être définies en fonction de la taille et de la précision souhaitées pour l'AFC. La fusion de réseaux est générique par rapport à l'abstraction aussi longtemps que l'AFC respecte deux propriétés clés : la *sûreté* et la *fidélité*.

DÉFINITION **3.4.1.** [**Sûreté**] *Soit un réseau k et une abstraction $k^\sharp = (\Sigma_{k^\sharp}, \alpha_0, \mathcal{E}_{k^\sharp}, \longrightarrow_{k^\sharp}, \mathcal{F}_{k^\sharp})$ alors $k^\sharp$ est une abstraction* sûre *de k si et seulement si*

$$\begin{array}{ll} & \forall s_0, alpha_0 \xrightarrow{a_0}_k \ldots \alpha_i \xrightarrow{a_i}_k \ldots \xrightarrow{a_{n-1}}_k \alpha_n) \in \mathit{Trace}[\![k]\!]s_0. \\ & \alpha_0 \approx_{abs} \alpha_0^\sharp \\ \wedge & (\alpha_i \approx_{abs} \alpha_i^\sharp \wedge \alpha_i \xrightarrow{a_i}_k \alpha_j \Rightarrow \exists \alpha_j^\sharp.\ \alpha_j \approx_{abs} \alpha_j^\sharp \wedge \alpha_i^\sharp \xrightarrow{a_i}_{k^\sharp} \alpha_j^\sharp) \end{array}$$

DÉFINITION **3.4.2.** [**Fidélité**] *Soit un réseau k et une abstraction $k^\sharp = (\Sigma_{k^\sharp}, \alpha_0, \mathcal{E}_{k^\sharp}, \longrightarrow_{k^\sharp}, \mathcal{F}_{k^\sharp})$ $k^\sharp$ est une abstraction* fidèle *de k si et seulement si*

$$\begin{array}{ll} & \forall s_0. \exists \alpha_0 \xrightarrow{a_0}_k \ldots \alpha_i \xrightarrow{a_i}_k \ldots \xrightarrow{a_{n-1}}_k \alpha_n) \in \mathit{Trace}[\![p]\!]s_0. \\ & (\alpha_i^\sharp \xrightarrow{c}_{k^\sharp} \alpha_j^\sharp \ \wedge\ \alpha_i \approx_{abs} \alpha_i^\sharp \Rightarrow ((\vee\ c = f?x \ \wedge\ \alpha_1[f] = \epsilon) \vee \alpha_j \approx_{abs} \alpha_j^\sharp \ \wedge\ \alpha_i \xrightarrow{c}_k \alpha_j)) \\ \wedge & (\alpha_1^\sharp \xrightarrow{g}_{k^\sharp} \alpha_2^\sharp \ \wedge\ \alpha_1 \approx_{abs} \alpha_1^\sharp \Rightarrow (\mathcal{G}[\![g]\!]\alpha_1 \Rightarrow \alpha_2 \approx_{abs} \alpha_2^\sharp \ \wedge\ \alpha_1 \xrightarrow{g}_k \alpha_2)) \end{array}$$

Ces propriétés sont similaires à celles proposées dans le chapitre 2, mis à part les opérations sur les files dans la propriété de fidélité. En particulier, une transition étiquetée par une lecture sur une file dans l'AFC ne correspond à l'exécution d'une lecture que si la file est non vide.

EXEMPLE. Une abstraction simple est le graphe de flot de contrôle du KPN où chaque nœud comporte une collection d'intervalles approximant les longueurs possibles des canaux internes aux points de programme que le nœud représente. Plus précisément :

- chaque état d'un composant est abstrait par le point de programme auquel il est associé. L'état (variables) des processus est complètement abstrait ;
- chaque état d'un canal est représenté par un intervalle approximant la longueur de sa file.

L'état de l'AFC est composé de l'état abstrait de chaque processus et chaque file. L'AFC pour notre exemple est donné figure 3.7. Sur cet exemple particulier, l'état de f est toujours approximé par l'intervalle $[0, +\infty[$ (l'information la plus imprécise). L'état initial est (p_0, c_0) (les deux processus sont dans leur état initial). De cet état il est possible d'effectuer la commande $x := x + 1$ qui fait passer l'état dans (p_1, c_0). Des AFCs plus précis pourraient naturellement être définis. En particulier, il peut être intéressant d'approximer les files par leur taille effective quand elles sont bornées afin d'optimiser le processus de fusion. ■

Les nœuds des AFCs représentent les étiquettes des programmes et les étiquettes des arcs représentent les commandes. La concrétisation d'un automate fini $k^\sharp = (\Sigma_{k^\sharp}, \alpha_0^\sharp, \mathcal{E}_{k^\sharp}, \longrightarrow_{k^\sharp})$ en un programme est formalisée par la transformation qui est déjà décrite en section 2.7. La concrétisation respecte les mêmes propriétés que celles du chapitre 2 modulo la gestion des lectures sur une file vide.

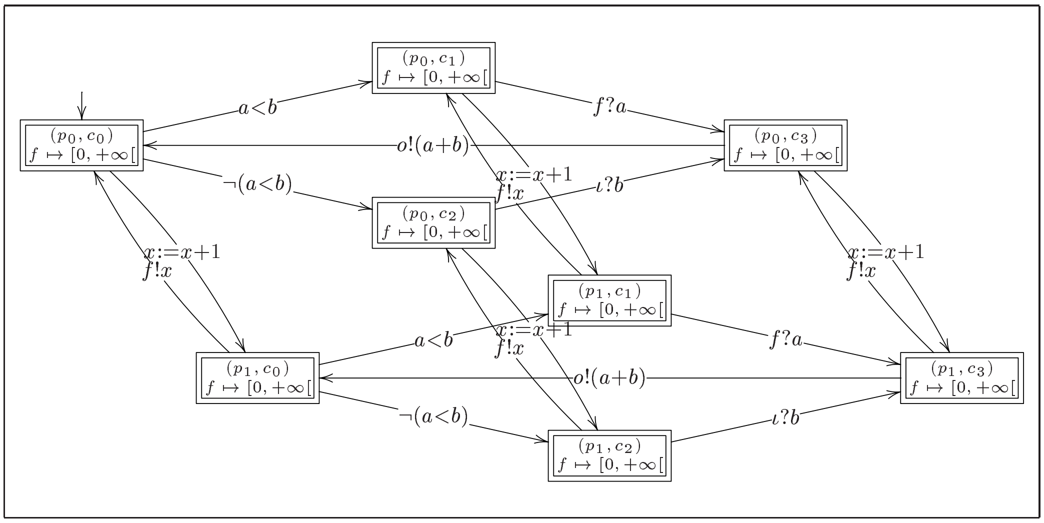


Fig. 3.7 – Exemple d'AFC pour le réseau précédent

3.5 Fusion

Un aspect d'ordonnancement spécifie des contraintes qui définissent un sous-ensemble des traces d'exécution. Les contraintes imposent des choix d'implémentation qui guident et éventuellement optimisent le processus de fusion. Les aspects sont exprimés en fonction des opérations d'entrée-sortie, des propriétés d'équité ou de la taille des files. Ils respectent la nature "boîte noire" des composants. Quand l'aspect séquentialise totalement l'exécution (*i.e.,* fait tous les choix), il spécifie un ordonnanceur séquentiel. En général, les contraintes peuvent être incomplètes et laissent des choix d'implémentation. Une étape supplémentaire est alors nécessaire pour séquentialiser totalement le programme tissé. Cette section décrit successivement les aspects, le produit d'automates et l'étape de séquentialisation.

3.5.1 Aspects d'ordonnancement

Nous décrivons les aspects d'ordonnancement par des automates et un langage permettant de décrire facilement des événements dynamiques et des ensembles d'actions. Un langage d'aspect plus textuel pourrait également être fourni à l'utilisateur. Il suffit simplement que les aspects d'ordonnancement puissent se traduire en automates finis.

Notre formalisme est suffisant pour exprimer une large classe de contraintes d'ordonnancement. Le langage $\mathcal{E}_\lambda$ d'un automate d'aspect contient des symboles de la forme (g, a). avec a une action du programme a et g une garde. La garde g peut être vue comme un événement sémantique comme décrit en section 2.9.3. Nous considérons deux gardes dédiées à l'expression de stratégies d'ordonnancement :

$$g \ ::= \overline{f} \ \ominus \ k \mid \mathfrak{B}_p \mid \ldots \qquad \text{où } \ominus \text{ est un opérateur de comparaison}$$

La première construction permet de comparer la taille d'une file à un entier k. Par exemple, $\overline{f} < 5$ est vrai si le canal f a moins de 5 éléments. La garde $\mathfrak{B}_p$ teste si le processus p est bloqué (par une lecture sur un canal vide ou par une autre contrainte d'ordonnancement).

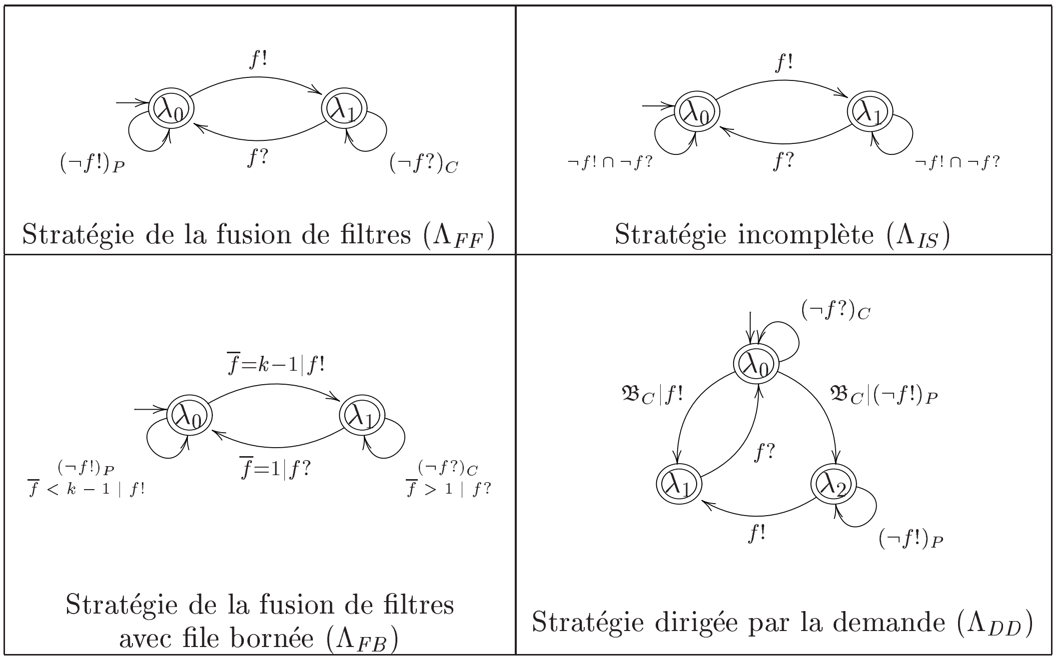
Nous utilisons les notations suivantes afin de spécifier plus facilement des ensembles d'actions :

$$\mathcal{A} \ ::= * \mid [f]? \mid [f]! \mid \neg\mathcal{A} \mid \mathcal{A}_1 \cap \mathcal{A}_2 \mid \mathcal{A}_p$$

où :
- $*$ représente l'ensemble des actions du réseau ;
- ? (resp. !) représente les actions de lecture (resp. écriture) et $f?$ (resp. $f!$) les actions de lecture (resp. écriture) sur la file f ;
- $\neg\mathcal{A}$ est l'ensemble complémentaire de $\mathcal{A}$;
- $\mathcal{A}_1 \cap \mathcal{A}_2$ est l'intersection des ensembles $\mathcal{A}_1$ et $\mathcal{A}_2$;
- $\mathcal{A}_p$ représente la projection de l'ensemble des actions $\mathcal{A}$ sur les commandes du composant p.

Par exemple, $(\neg ?)_p$ représente toutes les actions du composant p sauf les lectures. Ces notations ensemblistes ne sont que du "sucre syntaxique" ; un aspect peut être automatiquement traduit en un automate étiqueté par des commandes standards.

La figure 3.8 montre quatre exemples d'aspects pour un réseau ayant (au moins) deux composants P et C connectés par le canal f. Bien entendu, les aspects d'ordonnancement peuvent être spécifiés pour un nombre arbitraire de composants et canaux.
- L'aspect Λ_{FF} résume dans la stratégie utilisée par la *fusion de filtres* présentée en 3.2. Le producteur P commence et s'exécute jusqu'à une écriture sur f. Le contrôle est alors passé au consommateur C jusqu'à une lecture sur f et ainsi de suite. Cette stratégie borne la taille de la file f à au plus une place ce qui peut introduire des blocages artificiels dans certains réseaux (mais pas pour des compositions linéaires de filtres). Cet aspect séquentialise complètement l'exécution de P et C (il ne laisse aucun choix d'ordonnancement).
- L'aspect Λ_{IS} est similaire à Λ_{FF} excepté que l'exécution de P et C peut être entrelacée arbitrairement entre les écritures et les lectures sur f. Cet aspect laisse des choix d'ordonnancement.
- L'aspect Λ_{FB} est une généralisation de Λ_{FF} à une file f à k places (*i.e.,* P écrit k fois avant de transmettre le contrôle à C). Il s'agit de la formalisation de l'extension de la fusion de filtres proposée dans [CC97].
- L'aspect Λ_{DD} spécifie une stratégie à la demande. Le consommateur C est exécuté jusqu'à ce qu'il essaye de lire sur une file f vide. Ensuite, P est



Stratégie de la fusion de filtres (Λ_{FF})

Stratégie incomplète (Λ_{IS})

Stratégie de la fusion de filtres avec file bornée (Λ_{FB})

Stratégie dirigée par la demande (Λ_{DD})

FIG. 3.8 – Exemples d'aspects d'ordonnancement

exécuté jusqu'à ce qu'il produise une valeur sur f. Le contrôle est ensuite passé à C qui lit immédiatement la valeur sur f et continue.

Notre langage permet d'exprimer un grand nombre de stratégies d'ordonnancement. Il permet également de formaliser succinctement les stratégies de fusion de filtres et son extension qui n'étaient décrites qu'informellement.

3.5.2 Tissage

Tisser l'aspect $\Lambda = (\Sigma_\lambda, \lambda_0, \mathcal{E}_\lambda, \longrightarrow_\lambda, \mathcal{F}_\lambda)$ sur l'AFC $k^\sharp = (\Sigma_{k^\sharp}, \alpha_0^\sharp, \mathcal{E}_{k^\sharp}, \longrightarrow_{k^\sharp}, \mathcal{F}_{k^\sharp})$ s'exprime par le produit dédié $(k^\sharp \times_d \Lambda)$ qui est similaire au produit pour la gestion d'événements sémantiques (comme décrit dans la partie 2.9.3). Cette opération se définit formellement comme suit (nous supposons que les ensembles utilisés dans les aspects sont remplacés par les actions de l'AFC qu'ils représentent) :

$$k^\sharp \times_d \Lambda = (\Sigma_{k\lambda}, (\alpha_0^\sharp, \lambda_0), \mathcal{E}_{k\lambda}, \longrightarrow_{k\lambda}, \mathcal{F}_{k\lambda})$$

avec :

- l'ensemble d'états $\Sigma_{k\lambda} = \Sigma_{k^\sharp} \times \Sigma_\lambda \cup \Omega$ avec Ω représente un ensemble fini d'états supplémentaires ajoutés par le produit dédié ;
- l'état initial $(\alpha_0^\sharp, \lambda_0)$;
- l'alphabet $\mathcal{E}_{k\lambda} = \mathcal{E}_{k^\sharp} \cup \mathcal{T}$ avec $\mathcal{E}_{k^\sharp}$ l'ensemble des actions du programme initial et $\mathcal{T}$ l'ensemble de tests qui sont ajoutés par le produit ;

– la relation de transition $\longrightarrow_{k\lambda}$ est défini par les deux règles suivantes :

$$\frac{\alpha^\sharp \xrightarrow{a}_{k^\sharp} \alpha^{\sharp\prime} \quad \lambda \xrightarrow{a}_\lambda \lambda'}{(\alpha^\sharp, \lambda) \xrightarrow{a}_{k\lambda} (\alpha^{\sharp\prime}, \lambda')}$$

$$\frac{\alpha^\sharp \xrightarrow{a}_{k^\sharp} \alpha^{\sharp\prime} \quad \lambda \xrightarrow{t|a}_\lambda \lambda'}{(\alpha^\sharp, \lambda) \xrightarrow{t}_{k\lambda} \omega \xrightarrow{a}_{k\lambda} (\alpha^{\sharp\prime}, \lambda') \quad t \in \mathcal{T} \quad \omega \in \Omega}$$

– l'ensemble des états finals $\mathcal{F}_{k\lambda}$ est tel que :
$\sigma \in \mathcal{F}_{k\lambda} \implies (\sigma = (\alpha^\sharp, \lambda) \ \wedge \ \lambda \in \mathcal{F}_\lambda) \ \vee \ \sigma \in \Omega$

Une action de l'automate du programme est exécutée seulement si l'aspect peut aussi exécutée cette action (*i.e.,* une transition étiquetée par a ou (t, a)). Les aspects n'introduisent pas de nouvelles commandes mais peuvent introduire de nouveaux tests ($t \in \mathcal{T}$). Dans le cas de l'ajout d'un test t, un état supplémentaire ω est aussi ajouté.

