Preuve de propriétés de comportement deprogrammes ProActive

Rabea Boulifa, Eric Madelaine

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Preuve de propriétés de comportement de programmes ProActive

Rabéa Boulifa — Eric Madelaine

N° 4460

Mai 2002

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4460--FR

INRIA
SOPHIA ANTIPOLIS

Preuve de propriétés de comportement de programmes ProActive

Rabéa Boulifa*, Eric Madelaine†

Thème 2 — Génie logiciel
et calcul symbolique
Projet Oasis

Rapport de recherche n° 4460 — Mai 2002 — 17 pages

Résumé :

Nous étudions les méthodes de preuve des propriétés comportementales d'applications Java distribuées dans lesquelles les objets sur des sites différents coopèrent par appels distants de méthodes. La bibliothèque *ProActive* fournit un moyen simple et efficace pour mettre en oeuvre cette coopération par définition de primitives qui facilitent la programmation distribuée.

Nous présentons une méthode pour définir des modèles comportementaux, compositionnels, pour ces applications distribuées, en nous restreignant au cas sans création dynamique d'objets actifs. Nous montrons comment prouver que certaines approximations finies du modèle d'un objet actif représentent fidèlement son comportement. L'analyse des propriétés de l'application (recherche d'inter-blocage, preuves de propriétés de sûreté) peut alors être conduite, sur ces modèles finis compositionnels, en utilisant des outils de vérification automatiques basés sur une sémantique comportementale par bisimulation. Ces analyses compositionnelles ne nécessitant pas de construire l'espace d'états global, souffrent moins du problème d'*explosion de l'espace d'état* que la plupart des approches du model-checking.

Mots-clés : propriétés de programmes, vérification, modèles finis, algèbres de processus, sémantique compositionnelle, Java distribué, sécurité.

* Rabea.Boulifa@sophia.inria.fr
† Eric.Madelaine@sophia.inria.fr

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Proof of behaviour properties for ProActive programs

Abstract: We study a method for proving behavioural properties for distributed Java applications inside which objects on different setting cooperate by remote method calls. *ProActive* library provides high level primitives that ease the programming of those distributed applications.

In this paper, we give a method for constructing finite behavioural models for distributed Java applications, building the assumption without the dynamic creation of objects. We display, how by involving some finite approximation of the actif object depict closely his behaviour. Analysis of the application's properties (search of deadlock, safety properties) can be leaded on this compositionnels finite models, using verification toolsets based on behavioural semantic by bisimulation.

Those compostionnal analysis don't need building a global system, avoiding the problem of most model-checker, the *explosion states*.

Key-words: program properties, verification, finite models, processus algebra, compositional semantic, distributed Java, security.

1 Introduction

La recherche concernant les propriétés de programmes Java est actuellement très active, en particulier dans les domaines de la sécurité pour les activités de commerce en ligne (identification et authentification, confidentialité, de non-interférence), d'analyse de programmes multi-threadés, et aussi d'interaction d'applets, en particulier pour JavaCard.

Il y a moins de travaux qui s'intéressent aux propriétés comportementales d'applications distribuées sur Internet. Ces applications font un usage intensif de différents types de protocoles, entre les différentes entités d'un service Internet. Elles sont construites au-dessus de bibliothèques fournissant des primitives et des méthodes spécifiques (RMI, Jini, JCSP [15], Voyager[1], etc.). Aujourd'hui, s'il existe quelques plates-formes logicielles permettant l'analyse des propriétés fonctionnelles de programmes Java mono- ou multi-threadés, aucune n'offre des possibilités d'analyse s'adressant aux applications distribuées, prenant en compte les primitives de haut niveau fournies par les bibliothèques disponibles.

Le travail décrit ici se situe dans le contexte d'applications Java où des objets sur différents sites coopèrent par appels de procédures distants. La bibliothèque *ProActive* [8] fournit des primitives de haut niveau pour programmer ces objets actifs. Nous en définissons une sémantique en terme de systèmes de transitions étiquetées, dans lesquels les actions de synchronisation représentent les requêtes (appels de procédures) et les réponses (retours de valeurs) entre objets distants.

La construction de modèles finis pour des programmes Java (ou *ProActive*) réels est de manière générale une tâche difficile, mettant en jeu différentes techniques d'abstraction et d'approximation. Un bon exemple de ces techniques est fourni par le logiciel Bandera [9] ; il utilise des techniques sophistiquées de découpage (slicing) et d'interprétation abstraite pour construire un modèle fini analysable par des "model-checkers" classiques comme Spin et SMV. Nous proposons une approche similaire mais, pour traiter des objets distribués, nous préférons construire des modèles compositionnels basés sur les algèbres de processus. Une difficulté particulière au traitement des objets actifs asynchrones est l'utilisation de queues non bornées pour les appels distants ; *ProActive* permet au programmeur de spécifier très finement le traitement de ces queues, donc le comportement des objets. Nous définissons des méthodes spécifiques pour modéliser ces aspects.

