
Modélisation et vérification formelle de la composition des aspects

Farida Mostefaoui — Julie Vachon

*DIRO, Université de Montréal
C.P. 6128, Succ. Centre-Ville, Montréal, Québec, H3C 3J7, Canada
{mostefaf, vachon}@iro.umontreal.ca*

RÉSUMÉ. Résumé : La programmation orientée aspect s'avère particulièrement bien adaptée aux processus de développement incrémentaux qui nécessitent la prise en compte de nouvelles préoccupations transverses et l'ajout de fonctionnalités auxiliaires. Pour satisfaire aux besoins de ces processus, nous proposons une méthodologie de modélisation et de vérification des systèmes par aspects. La vérification s'applique ici aux modèles que nous développons avec Aspect-UML, notre profil pour la modélisation des aspects. Notre approche de vérification a deux objectifs. Le premier consiste à assurer que l'introduction d'un nouvel aspect ne compromet pas la correction du système initial (et ce, sans révérifier le système en entier). Le second objectif vise à détecter les problèmes liés à la composition d'aspects dont l'exécution peut s'avérer conflictuelle. Cet article présente la façon de modéliser les aspects en utilisant Aspect-UML. Nous expliquons ensuite comment exprimer formellement ces modèles en réseaux de Petri colorés afin qu'ils puissent être vérifiés par model-checking.

ABSTRACT.

MOTS-CLÉS : composition d'aspects, vérification formelle, modélisation uml, réseaux de Petri colorés, model-checking

KEYWORDS: aspect composition, formal verification, uml modelisation, colored petri nets. model-checking

1. Introduction

La programmation orientée aspect (Kiczales *et al.*, 1997) s'avère particulièrement bien adaptée aux processus de développement incrémentaux qui nécessitent la prise en compte de nouvelles préoccupations transverses et l'ajout de fonctionnalités auxiliaires. Pour satisfaire aux besoins de ces processus, nous proposons une méthodologie de modélisation et de vérification basée sur le paradigme aspect. S'appuyant sur les méthodes formelles et un certain nombre d'outils, cette méthodologie propose une phase de modélisation à *la UML* et une phase de vérification assurant la *bonne composition* des aspects. Cette vérification procède par model-checking et doit assurer que le système respecte les propriétés de *compositionnalité* qui peuvent conditionner sa sûreté et sa vivacité. S'appliquant aux modèles, la vérification vise plus précisément les deux objectifs suivants. Le premier consiste à assurer que l'introduction d'un nouvel aspect ne compromet pas la correction du système initial (et ce, sans révérifier le système en entier). Le second objectif vise à détecter les problèmes liés à la composition d'aspects dont l'exécution peut s'avérer conflictuelle.

La vérification s'applique aux modèles que nous développons avec Aspect-UML, le profil que nous introduisons pour la modélisation des aspects (section 2). Ces modèles Aspect-UML sont le produit des phases d'analyse et de conception du développement ou, éventuellement, le résultat d'un travail de rétro-ingénierie d'un programme par aspects. Nous avons privilégié une approche de modélisation de haut niveau à *la UML* qui n'exige pas de connaissances approfondies des méthodes formelles de la part du développeur. En fait, il est possible de procéder à la vérification de la composition des aspects en considérant l'ajout d'un certain nombre d'annotations formelles bien circonscrites. En effet, le paradigme aspect a l'avantage d'améliorer l'implémentation modulaire de fonctionnalités transverses (grâce aux aspects) mais également d'exposer clairement les contextes dans lequel ces dernières doivent s'insérer (grâce aux points de jointure). La vérification peut profiter de cette information sur le contexte et ainsi restreindre sa couverture à l'aspect introduit et à ses points de jointures. On évite ainsi d'avoir à vérifier un programme en entier après chaque incrément.

Pour procéder à la vérification formelle du système par model-checking et analyse, il importe de pouvoir donner une représentation formelle des modèles Aspect-UML. Dans un premier temps, le développeur est invité à documenter, au moyen d'annotations formelles de type OCL¹, les contextes où se composent et se tissent les aspects dans le système de base. Ces contextes sont clairement mis en évidence par les points de coupure et de jointure des modèles Aspect-UML. Nous pouvons alors extraire, du modèle Aspect-UML, un modèle formel exprimé en réseaux de Petri colorés (rdPc). Les rdPc proposent un modèle opérationnel particulièrement intéressant pour la spécification, la simulation et l'analyse des systèmes concurrents et, en l'occurrence, des aspects qui s'exécutent au même point de jointure. Nous présentons, à la section 3 la construction des rdPc décrivant formellement les concepts aspects d'Aspect-UML. Tel qu'expliqué à la section 4, nous pourrions profiter des techniques d'analyse (telle la

1. Object Constraint Language

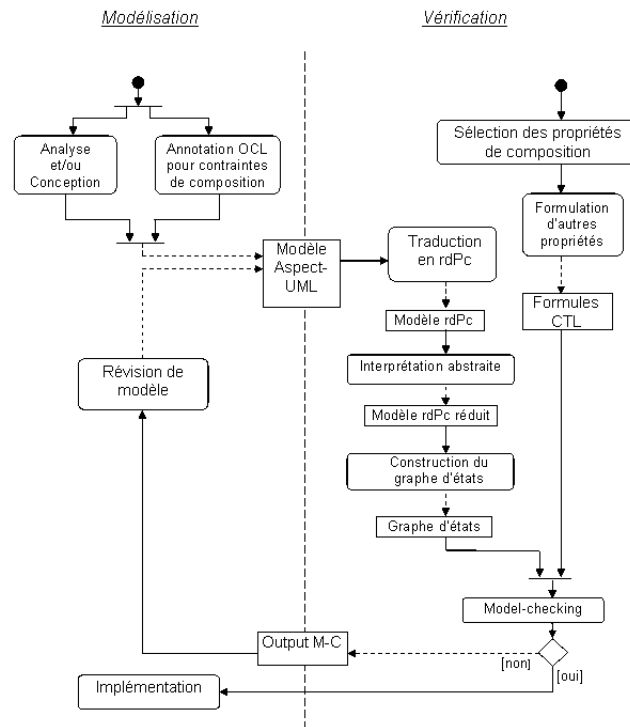


Figure 1. Méthodologie de modélisation et de vérification des systèmes par aspects

détection de blocage) déjà développées pour les rdPc et nous pourrions nous référer au graphe de marquages pour le soumettre à un outil de model-checking qui en vérifiera les propriétés souhaitées.