Pour simplifier la présentation, nous avons supposé dans les règles d'inférence ci-dessus que les gardes n'utilisaient pas la condition $\mathfrak{B}_p$. Nous présentons maintenant séparément la règle correspondant à cette condition. La construction $\mathfrak{B}_p$ sert à passer le contrôle à un autre composant quand le processus p est bloqué. Le processus p d'un Kpn est bloqué dans la configuration α s'il y a pas de transitions sortantes de α étiquetées par une commande de p. Pour un AFC, qui est une approximation comportant des faux chemins, une configuration abstraite peut représenter un état de blocage de p même si celle-ci possède une transition sortante étiquetée par une commande de p. En fait, p est bloqué dans une configuration abstraite si les transitions sortantes étiquetées par des commandes de p sont non exécutables : c'est à dire, soit elles ont une garde fausse, soit ce sont des lectures sur une file vide. Formellement,

Définition **3.5.1.** *soit* $a_1, \ldots, a_n$ *toutes les actions de* p *telles que* $\alpha^\sharp \xrightarrow{a_i}_{k^\sharp} \alpha_i^\sharp$ *et* $g_i = \begin{cases} \overline{f} = 0 & \textit{si } a_i = f?x \quad (\overline{f} \textit{ dénote la taille de la file } f) \\ \textbf{false} & \textit{sinon} \end{cases}$
alors la condition nécessaire et suffisante pour que p *soit bloqué dans* $\alpha^\sharp$ *est*

$$b_p(\alpha^\sharp) = \bigwedge_{i=1,\ldots,n} g_i$$

Le produit d'un AFC avec une transition gardée par $\mathfrak{B}_p$ est défini comme suit :

$$\frac{\alpha^\sharp \xrightarrow{a}_{k^\sharp} \alpha^{\sharp\prime} \quad \lambda \xrightarrow{\mathfrak{B}_p|a}_\lambda \lambda'}{(\alpha^\sharp, \lambda) \xrightarrow{b_p(\alpha^\sharp)}_{k\lambda} \omega \xrightarrow{a}_{k\lambda} (\alpha^{\sharp\prime}, \lambda') \quad \omega \in \Omega}$$

La figure 3.9 représente le produit de l'AFC de la figure 3.7 avec l'aspect Λ_{FF}. Le composant P est exécuté jusqu'à ce qu'il produise une valeur sur f,

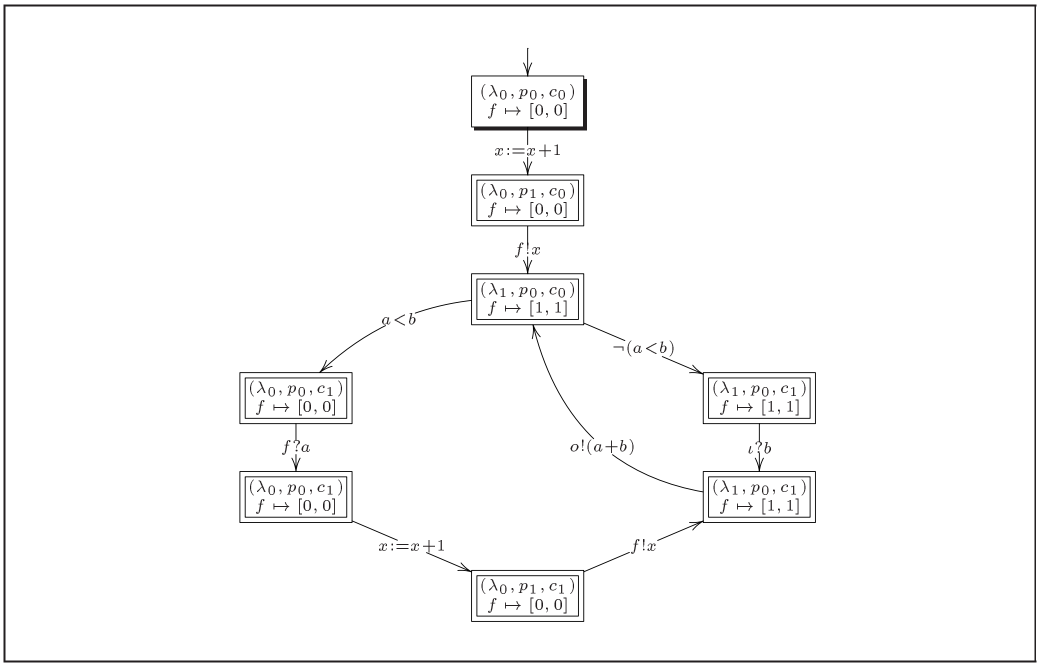


FIG. 3.9 – Tissage de l'exemple avec Λ_{FF}

ensuite C est exécuté jusqu'à ce qu'il lise une donnée sur f. Notons que si la garde $a \geq b$ reste toujours vraie alors P ne sera jamais exécuté. L'exécution n'est pas équitable mais est équivalente car elle génère les exécutions équitable du réseau source (P n'est jamais exécuté si sa production est inutile). L'aspect Λ_{FF} n'utilise pas de gardes ce qui fait qu'aucun nouveau test n'est introduit. Notons également que l'AFC contraint est complètement séquentialisé. La taille des files peut être réestimée sur l'AFC tissé en utilisant des techniques standards d'analyse statique. Ceci peut permettre d'éliminer des tests sur la taille des files. Nous avons indiqué dans la figure 3.9 les nouvelles approximations pour $\overline{f}$. Elles montrent que l'AFC est maintenant borné ($\overline{f} \leq 1$).

Il est facile de vérifier que cet l'AFC tissé peut être traduit par *Conc* (voir la section 3.4) en un programme séquentiel. Comme précisé précédemment, un des buts de la fusion est de supprimer les communications internes. Pour un AFC non borné, les lectures et écritures sur des canaux internes sont remplacées par des affectations sur des listes ou des files fifos. Ici, la file f étant au plus de taille 1 , elle peut donc être implémentée par une unique variable v_f. Les écritures $f!x$ et les lectures $f?x$ sont remplacées par les affectations $v_f := x$ et $x := v_f$. Ces affectations peuvent ensuite être supprimées en utilisant des optimisations classiques [ASU86]. Par exemple, $v_f := x; y = v_f$; sera remplacée par $y := x$.

Finalement, après un renommage des étiquettes, le programme fusionné s'écrit :

$$PC = \left\{ \begin{array}{lclcl} pc_0 & : & x := x+1 & \leadsto & pc_1; \\ pc_1 & : & a < b & \leadsto & pc_2\ ,\ pc_3; \\ pc_2 & : & a := x & \leadsto & pc_4; \\ pc_3 & : & \iota?b & \leadsto & pc_5; \\ pc_4 & : & x := x+1 & \leadsto & pc_5; \\ pc_5 & : & o!(a+b) & \leadsto & pc_1; \end{array} \right\}$$

En fonction de la taille des automates, le tissage par produit d'automates peut causer une explosion du nombre d'états. Ce produit peut également s'implémenter sans duplication d'états comme décrit dans l'extension 2.9.2.

3.5.3 Séquentialisation

En général, le tissage laisse des choix d'exécution qui doivent être résolus avant de produire un programme séquentiel. L'étape de séquentialisation fait ces choix en complétant automatiquement l'ordonnancement de l'exécution des composants. Un ordonnanceur valide doit être équitable (tout composant exécutable doit finalement être exécuté) et séquentiel (l'ordonnancement doit correspondre à un programme séquentiel). Comme les aspects, un ordonnanceur peut s'exprimer comme un automate.

Nous choisissons ici une politique simple et équitable pour séquentialiser les choix restants : l'ordonnancement *"round-robin"*. Les composants sont placés dans une liste circulaire et sélectionnés à tour de rôle. Soit le composant choisi est bloqué (par une lecture ou une contrainte d'un aspect d'ordonnancement), soit une de ses commandes est exécutée. Dans les deux cas, le composant suivant est sélectionné et ainsi de suite. La figure 3.10 formalise la stratégie round-robin comme un LTS gardé pour un réseau avec exactement deux composants P et C. Initialement une ins-

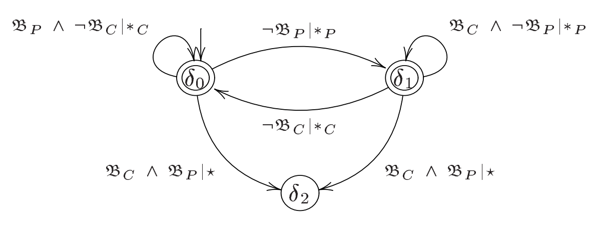


FIG. 3.10 – Ordonnancement Round-Robin

truction de P est exécutée, puis une instruction de C, etc. Si un des deux processus est bloqué alors qu'il a la main, une instruction de l'autre processus est exécutée à la place. Si les deux processus sont bloqués ($\mathfrak{B}_C \wedge \mathfrak{B}_P$) alors il s'agit d'un blocage artificiel qui correspond à une transition dans un état puits d'erreur qui est implémentée par l'arrêt du programme (*c.f.* la définition de la concrétisation). Quand cette transition apparaît dans le résultat de la fusion, l'utilisateur est averti de la possibilité d'un blocage artificiel. Cette stratégie est équitable et assure une séquentialisation complète de l'exécution. Une généralisation de cet automate pour un réseau avec un nombre quelconque de processus est aisée.

La stratégie de la figure 3.10 est appliquée en utilisant la même opération de produit que précédemment. Considérons l'ordonnancement de l'AFC original de la figure 3.7 pour lequel l'utilisateur n'a spécifié aucun aspect d'ordonnancement. L'étape de séquentialisation donne l'AFC ordonnancé obtenu (après simplifications) de la figure 3.11. Des simplifications sont nécessaires car le produit introduit des transitions inutiles. Le processus P n'est jamais bloqué (il n'essaye jamais de

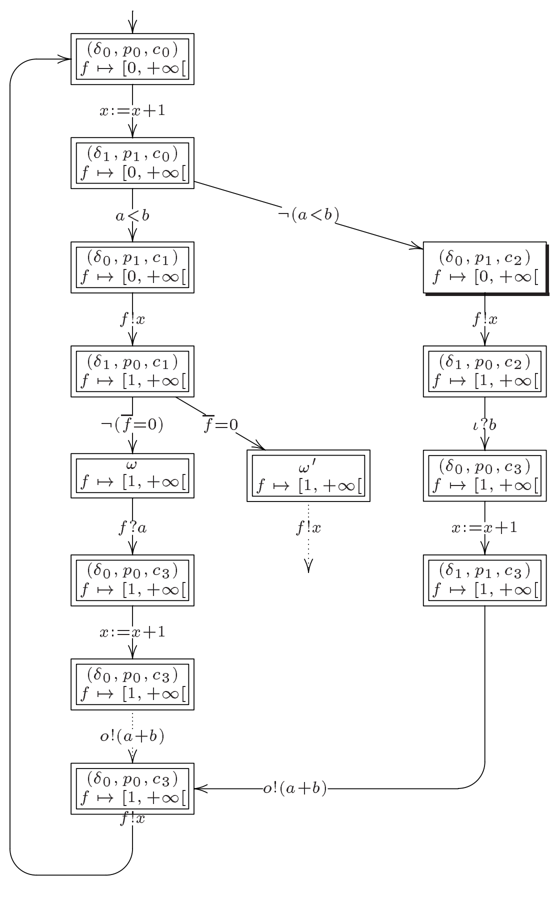


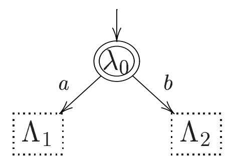
FIG. 3.11 – Séquentialisation de l'exemple avec la stratégie Round-Robin

lire), donc l'exécution peut commencer par $x := x + 1$. L'exécution peut ensuite continuer avec le processus C qui exécute le test $a < b$, puis par le processus P qui exécute l'écriture $f!x$. Ensuite, il y a deux cas : soit le test était faux $(\neg(a < b))$, C n'est pas bloqué et son action $(\iota?b)$ peut être exécutée, soit le test était vrai $(a < b)$ et C peut être bloqué sur une lecture du canal vide f. Si le canal f est vide $(\overline{f} = 0)$, la lecture est bloquante et la stratégie round-robin transmet le contrôle à P qui exécute l'action $x := x + 1$. Notons que l'on peut déduire par analyse statique que cette branche ne sera jamais prise et simplifier le programme obtenu. Si le canal f est non vide $(\neg\overline{f} = 0)$, la lecture sur le canal f peut être effectuée. Le produit continue de manière similaire pour donner, contrairement au produit avec Λ_{FF}, un programme résultant qui est équitable mais non borné. Ici, comme P est exécuté systématiquement en alternance avec C, des données s'accumulent de manière non bornée sur le canal f.

La correction du processus d'ordonnancement vient du fait que le produit avec l'ordonnanceur round-robin génère un AFC séquentiel, équitable et fidèle. Des politiques plus sophistiquées (utilisant plusieurs files, basées sur des priorités statiques ou dynamiques) peuvent être utilisées sans changer le reste du processus de fusion. D'autres ordonnancements équitables recherchant autant que possible une exécution bornée pourraient être considérés. Dans un cadre de programmes standards qui terminent, l'utilisation de files non bornées n'est pas critique. Comme le tissage des aspects, la séquentialisation peut être implémentée, soit par un produit classique, soit par un produit instrumenté.

3.5.4 Classe des aspects acceptables

Un aspect peut changer la sémantique du KPN. Par exemple, un aspect qui borne les canaux de communication causera un blocage artificiel s'il est appliqué sur un réseau non borné. L'utilisateur peut cependant vouloir tisser l'aspect même au prix de blocages potentiels. Nous considérons que de tels changements de sémantique sont acceptables tant qu'ils dépendent de l'utilisateur et restent sous son contrôle. Cependant cela nécessite de prendre certaines précautions. En effet, considérons l'aspect suivant :



où a et b sont deux commandes non mutuellement exclusives et Λ_1 et Λ_2 représentent des contraintes différentes. Puisque a et b peuvent être exécutés indifféremment, un choix est fait ensuite à la séquentialisation. Cependant en fonction de ce choix, les contraintes qui sont ensuite appliquées (Λ_1 ou Λ_2) sont différentes. Par exemple, Λ_1 pourrait impliquer un blocage artificiel dans certains cas (non statiquement déterminables) et Λ_2 dans d'autres cas. En d'autres mots, la sémantique du programme résultat dépend d'un choix fait à l'aveugle pendant la séquentialisation. De tels changements sont problématiques car ils sont hors du contrôle de l'utilisateur.

Notre solution est de restreindre la classe des aspects acceptables. Si un aspect laisse un choix non déterministe (comme entre a et b ci-dessus), alors il faut s'assurer que tous les processus pourront encore évoluer de la même façon quelque soit le choix effectué. A chaque choix (par ex. a ou b) correspond un ensemble (langage) de traces acceptables (par ex. $a.\mathcal{L}(\Lambda_1)$ et $b.\mathcal{L}(\Lambda_2)$). Un aspect est acceptable si pour chaque choix, les projections des langages correspondant sur les commandes d'un processus quelconque sont équivalentes. Sur l'exemple précédent, nous devons nous assurer que :

$$\forall p.\;\; a.\mathcal{L}(\Lambda_1) \downarrow p = b.\mathcal{L}(\Lambda_2) \downarrow p \qquad \text{(avec } \downarrow \text{ dénotant l'opération de projection)}$$

Avec cette condition, la séquentialisation automatique n'a pas d'impact sémantique. Tous les aspects de la figure 3.8 sont acceptables. Cette propriété est claire pour Λ_{FF}, Λ_{FB} et Λ_{DD} puisqu'ils ne laissent pas de choix non déterministes. Pour Λ_{IS}, les deux transitions étiquetées par $\neg f! \cap \neg f?$ laissent le choix d'exécuter P ou C. Cependant, à chaque fois, ces choix conduisent au même état (et donc génèrent les mêmes langages).

3.6 Conclusion

Un des points de départ de ce travail fut la fusion de filtres [PW96]. Cet algorithme considère des réseaux très spécifiques (des filtres en pipeline) et une stratégie d'ordonnancement fixée. Notre travail peut être vu comme une formalisation de la fusion de filtres basée sur les automates et un produit d'automates et, surtout, comme une généralisation à des réseaux arbitraires et à des stratégies définies par l'utilisateur. L'application de notre technique sur des filtres en pipeline avec l'aspect Λ_{FF} (figure 3.8) est équivalent à la fusion de filtres. L'extension proposée dans [CC97] est formalisée simplement dans notre cadre par l'aspect Λ_{FB} (figure 3.8).

Certaines transformations de programme de langages fonctionnels, comme la déforestation [Wad84, Wad90], ont des similarités avec notre technique. Comme la fusion de filtres, ces transformations considèrent des paires producteur-consommateur et ont une stratégie fixe. Le domaine des systèmes réactifs et embarqués a engendré de nombreux travaux sur l'ordonnancement statique. Lin [Lin98] étudie l'ordonnancement statique de processus synchrones. Cortadella *et al.* [CKL+99] ainsi que Strehl *et al.* [STZE99] considèrent l'ordonnancement statique de processus asynchrones formalisés par des réseaux de pétri. Comme Parks [Par95], ils s'intéressent à trouver des ordonnancements bornés et ne prennent pas en compte des stratégies définies par l'utilisateur. Strehl *et al.* [TSZ+99] proposent un modèle de conception qui permet la spécification de composants couplée à des contraintes d'ordonnancement. Ils dérivent un ordonnanceur séparé mais ne considèrent pas la fusion du système de composants.

Nous avons présenté une technique générique et flexible pour fusionner des réseaux de composants. Cette technique est une application directe de l'approche proposée dans le chapitre 2. Son utilisation permet de réconcilier la modularité des réseaux de composants et la performance d'un programme séquentiel. La fusion s'applique à une large classe de réseaux (KPNs) en fonction d'aspects d'ordonnancement définis par l'utilisateur. Elle peut être paramétrée par différentes abstractions, aspects et stratégies d'ordonnancement.

Des extensions à ce travail peuvent être envisagées. Récemment, un stage à l'INRIA Rhône-Alpes a consisté à étendre cette technique à des réseaux non déterministes. Le modèle de départ est un réseau de Kahn où les processus peuvent tester la taille des files (ce qui rend le modèle non déterministe). Un des buts de ce stage fut d'étendre la séquentialisation afin de produire un programme séquentiel (non déterministe) équivalent au réseau initial. Le but est de conserver tous les comportements possibles à l'aide d'une instruction séquentielle "choice" effectuant un choix non déterministe entre deux étiquettes.

La preuve de correction associée à cette technique reste à faire. Comme dans le chapitre 2, il faut montrer que la sémantique du réseau de base modulo les contraintes et la sémantique du réseau tissé sont égales. Comme les différences avec les propriétés du chapitre 2 se limitent à des ajouts pour ne pas avoir de lecture sur une file vide, nous pensons que les preuves de sûreté et de fidélité du tissage ont la même structure que celle du chapitre 2. La propriété de correction du tissage du chapitre 2 concernait des programmes déterministe et séquentiel qui terminent et nécessite d'être adaptée pour des réseaux de Kahn. Il serait nécessaire de poser en plus des propriétés sur les aspects définis par les utilisateurs de telle sorte que les traces du programme tissé soient équitables et maximales et ainsi assurer l'égalité sémantique avec le programme initial.

Nous avons implémenté un petit prototype afin de valider l'approche expérimentalement. Comme la fusion prend en compte uniquement les instructions d'entrée-sortie, la plupart des programmes peuvent s'abstraire en de petits automates. Avec l'utilisation du produit instrumenté dans les cas problématiques (*i.e.*, explosion de la taille du code), la fusion peut produire des programmes à la fois efficaces et de tailles raisonnables dans la pratique. Une perspective intéressante de ce travail serait d'appliquer la fusion sur un véritable réseaux de composants industriels afin de valider les gains de performance.

Plus généralement, nous voyons la fusion de réseaux comme un élément d'un environnement plus général d'assemblage de composants. La première extension vers cet environnement consisterait à définir de nouvelles constructions linguistiques pour mieux contrôler la fusion (*i.e.*, la taille et l'efficacité des programmes fusionnés). Une seconde extension consisterait à définir un langage de description d'architecture pour spécifier l'assemblage (*i.e.*, les ports et les connections) des composants.

Chapitre 4

Aspects de disponibilité

Sommaire

4.1 Introduction

Avec la confidentialité et l'intégrité, la *disponibilité* est l'une des trois grandes classes de propriétés de sécurité. La disponibilité assure que les demandes faites par des sujets autorisés sont toujours traitées ; on dit qu'il n'y a pas de *dénis de service* (DoS[1]). Dans ce chapitre, nous étudions la gestion des ressources en isolation (*i.e.,* séparée de la fonctionnalité de base) et la disponibilité (*i.e.,* la prévention des dénis de service) comme des aspects.

Le temps joue clairement un rôle central dans les propriétés de disponibilité. Ce chapitre peut se voir comme une adaptation de l'approche du chapitre 2 à des propriétés temporisées[2] où le temps est explicite. Nous proposons un langage d'aspects dédié permettant de prévenir les dénis de service liés à la gestion des ressources (famines, interblocages). Nos aspects spécifient des politiques de disponibilité en imposant des limites de temps d'allocation des ressources. Par exemple, on peut vouloir imposer qu'un service n'accapare pas la ressource `R` plus de 10 secondes ou qu'il alloue impérativement la ressource `R1` avant la ressource `R2`. A notre connaissance, ce travail est le premier à proposer une approche orientée aspect pour la disponibilité et la gestion des ressources.

