Larissa, un langage d'aspects pour le développement des systèmes réactifs sûrs

David Stauch, Karine Altisen, Florence Maraninchi

Laboratoire Verimag,
Centre équation - 2, avenue de Vignate, 38610 GIÈRES — France

RÉSUMÉ. *Les systèmes réactifs sont souvent programmés dans des langages dédiés, auxquels on ne peut pas appliquer les langages d'aspects existants. Pour cela, nous avons créé Larissa, un langage d'aspects pour le langage réactif synchrone Argos. Dans cet article, nous décrivons Larissa, et nous l'utilisons pour réaliser un exemple typique d'un système réactif, le contrôleur de porte d'un tramway.*

ABSTRACT. *Reactive systems are often programmed in dedicated languages, to which existing aspect languages cannot be applied. Therefore, we created Larissa, an aspect language for the synchronous programming language Argos. This article describes Larissa and its use to implement an example for a reactive system, a controller of the door of a tramway.*

MOTS-CLÉS : *programmation par aspects, langages synchrones, systèmes embarqués, sémantique formelle*

KEYWORDS: *aspect-oriented programming, embedded systems, synchronous languages, formal semantics*

1. Introduction

La programmation par aspects ajoute des mécanismes à un langage de base qui permettent d'encapsuler des *préoccupations transverses*, c'est-à-dire des préoccupations qui ne peuvent pas être encapsulées dans un module par la décomposition offerte par le langage de base. La programmation par aspects permet d'exprimer des préoccupations transverses dans des *aspects*, et de les *tisser* dans un programme. En général, un aspect comprend une *coupe*, qui choisit des *points de jointure* dans une exécution du programme de base, et un *insert*, qui modifie l'exécution à ces points. Des langages d'aspects tels que AspectJ (Kiczales *et al.*, 2001) ont eu beaucoup de succès, mais ils ne peuvent pas être appliqués à tous les systèmes existants. Par exemple, beaucoup de systèmes réactifs sont programmés dans des langages dédiés, auxquels AspectJ ne peut pas être appliqué.

Les systèmes réactifs sont des systèmes qui sont en interaction constante avec leur environnement et qui sont souvent utilisés dans des contextes critiques, comme des avions ou des centrales nucléaires. Ils nécessitent des langages de programmation dédiés, qui permettent la vérification formelle et qui sont plus adaptés à un style de programmation d'interaction constante avec l'environnement. Parmi ces langages dédiés, les langages synchrones ont eu beaucoup de succès. Ils offrent un parallélisme de haut niveau, avec une sémantique claire, qui est ensuite compilée vers un programme séquentiel. Des préoccupations transverses apparaissent dans ces langages, mais les langages d'aspects établis comme AspectJ ne peuvent pas être utilisés, d'une part parce qu'ils manquent de bases sémantiques nécessaires pour la vérification formelle, et d'autre part parce que les langages de bases sont très différents.

Pour cela nous avons développé Larissa (Altisen *et al.*, 2006a), qui est un langage d'aspects pour Argos. Argos est un langage synchrone à base d'automates qui est très simple, mais dispose de tous les éléments caractéristiques d'un langage synchrone. Il semble donc un bon candidat comme un langage de base pour Larissa. Un critère de développement pour Larissa était de lui donner une sémantique très simple, ce qui permet de garantir des propriétés intéressantes, comme par exemple la préservation de l'équivalence sémantique.

Dans cet article, nous utilisons Argos et Larissa pour réaliser un contrôleur de porte de tramway. Il y a deux versions différentes du contrôleur, une version qui contrôle l'ouverture et la fermeture automatique d'une porte, et une version qui contrôle en plus une passerelle extractible. Larissa est utilisé pour ajouter le contrôle de la passerelle au contrôleur, ce qui permet de réutiliser la première version pour réaliser la deuxième sans modifier la première. L'exemple montre donc qu'on peut construire un système par morceaux avec des aspects. Cela aide à la conception correcte, et permet une meilleure séparation des préoccupations.

L'article a la structure suivante : la section 2 décrit le langage de base Argos, la section 3 le langage d'aspects Larissa, la section 4 décrit le contrôleur de porte, la section 5 expose quelques travaux voisins, et la section 6 contient une conclusion.

2. Argos

Un programme Argos décrit le noyau réactif d'un système réactif. Un système réactif est un système qui est en interaction constante avec son environnement, et qui reçoit donc des signaux d'entrée de l'environnement et émet des signaux de sortie vers l'environnement. En Argos, ces signaux d'entrée et de sortie ont uniquement des valeurs booléennes. Alors que l'environnement évolue d'une façon continue, notre système réactif perçoit le temps comme échantillonné en instants, parce que c'est un programme. À chaque instant, le programme réagit aux entrées en émettant des sorties et en mettant à jour sa mémoire interne. Une telle réaction est atomique : le système ne lit pas les entrées pendant qu'il calcule les sorties et met à jour sa mémoire. Cette propriété caractérise les langages synchrones, dont Argos fait partie.

Argos est un langage à base d'automates. Ses composants de base sont des automates avec des transitions étiquetées avec des entrées et des sorties. Des composants plus complexes peuvent être obtenus en branchant des composants avec des opérateurs, notamment le produit parallèle entre automates et l'encapsulation, qui limite la portée d'un signal. Avec ces opérateurs, les composants peuvent communiquer entre eux : un signal local peut être émis par un composant et lu par un autre. La communication se fait par la diffusion synchrone, qui ne bloque pas les composants communicants qui émettent.