Nos outils de vérification [11, 14] sont issus de travaux sur les algèbres de processus [5], et ont été appliqués à différents langages synchrones ou asynchrones (e.g. [13]). Ils traitent de propriétés basées sur les équivalences de bisimulation, incluant les propriétés de sûreté et de vivacité des logiques temporelles modales, et plus généralement de l'équivalence de modèles ayant différents niveaux d'abstraction. Ils évitent les problèmes d'explosion des espaces d'états en utilisant les propriétés de congruence des systèmes, et combinent, selon la nature des problèmes, des techniques d'énumération explicite ou de codage implicite (BDDs). Nous montrons comment utiliser ces outils pour analyser les modèles compositionnels des applications *ProActive*.

Dans la section 2 nous donnons une introduction sur la bibliothèque *ProActive* et sur les outils de vérification utilisées. La section 3 modélise les primitives de *ProActive* qui sont utiles pour notre travail. La section 4 introduit un exemple de programmation distribuée

du diner des philosophes et en dérive un modèle fini. Nous donnons quelques exemples de preuves de propriétés dans les sections 5 et 6. Enfin, nous concluons avec quelques directions de travail futur.

2 Contexte

2.1 ProActive

ProActive [4, 8] est une bibliothèque 100% Java pour la programmation d'applications concurrentes, distribuées et mobiles. Elle offre un modèle [6, 7, 3] de programmation parallèle et répartie que l'on peut qualifier "*à objets actifs*" (par opposition à des approches où les activités sont orthogonales aux objets), et de "*hétérogène*" dans le sens où tous les objets ne sont pas actifs. Le modèle de *ProActive* présente les caractéristiques suivantes:

- des objets actifs et accessibles à distance,
- la séquentialité des activités locales (processus purement séquentiels),
- une communication par appel de méthode standard,
- des appels systématiquement asynchrones vers les objets actifs distants,
- un mécanisme d'attente par nécessité (futur transparent),
- les continuations automatiques (un mécanisme transparent de délégation),
- l'absence d'objets partagés,
- une programmation des activités qui est centralisée et explicite par défaut,
- du polymorphisme entre objets standards et objets actifs distants.

Une application distribuée est composée d'un certain nombre d'entités appelées *objets actifs* (figure 1). Chaque objet actif a un point d'entrée unique, sa *racine*; les autres objets composant celui-ci sont des *objets passifs* et ne peuvent pas être référencés directement depuis l'extérieur de l'objet actif.

Le programmeur d'un objet actif peut contrôler de quelle façon il exécute les appels arrivant dans la queue de requêtes. La bibliothèque fournit des primitives pour examiner les requêtes présentes dans la queue, les filtrer sur leur nom ou leurs arguments, et décider lesquelles servir et dans quel ordre. Les appels de méthodes vers un objet actif sont toujours asynchrones, leurs valeurs de retour sont portées par des *objets futurs* transparents, dont la synchronisation est assurée par un mécanisme d'*attente par nécessité*. Bien que ces mécanismes soient asynchrones, il y a une *garantie de réception* des messages, et de conservation de leur ordre, implémentée par rendez-vous au moment de l'envoi de la requête.

2.2 Méthodes et outils de vérification FC2Tools

Les outils de vérification que nous utilisons sont basés sur les algèbres de processus : les modèles sont des systèmes de transitions étiquetées, et leur sémantique est paramétrée par des équivalences de bisimulations (congruences). Ces outils ne sont pas liés à un langage spécifique pour exprimer le parallélisme, mais utilisent au contraire un format générique,

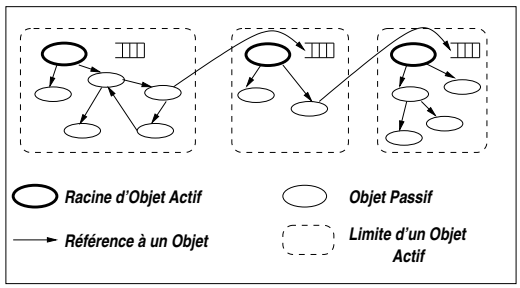


FIG. 1 – *Un graphe d'objets typique avec 3 objets actifs*

adopté par un certain nombre d'outils de vérification dans le cadre du projet européen CONCUR [11].

Ce format FC2 permet de coder des systèmes de transition finis de manière hiérarchique : un objet FC2 est un arbre dont les feuilles sont des systèmes de transitions représentés explicitement (par énumération des états et des transitions), et les noeuds sont des réseaux de synchronisation [2], qui expriment comment les comportements des composants sont combinés au niveau supérieur de la hiérarchie. Les états et les transitions sont décorés par des labels portant différents types d'information (noms et structures d'états, actions, formules logiques), ce qui nous permet de coder les informations utiles à toute une famille de sémantiques comportementales.

Définition 1 *Un* **label FC2** *d'un élément* e *est un 4-uple* $\{S,B,L,H\}$*, où* S *est la* structure *de* e*,* B *(behav) est le* comportement *de* e*,* L *est la* propriété logique *de* e*,* H *(hooks) est la liste d'*informations *de* e*, sous la forme d'un liste finie d'associations* $\{K \mapsto V\}$*,où* K *est une clé, et* V *une valeur.* S, B, L et V *sont des termes d'une algèbre de termes typés dont les opérateurs dépendent de l'application considérée ; chacun des champs d'un label peut être vide.*

Définition 2 *Un* **système de transitions FC2** $\{\mathcal{S},\mathcal{T},\mathcal{I},\mathcal{LS},\mathcal{LT}\}$ *est défini par :* $\mathcal{S}$ *un ensemble d'états,* $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{S}$ *un ensemble de transitions,* $\mathcal{I} \subseteq \mathcal{S}$ *un ensemble d'états initiaux,* $\mathcal{LS}$ *une fonction partielle associant un label aux états,* $\mathcal{LT}$ *une fonction partielle associant un label aux transitions.*

Les différents outils fondés sur ce format FC2 [11] utilisent différents types de labels pour coder les informations qui leur sont utiles. Dans le cadre de cet article, nous utiliserons

essentiellement les *comportements* des transitions et les *structures* des états, on en verra des exemples dans la suite.