Dans notre langage, un aspect peut être vu comme une propriété temporisée sur les traces d'exécution. A la différence des chapitres 2 et 3 qui reposaient sur les automates à états finis, les programmes et les aspects sont maintenant modélisés par des *automates temporisés* [AD94, HNSY92]. Dans ce cadre étendu, l'aspect décrit l'ensemble des traces temporisées autorisées. Ceci permet de voir le tissage comme un *produit d'automates temporisés* (*i.e.,* une intersection des traces d'exécution temporisées) qui restreint le comportement du programme aux traces autorisées par l'aspect. L'avantage principal de cette approche formelle est double :

- comme précédemment, elle permet de garder la maîtrise de l'impact sémantique du tissage et de raisonner sur les programmes tissés ;
- elle permet d'utiliser des outils automatiques de model-checking (par ex. UPPAAL [LPY97, BY04]) pour optimiser le tissage et vérifier des propriétés de disponibilité. Ce type d'outil est particulièrement intéressant dans ce contexte où il n'est pas toujours évident que les politiques de disponibilité imposées par les aspects garantissent la disponibilité attendue.

Ce chapitre est organisé comme suit. La section 4.2 rappelle rapidement les notions de base de la disponibilité et présente deux articles à la base de notre travail. La section 4.3 présente les systèmes et les problèmes de disponibilité considérés, notre approche et l'exemple utilisé au fil du chapitre pour illustrer les différentes étapes. En section 4.4, nous rappelons les caractéristiques du formalisme central

[1] Nous utilisons l'abbréviation anglaise DoS pour *Denial of Service*

[2] A ne pas confondre avec des propriétés temporelles qui portent sur l'ordre d'exécution des événements et dans lesquelles le temps n'est pas explicite.

à cette étude : les automates temporisés. La section 4.5 présente comment les ressources sont spécifiées et décrit deux exemples classiques de ressources. Les sections 4.6 et 4.7 présentent la syntaxe et la sémantique des services et les aspects de disponibilité, respectivement. La section 4.8, cœur technique du chapitre, décrit l'abstraction des services en automates temporisés, la sémantique des aspects en termes d'automates temporisés, le tissage comme un produit, ainsi que des optimisations et des vérifications sur l'automate résultant. La concrétisation de cet automate en un code source utilisant des commandes temporisées (minuteurs, attentes et interruptions) est esquissée en section 4.9. Nous discutons rapidement des travaux connexes et des extensions possibles en conclusion (section 4.10).

4.2 Rappels sur la disponibilité

Les systèmes informatiques sont de plus en plus souvent la cible d'actions malveillantes, qu'il s'agisse de tentatives d'intrusion externes ou d'attaques internes. Néanmoins, ces systèmes doivent rester disponibles ; c'est à dire continuer à fournir leurs services même s'ils sont attaqués. Il s'agit d'un enjeu important dont dépend en grande partie la crédibilité des futures infrastructures informatiques. La *disponibilité* assure que les demandes faites par des sujets autorisés sont toujours traitées ; on dit qu'il n'y a pas de *dénis de service*. Une première classification des dénis de service consiste à séparer les attaques à la disponibilité selon la ressource ciblée :

- les communications (erreur de routage, liaison défectueuse),
- le matériel (panne de courant, défaillance matérielle),
- l'application (monopolisation des ressources, "flooding").

Les dénis de service des communications et du système sont gérés par des techniques de tolérance aux fautes [Lap92, Rus94] (la redondance matérielle, les points de reprise ou les transactions). Nous nous concentrons sur les dénis de service des applications où les dénis de service sont généralement causés par une mauvaise utilisation des ressources partagées. Supposons deux applications qui nécessitent la même ressource pour fonctionner ; un mécanisme extérieur aux applications devra assurer un partage équitable de la ressource entre les deux applications. Pour effectivement rendre disponible une application, il serait nécessaire de combiner des mécanismes de gestion de ressources avec des techniques de tolérance aux fautes. La prévention des dénis de service causés par les ressources a été relativement peu étudiée. Nous présentons dans la suite les deux travaux qui ont été le point de départ de nos réflexions sur les aspects d'ordonnancement.

Une méthode de spécification et vérification pour prévenir les dénis de service [YG90]

Yu et Gligor ont proposé une méthode pour vérifier qu'un allocateur de ressources reste disponible. Notre approche peut être vu comme une généralisation de leur travail à des politiques en temps borné. Dans leur modèle, un dénis de service est défini comme une interaction entre un ou plusieurs utilisateurs qui permet de bloquer indéfiniment l'accès à la ressource à d'autres utilisateurs. Les systèmes considérés sont composés d'utilisateurs et de ressources. Les ressources sont spécifiées formellement par :

- les requêtes utilisateurs. Par exemple, une ressource accessible en exclusion mutuelle (`mutex`) définit deux requêtes prendre et relacher ;
- un état interne décrit par un ensemble de variables. Pour une ressource `mutex`, il faut un booléen indiquant si la ressource est occupée et une autre variable qui détermine l'utilisateur qui possède la ressource ;
- des propriétés qui déterminent la bonne utilisation de la ressource. Par exemple, les requêtes aux ressources sont traitées en exclusion mutuelles selon un ordre FIFO.

Les utilisateurs sont spécifiés par la suite des requêtes qu'ils effectuent. Ce modèle s'intéresse à des propriétés de disponibilité en temps d'attente fini ; c'est à dire que la ressource sera finalement (au bout d'un temps fini mais non borné) accordée ou que la requête sera finalement traitée. Les propriétés de disponibilité pourraient également être en temps borné afin d'exprimer des contraintes plus forte.

Les auteurs montrent qu'il est nécessaire de poser des contraintes sur les utilisateurs afin d'établir une propriété de disponibilité de la ressource. Le contrat utilisateur[1] définit les propriétés (pouvant être de vivacité) que les utilisateurs doivent vérifier. Par exemple, dans le cas d'une ressource `mutex`, un utilisateur qui effectue une requête prendre doit ensuite effectuer une requête relacher. Leur méthode de spécification permet de montrer que la spécification du système et les contrats utilisateurs assurent la propriété de disponibilité. Pour une ressource `mutex`, cela revient à montrer que toutes les requêtes prendre et relacher sont finalement traitées ; les auteurs montrent que cette politique est respectée si le contrat utilisateur impose que toute requête prendre est suivie par une requête relacher.

Un modèle d'allocation de ressources pour la prévention des dénis de service [Mil94]

Le modèle proposé par Yu et Gligor permet de représenter des politiques en temps fini, mais ne permet pas d'exprimer que la requête soit traitée avant cinq secondes. Le modèle de Millen propose d'autres types de politiques basées sur des temps d'attente maximum (la ressource sera accordée avant un délai donné) et des temps d'attente probabilistes (la ressource sera accordée selon une loi de probabilité). L'auteur montre qu'il est possible d'imposer des propriétés de disponibilité à l'aide d'un moniteur de l'ensemble des ressources. Ce moniteur global repose sur une *Denial Protection Base* (DBP) qui étend la notion habituelle de *Trusted Computing Base* (TCB). Les hypothèses d'une DPB sont que toutes les requêtes

[1] *user agreement* en anglais

de ressources passent par le moniteur et que le moniteur puisse annuler l'accès à une ressource. L'auteur constate aussi que la garantie d'accéder à une ressource n'est pas suffisant pour prévenir des dénis de services, car l'accès à la ressource pourrait être annulé juste après qu'il soit accordé. Il est nécessaire de définir une notion de tâche qui spécifie les ressources nécessaires mais aussi les besoins en temps de possession de la ressource.

Notre approche adapte le cadre proposé par Yu et Gligor à des politiques en temps borné. En particulier, nos services peuvent se voir comme leurs utilisateurs. Les aspects modélisent ainsi se les contrats utilisateurs et peuvent être tissés sur les services. De la même manière, nous utilisons les contrats utilisateurs et la spécification du système pour déduire la propriété de disponibilité du système. Comme les moniteurs de Millen, nous nous intéressons à des propriétés en temps borné. Néanmoins nous montrons que des politiques locales peuvent aussi être utilisées pour assurer la disponibilité. Les politiques locales de disponibilité que nous proposons ont l'avantage d'être facilement compréhensibles et tissables.

4.3 Cadre général

Nous commençons par décrire les systèmes dont nous voulons assurer la disponibilité. Nous présentons ensuite les grandes lignes de notre approche et l'exemple nous servant à illustrer les principales étapes par la suite.

4.3.1 Systèmes et disponibilité

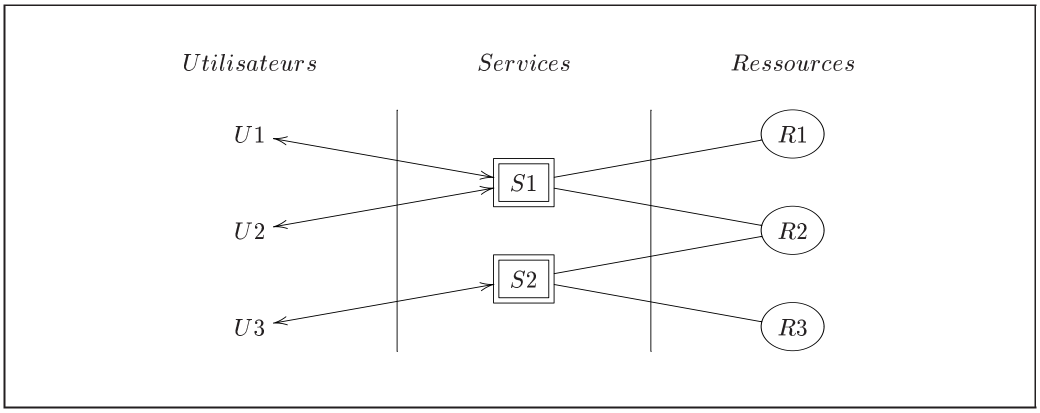


FIG. 4.1 – Modélisation en 3 couches

Les systèmes considérés sont structurés selon trois couches : les *utilisateurs*, les *services* et les *ressources* (figure 4.1). Les utilisateurs envoient des requêtes aux services et attendent la réponse à leur requête. Les services traitent séquentiellement les demandes des utilisateurs. Les requêtes des utilisateurs à un service sont mémorisées (par ex. dans une file FIFO) ; le traitement d'une requête par un service entraîne du calcul et le plus souvent des accès à des ressources. Les ressources sont des entités (logiques ou physiques) qui sont partagées entre les différents services. On peut citer comme exemples de ressources, des fichiers, une imprimante, un processeur ou encore le gestionnaire de connexions réseau. Notre modèle correspond à une architecture client-serveur. Ce type d'architecture est couramment utilisé et se retrouve, par exemple, sur la majorité des serveurs sur le web et applications distribuées. Nous faisons l'hypothèse d'un nombre fixé et connu de services et de ressources. Cette hypothèse, simplificatrice mais raisonnable, correspond, par exemple, au fonctionnement d'un serveur web [BDM99].

Chaque service peut être vu comme une boucle sans fin de traitement de requêtes : la requête d'un utilisateur est lue, le traitement correspondant est réalisé, le résultat est retourné à l'utilisateur, et ainsi de suite. Notre but étant la gestion des ressources et la prévention des dénis de service, nous ne décrivons pas les utilisateurs (qui peuvent être des processus quelconques) ni leurs interactions avec les services. Nous nous focalisons sur les interactions entre les services et les ressources. Les problèmes de disponibilité auxquels nous nous intéressons proviennent de la concurrence entre les services pour accéder aux ressources. Il s'agit, par exemple, de famine quand un service ne réussit pas à accéder à une ressource ou d'interblocage quand tous les services sont bloqués en attente des ressources possédées par d'autres services. Ces problèmes sont d'ordre logiciel et peuvent être prévenus par une gestion des ressources adaptée. La politique de disponibilité des ressources peut se décrire séparément comme un aspect et s'implémenter par tissage.

4.3.2 Approche

Notre système de gestion des ressources est composé de deux parties inspirées du modèle de Yu et Gligor. La première partie consiste en la spécification des ressources et leur traitement des requêtes. Les ressources sont spécifiées en termes d'automates UPPAAL suffisamment précis pour être traduits en programmes. La seconde partie décrit les contraintes que les services doivent respecter dans leur utilisation des ressources. Nous introduisons dans ce but des *aspects de disponibilité* qui sont tissés sur les services. L'originalité de ce type d'aspects est d'intégrer des notions temporelles pour pouvoir, par exemple, limiter le temps d'allocation d'une ressource à un service ou interdire des réallocations de ressources trop rapprochées (section 4.7). La figure 4.2 récapitule la structure de notre système de gestion de ressources. Celui-ci est constitué de la spécification des différentes ressources et des aspects de disponibilité.

Dans ce chapitre, nous décrivons succinctement les ressources et nous nous concentrons sur la partie orientée aspect du cadre. Les contraintes de gestion des

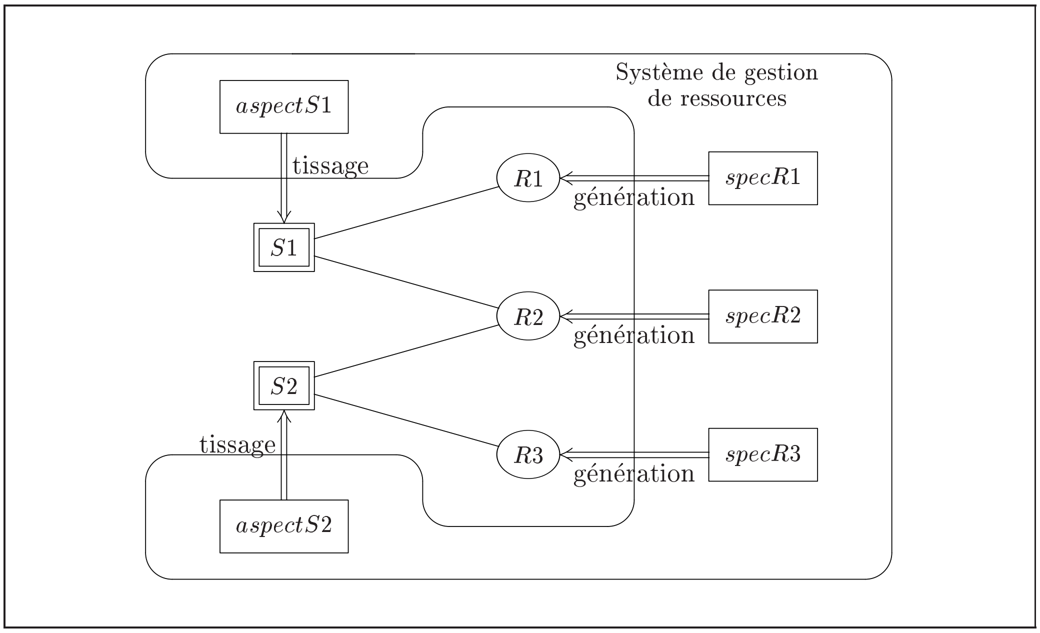


Fig. 4.2 – Vision globale du système avec gestion des ressources séparée

ressources sont spécifiées à l'aide d'un aspect par service. Chacun de ces aspects est indépendant et définit une propriété locale qui sera tissée sur le service. Ces aspects correspondent aux *user agreements* de Yu et Gligor. Nous avons fait le choix de ne pas considérer d'aspects globaux qui contraignent certains services en fonction du comportement d'autres. Les aspects globaux sont certes plus expressifs et plus précis mais ils ne sont pas facilement tissables sur les codes des différents services. Ils sont plus naturellement mis en œuvre par un moniteur observant l'exécution du système complet. Nous nous concentrons ici sur les aspects locaux qui sont suffisamment expressifs pour prévenir les dénis de service et dont l'implémentation peut être optimisée en les tissant statiquement.

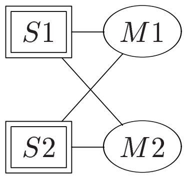
Le formalisme sous-jacent à toute l'approche est celui des automates temporisés et l'opération centrale de tissage est le produit d'automates temporisés. Le cœur technique de notre approche est similaire au chapitre 2 et est composé des phases suivantes :

- le service est abstrait en un automate temporisé sur-approximant ses traces d'exécution et son comportement temporisé (section 4.8.1) ;
- si les aspects sont spécifiés dans une syntaxe textuelle, leur sémantique est donnée par un automate temporisé (section 4.8.2) ;
- l'aspect est intégré au service en faisant le produit des deux automates. Le produit effectuant une intersection des traces d'exécution, l'automate résultant respectera le comportement du programme *et* les contraintes de l'aspect (section 4.8.3) ;

- afin d'optimiser l'automate obtenu, des informations sur les durées d'exécution des instructions du service sont prises en compte en utilisant une nouvelle fois le produit d'automates temporisés (section 4.8.4) ;
- à cette étape, il est possible de montrer automatiquement (par ex. avec UPPAAL) que le système possède les propriétés de disponibilité attendues (section 4.8.5) ;
- la dernière étape consiste à concrétiser l'automate (optimisé et vérifié) en code source instrumenté avec des commandes temporisées (minuteurs, attentes et interruptions) (section 4.9).

4.3.3 Exemple de système

Réseau



Services

```
       ⎧ l1 :  M1.prendre(); M2.prendre();
S1 =   ⎨       S1Calcul;
       ⎪       M2.relacher(); M1.relacher();
       ⎩       jmp l1;

       ⎧ l2 :  M2.prendre();
       ⎪       S2Calcul1;
       ⎪       if G then  M1.prendre();
S2 =   ⎨                  S2Calcul2;
       ⎪                  M1.relacher();
       ⎪       M2.relacher();
       ⎩       jmp l2;
```

Fig. 4.3 – Exemple de système à deux services et deux ressources

L'exemple de la figure 4.3 nous sert par la suite à illustrer les différentes étapes (abstraction, tissage, vérification, concrétisation). Ce petit système est composé de deux ressources (`M1` et `M2`) et de deux services (`S1` et `S2`) de type boucle sans fin. Les deux ressources sont supposées être des `mutex` ; à un instant donné, un seul processus peut posséder la ressource (une spécification complète de ce type de ressource est fournie en section 4.5.1). Le service `S1` commence par prendre la ressource `M1` puis `M2` (`M1.prendre() ; M2.prendre() ;`). Ensuite il effectue le calcul `S1Calcul` (qui dure entre 2 et 10 secondes), relâche les ressources `M2` puis `M1` et réitère. Le service `S2` modélise un service potentiellement dangereux. Il commence par prendre la ressource `M2` (`M2.prendre()`). Ensuite il effectue le calcul `S2Calcul1` qui dure au moins 1 seconde (et peut ne pas terminer). Si la garde `G` est vraie, il prend `M1`, effectue le calcul `S2Calcul2` (qui dure entre 3 et 20 sec.) et relâche `M1`. Il termine en relâchant `M2` et réitère.

Deux problèmes de disponibilité liés aux ressources peuvent apparaître dans ce système :
- un dénis de service de type famine peut se produire si `S2Calcul1` ne termine pas. Dans ce cas, le service `S2` ne relâche pas la ressource `M2` ce qui peut bloquer le service `S1` ;
- un interblocage peut également se produire : le service `S1` possède la ressource `M1` et attend la ressource `M2` alors que le service `S2` possède la ressource `M2` et attend la ressource `M1`.