Les programmes d'Argos sont déterministes et complets, c'est-à-dire que pour toute séquence d'entrées il y a une exécution unique du programme. La sémantique d'Argos est définie formellement par des traces d'exécution. Ces traces sont définies seulement avec les valeurs des entrées et des sorties à chaque instant, les états n'en font pas partie. L'équivalence sémantique entre programmes choisie est l'équivalence de traces.

2.1. *Syntaxe des constructions principales*

Le langage complet est décrit dans (Maraninchi *et al.*, 2001). Ici, nous le décrivons partiellement en utilisant un exemple. La figure 1(a) est un programme Argos composé de trois automates qui décrit un compteur de `a` modulo huit.

Dans les figures, les états de l'automate sont des rectangles aux coins arrondis, les transitions sont des flèches, et un automate est un ensemble d'états et transitions connectés. Les trois automates du programme dans la figure 1(a) ont les ensembles d'états suivants : `{A0, A1}`, `{B0, B1}`, `{C0, C1}`. Les transitions sont étiquetées par une condition booléenne sur les entrées et un ensemble de sorties émises, séparés par un `/`. Si l'ensemble des sorties est vide, il peut être omis. Dans la condition, la négation est notée par une ligne superposée et la conjonction par un point (par exemple `c/fin` ou $\texttt{a}.\overline{\texttt{b}}$). L'état initial est indiqué par une flèche sans état source. Un état peut être nommé, mais les noms sont seulement des commentaires et n'interviennent pas dans la définition de la sémantique du programme. Une flèche peut avoir plusieurs étiquettes,

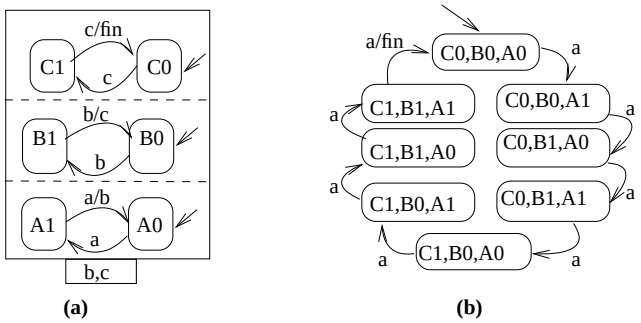


Figure 1. *Deux programmes Argos pour le compteur de* `a` *modulo 8.*

représentant plusieurs transitions. Par convention, chaque automate est complet : si un état n'a pas de transition pour une combinaison d'entrées, cela signifie qu'il y a une transition boucle sans sorties pour cette combinaison d'entrées.

Les trois automates du programme sont mis en parallèle, ce qui est noté par les lignes pointillées entre eux. Les signaux `b` et `c` sont encapsulés, et leur portée est limitée aux trois automates mis en parallèle. L'encapsulation se note avec les signaux encapsulés dans un petit rectangle en dessous du programme où ils sont valides. L'ensemble des entrées et l'ensemble des sorties constituent l'interface du programme. Les signaux encapsulés n'y apparaissent pas.

2.2. *Sémantique intuitive*

Nous donnons la sémantique intuitive des opérateurs, en expliquant le comportement du compteur. Le comportement est un automate plat, montré dans la figure 1(b). Le compteur a une seule entrée `a` et une seule sortie `fin`. Son comportement global est défini par : son état initial est `C0,B0,A0` ; quand il a reçu l'entrée `a` n fois, il est dans l'état $\texttt{C}_k\texttt{,B}_j\texttt{,A}_i$, où $i + 2j + 4k = n \bmod 8$; `fin` est émis tous les 8 `a`. Ce comportement est obtenu en connectant trois compteurs modulo deux. Le premier réagit à l'entrée externe `a` et émet `b` tous les deux `a`, le deuxième réagit à `b` et émet `c` tous les deux `b`, et le troisième émet `fin` tout les deux `c`. Parce que les réactions sont synchrones, une réaction à laquelle participe les trois bits est une réaction globale atomique (par exemple, la réaction à `a` de `C0,B1,A1` à `C1,B0,A0`).

Comme nous l'avons fait pour l'exemple, la sémantique de chaque opérateur Argos est donnée par une traduction d'un programme complexe en automate plat équivalent. La sémantique pour un automate plat est alors donnée par son ensemble de traces d'exécution, donc des entrées et sorties à chaque instant.

3. Larissa

Les opérateurs Argos sont déjà très puissants, mais il y a des cas où ils ne sont pas suffisants pour modulariser toutes les préoccupations d'un programme. Une petite modification du comportement du programme global peut nécessiter une modification de tous les composants parallèles. Dans le sous-ensemble d'Argos présenté ici, mais aussi dans le langage de programmation synchrone Lustre (Halbwachs, 1993), la réinitialisation d'un programme est un exemple pour une telle préoccupation.

Étant donné que le but de la programmation par aspects est de spécifier de telles préoccupations transverses, nous avons proposé Larissa, une extension d'aspects pour Argos, qui permet la modularisation de certains problèmes récurrents dans la programmation des systèmes réactifs, comme par exemple la réinitialisation. Cela nous a amenés à définir un nouvel opérateur, l'opérateur de tissage d'aspect. Cet opérateur a des propriétés sémantiques importantes : il préserve le déterminisme et la complétude des programmes, et aussi l'équivalence entre programmes.