La figure 1 résume notre méthodologie de modélisation et de vérification des systèmes par aspects et explique la coordination des différentes activités qui la composent.

Pour illustrer l'utilisation de notre méthodologie, nous ferons référence à un exemple, tiré de (AspectJ Team, 2002), qui présente une application simplifiée de simulation de télécommunication. Cette application permet de simuler un système de téléphonie dans lequel des usagers peuvent (1) initier, (2) recevoir, (3) rompre et (4) fusionner des communications téléphoniques. À ce système de base on souhaite greffer des fonctionnalités additionnelles pour le calcul de la durée des communications et leur facturation aux clients. À l'aide d'Aspect-UML, nous montrons, dans la section suivante, comment étendre le modèle UML de l'application de base afin d'intégrer ces deux nouvelles fonctionnalités.

2. Modélisation Aspect-UML

Aspect-UML (Vachon *et al.*, 2004a) est un profil UML permettant de décrire les principaux concepts du paradigme aspects. En intégrant la notion d'aspect non seulement dans les diagrammes de classes mais également dans les diagrammes de cas d'utilisation, Aspect-UML permet de prendre en compte les fonctionnalités de nature transversale dès les premières étapes de la spécification des besoins. Notre méthodologie est de nature itérative et procède donc par raffinements successifs des modèles. En plus d'introduire de nouveaux stéréotypes, Aspect-UML propose d'enrichir les modèles par des contraintes (dont certaines formulées en OCL) permettant de spécifier formellement les fragments du modèle qui sont pertinents pour la vérification de la composition des aspects.

2.1. Diagramme de cas d'utilisation

Les aspects peuvent représenter des fonctionnalités auxiliaires, des services secondaires ou des contraintes de qualité et d'utilisation du système. En général, ces traitements ne sont pas représentés dans la vue de cas d'utilisation standard, qui est souvent réservée pour décrire les services de base et les exigences fonctionnelles principales du système. Dans notre approche (Vachon *et al.*, 2004a), nous proposons de considérer les aspects dès la phase d'acquisition des besoins, et de les prendre en charge durant tout le cycle de développement. Pour cela, nous proposons de considérer les aspects comme des cas d'utilisations d'extension. Les points de coupure sont alors assimilés aux points d'extension, i.e. un ensemble de locations où l'on peut tisser les cas d'utilisation qui représentent un aspect. Dans notre exemple de l'application de Telecom, nous avons les fonctionnalités de base *initier un appel*, *rompre un appel* et *fusionner des appels*, représentées respectivement par les cas d'utilisation de base *complete*, *drop* et *merge*. L'intégration à l'application Telecom des deux fonctionnalités auxiliaires *calcul de la durée d'une communication* et *facturation des communications pour un client*, se fait grâce à la notion d'aspects. On introduit deux aspects *Timing* et *Billing*. L'aspect *Timing* intervient au début et à la fin de chaque communication. Par contre, l'aspect *Billing* doit être réalisé seulement à la fin de chaque communication. Par conséquent, ces deux aspects entrecoupent les deux fonctionnalités de base que sont *initier un appel* et *rompre un appel*. Dans la vue de cas d'utilisation proposée à la figure 2, les deux aspects *Timing* et *Billing* sont considérés comme des cas d'utilisation d'extension. Pour modéliser le comportement transverse de ces deux aspects et leur insertion dans les cas d'utilisation de base *complete* et *drop*, la relation « extend » de UML est utilisée. Les cas d'utilisation de base seront donc modifiés implicitement aux points d'extension indiqués *OpComplete* et *OpDrop*.

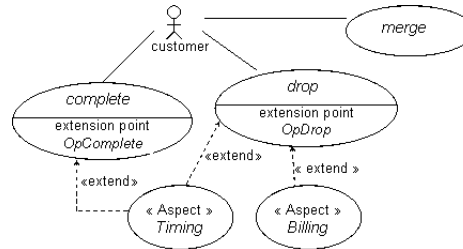


Figure 2. Diagramme de cas d'utilisation pour l'application Telecom

2.2. Diagramme de classes

Considérons les cas d'utilisation *Timing* et *Billing* de notre exemple. La figure 3 montre comment les exigences de ces fonctionnalités sont intégrées dans le diagramme de classes UML. Les comportements des cas d'utilisation *Timing* et *Billing* sont décrits par des classes nommées respectivement *Timing* et *Billing* portant le stéréotype « Aspect ». Quant aux points d'extension *OpComplete* et *OpDrop*, ils correspondent à des points de coupure et sont donc modélisés par des interfaces portant le stéréotype « Pointcut ». Un point de coupure est lié à un ensemble de points de jointure. Une relation de dépendance « crosscuts » relie donc chaque interface « Pointcut » à ses points de jointure. Les interfaces *OpComplete* et *OpDrop* ont chacune une relation « crosscuts » pointant respectivement vers les points de jointure correspondant à la méthode *complete()* et à la méthode *drop()* de la classe *connection*.

Les interfaces *OpComplete* et *OpDrop* contiennent chacune une opération abstraite nommée respectivement *opComplete* et *opDrop* qui doit être exécutée quand un de leurs points de jointure est atteint. L'aspect *Timing* implémente les deux interfaces *OpComplete* et *OpDrop* et propose donc deux méthodes, appelées advices, qui seront exécutées aux points de jointure spécifiés. À remarquer qu'un advice est annoté avec l'un des stéréotypes « before », « after » ou « around » selon qu'il est exécuté respectivement avant, après ou autour des points de jointure référencés par le point de coupure.