Nous montrons que ces deux problèmes peuvent être résolus en spécifiant et en tissant sur `S1` des aspects qui interdisent d'accaparer les ressources `M1` et `M2` plus de 25 et 35 secondes respectivement.

4.4 Les automates temporisés

Nous présentons dans cette section la syntaxe et la sémantique des automates temporisés, formalisme utilisé pour modéliser les programmes, les aspects et le tissage. Les automates temporisés permettent de spécifier le système et de vérifier des propriétés, où le temps est explicite. Nous décrivons les automates temporisés avec invariants d'état (*Timed Safety Automata*) [LPY97, Alu99] qui sont un modèle d'automates temporisés couramment utilisé.

4.4.1 Syntaxe

Nous notons $\mathcal{H}$ un ensemble de variables à valeurs réelles servant à représenter les horloges. Une *contrainte d'horloge* C est de la forme

$$C ::= x \odot k \ \mid \ x - y \odot k \quad \text{avec } x, y \in \mathcal{H}, \quad k \in \mathbb{N} \ \text{ et } \ \odot \in \{\leq, <, =, >, \geq\}$$

Il est uniquement possible de comparer les horloges ou la différence entre deux horloges à une constante entière. Les transitions des automates temporisés sont gardées par un ensemble de contraintes d'horloges qui doit être interprété comme la conjonction des différentes contraintes. Nous notons 2^C l'ensemble des gardes possibles (*i.e.,* les ensembles des contraintes d'horloges).

Un automate temporisé A est un n-uplet $(Q, q_0, H, \Sigma, \delta, I)$ où :
- Q est un ensemble fini d'états ;
- $q_0 \in Q$ est l'état initial ;
- H est un ensemble fini d'horloges ($H \in \mathcal{H}$) ;
- Σ est un alphabet fini dénotant les actions de l'automate ;
- $\delta \subseteq Q \times 2^C \times \Sigma \times 2^H \times Q$ est la relation de transition ;
- $I : Q \to 2^C$ est une fonction associant une garde (un invariant) aux états.

Une transition $(q, g, a, r, q') \in \delta$ spécifie que l'on peut passer de l'état q à l'état q', en effectuant l'action a et en remettant l'ensemble d'horloges r à zéro ($r \in H$) si la garde g est vraie. Le sous-ensemble des horloges r est appelé une *remise à zéro* *(raz)*. Nous restreignons les invariants d'état à des conjonctions de contraintes de la forme $x \leq k$ ou $x < k$ avec k un entier naturel. Nous dénotons par le même symbole . à la fois la garde vide (*i.e.*, $\emptyset$ ou **true**), la raz vide (*i.e.*, $\emptyset$) et l'action vide, notée plus traditionnellement ϵ ($\epsilon \notin \Sigma$). Nous utilisons également la notation $q \xrightarrow{g,a,r} q'$ pour les transitions ; par exemple, $q \xrightarrow{.,.,.} q'$ modélise la transition spontanée qui peut être activé à n'importe quel instant et qui n'a pas d'effet.

4.4.2 Sémantique

La sémantique opérationnelle d'un automate temporisé $(Q, q_0, H, \Sigma, \delta, I)$ est donnée par un système de transition où un état (q, u) est composé de l'état courant $q \in Q$ de l'automate temporisé et la fonction $u : H \to \mathbb{R}$ associe les horloges à leur valeur courante. L'état initial est composé de l'état initial de l'automate temporisé et de la fonction associant 0 à toutes les horloges.

Pour définir la relation de transition, nous utilisons les notations suivantes :
- $u \in g$ signifie que les valeurs des horloges définies par u vérifient la garde g ;
- $u + d$ signifie que l'on incrémente toutes les horloges par d ;
- $u[r \mapsto 0]$ signifie que les horloges de l'ensemble r sont remises à 0.

Les transitions sont :
- soit des transitions représentant un écoulement du temps

$$(q, u) \to (q, u + d) \text{ si } \forall d' : O \leq d' \leq d \Rightarrow u + d' \in I(q)$$

- soit des transitions représentant l'exécution d'une action

$$(q, u) \to (q', u') \text{ s'il existe } q \xrightarrow{g,a,r} q' \in \delta \text{ tel que } u \in g, u' \in I(q'), u' = u[r \mapsto 0]$$

Un écoulement du temps est possible uniquement s'il respecte l'invariant de l'état courant. Une transition de l'automate peut être effectuée si et seulement si sa garde et l'invariant du nouvel état sont vérifiés. La sémantique de l'automate temporisé est donnée par l'ensemble des traces de ce système de transition associé. Cette sémantique est infinie et non déterministe. En particulier, il est possible d'une part d'effectuer n'importe quel écoulement de temps tant que l'invariant est respecté, et d'autre part d'effectuer plusieurs écoulements à la suite.

4.4.3 Produit d'automates temporisés

Le produit de deux automates temporisés $X = (Q_x, x_0, H_x, \Sigma, \to_x, I_x)$ et $Y = (Q_y, y_0, H_y, \Sigma, \to_y, I_y)$ avec le même ensemble d'actions Σ et des horloges disjointes ($H_x \cap H_y = \emptyset$), est l'automate $X \otimes Y = (Q_x \times Q_y, (x_0, y_0), H_x \cup H_y, \Sigma, \to, I)$ avec :

$$I(x, y) = I_x(x) \cup I_y(y)$$

$$\textsc{Action}\quad \frac{x_1 \xrightarrow{g_x,a,r_x}_x x_2 \qquad y_1 \xrightarrow{g_y,a,r_y}_y y_2}{(x_1,y_1) \xrightarrow{g_x \cup g_y, a, r_x \cup r_y} (x_2,y_2)}$$

$$\textsc{Vide}_1\quad \frac{x_1 \xrightarrow{g_x,\cdot,r_x}_x x_2}{(x_1,y) \xrightarrow{g_x,\cdot,r_x} (x_2,y)} \qquad \textsc{Vide}_2\quad \frac{y_1 \xrightarrow{g_y,\cdot,r_y}_y y_2}{(x,y_1) \xrightarrow{g_y,\cdot,r_y} (x,y_2)}$$

L'ensemble d'états de l'automate produit est le produit cartésien des deux ensembles d'états des automates arguments. L'état initial est constitué des états initiaux des deux automates. L'invariant d'un état du produit est la conjonction des invariants des deux états constituants. Le produit calcule un nouvel automate temporisé qui reconnaît l'intersection des traces de deux automates temporisés. La règle Action modélise le cas où une action est effectuée par les 2 automates. La garde de cette transition est la conjonction des gardes temporisées et l'ensemble des horloges à remettre à zéro est l'union des deux raz. Les règles $\textsc{Vide}_1$ et $\textsc{Vide}_2$ modélisent le cas où un des deux automates effectue une action vide. Dans ce cas, l'automate produit peut évoluer de la même façon indépendamment du second automate. Les traces d'exécution de l'automate produit $X \otimes Y$ est l'intersection des traces des deux automates X et Y.

4.5 Les ressources

Les ressources (de communication, de stockage, de communication, *etc.*) jouent un rôle central dans notre approche. Elles sont spécifiées par :
- une *interface* qui énumère les opérations d'accès à la ressource ;
- un *automate* qui spécifie le comportement de la ressource, en particulier :
  - l'évolution de son état quand les services utilisent les opérations d'accès ;
  - la gestion de la file d'attente des requêtes aux opérations de la ressource.

Nous décrivons ci-dessous deux exemples de ressources. Nous avons choisi des types très communs (`mutex` et `partageable`) et des spécifications simples et directes. Bien d'autres spécifications et types de ressources existent.

4.5.1 Ressource de type `mutex`

Une ressource utilisée par un seul utilisateur à la fois est dite de type `mutex`. Elle est utilisée pour avoir une section critique et protéger la lecture ou l'écriture sur des données partagées. Il s'agit d'un mécanisme de base (comme les moniteurs) pour synchroniser plusieurs processus. Cette ressource est définie par :
- une interface qui recense les méthodes d'accès à la ressource :
  - `reqPrendre()` pour demander d'entrer en section critique ;

   - `traitPrendre()` quand la ressource accorde l'entrée en section critique ;
   - `relacher()` pour ressortir de la section critique.
- un comportement qui spécifie la gestion d'accès à la ressource :
   - les `reqPrendre()` sont accordés selon un ordre FIFO ;
   - les `traitPrendre()` sont effectués dès que la ressource est libre ;
   - les `relacher()` sont effectués sans délai.

Le comportement d'une ressource `mutex` est spécifié précisément par l'automate de la figure 4.4. Pour spécifier les ressources, nous utilisons les automates temporisés d'UPPAAL [LPY97, BY04]. Par rapport aux automates temporisés habituels, les automates d'UPPAAL offrent en plus l'utilisation de variables entières et du sucre syntaxique pour simplifier les automates. Si l'automate est déterministe et complet (tous les cas sont couverts), il est facile de générer du code exécutable d'une telle spécification. Cette approche est plus déclarative qu'un codage direct dans un langage de programmation.

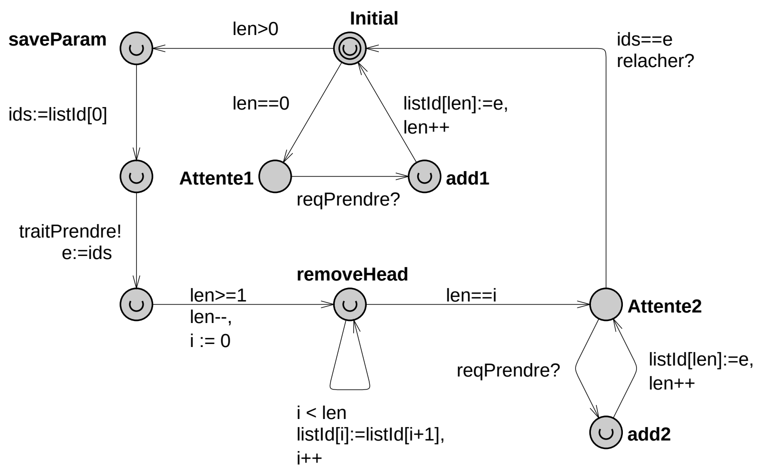


FIG. 4.4 – Spécification de la ressource `mutex` avec UPPAAL

Nous nous contentons de décrire la syntaxe UPPAAL au fil des exemples[1]. Dans la figure 4.4, l'automate manipule la liste `listld` pour implanter la file FIFO des requêtes. La variable interne `len` a pour valeur le nombre de requêtes stockées dans la liste `listld`. La variable partagée `e` dénote l'identité du service effectuant la requête courante. Dans l'état `Initial`, si la file est vide, il faut attendre une requête `reqPrendre()`, sinon la requête `traitPrendre !` du service correspondant au premier `reqPrendre ?` de la file (`listld[0]`) est effectuée (l'identité du service est passé dans `e`). La file FIFO est mise à jour (dans l'état `RemoveHead`) puis

[1]Le lecteur pourra trouver une abondante documentation sur UPPAAL (manuel, didacticiel, articles) sur le site `http ://www.uppaal.com/`

de nouvelles requêtes `reqPrendre?` peuvent être enregistrées dans la FIFO. La requête `relacher` correspondant aux requêtes `reqPrendre()` et `traitPrendre()` précédentes est effectuée (encore une fois, la variable partagée `e` est utilisée pour passer l'identité du service entre ressource et service). Les états étiquetés `U` sont dits "urgents" ; il n'est pas possible d'attendre dans ces états. En terme de programme, ils correspondent à l'opérateur de séquencement " ;". Les autres états correspondent à (et seront traduits par) l'attente d'une requête faite par un service.

Le comportement décrit par l'automate est complet et déterministe. Il peut se traduire directement en un code séquentiel. Nous indiquons ci-dessous uniquement les premières instructions.

```
Initial : if len==0 then wait(reqPrendre);
                         listld[len]:=e; len++;
          else ids:=listld[0]; traitPrendre; e:=ids;
               ...
          jmp Initial;
```

Des problèmes de disponibilité peuvent apparaître avec ce type de ressource :
- si un processus prend une ressource `mutex` et ne la relâche pas,
- plusieurs services accédant à plusieurs ressources `mutex` peuvent donner lieu à un interblocage.

4.5.2 Ressource de type `partageable`

Un raffinement de la ressource `mutex` est la ressource `partageable` qui comporte un ensemble de k parts pouvant être allouées à différents services. Ce type de ressource se rencontre fréquemment dans la littérature [LZ97] et correspond à de nombreuses ressources réelles (par exemple, l'ensemble des connections réseau, un serveur d'exécution, *etc.*). L'interface et le comportement d'une ressource partageable sont :
- interface :
  - `reqPrendre(i)` pour demander i parts de la ressource ;
  - `traitPrendre(i)` quand la ressource accorde les i parts ;
  - `relacher(i)` pour libérer i parts.
- comportement :
  - les `reqPrendre(i)` sont accordés selon un ordre FIFO ;
  - les `traitPrendre(i)` sont effectués dès que i parts sont libres ;
  - les `relacher(i)` sont effectués sans délai.

Nous ne donnons pas ici l'automate de comportement qui est similaire au précédent. Une ressource partageable pose les mêmes problèmes de disponibilité qu'une ressource `mutex`. comme elle peut être vue comme k ressources `mutex`. A la différence des ressources `mutex`, des interblocages peuvent apparaître avec une seule ressource partageable. Les ressources partageables permettent aussi des politiques de disponibilité plus fines en imposant des quotas sur le nombre de parts alloués par un service.

D'autres types de ressources ou des spécifications plus sophistiquées peuvent se décrire dans ce cadre. Par exemple, il serait possible de prendre en compte une notion de priorité associée aux services. Le traitement des requêtes se ferait alors en fonction de l'ordre d'arrivée *et* de la priorité du service.

4.6 Les services

Les services sont décrits dans le langage des programmes du chapitre 2.

4.6.1 Syntaxe

Comme en section 2.3.1, un service $S = (l_0, E)$ est défini par un ensemble d'*instructions* $E = \{I_1, \ldots, I_n\}$ de la forme

$$I \quad ::= \quad l_1 : c \rightsquigarrow l_2 \quad | \quad l_1 : g \rightsquigarrow l_2\ ,\ l_3$$

De plus, un service se représente par une boucle infinie de traitement qui attend une requête d'un utilisateur, effectue un traitement/calcul pour cette requête, renvoie à l'utilisateur la réponse à sa requête, puis traite une nouvelle requête. Un service commence par l'instruction l_0 : `getUser`() $\rightsquigarrow l_1$ qui attend et prend une nouvelle requête d'utilisateur et se termine par l_i : `endUser`() $\rightsquigarrow l_0$ qui renvoie les résultats à l'utilisateur et retourne à l_0 pour traiter une nouvelle requête.

Par exemple, le service `S1` de la figure 4.3 s'écrit dans cette syntaxe :

$$\texttt{S1} = \left\{ \begin{array}{lcll} l_0 & : & \texttt{getUser}() & \rightsquigarrow & l_1 \\ l_1 & : & \texttt{M1.prendre}() & \rightsquigarrow & l_2 \\ l_2 & : & \texttt{M2.prendre}() & \rightsquigarrow & l_3 \\ l_3 & : & \texttt{S1.calcul}() & \rightsquigarrow & l_4 \\ l_4 & : & \texttt{M2.relacher}() & \rightsquigarrow & l_5 \\ l_5 & : & \texttt{M1.relacher}() & \rightsquigarrow & l_6 \\ l_6 & : & \texttt{endUser}() & \rightsquigarrow & l_0 \end{array} \right\}$$

Les commandes `getUser`(), `prendre`() sont bloquantes (*e.g.,* s'il n'y a de requête ou si la ressource n'est pas libre) ; la commande `S1.calcul` correspond en fait à une agrégation d'actions élémentaires qui n'effectuent aucune opération de gestion des ressources.

4.6.2 Sémantique

La sémantique d'un service $S = (l_0, E)$ est exprimée, comme en section , par une fonction rendant pour chaque état initial la trace d'exécution du programme. Une trace d'exécution est décrit par un LTS $(\Sigma_S, (l_0, s_0), \mathcal{E}_S, \longrightarrow_S)$. Nous considérons une sémantique des services qui n'est pas temporisée. La sémantique temporisée des services (comme celle proposée en Annexe C) est uniquement nécessaire pour montrer la correction du tissage.

4.7 Aspects de disponibilité

Dans le domaine de la disponibilité, on considère souvent des politiques en *temps fini* où l'on assure (par analyse statique ou vérification) que les requêtes des utilisateurs seront finalement traitées "un jour". Ce style de propriété (de type vivacité) n'est pas imposable par un moniteur, par tissage ou instrumentation de code [Sch00]. Seules des propriétés de sûreté peuvent s'imposer de cette façon. Aussi, nous considérons des politiques de disponibilité plus forte de type *temps borné*. Plus précisément, nous souhaitons assurer une réponse en un temps borné (fixé par l'aspect) aux requêtes des services. D'autres propriétés temporisées correspondant à une gestion plus fine des ressources peuvent également être spécifiées. Par exemple, pour imposer un partage plus équitable des ressources, on peut limiter la fréquence de prise de ressource par un service (*i.e.,* imposer des attentes).

Nos aspects de disponibilité ont principalement comme effet d'imposer des durées maximales ou minimales. A la différence du chapitre 2, les aspects sont définis dans un langage textuel qui est ensuite traduit automatiquement en automate. Le formalisme des automates temporisés est clairement bien adapté pour représenter ce style de propriétés temporisées.

4.7.1 Syntaxe

Notre langage est inspiré des *stateful aspects* introduits dans [DFS02] qui prennent en compte l'histoire de l'exécution. La grammaire du langage est décrite en figure 4.5. Un aspect est composé d'une collection d'équations mutuellement récursives. Une équation est de la forme

$$a_i = (C \triangleright L); a_j$$

L'aspect attend l'évènement C qui, lorsqu'il survient, déclenche l'exécution d'une liste d'instructions L et fait passer le contrôle à l'équation a_j. En général une équation peut comporter des choix comme $(C \triangleright L); a \,\Box\, (C' \triangleright L'); a'$: l'aspect attend les évènements C ou C', le premier arrivant déclenche l'exécution des instructions et équations correspondantes (L et a ou L' et a'). Pour assurer le déterminisme, nous supposons que les choix sont exclusifs[1].

La grammaire de nos aspects est décrite figure 4.5. Le filtre d'évènement F, proche des points de coupure d'AspectJ [KHH$^+$01b] ou des compositions de filtres

[1] Une autre possibilité est de considérer comme dans [DFS04] que l'opérateur sélectionne le premier choix lorsque les deux sont possibles.

A	$::=$	$\{a_1 = E_1, \ldots, a_n = E_n\}$	*; équations mutuellement récursives*
E	$::=$	$E_1 \ \Box\ E_2$	*; choix*
	$\mid$	$((F, G) \triangleright L); a_i$	*; ajoute la liste d'inserts L si l'instruction*
			; est filtrée par le filtre d'évènements F et
			; la garde temporisée G et continue avec a_i
F	$::=$	$Motif \mid F_1 \wedge F_2 \mid \neg F$	*; filtre d'évènements de base*
G	$::=$	$\{\ldots, t \odot k, \ldots\}$	*; garde temporisée* $\odot \in \{\leq, <, >, \geq\}$
L	$::=$	$\{I; \ldots; I\}$	*; liste d'inserts*
I	$::=$	$reset\ (i, k)$	*; programme l'exception i pour un*
			; déclenchement dans k unités de temps
	$\mid$	$cancel\ (i)$	*; annule l'exception i*
	$\mid$	$start\ (t)$	*; initialise le minuteur t*
	$\mid$	$wait\ (t, k)$	*; attend jusqu'à ce que t vaille k*
	$\mid$	nop	*; opération vide*

La variable k représente un entier, i une interruption et t un minuteur.