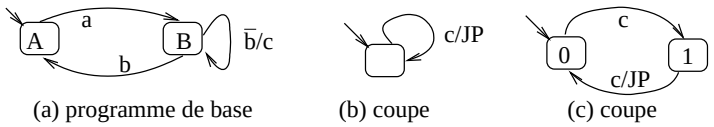
Toutes les extensions d'aspect existantes utilisent deux notions : les coupes et les inserts. Les coupes décrivent une propriété générale sur des points du programme (appelés points de jointure) où l'aspect intervient (par exemple toutes les méthodes de la classe `X`, ou toutes les méthodes qui s'appellent `toto`). L'insert spécifie ce qui est fait à ces points de jointure (exécuter du code avant d'appeler la méthode choisie, par exemple). Pour Larissa, nous avons adopté cette approche : un aspect est donné par sa coupe et son insert.

3.1. *Choix des points de jointure*

En Larissa, on n'exprime pas les coupes en terme de la structure interne du programme de base. Par exemple, nous ne permettons pas aux coupes de se référer explicitement aux noms d'états (contrairement à AspectJ, où les coupes peuvent se référer aux noms des méthodes privées). En conséquence, les coupes se réfèrent seulement au comportement observable, donc aux entrées et sorties globales du programme auquel l'aspect est appliqué.

Dans la famille des langages synchrones, où la communication entre composants parallèles se fait par la diffusion synchrone, les observateurs (Halbwachs *et al.*, 1993) sont un mécanisme puissant et bien connu, qui peut être utilisé pour exprimer des coupes. Un observateur est un programme qui observe les entrées et les sorties du programme de base sans modifier son comportement, et qui calcule une propriété de sûreté (au sens des propriétés de sûreté/vivacité comme définies dans (Lamport, 1977)).

Nous exprimons donc les coupes en Larissa avec des observateurs, qui choisissent un ensemble de *transitions de jointure* en émettant une seule sortie `JP`, appelée *signal de jointure*. Une transition T d'un programme P est choisie comme transition de jointure, si, pendant l'exécution parallèle de P et de la coupe, `JP` est émis quand T est



(a) programme de base (b) coupe (c) coupe

Figure 2. *Des coupes d'exemple.*

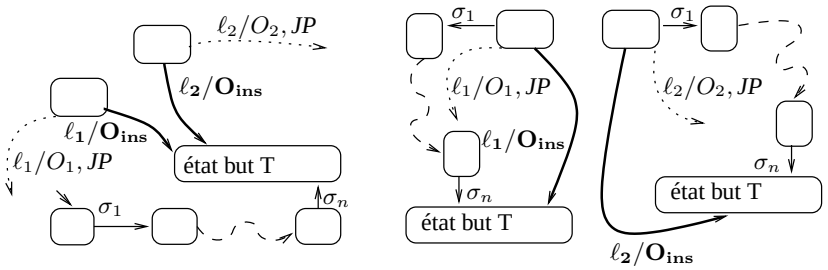
prise. Ceci est réalisé en mettant en parallèle le programme et la coupe et en choisissant ensuite les transitions de l'automate plat obtenu qui émettent `JP`. La figure 2 illustre le mécanisme des coupes. La coupe (b) choisit toute transition émettant `c` : dans le programme de base (a), la transition boucle est choisie comme transition de jointure. La coupe (c) spécifie toutes les occurrences paires de `c` : aucune transition du programme de base (a) ne correspond directement à cette condition. Comme les transitions de jointure sont choisies sur le produit parallèle du programme de base et de la coupe, la coupe introduit de la mémoire : le programme tissé mémorise si `c` a été émis un nombre pair ou impair de fois.

Les coupes peuvent être construites en composant d'autres coupes avec des opérateurs Argos. Par exemple, des coupes peuvent être mises en parallèles avec un automate qui prend leurs signaux de jointure en entrées et émet le signal de jointure de la coupe composée. De cette façon, des expressions comme "coupe A et non coupe B" ou "coupe A jusqu'à `a` et ensuite coupe B" s'expriment de manière modulaire.

3.2. *Spécification de l'insert*

Un insert exprime habituellement les modifications appliquées au programme de base. Dans notre environnement, nous considérons que le programme de base a été mis à plat d'abord, comme expliqué dans la section 2.2. En Larissa nous définissons deux types d'insert : un insert du premier type remplace les transitions de jointure par des *transitions d'insert* allant vers un état but existant ; un insert du second type introduit un programme complet entre l'état source de la transition de jointure et un état but existant. Dans tous les cas, l'état but doit être spécifié sans se référer explicitement aux noms d'états.

Pour spécifier l'état but, nous utilisons une trace finie sur les entrées, c'est-à-dire une suite dont chaque élément contient une valuation de toutes les entrées. Quand elle est exécutée à partir d'un état elle mène toujours à exactement un autre état, puisque l'automate est déterministe et complet. Nous considérons quatre façons de spécifier l'état but T parmi les états existants du programme de base P : 1) T est l'état de P qui est atteint en exécutant une trace depuis l'état initial de P, appelé un insert *toInit* ; 2) T est l'état depuis P qui est atteint en exécutant une trace de l'état source de la transition de jointure, appelé un insert *toCurrent* ; 3) T est l'état de P qui est atteint en exécutant une trace depuis l'état cible de la transition de jointure, appelé un insert *toTarget* ; 4) nous définissons d'abord quelques états de sauvegarde entre les états de P ; T est alors



(a) un aspect toInit (b) un aspect toCurrent

Figure 3. *Schéma pour les aspects toInit et toCurrent. Les transition de jointure (pointillées) sont remplacées par les transitions d'insert (en gras), qui mènent à l'état but T désigné par la trace* σ.

l'état de sauvegarde qui a été rencontré le plus récemment. Ce quatrième type ne sera pas utilisé dans cet article (*cf.* (Altisen *et al.*, 2006a) pour plus de détails).