2.3. Contraintes formelles

La vérification que nous comptons effectuer ne concerne que la composition des aspects et non la correction du modèle en entier. Il nous faut donc déclarer (à défaut de les vérifier) les propriétés caractérisant le contexte et les contraintes de composition des aspects. En l'occurrence, le développeur doit enrichir le diagramme de classes avec des contraintes formelles dont certaines sont exprimées dans le langage OCL. Un modèle Aspect-UML requiert les quatre sortes de contraintes suivantes :

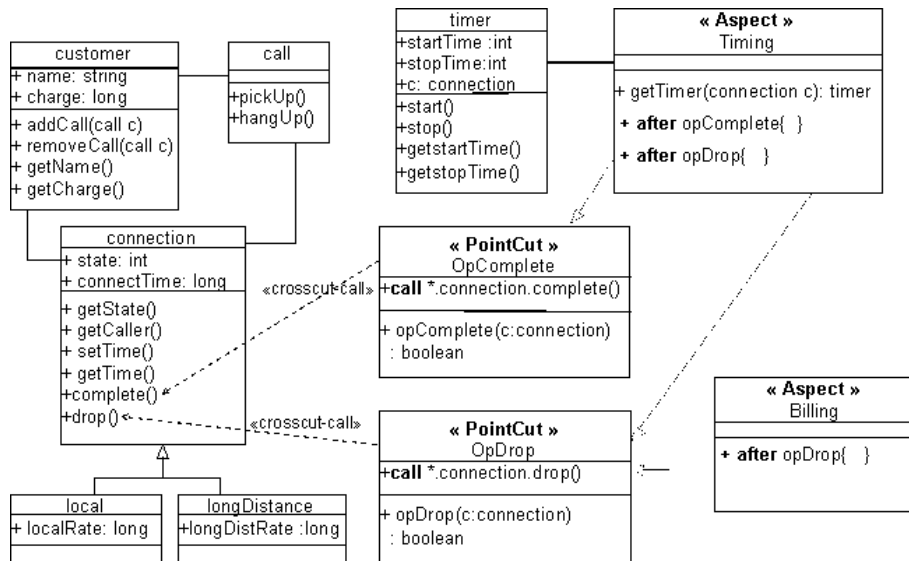


Figure 3. Diagramme de classes pour l'application Telecom

Contraintes de précedence. Il est possible que plusieurs aspects s'entrecroisent à un même point de jointure et que tous proposent un advice de la même sorte (`before` ou `after`). Sur un diagramme de classes Aspect-UML, on peut identifier les aspects conflictuels en repérant les advices de la même sorte qui implémentent une même interface « Pointcut ». C'est effectivement le cas des aspects *Billing* et *Timing* de l'application Telecom qui implémentent tous deux des advices « after » au point du coupure *OpDrop*. Il est toutefois possible de résoudre les conflits en définissant une relation de précedence entre les aspects. Pour ce faire, il suffit d'ajouter au diagramme de classes une contrainte globale de la forme $aspect_1 < aspect_2$ où $<$ est une relation d'ordre partiel. Dans le cas de l'application Telecom, puisqu'il importe de calculer la durée d'un appel pour pouvoir le facturer, la contrainte $Timing < Billing$ est ajoutée au diagramme de classes.

Spécification du passage du contexte d'un point de jointure à un advice. L'implémentation d'un advice peut nécessiter l'accès aux données d'un point de jointure. Le point de coupure permet d'exposer le *contexte* d'un point de jointure. Ce contexte contient notamment l'objet dont on invoque la méthode au point de jointure et les arguments de l'appel. Le passage du contexte est décrit au moyen d'une contrainte Aspect-UML spécifiant la façon dont les paramètres formels du point de coupure sont liés aux éléments du contexte.

```

<pointcut> :: Binding
  ToJointPoint : <jointpointname>
  Binds : <pcArg>_1 ← <context_elem>; ...; <pcArg>_n ← <context_elem>;

```

Dans cette contrainte, $\langle pcArg \rangle_1, \dots, \langle pcArg \rangle_n$ sont les paramètres du point de coupure et $context_elem$ est soit un paramètre du point de jointure, soit la variable $target$ qui retourne une référence sur l'objet appelé. Dans notre exemple, la contrainte suivante spécifie la passage du contexte du point de jointure $connection :: drop()$ au point de coupure $Opdrop$:

```
Opdrop : :Binding
ToJointPoint : connection : :drop()
Binds : c ← target
```

Spécification des points de jointure. Pour focaliser la vérification sur la correction de la composition des aspects, nous faisons l'hypothèse que le modèle initial est correct et que nous disposons de la spécification de certains fragments de ce modèle. Entre autre, il est attendu que le développeur fournisse, dans le modèle Aspect-UML, la spécification des méthodes constituant des points de jointure. Définie indépendamment du contexte d'exécution de la méthode, cette spécification est exprimée en OCL par des pré et post conditions et est notée de la façon suivante :

```
context nom_classe : :nom_méthode()
pre : condition1,...
post : condition2,...
inv : condition3,...
```

Le nom du contexte est celui de la méthode (préfixé par le nom de la classe à laquelle elle appartient), constituant le point de jointure. Les conditions sont des expressions booléennes respectant la syntaxe et la sémantique d'OCL². Une pré-condition exprime une propriété nécessaire à l'exécution de la méthode, alors qu'une post-condition exprime une propriété assurée suite à l'exécution de la méthode. Un invariant est une propriété qui est à la fois une pré et une post-condition. Pour l'application Telecom, la spécification du point de jointure $connection :: drop()$ est comme suit :

```
context connection : :drop()
pre : state = 1
post : state = 2
```

Spécification des advices. Pour vérifier la composition d'un aspect, nous devons connaître la spécification de ses advices indépendamment de leur éventuel contexte d'exécution. Nous pourrions alors vérifier que le tissage des advices n'entrave pas la correction du système initial. Pour l'instant, nous faisons l'hypothèse que le corps d'un advice respecte effectivement la spécification qui en est donnée³. Cette spécification est formulée en OCL de la même façon que pour les points de jointure. Cette fois, le