Définition 3 *Un* **réseau de synchronisation FC2** $\{n,\vec{\mathcal{A}},\mathcal{T}\}$ *est défini par : n l'arité du réseau ; $\vec{\mathcal{A}}$ la structure du réseau, c'est un vecteur de longueur n décrivant l'interface des arguments du réseau : pour chacun on énumère sa* sorte, *c'est à dire l'ensemble de ses* comportements *possibles ; $\mathcal{T}$ les* transitions *du réseau, chacune est un* vecteur de synchronisation *de longueur $n+1$ dont les n premiers éléments spécifient un comportement de l'argument correspondant (éventuellement* inactif*), et le dernier est le comportement correspondant du réseau.*

Un tel réseau de synchronisation est une manière de représenter un opérateur de parallèlisme généralisé, ou plus exactement, étant donné les sortes (finies) d'un nombre fini de processus, de coder leur synchronisation.

Un système FC2 peut être construit à l'aide d'un éditeur graphique, nommé ATG. Toutes les figures de cet article ont été obtenues de cette manière; par exemple la figure 2 exprime la synchronisation entre deux objets actifs *oc* et *od*, dont chaque "canal" représente un vecteur de synchronisation, le vecteur : `Req_mi <!Req_mi, ?Req_mi` correspond à l'émission d'un message `Req_mi` par *oc* et sa réception par *od*.

Un système FC2 peut aussi être calculé par analyse statique de programmes : un système de transition code le comportement d'un sous-système (clos) du programme, alors qu'un réseau de synchronisation code un schéma de programme, un opérateur ou une expression ouverte d'une algèbre de processus. Cela a été utilisé par exemple pour des programmes CCS, LOTOS ou Esterel, et c'est ce que nous allons développer ici pour nos applications Java distribuées.

Fonctions d'analyse : La boite à outils comporte plusieurs fonctions de minimisation (par des bisimulations forte, faible, ou *branching*, modulo un ensemble d'actions visibles, ou modulo une abstraction du comportement), et des fonctions d'analyse (inter-blocage, divergence, et équivalence de systèmes). Certaines de ces fonctions utilisent une représentation explicite des systèmes, d'autres un codage implicite (BDD) de l'espace d'états. La vérification de formules de logique temporelle est réalisée indirectement à travers un codage de ces formules sous forme d'un système abstrait représentant la *spécification* du programme, et que l'on prouvera équivalent au système concret ; ou bien en codant la négation de la formule par un comportement indésirable (une action abstraite) que l'on prouvera non atteignable. Le résultat d'une analyse est en général soit un ensemble d'états (indésirables), soit un chemin d'exécution dénotant un contre-exemple. Les outils fournissent des moyens pour remonter ces diagnostiques au niveau des composants originels du système.

Lors de l'analyse d'un système avec les outils FC2Tools, on évite le plus possible de construire un système d'états global. Au contraire, chaque composant est réduit par une relation de bisimulation adéquate avant composition, à chaque niveau de la hiérarchie. Certaines des fonctions d'analyse peuvent aussi être calculées au vol sans construire l'espace

d'états complet. Dans beaucoup de cas pratiques, ceci apporte une réponse efficace au problème d'*explosion de l'espace d'état* dont souffrent toutes les approches du model-checking.

3 Modèles finis pour les applications distribuées Proactive

Nous définissons maintenant la construction de modèles comportementaux pour les applications distribuées *ProActive*. Les modèles sémantiques standards du comportement d'applications Java ou *ProActive* sont infinis. Pour pouvoir utiliser des outils de vérification, et en particulier les FC2Tools, nous devons nous ramener à des modèles finis ; de plus il est crucial de restreindre au mieux la taille des modèles construits, pour éviter les problèmes d'explosion combinatoire inhérents à ces techniques. Ceci peut être réalisé par la combinaison de plusieurs techniques, depuis la définition de domaines abstraits finis, représentant les domaines infinis que peuvent prendre les variables du programme, les techniques d'analyse de flot de données déterminant les parties de l'application influençant une propriété donnée, jusqu'à des méthodes d'approximation de structures infinies par des énumérations bornées. Le logiciel Bandera [9], par exemple, utilise une combinaison de ces techniques pour produire à partir d'un programme Java un modèle fini analysable par des *model-checkers* comme Spin ou SMV. Cela permet de vérifier des propriétés fonctionnelles de ces programmes, ainsi que des propriétés relatives au comportement des threads et de leurs verrous.

Le cas des applications Java distribuées est un peu plus compliqué à cause de la possibilité de création dynamique de processus. On peut cependant utiliser des méthodes d'analyse statique pour établir la finitude de la topologie de l'application [12], ou bien lorsque cette analyse échoue considérer des approximations finies de cette topologie. Nous faisons donc l'hypothèse dans la suite d'une topologie fixe et finie d'objets actifs. Nous définissons maintenant la construction du modèle d'une application *ProActive*, en commençant par le calcul du réseau de synchronisation des objets actifs, puis par le modèle (lui-même hiérarchique) du comportement de chacun de ces objets.

