UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS UFR Sciences

École Doctorale STIC

THÈSE

pour obtenir le titre de

Docteur en Sciences

de l'Université de Nice - Sophia Antipolis

Spécialité : INFORMATIQUE

présentée et soutenue par

Rabéa AMEUR-BOULIFA

Équipe d'accueil : OASIS – INRIA Sophia-Antipolis - CNRS - I3S - UNSA

Génération de modèles comportementaux des applications réparties

Thèse dirigée par Eric MADELAINE

Présentée publiquement le mercredi 15 décembre 2004
devant le jury composé de :

Président	Jean-Paul Rigault	Université, Nice-Sophia Antipolis
Rapporteurs :	François Vernadat	INSA, Toulouse
	Alessandro Fantechi	Université de Florence
Examinateurs :	Thomas Jensen	IRISA, Rennes
	Radu Mateescu	INRIA, Rhône-Alpes
Directeur de thèse :	Eric Madelaine	INRIA, Sophia-Antipolis

Génération de modèles des applications réparties

Thèse de doctorat

Rabéa Ameur-Boulifa

Décembre 2004

Remerciements

Je tiens à remercier Alessandro Fantechi *et François* Vernadat *d'avoir accepté la tâche délicate d'être rapporteurs, Jean-Paul* Rigault*, Thomas* Jensen *et Radu* Mateescu *d'avoir bien voulu m'accorder de leur temps en faisant partie du Jury.*

Eric Madelaine *dont j'ai pu apprécier durant ces années de thèse les grandes qualités humaines et scientifiques, la disponibilité malgré des activités débordantes, illustre parfaitement celles-ci.*

Je remercie également tous les membres de mon équipe d'accueil (OASIS) avec lesquels j'ai passé trois années riches et agréables. Je pense particulièrement à Isabelle Attali *pour l'énergie déployée pour créer une atmosphère de travail accueillante, pour ses "autorisations" et ses relectures ; je pense également à Françoise* Baude *et Bernard* Serpette *pour leurs relectures et leurs conseils. Que le personnel de l'INRIA trouve ici l'expression de ma reconnaissance.*

J'estime tout le long de mes études avoir eu beaucoup de chance dans les personnes que j'ai rencontré. Je leur dois énormément. Il semblera peut-être fastidieux au lecteur de les énumérer mais ils s'y reconnaîtront.

Mes plus vifs remerciements à mes parents et ma famille. Je ne leur serai probablement jamais assez reconnaissante de m'avoir soutenu tout au long de mes études et d'avoir supporté mon départ si loin d'eux.

Plus près de moi il y a Amar *dont la tendresse et la patience m'a apporté la motivation nécessaire à aller jusqu'au bout et pour qui un simple merci ne suffira jamais.*

Table des matières

Table des figures

Introduction

Le succès reconnu des techniques formelles de vérification dans le domaine de la conception de circuits, des systèmes hybrides et des logiciels embarqués laissait imaginer un succès similaire dans le domaine des logiciels répartis et plus généralement des systèmes communicants. Les exemples sont nombreux dans l'aéronautique, le spatial, ou encore le commerce électronique, domaines dans lesquels les enjeux sont importants en terme de fiabilité et de sécurité.

Pourtant, il reste de nombreux défis à relever : la mise au point de tels systèmes reste délicate, à cause notamment de la non-reproductibilité des comportements. De plus, la validation et la vérification de ces systèmes se heurtent fréquemment à une explosion combinatoire qui freine sérieusement le passage à l'échelle. D'autre part, alors que les méthodologies de spécification, de conception ou d'implémentation sont communément utilisées, le degré de spécialisation des ingénieurs capables de maîtriser la complexité des modèles, des outils et des techniques de validation et vérification formelles restent un facteur de retard avéré et de coût additionnel.

L'activité de vérification a justement pour vocation d'assurer la fiabilité des systèmes et leur bon fonctionnement. Étant donné une spécification d'un système, exprimée dans un modèle ayant une sémantique formelle, il va s'agir d'examiner des propriétés comportementales de la spécification, afin de détecter des erreurs éventuelles.

C'est dans ce cadre général que se situe notre travail dans cette thèse. Notre ambition est de construire des outils de preuve formelle pour des programmes distribués. Il s'agit, en utilisant des techniques de vérification fondées sur un *modèle* (et pas sur des prouveurs de théorèmes), de fournir des méthodes automatiques pour vérifier des programmes distribués ayant une définition sémantique formelle. L'idée à terme est de fournir des outils puissants et faciles d'accès à des ingénieurs de développement.

En effet, dans le but de construire une plate-forme dédiée à l'analyse des applications Java distribuées par *model-checking*. Le challenge est de générer à partir d'une implémentation, en l'occurrence du code Java, un modèle suffisamment précis pour capturer les propriétés à vérifier et suffisamment réduit pour utiliser des outils de vérification automatiques.

La méthode de vérification fondée sur un modèle a pour objectif premier de dé-

terminer l'espace d'états du système étudié, c'est à dire l'ensemble des états auxquels le système peut accéder à partir de l'état initial. Puis, la vérification s'applique au graphe d'accessibilité : graphe représentant le comportement du système réparti dont les sommets correspondent aux états accessibles, et les arcs aux évolutions du système.

La vérification par modèle se divise en deux options :

- soit vérifier que le graphe d'accessibilité satisfait une certaine propriété, formulée dans une logique temporelle et liée à la succession des états et/ou actions dans le graphe d'accessibilité, ce qui correspond à l'évolution dans le temps du système réparti.
- soit vérifier que le graphe d'accessibilité satisfait une certaine relation, d'équivalence ou de pré-ordre, avec un graphe d'accessibilité plus petit dont on sait qu'il vérifie une propriété donnée. Nous prouvons alors, que le système étudié, "implémente" le système plus simple.

L'activité de vérification par modèle se décrit en trois phases : une phase de *modélisation* permet de produire un modèle du comportement d'un système sous la forme de graphe d'accessibilité. Une phase de *formalisation* traduit la propriété à vérifier sur le système en une formule de logique temporelle. Une phase de *vérification* permet de savoir si le graphe d'accessibilité est bien un modèle de la formule donnée, c'est à dire, si le système étudié vérifie bien la propriété qui nous intéresse.

Notre travail se situe à la phase de modélisation : proposer des formalismes et générer des modèles pour des applications réparties.

En effet, le terme modèle regroupe un ensemble de langages de spécification assez disparates suivant les outils de vérification utilisés. Parmi eux, les systèmes de transitions (simple, étiquetées, paramétrées) sont les plus connus et largement utilisés. Il est donc naturel de les choisir comme représentant du comportement de nos applications.

Nous nous intéressons à un modèle de systèmes répartis décrits sous forme d'automates communiquants par des messages envoyés et représentés par des réseaux de synchronisation. Ce modèle de distribution est relativement ancien, introduit par Nivat et Arnold, il est particulièrement utilisé dans le domaine des protocoles de communications.

L'intérêt de ce modèle est qu'il correspond à une description assez "proche" de la distribution. En effet, un système spécifié par des automates communiquants fournit un cadre d'une implémentation par des protocoles de communication asynchrones. D'autre part, proche des théories d'algèbres de processus dont les caractéristiques sont attrayantes : en particulier l'existence d'équivalences sémantiques compatibles avec les opérateurs de composition parallèle.

L'approche adoptée pour la génération du modèle global d'une application consiste à construire un modèle hiérarchique (Figure 1) pour chaque processus. En effet, à chaque fois que c'est possible le modèle d'un processus sera construit par composition des modèles des (sous)-modèles correspondants aux sous-processus dont il est formé. Cette manière com-

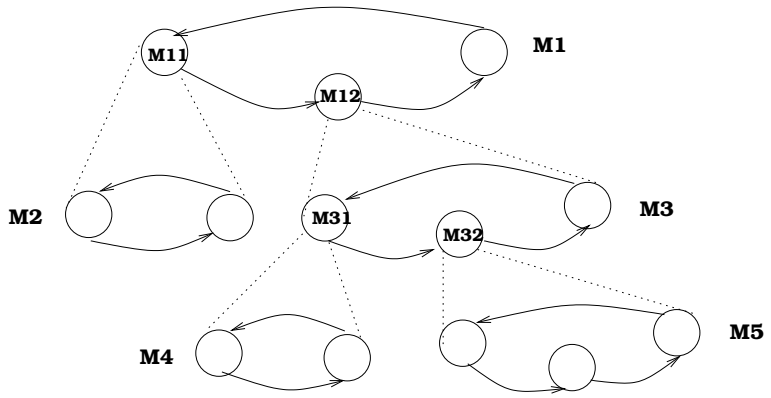


FIG. 1 – Modèle de construction hiérarchique

positionnelle de procéder permet d'asservir (même partiellement) le problème d'explosion combinatoire ; à chaque sous-modèle généré nous appliquons l'opérateur de minimisation. D'autre part, elle permet la réutilisation de modèle et donc de réduire le temps de développement d'une preuve.

Plan du document

Le chapitre1 contient un état de l'art dans le domaine de la vérification, de programmes distribuées, par modèles : les différentes sémantiques proposées pour la vérification des programmes Java ; les différents formalismes pour la représentation des programmes répartis et des définitions de base des formalismes choisis ; et enfin, quelques outils de vérification de modèles.

Le chapitre 2 décrit le contexte de notre travail. Nos modèles comportementaux sont bâtis pour des applications d'une bibliothèque Java, nommée *ProActive*, basée sur la notion d'objets actifs. L'étude de la sémantique de cette bibliothèque et de ses différents mécanismes permettant d'assurer la communication ; ce qui nous a permis de définir des règles sémantiques (opérationnelles) pour la construction de modèles et la vérification de propriétés.

Le chapitre 3 constitue la première partie de notre contribution et qui est la génération de modèles finis. Dans cette partie nous définissons, plusieurs notions, à savoir, la définition, mathématique et graphique, des modèles comportementaux retenus pour représenter une application distribuée. Puis, une définition formelle d'un graphe d'appel de méthodes pour le code source des applications *ProActive*, forme intermédiaire à partir de laquelle seront bâtis les modèles comportementaux. Dans ce chapitre nous trouvons également les règles

accomplissant cette construction et des résultats intéressants quant aux modèles générés et à la procédure de construction.

Le chapitre 4 les résultats obtenus dans le chapitre précédent, bien que, satisfaisants sont limités par la précision des événements analysés ; ils sont valides dans un univers fini et borné. Ce chapitre définit une extension des modèles vus précédemment en modèles paramétrés, pour justement prendre en compte de l'information qui jusqu'ici était ignorée. Il contient donc des définitions formelles des modèles paramétrés.

Le chapitre 5 est consacré à la définition et la description de la procédure de construction de ces nouveaux modèles à partir du code *ProActive*. Bien entendu, toutes les structures ayant servi à la construction des modèles finis sont également étendues et enrichies pour pouvoir en générer des modèles paramétrés.

Le chapitre 6 est consacrée aux illustrations ; nous utilisons trois petits exemples pour montrer les traits principaux de notre construction. Pour chacun nous commentons l'ensemble des étapes, du code source au modèle. Nous décrivons également l'état actuel des prototypes logiciels développés dans l'équipe pour supporter nos méthodes.

Chapitre 1

État de l'art

Notre travail consiste à mettre en œuvre des techniques formelles pour la vérification et validation de systèmes concurrents et distribués.

Par *vérification*, nous entendons aussi bien la spécification, la génération de modèles à partir du système de base, la validation de propriétés comportementales respectées par la spécification, ainsi que la comparaison entre la spécification et les modèles générés.

De nombreuses techniques de *modélisation* et de *spécification* existent, citons :

- les approches basées sur la logique, *theorem-proving* : théorie des ensembles et substitutions généralisées (B), logique d'ordre supérieur (COQ, PVS) ;
- les approches algébriques : algèbres de processus (CCS, LOTOS) ;
- les approches par modèles à états tels que les systèmes avec StateCharts et les Réseaux de Petri ;
- les approches synchrones : les langages à flots de données synchrones (LUSTRE).

À ces techniques correspondent des *outils* et *méthodes de vérification* :

- la construction assistée de preuves (outils COQ, PVS) ;
- la vérification automatique de modèles (outils SPIN, SMV) ;
- la génération de jeux de tests à partir des modèles formels ;
- la vérification par analyse statique.