FIG. 4.5 – Syntaxe du langage d'aspects

[AWBB94], est soit un motif, soit une combinaison logique de filtres. Un motif est un terme (éventuellement avec des jokers `*`) filtrant les instructions du programmes. Par exemple, `R.prendre` filtre uniquement la prise de la ressource `R`, `*.prendre` filtre toutes les prises de ressources et `R.*` toutes les opérations sur `R`. Une garde G est une conjonction (représentée par un ensemble) de comparaisons de minuteurs à des entiers.

La liste L dénote une liste d'inserts à exécuter lorsque l'évènement correspondant a été émis. Nos aspects de disponibilité sont limités à cinq types d'inserts :

- *reset*(i, k) consiste à programmer la libération de toutes les ressources allouées et à terminer le service (*i.e.,* la boucle de traitement courante) au bout de k unités de temps. Nous faisons l'hypothèse qu'un mécanisme transactionnel assure qu'un $reset$ ramène le service dans un état initial sain sans compromettre sa sûreté. Nous modélisons cette action par une transition vers un état puits RESET. Cet état sera traduit par un relâchement de toutes les ressources allouées au service et la réinitialisation de celui-ci au début de sa boucle de traitement des requêtes ;
- *cancel*(i) annule l'interruption i ;
- *start*(t) initialise le minuteur t ;
- *wait*(t, k) modélise une attente jusqu'à ce que $t = k$. Si $t \geq k$ alors l'instruction ne fait rien (*wait*$(t, k) \equiv$ *nop*) ;

– *nop* permet de faire évoluer l'aspect sans effectuer d'action.

Tous les inserts sont insérés après l'exécution de l'événement filtré (comme les inserts `after` de AspectJ), sauf l'insert $wait(t, k)$ qui est effectué avant d'exécuter l'instruction filtrée. Par exemple, $\texttt{R.prendre} \triangleright \{wait(t, 20); start(t)\}$ peut se voir comme l'insertion d'une attente jusqu'à ce que $t = 20$ avant les actions `R.prendre` et l'insertion de l'initialisation du timer après ces actions. Nous imposons la restriction syntaxique interdisant de programmer et d'annuler une même exception ($reset(i, k)$ et $cancel(i)$) dans la même liste d'inserts. Pour simplifier l'écriture des aspects, nous omettons de préciser la garde quand il s'agit de *true* et d'utiliser la notation de liste lorsqu'il n'y a qu'un seul insert. Par exemple, $(true, \texttt{M1.prendre}) \triangleright \{reset(i_1, 25)\}$ est notée $\texttt{M1.prendre} \triangleright reset(i_1, 25)$.

Nos aspects peuvent uniquement insérer des gardes ou des instructions temporisées qui ne modifient pas l'état du service. L'effet sur la sémantique du service est uniquement d'interdire certaines exécutions : soit elles sont coupées par un *reset*, soit leur comportement temporisé est modifié par un *wait*. Ces restrictions autorisent à voir les aspects comme des propriétés temporisées et permettent de raisonner sur les programmes tissés. Comme précédemment, si le service tissé n'est pas interrompu, alors le service tissé a le même comportement que le service de base.

4.7.2 Exemples d'aspects

Nous illustrons notre langage d'aspects par plusieurs exemples classiques d'utilisation : contrôler la durée d'utilisation des ressources, contrôler la fréquence d'utilisation de ressources, contrôler la durée suivant la fréquence et enfin imposer un ordre d'acquisition des ressources.

Contrôler la durée d'utilisation des ressources On peut vouloir imposer les deux aspects de disponibilité suivants au service `S1` de la figure 4.3 :

– un aspect A_1 qui assure que la ressource `M1` est relâchée avant 25 secondes ;

– un aspect A_2 qui assure que la ressource `M2` est relâchée avant 35 secondes.

Ces deux aspects sont spécifiés dans la syntaxe précédente comme suit :

$$A_1 = \left\{ \begin{array}{ll} a_1 = & \texttt{M1.prendre} \triangleright reset(i_1, 25);\ a_2 \\ a_2 = & \texttt{M1.relacher} \triangleright cancel(i_1);\ a_1 \end{array} \right\}$$

$$A_2 = \left\{ \begin{array}{ll} a_1 = & \texttt{M2.prendre} \triangleright reset(i_2, 35);\ a_2 \\ a_2 = & \texttt{M2.relacher} \triangleright cancel(i_2);\ a_1 \end{array} \right\}$$

Dès que l'évènement `M1.prendre` (resp. `M1.prendre`) est produit, une interruption (*i.e.*, un `reset`) est programmée pour se déclencher 25 secondes (resp. 35 secondes) plus tard. Si l'évènement `M1.relacher` (resp. `M2.relacher`) se produit avant, l'interruption est annulée.

Contrôler la fréquence d'utilisation des ressources Il s'agit ici d'empêcher un service d'accaparer une ressource en la redemandant tout de suite. Cette politique s'applique à des ressources qui sont constamment demandées par plusieurs services. Elle impose de façon simple une répartition plus équitable des ressources. Pour donner une idée, prenons le cas de deux services S_1 et S_2 qui nécessitent la ressource M pour fonctionner. Le service S_1 essaye de reprendre la ressource immédiatement après l'avoir relâchée alors que S_2 la redemande après 20 secondes. On peut assurer une plus grande équité entre les deux services en imposant au service S_1 d'attendre au moins 20 secondes entre chaque prise de la ressource. L'aspect suivant impose une telle propriété :

$$\left\{ \begin{array}{lcl} a_1 & = & \texttt{M.prendre} \triangleright start(t);\ a_2 \\ a_2 & = & \texttt{M.prendre} \triangleright \{wait(t,20); start(t)\};\ a_2 \end{array} \right\}$$

Dès que l'évènement `M.prendre` est produit, un minuteur t est lancé. Avant de permettre l'évènement `M.prendre` suivant, on force une attente d'au moins 20 secondes . ($wait(t, 20)$) puis `M.prendre` sont effectués, ensuite le minuteur est remis à zéro.

Contrôler la durée d'utilisation suivant la fréquence En cas d'utilisation trop fréquente, au lieu de diminuer la fréquence, une autre possibilité est de limiter le temps d'utilisation de la ressource. Par exemple, on fixe une durée d'utilisation maximale de 10 secondes ($reset(i, 10)$) sauf si la ressource était déjà utilisée moins de 20 secondes avant ($t < 20$). Dans ce cas, on impose une durée d'utilisation maximale de 5 secondes ($reset(i, 5)$) seulement.

$$\left\{ \begin{array}{lcl} a_1 & = & \texttt{M.prendre} \triangleright reset(i,10);\ a_2 \\ a_2 & = & \texttt{M.relacher} \triangleright \{cancel(i); start(t)\};\ a_1 \\ a_3 & = & (t < 20, \texttt{M.prendre}) \triangleright reset(i,5);\ a_2 \\ & \Box & (t \geq 20, \texttt{M.prendre}) \triangleright reset(i,10);\ a_2 \end{array} \right\}$$

Imposer un ordre d'acquisition des ressources Des aspects spécifiant des propriétés non temporisées peuvent également être décrits dans le langage. Par exemple, on peut vouloir imposer un ordre d'acquisition des ressources pour éviter les interblocages. Par exemple, l'aspect suivant interdit l'acquisition de la ressource `M1` si le service utilise déjà la ressource `M2`. Dans ce cas, le service est coupé immédiatement en utilisant $reset(i, 0)$. Il est possible d'utiliser `M1` et `M2` en les acquérant dans cet ordre ou de relâcher `M2` avant d'acquérir `M1`.

$$\left\{ \begin{array}{lcl} a_1 & = & \texttt{M2.prendre} \triangleright \{\};\ a_2 \\ a_2 & = & \texttt{M1.prendre} \triangleright reset(i,0);\ a_3 \\ & \Box & \texttt{M2.relacher} \triangleright \{\};\ a_1 \end{array} \right\}$$

Bien d'autres politiques de disponibilité peuvent se décrire dans le langage. Par exemple, il serait possible d'associer des priorités aux services pour réguler l'accès aux ressources. Les priorités pourraient évoluer suivant le respect des contraintes ou, plus généralement, le comportement temporisé des services.

4.8 Tissage

Le tissage d'un aspect repose sur l'opération de produit d'automates temporisés. Un service est représenté par un automate temporisé surapproximant les traces d'exécution du code source. La sémantique des aspects est donnée également sous forme d'automate temporisé. L'automate représentant l'aspect décrit l'ensemble des traces temporisées permises. Le produit de ces deux automates fait l'intersection des traces d'exécutions et impose les contraintes spécifiées par l'aspect. En pratique, cela revient à couper certaines traces (interruptions et retour vers l'état initial) ou en allonger certaines (attentes dans certains états). L'attente dans un état peut aussi être vue comme la sélection des traces qui restent suffisamment longtemps dans l'état, ou comme l'interdiction des traces qui ne restent pas suffisamment longtemps dans l'état.

Dans un premier temps, nous décrivons comment les services sont abstraits en automates temporisés. Cette abstraction est le graphe de flot de contrôle du programme sans aucune contrainte temporisée. Dans un second temps, nous donnons la sémantique du langage d'aspect précédent en termes d'automates temporisés. L'étape suivante combine l'automate représentant le service et l'automate de l'aspect en un unique automate temporisé. Cette étape se résume à effectuer l'opération classique de produit. L'automate résultant décrit le comportement du service respectant la propriété spécifiée par l'aspect. La quatrième étape consiste à optimiser cet automate en prenant en compte les durées d'exécution des instructions du service. Cette étape repose sur une fonction de coût encadrant le temps d'exécution de chaque instruction par un intervalle $[min, max]$ qui peut être vu comme une contrainte temporisée interdisant les exécutions où l'instruction prendrait moins de *min* ou plus de *max*. Ces contraintes sont intégrées une nouvelle fois par produit. Elles permettent d'optimiser le tissage (*e.g.*, enlever des attentes ou des minuteurs inutiles).

4.8.1 Abstraction des services en automates temporisés

Nous utilisons une abstraction surapproximant toutes les traces d'exécution du service et qui ne prend pas en compte les durées d'exécution des instructions (*i.e.*, les instructions prennent entre $[0, +\infty[$ unités de temps). Ceci correspond à la plus grande surapproximation du point de vue temporisée. Un service est représenté par un automate temporisé qui peut se voir comme le graphe de flot de contrôle du service.

Nous décrivons cette abstraction à l'aide de la relation $\mathbf{next}_S(l_1, a, l_2)$ qui indique qu'une instruction du service $S = (l_0, E)$ permet de passer de l'étiquette

l_1 à l_2 en effectuant l'action a. Cette relation est définie par :

$$\mathbf{next}_S(l_1, a, l_2) \quad ssi \quad l_1 : a \rightsquigarrow l_2 \in E \ \vee \ l_1 : a \rightsquigarrow l_2 \ , \ l \in E \ \vee \ l_1 : \neg a \rightsquigarrow l \ , \ l_2 \in E$$

Il est possible de passer d'une étiquette à une autre par une action, une garde évaluée à vraie, ou une garde évaluée à faux. Cette relation est une surapproximation du flot de contrôle puisqu'elle ne prend pas en compte l'évaluation des tests.

Le service $S = (l_0, E)$ est *abstrait* par l'automate temporisé

$$S^\sharp = (\Sigma_{S^\sharp}, l_0, \emptyset, \mathcal{E}_{S^\sharp}, \longrightarrow_{S^\sharp}, I_{S^\sharp}) \qquad \text{où}$$

- $\Sigma_{S^\sharp}$, l'ensemble des états abstraits, est constitué de l'ensemble des étiquettes du programmes et d'un ensemble d'états intermédiaires représentant le passage d'une étiquette à une autre. Formellement :

$$\begin{aligned} \Sigma_{S^\sharp} = \ & \{l \mid \mathbf{next}_S(l, a, l')\} \\ & \cup \{l_a \mid \mathbf{next}_S(l, a, l')\} \end{aligned}$$

- l'état initial abstrait est l'étiquette initiale l_0 ;
- l'ensemble des horloges est vide ;
- l'ensemble des actions est celui de S dans lequel chaque action a est découpée en deux "instants" *deb a* et *fin a* et l'instruction spéciale *wait* :

$$\mathcal{E}_{S^\sharp} = \{deb\ a, fin\ a \mid a \in \mathcal{E}_S\} \cup \{wait\}$$

  Cette découpe, qui est spécifique à ce chapitre, nous permettra de prendre en compte la durée d'exécution de a. L'instruction *wait* ne change pas la mémoire et nous sert à modéliser l'attente dans un état ;
- la relation de transition $\longrightarrow_{S^\sharp}$ est définie par :

$$l \xrightarrow{.,deb\ a,.}_{S^\sharp} l_a \ \wedge \ l_a \xrightarrow{.,\text{fin}\ a,.}_{S^\sharp} l') \quad ssi \quad \mathbf{next}_S(l, a, l')$$

  Chaque action a passant d'une étiquette à une autre est représentée à l'aide de l'état intermédiaire l_a et de deux transitions correspondant aux deux instants *deb a* et fin a de l'action sans contrainte temporisée ; Pour modéliser le temps pouvant s'écouler entre deux actions, nous ajoutons pour chaque état l une transition *wait* :

$$\forall l \in \Sigma_{S^\sharp}.\ (l, ., wait, ., l) \ \in \longrightarrow_{S^\sharp}$$

- la fonction $I_{S^\sharp}$ ne place aucune contrainte temporisée sur les états, c'est à dire $\forall l \in \Sigma_{S^\sharp}.I_{S^\sharp}(l) = \emptyset$.

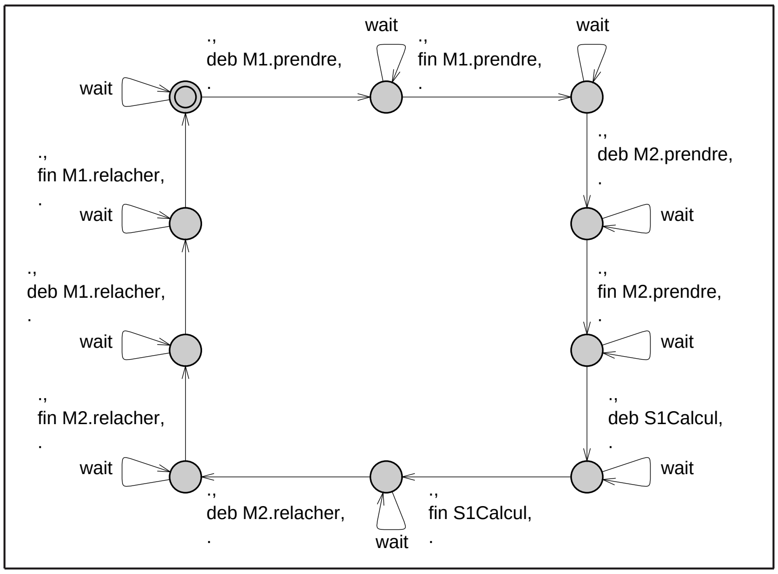


FIG. 4.6 – Abstraction du service `S1`

L'absence de contrainte temporisée fait que l'automate représente toutes les durées d'exécution possibles pour chaque action. La figure 4.3 illustre cette étape en présentant l'abstraction du service `S1` par un automate.

Cette abstraction respecte des propriétés similaires à la sûreté et à la fidélité de l'abstraction. Il convient néanmoins d'adapter ces deux propriétés au fait que l'exécution d'une action est abstraite par deux transitions de début et de fin de l'action.

4.8.2 Sémantique des aspects sous forme d'automates temporisés

L'aspect spécifie une propriété temporisée et les traces de cet automate sont les traces temporisées autorisées par l'aspect. La sémantique des aspects est donnée par des automates temporisés $A = (N_a, l_{a0}, H_a, \mathcal{E}_a, \longrightarrow_{asp}, I_e)$ où :

- l'ensemble d'états N_a est composé d'un état puits RESET et de couples (q, e) où q représente l'état de l'aspect et e l'environnement d'interruption courant ;
- $l_{a0} = (a_0, \{\})$ est l'état initial ;
- H_a est l'ensemble des horloges (interruptions et minuteurs) utilisées dans l'aspect ;
- $\mathcal{E}_a$ définit les mêmes commandes que le service abstrait sur lequel A s'applique ;
- I_a la fonction associe à chaque état (q, e) un invariant imposant qu'aucune interruption valide (*i.e.*, définie dans e) n'ait lieu. Cette fonction se définit ainsi

$$I_a(q, e) = \{i \leq e(i) \;\mid\; \forall i.e(i) \neq \perp\}$$

– la relation de transition $\longrightarrow_{asp}$ étiquetée entre les états.

Nous décrivons maintenant comment un aspect est traduit en un automate temporisé. Les états de l'automate temporisé sont définis comme des couples (q, e) où q dénote l'équation courante de l'aspect et e représente l'environnement d'interruption (les conditions imposées par les interruptions). Lors de cette étape, nous utilisons la transition $(q, e) \xrightarrow{sinon}_{asp} (q, e)$ qui précise que si aucune des transitions sortantes de l'état (q, e) ne s'applique alors l'aspect reste dans le même état. Cette notation est du sucre syntaxique qui peut être remplacé dans un deuxième temps par une collection de transitions (dénotant le complémentaire des transitions sortantes). La construction de l'automate est effectuée récursivement à partir des règles suivantes sur la syntaxe de l'aspect :

– $[a_0 = E_0] \quad = \quad \{(a_0, \{\}) \xrightarrow{sinon}_{asp} (a_0, \{\})\} \quad \cup \quad [E_0]^{(a_0, \{\})}$

  On produit l'automate correspondant à E_0 avec comme état initial $(a_0, \{\})$. Aucune interruption n'est activée, et comme tout état, l'état initial a une transition en boucle *sinon*.

– $[E_1 \Box E_2]^{(q,e)} \quad = \quad [E_1]^{(q,e)} \cup [E_2]^{(q,e)}$

  Les transitions correspondant à un choix exclusif sont juste l'union des transitions pour les deux choix.

– $$\begin{array}{rcl} [(F, G) \rhd L; a_i]^{(q,e)} & = & [(F, G) \rhd L]^{(q,e)}_{(a_i, e')} \quad \cup \quad [E_i]^{(a_i, e')} \quad (\text{avec } \{a_i = E_i\} \in A) \\ & \cup & \{(a_i, e') \xrightarrow{sinon}_{asp} (a_i, e')\} \quad \cup \quad interrupt(a_i, e') \end{array}$$

  Le traitement d'une règle $(F, G) \rhd L$ calcule un nouvel environnement e' d'interruption (règle suivante) et produit des transitions. L'automate correspondant à la suite de l'aspect est produit à partir de ce nouvel état. Pour tout nouvel état, on génère la transition *sinon* et les transitions modélisant les interruptions contenues dans l'environnement courant (fonction *interrupt*).