2. Entre autres, seules les opérations sans effets de bord sont autorisées.

3. Dans la prochaine étape de ce projet, nous envisageons toutefois d'étendre Aspect-UML à la modélisation du corps des advices au moyen de diagramme d'états UML. L'analyse de ces diagrammes permettra de déduire les pré-conditions (les plus faibles) et les post-conditions (les plus fortes) des advices au lieu de simplement les déclarer sans égard au corps de ces advices.

nom du contexte est cependant de la forme *nom_aspect* : *:nom_advice()*. L'exemple suivant décrit la spécification de l'advice *opDrop()* de l'aspect *Billing* de l'application Telecom.

```

context Billing : opDrop
  inv : connection=c
  inv : c.getTime() !=Null
  post : c.getCaller.getCharge() ≥ c.getCaller.getCharge()@pre

```

Nous pourrions assurer la sûreté du système après le tissage d'un advice en appliquant les règles de contra-variance et de co-variance. Ces règles stipulent respectivement que les pré-conditions de l'advice doivent être plus faibles que les pré-conditions du point de jointure et que les post-conditions de l'advice doivent être plus fortes que (i.e. doivent impliquer) les post-conditions du point de jointure. Si ces règles ne sont pas respectées, la vérification devra émettre un avertissement. S'il y a carrément contradiction entre les pré-conditions d'un advice et celles d'un de ses points de jointures, ou entre leurs post-conditions, la vérification devra signaler une erreur.

Les contraintes d'Aspect-UML peuvent être spécifiées directement sur le diagramme de classes au moyen d'une note portant le stéréotype « Aspect-UML Constraint » et qu'on rattache à l'élément contraint. On peut également regrouper les contraintes dans un fichier indépendant.

3. Sémantique du profil Aspect-UML en termes de réseaux de Petri colorés

La modélisation avec Aspect-UML demeure de haut niveau tout en étant rigoureuse. Son utilisation ne requiert toutefois pas une connaissance approfondie des méthodes formelles. Néanmoins, l'information fournie dans un modèle Aspect-UML est suffisante pour en extraire un modèle formel décrivant la composition des aspects aux points de jointure. Nous avons besoin de cette représentation formelle pour pouvoir vérifier la composition des aspects spécifiés dans le modèle Aspect-UML.

Nous proposons de décrire ce modèle formel à l'aide de réseaux de Petri colorés (rdPc) (Jensen, 1997). En plus de disposer de plusieurs outils de simulation et de techniques d'analyse automatique, les rdPc s'avèrent particulièrement bien adaptés à la description des systèmes distribués et concurrents. Nous avons donc choisi ce formalisme pour modéliser la composition des aspects conflictuels. Dans ce qui suit, nous considérons un type de rdPc similaire à celui supporté par l'outil Design/CPN (Kristensen *et al.*, 1998). Ces réseaux font appels au langage fonctionnel ML pour la spécification des données et sont dotés de constructions hiérarchiques facilitant la description de grands systèmes. Nous référons le lecteur à (Jensen, 1997, Peterson, 1981, Kristensen *et al.*, 1998) pour une présentation du formalisme des rdPc.

Il est à noter que la plupart des formalismes rdPc ne sont pas orientés objets. Entre autres, l'outil Design/CPN utilise le langage fonctionnel ML pour décrire et manipuler les données qui colorent les jetons. Pour pallier les différences de paradigmes entre Aspect-UML et Design/CPN, nous proposons de simuler les concepts orientés objets

Concept du paradigme objet	Concept du paradigme fonctionnel
Classe C	Définition d'un nouveau type record C .
Attribut A de la classe C	Champ A dans le type record C
Méthode $m(a_1 : T_1, a_n : T_n) : T_r$ de la classe C	Fonction $m : (C, T_1, \dots, T_n) \rightarrow T_r$
Identité d'un objet	Champ <i>Ident</i> dans tous les types record décrivant une classe; la valeur de ce champ est gérée comme une clef et permet d'identifier chaque objet de façon unique.
Héritage de la sous-classe B par la super-classe A (par extension)	Définition d'un type <i>tagged union</i> pour la sous-classe B. Le type comportera deux tags, l'un associé à un record type A, l'autre associé à un record type B (regroupant les attributs spécifiques à B).

Tableau 1. Simulation des concepts objets dans le paradigme fonctionnel

par des mécanismes fonctionnels. La table 1 résume l'approche que nous avons déjà adoptée dans (Vachon *et al.*, 2004b)

Les sections qui suivent présentent les étapes de construction du réseau de Petri coloré qui décrit la sémantique d'un modèle Aspect-UML. Cette sémantique ne couvre que les éléments du modèle pertinents à la vérification de la composition des aspects. Dans un premier temps, on construit pour chaque advice et chaque point de jointure un réseau de Petri le décrivant. Les rdPc des points de coupure sont ensuite élaborés et liés aux rdPc de leurs points de jointure. La dernière étape consiste à composer les rdPc des advices avec les rdPc des points de jointure en respectant les contraintes de précedence Aspect-UML éventuellement spécifiées.

3.1. Modélisation des advices et des points de jointure

Les advices, tout comme les points de jointure, sont des opérations devant respecter les pré et post conditions qu'on leur a attribuées dans le modèle Aspect-UML. La modélisation en rdPc d'un advice ou d'un point de jointure associé à une opération op , est notée et réalisée de la façon suivante :

Création des places et transitions. Soit $op(a_1 : t_1, \dots, a_n : t_n) : t_r$ la signature d'un advice (ou d'un point de jointure), l'ensemble des arguments de op est noté $Args^{op} = \{a_i : t_i \mid 1 \leq i \leq n\}$ et l'ensemble des attributs de l'aspect (ou de la classe) dans lequel il est déclaré est noté $Att^{op} = \{v_1 : t_1, \dots, v_m : t_m\}$. Nous appelons $Var^{op} = Args^{op} \cup Att^{op}$ l'ensemble des variables dans la portée de op . Le rdPc décrivant l'opération op est obtenu en créant (1) une transition T^{op} , (2) une place $P_{x_i}^{op}$ de type t_i pour chacune des variables $x_i \in Var^{op}$, $1 \leq i \leq n + m$ ainsi que (3) des places P_{in}^{op} et P_{out}^{op} de type *ObjectId* utilisée pour marquer l'invocation de l'opération d'un objet donné.