Dans ce chapitre nous présentons un état de l'art sur le domaine récent, mais en évolution rapide, de la vérification de systèmes distribués. Nous commençons par présenter (section 1.1) les différentes propositions de sémantique formelle pour le langage Java. Dans le but de justifier le choix du formalisme adopté dans ce travail pour représenter le comportement des applications distribuées, nous présentons (section 1.2) la modélisation par des processus algébriques et ses avantages. Nous verrons également dans cette section des travaux faits pour la génération de modèles et des études de modèles paramétrés. Ensuite

(section 1.3), nous décrivons des plates-formes d'analyse de programmes ayant la même vocation que notre travail, et les différents outils de vérification et leurs caractéristiques.

1.1 Sémantique Formelle de Java et applications réparties

Formaliser un modèle d'un langage nécessite la définition et la compréhension complète de la syntaxe et de la sémantique de ce langage.

Java [30] est un langage à objets initialement conçu pour les systèmes séquentiels et embarqués, il fut par la suite largement utilisé pour le calcul réparti sur réseaux.

Par ses caractéristiques il s'est rapidement imposé sur le marché des langages de programmation. Ceci a motivé plusieurs travaux de recherche afin d'élaborer, pour divers buts, une sémantique formelle de ce langage, ou plus précisément d'un sous ensemble de celui-ci. Chaque sémantique proposée traite d'un certain nombre d'aspects du langage (code source, JVM, ClassLoader, etc...). De plus, certaines ont été élaborées juste pour formaliser un modèle mathématique de ce langage ; par contre d'autres, c'est dans le but de fournir une formalisation appropriée pour la preuve de propriétés du langage et de programmes écrits dans ce dernier.

Parmi les formalisations de la sémantique du source Java , nous trouvons les travaux de Alves-Foss et Lam [26] qui présentent une spécification en sémantique dénotationnelle, bien détaillée d'un sous-ensemble Java séquentiel (sans multi-threads). Cenciarelli [55] propose aussi une autre sémantique dénotationnelle, écrite dans un autre style (monadique), d'un sous-ensemble incluant la concurrence (multi-threads). Börger et al [40, 102] dans leurs travaux spécifient une sémantique formelle de l'ensemble du langage Java et proposent une modélisation dirigée par l'exécution, en utilisant les machines à états abstraites (ASM [41]). Le but de leur formalisation est double, d'une part, proposer un modèle mathématique prouvable automatiquement, d'autre part, valider expérimentalement la correction du compilateur et certaines propriétés de sûreté de la JVM.

Dans [32] Attali et al définissent une sémantique opérationnelle d'un sous-ensemble Java. L'objectif de leurs travaux est de générer une spécification exécutable de Java à partir d'une spécification formelle en utilisant l'outil *Centaur*.

Dans le but de prouver la propriété de sûreté de type au sein d'un sous-ensemble de Java, Drossopoulou et al [67, 66] et Nipkow [92] proposent deux sémantiques différentes d'un sous-ensembles de Java. La première spécifie un sous-ensemble de Java séquentiel, et prouvent que l'exécution des programmes préserve la notion de typage au moyen des techniques de Subject Reduction. Sémantique reprise par Syme [105] pour en faire une preuve automatique dans le système *DECLARE*. La seconde utilise le même sous ensemble pour faire la preuve de correction de propriétés conformément à des spécifications avec l'outil *Isabelle/HOL*.

Nous trouvons également les travaux de Pierce et al [76] qui proposent un calcul

de type fonctionnel pour Java, semblable au λ-calcul pour le langage ML. Et plus récemment, Henrio et al [51, 50] proposent une sémantique pour les calculs objets impératifs et les calculs objets distribués et mobiles, le calcul nommé ASP (Asynchronous Sequential Processes) qui est une extension du formalisme introduit par Abadi et Cardelli [24].

Cette liste n'est, bien entendu, pas exhaustive une étude plus détaillée des sémantiques de Java est donnée dans [74].

1.2 Modèles comportementaux des applications parallèles

Pour une description formelle et une représentation de comportement d'un système de processus parallèles, concurrents et communiquants les modèles dits *systèmes de transitions* sont les plus répandus. Parmi ces modèles, nous trouvons les *réseaux de Petri* [45] et les *algèbres de processus* [39, 71]. Ces modèles sont spécifiés par un ensemble d'états et de transitions.

Les états dans un réseau de Petri sont les marquages ; les transitions entre les états sont réalisées par tirage, simultané ou non, des transitions du réseau.

Dans les algèbres de processus , les états sont des termes et les transitions sont définies par une sémantique opérationnelle, qui décrit comment et sous quelles conditions un terme se transforme en un autre.

Il existe différents formalismes appelés "algèbres de processus" [39] parmi lesquels π-calcul [99, 89], CSP, CCS[90] et LOTOS[93] sont les plus connus. Ces théories partagent les mêmes notions clés, à savoir :

- *Modélisation compositionelle* : elles offrent un nombre réduit de constructeurs permettant de construire des systèmes complexes à partir de plus petits.
- *Sémantique opérationnelle* : elles utilisent la sémantique opérationnelle structurelle (SOS) [96] pour décrire l'exécution d'un processus. Un système représenté par ces algèbres peut être traduit naturellement en système de transitions étiquetées.
- *Analyse de comportement par des équivalences ou par pré-ordre* : ces théories utilisent les relations comportementales comme moyen de combinaison des différents systèmes décrits dans cette algèbre. En général, ces relations sont des équivalences qui encapsulent la notion du "même comportement", ou des pré-ordres qui elles encapsulent la notion de "raffinement".

Deux avantages, au moins, découlent de ces caractéristiques :

- *Composition* : un modèle peut être hiérarchisé et sa construction peut donc se faire graduellement.
- *Substitution* : dans un modèle global d'un système, un processus peut être remplacé par un autre ayant le même comportement. Ce qui convient bien pour l'analyse

modulaire de systèmes.
– *Minimisation* : un modèle de processus peut être réduit avant analyse, grâce à la relation d'équivalence. Ce qui peut réduire considérablement la taille des modèles.

La vérification de système d'une algèbre de processus consiste à écrire deux spécifications ; d'un côté une spécification (`Sys`) décrivant le comportement effectif du système ; de l'autre une spécification (`Spec`) décrivant une abstraction donnée de celui-ci. Puis à établir la correction de `Sys` par rapport à `Spec` soit en montrant (par équivalence) que `Sys` se comporte de la même manière que `Spec`, ou en montrant (par pré-ordre) qu'il le raffine.

Il existe deux procédés différents pour montrer la correction d'une spécification par rapport à une autre : le procédé orienté syntaxe et le procédé orienté sémantique. Dans le premier cas une axiomatisation d'une relation comportementale (choisie) est utilisée pour montrer qu'une expression se transforme syntaxiquement en une autre ; dans le second cas, en utilisant directement la définition de la relation et la sémantique de deux expressions pour montrer leurs liens.

Dans certains cas, par exemple lorsque les spécifications sont finies, la vérification, basée sur la syntaxe ou basée sur la sémantique, peut être réalisée automatiquement.

1.2.1 Génération de modèles

Parmi les travaux s'intéressant à l'étude de la modélisation des programmes distribués, les premiers sont consacrés essentiellement à une modélisation par réseaux de Petri [72] de programmes Ada [88] ou encore de programmes décrits dans des formalismes distribués dédiés [79].

Parmi les plus proches à savoir ceux s'intéressant au programmes Java concurrents et communiquant par le mécanisme (RPC), par appel de méthodes distantes. Nous trouvons les travaux de Corbett [59] décrivant une technique de construction de modèles compacts et finis de programmes Java. Afin de réduire la taille des modèles d'application concurrents. Son approche s'appuie sur l'algorithme de flot de données pour construire une approximation de la structure dynamique d'un programme dans le tas.

Demartini et Sisto dans [65] décrivent une technique de modélisation de programmes concurrents Java en code Promela pour une analyse et étude de problèmes d'atteignabilité. L'analyse d'atteignabilité requiert que le modèle à analyser soit fini, ils la restreignent donc à l'analyse de programmes dont le code est sans récursion et dont les variables sont accédées par exclusion mutuelle.

Naumovich et al dans [91] adaptent la technique d'analyse de flot de données FLAVERS, dédié aux systèmes concurrents communiquant par rendez-vous décrit au départ pour le langage Ada, à la vérification de propriétés de programme Java concurrent. La démarche consiste à construire un modèle du code en émettant des contraintes de faisabilité.

1.2.2 Modèles paramétrés

Plusieurs systèmes sont définis comme étant des systèmes paramétrés par le nombre d'instances de processus identiques. Les processus sont généralement d'états finis donc chaque instance est finie, mais le nombre d'instances peut être infini donc le système serait à états infinis. Un système paramétré peut être formulé comme suit : soit S_i un système de taille i ; $\mathcal{S} = \{S_i\}_{i=1}^{\infty}$ une famille de systèmes S_i.

Le problème de vérification de ces systèmes par model-checking, PMCP (parametrized model-checking problem), consiste à déterminer si une propriété donnée ϕ est vérifiée sur un nombre quelconque d'instances de $\mathcal{S}$, i.e., si tous les S_i de $\mathcal{S}$ vérifient ϕ.

Bien qu'en général, le problème de vérification d'un système paramétrés de processus de taille du nombre d'instances inconnu (càd infinie) est prouvé indécidable [27], beaucoup de travaux sont faits dans le but d'identifier des sous-classes de systèmes paramétrés et décidables. Les deux principales directions sont :

- classe de systèmes composés de processus identiques (semblables) ; nous trouvons par exemple les travaux de Emerson et Namjoshi [69] et les travaux de Sistla [73] propose une approche complètement automatique de model-checking paramétrés de systèmes synchrones ayant un seul processus de contrôle (serveur) et un nombre arbitraire de processus utilisateurs (clients). Approche utilisée pour prouver des propriétés de sûreté de programmes, C [34] et Java [64], multi-threadés.
- classe de processus liés par une relation de pré-ordre [80], de sorte que si un processus vérifie une propriété donnée le processus "raffiné" la vérifie aussi. Cette approche est étendue à une approche appelée *network invariant* [83, 109] qui consiste à trouver un processus (invariant) satisfaisant les hypothèses d'inductions.

1.2.3 Systèmes de transitions

La manière la plus répandue de spécifier les différents formalismes de systèmes répartis consiste à associer à chaque spécification un *système de transitions* (simple, étiquetés ou paramétrés). Nous donnons dans cette section les définitions, ainsi données par Arnold [28], de cette notion. Par la suite, nous étendrons ces notions pour prendre en compte nos systèmes hiérarchiques et paramétrés.

Définition 1 Système de transitions étiquetées. *Un système de transitions étiquetées par un alphabet A est un 5-tuple :*

$$\mathcal{A} =< S, T, \alpha, \beta, \lambda >$$

où

- *S est un ensemble fini ou infini d'états,*

- *T est un ensemble fini ou infini de transitions,*
- *α et β sont deux mapping de T vers S qui associe à toute transition t de T deux états $\alpha(t)$ et $\beta(t)$, respectivement la source et la cible de la transition t.*
- *λ est un mapping de T vers A associant chaque transition t à son étiquette $\lambda(t)$.*

Un système de transitions est paramétré en ajoutant des paramètres aux états et aux transitions.

Définition 2 Système de transitions paramétré. *Un système de transitions paramétré par $(\mathcal{X}, \mathcal{Y})$, où*

- *$\mathcal{X} = \{X_1, \ldots, X_n\}$ est un ensemble fini de noms de paramètres d'états et*
- *$\mathcal{Y} = \{Y_1, \ldots, Y_n\}$ est un ensemble fini de noms de paramètres de transitions,*

est un système de transitions $< S, T, \alpha, \beta >$ les sous-ensembles S_X de S (paramètres d'états) et T_Y de T (paramètres des transitions) sont définis pour tout X de $\mathcal{X}$ et tout Y de $\mathcal{Y}$.

Le *produit de synchronisation* permet d'obtenir le système de transition d'un système composé de processus communiquants à partir des systèmes de transitions de chaque composant.