– $$[(F, G) \rhd L]^{(q_1, e_1)}_{(q_2, e_2)} =$$
$$\left\{ \begin{array}{l|l} (q_1, e_1) \xrightarrow{g, deb(a), .}_{asp} (q', e_1) & \forall a.match(a, F).q' \text{état neuf} \\ (q', e_1) \xrightarrow{conj(e), \text{fin}(a), r}_{asp} (q_2, e_2) & \wedge\ ins(e_1, L) = (g_i, r, e_2) \\ interrupt(q', e_1) & \wedge\ \ (g = conj(e_1) \cup G \cup g_i) \end{array} \right\}$$
$$\cup \{(q_1, e_1) \xrightarrow{t<k \wedge G \wedge conj(e), wait, .}_{asp} (q_1, e_1) \mid wait(i, k) \in L\}$$

  Pour chaque action a filtrée par F, deux transitions (*deb a* et *fin a*) sont produites à l'aide d'un nouvel état intermédiaire (q', e_1). Les transitions modélisant les interruptions possibles sont ajoutées sur cet état. La fonction *ins* analyse la liste d'inserts L pour calculer les gardes et les raz des transitions ainsi que le nouvel environnement d'interruptions. Si L contient des *wait*, les transitions d'attentes correspondantes sont ajoutées sur l'état de départ.

Les fonctions intermédiaires sont définies ainsi :

- La fonction *interrupt* prend un état (q, e) et retourne l'ensemble des transitions à ajouter pour modéliser les interruptions possibles dans cet état.

$$interrupt((q, e)) = \{(q, e) \xrightarrow{i \geq e(i),.,.}_{asp} \text{Reset} \mid e(i) \neq \perp\}$$

  En d'autres termes, il y a une interruption à chaque fois qu'un minuteur i dépasse sa valeur de déclenchement mémorisée dans l'environnement e.

- La fonction $match(a, F)$ retourne vraie si l'action a est filtrée par F.

- La fonction *ins* prend un environnement d'interruptions, une liste d'inserts et calcule la garde correspondant aux *wait* des inserts, les remises à zéro correspondant aux *reset* et *start* des inserts et, enfin, le nouvel environnement d'interruptions calculé en prenant en compte les *reset* et *cancel* des inserts.

$$ins(e, L) = (\{t \geq k \mid wait(t, k) \in L\}, \quad \{z | reset(z, k) \in L \lor start(z) \in L\}, \quad e')$$

$$\text{avec} \quad \begin{cases} e'(i) = \perp & \text{si } cancel(i) \in L \\ e'(i) = k & \text{si } reset(i, k) \in L \\ e'(i) = e(i) & \text{sinon} \end{cases}$$

- La fonction *conj* prend un environnement et retourne la garde correspondant au cas où il n'y a pas d'interruptions à déclencher :

$$conj(e) = \{i < e(i) \mid e(i) \neq \perp\}$$

L'automate est produit en dépliant les définitions récursives des aspects. Le processus termine car il y a un nombre fini de définitions récursives $a_i = \ldots$ et d'environnements possibles (les durées initiales des exceptions sont fixés dans l'aspect).

La figure 4.7 montre les automates obtenus pour les aspects A_1 et A_2 définis précédemment. Intuitivement, le tissage consistera à initialiser une horloge quand le service prend la ressource puis à interrompre le service (en l'emmenant dans l'état Reset) lorsque le minuteur dépasse le quota de temps autorisé (*i.e.,* 25 sec. ou 35 sec.). L'état puits Reset ajouté par les aspects de disponibilité, sera interprété à la concrétisation en ajoutant des transitions relâchant les ressources suivies par une transition qui revient au début de la boucle de traitement des requêtes.

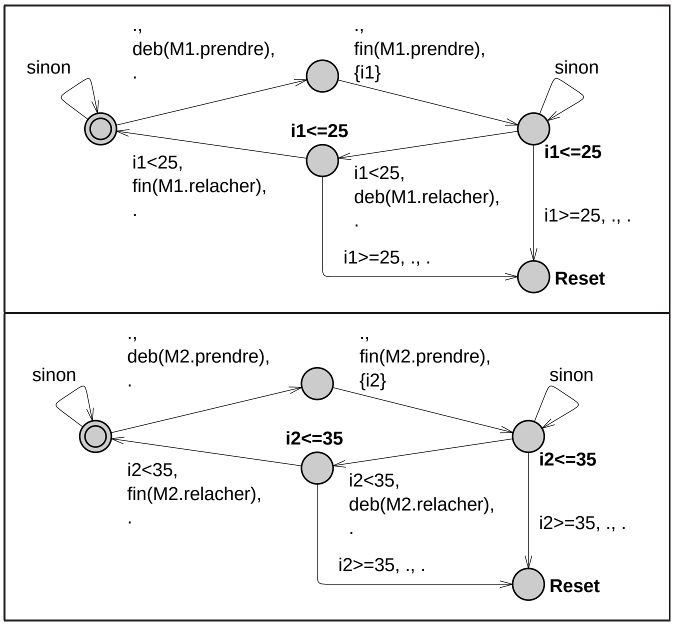


FIG. 4.7 – Automates temporisés représentant les aspects A_1 (en haut) et A_2 (en bas)

4.8.3 Intégration d'un aspect à un service

Le tissage à proprement parler est juste le produit (comme décrit en section 4.4.3) des automates temporisés représentant le service et l'aspect. L'impact sémantique de ce tissage est de restreindre le comportement du service en sélectionnant les traces autorisées par l'aspect L'automate de l'aspect décrit les traces autorisées du service et est annoté (par des initialisations d'horloges, des gardes temporisées, ou des invariants d'états) pour gérer les inserts et les gardes temporisées. Le produit d'automates (*i.e.,* le tissage) effectue l'intersection des traces des automates du service et de l'aspect.

La figure 4.8 montre le produit de l'abstraction du service `S1` avec les aspects A_1 et A_2. Dans l'automate résultant (*i.e.,* le service tissé), une interruption est programmée après `M1.prendre` et une autre après `M2.prendre`. Si un `M1.relacher` (resp. `M2.relacher`) n'est pas effectué dans les 25 secondes (resp. 35 secondes), l'automate passe dans l'état *Reset*.

4.8.4 Prise en compte des durées d'exécution et optimisations

La prise en compte de durées d'exécution repose sur une fonction de coût f_{cout} qui prend une instruction (ou un bloc d'instructions) I et retourne un intervalle de temps $[\text{BCET}(I), \text{WCET}(I)]$ où $\text{BCET}(I)$ (resp. $\text{WCET}(I)$) représente la

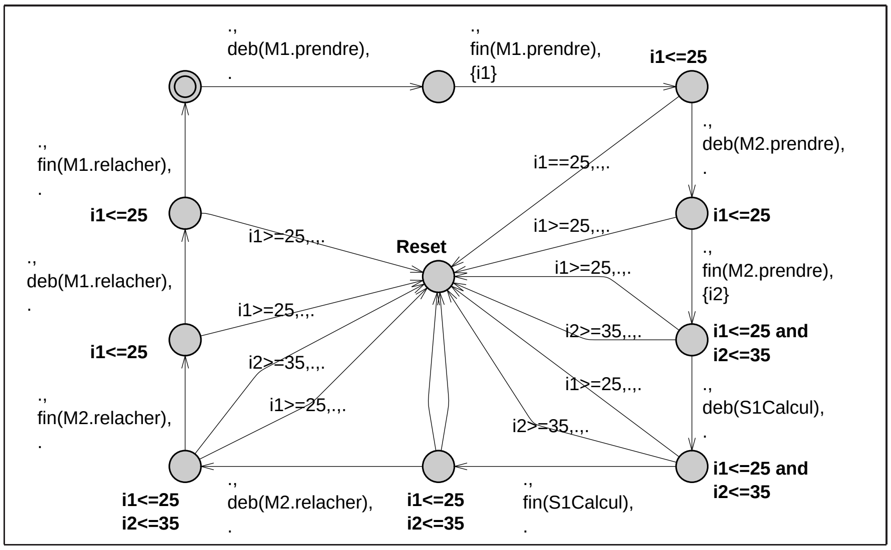


FIG. 4.8 – Produit du service S1 avec les aspects A_1 et A_2

borne inférieure (resp. supérieure) du temps d'exécution de I. Notons que l'on peut toujours concevoir une telle fonction de coût puisque l'approximation triviale $f_{cout}(I) = [0, +\infty]$ est toujours valide. Cependant une fonction de coût plus précise (voir par exemple [PK89, LMR05]) peut éviter au tisseur d'insérer des tests. Par exemple, si f_{cout} permet de déterminer statiquement qu'un service relâche toujours une ressource dans les temps requis par un aspect de disponibilité, aucun test ou instrumentation du service ne sera nécessaire.

L'évaluation du temps d'exécution est un domaine de recherche à part entière que nous ne décrivons pas plus. Par la suite, nous faisons l'hypothèse que nous possédons une telle fonction de coût et qu'elle rend les résultats suivants pour les actions du service S1 :

$$\begin{array}{lcllcl} f_{cout}(\texttt{S1calcul}) & = & [2,10] & & & \\ f_{cout}(\texttt{M1.Prendre}()) & = & [0,+\infty] & f_{cout}(\texttt{M2.Prendre}()) & = & [0,+\infty] \\ f_{cout}(\texttt{M1.relacher}()) & = & [0,0] & f_{cout}(\texttt{M2.relacher}()) & = & [0,0] \end{array}$$

La prise en compte de la fonction de coût est effectuée par un produit d'automates temporisés avec l'automate $C = (N_c, c_0, \{k\}, \mathcal{E}_{S\sharp}, \longrightarrow_c, I_c)$ où :

– pour toute action a telle que $f_{cout}(a) = [min, max]$ on a les transitions

$$c_0 \xrightarrow{.,deb(a),\{k\}}_c q \quad \text{et} \quad q \xrightarrow{k \geq min, fin(a),.}_c c_0 \quad \text{avec } q \text{ un nouvel état}$$

– l'invariant d'état spécifie qu'on ne peut y rester que le maximum du coût de l'action qui nous y a conduit. C'est à dire :

$$I_c(q) = \{k \leq max \mid q \xrightarrow{k \geq min, fin(a),.}_c c_0\}$$

L'horloge k est remise à zéro au début de l'action a. On reste dans l'état intermédiaire au plus tard jusqu'à ce que $k = max$ et on en sort à la fin de l'action au plus tôt quand $k \geq min$. La figure **??** décrit l'automate C pour le service `S1`.

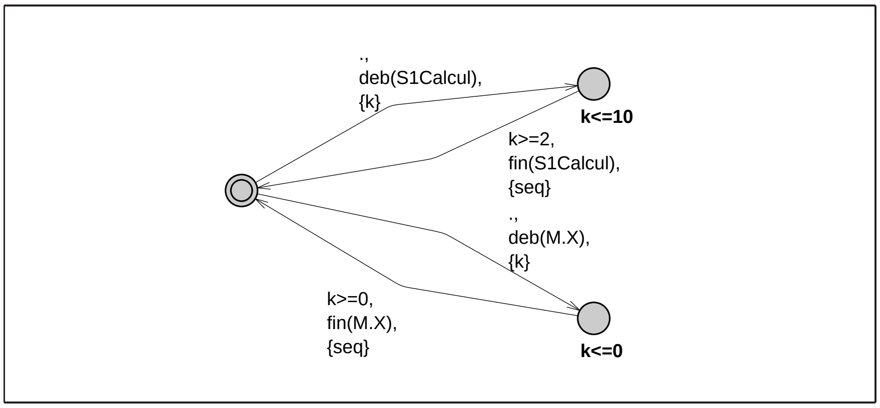


FIG. 4.9 – Automate modélisant la fonction de coût du service `S1`

Une autre information à prendre en compte est que le séquencement entre les actions est instantané (les ; sont exécutés immédiatement). Ceci est décrit par un produit avec l'automate temporisé à deux états $E = (\{e_0, e_1\}, e_0, \{seq\}, \mathcal{E}_{S\sharp}, \longrightarrow_e, I_e)$ où :

- chaque début d'action fait passer à l'état e_1 et chaque fin d'action, fait passer à l'état e_0 en remettant à zéro l'horloge dédiée seq. Intuitivement, l'état e_0 représente les points de jonction des actions (qui ne prennent pas de temps) alors que l'état e_1 représente le corps de l'action (où du temps peut s'écouler) ;

$$\longrightarrow_e = \left\{ \begin{array}{l} e_0 \xrightarrow{.,deb(a),.}_e e_1 \\ e_1 \xrightarrow{.,fin(a),\{seq\}}_e e_0 \end{array} \;\middle|\; deb(a) \in \mathcal{E}_{S\sharp} \wedge fin(a) \in \mathcal{E}_{S\sharp} \right\}.$$

- l'invariant de l'état e_0 (représentant le séquencement des actions) assure que le temps ne peut s'écouler dans cet état. Aucune contrainte n'est imposée sur l'état e_1.

$$I_e(e_0) = \{seq \leq 0\} \qquad I_e(e_1) = \emptyset$$

La figure 4.8.4 montre l'automate E pour le service `S1`.

L'automate temporisé obtenu après le produit avec les deux automates précédents est plus précis. Le produit a restreint l'ensemble des traces possibles et a intégré les informations de durées d'exécution disponibles. L'automate peut maintenant être analysé pour enlever les gardes, horloges et invariants d'états inutiles ainsi que les états non atteignables. Cette optimisation permet de diminuer le surcoût in-

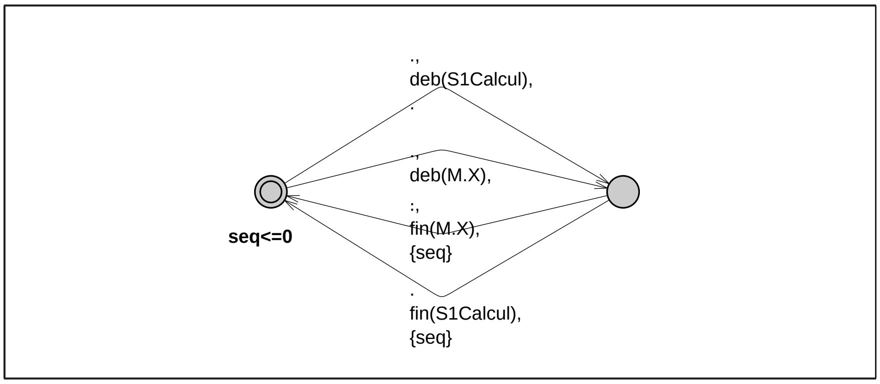


Fig. 4.10 – Automate modélisant le séquencement instantané entre les actions du service `S1`

troduit par l'aspect. Elle est faite automatiquement dans des outils tels qu'UPPAAL.

La figure 4.11 montre l'automate obtenu pour le service tissé `S1` de la figure 4.8 après produit avec les automates C et E (des figures et) et optimisation. L'aspect A_2 interdit au service de retenir la ressource `M2` plus de 35 sec. Le tis-

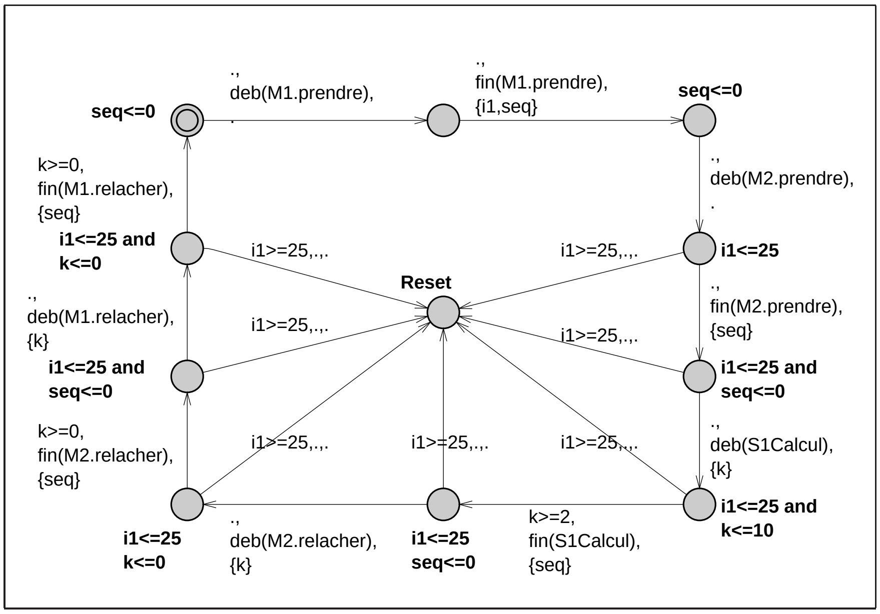


Fig. 4.11 – Automate temporisé du service `S1` après tissage et optimisation

sage de cet aspect n'a aucun impact sur le code car l'automate du service indique clairement que `S1calcul` (et donc l'utilisation de `M2`) dure au plus 10 sec. Cette information, donnée initialement par la fonction de coût et intégrée dans l'automate représentant le service, permet d'optimiser le tissage et d'éviter d'introduire un minuteur inutile. Le tissage de l'aspect A_1 ne peut pas quand à lui être optimisé, car le temps d'accès pour accéder à la ressource `M2` dépend des autres services et pourrait dépasser 15 secondes.

4.8.5 Vérification

L'automate obtenu précédemment est une représentation formelle du service tissé. A cette étape, on peut vouloir vérifier que les services tissés garantissent des propriétés de disponibilité. En effet, les aspects sont un ensemble de contraintes temporisées censé impliquer une propriété de disponibilité de plus haut niveau. Ce type de propriété peut être vérifié par model-checking sur le système obtenu après tissage. Cette étape permet également de vérifier que les aspects ne sont pas contradictoires. En effet, certains aspects peuvent entrer en conflit (par ex. un aspect qui prévient les interblocages en augmentant le temps d'allocation d'une ressource peut rentrer en conflit avec un aspect temporisé limitant ce même temps d'allocation). Dans ce cas, on aurait des traces toujours coupées. Comme dans le chapitre 2, le tissage est associatif et commutatif, et il est possible de construire l'automate représentant les différents aspects par produit des automates temporisés (l'intersection des traces). Cependant, la gestion des délais peut être conflictuelle entre les différents aspects de disponibilité. Il convient donc de vérifier comme dans [CS99] que la politique de disponibilité est cohérente (*i.e.*, elle ne dit pas une chose et son contraire).

Nous avons utilisé UPPAAL qui permet de représenter les services, d'exécuter (et de mettre au point) les modélisations, pour vérifier des propriétés exprimées en LTL. Nous avons représenté et analysé l'exemple pris dans ce chapitre avec UPPAAL. Nous avons vérifié que le système tissé respectait les propriétés suivantes :

- *le système est bien temporisé et n'arrive pas à des situations d'interblocage.* Nous n'avons pas explicitement prévenu les interblocages qui peuvent survenir entre `S1` et `S2`. Néanmoins, ceux-ci sont traités par l'aspect interrompant `S2` au bout de 35 secondes.
- *le service* `S1` *accomplit un tour de boucle en moins de* 45 *secondes.* Cette propriété s'exprime en introduisant un nouveau compteur `dispo` mis à zéro au début de la boucle de traitement et en vérifiant qu'il n'est, dans aucun état, supérieur à 45 secondes. Cette propriété a été garantie par tissage. En effet, après tissage le service `S2` relâche la ressource `M2` après au plus 35 sec. et comme `S1calcul` prend au plus 10 secondes, `S1` aura terminé en au plus 45 secondes. Ceci signifie aussi que le service `S1` a toujours accès aux ressources et qu'il ne peut pas y avoir de dénis d'accès aux ressources dans le système (contrairement à avant le tissage).