Création des arcs. Soient respectivement $preConds^{op}(x)$ et $postConds^{op}(x)$ l'ensemble de toutes les pré-conditions et post-conditions de op contenant la variable $x \in Var^{op}$. L'expression $eval(preConds^{op}(x), y)$ est une fonction qui, étant donné un paramètre y , calcule une valeur de x pouvant satisfaire les contraintes dans $preConds^{op}(x)$. De façon similaire, l'expression $eval(postConds^{op}(x), y)$ permet de générer une valeur satisfaisant $postConds^{op}(x)$.

Pour créer les arcs du rdPc correspondant à op , on applique les règles ci-dessous à chaque $x \in Var^{op}$:

- Si $preConds^{op}(x) \neq \{\}$, on crée un arc reliant la place P_x à la transition T^{op} en lui adjoignant l'expression $eval(preConds^{op}(x), y)$.
- Si $postConds^{op}(x) = \emptyset$, on doit créer un arc de T^{op} à P_x , aussi étiqueté par $eval(preConds^{op}(x), y)$ pour remettre dans sa place initiale un jeton coloré qui n'a été utilisé que pour consultation.
- Si $postConds^{op}(x) \neq \{\}$, on crée un arc de la transition T^{op} à P_x avec l'étiquette $eval(postConds^{op}(x), y)$.
- Pour représenter le flot d'exécution traversant op , on doit finalement ajouter un arc de la place P_{in}^{op} à T^{op} , et un autre de T^{op} à P_{out}^{op} . Ces arcs sont identifiés par l'identité de l'objet invoqué par cette opération.

On note $rdPc(opName)$ le réseau de Petri coloré modélisant un advice ou un point de jointure nommé $opName$. La partie A de la figure 4 montre le rdPc construit pour représenter un point de jointure correspondant à une opération définie par des préconditions sur $arg1, arg2, att1, att2$ et une seule post-condition sur $arg2$.

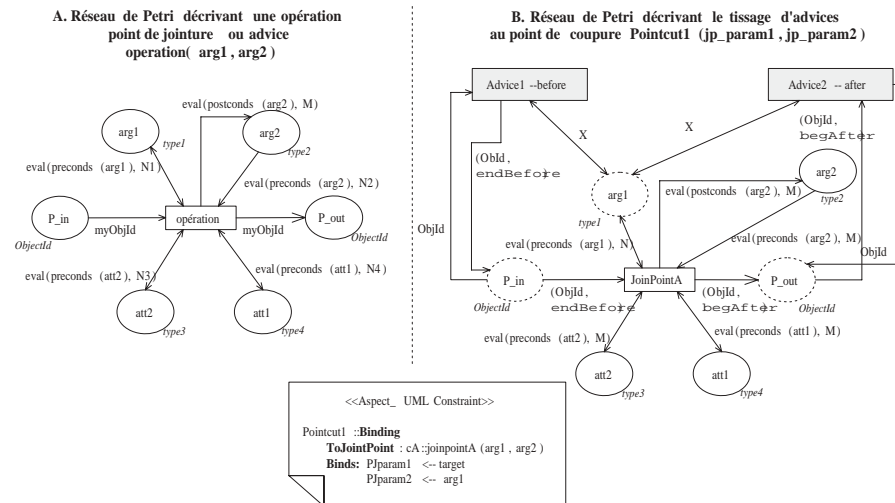


Figure 4. rdPc d'un point de jointure et de son entrecoupage par un point de coupure

3.2. Modélisation des points de coupure

Les points de coupure ont pour principale fonction de regrouper certains points de jointure et d'en exposer le contexte pour le passer aux advices qui doivent s'y tisser. Soit PC un point de coupure associé à un ensemble de points de jointure $PJ^{PC} = \{pj_i | 1 \leq i \leq n\}$ et dont les paramètres formels sont représentés par les variables $Param^{PC} = \{v_1 : t_1, \dots, v_m : t_m\}$. Dans le modèle Aspect-UML, chaque $pj_i \in PJ^{PC}$ est associé à une contrainte $\Gamma(pj_i, PC) \subseteq Param^{PC} \times Args^{pj_i}$ qui définit le passage du contexte de pj_i à PC . Ce contexte, composé par les valeurs des arguments et/ou par l'identité de l'objet invoqué au point de jointure pj_i , est lié aux paramètres formels de PC .

Pour décrire la liaison du contexte d'un point de jointure à un advice, via un point de coupure PC , on associe à chaque $pj \in PJ^{PC}$ une fonction $bind_{pj,PC}(adv, t) \subseteq places(rdPc(adv)) \times places(rdPc(pj))$. Étant donnée la contrainte $\Gamma(pj_i, PC)$, cette fonction calcule la relation entre les places du rdPc de l'advice adv (de type t) et celles du rdPc décrivant le point de jointure pj de PC . La fonction $bind_{pj,PC}(adv, t)$ est définie comme suit :

- $(P_{target}^{adv}, P_{in}^{pj}) \in bind_{pj,PC}$, si $t = \text{before}$ et $\forall v \in Param^{PC}, v \leftarrow target \notin \Gamma(pj, PC)$;
- $(P_{target}^{adv}, P_{out}^{pj}) \in bind_{pj,PC}$, si $t = \text{after}$ et $\forall v \in Param^{PC}, v \leftarrow target \notin \Gamma(pj, PC)$;
- $(P_{x_k}^{adv}, P_y^{pj}) \in bind_{pj,PC}$, si $v_k \leftarrow y \in \Gamma(pj_i, PC)$, $x_k \in Args(adv)$, $y \in Args(pj)$ et $k \in \{1, \dots, m\}$;
- $(P_{x_k}^{adv}, P_{in}^{pj}) \in bind_{pj,PC}$, si $t = \text{before}$, $v_k \leftarrow target \in \Gamma(pj_i, PC)$, $x_k \in Args(adv)$ et $k \in \{1, \dots, m\}$;
- $(P_{x_k}^{adv}, P_{out}^{pj}) \in bind_{pj,PC}$, si $t = \text{after}$, $v_k \leftarrow target \in \Gamma(pj_i, PC)$, $x_k \in Args(adv)$ et $k \in \{1, \dots, m\}$;