Définition 3 Le produit de systèmes de transitions. *Considérons n systèmes de transitions $\mathcal{A}_i =< S_i, T_i, \alpha_i, \beta_i, \lambda_i >$, $i = 1, \ldots, n$. Le produit $\mathcal{A}_1 \times \ldots \times \mathcal{A}_n$ de ces systèmes de transitions, est un système de transition $\mathcal{A} =< S, T, \alpha, \beta, \lambda >$ défini par :*

$$\begin{aligned} S &= S_1 \times \ldots \times S_n, \\ T &= T_1 \times \ldots \times T_n, \\ \alpha(t_1, \ldots, t_n) &= < \alpha_1(t_1), \ldots, \alpha_n(t_n) > \\ \beta(t_1, \ldots, t_n) &= < \beta_1(t_1), \ldots, \beta_n(t_n) > \\ \lambda(t_1, \ldots, t_n) &= < \lambda_1(t_1), \ldots, \lambda_n(t_n) > \end{aligned}$$

Le produit de synchronisation peut également être défini par des vecteurs de synchronisation.

Définition 4 Vecteurs de synchronisation. *Soient $\mathcal{A}_i$, $i = 1, \ldots, n$, n systèmes de transitions étiquetées sur l'alphabet A_i, et soit $I \subseteq A_1 \times \ldots \times A_n$ une contrainte de synchronisation, le produit de synchronisation des $\mathcal{A}_i$ sous I, noté $< \mathcal{A}_1, \ldots, \mathcal{A}_n; I >$, est un (sous)-système de transitions ayant des transitions $< t_1, \ldots, t_n >$ dont les vecteurs d'étiquettes $< \lambda_1(t_1), \ldots, \lambda_n(t_n) >$ sont des éléments de I.*

1.3 Vérification de modèles

La vérification de programmes par la technique de vérification de modèles (*model-checking*) [56], consiste à *vérifier* si une certaine propriété est satisfaite par un état (ou

des états) du *modèle* d'un système. Les systèmes les plus appropriés pour la vérification par cette technique sont ceux qui sont aisément modélisés par des automates (finis) : des machines évoluant d'un *état* à un autre sous l'action de *transitions* ; et les propriétés à examiner sont généralement des formules de la logique temporelle [38].

1.3.1 Plate-forme d'analyse et vérification de programmes

Il existe plusieurs plates-formes dédiés à l'analyse et la vérification d'applications par la technique de model-checking. Il est intéressant de distinguer entre les grandes familles :

- Analyse de programmes C : Slam [18] (Software (Specifications), Languages, Analysis, and Model checking) est un "outil" développé par Microsoft, initialement conçu pour la validation des propriétés des drivers Windows XP, il fut par la suite utilisé pour la vérification de propriétés de sûreté de programmes C, séquentiels. Il analyse, essentiellement, des programmes admettant que des variables de types booléens à valeur de retour multiples. La vérification se base sur une technique d'abstraction par prédicats [100] : afin de montrer qu'une propriété (de sûreté) est satisfaite par un système, il suffit de montrer qu'elle est satisfaite par une abstraction du système. À partir d'un ensemble de prédicats, invariants du système, fournis par l'utilisateur l'abstraction et en exploitant les contre-exemples générés par l'étape de model-checking, une abstraction originelle est raffinée. Ainsi l'analyse se poursuit par une boucle automatique abstraction-model-checking-raffinement, jusqu'à déterminer si le système satisfait la propriété.
  Un autre "outil" Blast [3] (Berkeley Lazy Abstraction Software Verification Tool) outil similaire à Slam à savoir le procédé de vérification effectué par des abstractions et des raffinements guidé par contre-exemple et il est également utilisé pour la vérification de propriétés comportementales de programmes C. En revanche, il diffère principalement par : à chaque étape de la boucle, il réutilise le résultat de la boucle précédente. Les abstractions ne sont pas systématiques (pour tous les systèmes) mais juste où elle est requise et par utilisation de prédicats addionnels.
  Modex/FeaVer [14] est un autre outil pour la vérification de programmes C séquentiels. Il est composé d'un module d'extraction automatique de modèle à partir du code source C, Modex (Model Extractor) ; et d'un module, FeaVer, constitué d'une interface utilisateur et du model-checker SPIN.
- Analyse de programmes Java : Bandera [1] fournit une plate-forme pour la construction et l'analyse automatique de programmes Java concurrents par model-checking. Il permet la vérification de propriétés de sûreté et de vivacité, utilisant des outils existants (SMV, SPIN), analyse des programmes relativement larges, les contre-exemples sont traduits en code source, utilise une interface graphique.
  Très proche de Bandera, JavaPathFinder [12] est une plate-forme développée par la NASA pour la détection de deadlock ou d'états pouvant y mener.
  Zing [23] outil développé par Microsoft pour validation de divers types de logiciels : les spécifications de protocoles et des flots de contrôle, les web services. Cet outil a

été utilisé pour l'analyse de programmes concurrents et communiquants par appels de méthodes. Un des points forts de cet outil est la garantie que sous certaines hypothèses l'algorithme d'analyse termine même en présence de récursion et de concurrence [97].

– Analyse d'applications distribués asynchrones : le projet BEHAVE ! [2] développé également par Microsoft pour l'analyse et la vérification de programmes concurrents, asynchrones, communiquants par échange de messages. Afin d'analyser les scénarios des applications décrites en π-calcul, cet "outil" utilise le système de typage pour extraire de cette application des abstractions comportementales, dites types comportementaux. La vérification est alors exécutée par les règles d'inférence sur ces types.

Toutes ces plates-formes utilisent divers outils de vérification pour valider les propriétés étudiées.

1.3.2 Outils de vérification

Il existe plusieurs outils [5, 38] logiciels, appelés *model-checkers*, permettant de réaliser cette vérification. Ces derniers diffèrent principalement par les modèles des systèmes qu'ils acceptent en entrée et par le formalisme des propriétés à vérifier. Nous trouvons des outils pour la vérification de systèmes finis, tels que SMV, SPIN, JACK, CADP, Esterel, Murphi et FcTools ; d'autres pour la vérification de systèmes infinis, acceptant en entrée des automates temporisés ou paramétrés tels que Trex, UPPAAL, HyTech, Kronos et Murphi ; et enfin ceux analysant des réseaux de Pétri : DESIGN/CPN et CADP. Donnons une description succincte de chaque outil :

– *SMV* (Symbolic Model Checking) [19] accepte en entrée un réseau d'automates qui coopèrent par variables partagées. Il permet la composition d'automates et la construction hiérarchique. Les automates sont décrits textuellement et les propriétés à vérifier sont exprimées en CTL. Dans le cas où la propriété analysée n'est pas satisfaite, SMV fournit un contre-exemple.
– SPIN [21] : le modèle d'entrée d'un système à analyser avec SPIN est décrit en *Promela* un langage semblable au langage C enrichi par des primitives de communication. Ainsi, Promela permet de décrire le comportement de chaque processus d'un système, ainsi que les interactions entre eux. La communication entre les processus est FIFO à travers des canaux, par rendez vous ou par variables partagées. SPIN permet la simulation de systèmes et la vérification de propriétés exprimées en LTL.
– DESIGN/CPN [7] permet d'éditer, simuler et vérifier des réseaux de Petri colorés. L'édition et la simulation peuvent être faites graphiquement. Les systèmes pouvant être infinis, cet outil offre à l'utilisateur un moyen de fournir un critère d'arrêt. Les propriétés quant à elles sont définies en utilisant le langage ML.
– UPPAAL [22] permet l'analyse de réseaux d'automates temporisés. Grâce à un éditeur et un simulateur graphique le comportement d'un système peut être exécuté

par des séquences de transitions. Par contre, seules les propriétés d'atteignabilité peuvent être analysées par cet outil.

- Kronos [13] model-checker pour la vérification de propriétés exprimées en TCTL sur des automates temporisés et décrits dans une forme textuelle.
- HyTech [10] est un outil pour l'analyse des applications modélisées par des automates linéaires hybrides – automates dotés de variables globales et dont les valeurs s'accroissent linéairement avec le temps. Cet outil est dédié à des analyses paramétrées dont le modèle est fourni sous forme textuelle.
- JACK [11] (Just Another Concurrency Kit) offre un environnement de vérification de propriétés sur des modèles d'applications concurrentes ; ils comportent deux model-checkers AMC and FMC. Le premier dédié à la vérification des propriétés ACTL sur des automates. Le second à la vérification des propriétés μ-ACTL sur des réseaux.
- CADP [4] (CAESAR/ALDEBARAN Development package) est une boîte à outils pour la technologie des protocoles. Il offre un éventail de fonctionnalités, de la simulation interactive aux techniques de vérification formelles les plus récentes. Il est consacré à la compilation, à la simulation, à la vérification formelle, et à l'essai des descriptions écrites dans la langage LOTOS. Il inclut des outils de vérification de modèles (Model checking). Cet outil support différents langages de formalisation : algèbre de process, machines à états finis et des réseaux de machines à états finis communicantes des réseaux de Petri.
- Esterel [8], le langage Esterel dédié à la programmation de systèmes réactifs, inclut dans sa boite à outil, en plus du compilateur traduisant les programmes dans des langages synchrones tels que Lustre ou SynChart, un outil de vérification, *Xeve*. Cet outil doté d'un environnement graphique permet une analyse symbolique de programmes Esterel représentés par des machines d'états finis.
- Murphi [15] est un système de vérification incluant un langage de description et un compilateur. Le compilateur génère à partir d'une description donnée un vérificateur spécifique pour analyser des propriétés. Le langage de description permet de décrire des systèmes concurrents, asynchrones, à états finis par des structures riches (variables globales, procédures, types...) et des règles de transitions gardées par des expressions booléennes et des actions de modifications des variables globales.
- FcTools [9] boîte à outils permettant l'analyse et la vérification de réseaux de processus communiquants. À l'instar de CADP, il offre plusieurs fonctionnalités de la simulation à la vérification formelle. En outre, il offre, grâce au module d'édition de fichiers, la possibilité d'une analyse compositionnelle et hiérarchique du système étudié.

1.4 Discussion

Il est maintenant clair que les automates communiquants sont de bons candidats pour spécifier les systèmes répartis. Par leur sémantique permettant la spécification des processus asynchrones, où la séquence d'événements peut être définie sans se soucier du moment de leur apparition, et des processus synchrones, où chaque événement est lié à une horloge. Et par leur qualité de pouvoir combiner des automates pour en générer d'autres.

En effet, la construction, des modèles comportementaux des applications que nous étudions, est menée de manière compositionnelle et hiérarchique afin, d'une part, d'asservir (au moins partiellement) le problème d'explosion d'états, d'autre part, de réutiliser des composants dans d'autres constructions et réduire ainsi le cycle d'analyse d'une application.

La génération est faite à partir de la sémantique opérationnelle du langage de description de ces applications qui est des algèbres de processus, d'où bien adaptée à ce propos.

Certains model-checkers acceptent en entrée et génèrent en sortie des descriptions de systèmes données dans un format appelé *fc2*. Ce format est un langage de description d'automates et de réseaux commun à plusieurs model-checkers, par exemple JACK, CADP, Esterel et Fc2tools, mais également à différents outils graphiques tels que Autograph [98]. Ce qui offre l'avantage de pouvoir combiner et connecter différents modules et fonctionnalités de multiples outils de vérification, en plus, de l'avantage de visualisation par interface graphique des automates résultants de l'analyse.

Chapitre 2

Bibliothèque ProActive

Parmi les langages de programmation intégrant des mécanismes de distribution, Java est le plus connu et largement utilisé (par exemple JavaRMI). *ProActive* [37, 53, 17] est une bibliothèque 100% Java pour la programmation d'applications concurrentes, distribuées et mobiles. Nous avons choisi cette bibliothèque comme illustration de notre cadre d'étude pour ses caractéristiques intéressantes dont les principales sont : la transparence de la communication et de la synchronisation, sa portabilité sur n'importe quelle plate-forme Java, et la possibilité d'utilisation des outils associés (Javac et de chargement de classes) sans aucune modification de l'environnement, ni de la machine virtuelle, et sans utilisation de pré-traitement ou de compilation spécifique.

