

Spécification et Vérification par Interprétation Abstraite d'Aspects pour la Distribution

PIERRE-LOÏC GAROCHE MARC PANTEL
XAVIER THIRIOUX

Institut de Recherche en Informatique de Toulouse, France
{ garoche, pantel, thirioux } @enseeiht.fr

Abstract

Nous présentons une méthode pour spécifier et vérifier des programmes en séparant les préoccupations. En particulier, nous décrivons la spécification puis la vérification par interprétation abstraite de programmes distribués. Un programme est composé de deux parties, une partie concurrente et une partie fonctionnelle. Elles sont ensuite composées en utilisant un mécanisme de tissage simple (syntaxique). Nous décrivons alors comment combiner les analyses spécifiques à chaque partie afin de calculer une approximation du plus petit point fixe de la sémantique collectrice du système tissé. Un tel cadre permet de vérifier des propriétés de sûreté sur des programmes distribués spécifiés dans une approche par aspects. Ces travaux sont les prémisses d'une approche formelle et générique de l'analyse d'autres aspects par interprétation abstraite, en combinant des sémantiques de natures différentes.

1 Introduction

La conception des programmes a beaucoup évolué. Les programmes sont désormais définis puis assemblés de manière compositionnelle. La programmation orientée par aspect est l'une des futures orientations possibles pour la prochaine génération de programmes. Le concept principal de cette approche est qu'une propriété "qui ne peut être proprement encapsulée doit être implantée par un aspect" [17]. Cet article présente la description d'un programme de façon modulaire, en séparant les *préoccupations*, mais surtout l'application de la théorie de l'interprétation abstraite à la vérification de tels programmes en exploitant leur structure. Nous décrivons ici cette méthodologie dans la conception de programmes distribués. L'approche choisie s'inscrit dans la démarche originelle de l'AOP et ne correspond pas exactement à l'approche "AspectJ". Les sémantiques formelles associées aux différentes préoccupations sont exprimées puis analysées indépendamment les unes des autres. Enfin la sémantique du programme final est exprimée comme la combinaison de ces sémantiques. Nous nous intéressons ici spécifiquement à des propriétés de sûreté. Nous approximerons donc la sémantique collectrice du programme tissé.

Les deux principales contributions de ce travail concernent d'une part la définition d'un aspect pour décrire la concurrence et d'autre part la vérification formelle de programmes tissés. Les travaux de l'état de l'art relatifs à la concurrence s'intéressent peu à l'expression d'un aspect spécifique pour décrire cette concurrence, mais plutôt à l'application concurrente d'aspects comme dans [3, 6]. Dans [20], les auteurs décrivent un langage permettant de décrire l'application d'aspects à partir d'événements distribués. Le programme doit donc être initialement distribué. L'analyse formelle d'aspects reste un domaine assez récent. Plusieurs travaux comme [7] et [15] tendent à décrire de façon formelle la sémantique des programmes orientés aspects pour ensuite détecter et résoudre les conflits d'applications des aspects dans un cas, ou garantir la consistance du programme par construction dans l'autre cas. Peu de travaux concernent spécifiquement la vérification de propriétés du programme tissé. [18] décrit comment factoriser l'utilisation de pre et post conditions comme aspects du programme et [21] utilise des techniques d'analyse statique pour optimiser l'application dynamique d'aspects. Pour finir ce bref aperçu des travaux connexes à ce travail, il faut mentionner les efforts des trente dernières années quant à la vérification statique formelle de propriété pour les langages de programmation. L'interprétation abstraite [4] est l'une de ces approches et décrit un cadre générique pour l'analyse statique. Des travaux plus récents s'intéressent à l'expression de propriétés complexes et spécifiques à différents paradigmes comme l'objet [2] ou les systèmes mobiles [10].

Le présent article sera présenté en deux parties. La première décrira la spécification de systèmes distribués en introduisant les différents aspects avec leur sémantique associée ainsi que leur combinaison. La seconde partie s'intéressera à la vérification de la sémantique collectrice des différents aspects ainsi que de leur composition.

```

25: bid(Actor x, int id, int val) {
  if (items[myid][id].active) { // the object is active
    if (val > items[myid][id].val) {
      items[myid][id].owner = x;
      return (1,[]);
    }
    else {
      return (2,[( ``value `` , items[myid][id].val )]);
    }
  }
  else { // this object is inactive
    return (3,[]);
  }
}

```

Figure 3: Fonction bid de la partie fonctionnelle

un acteur est associé à cette adresse au point 18 pendant qu'un message est envoyé à l'argument du message x au point 17. La continuation de la branche 25 est spécifiée entre accolades pour indiquer un choix non déterministe.

2.2 Spécification fonctionnelle

L'aspect fonctionnel du système est décrit dans une partie spécifique. Nous choisissons, pour le représenter, un langage impératif typé simple. La description fonctionnelle est ici constituée d'un ensemble de variables globales et de fonctions, ou procédures, qui définissent un ensemble de services.

Nous introduisons un type cap_t , spécifique à nos variables concurrentes qui peuvent seulement être comparées ou affectées. L'opérateur *return* est surchargé afin de pouvoir renvoyer un entier ainsi qu'une liste de couple (nom, valeur).