3.2.1. *Les transitions d'insert*

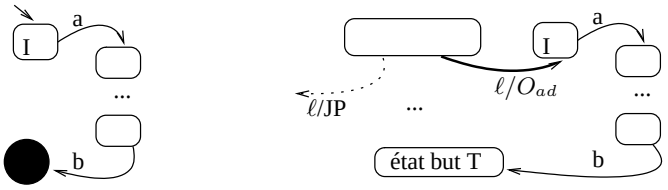
Le premier type d'insert remplace chaque transition de jointure par une transition d'insert. Une fois l'état but défini par une trace finie $\sigma = \sigma_1 \dots \sigma_n$, il reste seulement l'étiquette de la transition à déterminer. Nous ne changeons pas les entrées de l'étiquette pour que l'automate reste déterministe et complet, mais nous remplaçons les sorties par des *sorties d'insert* O_{ins}. Ce seront les mêmes pour toutes les transitions d'insert, et elles seront donc spécifiées dans l'insert. Les transitions d'insert sont illustrées dans la figure 3.

3.2.2. *Les programmes d'insert*

Il ne suffit pas toujours d'uniquement modifier des transitions seules, donc de juste sauter à un autre état de l'automate d'un seul pas, mais il peut être nécessaire d'exécuter du code arbitraire quand un aspect est activé. Dans ces cas, nous insérons un automate entre l'état source de la transition de jointure et l'état but.

Pour cela, nous utilisons un *automate inséré* qui *termine*. Comme Argos n'a pas de notion explicite de terminaison, le programmeur doit identifier un état final F (noté avec un cercle noir dans les figures).

Insérer un automate est similaire à insérer une transition. Nous spécifions d'abord un état but T avec une trace finie. Ensuite, pour chaque T, nous insérons une copie de l'automate A_{ins}, ce qui se traduit en 1) remplacer chaque transition de jointure avec état but T par une transition menant à l'état initial I de l'instance de A_{ins} correspondante à T. Comme pour les transitions d'insert, les entrées de l'étiquette restent inchangées et les sorties sont remplacées par les sorties d'insert ; 2) connecter toutes les transitions de A_{ins} menant à F à T. Les programmes d'insert sont illustrés dans la figure 4.



(a) automate inséré A_{ins} (b) programme tissé

Figure 4. *L'insertion d'un programme d'insert. L'automate inséré A_{ins} est inséré après la transition d'insert, et les transitions qui mènent à l'état final dans A_{ins} mènent à l'état but dans le programme tissé.*

3.3. *Spécification complète d'un aspect*

Un aspect est donné par la spécification de sa coupe et de son insert : $asp = (Coupe, Insert)$. *Coupe* est un programme Argos avec une seule sortie `JP`. *Insert* est un tuple qui contient 1) les sorties d'aspect O_{ins} ; 2) Le *type* de l'insert (*toInit*, *toCurrent* ou *toTarget*) ; 3) la trace finie sur les entrées σ ; et facultativement 4) P_{ins}, le programme d'insert. En résumé, quand on ajoute des transitions d'insert, $Insert = < O_{\text{ins}}, type, \sigma >$, et quand on ajoute des programmes d'insert $Insert = < O_{\text{ins}}, type, \sigma, P_{ins} >$, avec $type \in \{toCurrent, toInit, toTarget\}$.

3.4. *Définitions formelles et implémentation*

Dans (Altisen *et al.*, 2006a), nous définissons Larissa formellement, et démontrons les propriétés centrales : le tissage préserve l'équivalence comportementale habituelle, et aussi le déterminisme et la complétude du programme de base.

Notons $P \triangleleft asp$ le résultat du tissage d'un aspect asp dans un programme P. La préservation de l'équivalence, notée $\sim$, exprime que si $P \sim Q$, alors, pour tout aspect asp, $(P \triangleleft asp) \sim (Q \triangleleft asp)$. Avec cette propriété, le tissage d'aspect peut être considéré comme un véritable opérateur. Ce nouvel opérateur peut être utilisé librement dans des expressions comme par exemple $(P \| (Q \triangleleft asp_1)) \triangleleft asp_2$. Ensuite, chaque composant peut être remplacé par un composant équivalent sans changer le comportement du programme.

Un compilateur (Stauch, n.d.b) pour Larissa a été développé comme extension d'un compilateur Argos existant. Cet outil est connecté à des outils de simulation, test, et vérification formelle comme Lesar (Halbwachs *et al.*, 1992).