Dans ce chapitre, nous présentons donc le modèle de base de cette bibliothèque : la structure d'une application *ProActive* et le principe de communication, par envoi et réception de requêtes (Section 2.1). Et ensuite, celui de la synchronisation des entités réparties sur différents sites (Section 2.2). Et enfin, (Section 2.3) nous verrons comment sont servies et traitées les requêtes par le serveur.

2.1 Modèles de programmation

Le modèle de distribution et d'activité de *ProActive* est basé sur la notion d'"*objets actifs*" (objet avec une activité) et d' "*hétérogénéité*" dans le sens où tous les objets ne sont pas actifs mais passifs, càd objets standards Java. Il a été étudié et appliqué à plusieurs domaines aussi bien pratiques que théoriques [46, 49, 31, 52].

2.1.1 Objet Actif

Un *objet actif* est l'unité de base de distribution d'une application *ProActive*. Un objet actif a son propre fil de contrôle (activité) : une mémoire locale et un comportement

spécifique qui gère les appels de méthodes, les stocke dans sa queue de requêtes et décide de l'ordre de leur service. Contrairement à Java standard, le programmeur en ProActive n'a pas à manipuler explicitement les threads pour gérer les échanges.

Un objet actif peut être créé sur n'importe quelle machine hôte. Une fois que celui ci est créé, son activité et sa localisation (local ou distant) sont complètement transparents, si bien qu'il est manipulé comme un objet passif standard.

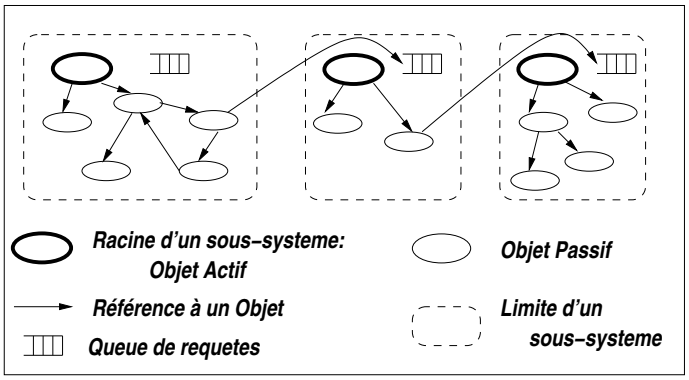


FIG. 2.1 – Graphe d'objets avec objets actifs

En *ProActive* une application distribuée est composée d'un certain nombre d'entités appelées *objets actifs* (figure 2.1). Elle est structurée en sous-systèmes (activités), pouvant être situés sur des machines différentes, coopérant les uns et les autres :

- Chaque sous-système a un point d'entrée unique, sa *racine* qui est un objet actif ; les autres objets composant celui-ci sont des *objets passifs*. L'objet actif d'un sous-système ne peut exécuter que ses propres méthodes (celles de l'objet actif lui même et celles des objets passifs lui appartenant) ;
- Il n'y a pas de partage d'objets passifs entre sous-systèmes.

Ces caractéristiques ont des conséquences importantes sur la topologie des applications et sur la procédure de communication entre les sous-systèmes :

- De tous les objets composant un sous-système, il n'y a que l'objet actif qui est connu de l'extérieur de celui-ci ;
- Tous les objets (actifs et passifs) peuvent avoir une référence sur un objet actif distant ;
- Si un objet O_1 a une référence sur un objet passif O_2 alors O_1 et O_2 appartiennent au même sous-système ;

- Appel par valeur, les arguments d'appel vers des objets actifs sont passés par copie profonde ;
- La syntaxe de communication se réduit aux appels de Java standard ;
- La sémantique de communication est indépendante de la distribution et donc de la localisation des objets.

D'un point de vue de la structure interne, un objet actif est composé de plusieurs méta-objets dont : le *body* et le *proxy* .

- *Body* est le point d'entrée d'un objet actif la partie accessible à distance. Il a la charge de régenter les communications : reçoit les appels distants destinés à l'objet, les range dans la queue, les retire ultérieurement selon une politique de service pour que l'objet exécute le code. Il assure également la coordination du travail des autres méta-objets.
- *Proxy* est un méta-objet qui sert à maintenir des références sur les appels d'objets actifs ; il intercepte les appels de méthodes, les transforme en des *requêtes* par ajout de paramètres (adresse de l'appelant et du destinataire) et génère les objets futurs qui sont les promesses de résultat.

Le comportement d'un objet actif de *ProActive* est spécifié explicitement par le programmeur dans le corps de la méthode principale *runActivity* associée au body ; ce code contient des aspects *serveur* (comment les requêtes arrivant dans la queue sont servies), et des aspects *client* (gestion des calculs locaux, et requêtes à des objets distants).

2.1.2 Création d'objets actifs

Étant donnée par exemple une classe A. La création d'une instance passive de A se fait grâce à l'instruction de Java standard :

```
A a = new A(4,"string");
```

par appel au constructeur de la classe A avec les paramètres effectifs, par exemple l'entier 4 et la chaîne "string".

Par contre, la création d'instances actives peut se faire de trois manières différentes grâce à des primitives spécifiques :

- La création *basée instantiation* par l'instruction :

  ```
  A a = (A) ProActive.newActive ("Type", Args, Site);
  ```

  crée une nouvelle instance *Active* à partir de la classe standard et existante nommée *"Type"* ; le second paramètre de cet appel *Args* correspond aux paramètres effectifs du constructeur de cette classe. Et le dernier paramètre optionnel *Site* spécifie la machine (JVM) hôte où doit être placé cet objet. Dans le cas où celui ci est *null*, la création se fait sur la JVM courante.

– La création *basée classe* par le code :

```
class pA extends A implements Active{...}
...
A a = (A)ProActive.newActive ("pA", Args, Site);
```

crée une nouvelle instance *Active* à partir d'une nouvelle classe *pA* qui implémente l'interface *Active* et hérite (éventuellement) d'une classe existante *A*.

– La création *basée objet* par l'instruction :

```
a = (A) ProActive.turnActive (a, Site);
```

prend un objet déjà créé (ici l'objet *a*) et en fait un objet actif accessible à distance. Le paramètre *Site* a bien sûr le même rôle que précédemment.

2.1.3 Communication entre objets actifs

Les objets actifs sont des entités autonomes s'exécutant en parallèle, concurremment, et communiquant entre eux par échanges de messages de manière synchrone ou asynchrone. La communication se fait par le protocole d'appel de procédure à distance, RPC (de l'anglais Remote Procedure Call) [103], qui permet des appels de méthodes à travers un échange de messages en s'abstrayant complètement du réseau sur lequel s'effectue cette communication. Lors d'un appel de méthode sur un objet distant, une requête est envoyée automatiquement sur le réseau, le protocole RPC se chargeant d'encoder les arguments d'appels (notamment l'adresse du source et du destinataire) et la valeur de retour ;

Les appels de méthodes vers les objets actifs distants ont la syntaxe des appels Java standards :

```
ro.m (arg1, ... , argn);
```

Un appel en *ProActive* est systématiquement asynchrone, la synchronisation se fait grâce au mécanisme d'attente par nécessité à travers les objets futurs. Cette attente est a priori implicite, le résultat de l'appel n'est requis qu'au moment de son utilisation tel qu'illustré par le code suivant :

```
X = ro.m(arg1,...,argn);
    xxx; yyy; // du code n'utilisant pas X
X.f(args);
```

la valeur de la variable X n'est requise qu'au moment d'invoquer la méthode f.

Néanmoins, cette attente peut être explicite en utilisant la primitive spécifique :

```
waitFor(ro.m(arg1,...,argn));
```

qui attend expressément le retour du résultat de l'appel *ro.m(arg1,...,argn)*.

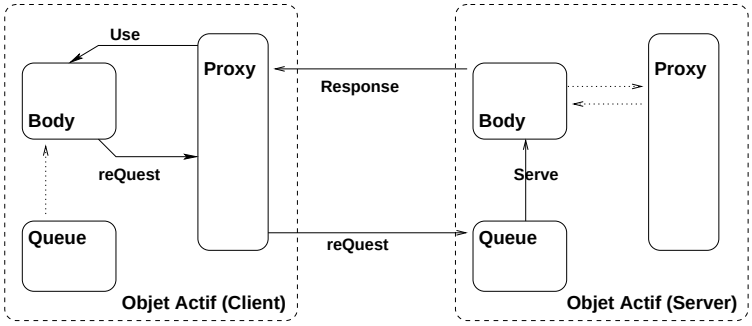


FIG. 2.2 – Communication entre deux objets distants

La communication entre un objet local et un objet distant se fait en deux étapes : l'envoi de la requête, et après le traitement de celle-ci par l'appelé, envoi d'une réponse. Le déroulement de la communication est décrit par les étapes suivantes :

1. l'appelant qui veut communiquer envoie une requête qui est interceptée par le *proxy*,
2. ce dernier créé un objet appelé *futur* qu'il ajoute au *pool* des futurs,
3. la communication réseau s'effectue directement avec le *body* distant,
4. une fois reçue, la requête est directement placée dans la *queue* pour attente de service,
5. avant de poursuivre son exécution l'appelant reçoit auparavant un accusé de réception de l'appelé,

   l'atomicité des opérations 1 à 5 est garantie par le mécanisme de **rendez-vous**
6. suivant une politique de service, la requête est retirée de la queue et exécutée,
7. dans le cas où une réponse est attendue, le résultat de l'exécution est transmis au proxy de l'appelant,
8. qui remet à jour l'objet futur.

2.2 Asynchronisme et Futurs

Dans *ProActive* la communication se fait de manière asynchrone, un objet courant envoie une requête (un appel de méthode) à l'objet distant, qui, une fois la méthode exécutée lui enverra une réponse. Ce mode de communication permet à l'objet ayant émis l'appel de poursuivre son exécution et de récupérer les résultats ultérieurement.

À chaque appel de méthode, un objet futur est créé par le proxy. Cet objet joue le rôle de "garde-place" pour le résultat de l'appel non encore exécuté. Quand une réponse est envoyée par l'appelé, l'objet futur est mis à jour automatiquement. Grâce à la notion de futurs, l'appelant peut continuer son activité tant que le résultat n'est pas requis, auquel

cas il se bloquerait jusqu'à la mise à jour de l'objet futur (retour de réponse) : c'est le mécanisme d'*attente par nécessité*.

Pendant le cycle de vie d'un objet actif, plusieurs futurs peuvent être créés, autant de futurs que d'appels distants. À chaque création d'un futur, celui ci est inséré dans un tableau dit *pool* de futurs de l'objet actif correspondant. Dès que le résultat est disponible, le futur est mis à jour et retiré du *pool*.

L'objet futur est associé à un point d'appel de méthode distante qui est son point de *Définition* . Et il n'est réclamé qu'au moment de son exploitation, au point d'*Utilisation* ; ces points notés respectivement *Def* et *Use* sont définis par :

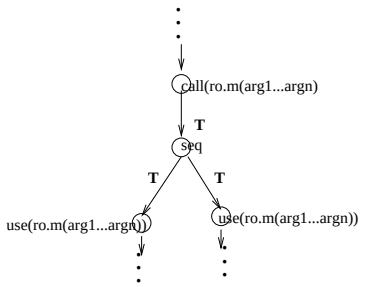
Définition 5 Points Def-Use de futur.

- *Un point de* **Def** *d'un futur est un point d'affectation de la valeur résultant d'un appel de méthode distante à une variable. Par exemple,* $fut = ro.m(arg_1, \ldots, arg_n)$ *;*
- *Un point de* **Use** *d'un futur est un point de programme où la variable de ce futur est requise en ce point. Par exemple, l'accès à un attribut de l'objet fut.*

Ces points que nous reverrons ultérieurement s'avèrent cruciaux pour notre analyse et modélisation, il y a donc besoin de les définir et les calculer.