Nous utilisons une sémantique opérationnelle petit-pas afin de représenter l'évaluation des fonctions. Un état est représenté par un couple décrivant l'environnement global ainsi que la pile des appels associés aux environnements locaux.

La description fonctionnelle du système d'enchères décrit seulement localement comment le système réagit aux messages suivants leurs arguments ou ses variables globales. Par exemple, lorsqu'un acteur demande à un autre sa liste d'objets, il reçoit ensuite une liste d'objets; mais la description concurrente ne spécifie pas l'utilisation de cette liste. La partie fonctionnelle décrit également la technique d'enchère, la réponse aux enchères des autres, mais s'abstrait d'un protocole de communication lui-même, défini dans la partie concurrente. Nous présentons dans la figure 3 la fonction qui prend en compte une enchère. Les variables *items* et *myid* sont des variables globales.

L'implémentation de la fonction *bid* est très naturelle. Elle est appelée lorsque l'acteur reçoit une enchère sur l'un de ses objets. Si l'enchère est correcte, l'acteur qui a enchéri est mémorisé. Les autres cas traitent les résultats invalides.

2.3 Le mécanisme de tissage

Les deux descriptions, concurrente et fonctionnelle, doivent maintenant être tissées afin de construire un programme distribué. Pour conserver une présentation simple et afin de faciliter l'étape suivante, c'est à dire la vérification par interprétation abstraite de programmes construits en combinant des sémantiques de natures différentes, nous avons choisi d'utiliser un tissage syntaxique. Une procédure de nom X^l est associée à la branche de comportement qui reçoit des messages étiquetés X au point de programme l avec un nombre identique d'arguments. Le tissage est défini en quatre parties. Nous introduisons tout d'abord JavAct, un intergiciel Java dans lequel nous construisons le programme final. Nous expliquons ensuite dans la seconde et la troisième partie comment exprimer des termes CAP dans JavAct puis comment introduire la partie fonctionnelle afin de construire le système final. La dernière partie formalise la sémantique résultante.

2.3.1 JavAct: un intergiciel Java pour les agents mobiles adaptables

JavAct est un environnement permettant le développement de programmes Java concurrents et distribués suivant le modèle des acteurs. Comparé aux mécanismes classiques de bas niveau comme les sockets, le RMI et les services Web ou SOAP, il possède un haut niveau d'abstraction. Il fournit des primitives de création d'acteurs, de changement de leur comportements, de localisation, de communication par envoi de messages.

JavAct n'est pas l'unique environnement développé dans ce domaine. Par exemple, l'ActorFoundry de AGHA est aussi dédié à la conception de systèmes distribués dans le modèle d'acteurs. Néanmoins, JavAct a été défini

sur les mêmes principes que ceux utilisés dans CAP. Par conséquent, il est plus adapté pour exprimer la traduction de termes CAP en programmes Java réalistes et exécutables. Cependant, certaines différences minimes existent entre le modèle d'acteurs de CAP et celui de JavAct. En CAP par exemple, par défaut un acteur disparaît après le traitement d'un message s'il n'est pas réinstallé explicitement sur un comportement. Au contraire, en JavAct, la réplication est le comportement par défaut, c'est-à-dire qu'implicitement toute continuation est de la forme $e \triangleright s \parallel C$. La primitive spéciale *suicide()* permet alors d'exprimer explicitement la terminaison d'un acteur.

Pour utiliser JavAct, on procède en deux étapes. Premièrement, on doit définir les comportements au moyen d'interfaces Java, où chaque méthode correspond à une branche de comportement. Puis à partir de cette description, un pré-processeur engendre des classes pour les comportements ainsi que pour chaque étiquette de message. Secondement, l'utilisateur doit définir les corps de méthodes dans les classes engendrées, c'est-à-dire les instructions Java et JavAct qui seront exécutées au traitement des messages. Toutes les primitives JavAct appelées dans une méthode sont exécutées seulement lorsqu'on quitte la méthode, afin que si plusieurs changements de comportement sont définis, seul le dernier soit effectué.

De CAP à JavAct. Nous définissons ici assez directement une transformation automatique de CAP vers une spécification JavAct. Chaque comportement défini syntaxiquement doit être déclaré. Dans notre système d'enchères, nous avons deux comportements: l'un représentant l'attente, et l'autre l'envoi d'une nouvelle enchère à un acteur.

Voici les interfaces à définir:

```

interface Standby extends BehaviorProfile {
    ...
    public void send_bid(Item it);
    public void bid(Actor x, int id, int val);
}

interface Bid extends BehaviorProfile {
    public void bid_ok();
    public void value(int val);
    public void finished();
    public void unknown_object();
}

```

Insertion de la partie fonctionnelle. Le pré-processeur JavAct prend la description précédente et engendre les classes correspondantes. Nous devons maintenant entrelacer la description fonctionnelle avec les continuations des branches de comportements. Le mécanisme de tissage est simple et complètement syntaxique. Illustrons-le par un exemple: La branche $bid(x, id, val)$ est associée à la continuation

$$bid^{25}(x, id, val) = \zeta(e, s) (\{ e \triangleright^{26} s \parallel x \triangleleft^{27} bid_ok(e, id) \}, \\ \{ e \triangleright^{28} s \parallel x \triangleleft^{29} value(e, id, value) \}, \\ \{ e \triangleright^{30} s \parallel x \triangleleft^{31} finished(e, id) \})$$

Et la description fonctionnelle correspondante est définie dans la figure 3. Afin que la spécification soit consistante, la partie fonctionnelle doit définir le même nombre d'instructions return que le nombre de continuations possibles d'une branche. Chaque i -ème return est alors associé à la i -ème continuation, dans laquelle les valeurs renvoyées sont liées aux noms spécifiés.