Soit PC_ψ , un point de coupure composé d'un ensemble ψ de points de jointure. La sémantique de PC_ψ définit la fusion des places du rdPc des points de jointure de ψ à celles du rdPc des advices qui implémentent PC :

$$sem(PC_\psi) = \{(pj, b) | pj \in \psi, b = bind_{pj,PC}\}$$

3.3. Composition des advices aux points de coupure

À un point de coupure donné plusieurs advices peuvent être candidats à l'exécution. Le cas échéant, ces advices sont statiquement conflictuels s'ils sont de la même catégorie (*before* ou *after*). Selon la modélisation Aspect-UML, une relation de précedence peut ou non avoir été définie entre les aspects contenant les advices. Dans le premier cas, il suffit de composer séquentiellement les advices selon l'ordre pres-

crit. Dans le second cas, ne pouvant faire d'hypothèse sur l'ordre d'exécution, nous composons les advices par entrelacement non déterministe.

Soient deux advices conflictuels Ad_1 et Ad_2 (à un point de coupure donné) et les aspects A_1 et A_2 qui les encapsulent respectivement. Pour composer ces advices à un point de coupure, nous utilisons l'une des deux opérations de composition suivantes :

Composition séquentielle. Si la relation de précédence du modèle Aspect-UML détermine que $A_1 < A_2$, la composition des réseaux de Petri décrivant respectivement Ad_1 et Ad_2 , se note $rdPc(Ad_1) \odot rdPc(Ad_2)$ et suit le patron de composition séquentielle indiqué à la figure 5 (gauche).

Entrelacement non déterministe. Si la relation de précédence du modèle Aspect-UML n'est pas définie pour A_1 et A_2 , la composition des réseaux de Petri décrivant respectivement Ad_1 et Ad_2 se note $rdPc(Ad_1) \oplus rdPc(Ad_2)$ et suit le patron d'entrelacement non déterministe indiqué à la figure 5 (droite).

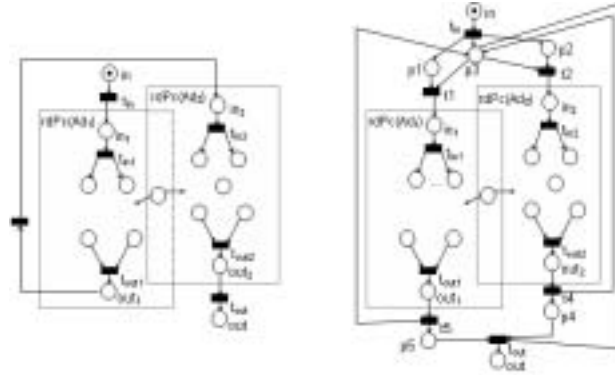


Figure 5. *Patrons de composition séquentielle et d'entrelacement non déterministe*

Une fois les advices conflictuels composés, il suffit maintenant de les lier aux points de coupure qu'ils implémentent. Nous appelons cette opération *tissage*. Le tissage du rdPc de l'advice Ad au point de coupure PC_ψ est noté $rdPc(Ad) \dagger_t sem(PC_\psi)$ (où $t \in \{\text{before}, \text{after}\}$), et est réalisé de la façon suivante :

1. $\forall (pj, bind) \in sem(PC_\psi)$ et $\forall (x, y) \in bind(Ad, t)$
 - Si $x = P_{target}^{Ad}$, alors on crée un arc de la place y vers la transition T^{Ad} avec l'étiquette $ObjId$ si $t = \text{before}$, ou l'étiquette $(ObjId, begAfter)$ si $t = \text{after}$;
 - Si $x \neq P_{target}^{Ad}$, on fusionne la place x du point de jointure à la place y de l'advice, en s'assurant que l'étiquette est $ObjId$ si $t = \text{before}$, ou $(ObjId, begAfter)$ si $t = \text{after}$;
2. Si $t = \text{before}$, on crée un arc de la transition T^{Ad} vers la place P_{in}^{pj} (s'il n'existe pas déjà) en prenant soin de lui adjoindre l'étiquette $(ObjId, endBefore)$. Aussi, on doit changer l'étiquette qui se trouve sur l'arc reliant P_{in}^{pj} à T^{pj} et la remplacer par $(ObjId, endBefore)$.

3. Si $t = \text{after}$, on crée un arc de la transition T^{Ad} vers la place P_{out}^{pj} (s'il n'existe pas déjà) en lui adjoignant l'étiquette $ObjId$. Aussi, on doit changer l'étiquette qui se trouve sur l'arc reliant T^{pj} à P_{out}^{pj} et la remplacer par $(ObjId, \text{begAfter})$

La partie B (à droite) de la figure 4 illustre le tissage de deux advices, un avant et un après le point de jointure $JoinPointA$ du point de coupure $Pointcut1$. Le tissage est réalisé suivant la contrainte de passage de contexte spécifiée par la note Aspect-UML. Les places fusionnées sont indiquées en pointillés.