En fait, un futur est requis au moment de sa première utilisation autrement, avant cela il est ignoré, après il est supposé disponible. Cependant, à un point de définition peuvent correspondre plusieurs points d'utilisation. Par exemple dans le fragment du code ci-dessous, au point de définition $ro.m(arg1, .., argn)$ correspondent deux points d'utilisation différents $X.a++$ et $X.a--$ représentés, dans la partie du graphe de contrôle ci-contre, par une branche d'alternative.

```
    ...
X=ro.m(arg1,...,argn);
if(cond) X.a++;
          ...
else X.a--;
     ...
```



Par conséquent, nous ne calculons pas un premier point d'utilisation mais un ensemble de premiers points d'utilisation d'un futur. Pour définir cet ensemble commençons par définir l'ordre d'exécution des points d'un programme $\mathcal{P}$ déterminé par son graphe de contrôle :

Définition 6 Ordre d'exécution $\widehat{\mathcal{R}}$. *Soit* $\mathcal{R}$ *une relation séquentielle entre les nœuds du graphe de contrôle,* $\mathcal{R} \subseteq Node \times Node$*. Et* $\mathcal{D}$ *le DAG*[1] *(potentiellement infini) obtenu après*

[1]Directed Acyclic Graph

dépliage des boucles de $\mathcal{P}$. $\mathcal{R}$ induit un ordre partiel, noté $\widehat{\mathcal{R}}$, sur $\mathcal{D}$ qui représente l'ordre d'exécution des points de programmes, $\mathcal{D} = (Node, \widehat{\mathcal{R}})$.

Effectivement, $\mathcal{D}$ code l'ensemble de toutes les exécutions possibles du programme $\mathcal{P}$. Notons par Use l'ensemble des points d'utilisation d'un futur et par $First\_Use$ l'ensemble des premiers, $First\_Use \subseteq Use$.

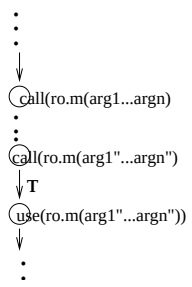
Définition 7 Points First_Use . *Les premiers points d'utilisation d'un futur sont les points d'utilisation minimaux selon l'ordre d'exécution à partir du point de définition de ce futur. Formellement,*

$$First\_Use(fut) \stackrel{def}{=} \{pt \in Use(fut) | \nexists pt' \in \mathcal{D}, (pt' \widehat{\mathcal{R}} pt) \wedge (pt' \in Use(fut)\}$$

Dans ce calcul nous retrouvons les problèmes de propagation d'informations des analyses inter-procédurales, dont le résultat n'est pas toujours précis ; néanmoins, il existe des analyses qui retournent de bonnes approximations [94].

De manière analogue, à un point d'utilisation peuvent correspondre plusieurs points de définition tel qu'illustré dans le code et graphe ci-dessous :

```
    ...
X=ro.m(arg1,...,argn);
... // du code n'utilisant pas X
X=ro.m(arg1'',...,argn'');
X.a++;
    ...
```



Pour ce cas le problème d'association du point de définition au point d'utilisation ne se pose pas, nous considérons que le point d'utilisation d'un futur est associé à son dernier point de définition, les définitions précédentes sont donc ignorées. À l'instar de ce que l'on fait en pratique, une nouvelle affectation d'une variable écrase sa valeur antérieure.

2.3 Service et Queue de Requêtes

Du côté de l'objet distant (l'appelé), les appels sont mémorisés sous forme de *requêtes* dans une file d'attente, appelée *queue de requêtes* . D'un point de vue pratique, une queue est un méta-objet séparé du body et du proxy qui reçoit en permanence (*input-enable*) des appels d'autres objets distants. À un moment donné, le body se charge de servir une et une seule requête à la fois. Une requête est dite *servie* lorsqu'elle est *retirée* de la queue et le code de la méthode qu'elle représente est *exécuté*. Ce code peut à son tour invoquer

des méthodes d'autres objets distants, et son exécution 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 contiguës.

ProActive fournit à l'utilisateur plusieurs primitives de sélection et de service des requêtes à partir de la queue. Celles-ci offrent de nombreuses manières de filtrer une requête, par son nom, par les arguments, et de les servir dans un ordre d'arrivée ou dans l'ordre inverse, mais également des itérateurs permettant d'étendre les façons de traiter les requêtes. Ces primitives sont définies dans la classe *Serve* de la bibliothèque ProActive, e.g. :

- **void** $serveAll(String\ m);$

  sert toutes les requêtes de la méthode appelée m (retire successivement de la queue toutes les méthodes m et exécute le code de chacune).

- **void** $serveAll(RequestFilter\ f);$

  sert toutes les requêtes acceptant l'objet f filtre qui implémente un prédicat donné.

- **void** $serveOldest()$ $(serveYoungest();)$

  sert la plus ancienne (récente) requête de la queue.

- **void** $serveOldest(String\ m)$ $(serveYoungest(String\ m));$

  sert la plus ancienne (récente) méthode appelée m.

- **void** $FlushingServeOldestRequest()$ $(FlushingServeYoungestRequest());$

  ne sert que la plus ancienne (récente) requête et supprime les autres.

- **void** $serveOldest(RequestFilter\ f)$ $(serveYoungest(RequestFilter\ f));$

  Sert la plus ancienne (récente) requête satisfaisant le filtre f.

2.4 Bilan

Nous venons de présenter un noyau de *ProActive* permettant la mise en œuvre d'applications distribuées et la communication entre les objets actifs. Ce noyau, qui sera le cadre de notre étude, est constitué d'un ensemble de primitives de haut niveau ayant une sémantique formelle et garantissant de bonnes propriétés comportementales.

En résumé, ce noyau est composé de primitives de création d'objets actifs, de primitives d'appel (bloquant ou non bloquant) de méthodes et de primitives de services de requêtes selon différents modes. Bien entendu, *ProActive* présente d'autres caractéristiques et d'autres primitives mais que nous ne traitons pas dans la suite, notamment, la communication de groupe, la sécurité des communications, la mobilité et la migration des objets, et le traitement des exceptions.

Chapitre 3

Modèles Comportementaux Finis d'applications *ProActive*

La construction des modèles se réalise dans la distance qui sépare le langage-objet du métalangage.
Greimas-Courtés, 1979.

Notre but est de développer la technologie nécessaire pour pouvoir vérifier des propriétés comportementales d'applications distribuées, notamment d'applications *ProActive*. Pour cela, il est indispensable de concevoir le modèle comportemental général d'une application avant la vérification. Dans ce chapitre, il est donc question de la définition et la génération de modèles comportementaux (finis) d'applications *ProActive*.

Notre approche [43] de modélisation est basée sur la sémantique des systèmes parallèles et concurrents. La formalisation des interactions entre les objets distribués (communication et synchronisation) est définie à un haut niveau d'abstraction par des systèmes de transitions. Par ailleurs, le modèle de toute l'application est obtenu par composition hiérarchique de (sous-)modèles correspondants aux objets la constituant.

L'avantage de cette approche est d'une part la parallélisation de la procédure de construction des modèles et la réutilisation de ces derniers ; les modèles de tous les objets peuvent être construits simultanément, une fois construit le modèle comportemental d'un objet peut être utilisé comme composant ayant une interface : un ensemble de services requis et offerts. D'autre part, l'asservissement du problème principal de model-checking, à savoir l'explosion d'états ; avant d'appliquer l'opération de composition un (sous-)modèle peut être réduit par opération de minimisation, ce qui réduit considérablement le modèle global.

La première section de ce chapitre (Section 3.1) propose les définitions théoriques des formalismes choisis pour la modélisation d'un comportement. Nous présentons formellement les modèles puis leur syntaxe graphique. Nous verrons par un mapping des graphiques

vers le modèle que le langage graphique ne couvre pas tout le modèle formel. Dans la partie suivante (Section 3.2), nous définissons le graphe d'appel de méthodes d'un programme *ProActive*, qui sera la structure intermédiaire à partir de laquelle seront générés les modèles comportementaux. Puis dans la section 3.3, nous proposons les règles de construction de ces modèles. Nous illustrons (Section 3.4) par un exemple les développements présentés dans ce chapitre. Et enfin, nous étudierons (Section 3.5) quelques propriétés des modèles et de l'algorithme de construction.

3.1 Modèles Comportementaux

Dans le même esprit que la sémantique de l'algèbre des processus (CCS ou CSP), le comportement d'un processus (objet actif) peut être représenté par un système de transitions dont les états sont des actions des programmes (événements observables) et les transitions dénotent les changements suite aux actions effectuées durant son exécution. Ces structures sont également utilisées comme modèles d'interprétation pour les logiques modales et temporelles. Le modèle global d'un système de processus distribués, communicants et asynchrones est produit par une composition de ces systèmes de transitions grâce aux *réseaux de synchronisation*.

3.1.1 Modèles Formels

Parmi la famille des systèmes de transitions nous trouvons les *systèmes de transitions étiquetées* [90] (ou automates).

Systèmes de transitions étiquetées

Ensemble de sommets (états) reliés par des arcs (transitions), auxquels sont attachés des labels encodant des actions observables. Avant de donner la définition formelle d'un système de transitions étiquetées, nous introduisons le terme Act pour désigner un alphabet d'actions : l'ensemble des actions observables et l'action non-observable τ, $Act = \mathcal{L} \cup \tau$.

Une autre définition d'un système de transitions étiquetées tel que c'est donné, par exemple, dans [47] (différente de la définition 1) :

Définition 8 LTS. *Un système de transitions étiquetées (de l'anglais labelled transition system) est un tuple :*

$$LTS \stackrel{def}{=} (S, s_0, L, \rightarrow)$$

où

1. S *est l'ensemble des états,*
2. $s_0 \in S$ *est l'état initial,*

3. *L est l'ensemble des labels (étiquettes), représentant les appels locaux et distants de méthodes, ainsi que les réponses,* $L \subseteq Act$,
4. $\rightarrow$ *est l'ensemble des transitions :* $\rightarrow \subseteq S \times L \times S$

Nous écrirons $s \xrightarrow{\alpha} s'$ *au lieu* $(s, \alpha, s') \in \rightarrow$.

Le produit d'automates définit un réseau de synchronisation.

Produit de synchronisation

Lors de la modélisation d'une application composée d'un ensemble de processus coopérants entre eux, il n'y a pas besoin de représenter l'ensemble des actions possible de celle-ci ; puisque les interactions entre les processus sont sujettes à des communications et à des contraintes de synchronisation ; effectivement, la synchronisation a pour effet de contraindre certaines actions d'un groupe de processus à s'exécuter simultanément, ou, a contrario, d'interdire leur occurrence simultanée [29]. Ainsi, elle peut être décrite par une liste (pas nécessairement finie) de vecteurs d'actions pouvant ou devant se produire simultanément (ou par le complément, l'ensemble des vecteurs qui ne peuvent se produire simultanément).

Le modèle global (LTS) d'une application est alors qu'un sous-système obtenu par composition parallèle et hiérarchique des modèles associés à ses processus. L'opérateur de composition est le réseau de synchronisation dont les feuilles sont des LTSs ayant une sorte.

Définition 9 Sorte. *Une sorte est un ensemble d'actions noté I,* $I \subseteq Act$.

Une Sorte est l'ensemble des paramètres des arguments attendus par un opérateur, en l'occurrence l'opérateur réseau.

Un réseau de synchronisation définit un ensemble de vecteurs de synchronisation d'un nombre fini n de processus P_i. Chaque vecteur est de la forme $A = A_1^* \times \ldots \times A_n^*$ tel que $A_i^* = A_i \cup \{*\}$ est l'alphabet d'actions de P_i étendu par l'ajout du symbole $*$ interprété comme étant une "non action". Nous définissons un réseau de synchronisation, différemment de la définition 4, au moyen de transducteurs comme étant un opérateur parallèle généralisé. Formellement :

Définition 10 Réseau de synchronisation. *Un réseau est un tuple, noté Net (de l'anglais) :*

$$Net \stackrel{def}{=} < A_G, I, T >$$

où :

- A_G *est la sorte du réseau,*
- I *est un ensemble fini de sortes* $I = \{I_i\}_{i=1,\ldots,n}$,

– *T (comme Transducteur) est un LTS* $(S_T, s_{0_T}, L_T, \rightarrow_T)$ *tel que* $\forall \overrightarrow{v} \in L_T, \overrightarrow{v} =< l_t, \alpha_1, \ldots, \alpha_n >$ *où* $l_t \in A_G$ *et* $\forall i \in [1 \ldots n], \alpha_i \in I_i \cup \{*\}$.