4. Un exemple : un contrôleur de porte de tramway

Nous implantons et vérifions un exemple plus large, un contrôleur de porte d'un

Les entrées du contrôleur :

enStation	Le tram est dans une station
attDep	Le tram veut partir
porteOuv	La porte est ouverte
porteFermé	La porte est fermée
demPorte	Un passager veut quitter le tram
timer	Le timer déclenche

Les sorties du contrôleur :

porteOK	La porte fermée et prêt à partir
ouvrirPorte	ouvre la porte
fermerPorte	ferme la porte
beep	émet un son
setTimer	démarre un timer

Les entrées de la passerelle :

passSort	La passerelle est sortie
passEnt	La passerelle est rentrée
demPass	Un passager veut utiliser la passerelle

Les sorties de la passerelle :

sortirPass	Sort la passerelle
rentrerPass	Rentre la passerelle

Les signaux auxiliaires :

accepterDem	Le passager peut demander la porte ou la passerelle
porteDem	Le passager a demandé l'ouverture de la porte
passDem	Le passager a demandé la passerelle
depImm	Le tram veut quitter la station

Figure 5. *Les interfaces du contrôleur et de la passerelle, et les signaux auxiliaires.*

tramway. Cette exemple a été pris du tutoriel Lustre (Raymond, n.d.). Le contrôleur de porte doit ouvrir la porte quand un passager veut quitter le tram, et la fermer avant que le tram parte. Certaines portes disposent aussi d'une passerelle, qui peut être étendue pour que les passagers dans des fauteuils roulants ou avec des poussettes puissent utiliser le tram sans aide. Un mauvais fonctionnement du contrôleur peut avoir des conséquences graves, menant jusqu'à la mort des usagers du tram. Pour éviter des situations dangereuses, le contrôleur doit assurer que :

1. la porte n'est jamais ouverte en dehors d'une station,
2. la passerelle n'est jamais sortie en dehors d'une station, et
3. la passerelle ne bouge pas si la porte n'est pas fermée.

Nous implantons le contrôleur en Argos. D'abord, nous développons un contrôleur de porte sans passerelle, et ensuite nous ajoutons la partie relative à la passerelle avec des aspects.

Le contrôleur interagit avec trois éléments dans son environnement : la porte, l'usager du tram et le conducteur du tram. De ces éléments, il reçoit différentes entrées, et il émet différentes sorties vers eux. Ces entrées et sorties sont données dans la figure 5, ainsi que celles qui sont ajoutées par la passerelle. En plus, le contrôleur utilise des entrées additionnelles, appelées *signaux auxiliaires*. Ils sont aussi donnés dans la figure 5, y inclut le signal passDem, qui sera utilisé par les aspects qui ajoutent la passerelle. Ils sont calculés à partir des entrées régulières par le programme montré dans la figure 6. Le signal dep sert seulement à calculer les autres signaux auxiliaires, et est encapsulé.

La figure 7 montre un contrôleur de porte, qui ne prend pas en compte la passerelle. L'état `Dehors` signifie que le tram n'est pas dans une station. Quand il entre dans une station, le contrôleur passe à l'état `Fermée`, qui signifie que le tram est en station et la porte est fermée. Si le contrôleur y reçoit *depImm*, il va dans l'état `OK`, émet *porteOK*, et attend que le tram quitte la station. Si, dans l'état `Fermée`, le contrôleur

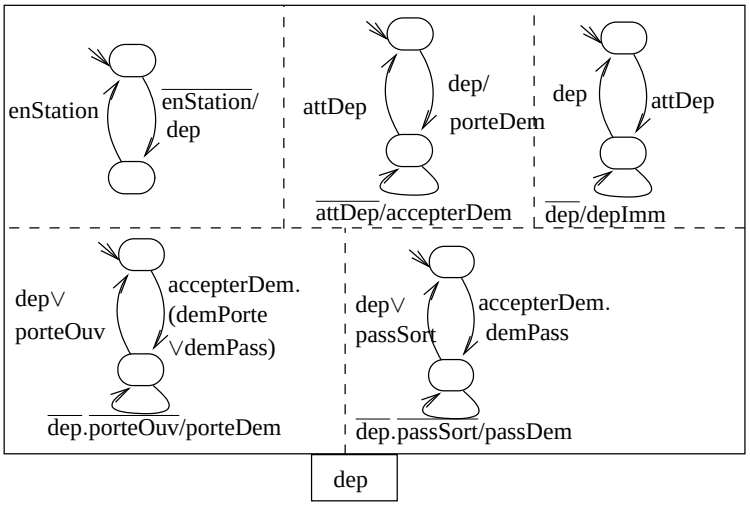


Figure 6. *L'automate qui produit les signaux auxiliaires.*

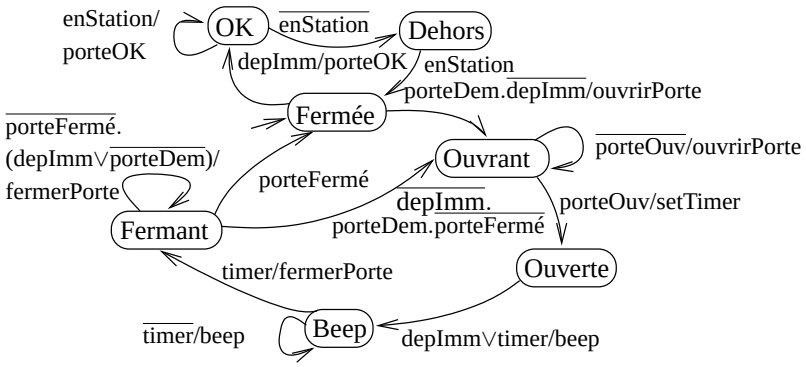


Figure 7. *Un contrôleur de porte du tramway.*

reçoit *porteDem*, il commence à ouvrir la porte : il va dans l'état `Ouvrant` et émet *ouvrirPorte* jusqu'à ce que la porte soit ouverte, et passe ensuite dans l'état `Ouverte`. Après un certain temps, ou quand le tram veut quitter la station, le contrôleur émet un signal sonore (état `Beep`) et commence à fermer la porte (état `Fermant`). Quand la porte est fermée, il retourne dans l'état `Fermée`.