3.4. Construction de la sémantique d'un modèle Aspect-UML

Soit un modèle Aspect-UML composé d'un ensemble de points de jointure $PJset$, d'un ensemble de point de coupure $PCset$, d'un ensemble d'advices $Adset$ et d'un ensemble d'aspects $Aspects$. On considère également une fonction $getJP(pc)$ qui retourne l'ensemble des points de jointure regroupé par $pc \in PCset$, une fonction $getPC(ad)$ qui retourne les points de coupure de l'advice $ad \in Adset$, une fonction $aspectOf(ad)$ qui retourne l'aspect $a \in Aspects$ auquel appartient $ad \in Adset$ et une fonction $categ(ad)$ qui retourne la catégorie de l'advice ad . Finalement, on dispose d'une relation de précédence $<$ sur les aspects (i.e. un ordre partiel sur $Aspects$) ainsi que des ensembles de mappings $\Gamma(pj, pc)$ entre les arguments d'un point de jointure pj et de point de coupure pc .

La méthode de construction de la sémantique d'un modèle Aspect-UML est la suivante :

1. On construit le réseau $rdPc(ad)$ de chaque $ad \in Adset$ et le réseau $rdPc(pj)$ de chaque $pj \in PJset$. (section 3.1)
2. Pour chaque $pc \in PCset$ tel que $getPJ(pc) = \psi$, on calcule la sémantique de $sem(pc_\psi)$ selon les contraintes de passage de contexte $\Gamma(pj, pc)_{pj \in \psi}$ indiquées.
3. Pour chaque point de coupure, on identifie les aspects conflictuels. Si Ad_1 et Ad_2 sont conflictuels, on construit un nouvel advice par composition séquentielle $(rdPc(Ad_1) \odot rdPc(Ad_2))$ ou par entrelacement non déterminisme $(rdPc(Ad_1) \oplus rdPc(Ad_2))$ selon les indications de la relation de précédence entre $aspectOf(Ad_1)$ et $aspectOf(Ad_2)$. On obtient un nouvel ensemble d'advices $Adset'$ dans lequel les aspects conflictuels ont été composés. (section 3.2)
4. Pour chaque advice $ad \in Adset'$ et chaque $pc_\psi \in getPC(ad)$, on construit par tissage un nouveau réseau $rdPc(ad_{pc_\psi}) = rdPc(ad) \dagger_{categ(ad)} sem(pc_\psi)$. L'ensemble de ces réseaux constituent la sémantique du modèle Aspect-UML.

4. Vérification formelle de la composition des aspects

Un modèle Aspect-UML est susceptible de comporter des erreurs (par exemple un aspect qui viole les propriétés d'un point de jointure, ou encore l'ordre de composition

des aspects qui engendrent un état de blocage). Nous nous intéressons à identifier les erreurs liées à la composition des aspects dans le modèle. Pour ce faire, nous formulons en logique temporelle les propriétés de sûreté et de vivacité que le modèle doit vérifier. Ces propriétés seront vérifiées⁴ par model-checking sur le graphe de marquages du réseau de Petri. Le but visé est de fournir un outil de détection d'erreurs signalant aux utilisateurs les possibles anomalies dans la composition des aspects. Cette approche procède en quatre étapes : (1) la construction du réseau de Petri coloré (rdPc) décrivant le modèle Aspect-UML à vérifier, (2) l'application des techniques d'interprétation abstraite (Cousot *et al.*, 2002) afin de générer une version réduite du graphe de marquages du rdPc, mais toujours sûre vis à vis du graphe original, (3) la formulation en logique temporelle des contraintes à vérifier puis (4) la détection automatique d'éventuelles anomalies dans la composition des aspects en vérifiant, par analyse et model-checking (Berard *et al.*, 2001), les propriétés sur le graphe de marquages.

La vérification des propriétés formulées est directement liée à la composition des aspects i.e. à leur tissage dans le système initial ou à leur composition non déterministe (aspects concurrents). Les erreurs auxquelles on s'intéresse le plus fréquemment sont 1) la violation d'invariants (l'aspect tissé viole les propriétés locales spécifiées au point de jointure) et 2) l'introduction d'états de blocage indésirables (l'ajout d'un aspect bloque l'exécution du système).

Dans le cas de l'application Telecom, on peut souhaiter vérifier l'invariant suivant : "Lorsqu'une connexion est terminée, elle doit immédiatement passer à l'état 2". Autrement dit, la post-condition $state = 2$ au point de jointure $connection :: drop()$ doit toujours être vérifiée, même après le tissage des aspects *Timing* et *Billing*. Cette propriété devra être décrite en logique temporelle avant d'être vérifiée par model-checking sur le graphe de marquages du rdPc. Quant au problème des blocages indésirables, on peut les repérer en appliquant les techniques de détection de blocages des réseaux de Petri. L'utilisateur devra préalablement avoir défini un prédicat permettant de distinguer les états bloquants acceptables de ceux qui sont indésirables. Ainsi, dans l'exemple Telecom, on pourra conclure que l'aspect *Timing* doit explicitement précéder l'aspect *Billing*, sous peine d'engendrer un blocage indésirable dans le graphe de marquages du rdPc.

5. Travaux connexes

Dans la littérature, beaucoup de travaux se sont intéressés à la modélisation des aspects, en revanche peu ont soulevé le problème de la vérification de la composition de ceux-ci. Les travaux présentés dans (Suzuki *et al.*, 1999), (Stein *et al.*, 2002) et (Aldawud *et al.*, 2003) portent sur la modélisation des aspects en utilisant les mécanismes d'extension d'UML. Jacobson (Jacobson, 2003) propose l'analogie entre cas d'utilisation et aspects. Ces approches couvrent plus ou moins superficiellement le

4. Bien sûr la vérification de la composition des aspects dépend directement de la complétude du système spécifié.

paradigme par aspect et n'offrent pas de réel moyen pour prendre en charge la composition et les conflits entre aspects. Peu de travaux ont d'ailleurs abordé ce problème ni du côté de la spécification, ni de celui de la vérification.