Notons qu'un vecteur de synchronisation peut définir une synchronisation entre une, deux, ou plusieurs actions de différents arguments du réseau. Si le vecteur de synchronisation ne comporte qu'une seule action (élément), cette action peut alors s'exécuter de manière autonome, sans contrainte.

Un LTS $(S, s_0, L, \rightarrow)$ ne peut être un argument d'un Net que si l'ensemble de ses labels s'accorde avec la sorte de ce dernier, i.e. $(L \subseteq I_i)$. À cet égard, une Sorte caractérise une famille de LTSs satisfaisant cette propriété d'inclusion.

Les réseaux ainsi définis (Définition 10) décrivent des configurations dynamiques de processus, dans lesquelles il est possible à une transition de changer l'état du réseau. Nous utilisons la notion de *Transducteur*, dans le même sens que les expressions d'Open Lotos [81], pour désigner un LTS dont les labels sont des vecteurs de synchronisation. Lorsque le transducteur ne contient qu'un seul état, le réseau est dit *statique*.

La sémantique d'un réseau de synchronisation est donnée par le *produit de synchronisation* défini par :

Définition 11 Produit de synchronisation . *Étant donné un ensemble fini de LTS* $\{LTS_i = (S_i,\ s_{0_i},\ L_i, \rightarrow_i)\}_{i=1\ldots n}$ *et un réseau* $< A_G, \{I_i\}_{i=1\ldots n}, (S_T, s_{0_T}, L_T, \rightarrow_T) >$, *tel que* $\forall i \in [1 \ldots n], L_i \subseteq I_i$, *le produit est un LTS* $(S,\ s_0,\ L,\ \rightarrow)$ *où :*

– $S = S_T \times \Pi_{i=1}^{n}(S_i)$,
– $s_0 = s_{0_T} \times \Pi_{i=1}^{n}(s_{0_i})$,
– $L = A_G$,
– $\rightarrow \triangleq \{s \xrightarrow{l_t} s' |\ s =< s_t, s_1, \ldots, s_n >, s' =< s'_t, s'_1, \ldots, s'_n >, \exists\ s_t \xrightarrow{\overrightarrow{v}} s'_t \in \rightarrow_T,$
$\overrightarrow{v} =< l_t, \alpha_1, \ldots, \alpha_n >,\ \forall i \in [1 \ldots n],\ (\alpha_i \neq * \wedge s_i \xrightarrow{\alpha_i} s'_i \in \rightarrow_i) \vee (\alpha_i = * \wedge s_i = s'_i)$

Notons que le résultat obtenu par le produit est également un LTS, pouvant donc à son tour se synchroniser avec d'autres LTSs du réseau. Par conséquent, cette propriété permet d'avoir plusieurs niveaux de synchronisation, i.e. une définition hiérarchique du système.

3.1.2 Représentation graphique

Tout comme pour la représentation des modèles objets (par exemple UML), une description graphique ou visuelle permet d'améliorer la lisibilité et l'analyse des modèles. Afin d'illustrer nos modèles et de les décrire naturellement et immédiatement nous utilisons l'éditeur graphique, Autograph [98]. Cet éditeur offre une interface simple et puissante pour la représentation d'automates et de réseaux, il permet également de traduire un dessin en format *fc2* et donc d'être interfacé avec différents modules de vérification.

Ainsi nous proposons des définitions d'automates et de réseaux décrits graphiquement.

Système de transitions étiquetées

Un LTS est schématisé graphiquement par des ronds représentant les états et des arcs sous-tendant des labels, représentant les transitions étiquetées par des actions. La syntaxe abstraite d'un LTS graphique :

Définition 12 LTS graphique. *Un LTS graphique, noté* $\widehat{LTS}$*, est un couple :*

$$\widehat{LTS} \stackrel{def}{=} (nom,\ \mathcal{R},\ \mathcal{L})$$

avec

- *nom est l'identifiant du LTS.*
- $\mathcal{R}$ *est un ensemble de ronds dont certains peuvent être marqués.*
- $\mathcal{L}$ *est un ensemble de arcs orientés entre les ronds, chaque arc est de la forme* $R1 \stackrel{l}{\rightarrow} R2$*.*

et dans lequel les labels des arcs sont des actions d'un alphabet Act.

La Figure 3.1 est le graphique d'un exemple de LTS, nommé *Automaton*, ayant deux ronds dont un est marqué (rond doublé) et quatre arcs étiquetés par les événements $!Req\_m_i$ et $?Rep\_m_i$.

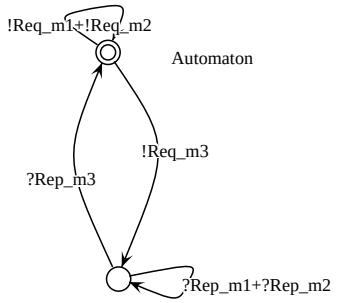


FIG. 3.1 – Exemple d'une représentation graphique d'un LTS

Réseau de synchronisation

Nous avons défini (Définition 10), un réseau comme étant un opérateur de synchronisation de deux ou plusieurs processus désignés par leurs sortes. Nous définissons alors

graphiquement un réseau par des boites (trous) dotées de ports (sortes) et reliées entre elles par des liens (synchronisations).

Une boite est définie par :

Définition 13 Box. *Une boite, B est un rectangle doté d'un nom $Name(B)$ et décorée à l'entour par un ensemble de ronds étiquetés et appelés* **ports**. *Notons par $Ports(B)$ l'ensemble des ports d'une boite et par $Label(p)$ le nom (étiquette) d'un port.*

Un réseau graphique est défini comme suit :

Définition 14 Réseau graphique. *Un réseau graphique, noté $\widehat{Net}$, est le couple :*

$$\widehat{Net} \stackrel{def}{=} (\mathcal{B},\ \mathcal{L})$$

avec

- *$\mathcal{B}$ est un ensemble fini de boites dont l'une est englobante (une boite B qui englobe une boite B', est noté par $B' \subset B$) ; $\forall B_i \subset B_0$, B_0 étant une boite englobante.*
- *$\mathcal{L}$ est un ensemble de liens entre les ports, chaque lien est de la forme $B1.p1 \xrightarrow{l} B2.p2$ dans laquelle $p1$ est un port de la boite $B1$ et $p2$ est un port de la boite $B2$,*

et dans lequel les noms des ports et les labels des liens sont des actions de l'alphabet Act. Nous notons par $Actions(B_i)$ et $Actions(\mathcal{L})$ respectivement l'ensemble des noms de ports d'une boite B_i et l'ensemble des noms de labels des liens $\mathcal{L}$.

La Figure 3.2 est un exemple d'un graphique d'un Net : deux boites désignées $box1$ et $box2$ ayant respectivement deux et trois ports et reliées entre elles par deux arcs Req_m et Rep_m ; et une boite $box3$ (boite englobante) portant trois des ports.

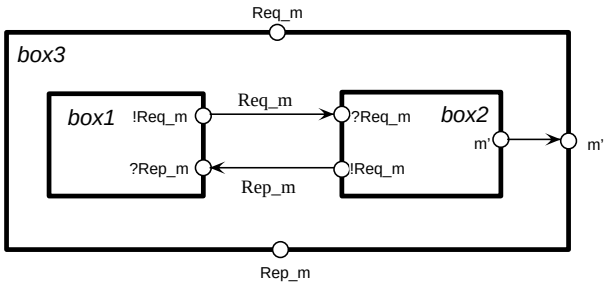


FIG. 3.2 – Exemple d'une représentation graphique d'un Net

En fait, on peut ne pas représenter la boite englobante, elle a pour rôle de rendre visible les ports libres et les synchronisations.

Par la définition précédente d'un réseau nous n'exprimons pas l'aspect hiérarchique d'un réseau i.e, une boite peut à son tour contenir un réseau. En effet, un réseau a une structure d'arbre dont les feuilles sont des LTSs. L'arbre peut être réduit à un seul élément, la racine, qui est un LTS ou une famille de boites gigognes. Par analogie aux termes algébriques, nous définissons précisément cette description informelle par un réseau graphique clos :

Définition 15 Réseau Graphique Clos. *La syntaxe abstraite d'un réseau graphique clos est :*

$$\begin{array}{lll} B_i & \stackrel{def}{=} & \widehat{Net} \mid \widehat{LTS} \mid box \\ \widehat{Net} & \stackrel{def}{=} & (B_g < B_1, \ldots, B_n >,\ \mathcal{L}) \\ \widehat{B_g} & \stackrel{def}{=} & box \\ Name(B) & \stackrel{def}{=} & nom \end{array}$$

et tel que

1. $\forall B_*,\ B_*.p \in Ports(\mathcal{B})$
2. $\forall i \in [1 \ldots n]\ B_i \subset B_g$ *alors* $\exists l \in \mathcal{L}$ *tel que* $B_i.p_j \xrightarrow{l} B_g.p_k$
3. $\exists B_i, B_j \in \mathcal{B}\ B_i, B_j \neq B_g$ *tel que* $\exists l \in \mathcal{L}\ B_i.p \xrightarrow{l} B_j.p' \vee B_j.p' \xrightarrow{l} B_i.p$

Nous constatons par cette définition que les réseaux ne sont pas des structures fermées ; une boite peut être un réseau, un automate ou encore un trou, une boite libre, B, désignée par son nom ($Name(B)$) et ses ports ($Ports(B)$). L'opération de substitution de boites libres se fait à l'édition de lien ; chaque boite libre, B_i est remplacée par un Net ou un LTS ayant une sorte conforme aux $Actions(B_i)$.

3.1.3 Réseaux graphiques vs modèles mathématiques

Nos réseaux graphiques sont à l'évidence moins expressifs que le modèle des réseaux de synchronisation définis à la section précédente :

- d'une part, d'après la définition mathématique un LTS est, a priori, non fini or par la syntaxe graphique nous dérivons nécessairement un LTS fini, le nombre d'états et de transitions est égal au nombre de ronds et d'arcs dessinés.
- d'autre part, la syntaxe graphique d'un réseau ne décrit que des configurations statiques d'un réseau de synchronisation alors que mathématiquement nous en décrivons même des dynamiques grâce à la notion de transducteur.

Pour préciser ces différences, et en même temps pour donner une signification à nos modèles, nous définissons une traduction de la syntaxe d'un modèle graphique dans la sémantique de sa forme mathématique. Au passage nous listerons les conditions sous lesquelles cette traduction est correctement définie.

Commençons par définir la traduction des LTSs. Dans le cas de nos dessins, les LTSs analysés sont munis d'un seul rond marqué, dit initial et caractérisé par un rond doublé. La fonction de traduction, notée $[\![.]\!]_L$, d'un LTS graphique, $< \mathcal{R},\ \mathcal{L} >$ en un correspondant formel $< S, s_0, L, \rightarrow >$ est définie par :

$$[\![< \mathcal{R}, \mathcal{L} >]\!]_L =< S, s_0, L, \rightarrow >$$

tel que

$$\begin{array}{lcl} S & = & R \\ s_0 & = & R_0 \text{ étant le rond initial} \\ L & = & \{l \in \mathcal{L} / R \xrightarrow{l} R'\} \\ \rightarrow & = & \{s \xrightarrow{l} s' / \exists R \xrightarrow{l} R' \in \mathcal{R} \wedge s = R, s' = R'\} \end{array}$$

La fonction de traduction d'un réseau graphique $(B_g < B_1, \ldots, B_n >,\ \mathcal{L})$ vers son homologue formel $< A_G, I, T = (S_T = \{s_0\}, s_0, L_T, \rightarrow_T) >$ est notée $[\![.]\!]_N$ et définie par :

$$[\![(B_g < B_1, \ldots, B_n >,\ \mathcal{L})]\!]_N =< A_G, I, T = (S_T = \{s_0\}, s_0, L_T, \rightarrow_T) >$$

tel que

$$\begin{array}{lcl} A_G & = & Ports(B_g) \\ I & = & \{Actions(B_i)\}_{[1\ldots n]} \\ L_T & = & \{\overrightarrow{v} / \exists B_i.p \xrightarrow{l} B_j.p' \in \mathcal{L}, \quad \overrightarrow{v} =< l, v_1, \ldots, v_n > \quad \forall i,j\ v_i = Label(p)_i, \\ & & \quad v_j = Label(p')_j \ et\ \forall k \neq i,j\ v_k = *\} \\ \rightarrow_T & = & (s_0, \overrightarrow{v}, s_0) \quad \forall \overrightarrow{v} \in L_T \end{array}$$

Notons que cette traduction suppose que toutes les boites sont classées par un indice i tel que $i \in [1 \ldots Card(\mathcal{B}) - 1]$. Chaque position correspond à une boite, c'est à dire, un processus dans le modèle. Ainsi, la longueur des vecteurs de synchronisation $\overrightarrow{v}$ est égale à $Card(\mathcal{B}) + 1$.