3.1 Modélisation du réseau

Nous associons naturellement un *processus* à chaque objet actif créé dans l'application. Des techniques d'analyse statique sont utilisées pour calculer cet ensemble de processus, à partir d'un *graphe d'appels de méthodes* [10], et en ajoutant l'interprétation des primitives spécifiques de *ProActive* :

ProActive.newActive (`"Type"`, *Args*, *Site*)
ProActive.turnActive (*Object*, *Site*)

Pour chaque processus, on calcule l'ensemble des services qu'il offre (*méthodes publiques*) et qu'il utilise (*appels de méthodes distantes*).

Considérons un objet actif courant *oc* et un un objet actif distant *od*. La communication entre ces objets se fait par échanges de messages (requêtes et réponses), figure 2 ;

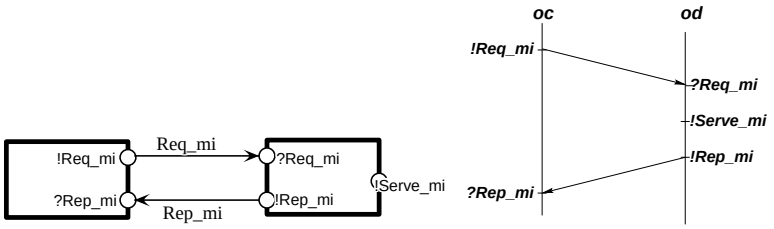


FIG. 2 – *Échange de messages entre les processus oc et od*

ProActive garantit la réception et l'ordre d'arrivée des messages entre ces deux objets actifs, quelque soit la localisation physique de l'objet distant.

Supposons que l'objet courant *oc* appelle la méthode m_i de l'objet distant *od*. Cet appel est modélisé par l'émission d'une requête $!Req\_m_i$, reçue instantanément par la queue de *od* sous le nom $?Req\_m_i$. Cette requête sera éventuellement servie par *od*. Le début d'exécution de la méthode correspondant au service de la méthode m_i :

ProActive.Serve`*("mi")`;

est modélisé par un message $Serve\_m_i$ local à *od*. Finalement, si la méthode retourne un résultat, *od* émettra un message de réponse $!Rep\_m_i$, reçu par *oc* sous le nom $?Rep\_m_i$.

3.2 Modélisation du comportement d'un objet

Le comportement d'un objet actif de *ProActive* est spécifié dans le corps de sa méthode *live* ; ce code contient des aspects *serveur* (comment les requêtes arrivant dans la queue sont servies), et des aspects *client* (calculs locaux, et requêtes à des objets distants).

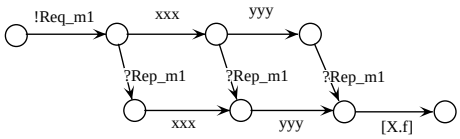
La construction du modèle (système de transition) de chacun des processus dérive de son graphe d'appels de méthodes dans lequel on ne retiendra que les appels et les retours des méthodes des objets distants.

Lorsqu'une requête est servie, elle est d'abord retirée de la queue, puis le code correspondant est exécuté. Ce code peut à son tour invoquer des méthodes d'autres objets distants, et ne terminera que dans un futur plus ou moins lointain. Dans le cas où le code d'une méthode publique ne contient pas d'invocation de méthodes distantes, nous pouvons considérer que les actions de service et de réponse de cette méthode sont contigües.

La sémantique d'*attente par nécessité* de *ProActive* spécifie qu'un message de retour peut être reçu à n'importe quel instant entre l'émission de la requête et la première utilisation du résultat, c'est à dire un accès à un champ ou une méthode de l'objet retourné, ou une attente explicite de celui-ci:

X = *od.m1* (*arg1* , ... , *argn*);
// code n' utilisant pas X
$X.f$ (*args*);
// ou explicitement "ProActive.waitFor (X);"

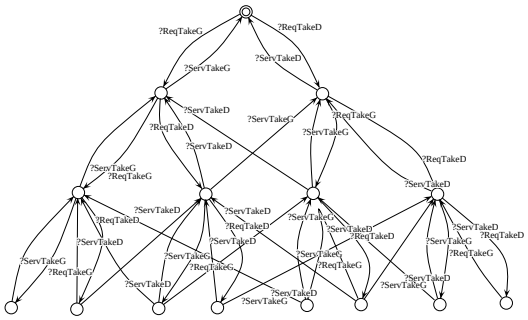
Le code entre ces deux points sera exécuté en parallèle avec l'attente du résultat, comme dans :



3.3 Modélisation de la queue de requêtes

La difficulté principale dans la modélisation des objets concerne la modélisation de la queue des requêtes. Le problème est que cette queue est a priori non bornée, donc que nous ne pouvons pas en donner un modèle générique suffisant pour analyser tout type de propriétés. Nous utiliserons des approximations finies, et essaierons d'utiliser les informations extraites du code de l'application pour optimiser ces modèles.