En ce qui concerne la vérification des systèmes par aspect, la plupart des approches existantes sont axées sur l'analyse de code ou de traces (Douence *et al.*, 2002, Storzer *et al.*, 2003). Dans (Krishnamurthi *et al.*, 2004), les auteurs proposent une technique de vérification incrémentale par model-checking des programmes orientés aspect. L'approche est intéressante mais la construction du modèle formel du programme AspectJ à vérifier ne semble pas automatisée. Dans un même esprit, (Denaro *et al.*, 2001) propose de traduire les programmes AspectJ en PROMELA pour ensuite les vérifier avec le model-checker Spin. On mentionnera également l'approche de (Tessier *et al.*, 2004) qui vise à détecter les conflits directs entre aspects. On ne vérifie toutefois pas si l'introduction des aspects viole ou non la correction du programme.

6. Conclusion et travaux futurs

Le présent travail expose une méthodologie pour la modélisation et la vérification des systèmes par aspects. Nous avons d'abord présenté le profil Aspect-UML comme notation de modélisation intégrant les constructions du paradigme aspect. Aspect-UML permet, entre autres, de mettre en évidence les contraintes de précédence entre aspects et les propriétés caractérisant le contexte de leur composition. Tel qu'expliqué, les modèles Aspect-UML peuvent être traduits formellement en termes de réseaux de Petri colorés (rdPc). Les propriétés de sûreté et vivacité spécifiées en logique temporelle par l'utilisateur seront vérifiées par model-checking, sur le graphe de marquages des rdPc, pour assurer que la composition des aspects ne compromet pas la correction du modèle.

L'approche présentée est en cours de réalisation, un certain nombre de travaux futurs sont à envisager. D'abord, définir les règles formelles de transformation des modèles Aspect-UML en rdPc ; cette transformation sera entièrement automatisée. Ensuite, appliquer les techniques d'abstraction (Clarke *et al.*, 2000) afin de limiter le problème d'explosion d'états, nous nous intéresserons particulièrement à l'abstraction de variables (Kurshan, 1994) et à l'abstraction de valeurs (Peng, 2002). Ainsi, le model-checking sera appliqué sur les systèmes de transitions abstraits obtenus. À l'instar de (Krishnamurthi *et al.*, 2004), il serait intéressant d'inclure dans notre approche la vérification du corps de l'advice, au lieu de restreindre celui-ci à des pré et post conditions définies par l'utilisateur. Il nous suffirait d'inclure dans Aspect-UML les diagrammes d'états décrivant les advices et d'en dériver des rdPc. Finalement, nous visons également la mise au point d'une méthode de vérification incrémentale des modèles par aspects. En effet, quand on ajoute un aspect, les points de l'exécution touchés par la modification sont clairement mis en évidence. On peut ainsi restreindre la vérification à ces points précis, sans avoir à revérifier tout le modèle. Les processus de développement incrémentaux pourront ainsi tirer profit du paradigme aspect, pour l'ajout de fonctionnalités transversales, tant lors du design que de la vérification.

7. Bibliographie

- Aldawud O., Elrad T., Bader A., « A UML Profile for Aspect Oriented Software Development », *Intl Workshop on AOP (Intl Conf on AOSD)*, March, 2003.
- AspectJ Team, « The AspectJ Programming Guide. », February, 2002. Xerox Corporation.
- Berard B., Bidoit M., Finkel A., Laroussinie F., Petit A., Petrucci L., Schnoebelen P., McKenzie P., *Systems and Software Verification*, Springer-Verlag, 2001.
- Clarke E., Grumberg O., Peled D., *Model Checking*, MIT Press, 2000.
- Cousot P., Cousot R., « Systematic Design of Program Transformation Frameworks by Abstract Interpretation », *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p. 178-190, 2002.
- Denaro G., Monga M., « An Experience on Verification of Aspect Properties », *International Workshop on Principles of Software Evolution*, Sept, 2001.
- Douence R., Fradet P., Sudholt M., Detection and Resolution of Aspects Interactions, Technical Report n° RR-4435, INRIA, April, 2002.
- Jacobson I., « Use Cases and Aspects - Working Seamlessly Together », *Journal of Object Technology*, www.jot.fm/issues/issue-2003-07/column1, vol. 2, n° 4, p. 7-28, 2003.
- Jensen K., *Colored Petri Nets. Basic concepts, analysis methods and practical use*, Springer, 1997.
- Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J.-M., Irwin J., « Aspect-oriented Programming », *ECOOP, LNCS 1241*, p. 220-242, June, 1997.
- Krishnamurthi S., Fisler K., Greenberg M., « Verifying Aspect Advice Modularly », *SIGSOFT/FSE-12*, Oct-Nov, 2004.
- Kristensen L. M., Christensen S., Jensen K., « The practitioner's guide to coloured Petri nets », *International Journal on Software Tools for Technology Transfer*, vol. 2, n° 2 : Special section on coloured Petri nets, p. 98-132, 1998.
- Kurshan R. P., « Computer aided verification of coordinating processes : the automata theoretic approach », *Princeton University Press*, 1994.
- Peng H., Improving compositional verification environment synthesis and syntactic model reduction, PhD thesis, Concordia University, Montreal, Québec, Canada, 2002.
- Peterson J., *Petri net Theory and the modelling of system*, Prentice Hall, 1981.
- Stein D., Hanenberg S., Unland R., « A UML-based aspect-oriented design notation for AspectJ », *Intl Conf on AOSD*, p. 106-112, 2002.
- Storzer M., Krinke J., Breu S., « Trace Analysis for Aspect Application », *AAOS'03*, 2003.
- Suzuki J., Yamamoto Y., « Extending UML with Aspects : Aspect support in the design phase », *AOP Workshop at ECOOP*, June, 1999.
- Tessier F., Badri L., Badri M., « Towards a Formal Detection of Semantic Conflicts Between Aspects : A Model-Based Approach », *AOM Workshop (7th UML Conference)*, 2004.
- Vachon J., Mostefaoui F., « Achieving supplementary requirements using aspect-oriented development », *Intl Conf on Enterprise Information Systems (ICEIS)*, p. 584-587, April, 2004a.
- Vachon J., Sahraoui H., Essalih M., Mili. H., « Vérification par model-checking de systèmes hybrides objets-règles », *Langages et Modèles à Objets (LMO), Revue L'Objet*, vol. 10, p. 259-275, 2004b.