L'analyse de ces propriétés est très rapide (moins d'une seconde). Nous supposons qu'UPPAAL pourrait traiter des services tissés de grande taille car il a déjà servi pour analyser des protocoles complexes.

4.9 Concrétisation

La génération du code à partir d'un automate à états finis est simple (*c.f.* la section 2.7). La nouveauté apportée par les automates temporisés réside dans les instructions temporisées (initialisation d'horloge, invariants d'états et gardes temporisées).

Afin de prendre en compte les actions introduites dans le tissage, nous étendons notre langage source avec des commandes temporisées et des comparaisons de minuteurs. Le langage est étendu avec les commandes suivantes :

$$\mathit{start}(t) \;\mid\; \mathit{wait}(t,k) \;\mid\; \mathit{reset}(i,k) \;\mid\; \mathit{cancel}(i)$$

où t dénote un identificateur de minuteur, i un identificateur d'interruption et k un entier. Les tests sont étendus par des comparaisons de minuteurs :

$$t \odot k \text{ avec } \odot \in \{<, >, \leq, \ldots\}$$

La commande $\mathit{start}(t)$ déclenche (initialise à zéro) un minuteur t qui pourra être comparé à des entiers par des tests. Les minuteurs sont également utilisés pour retarder l'exécution à l'aide de la commande $\mathit{wait}(t,k)$ qui attend tant que $t < k$. La commande $\mathit{reset}(i,k)$ programme une interruption i pour se déclencher k unités de temps plus tard. L'instruction $\mathit{cancel}(i)$ déprogramme l'interruption i. Une sémantique précise de ces commandes est donnée en annexe C.

Nous décrivons informellement comment un automate temporisé est traduit dans ce langage enrichi. Les informations temporelles introduites par les automates C et E sont ignorées car elles correspondent à des propriétés non fonctionnelles et non à du code de programmes. Nous supposons que les informations temporelles sont étiquetées par une information qui précise si elle correspond à l'automate C, à l'automate E, à un timer, ou à une interruption. La concrétisation suit les règles suivantes :

- les couples de transitions

  $$(q_1 \xrightarrow{.,deb(c),.} q',\quad q' \xrightarrow{.,fin(c),.} q_2)$$

  correspondent à une commande c et sont traduites par l'instruction

  $$l_{q1} : A \rightsquigarrow l_{q2}$$

- les quadruplets de transitions

  $$(q_1 \xrightarrow{.,deb(g),.} q',\quad q' \xrightarrow{.,fin(g),.} q_2,\quad q_1 \xrightarrow{.,deb(\neg g),.} q'',\quad q'' \xrightarrow{.,fin(\neg g),.} q_3)$$

  correspondent à un test g et sont traduits par l'instruction $l_{q1} : g \rightsquigarrow l_{q2}\ ,\ l_{q3}$;

- une boucle $q \xrightarrow{t<k, wait, .} q$ est traduite par l'insertion d'une commande $wait(t,k)$ avant le point de programme correspondant (*i.e.*, l_q) ;
- la remise à zéro d'un minuteur t sur la transition $q \xrightarrow{g, fin(a), \{t\}} q'$ est traduite par l'insertion d'une commande $start(t)$ après le point de programme correspondant à q' ($l_{q'}$) ;
- la gestion des interruptions est effectuée en insérant la commande $reset(i,k)$ à l'initialisation de l'interruption i (*i.e.*, i dans un raz) et en insérant la commande $cancel(i)$ au premier état q où il n'y a plus de transition $q \xrightarrow{i \geq k, \ldots} $ Reset.

La figure 4.12 montre le code source du service `S1` obtenu après concrétisation de l'automate de la figure 4.11. Après l'instruction `M1.prendre()`, une nouvelle

$$\texttt{S1} = \left\{ \begin{array}{lcll} l_0 & : & \texttt{getUser}() & \rightsquigarrow & l_1 \\ l_1 & : & \texttt{M1.prendre}() & \rightsquigarrow & l'_1 \\ l'_1 & : & reset(i, 25) & \rightsquigarrow & l_2 \\ l_2 & : & \texttt{M2.prendre}() & \rightsquigarrow & l_3 \\ l_3 & : & \texttt{S1.calcul}() & \rightsquigarrow & l_4 \\ l_4 & : & \texttt{M2.relacher}() & \rightsquigarrow & l_5 \\ l_5 & : & \texttt{M1.relacher}() & \rightsquigarrow & l'_6 \\ l'_6 & : & cancel(i) & \rightsquigarrow & l_6 \\ l_6 & : & \texttt{endUser}() & \rightsquigarrow & l_0 \end{array} \right\}$$

FIG. 4.12 – Code du service `S1` après tissage

interruption `i` est initialisée pour s'exécuter après 25 secondes. Dans le cas où le service prend moins de 25 secondes pour finir son traitement, la ressource `M1` est relâchée (`M1.relacher()`) et l'interruption est déprogrammée ($cancel(i)$).

4.10 Conclusion

Nous avons présenté un cadre formel permettant d'imposer des propriétés de disponibilité sur un système de services partageant des ressources. Cette approche est une généralisation de l'approche proposé dans le chapitre 2 à des propriétés temporisées. Un aspect peut couper la trace d'exécution si la propriété n'est plus respectée mais aussi forcer des attentes pour éviter de violer la propriété. En considérant que le programme de base peut avoir tous les comportements temporels possibles, l'insertion de délais peut être vue comme une sélection de trace. Notre contribution tient à la fois dans la proposition d'un langage d'aspect dédié à la prévention des dénis de service et à l'approche qui présente les aspects comme des propriétés temporisées et le tissage comme un produit d'automates. Ce travail a été accompli dans le cadre de l'ACI Dispo[1] qui nous a permis d'avoir des échanges réguliers avec des chercheurs spécialistes des composants et de la disponibilité. Les différentes réunions ont permis de valider notre modèle et notre approche pour la disponibilité. Nous n'avons pas mis en œuvre cette technique mais nous pensons néanmoins qu'elle est réaliste. Les services devraient rester de taille raisonnable puisque le code sans rapport avec la gestion des ressources n'a pas à être pris en compte. Par ailleurs, on peut maîtriser les coûts des analyses (flot de contrôle, temps d'exécution) en adaptant leurs approximations (*i.e.,* leurs précisions). Enfin, si le tissage par un produit d'automates naïf peut dans certains cas entraîner une explosion de code, il est facile d'éviter ce problème en instrumentant le code au lieu de le dupliquer comme présenté dans la partie 2.9.2 du chapitre 2.

Yu et Gligor [YG90] ont proposé une méthode pour vérifier qu'un allocateur de ressources reste disponible. Notre cadre est inspiré de leur approche, et en particulier, nos services peuvent se voir comme leurs utilisateurs. Les aspects modélisent ainsi les contrats utilisateurs et peuvent être tissés sur les services. De la même manière, nous utilisons les contrats utilisateurs et la spécification du système pour déduire la propriété de disponibilité du système. Millen [Mil94] propose un modèle de moniteur pour la disponibilité qui impose une politique globale de disponibilité en temps borné Notre approche montre aussi que des politiques locales peuvent aussi être utilisés pour assurer la disponibilité. Les politiques locales de disponibilité proposées ont l'avantage d'être facilement compréhensibles et tissables sur les services. De plus, l'utilisation de la programmation par aspects permet une instrumentation automatique des programmes par tissage qui est plus efficace que qu'un moniteur. Cuppens et Saurel [CS99] ont proposé un cadre logique pour exprimer et vérifier des politiques de disponibilité. Le modèle est précis et expressif mais il n'a pas été conçu dans le but de faire respecter ces politiques mais de montrer la cohérence de la politique ou de prouver des propriétés de disponibilité. J-Seal2 [BHV01] propose un mécanisme simple et global pour assurer la disponibilité du processeur et de la mémoire. Ils décrivent l'implémentation en termes d'instrumentation des services mais cette technique n'est pas générique et ne s'applique pas à d'autres types de ressources (par exemple, les ressources en accès exclusif).

Les preuves de la concrétisation et la preuve de correction associée restent à faire. Comme dans le chapitre 2, il faut montrer que la concrétisation est sûre et fidèle. Cela nécessite d'utiliser une sémantique temporisée des services (comme celle proposé en annexe C). Nous pensons néanmoins que les preuves de la concrétisation sont similaires avec celles du chapitre 2 modulo la gestion des traits temporels. La preuve de correction de l'approche nécessite de mettre en relation les traces temporisées du service respectant l'aspect et les traces temporisées du service tissés. Cette preuve peut se voir comme la généralisation de la preuve du chapitre 2 à des propriétés temporisées et devrait respecter la même structure.

[1] http://www.irisa.fr/lande/jensen/dispo.html

Un autre sujet, que nous n'avons fait qu'effleurer ici, est celui de la prévention des interblocages. Limiter la durée de possession des ressources ou imposer un ordre d'acquisition (*c.f.* section 4.7.2) permet de prévenir les interblocages. Pourtant, ces techniques ne sont pas idéales : le système peut se retrouver bloqué à attendre qu'une limite de temps soit atteinte et imposer un ordre d'acquisition peut résulter dans l'interruption systématique de services qui ne pourront plus remplir leur tâche. Une autre solution serait de transformer les services (*e.g.*, avancer l'allocation de certaines ressources ou utiliser des variables partagées) afin qu'ils respectent l'ordre d'acquisition spécifié par l'aspect. Nous n'avons pas encore formalisé cette transformation mais il semble clair que des techniques d'analyse statique seraient utiles pour satisfaire l'ordre d'acquisition tout en retenant le moins longtemps possible les ressources. La dernière solution envisagée serait de tisser une politique globale de prévention des interblocages en utilisant des variables partagées afin de forcer des choix dans l'ordonnancement des services. Dans un nouveau langage d'aspects, l'aspect pourrait décrire que la prise de la ressource mutex `R1` n'est pas possible si la ressource `R2` est déjà possédée par un autre service. Le tissage de cet aspect utiliserait une variable partagée v_2 représentant l'état de la ressource `R2` et insérerait une attente sur le prédicat $v_2 ==$ libre avant les prises de la ressource `R1`.

Conclusion

Nous avons proposé une approche originale pour la programmation par aspects qui restreint l'expressivité des aspects pour faciliter les raisonnements sur les programmes. Les aspects représentent des propriétés qui spécifient l'ensemble des traces d'exécution autorisées. L'effet du tissage est de sélectionner les traces autorisées ou de couper les traces interdites. Cette approche repose sur un formalisme simple et intuitif, les automates (à états finis ou temporisés). Les programmes et les aspects sont représentés comme des automates et le tissage comme un produit d'automates.

Le chapitre 2 présente les principales idées et techniques de l'approche dans un cadre simple de programmes déterministes et séquentiels. Les aspects spécifient des propriétés de sûreté et le tissage reviens à interrompre le programme juste avant de que la propriété soit violée. Le chapitre 3 étend le cadre à des réseaux de composants concurrents. Les aspects expriment alors le choix d'un ordonnancement qui permet de fusionner automatiquement un réseau en un programme séquentiel équivalent. Ces aspects permettent de réconcilier la performance et la modularité en supprimant une bonne part du surcoût induit par les composants. Comme une exécution parallèle se représente par un arbre des exécutions possibles, la sélection de trace imposées par l'aspect peut se voir comme le choix d'un ordonnancement. Le chapitre 4 étend l'approche à des programmes et des propriétés temporisés. Les aspects peuvent être vus comme la spécification d'une politique de disponibilité prévenant les famines et les interblocages. Le tissage revient à insérer des délais ou arrêter le programme. Le tissage est décrit comme un produit d'automates temporisés (*i.e.,* une intersection des traces d'exécution temporisées) qui restreint le comportement du programme aux traces autorisées par l'aspect.

La contribution de ce travail tient dans l'originalité de l'approche, le travail de formalisation associé et les deux applications qui ont été proposées.

- De manière générale, voir (et restreindre) les aspects à des propriétés semble être une solution intéressante pour maîtriser l'impact sémantique du tissage. Notre approche permet au programmeur de déduire le comportement de l'application tissée sans regarder (ou imaginer) son code.
- Nous avons montré que l'abstraction, le tissage et la concrétisation tels que nous les définissons permettent d'imposer la propriété au programme. Nous avons prouvé la correction de l'approche dans la cadre de programmes séquentiels et déterministes. Même si les preuves de correction des chapitres 3 et 4 n'ont pas été détaillées, nous pensons qu'elles sont très proche de celles du chapitre 2.
- La contribution de cette thèse tient aussi dans les deux applications proposées qui sont à notre connaissance les seuls travaux où la PPA est appliquée à l'ordonnancement et la disponibilité. Nous avons proposé des langages d'aspects dédiés et décrit comment étendre notre cadre à des programmes parallèle et des propriétés temporisées.

Une première perspective pour continuer ces travaux est de combiner les aspects d'ordonnancement et les aspects de disponibilité. pour des services décrits comme des réseaux de composants. La combinaison de ces deux types d'aspects offre la possibilité d'optimisations supplémentaires. De manière naïve, on peut tisser l'aspect d'ordonnancement sur le réseau de composants représentant le service et ensuite tissé l'aspect de disponibilité sur le service séquentialisé. Cependant on peut aussi tisser les deux aspects simultanément. Le tissage de l'aspect d'ordonnancement peut laisser des choix d'exécution qui sont résolu automatiquement lors de l'étape de séquentialisation. L'aspect de disponibilité peut quand à lui couper des traces et terminer le programme en cas de violation de la propriété. L'idée serait d'optimiser l'étape de séquentialisation du processus de fusion en faisant les choix qui garantissent ou facilitent les propriétés de disponibilité. L'aspect d'ordonnancement peut également exprimer des ordres d'allocations des ressources des différents services et prévenir ainsi des interblocages.

Dans notre approche, les aspects représentent des propriétés de sûreté qui sélectionnent des traces d'exécutions. Une seconde perspective plus générale serait d'étendre l'approche à d'autres effets que la sélection des traces et ainsi d'augmenter l'expressivité des aspects. Le formalisme des *Edit Automata* [LBW05] permet de modéliser des moniteurs qui peuvent *transformer* la trace d'exécution pour qu'elle respecte une propriété. Ces moniteurs peuvent couper la trace, mais aussi ajouter et supprimer des événements dans la trace. La suppression d'événements peut servir à modéliser des transactions et ainsi offre la possibilité d'imposer des propriétés de vivacité sur le programme [Gra06]. Par exemple, on peut imposer la propriété que pour chaque `begin` dans la trace on trouve un `end`, en commençant une transaction au premier `begin` et en effectuant le commit de cette transaction quand le nombre de `end` est égal au nombre de `begin`. Dans le cas d'une exécution qui ne vérifie pas la propriété, la transaction ne termine pas de telle sorte que la trace d'exécution respecte la propriété. Le tissage d'*Edit Automaton* sur un programme nous semble donc une piste de recherche intéressante à explorer pour augmenter l'expressivité de nos aspects tout en gardant un cadre formel permettant de raisonner sur l'application de base et les aspects.

A plus long terme, un projet naturel serait de décrire et d'implémenter un environnement complet de programmation par propriétés. Dans cet environnement, le programme serait décrit par un programme de base et par un ensemble de propriétés non fonctionnelles. Cet environnement garantirait d'une part que le programme tissé respecte les propriétés définies par les aspects. Cet environnement nécessite des travaux suivant trois axes :

– la définition d'un cadre suffisamment expressif pour décrire et tisser une large classe de propriétés. L'approche proposée dans cette thèse et la seconde perspective (tisser des *Edit Automata*) est un point de départ ;
– l'application du cadre à différents domaines et la définition de langages d'aspects dédiés. Nous avons déjà appliqué l'approche de cette thèse à l'ordon-

nancement et la disponibilité. Le domaines de la tolérance aux fautes et des transactions est un candidat naturel comme nouveau domaine applicatif.
- l'implémentation de ces techniques dans un prototype semble indispensable afin de valider les différentes applications de l'approche et de permettre de nouvelles utilisations et extensions par la communauté.

Bibliographie

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2) :183–235, 1994.

[AGMO06] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. Overview of caesarj. *Transactions on AOSD I, LNCS*, 3880 :135 – 173, 2006.

[Alu99] Rajeev Alur. Timed automata. In *Computer Aided Verification*, pages 8–22, 1999.

[AMS04] K. Altisen, F. Maraninchi, and D. Stauch. Exploring aspects in the context of reactive systems. In *Workshop on the Foundations of Aspect-Oriented Languages (FOAL)*, 2004.

[AMS06] Karine Altisen, Florence Maraninchi, and David Stauch. Aspect-oriented programming for reactive systems : Larissa, a proposal in the synchronous framework. *Science of Computer Programming, Special Issue on Foundations of Aspect-Oriented Programming*, 2006. To appear.

[And01] James H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *REFLECTION '01 : Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 187–209, London, UK, 2001. Springer-Verlag.

[AR05] N. M. Ali and A. Rashid. A state-based join point model for aop. In *Proceedings of a workshop on views, aspects and roles (VAR)*, 2005.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[AWBB94] M. Aksit, K. Wakita, J. Bosch, and L. Bergmans. Abstracting object interactions using composition filters. In *ECOOP'93 : Workshop on Object-Based Distributed Programming*, LNCS vol. 791, pages 152–184, 1994.

[BDM99] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers : a new facility for resource management in server systems. In *OSDI '99 : Proceedings of the third symposium on Operating systems design and implementation*, pages 45–58. USENIX Association, 1999.

[BdPK05] Sara Bouchenak, Noël de Palma, and Sacha Krakowiak. Tolérance aux fautes dans les grappes d'applications Internet. In *4ème Conférence Française sur les Systèmes d'Exploitation (CFSE 2005)*, Le Croisic, France, April 2005.

[BG92] Gérard Berry and Georges Gonthier. The esterel synchronous programming language : design, semantics, implementation. *Sci. Comput. Program.*, 19(2) :87–152, 1992.

[BH05] N. Boucke and T. Holvoet. State-based. join-points : Motivation and requirements. In *International Workshop on Dynamic Aspects at AOSD (DAW'05)*, pages 1–4, March 2005.

[BHV01] Walter Binder, Jane G. Hulaas, and Alex Villaz. Portable resource control in java. In *OOPSLA '01 : Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 139–155. ACM Press, 2001.

[BJJR04] G. Bruns, R. Jagadeesan, A. Jeffrey, and J. Riely. muabc : A minimal aspect calculus. In *CONCUR 2004 : Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 209–224, August 2004.

[Boa96] Board of Inquiry Report. Ariane 5 - flight 501 failure. Technical report, European Space Agency, July 1996.

[BY04] J. Bengtsson and W. Yi. Timed automata : Semantics, algorithms and tools. In *Concurrency and Petri Nets*, LNCS vol. 3098. Springer–Verlag, 2004.

[CC97] R. Clayton and K. Calvert. Augmenting the proebsting-watterson filter fusion algorithm, 1997. http ://citeseer.ist.psu.edu/186288.html.

[CF00] Thomas Colcombet and Pascal Fradet. Enforcing trace properties by program transformation. In *Symposium on Principles of Programming Languages (POPL'00)*, pages 54–66, 2000.

[CKL+99] Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, Marc Massot, Sandra Moral, Claudio Passerone, Yosinori Watanabe, and Alberto L. Sangiovanni-Vincentelli. Task generation and compile-time scheduling for mixed data-control embedded software. Technical Report LSI-99-47-R, Dept. of Software, Universitat Politecnica de Catalunya, 1999.