Un modèle général serait une queue finie de longueur n, dans laquelle chaque cellule pourrait recevoir toutes les valeurs possibles des requêtes. Ces valeurs sont les éléments du produit cartésien des différentes méthodes publiques de *oc*, par les différents objets pouvant les invoquer, par les arguments possibles de ces appels (lorsque ces arguments sont utiles pour l'analyse). Naturellement, ceci causerait une explosion exponentielle de l'espace d'états du modèle ; par exemple une queue de longueur 3 recevant 2 requêtes distinctes peut être modélisé par:



Dans le cas général, une queue de taille S recevant N messages différents peut être modélisée avec $(N^{S+1}) - 1$ états (ou S états si $N = 1$) et $\mathcal{O}(N^{S+1})$ transitions.

La méthode la plus courante en *ProActive* pour filtrer les requêtes dans la queue est d'utiliser leur nom, et de les servir dans l'ordre d'arrivée (ou l'ordre inverse), par exemple :

ProActive.ServeOldest(`"foo"`);
ProActive.ServeOldest(`"foo", "bar"`);
ProActive.ServeLatest(`"foo", "bar"`);

Les versions à arguments multiples signifient typiquement "servir la requête la plus ancienne de nom `foo` ou `bar` (et l'enlever de la queue)". Supposons que nous ayons une partition

$\{\mathcal{M}_k\}$ des noms des services fournis par notre objet *oc*, telle que tout appel de service reste à l'intérieur d'un des éléments de $\{\mathcal{M}_k\}$; alors nous pouvons modéliser séparément chaque élément de $\{\mathcal{M}_k\}$ comme une queue $\mathcal{Q}_k$, traitant des requêtes concernées, et estimer une borne supérieure pour la longueur de chacune. Le modèle complet est alors obtenu comme le produit parallèle de ces queues. Nous y gagnerons dans la mesure où nous ne calculerons jamais ce produit indépendamment de son contexte d'utilisation, c'est à dire de l'automate modélisant le comportement de *oc*.

Ceci termine la construction du modèle d'un objet actif : il suffit de synchroniser le modèle (factorisé) de sa queue de requêtes avec le modèle de son comportement propre obtenu plus tôt. On verra un exemple en figure 5.

4 Les Philosophes en ProActive

Nous illustrons notre travail par l'exemple classique du "diner des philosophes", écrit dans une version *ProActive* distribuée paramétrée par le nombre de philosophes, que nous fixons ici à 3. Chaque philosophe et chaque fourchette est un objet actif.

Le résultat de l'analyse statique du code de notre exemple permet d'identifier 6 objets actifs ce qui nous permet de construire le réseau de synchronisation suivant :

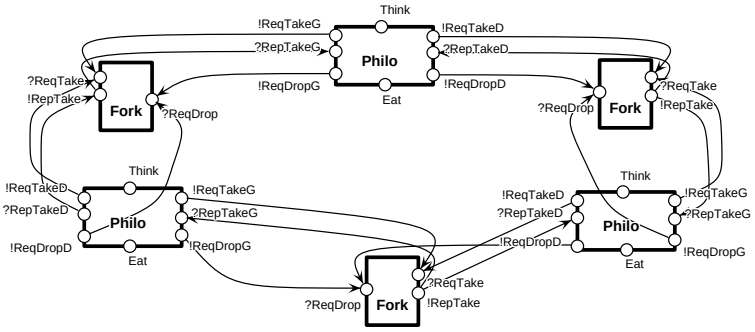


FIG. 3 – *Le réseau synchronisé Philosophes-Fourchettes*

Les objets actifs (3 philosophes et 3 fourchettes) sont représentés par des boites hiérarchiques avec des ports et la communications entre eux est représenté par des liaisons entre ces ports. La synchronisation de ces derniers est modélisée par le réseau, figure 3 ; chaque philosophe i communique avec les fourchettes voisines i et $i+1$ (gauche et droite) : par exemple, les messages `ReqTakeD` et `RepTakeD` pour les appels aux méthodes *Fork.take* qui se synchronisent sur leur retour, et le message `ReqDropD` qui lui n'attend pas de retour. Apparaissent également les actions `Think` et `Eat` qui sont simplement des événements observables, sans synchronisation.

4.1 Les philosophes

Un philosophe est un objet qui ne reçoit aucune requête externe. Son comportement est une simple boucle entre les actions : réfléchir, saisir les fourchettes, manger et déposer les fourchettes. Il peut être dans un état bloqué, en attente que les fourchettes soient libérées.

```
public void live(fr.inria.proactive.Body b)
  { while (true) {
        if (hasBothForks) {
                eat();
                putForks();
        } else {
                think();
                getForks();
        }}}

public void getForks()
  { fr.inria.proactive.ProActive.waitFor(Fork[id].take());
    display.update(id,1);
    fr.inria.proactive.ProActive.waitFor(Fork[rightForkIndex].take());
    display.update(id,2);
    hasBothForks = true;
    }

public void putForks()
  { Fork[id].drop();
    Fork[rightForkIndex].drop();
    display.update(id,0);
    hasBothForks = false;
    }
```

Le comportement d'un philosophe est encapsulé dans la méthode *live* et découle du graphe d'appel de son corps qui commence par la boucle *while (true)*. I est décrit dans la figure 4 par les actions : émission des actions visibles (`Eat` et `Think`) vers le monde extérieur ; émission de la requête `!ReqTake` aux objets fourchette (droite et gauche) ; réception du résultat de sa requête `?RepTake` correspondante ; et émission de la requête `!ReqDrop` aux fourchettes respectives (sans attente de réponse). Le philosophe ne reçoit pas de requêtes externes, il est donc inutile de modéliser sa queue de requêtes.