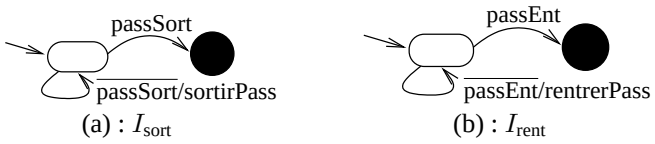


Figure 8. *Les automates insérés pour l'aspect de l'ouverture (a) et pour l'aspect de la fermeture (b).*

4.1. *L'ajout d'une passerelle*

Nous utilisons deux aspects pour ajouter la partie de contrôle relative à la passerelle au contrôleur. Le premier aspect, appelé l'aspect de l'ouverture, sort la passerelle quand un usager l'a demandée, et le deuxième aspect, appelé l'aspect de la fermeture, la rentre quand le tram veut quitter la station.

La coupe de l'aspect de l'ouverture, PC_{sort}, consiste d'un seul état avec une transition qui émet *JP* si $\mathit{ouvrirPorte}.\mathit{passDem}.\mathit{porteFermé}.\overline{\mathit{passSort}}$ est vrai. Il intervient dans le contrôleur de porte quand celui-ci veut ouvrir la porte, qu'un passager a demandé la passerelle, que la porte est encore fermée et que la passerelle n'est pas encore sortie. L'aspect respecte la propriété (2) (*cf.* page 9) en sortant la passerelle seulement quand il est possible d'ouvrir la porte, et la propriété (3) en la sortant seulement si la porte est encore fermée. L'insert de l'aspect de l'ouverture insère un automate I_{sort} qui émet *sortirPass* jusqu'à ce que la passerelle soit sortie, et va ensuite dans l'état où menait la transition de jointure. L'automate inséré est montré dans la figure 8(a), et l'aspect complet est spécifié par $(\text{PC}_{\text{sort}}, < \{\mathit{sortirPass}\}, \mathit{toTarget}, (), I_{\text{sort}} >)$, où $()$ dénote la trace vide. Au lieu d'utiliser un insert toTarget et une trace vide, on peut aussi utiliser un insert toInit avec une trace de longueur un et la valuation $\mathit{porteDem}.\overline{\mathit{depImm}}.\overline{\mathit{porteOuv}}.\mathit{enStation}.\mathit{porteFermé}$. Dans le contrôleur donné dans la figure 7, l'aspect aura le même effet, l'insertion de l'automate I_{sort} devant l'état `Ouvrant`.

La coupe de l'aspect de fermeture, PC_{rent}, sélectionne toutes les transitions où $\mathit{porteOK}!.\overline{\mathit{passEnt}}$ est vrai, et son insert insère un automate I_{rent} qui émet *rentrerPass* jusqu'à ce que la passerelle soit rentrée. Il s'insère donc toujours quand le contrôleur veut émettre *porteOK* et que la passerelle n'est pas rentrée. Il respecte donc la propriété (2) en assurant que la passerelle est toujours rentrée avant que *porteOK* soit émis. La propriété (3) est aussi respectée, puisque le contrôleur peut seulement émettre *rentrerPass* si la porte est déjà fermée. L'aspect est complètement spécifié par $(\text{PC}_{\text{ret}}, < \{\mathit{rentrerPass}\}, \mathit{toTarget}, (), I_{\text{rent}} >)$.

L'aspect de fermeture rentre la passerelle seulement quand le tram veut partir, même si la porte s'est fermée auparavant. On peut s'imaginer une version où on préfère rentrer la passerelle dès que la porte est fermée. Pour cela, il suffit de remplacer la coupe de l'aspect de fermeture par l'automate de la figure 9 : on choisit alors toutes les transitions où on vient de s'arrêter d'émettre *fermerPorte*, la porte est fermée et

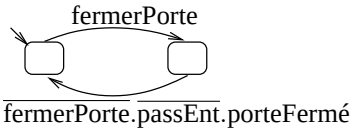


Figure 9. *Une autre version possible de la coupe de l'aspect de fermeture.*

la passerelle encore sortie. Cet aspect respecte la propriété (2) parce qu'il rentre la passerelle toujours après la fermeture de la porte, donc avant que le contrôleur puisse émettre *porteOK*, et aussi la propriété (3), parce qu'il rentre la passerelle seulement si la porte est fermée. Cette version montre l'utilisation d'une coupe temporelle, qui prend en compte les signaux émis auparavant. Ces coupes s'expriment très bien en Larissa, parce que les coupes sont des automates.

4.2. *Vérification formelle*

Étant donné l'importance des propriétés de sécurité, nous voulons démontrer formellement que le contrôleur ne les viole pas, après avoir tissé les aspects. Alors que le contrôleur de porte ne peut pas garantir ces propriétés tout seul, nous voulons être sûr que si l'environnement du contrôleur se comporte comme prévu (la porte ne s'ouvre pas toute seule, le tram ne part pas de la station si le contrôleur n'a pas émis *porteOK*,...), elles sont toujours vraies.