Comme expliqué précédemment, la réplication est implicite en JavAct. Le code Java résultant pour la méthode *get_value* est alors:

```

public void bid(Actor x, int id, int val) {
    if (items[myid][id].active) { // this object is active
        if (val > items[myid][id].val) {
            items[myid][id].owner = x;
            send(new JAMbid_ok(), x);
        }
        else {
            send(new JAMvalue(items[myid][id].val), x);
        }
    }
    else { // this object is inactive
        send(new JAMfinished(), x);
    }
}

```

La sémantique du système complet. Nous définissons la sémantique du système global, exprimant le tissage des aspects fonctionnels et concurrents.

Soit \rightarrow_1 (respectivement \rightarrow_2) la relation de transition de la partie concurrente (respectivement fonctionnelle). On définit $t \rightarrow_2^0 t'$, avec t' le dernier état d'une exécution maximale finie de \rightarrow_2 partant de t , si elle existe.

Rappelons la relation de transition concurrente. Une configuration C qui contient un acteur et un message peut réaliser une transition entre les deux si et seulement s'ils partagent la même adresse et si l'étiquette du message $m()$ est comprise par l'acteur. Le terme réduit contient alors une des continuations possibles de l'acteur, obtenue par un choix non-déterministe, ainsi que les autres composants qui n'ont pas interagi. Le message et l'acteur impliqué ne sont donc plus présents.

$$a \triangleright [m(\tilde{x}_i) = \zeta(e, s)(C_1 \dots C_n)] \parallel a \triangleleft m() \rightarrow C_i$$

La représentation d'un état du système complet correspond à une version enrichie d'un terme CAP où chaque acteur est associé à un environnement fonctionnel Γ

$$a \triangleright [\Gamma : m(\tilde{x}_i) = \zeta(e, s)(C_1 \dots C_n)] \parallel a \triangleleft m() \rightarrow C_i$$

Soit $fun_m(p_1, \dots, p_n)$ la fonction associée syntaxiquement à cette branche $m()$. Finalement, la sémantique du système complet est:

$$a \triangleright [\Gamma : m(\tilde{x}_i) = \zeta(e, s)(C_1 \dots C_n)] \parallel a \triangleleft m() \rightarrow C_i[name_k \mapsto val_k]$$

avec la transition $\Gamma, fun_m(\tilde{x}_i) \rightarrow_2^\omega \Gamma', (i, \{(name_k, val_k)^k\})$ et les acteurs de C_i associés à l'environnement global Γ' .

3 Vérification de propriétés de sûreté par interprétation abstraite

La partie précédente s'intéressait à la spécification par préoccupation de programmes Java distribués. Nous décrivons maintenant comment vérifier des propriétés sur le programme final en considérant des analyses spécifiques à chaque spécification ainsi qu'à la façon dont sont combinées leur sémantique. L'interprétation abstraite est une méthode générale pour la manipulation et l'abstraction de sémantiques. Dans cette partie, nous utiliserons un tel cadre pour approximer la sémantique collectrice du programme tissé. Une telle combinaison d'analyses va permettre d'obtenir des résultats bien plus précis qu'une analyse séparée. Par exemple, dans notre cas distribué, l'analyse du terme CAP sans la partie fonctionnelle doit considérer toutes les continuations de façon non déterministe et va donc rendre les résultats de l'analyse imprécis. Nous donnons d'abord quelques rappels sur l'interprétation abstraite, puis décrivons les différentes analyses et leur combinaison.

3.1 Interprétation abstraite

L'interprétation abstraite [4] est une théorie de l'approximation discrète de sémantiques. Un de ses aspects fondamental est que toute sémantique peut être exprimée comme points fixes d'opérateurs monotones sur des ordres partiels complets. Une sémantique est définie par un n-uplet $(S, \subseteq, \perp, \cup, \top, \cap)$ et par un ensemble de primitives modélisant les fonctions de transitions. D'après [5], une sémantique abstraite est définie par un treillis $(S^\#, \sqsubseteq)$, une base abstraite d'itération $\perp^\#$, une fonction de concrétisation $\gamma : S^\# \rightarrow S$ et une fonction sémantique abstraite $\mathbb{F}^\#$. Le plus petit point fixe de la sémantique concrète peut être approximé par le plus petit point fixe de la sémantique abstraite. Le cadre de l'interprétation abstraite donne les outils nécessaires à la définition d'abstractions correctes par construction ainsi qu'au calcul des points fixes.