En fait, les liens port à port (Définition 15) entre les boites définissent une synchronisation pair à pair entre les processus ($v_k = *$ pour tout $k \neq i, j$).

Notons également que l'ensemble des nœuds du transducteur T est réduit au seul élément s_0, donc à un seul état ; effectivement, nos réseaux graphiques ne décrivent qu'une configuration statique d'une synchronisation de processus.

3.2 Forme intermédiaire de programmes

Construire nos modèles directement à partir du code source Java ou (une version simplifiée) du bytecode [86], ou encore du JIMPLE [107] demande, au préalable, une analyse statique. Afin de dissocier cette étape de l'étape de construction et dans le but d'élaborer une plate-forme la plus indépendante possible du langage de programmation, notre analyse se base sur une représentation abstraite des applications. Cette représentation construite à partir des techniques d'analyse statique, en se basant sur l'analyse de classes [78], symbolise les instructions sous forme de graphe [77] ; elle fait abstraction du flot de données et se focalise sur le flot de contrôle, càd, les méthodes invoquées lors de l'exécution et l'ordre de leur invocation. Cette représentation, qui est le *graphe d'appel de méthodes*, perd toute information relative aux données. Par contre, elle encapsule une approximation de la hiérarchie de classes assez précise pour élaborer le graphe de flot de contrôle.

3.2.1 Graphe d'appel de méthodes

Le graphe d'appel de méthodes, noté *MCG* (de l'anglais method call graph), est une représentation sous forme de graphe orienté d'un programme. Les nœuds du graphe correspondent à des instructions et les liens reliant les nœuds représentent deux types de dépendances : celles de séquences et celle d'appels.

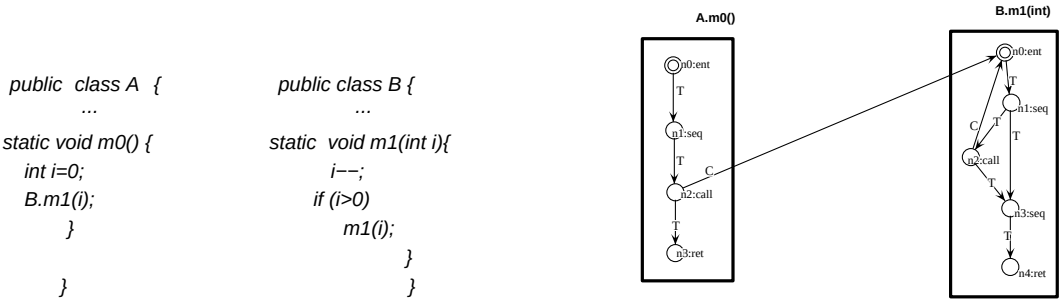
Les dépendances d'*appels* ont lieu entre deux instructions dans le cas où la première est une instruction d'appel de méthode et la deuxième est la première instruction de cette méthode (appelée). Les dépendances de *séquences* apparaissent entre une instruction quelconque et celle dont l'exécution suit directement, elles expriment également les branchements et les boucles.

Les MCGs (exemple Figure 3.3) sont des structures classiques utilisées dans la compilation [25] et l'analyse d'environnements. Leur construction se base essentiellement sur les techniques d'abstraction [62, 68] et les techniques d'analyse statique : analyse interprocédurale [75, 95] et notamment l'analyse de classes [82, 78].

Pour représenter un programme "monolithique", c'est à dire formé d'une méthode unique, deux types de nœuds sont nécessaires : ceux qui représentent les instructions et ceux (celui) qui représentent le début de la méthode, appelé *point d'entrée*.

Définition 16 *Le MCG d'un programme (ayant une ou plusieurs méthodes) est constitué d'un ensemble de nœuds (points de programme) dont un dit initial (le point d'entrée de la méthode principale) et d'arcs orientés : des arcs dits intra-procéduraux (et notés $\rightarrow^T$) relient les nœuds d'une même méthode et d'autres dits inter-procéduraux (et notés $\rightarrow^C$) relient les nœuds d'appel d'une méthode aux nœuds d'entrée d'une autre.*

Le MCG ainsi défini concerne la représentation de programmes simplement séquentiels. Nous proposons alors une extension de la notion de graphe d'appel de méthodes vers



(a) Programme Java (b) Graphe d'appel

FIG. 3.3 – Un programme Java (a) et le MCG correspondant (b)

la notion de *graphe d'appel de méthodes distribuées*, de façon à permettre la représentation de programmes distribués, précisément des programmes écrits en *ProActive*.

3.2.2 Graphe d'appel de méthodes réparties

Le graphe étendu, noté dMCG (acronyme de Distributed Method Call Graph) reprend les éléments du MCG standard et rajoute en plus certains nœuds et liens permettant de représenter les spécificités des applications que nous étudions, en l'occurrence les applications *ProActive* : de façon à marquer la distinction entre les appels locaux et les appels distants, les points de service des requêtes, les points de réponse à celles ci, ainsi que les points d'utilisation des résultats.

Les nœuds rajoutés :

- Nœuds de *réponse* à un appel : associés à tous les points où le résultat d'une exécution distante est retournée.
- Nœuds de *service* : ces nœuds représentent les points de programme où une méthode est retirée de la queue de requête et son code est exécuté.
- Nœuds d'*utilisation* : les nœuds liés aux points de programme où le résultat d'un appel distant est requis.
- Nœuds d'appels *distants* : quand c'est possible, les nœuds d'appel de méthodes locales sont différenciés de ceux d'appels distants.

Formellement, un graphe d'appel de méthodes distribuées est défini par :

Définition 17 *À partir de la méthode principale d'une application, le* **graphe d'appel de méthodes distribuées** *est défini par le tuple* $dMCG \stackrel{def}{=} (id, V, \rightarrow^C, \rightarrow^T)$ *où*

1. *id est un nom désigné d'une méthode ayant la forme Class.identifier tel que Class représente un objet local ou un objet distant,*
2. *V est un ensemble de nœuds, chacun muni d'un type appartenant à $\{ent(id),\ call(id),\ serv,\ rep(id),\ seq,\ use(id),\ ret\}$,*
3. *$\rightarrow^C \subseteq V \times V$ sont les arcs inter-procéduraux (call) du dMCG,*
4. *$\rightarrow^T \subseteq V \times V$ sont les arcs intra-procéduraux (transfert) du dMCG.*

Selon le type d'un nœud de V, celui ci indique si c'est un point d'entrée *ent(id)* d'une méthode, un appel *call(id)* à une autre méthode (locale ou distante), un service *serv* d'une méthode, une réponse *rep(id)* à un appel de méthode distante, un nœud séquence *seq* (représentant les instructions standards incluant les branchements), un nœud d'utilisation *use(id)* (un point de programme où l'objet futur sera utilisé), ou nœud *ret* point où l'exécution d'une méthode s'achève normalement ou par une exception.

Le domaine d'une méthode *id* est l'ensemble de ces nœuds, noté $\mathcal{D}$, $\mathcal{D}(id) \subseteq V$.

Nœuds spécifiques :

Notons que les nœuds de types *serv*, *rep(id)* et *use(id)* ne concernent pas les MCGs classiques ; en effet, ils codent les caractéristiques de *ProActive*. Par exemple, la primitive *waitFor(ro.m)* exécutant l'attente bloquante sur une valeur de retour d'un appel à la méthode distante m, est décrite par une partie d'un dMCG de la Figure 3.4. Le nœud d'appel *call(ro.m)* symbolisant l'appel distant, suivi directement du nœud *use(ro.m)* caractérise clairement l'attente.

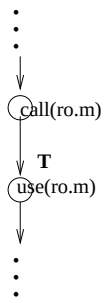


FIG. 3.4 – Le sous-graphe de la primitive *waitFor*

Par ailleurs, remarquons que le nœud *call(ro.m)* bien que c'est un noeud d'appel de méthode ne comporte pas d'arc d'appel (arc de type $\rightarrow^C$) ; en effet, vu qu'un objet actif ne peut exécuter le code des méthodes d'autres objets actifs, dans le dMCG de cet objet nous ne représentons pas le corps des méthodes distantes.

Nœuds d'appels :

Une caractéristique de Java (et langages orientés objets) réside dans le fait qu'une classe peut transmettre des propriétés à une autre appelée sous-classe, par la notion d'*héritage*. Les *héritiers* possèdent donc les mêmes attributs que les parents (et éventuellement des attributs supplémentaires).

Une des conséquences du concept d'héritage de classe est la redéfinition de méthode qui consiste à fournir une implantation différente d'une méthode de même signature que celle fournie par la classe de base ; en réécrivant totalement la méthode initiale ou en y ajoutant simplement du code. Cette possibilité complique la construction du graphe d'appel ; en effet, le type de l'objet d'une méthode appelée ne peut être résolu que dynamiquement d'où l'imprécision de l'analyse statique. À chaque point d'appel de méthode l'analyse inter-procédurale retourne une approximation, un ensemble de méthodes pouvant éventuellement être invoquées en ce point. Par exemple, considérons le code suivant :

```
public class A { void foo(){ System.out.println(''A'');} }
public class B extends A { void foo(){ System.out.println(''B''); } }
public class C extends B { void  bar(B O){ O.foo(); }
```

La méthode *foo* est définie dans la classe A et redéfinie dans la sous classe B. Par analyse statique, l'appel *O.foo()* de la classe C génère l'ensemble *{A.foo(), B.foo()}* qui est l'ensemble potentiel des méthodes pouvant être effectivement appelées et qui se traduit au niveau du graphe d'appel de méthodes par de multiples arcs d'appels, (arc de type $\rightarrow^C$), à partir d'un même point (voir Figure 3.5).

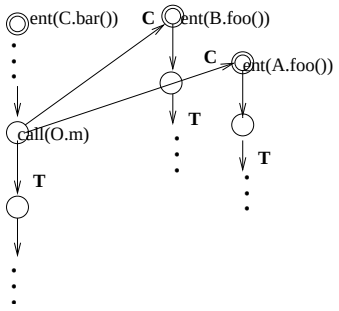


FIG. 3.5 – Sous-graphe de la classe C

Toutefois, différentes analyses existent afin d'améliorer la précision de ces approximations : analyse de la hiérarchie de la classe [63] ainsi que l'analyse des types de variables [104].

Par ailleurs, l'approximation de l'analyse statique peut fournir un indéterminisme sur le type d'un objet. À un point d'appel donné un objet peut avoir un type indéfini : local ou distant. Par conséquence, à ce même point peut correspondre les deux types d'appels : local et distant, ce qui implique également qu'au niveau du dMCG un nœud peut avoir

plus d'un type. Dans ce chapitre, nous n'abordons pas ce problème (nous le traitons dans le chapitre suivant) ; nous supposons qu'à tous les points de programmes l'analyse statique peut déterminer expressément le type, local ou distant, d'un objet défini.

Afin de donner les caractéristiques d'un dMCG, définissons ce qu'est un dMCG correctement construit : un dMCG *bien-formé* est un MCG (classique) bien-formé avec des structures ajoutées également bien-formées.