Comme cela est spécifié dans le code, les réponses aux requêtes `Take` sont attendues par un état bloquant ; nous pouvons optimiser le modèle du comportement du philosophe : aucune action autre que `?RepTake` n'est possible immédiatement après l'action `!ReqTake`.

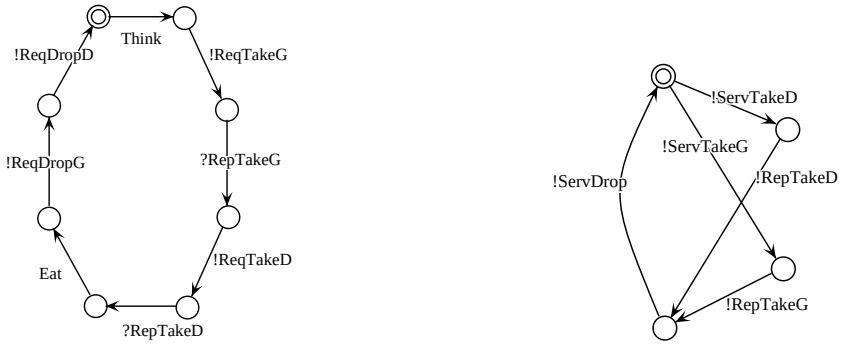


FIG. 4 – *Comportement d'un Philosophe et d'une Fourchette*

4.2 Les fourchettes

Chaque fourchette est un objet actif, qui reçoit les requêtes `Take` et `Drop` des philosophes. Son comportement est contrôlé par la variable d'état `FreeFork`.

```
public void live(fr.inria.proactive.Body myBody)
  {     while(true){
        if (this.FreeFork)
           { myBody.getService().serveOldestWithoutBlocking("take"); }
        else
           { myBody.getService().serveOldestWithoutBlocking("drop"); }
  }}

public ObjectForSynchro take()
  {     FreeFork=false;
        layout.updateFork(id,4);
        return new ObjectForSynchro()
  }

public void drop()
  {     FreeFork=true;
        layout.updateFork(id,3);
  }
```

La figure 4 donne un modèle de chaque objet fourchette. Notons que les requêtes `Take` et `Drop` ne sont pas traitées de la même manière ; en effet, `Take` retourne un résultat, celui-ci

doit être synchronisé avec l'attente d'un philosophe, tandis que `Drop` ne l'est pas et son exécution est modélisée uniquement par l'action `ServDrop` de la fourchette.

L'analyse statique produit des informations intéressantes pour la modélisation complète de la queue des requêtes des fourchettes ; les requêtes `Take` et `Drop` de la fourchette sont traitées séparément par le code utilisateur, à travers la primitive `serveOldest{<name>}` : le comportement de la fourchette est indépendant de l'ordre d'arrivée des requêtes `Take` et `Drop`. Nous pouvons utiliser un modèle dans lequel les queues séparées des requêtes `Take` et `Drop` sont simplement mises en parallèle.

Nous donnons un modèle, figure 5, dans lequel nous faisons l'hypothèse qu'à un moment donné, pas plus de deux requêtes `Take` peuvent se trouver dans la queue des requêtes `Take` et pas plus d'une requête `Drop` dans la queue des `Drop`. Cette hypothèse n'est correcte que si le système fonctionne convenablement, et nous prouverons qu'en effet c'est le cas.

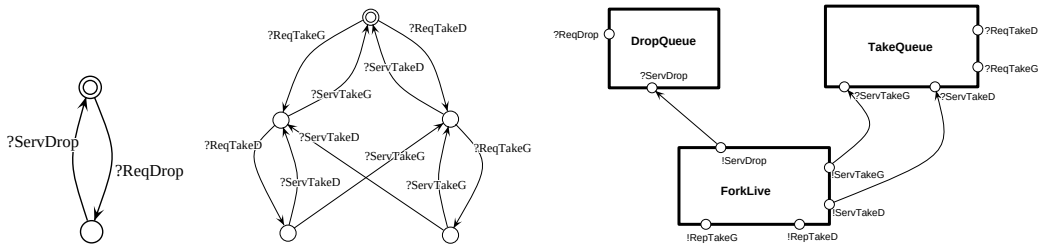
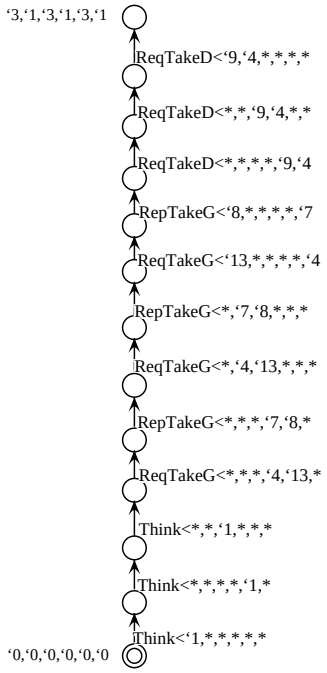


FIG. 5 – *Modèle de la queue de la fourchette : les queues de Drop et Take, l'automate de synchronisation*

5 Détection d'inter-blocages

A présent nous avons spécifié les éléments de notre système. Dans la pratique, tous ces automates et ce réseau ont été dessinés en utilisant l'éditeur ATG ; celui-ci produit également des fichiers FC2 de chaque élément. Ceux-ci sont *linkés* entre eux en un seul fichier FC2 représentant tout l'arbre.