3.2 Analyse de sûreté pour CAP

Notre calcul de processus, CAP, nous permet de spécifier des systèmes distribués complexes et de calculer leurs évolutions. Nous nous intéressons ici particulièrement à la vérification de propriétés de sûreté sur les termes de CAP, c-à-d à aux propriétés qui doivent être vérifiées dans tous les états atteignables du système. L'ensemble de ces états est décrit par la sémantique collectrice du terme. Les travaux de Feret [10] sur les systèmes mobiles sont le cadre principal dans lequel nous instancions notre calcul CAP. Ce cadre permet d'exprimer des calculs de processus dans une forme unifiée, dite non standard, et de développer des analyses sur ces calculs en approximant leur sémantique collectrice.

3.2.1 La sémantique collectrice non standard

Un calcul de processus exprimé sous la forme non standard de [10] est défini par un ensemble d'éléments : il faut caractériser les points de programmes du calcul, les capacités d'interactions associées à ces points de programme, les règles de transitions entre ces interactions et enfin une fonction d'extraction qui permet de passer un terme du calcul à un terme non standard. Une configuration non standard est alors constituée d'un ensemble de processus. Les processus sont des triplets (point de programme, marqueur identité, environnement). Pour effectuer une

transition, il faut mettre en relation des processus présents dans la configuration, vérifier des conditions spécifiées par la règle, calculer un passage de valeurs et introduire de nouveaux processus. Les processus interagissant étant supprimés lors du calcul de la transition.

La sémantique collectrice d'un terme \mathcal{S} dans cette sémantique non standard peut être exprimée comme le plus petit point fixe du morphisme complet pour l'union \mathbb{F} sur le treillis complet $\wp(\Sigma^* \times \mathcal{C})$ défini par :

$$\mathbb{F}(X) = (\{\varepsilon\} \times \mathcal{C}_0(\mathcal{S})) \cup \left\{ (u, \lambda, C') \mid \begin{array}{l} \exists C \in \mathcal{C}, (u, C) \in X \\ \text{et } C \xrightarrow{\lambda} C' \end{array} \right\}$$

Les éléments de ce treillis sont les paires (u, C) où u est le mot des transitions qui ont conduit à la configuration C . La première partie décrit l'ensemble des configurations initiales : $u = \varepsilon$.

Afin d'exprimer CAP sous cette forme, nous avons dû caractériser les éléments décrits ci-dessus. Les détails complets peuvent être trouvés dans [13] et la preuve de la bisimulation entre la forme standard et non standard dans [11]. Les processus représentent les acteurs, les messages ainsi que les branches de comportements des acteurs dynamiques ($a \triangleright s$).

3.2.2 Domaines abstraits

Afin d'abstraire la sémantique collectrice concrète d'un terme CAP exprimé sous sa forme non standard, nous devons sur-approximer les ensembles de configurations par des éléments de domaines abstraits. Le calcul d'une transition nécessite des informations sur les valeurs des environnements associés aux différents processus. Ainsi le domaine abstrait utilisé pour approximer la sémantique collectrice doit nécessairement contenir des informations permettant de calculer le flot de contrôle. Pour vérifier des propriétés non représentables par des informations de flot de contrôle, nous construisons un domaine abstrait en calculant le produit cartésien d'un domaine de flot avec un domaine plus apte à représenter telle ou telle propriété. Nous présentons maintenant trois familles de domaines abstraits pour l'analyse de CAP.

Propriétés du flot de contrôle Ce domaine abstrait présenté dans [9], approxime les valeurs des variables dans les environnements associés aux processus. Il est paramétré par un domaine abstrait appelé domaine Atome. Chaque point de programme est associé à un tel atome qui représente une abstraction de son marqueur identité ainsi que de son environnement. Lors du calcul d'une transition, les atomes interagissant sont liés avec les contraintes spécifiques à la transition (même adresse, même étiquette de message, ...). Si ces contraintes sont satisfaisables, la transition a lieu; le passage de valeur, via les atomes, est calculé et les points de programmes correspondant aux nouveaux acteurs ou messages sont mis à jour. Dans ce domaine, nous nous focalisons sur les valeurs et abstrayons complètement les occurrences des processus. La consommation de l'acteur et du message lors de la transition n'apparaît donc pas dans la sémantique abstraite de ce domaine.

Analyse de dénombrement Ce domaine décrit dans [8], permet de dénombrer le nombre de processus associés à un point de programme donné ainsi que le nombre de transitions effectuées. Cette approximation est effectuée en plusieurs étapes. Une configuration donnée est approximée par une fonction qui associe à chaque point de programme et à chaque étiquette de transition son occurrence. Comme nous nous intéressons à la sémantique collectrice du terme analysé, nous obtenons donc un ensemble de telles fonctions. Cet ensemble est ensuite approximé par un produit réduit de domaines abstraits numériques. Le premier élément associé à chaque point de programme ou étiquette de transition un intervalle entier et le second est le domaine numérique relationnel [16], le domaine des égalités affines entre ces variables (points de programmes et étiquettes de transitions). Lors du calcul d'une transition, ce domaine permet de vérifier que les agents (acteurs et messages) interagissant sont bien présents en nombre suffisant dans la configuration. Le calcul d'une transition va ensuite retirer de la configuration résultante l'acteur et le message participant ainsi que rajouter les nouveaux agents.