Pour cela, nous modélisons l'environnement du contrôleur et les propriétés comme des ensembles de traces sur les entrées et les sorties du programme. Le modèle E de l'environnement du contrôleur contient toutes les traces qui peuvent être produites par l'environnement, et le modèle P des propriétés de sécurité contient toutes les traces qui vérifient les propriétés. Nous démontrons que les traces produites par le contrôleur tissé qui sont dans E sont aussi dans P.

Le modèle de l'environnement s'exprime avec des automates Argos avec un état spécial `Erreur`. Les traces possibles de E sont alors celles qui ne mènent pas à l'état `Erreur`. La figure 10 montre le modèle du tram ; il exige principalement que le tram ne quitte pas la station avant avoir reçu *porteOK*. La figure 11 montre le modèle de la porte ; il exige que la porte s'ouvre seulement si on a émis *ouvrirPorte*, et se ferme seulement si on a émis *fermerPorte*. Le modèle des propriétés de sécurité s'exprime facilement de la même manière, la première propriété par exemple avec un automate qui passe à l'état `Erreur` si *porteOuv*.$\overline{enStation}$ est vrai.

Pour prouver que les propriétés de sécurité sont respectées, nous avons combiné le contrôleur avec l'aspect d'ouverture et un des deux aspects de fermeture avec notre implémentation. Ensuite, nous avons combiné le contrôleur tissé avec le modèle de l'environnement et des propriétés de sécurité, et nous avons vérifié qu'on peut seulement arriver dans un état `Erreur` des propriétés de sécurité si on est dans l'état `Erreur`

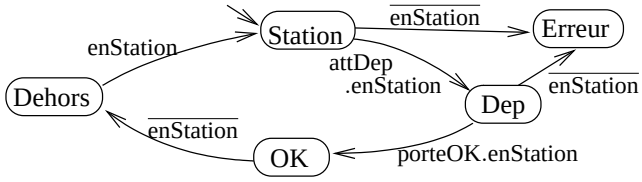


Figure 10. *Le modèle du tramway.*

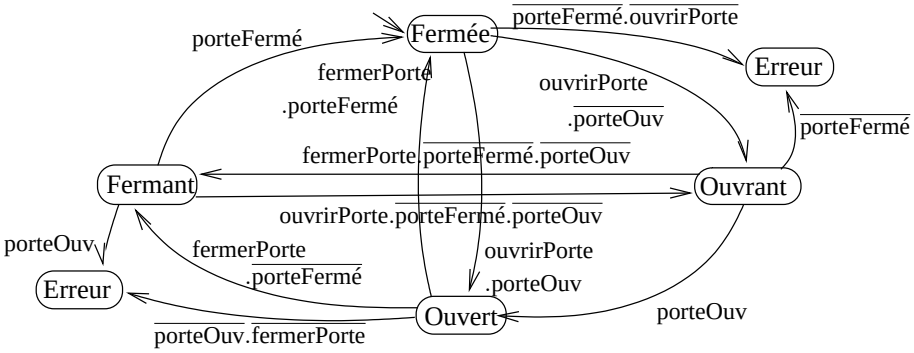


Figure 11. *Le modèle de la porte.*

du modèle de l'environnement. Le code source du contrôleur et du modèle de l'environnement peut être trouvé à (Stauch, n.d.a).

5. Travaux connexes

Plusieurs approches existent pour modéliser des aspects dans des automates. Ils se concentrent surtout sur la modélisation des langages d'aspects existants dans des machines d'états d'UML. Des approches autour de la modélisation d'aspects, comme Theme/UML (Clarke *et al.*, 2005), servent surtout à la spécification et la documentation des aspects, et ne peuvent pas être tissées automatiquement. Dans notre approche, les automates sont entièrement compilés, les aspects Larissa ne décrivent donc pas un langage d'aspects d'un niveau plus bas. Au lieu de modéliser un langage existant, nous avons plutôt essayé d'obtenir des propriétés sémantiques fortes.

(Klein *et al.*, 2006) propose un langage d'aspects pour des High-level Message Sequence Charts (HMSC), un langage de spécification de scénario. Les HLMC sont des automates avec des transitions qui comporte une suite de communication entre des instances. Les aspects proposés choisissent une partie d'une trace d'exécution, et la remplace avec une autre. Comme Larissa, ils se réfèrent à la sémantique du programme. Par contre, ce type d'aspect ne peut pas être implémenté dans un langage de

programmation, parce qu'on ne connaît pas à l'exécution la suite de la trace, et ne peut donc pas savoir si il faut la remplacer.

Il existe de nombreuses approches pour formaliser la notion d'aspect. (Douence *et al.,* 2005) explique la plupart des formalismes existants. Certains auteurs ont utilisé des cadres formels pour montrer des propriétés intéressantes de leur langage d'aspects. Par exemple, (Dantas *et al.,* 2006) réduit le pouvoir de leurs aspects considérablement, et (Douence *et al.,* 2004) examine l'interaction entre plusieurs aspects.

6. Conclusion

Dans cet article, nous avons décrit le langage d'aspects Larissa et un exemple, où nous avons ajouté une fonctionnalité à un contrôleur de porte de tramway avec des aspects. Nous pouvions ajouter cette fonctionnalité seulement en regardant l'interface du programme de base, sans regarder les détails de l'implémentation. C'est la première fois que nous traitons un exemple d'un système réactif critique avec Larissa.