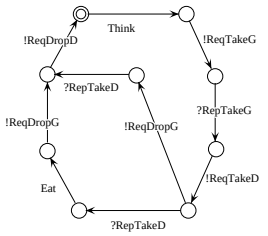


Nous exécutons maintenant l'outil de recherche de deadlock. Si des états bloqués (deadlock) sont trouvés, l'outil pourra extraire un chemin dans un format FC2 de l'un d'entre eux.

$ fc2link −main Net.fc2 Philo.fc2 Fork.fc2 > Res.fc2
$ fc2implicit −dead Res.fc2

L'outil trouve en effet un deadlock et produit un chemin que nous visualisons (voir ci-contre) à l'aide d'ATG : il mène de l'état initial (en bas) à l'état bloqué (en haut) d'où ne sort aucune transition. Un chemin est trouvé donc la transition de deadlock est possible. L'outil décore les noeuds et les arcs de ce chemin par des informations structurelles qui nous permettent d'identifier le comportement et les états de chaque comportement. Par exemple, dans le dernier état (bloqué) tous les philosophes sont à l'état 3 et toutes les fourchettes sont à l'état 1. ATG permet également d'exhiber ces éléments de manière graphique sur les composants du système.

Une solution possible pour éliminer ce deadlock est de donner au philosophe la possibilité de relâcher une fourchette avant même de saisir la deuxième :

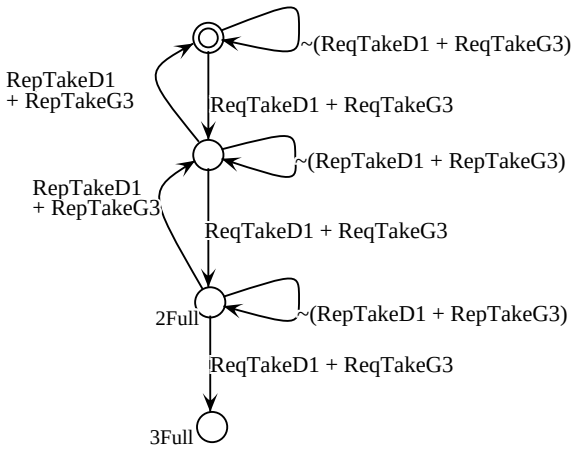


Cette fois-ci, l'analyse pour la recherche de deadlock est positive : pas de deadlock. En outre, il est possible de prouver la propriété d'équité, par exemple : "pour chaque philosophe, il est toujours possible d'atteindre l'action `Eat`".

6 Preuve de complétude

Nous allons montrer à présent que l'approximation finie du modèle de la queue des requêtes de la fourchette est correcte c'est à dire que la taille utile de la queue des requêtes `Take` est bornée et qu'il est suffisant de la considérer de longueur 2 : à aucun moment il ne peut y avoir 3 requêtes `Take` présentes dans la queue. Intuitivement, chaque fourchette ne peut être accédée que par au plus deux philosophes, et ces derniers ne peuvent envoyer plus d'une requête `Take` sans l'avoir relâchée. Naturellement, notre hypothèse ne serait pas correcte dans le cas où le système fonctionnerait anormalement ; par exemple si le code d'un philosophe était erroné.

Pour cette preuve, nous utilisons un *critère d'abstraction* de l'outil FC2 : pour chaque fourchette, nous définissons une action abstraite `3Full` comme étant une séquence d'actions qui matérialise une séquence de 3 requêtes `Take` sans service. Nous construisons l'abstraction du système global par cette action abstraite : si l'automate abstrait n'exécute jamais `3Full`, alors notre propriété est prouvée. Il nous suffit alors de réaliser cette preuve dans un modèle où la queue des `Take` est de longueur 3 et celle des `drop` de longueur 2.



La fonction d'abstraction de Fc2Tools exécute parallèlement l'automate de l'action abstraite (ci-contre) et celui du système concret pour construire un automate abstrait : une action abstraite est engendrée lorsque l'action abstraite atteint son état final (ici `3Full`).
Ici le résultat de l'analyse est un seul état bloqué, notre propriété est donc prouvée. Nous obtenons le même résultat avec la queue des `Drop`, qui n'a besoin, elle, que d'une seule place.

La modélisation de queues est stratégique en terme de complexité : les automates modélisant le comportement des méthodes utilisateurs sont linéaires dans la taille du code. Par contre les automates des queues sont exponentiels dans le nombre de messages différents pouvant être reçus.

Si la queue de la fourchette était construite brutalement, nous aurions besoin de 5 places dans la queue des requêtes (2 pour les requêtes `Drop`, 3 pour les requêtes `Take`), avec 3 valeurs possibles, i.e. 728 états. Par contre, dans le cas de la queue optimisée, nous appliquerons plutôt le produit de 2 états de la queue `Drop` et les 5 états de la queue `Take`, et nous n'avons pas à construire explicitement ce produit.