Analyse de linéarité Les domaines abstraits précédents étaient indépendant du calcul de processus représenté sous forme non standard. Nous présentons ici le domaine abstrait défini dans [12] et spécifique au calcul CAP. Il permet de vérifier que dans toutes les configurations atteignables, une adresse est associée à au plus un acteur. Ce domaine abstrait permet de calculer un mode d'utilisation pour chaque variable dans les environnements. Ce mode d'utilisation dans $\{\perp, \circ, \bullet, \top\}$ calcule si la variable peut être liée à zéro, un ou plusieurs acteurs. Ce domaine abstrait et sa sémantique associée sont inspirés des domaines abstraits pour le flot de contrôle mais présente des différences majeures. Au lieu de calculer seulement un flot en avant et en mettant à jour les environnements associés aux nouveaux acteurs ou messages, la sémantique abstraite est constituée de deux flots, un premier en avant puis un second en arrière. Le premier calcule le passage de paramètre ainsi que l'utilisation de ces valeurs. Ensuite, le second permet de refléter sur les acteurs et messages interagissant lors de la transition calculée les résultats obtenus par le premier flot.

3.3 Analyse de flot de contrôle pour la partie fonctionnelle

3.3.1 Sémantique collectrice concrète

La relation de transition \rightarrow_2 définissant la sémantique opérationnelle petit-pas de notre langage impératif peut maintenant être utilisée pour décrire la sémantique collectrice de la partie fonctionnelle. Les états de cette sémantique sont des couples dont le premier élément représente l'environnement global et le second la séquence de paires (point de programme, environnement local) des appels.

Nous nous intéressons ici à abstraire la sémantique collectrice induite par la relation de transition de la partie fonctionnelle \rightarrow_2 . Elle est définie comme le plus petit point fixe du morphisme monotone \mathbb{F} défini par

$$\mathbb{F}(X) = (\Gamma_0, \square) \cup \left\{ f' \mid \begin{array}{l} \exists f \in \mathcal{F}, f \in X \\ \text{and } f \rightarrow_2 f' \end{array} \right\}$$

où Γ_0 est l'environnement global initial.

3.3.2 Domaines abstraits et analyse statique

Nous esquissons l'analyse de la sémantique collectrice de programmes impératifs par interprétation abstraite en soulignant les spécificités de notre approche. Afin de calculer une abstraction correcte de cette sémantique, nous devons définir un domaine abstrait $F^\#$ permettant de représenter les propriétés réalisées par un ensemble de couples environnements globaux et piles d'appels.

$$(\wp(\mathcal{F}), \subseteq) \xrightleftharpoons[\alpha_F]{\gamma_F} (F^\#, \sqsubseteq^\#)$$

Nous approximations ces spécifications fonctionnelles avec comme objectif l'expression des problématiques liées au flot de contrôle, afin d'obtenir des résultats précis sur les propriétés liées à la concurrence, dans le programme global. En effet, l'analyse précise des flots de valeur cap_t à travers les fonctions de cette partie va permettre de réduire le non déterminisme lors de l'analyse de la partie concurrente.

Nous devons tout d'abord abstraire les environnements. Nous partitionnons l'ensemble des variables de l'environnement afin de séparer les variables de type cap_t des autres. Les premières seront directement associées aux valeurs abstraites de la partie concurrente, mais les secondes pourront être abstraites de façon paramétrique suivant leur type et la précision souhaitée.

Ensuite, nous approximations les appels de piles. Nous introduisons un domaine abstrait $S^\#$ pour représenter les propriétés de piles d'appels. Nous définissons $S^\#$ comme l'abstraction de $S = (L_p \times E)list$ en définissant la correspondance de Galois (α_S, γ_S) . Par exemple, nous pouvons choisir ces fonctions telles que les différents éléments soient fusionnés et associer à chaque point de programme un environnement abstrait.

$$(\wp(S), \subseteq) \xrightleftharpoons[\alpha_S]{\gamma_S} (L_p \mapsto E^\#, \sqsubseteq)$$

Enfin, nous devons représenter un ensemble d'états. Nous la suite, nous l'approximations par le couple (environnement abstrait, pile abstraite). Nous avons alors $F^\# = E^\# \times S^\#$ et α_F et γ_F sont tels que $\forall f \in \wp(F), f \subseteq \gamma(\alpha(f))$. $\sqsubseteq^\#$ est défini comme l'extension point-à-point des relations de pré-ordre.

Les domaines génériques pour les variables de type autre que cap_t peuvent maintenant être instanciés par des domaines abstraits numériques et symboliques comme [16] ou [19]. Le domaine abstrait pour les variables cap (de type cap_t) est laissé non spécifié. C'est l'analyse de la partie concurrente qui le précisera. Les seules opérations autorisées (et donc leur contreparties abstraites), sont la comparaison et l'affectation d'une valeur à une variable existante. Une telle restriction permet de garantir la monotonie de la transition abstraite tout en conservant inchangées les informations sur les variables cap .