L'exemple est différent des exemples AspectJ les plus communs dans le sens où l'aspect implémente une préoccupation fonctionnelle du programme, alors que les aspects sont souvent utilisés dans des intergiciels pour implémenter des préoccupations non fonctionnelles, telles que la journalisation, la sécurité ou la gestion des transactions. Dans ce sens, nous sommes plus proches des approches qui utilisent la programmation par aspect pour réaliser des lignes de produits, comme par exemple (Lopez-Herrejon *et al.,* 2005). Dans les autres exemples que nous avons traités (Altisen *et al.,* 2006a, Altisen *et al.,* 2006b), les aspects ajoutent aussi une préoccupation fonctionnelle au programme. Une raison pour cela est qu'un programme Argos décrit seulement le noyau réactif du programme, des préoccupations non fonctionnelles sont souvent traitées par l'environnement. En plus, Larissa semble bien adapté pour une programmation incrémentale, parce que la préservation de l'équivalence permet de modifier l'implémentation du programme de base sans avoir besoin de changer l'aspect.

Nous pensons que des préoccupations non fonctionnelles existent aussi dans le domaine des systèmes réactifs. Une perspective intéressante de Larissa dans cette direction concerne la construction de modèles non fonctionnels à partir de modèles fonctionnels, dans les cas où cela demande d'enrichir les modèles fonctionnels avec des nouveaux comportements. Plus précisément : dans de nombreux domaines de la conception et modélisation des systèmes embarqués, on écrit d'abord des modèles fonctionnels très abstraits pour réfléchir aux fonctionnalités du système, et ensuite seulement on les enrichit avec des détails (par exemple sur la plate-forme d'exécution) pour essayer d'en tirer aussi des estimations de performance temporelle. Or il apparaît souvent que si l'on veut obtenir des évaluations réalistes des performances temporelles, les détails à ajouter sont en fait des détails de comportement, et pas seulement un ensemble de durées associées à des opérations de base dans le modèle fonctionnel. Par exemple, dans un système sur puce, il faut bien connaître le comportement

du bus si l'on veut prédire de manière réaliste le temps que prennent des échanges entre plusieurs composants connectés au même bus. Le modèle non fonctionnel inclut donc les aspects fonctionnels abstraits, plus les détails de comportement de certains composants cruciaux comme les bus. Pour fabriquer ce modèle non fonctionnel, on doit ajouter du comportement au modèle fonctionnel. Nous commençons à travailler sur l'idée de décrire les modèles en Argos, et la démarche d'enrichissement des modèles fonctionnels avec Larissa. Cela devrait permettre de bien séparer la description des fonctions et l'ajout de détails, comme cela a déjà été montré dans le cas de la programmation, par exemple pour le tramway.

7. Bibliographie

Altisen K., Maraninchi F., Stauch D., « 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*, 2006a.

Altisen K., Maraninchi F., Stauch D., « Larissa : Modular Design of Man-Machine Interfaces with Aspects », *5th International Symposium on Software Composition*, vol. 4089 of *LNCS*, Vienna, Austria, March, 2006b.

Clarke S., Baniassad E., *Aspect-Oriented Analysis and Design : The Theme Approach*, Addison-Wesley, 2005.

Dantas D., Walker D., « Harmless Advice », *Proceedings of the 33th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL-06)",*, vol. 41, 1, p. 383-396, January, 2006.

Douence R., Fradet P., Südholt M., « Composition, Reuse and Interaction Analysis of Stateful Aspects », *Proceedings of the 3nd international conference on Aspect-oriented software development*, ACM Press, 2004.

Douence R., Le Botlan D., Towards a Taxonomy of AOP semantics : Milestone m8.1 for AOSD NOE, Technical Report n° M8.1, INRIA, 2005.

Halbwachs N., *Synchronous programming of reactive systems*, Kluwer Academic Pub., 1993.

Halbwachs N., Lagnier F., Ratel C., « Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE », *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, September, 1992.

Halbwachs N., Lagnier F., Raymond P., « Synchronous observers and the verification of reactive systems », *in* , M. Nivat, , C. Rattray, , T. Rus, , G. Scollo (eds), *Algebraic Methodology and Software Technology, AMAST'93*, June, 1993.

Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W. G., « An Overview of AspectJ », *LNCS*, vol. 2072, p. 327-353, 2001.

Klein J., Hélouet L., Jézéquel J.-M., « Semantic-based Weaving of Scenarios », *proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06)*, ACM, Bonn, Germany, march, 2006.

Lamport L., « Proving the correctness of multiprocess programs », *ACM Transactions on Programming Languages and Systems*, vol. SE-3, n° 2, p. 125-143, 1977.

Lopez-Herrejon R. E., Batory D., « Improving Incremental Development in AspectJ by Bounding Quantification », *in* , L. Bergmans, , K. Gybels, , P. Tarr, , E. Ernst (eds), *Software Engineering Properties of Languages and Aspect Technologies*, March, 2005.

Maraninchi F., Rémond Y., « Argos : an automaton-based synchronous language. », *Computer Languages*, vol. 27, n° 1/3, p. 61-92, 2001.

Raymond P., « Le tutoriel Lustre. », n.d. http ://www-verimag.imag.fr/∼raymond/edu/ tp.ps.gz.

Stauch D., « Code source de l'exemple du tram. », n.d.a. http ://www-verimag.imag.fr/∼stauch/ArgosCompiler/contracts.html.

Stauch D., « Le compilateur Larissa. », n.d.b. http ://www-verimag.imag.fr/∼stauch/Argos Compiler/.