7 Autres travaux et conclusion

Il existe peu de travaux, et encore moins d'outils logiciels, permettant de faire des preuves de propriétés de comportement et de sécurité pour les applications Java distribuées.

Pour des applications Java séquentielles ou multi-threadées, nous avons mentionné Bandera [9], qui fournit un environnement sophistiqué pour construire des modèles finis analysables par des outils de model-checking. Cela permet de réaliser des preuves pour les propriétés fonctionnelles de ces applications, ainsi que dans le domaine des propriétés de

synchronisation des fils de contrôle. Ces mécanismes sont de très bas niveau, et restent dans le cadre d'applications locales à une machine virtuelle Java.

Une autre approche, plus proche des modèles formels issus des travaux sur les algèbres de processus, a mené à l'extension de Java par des primitives de communication asynchrone dans JCSP [15]. Cette librairie fournit des réseaux de processus communiquant exclusivement à travers des canaux à la CSP. Les auteurs utilisent également des outils de vérification (par exemple FDR) pour prouver des propriétés d'équivalence comportementale des processus JCSP. Ces processus sont aujourd'hui implantés localement à une JVM, bien que les auteurs promettent une implantation distribuée dans un futur proche.

Enfin il existe de nombreux travaux sur les modèles susceptibles de rendre compte de différentes notions de sécurité, d'authentification, de confidentialité, dans le cadre d'applications réparties ou mobiles. Mais il y a une grande distance entre la définition d'un modèle formel et son application au traitement d'un langage de programmation réaliste et complet. C'est dans ce cadre que se situe notre travail : nous y avons défini une méthode de construction de modèles comportementaux finis pour des applications Java distribuées construites avec la librairie *ProActive*, prenant en compte les communications asynchrones (appels de méthodes distantes) entre objets actifs potentiellement distribués sur le réseau. La construction de modèles finis met en jeu des techniques classiques d'abstraction et d'analyse statique, des éléments de modélisation spécifiques pour les primitives *ProActive*, et une analyse détaillée du comportement de la queue de requête des objets. Les modèles obtenus sont compositionnels, ce qui permet d'utiliser des méthodes de preuve adaptées à la manipulation de grands systèmes.

Nous avons illustré cette approche sur l'exemple d'école du diner des philosophes, pour lequel nous avons prouvé une borne finie de la taille utile des queues de requêtes, ainsi que des propriétés classiques d'inter-blocage.

Le domaine d'application privilégié de cette approche est celui des applications Java distribuées sur le réseau, pour lesquels nous pourrons analyser des propriétés de sureté d'algorithmes distribués, blocage ou divergence, terminaison, équivalence avec une spécification comportementale de plus haut niveau. Pour appliquer nos méthodes à des cas réalistes, il faut bien sur automatiser la construction des modèles finis, et nous avons commencé à construire une plate-forme d'analyse à cette fin. Une intégration forte entre les outils d'analyse et de vérification nous permettra de fournir des diagnostics au plus près du développeur de l'application.

Références

[1] Voyager. www.recursionsw.com, 1999.

[2] A. Arnold. *Finite transition systems. Semantics of communicating sytems.* Prentice-Hall, 1994.

[3] I. Attali, D. Caromel, and S. O. Ehmety. Formal Properties of the Eiffel// Model. In *Object-Oriented Parallel and Distributed Computing.* Hermes Science Pub., 2000.

[4] F. Baude, D. Caromel, F. Huet, and J. Vayssière. Objets actifs mobiles et communicants. *Technique et science informatiques*, 21(6–2002):1–26.

[5] J.A. Bergstra, A. Pose, and S.A. Smolka. *Handbook of Process Algebra*. North-Holland, 2001.

[6] J.P. Briot, R. Guerraoui, and K.P. Löhr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, September 1998.

[7] D. Caromel, F. Belloncle, and Y. Roudier. The C++// Language. In *Parallel Programming using C++*, pages 257–296. MIT Press, 1996. ISBN 0-262-73118-5.

[8] D. Caromel, W. Klauser, and J. Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency Practice and Experience*, 10(11–13):1043–1061, November 1998.

[9] J.C. Corbett et all. Bandera: Extracting finite-state models from java source code. *Int. Conference on Software Engineering (ICSE 2000)*, 2000.

[10] T. Jensen, F. Besson, D. Le Métayer, and T. Thorn. Model checking security of control flow graphs. *Journal of Computer Security*, 2001.

[11] E. Madelaine. Verification tools from the concur project. *EATCS Bull.*, 47, 1992.

[12] E. Madelaine and D. Vergamini. Finiteness conditions and structural construction of automata for all process algebras. In *CAV'90*, Princetown, New-Jersey, 1990.

[13] E. Madelaine and D. Vergamini. Specification and verification of a sliding window protocol. In *FORTE'91 conference*, Sydney, 1991. North-Holland.

[14] A. Ressouche, R. de Simone, A. Bouali, and V. Roy. The fctool user manuel. http://www-sop.inria.fr/meije/verification/, 1994.

[15] P. H. Welch and J. M.R. Martin. Formal analysis of concurrent java systems. In Peter Welch and Andre Bakkers, editors, *Communicating Process Architectures 2000*, Concurrent Systems Engineering, Amsterdam, September. IOS Press.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399