La sur-approximation de tous les états accessibles nous permet :

- d'approximer la valeur associée avec les variables cap dans les messages qui peut ensuite être utilisée dans la partie concurrente;
- de déterminer un sur-ensemble correct des instructions return accessibles afin d'obtenir une sur-approximation précise des transitions dans la partie concurrente.

La famille des analyses présentées ici est seulement dédiée à l'analyse de flot de contrôle, mais de telles analyses peuvent être aisément étendues afin d'observer des propriétés plus fines de la partie fonctionnelle.

3.4 Combinaison des analyses

Nous décrivons maintenant comment les deux analyses peuvent être combinées afin de calculer un ensemble d'invariants du système global, c-à-d des propriétés vérifiées dans tous les états accessibles. Le tissage abstrait est décrit en deux parties. La première décrit le calcul du point fixe de la combinaison des sémantiques. La seconde décrit la combinaison effective des domaines abstraits.

3.4.1 Expression du plus petit point fixe de la sémantique collectrice du programme tissé.

Soit un programme décrit par plusieurs descriptions \mathcal{D}_i et un tissage \otimes . Chaque description \mathcal{D}_i est associée à une sémantique spécifique S_i sur le domaine D_i et un état initial I_i . Le tissage \otimes décrit comment l'état initial est construit et où et comment le flot d'évaluation de la sémantique globale passe d'une description à l'autre. Il doit en particulier spécifier les conditions statiques ou dynamiques de tels changements. Un tissage \otimes entre les descriptions $\mathcal{D}_i = (S_i, D_i, I_i)$ est défini par le n-uplet $(I, (J_i)_i, M, S_k)$ où I est l'état initial et J_i l'ensemble des *joinpoints* de la sémantique S_i . M définit la fonction qui associe aux *joinpoints* des points de programme et permet de décrire le changement de sémantique. S_k est la sémantique utilisée dans l'état initial. Lors de l'exécution du programme, la rencontre d'un *joinpoint* détermine la prochaine transition à calculer. Lorsque cet appel est terminé, le programme continue dans la sémantique précédente.

Dans l'exemple distribué, soit \mathcal{D}_1 la partie concurrente et \mathcal{D}_2 la partie fonctionnelle. I_1 est défini comme l'état initial du terme CAP, dans sa forme non standard, un ensemble de processus concurrents. I_2 associe à chaque processus cap un environnement fonctionnel global avec ses valeurs initialisées. L'état initial I du programme tissé est donc un couple $(I_1, \lambda x \in I_1. I_2)$. Le tissage est ici statique, l'ensemble des *joinpoints* J_1 est l'ensemble des points de programme de comportement. La description \mathcal{D}_2 n'appelle pas d'autres description, J_2 est vide. Dans cette présentation, la description fonctionnelle agit comme un aspect de l'approche AspectJ. Chaque fonction de type de retour *cap_t* est une définition d'*advice* et la fonction M lie les *joinpoints* avec les *advices*. La sémantique utilisée dans l'état initial est la sémantique de la description concurrente (S_1). La passage de S_1 à S_2 est défini par M , de façon syntaxique : \rightarrow_2 est encapsulé dans \rightarrow_1 . La relation de transition \rightarrow_1 peut être définie par : $x \xrightarrow{\alpha}_1 x \setminus removed_threads \cup new_threads$ où x est l'ensemble des processus, *removed_threads* les processus consommés (*i.e.* l'acteur et le message) et *new_threads* les processus de la continuation. La continuation est déterminée de façon non déterministe dans les analyses précédentes, elle dépend maintenant de la réponse de la partie fonctionnelle.

Soit la configuration (C, F) et la transition entre un acteur (I_a, id_a, E_a) , un comportement (I_b, id_b, E_b) et un message (I_m, id_m, E_m) de C . Lorsque la transition (I_a, I_b, I_m) est calculée, la partie fonctionnelle correspondante est appelée. Soit *mess* l'étiquette du message reçu. La fonction M associe la fonction *mess* d'étiquette I_b au *joinpoint* courant I_b . Cette fonction est appelée dans l'environnement global défini par l'élément $F((I_b, id_b, E_b))$. Le résultat est une paire (k, env) où k est la k -ième continuation à lancer et *env* une fonction associant une valeur aux noms de variables libres dans la k -ième continuation. Le calcul est maintenant à nouveau dans la sémantique S_1 et les nouveaux processus sont lancés. La configuration résultante est la paire (C', F') où C' est la nouvelle configuration concurrente et F' est définie par :

$$F'(x) = \begin{cases} F((I_b, id_b, E_b)) \cup env & \text{si } x \in new_threads \\ F(x) & \text{sinon} \end{cases}$$

3.4.2 Approximation du plus petit point fixe de la sémantique collectrice du programme tissé.

Afin de vérifier des propriétés de sûreté sur le programme tissé, nous devons calculer sa sémantique collectrice. Cette sémantique n'étant, en général, pas calculable, nous devons en calculer une approximation correcte

$$\text{décidable} : \otimes_i D_i \xrightarrow[\alpha]{\gamma} \otimes_i^\# D_i^\#$$