[CS99] F. Cuppens and C. Saurel. Towards a formalization of availability and denial of service. In *Information Systems Technology Panel Symposium on Protecting Nato Information Systems in the 21st century*, 1999.

[DBNS06] R. Douence, D.Le Botlan, J. Noyé, and M. Südholt. Concurrent aspects. Technical Report RR-5873, INRIA - Rennes - Equipe OBASCO, March 2006.

[DDFL06] S. Djoko Djoko, R. Douence, P. Fradet, and D. Le Botlan. Common aspect semantics base. *Livrable. Document public du réseau d'excellence européen AOSD*, 2006.

[DFL+05] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Ségura-Devillechaise, and M. Südholt. An expressive aspect language for system applications with arachne. In *AOSD '05 : Proceedings of the 4th international conference on Aspect-oriented software development*, pages 27–38, New York, NY, USA, 2005. ACM Press.

[DFS02] Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In *Proc. of Conference on Generative Programming and Component Engineering (GPCE'02)*, LNCS vol. 2487. Springer–Verlag, 2002.

[DFS04] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In Karl Lieberherr, editor, *Proc. of the 3rd Int' Conf. on Aspect-Oriented Software Development(AOSD'04)*, pages 141–150. ACM Press, mar 2004.

[DGB+06] P. Durr, G. Gulesir, L. Bergmans, M. Aksit, and R. van Engelen. Applying aop in an industrial context. In *Workshop on Best Practices in Applying Aspect-Oriented Software Development*, 2006.

[DSBA05] P. Durr, T. Staijen, L. Bergmans, and M. Aksit. Reasoning about semantic conflicts between aspects. In *EIWAS 2005 : 2nd European Interactive Workshop on Aspects in Software*, 2005.

[Duc06] Allison Duck. Implementation of aop in non-academic projects. In *AOSD 2006 - Industry Track Proceedings*, 2006.

[DW06] D. S. Dantas and D. Walker. Harmless advice. In *POPL '06 : Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 383–396, 2006.

[Eme90] E. Allen Emerson. *Temporal and modal logic.* MIT Press, Cambridge, MA, USA, 1990.

[FD06] J. Fabry and T. D'Hondt. Kala : Kernel aspect language for advanced transactions. In *Proceedings of ACM Symposium on Applied Computing 2006*, volume 2, pages 1615 – 1620. ACM Press, 2006.

[FH04] P. Fradet and Hong Tuan Ha S. Network fusion. In *Programming Languages and Systems : Second Asian Symposium, (APLAS'04)*, LNCS vol. 3302, 2004.

[FH05] P. Fradet and S. Hong Tuan Ha. Systèmes de gestion de ressources et aspects de disponibilité. In *2e Journée Francophone sur le Développement de Logiciels Par Aspects (JFDLPA 2005)*, Lille, France, September 2005. (In French).

[FH06] Pascal Fradet and S. Hong Tuan Ha. Systèmes de gestion de ressources et aspects de disponibilité. *L'objet*, 12(2-3) :183–210, 2006.

[FL06] P. Fradet and R. Lämmel. Special issue on foundations of aspect-oriented programming. *Sci. Comput. Program.*, 63(3), 2006.

[FP06] Johan Fabry and Nicolas Pessemier. Kala : A domain-specific solution to tangled aspect code. In *Proceedings of the 1st GPCE Workshop on Domain-Specific Aspect Languages (DSAL'06)*, oct 2006.

[GB03] Marc Geilen and Twan Basten. Requirements on the execution of Kahn process networks. In *European Symposium on Programming (ESOP'03)*, pages 319–334, 2003.

[GK06] M. Goldman and S. Katz. Modular generic verification of LTL properties for aspects. In *Workshop on the Foundations of Aspect-Oriented Languages (FOAL)*, 2006.

[Gra06] H. Grall. The synthesis of reference monitors. Working paper, dec 2006.

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language lustre. *IEEE Trans. Softw. Eng.*, 18(9) :785–793, 1992.

[HMU00] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computability.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[HNSY92] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. In *7th Symposium of Logics in Computer Science*, pages 394–406, 1992.

[HR06] Peter Hui and James Riely. Temporal aspects as security automata. In *Workshop on the Foundations of Aspect-Oriented Languages (FOAL)*, 2006.

[JLZ06] D. Walker J Ligatti and S. Zdancewic. A type-theoretic interpretation of pointcuts and advice. *Science of Computer Programming*, 2006.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress (Information Processing'74)*, pages 471–475, 1974.

[Kat06] S. Katz. Aspect categories and classes of temporal properties. In *T. Aspect-Oriented Software Development I*, pages 106–134, 2006.

[KHH+01a] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with aspectj. *Commun. ACM*, 44(10) :59–65, 2001.

[KHH+01b] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072 :327–355, 2001.

[KHJ06] J. Klein, L. Hélouët, and J.-M. Jézéquel. Semantic-based weaving of scenarios. In *AOSD '06 : Proceedings of the 5th international conference on Aspect-oriented software development*, pages 27–38, 2006.

[KLM+97] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, 1997.

[KS99] Nils Klarlund and Michael L. Schwartzbach. A domain-specific language for regular sets of strings and trees. *IEEE Trans. Softw. Eng.*, 25(3) :378–386, 1999.

[LÖ2] R. Lämmel. A semantical approach to method-call interception. In *Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002)*, pages 41–55, April 2002.

[Lap92] J.-C. Laprie. *Dependability : Basic Concepts and Terminology*. Dependable Computing and Fault-Tolerant Systems. Springer-Verlag, 1992.

[LBW05] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata : enforcement mechanisms for run-time security policies. *Int. J. Inf. Sec.*, 4(1-2) :2–16, 2005.

[Lin98] Bill Lin. Software synthesis of process-based concurrent programs. In *Proceedings of the 1998 Conference on Design Automation (DAC-98)*, pages 502–505, 1998.

[LK98] C. V. Lopes and G. Kiczales. Recent developments in aspectj. In *ECOOP '98 : Workshop on Object-Oriented Technology*, pages 398–401. Springer-Verlag, 1998.

[LL00] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *ICSE '00 : Proceedings of the 22nd international conference on Software engineering*, pages 418–427, 2000.

[LM91] Xuejia Lai and James L. Massey. A proposal for a new block encryption standard. In *EUROCRYPT '90 : Proceedings of the workshop on the theory and application of cryptographic techniques on Advances in cryptology*, pages 389–404, 1991.

[LMR05] Xianfeng Li, Tulika Mitra, and Abhik Roychoudhury. Modeling control speculation for timing analysis. *Real-Time Syst.*, 29(1) :27–58, 2005.

[Lop97] C. V. Lopes. *D : A Language Framework for Distributed Programming*. Phd thesis, College of Computer Science, Northeastern University, nov 1997.

[LPY97] K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2) :134–152, 1997.

[LZ97] Jussipekka Leiwo and Yuliang Zheng. A method to implement a denial of service protection base. In *ACISP '97 : Proceedings of the Second Australasian Conference on Information Security and Privacy*, pages 90–101. Springer-Verlag, 1997.

[Mil94] J. K. Millen. A resource allocation model for denial of service protection. *Journal of Computer Security*, 2(2), 1994.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications : a formal introduction.* John Wiley & Sons, Inc., New York, NY, USA, 1992.

[NNH99] F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.

[Par95] Thomas Martyn Parks. *Bounded scheduling of process networks.* PhD thesis, University of California, Berkeley, 1995.

[PK89] P. Puschner and Ch. Koza. Calculating the maximum, execution time of real-time programs. *Real-Time Syst.*, 1(2) :159–176, 1989.

[PW96] Todd A. Proebsting and Scott A. Watterson. Filter fusion. In *Symposium on Principles of Programming Languages (POPL'96)*, pages 119–130, 1996.

[RSB04] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *SIGSOFT '04/FSE-12 : Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 147–158, 2004.

[Rus94] J. Rushby. Critical system properties : Survey and taxonomy. *Reliability Engineering and Systems Safety*, 43(2) :189–219, 1994.

[Sch00] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1) :1–50, February 2000.

[SDMML03] M. Ségura-Devillechaise, J.-M. Menaud, G. Muller, and J. L. Lawall. Web cache prefetching as an aspect : towards a dynamic-weaving based solution. In *AOSD '03 : Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 110–119, 2003.

[STZE99] K. Strehl, L. Thiele, D. Ziegenbein, and R. Ernst. Scheduling hardware/software systems using symbolic techniques. In *Proceedings of the seventh international workshop on Hardware/software codesign (CODES '99)*, pages 173–177, 1999.

[TMY05] H. Tatsuzawa, H. Masuhara, and A. Yonezawa. Aspectual caml : an aspect-oriented functional language. In *Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, pages 39–50, 2005.

[TSZ+99] L. Thiele, K. Strehl, D. Ziegenbein, R. Ernst, and J. Teich. Funstate - an internal design representation for codesign. In *International Conference on Computer-Aided Design (ICCAD '99)*, pages 558–565, 1999.

[Wad84] P. Wadler. Listlessness is better than laziness. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 45–52, 1984.

[Wad90] P. Wadler. Deforestation : transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2) :231–248, 1990.

[WKD04] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Program. Lang. Syst.*, 26(5) :890–910, 2004.

[YG90] Che-Fn Yu and V. D. Gligor. A specification and verification method for preventing denial of service. *IEEE Trans. Softw. Eng.*, 16(6) :581–592, 1990.

Annexe A : Preuves du chapitre 2

Cette annexe présente les preuves des propriétés énoncées dans le chapitre 2. Ces preuves consistent pour la plupart à faire une induction sur la taille du mot et à appliquer les différentes définitions dans le bon ordre.

Cette annexe A doit être mise à jour pour correspondre à la dernière version de nos notations et vous sera envoyée prochainement.

Annexe B : Autres opérations sur les automates

Cette annexe complémente la partie "Rappels sur les automates" du chapitre 2 en décrivant les opérations sur les automates qui sont utilisées dans la partie 2.9.4.

Coaccessibilité Un état $\alpha \in \Sigma$ est dit coaccessible s'il y a un mot dans $L(A)$ qui passe par cet état. Nous considérons l'opération qui supprime tous les états qui ne sont pas coaccessibles.

Soit l'automate $A = (\Sigma, \alpha_0, \mathcal{E}, f, \mathcal{F})$, la coaccessibilité de l'automate A est noté $CoAc(A)$ et est défini par $Ac(A) = (\Sigma_{coac}, \alpha_{coac0}, \mathcal{E}, f_{coac}, \mathcal{F})$ avec :
- $\Sigma_{coac} = \{\alpha \in \Sigma | \exists w \in \mathcal{E}^*, \alpha' \in \mathcal{F}.\alpha \xrightarrow{w}{}^{*} \alpha'\}$
- $\alpha_{coac0} = \begin{cases} \alpha & \text{si } \alpha_0 \in \Sigma_{coac} \\ \text{non défini} & \text{sinon} \end{cases}$
- f_{coac} est défini en restreignant la fonction f au domaine des états accessible Σ_{coac}.

Cette opération peut diminuer $T(A)$, le langage reconnu par l'automate, mais ne modifie pas $L(A)$, le langage accepté par l'automate.

Composition parallèle Nous considérons les 2 automates $A_a = (\Sigma_a, \alpha_{a0}, \mathcal{E}_a, f_a, \mathcal{F}_a)$ et $A_b = (\Sigma_b, \alpha_{b0}, \mathcal{E}_b, f_b, \mathcal{F}_b)$.

Nous définissons aussi la composition parallèle entre 2 automates. Le produit entre les automates A_a et A_b est noté $A_a \parallel A_b$ et est défini par l'automate $(\Sigma_a \parallel \Sigma_b, (\alpha_{a0}, \alpha_{b0}), \mathcal{E}_a \cup \mathcal{E}_b, f, \mathcal{F}_a \times \mathcal{F}_b)$, avec :

$$f = \begin{cases} (\alpha_a, \alpha_b) \xrightarrow{k} (\alpha'_a, \alpha'_b) & \text{si } \alpha_a \xrightarrow{k}_a \alpha'_a \wedge \alpha_b \xrightarrow{k}_b \alpha'_b \\ (\alpha_a, \alpha_b) \xrightarrow{k} (\alpha'_a, \alpha_b) & \text{si } \alpha_a \xrightarrow{k}_a \alpha'_a \wedge k \notin \mathcal{E}_b \\ (\alpha_a, \alpha_b) \xrightarrow{k} (\alpha_a, \alpha'_b) & \text{si } \alpha_b \xrightarrow{k}_b \alpha'_b \wedge k \notin \mathcal{E}_a \\ \text{non défini} & \text{sinon} \end{cases}$$

De même que pour le produit, un état de la composition parallèle modélise l'état des deux automates et est noté comme une paire (α_a, α_b). Une transition du produit sur le symbole $k \in \mathcal{E}_a \cap \mathcal{E}_b$ est possible uniquement si les deux automates peuvent effectuer une transition Une transition sur le symbole $k \in (\mathcal{E}_a\ \mathcal{E}_b)$ (resp. $k \in \mathcal{E}_b \times \mathcal{E}_a$), peut être effectué et dans ce cas seul l'automate A_a (resp. A_b) évolue.

La figure B.1 montre le résultat de la composition parallèle entre les automates A_1 et A_2 présentés dans la section 2.2.

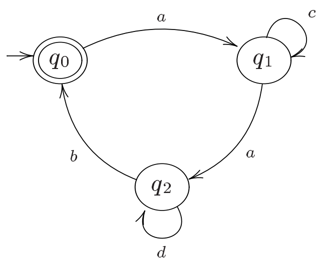


FIG. B.1 – Composition parallèle $A_1 \parallel A_2$

Annexe C : Sémantique temporisée des services

Cette annexe présente la sémantique temporisée des services du chapitre 4.

En supposant une fonction *tps* qui donne le temps pris par toute action, on peut définir la sémantique temporisée d'un service en ajoutant une horloge globale (un réel) h dans les états (l'état initial devenant $(l_0, s_0, 0)$) et en modifiant la fonction de transition pour prendre en compte le temps pris par l'action, c'est à dire en réécrivant toute transition $(l_1, s_1) \xrightarrow{a}_S (l_2, s_2)$ en

$$(l_1, s_1, h) \xrightarrow{a}_S (l_2, s_2, h + tps(a))$$

Même si les services originels ne comportent ni minuteur ni test d'horloge, ce style d'instruction est introduit par les aspects. Nous décrivons ici la sémantique des services obtenus après tissage.

Les commandes temporelles introduites par les aspects sont les suivantes :

$$C ::= \mathit{start}(t) \mid \mathit{wait}(t, k) \mid \mathit{reset}(i, k) \mid \mathit{cancel}(i) \mid \ldots$$

où t dénote un identificateur de minuteur, i un identificateur d'interuption et k un entier.

Les tests sont étendus par des comparaisons de minuteurs :

$$g ::= t \odot k \mid \ldots \text{ avec } \odot \in \{<, >, \leq, \ldots\}$$

La commande $\mathit{start}(t)$ déclenche (initialize à zéro) un minuteur t qui pourra être comparé à des entiers dans des tests. Les minuteurs sont également ou utilisés pour retarder l'exécution à l'aide de la commande $\mathit{wait}(t, k)$ qui attend jusqu'à ce que $t \geq k$.

La commande $\mathit{reset}(i, k)$ programme une interruption i pour se déclencher dans k unités de temps. L'instruction $\mathit{cancel}(i)$ déprogramme l'interruption i.

La formalisation de ces comportements nécessite de raffiner la mémoire s en un triplet (m, τ, ι) ou τ et ι dénotent deux environnements pour les minuteurs et les interruptions respectivement et m représente le reste de la mémoire.

Les environnements

$$\tau : \mathit{Timer} \rightarrow \mathit{Int} \qquad \iota : \mathit{Interruption} \rightarrow \mathit{Int}$$

associent chaque identificateur (de minuteur ou d'interruption) à un entier qui représente un temps de déclenchement. Initialement, ces environnements sont vides

$$\forall x. \tau_0(x) = \perp \quad \wedge \quad \iota_0(x) = \perp$$

Deux opérations permettent d'ajouter et retirer un identificateur à un environnement. Ces opérations sont définies comme suit :

$$\begin{array}{rcll} \rho[t \mapsto k](x) & = & \rho(x) & \text{if } x \neq t \\ & = & k & \text{if } x = t \end{array} \qquad \begin{array}{rcll} (\rho \backslash t)(x) & = & \rho(x) & \text{if } x \neq t \\ & = & \perp & \text{if } x = t \end{array}$$

Les règles de transitions deviennent :

$$\textsc{Comm} \quad \frac{l_1 : c \rightsquigarrow l_2 \in S \quad s_1 = (m, \tau, \iota) \quad \forall i.\ \iota(i) > h \quad \mathcal{C}[\![c]\!] s_1 h = s_2}{(l_1, s_1, h) \xrightarrow{c}_S (l_2, s_2, h + tps(c))}$$

Une commande s'exécute comme précédemment si aucune interruption ne doit être déclenchée ($\forall i.\iota(i) > h$). L'horloge globale est mise à jour pour prendre en compte le temps d'exécution d'une commande. Les règles Then et Else sont modifiées de la même façon. Une règle supplémentaire est nécessaire pour représenter le déclenchement d'une interruption.

$$\textsc{Reset} \quad \frac{s_1 = (m_1, \tau, \iota) \quad \exists i.\iota(i) \leq h}{(l_1, s_1, h) \xrightarrow{reset\ i}_S (l_0, s_0, 0)}$$

Le déclenchement d'une interruption se produit lorsque l'entier associé à une interruption est inférieur ou égal au temps courant. L'effet d'une interruption consiste à retourner dans l'état initial. Dans le même temps l'instruction *Reset* libère les ressources et envoie une notification à l'utilisateur courant.

La sémantique des nouvelles commandes est donnée par la fonction $\mathcal{C}$ qui prend maintenant également en paramètre l'horloge.

$$\begin{array}{lcl} \mathcal{C}[\![start(t)]\!](m, \tau, \iota)\ h & = & (m, \tau[t \mapsto h], \iota) \\ \mathcal{C}[\![reset(i,k)]\!](m, \tau, \iota)\ h & = & (m, \tau, \iota[i \mapsto h + k]) \\ \mathcal{C}[\![cancel(i)]\!](m, \tau, \iota)\ h & = & (m, \tau, \iota \backslash i) \end{array}$$

L'instruction *wait* (t,k) ne modifie pas la mémoire mais fait écouler le temps. Son effet est modélisé par la fonction *tps* qui prends maintenant l'environnement des minuteurs, des interruptions et l'horloge. Depuis le déclenchement de t il s'est écoulé $h - \tau(t)$, il faut donc attendre $W = max(0, k - (h - \tau(t)))$. Toutefois, il faut prendre en compte le cas où une interruption doit se déclencher avant. Il faut attendre uniquement $I_{min} = Min_i(\iota(i)) - h$ (*i.e.*, le temps minimal avant la prochaine interruption) si celui-ci est inférieur à W. D'où :

$$tps(wait(t,k), \tau, \iota, h) = h + min(W, I_{min})$$

Pour prendre en compte les comparaisons de minuteurs, la fonction $\mathcal{G}$ prend l'horloge globale en paramètre. Par exemple, le test d'égalité avec un entier est spécifié par

$$\mathcal{G}[\![t = k]\!](m, \tau, \iota)\ h = (h - \tau(t) = k)$$

Les autres opérateurs de comparaison sont spécifiés de la même manière.