L'approximation de la sémantique collectrice intervient à plusieurs étapes du calcul des transitions abstraites. Tout d'abord, lors du calcul d'une transition d'une sémantique du programme tissé. C'est l'abstraction spécifique à la sémantique, déjà introduite dans les sections précédentes. Nous en avons une pour les termes CAP (cf. 3.2) et une autre pour la partie fonctionnelle (cf. 3.3). Nous utilisons l'information contenue dans les domaines abstraits pour calculer une sur-approximation des transitions réellement calculables. Une seconde étape d'approximation intervient lors du tissage abstrait. Lors du calcul d'une transition d'une sémantique donné, cette transition peut faire appel à d'autres sémantiques. C'est l'approximation de l'application des *advices* aux *joinpoints* du programme. Nous devons donc calculer une sur-approximation de la détection des *advices* applicables à un point de programme donné. Notre mapping M étant purement syntaxique, nous n'introduisons ici aucune perte d'information lors de l'approximation du tissage.

Dans notre exemple distribué, la sémantique S_2 est encapsulée dans la sémantique S_1 . L'approximation de la sémantique collectrice tissée est défini dans la figure 4.

3.4.3 Problématique sur la combinaison de domaines abstraits

Les sémantiques associées aux différentes descriptions ne sont pas identiques, en général. Les domaines abstraits utilisés pour représenter les propriétés vérifiées dans un ensemble d'états du système tissé sont donc différents. Dans l'exemple présenté ici, les premiers domaines abstraits approximent les informations de flot de contrôle des processus dans les configurations ainsi que des propriétés plus globales de ces configurations tandis que les

Soit $(c^\#, f^\#)^\#, (c'^\#, f'^\#)^\#$ deux configurations tissées abstraites. Soit $\alpha \in (\mathcal{L}_p^2 \cup \mathcal{L}_p^3)$ une étiquette de transition. Soit l_b le point de programme de α décrivant la branche de comportement. Nous définissons par \tilde{y} les paramètres du message envoyé, c -à-d également les arguments de l'appel de fonction. Nous avons alors :

$$(c^\#, f^\#) \xrightarrow{1}^\alpha (c'^\#, f'^\#)$$

ssi

$$\exists c \in \gamma(c^\#) \text{ tq. } c \xrightarrow{1} c \setminus \text{removed_threads} \cup \text{new_threads}$$

alors

- $\Gamma^\# = f^\#(l_b)$
- $\Gamma'^\#, \text{result}^\# = \text{lfp}_\perp \lambda s. \{\Gamma^\#, M(l_b)(\tilde{y})\} \cup \{s' \mid (\Gamma, s) \xrightarrow{2}^\# (\Gamma', s')\}$
- $(n, \text{env}) \in \gamma(\text{result}^\#)$
- $\text{new_threads} = \text{launch}^\# \left((p_k, \text{cont}_k^n)_k, c^\#, \alpha \right)$
- $f'^\# = f^\# \sqcup \{l \in \text{new_threads}, l \mapsto \Gamma'^\#\}$

ǎ Si la transition est possible, l'élément abstrait résultat contient une partie concurrente avec une sur-approximation des transitions calculées et une partie fonctionnelle mise à jour en utilisant l'environnement global résultant du calcul du plus petit point fixe de la sémantique de la fonction associée à la transition. La primitive $\text{launch}^\#$ non spécifiée est dépendante des domaines sous-jacents. L'approximation de la partie fonctionnelle abstrait les marqueurs identités des processus et associe un environnement (fonctionnel) abstrait à chaque point de programme cap .

Figure 4: Sémantique abstraite tissée du cas distribué.

seconds domaines approximent un environnement global et une pile d'appels de fonctions. Dans la sémantique concrète du programme tissé, un état peut être vu comme un n-uplet $(e_i)_i$ où chaque e_i est un élément abstrait pour la description \mathcal{D}_i .

Lors du calcul des transitions, nous manipulons des valeurs spécifiques à la sémantique ‘‘courante’’ (cf. remarques sur les valeurs cap dans la section 3.3). Le passage d'une sémantique à une autre nécessite donc un soin particulier dans le traitement du passage de valeur d'un domaine à un autre. Un tel passage de valeur peut être représenté à la manière d'une synchronisation, dans un domaine abstrait relationnel.

La représentation abstraite d'un état du système tissé est donc un triplet (c, f, w) où c et f décrivent respectivement la partie concurrente et fonctionnelle tandis que le domaine w est un domaine des égalités entre toutes les variables du programme, domaine d'alias. La sémantique opérationnelle utilisée lors du calcul des transition peut utiliser la notion de réduction de domaine abstraits afin de calculer des informations cohérentes entre les différents domaines associés.

Pour terminer, nous pouvons remarquer qu'un tel mécanisme de synchronisation n'est pas nécessaire lorsque toutes les descriptions partagent la même sémantique, comme c'est le cas pour AspectJ.

4 Conclusion

Cet article présente deux contributions : l'utilisation d'un calcul de processus, ici un calcul d'acteur, afin de décrire l'aspect distribué d'un système ainsi qu'une première application de l'interprétation abstraite au paradigme orienté aspect. La première partie présente le calcul d'acteur CAP ainsi qu'un langage impératif et décrit comment tisser un programme distribué à partir de ces deux spécifications de natures différentes. La seconde partie est consacrée à la vérification d'un tel programme en calculant le tissage des analyses associées aux deux spécifications. Il n'existe pas aujourd'hui, à notre connaissance, de travaux dans l'état de l'art qui proposent des solutions pour les problèmes que nous avons traités ici. Le lien entre concurrence et AOP est pour l'instant principalement dans l'application d'aspects de façon concurrente sur un programme ou dans la réaction à des événements distribués mais aucun de ces travaux ne permet de spécifier, par une description séparée, la distribution d'un programme. La vérification de programme orienté aspect est également un domaine peu exploré. L'utilisation de l'interprétation abstraite afin de vérifier la sémantique collectrice du programme tissé est l'une des premières étapes vers une vérification formelle des langages orientés aspects. Dans cet article, nous avons choisi de présenter un tissage simple (syntaxique) afin de pouvoir présenter à la fois notre aspect pour la concurrence ainsi que l'utilisation de l'interprétation abstraite sur des sémantiques tissées.

Ce travail ouvre beaucoup de perspectives dans la vérification formelle de langages orientés aspects mais également dans la définition de nouveaux aspects pour des préoccupations de natures différentes, exprimées dans des sémantiques différentes. Parmi nos prochains objectifs concernant ces travaux, nous voulons étendre les types de propriétés à vérifier. L'analyse présentée ici est plus focalisée sur la partie distribuée, la partie fonctionnelle étant simplement abstraite par un domaine permettant d'approximer le flot de contrôle. Nous pouvons aisément compléter l'analyse en utilisant des domaines abstraits spécifiques à des propriétés fonctionnelles, comme des analyses de classe, de forme, ... Un autre objectif important est le traitement de tissages plus complexes afin de se rapprocher des langages comme AspectJ. Afin d'exprimer des modèles de tissage plus réaliste, il faut définir un domaine abstrait permettant de sur-approximer l'application des différents aspects dans le programme. Typiquement ce domaine abstrait approxime le modèle de tissage et doit représenter la pile des appels à un point de programme donné. L'analyse d'un programme AspectJ reviendra à analyser des spécifications ayant la même sémantique, celle de Java avec les primitives AspectJ, avec un domaine supplémentaire pour le modèle de tissage.

Pour conclure, ce travail est une application de l'interprétation abstraite à la programmation par aspects. L'interprétation abstraite est le cadre formel permettant de dériver des analyses à partir de spécification de sémantique afin de calculer de façon automatique des invariants de ces sémantiques. C'est donc le cadre théorique idéal pour la communauté AOP dans laquelle les programmes sont composés de façon systématique.

References

- [1] G. Agha. *Actors: A model of concurrent computation in distributed systems*. MIT Press, Cambridge, Mass., 1986.
- [2] Bruno Blanchet. *Escape Analysis. Applications to ML and Java(TM)*. PhD thesis, École Polytechnique, 7 December 2000.
- [3] Glenn Bruns, Radha Jagadeesan, Alan Jeffrey, and James Riely. μ abc : a minimal aspect calculus. In *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 209–224. Springer, 2004.
- [4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL'77*, pages 238–252. ACM Press, 1977.
- [5] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.
- [6] R. Douence, D. Le Botlan, J. Noyé, and M. Südholt. Concurrent aspects. In *GPCE*, Lecture Notes in Computer Science. Springer, 2006.
- [7] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD*, pages 141–150. ACM, 2004.
- [8] J. Feret. Occurrence counting analysis for the pi-calculus. In *1st Workshop on GEometry and Topology in CONcurrency Theory*, volume 39.2 of *ENTCS*. Elsevier, 2001.
- [9] J. Feret. Dependency analysis of mobile systems. In *ESOP*, number 2305 in LNCS. Springer, 2002.
- [10] J. Feret. *Analyse des systèmes mobiles par interprétation abstraite*. PhD thesis, École polytechnique, Paris, France, 2005.
- [11] P.-L. Garoche, M. Pantel, and X. Thirioux. Static Safety for an Actor Dedicated Process Calculus by Abstract Interpretation . Rapport de recherche, IRIT, décembre 2005.
- [12] P.-L. Garoche, M. Pantel, and X. Thirioux. Static Analysis of Actors: From Type Systems to Abstract Interpretation . In *EAAI*, 2006.
- [13] P.-L. Garoche, M. Pantel, and X. Thirioux. Static Safety for an Actor Dedicated Process Calculus by Abstract Interpretation. In *FMOODS*. LNCS, 2006.
- [14] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proc. of IJCAI'73*, 1973.
- [15] R. Jagadeesan, Alan Jeffrey, and James Riely. A calculus of untyped aspect-oriented programs. In *ECOOP*, 2003.
- [16] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133 – 151, 1976.
- [17] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.
- [18] P. Lam, V. Kuncak, and M. Rinard. Crosscutting techniques in program specification and analysis. In *AOSD*, 2005.

- [19] A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.
- [20] L. D. Benavides Navarro, M. Südholt, W. Vanderperren, B. De Fraine, and D. Suvée. Explicitly distributed aop using awed. In *AOSD*, 2006.
- [21] Damien Sereni and Oege de Moor. Static analysis of aspects. In *AOSD*, pages 30–39, 2003.