Définition 18 Bien-formé. *Un dMGC bien-formé a un seul nœud initial (le nœud d'entrée à la méthode principale) et a un seul nœud de sortie (le nœud de retour de la méthode principale), de plus tel qu'à partir du nœud de type :*

- *Ent part un unique arc de transfert :*

$$v = ent(m) \quad v \rightarrow^T v'$$

- *Call (à une méthode locale) part un (ou plusieurs [1]) arcs d'appels et exactement un seul arc de transfert :*

$$v = call(lo.m) \quad v \rightarrow^T v' \quad v \rightarrow^C v'' \quad v'' = ent(lo.m)$$

- *Seq part au moins un arc de transfert :*

$$v_1 = seq \quad v_1 \rightarrow^T v_2 \quad v_1 \rightarrow^T v_3 \quad \dots \quad v_1 \rightarrow^T v_n$$

- *Ret ne dérive aucun arc.*

Pour le reste, concernant les nœuds intrinsèques aux dMCGs de nos applications, à partir d'un nœud de type :

- *Call (à une méthode distante) ne part qu'un seul arc de transfert (pas d'arc de d'appel) :*

$$v = call(ro.m) \quad v \rightarrow^T v'$$

- *Rep part exactement un arc de transfert :*

$$v = rep(m) \quad v \rightarrow^T v'$$

- *Serv part un arc transfert suivi directement d'un nœud de type Call (à une méthode locale) :*

$$v = serv \quad v \rightarrow^T v' \quad v' = call(lo.m)$$

- *Use part également exactement un seul arc, transfert :*

$$v = use(ro.m) \quad v \rightarrow^T v'$$

[1] Hiérarchie de classes

3.3 Sémantique informelle de ProActive et construction des modèles

Dans la version actuelle de *ProActive*, une application est une collection (aplatie) d'activités, d'*objets actifs*, qui peuvent être créés dynamiquement, migrer et terminer (mourir). Ainsi pour modéliser une application, nous associons à chaque *objet actif* un *processus* et la communication entre ces objets actifs sera caractérisée par un réseau de synchronisation.

Le modèle global de l'application est construit de manière compositionnelle à partir du réseau de synchronisation des (sous)-modèles de ses objets actifs. Le modèle de base dans cette construction est un système de transitions étiquetées (LTS).

Étant donné une propriété à prouver, nous demandons à l'utilisateur de fournir une interprétation abstraite de tous les types de données de l'application. Cette abstraction préservera les informations sur les données (paramètres des objets actifs et ceux des messages) pertinentes pour cette propriété. En pratique, les outils tels que Bandera abstract specification language (BASL) définissent bien ces abstractions.

À partir du code source d'une application, la procédure de construction du modèle global moyennant des abstractions finies des paramètres génère le modèle fini du réseau (Net) de l'application et le modèle (LTS) de chaque classe d'objet actif (ainsi que toutes ses classes passives requises).

Dans ce chapitre nous nous intéressons aux applications ayant des topologies statiques et bornées en nombre d'objets ; ce nombre peut être une approximation calculée en utilisant les techniques d'analyse de pointeurs [84, 20] et d'aliasing [70] et déterminé à partir des points de création des objets actifs définis par les primitives :

$newActive\ (Class,\ Args,\ Site);$

ou encore

$turnActive\ (Class,\ Args,\ Site);$

Par contre, étant donné que nous ne nous intéressons pas aux propriétés d'emplacement physique des objets dans le réseau, nous écartons toutes les informations liées à la situation d'un objet et à la migration.

Nous supposons que nous avons un ensemble (fini) de points de création et un domaine fini des paramètres des objets actifs, nous obtenons une énumération finie d'objets actifs formée par :

1. $\mathcal{O} = \{O_i\}$ un nombre fini de classes d'objets actifs.
2. Un nombre fini d'instantiations pour chaque classe d'objet actif, suivant les valeurs possibles des paramètres passées lors de la création, noté $Dom(O_i)$.

Pour chaque classe d'objet actif, ou pour chaque point de création d'objet actif, une application créé habituellement de manière dynamique un nombre arbitraire d'objets. En revanche, la plupart des propriétés n'impliquent que certaines instantiations finies d'une certaine classe d'objets actifs. Nous ne considérons alors qu'un nombre fini d'objets.

L'exactitude du modèle (en terme du nombre des objets actifs et la granularité des observations du contenu des messages) peut être alors ajustée selon les preuves à faire.

Les actions, que nous souhaitons observer sur nos modèles, sont tous les appels de méthodes (locales ou distantes), les retours des résultats, les services offerts par les objets, ainsi que la disponibilité des futurs. L'alphabet d'actions peut alors être défini par :

Définition 19 Action. *L'alphabet d'actions est l'ensemble*

$$Act = \{!Req\_m, !Rep\_m, !Serv\_m, !Fut\_m, m, ?Req\_m, ?Rep\_m, ?Serv\_m, ?Fut\_m, \tau\}$$

représentant respectivement l'appel de méthode distante (requête), l'envoi de résultat (réponse), l'invocation de service, l'envoi d'un futur, l'appel de méthode locale, la réception de l'appel, la réception du résultat, la réception de l'invocation de service, la réception d'un futur, et l'action non-observable .

Considérons un objet *co* (l'objet actif courant), appelant une méthode m d'un objet actif distant *ro* (avec la valeur de retour différente de *void*) :

```
x = ro.m (arg1, ... , argn);
```

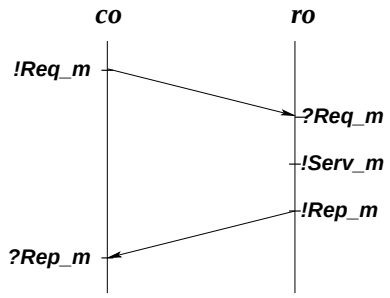


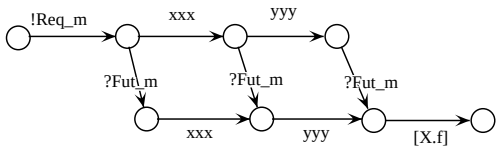
FIG. 3.6 – Échange de messages entre les processus *co* et *ro*

Cet appel correspond à un échange de messages (Figure 3.6) entre ces processus : l'objet courant *co* émet une requête $!Req\_m$, qui sera instantanément reçue dans la queue de l'objet distant *ro* ; celui-ci décidera, au bout d'un temps fini, de servir cette requête et donc exécuter le code de la méthode m ; ce qui est représenté par l'émission du message $!Serv\_m$ vers sa queue de requêtes. À la fin, au cas où cette méthode a un résultat de retour, *ro* émet à son tour le message de réponse, $!Rep\_m$, qui sera reçu de manière synchrone par *co* par le message $?Rep\_m$.

Du point de vue de l'objet courant, la réponse à un appel est asynchrone : la sémantique *attente par nécessité* de *ProActive* préconise que le message de réponse peut être reçu à n'importe quel moment entre l'instant où la requête est émise et l'instant de la première utilisation effective de cette réponse, elle est appelée *futur*. Par exemple dans le code :

```
X = ro.m(arg1,...,argn);
      xxx;  yyy; // du code n'utilisant pas X
X.f (args);
```

L'attente du résultat de l'appel de la méthode m est implicite ; en effet, le futur n'est indispensable qu'au moment de son utilisation, en l'occurrence lors de l'accès au champ f. Ce comportement sera modélisé par l'entrelacement de l'action : avènement du futur (retour du résultat par ro) et les actions représentant le comportement de co entre le point de définition et le point d'utilisation de ce futur :

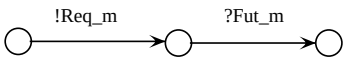


Cet entrelacement est obtenu par composition parallèle du modèle du futur (proxy) et celui du corps de l'objet (body).

Par contre, une attente explicite d'un résultat peut être exécutée par un appel bloquant grâce à la primitive :

```
waitFor(ro.m(arg1,...,argn));
```

Dans ce cas, aucune action ne peut être accomplie avant l'avènement de la réponse, ce qui est modélisé par :



Notons que la notation m des labels des transitions est une abstraction d'un appel de méthode $O.m(arg1, ..., argn)$; il englobe à la fois le nom de classe de l'objet O déterminée par analyse statique, le nom de méthode ainsi qu'une abstraction finie de ses paramètres d'appel.

Les messages reçus par un objet sont déposés dans sa queue de requêtes qui les ordonne et les traitera après service. La queue accepte toujours les requêtes correctes. Son comportement se synchronise avec l'objet émetteur par les messages de réception de requêtes et avec le comportement du corps de l'objet auquel elle appartient par les messages de service qu'il lui délivre. Nous verrons en Section 3.3.4, comment est construit le modèle

d'une queue ainsi que son optimisation en une queue bornée.

Dans la suite, nous traitons d'un sous-ensemble de *ProActive* qui est défini par toutes les primitives d'invocation de méthodes : locales ou distantes, bloquantes ou non bloquantes ainsi que les primitives de services de la queue de requêtes et de réponses. Par contre, nous ne nous intéressons pas aux programmes anormaux, par exemple un programme qui ne se termine jamais, ni aux exceptions.

3.3.1 Construction du réseau

Le modèle du réseau d'une application est déterminé à partir l'ensemble des objets actifs, $\mathcal{O}$, et l'ensemble des requêtes échangées entre les objets.

Pour chaque objet actif, nous construisons une boite, notée $B(Name(O_i), k)$, qualifiée par le nom de la classe et un indice k de $Dom(O_i)$. Afin de calculer les ports de chaque boite et les liens associés, nous analysons le code source de la classe de l'objet correspondant :

1. L'ensemble des méthodes publiques de la classe principale d'un object actif nous fournit l'ensemble des "requêtes admises" par cet objet, donc l'ensemble des ports de type $?Req\_m$ de la boite.
2. L'ensemble des points d'appel à des méthodes d'objets distants détermine l'ensemble des "requêtes émises", donc l'ensemble des ports de type $!Req\_m$; de même que, le sous-ensemble des méthodes retournant un résultat détermine les ports de type $?Rep\_m$ (" réception de réponse").
3. Pour chaque objet courant, nous considérons l'ensemble des méthodes observables : méthodes pertinentes pour la présente preuve ; cet ensemble définit l'ensemble des ports étiquetés m.

Entre chaque paire de ports analogues : émission et réception d'un même événement, $(!\_, ?\_)$, un lien étiqueté par l'événement est construit.

Par exemple, la communication entre les processus *co* et *ro* de la Figure 3.6 est représentée par un réseau donné dans la Figure 3.7 : deux boites nommées par les processus, des ports étiquetés par les actions exécutées par chaque processus et des liens également étiquetés par les actions de communication.

Un objet actif est constitué principalement de trois unités : l'unité encodant, le *body*, le comportement de l'objet que l'utilisateur aura programmé dans le code d'une méthode spécifique, nommée *runActivity*, l'unité encodant, le *proxy*, les objets futurs et enfin celle encodant (une approximation finie de) la *queue* d'attente des requêtes.

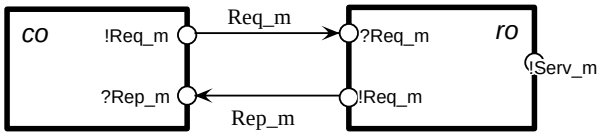


FIG. 3.7 – Réseau des processus *co* et *ro*

Nous associons alors un processus à chaque unité et le comportement global de cet objet est obtenu par composition parallèle de ces processus.

Pour chaque objet actif, nous construisons :

- une boite pour le body, $B(\mathcal{A})$;
- une boite pour la queue, $B(\mathcal{Q})$;
- une boite $B(\mathcal{F}_i)$ pour chaque proxy généré (à chaque nœud du dMCG de type *use(o.m)*) ;
- et une boite englobante $B(Name(O_i))$ caractérisant l'objet actif.

Les ports et les liens correspondants sont également construits de la manière suivante :

- à chaque méthode publique m de cet objet actif, les ports $!Serv\_m$ ("émission ordre de service") et $?Serv\_m$ ("réception ordre de service") sont ajoutés respectivement aux boites $B(\mathcal{A})$ et $B(\mathcal{Q})$ et le lien correspondant ;
- si le résultat de retour de m est différent de *void*, un port "émission de réponse", $!Rep\_m$ est alors ajouté à la boite $B(\mathcal{A})$;
- à chaque nœud du dMCG de type *use(o.m)*, un port $?Rep\_m$ est ajouté à la boite $B(\mathcal{F}_i)$ associée à ce nœud ; puis deux ports "émission de futur" $!Fut\_m$ et "réception de futur" $?Fut\_m$ sont également ajoutés respectivement aux boites $B(\mathcal{F}_i)$ et $B(\mathcal{A})$ et un lien entre eux.

À la fin de la construction de toutes les boites, des liens sont construits entre chaque boite $B(\mathcal{F}_i)$ et la boite de l'objet serveur.

Un exemple complet de réseau est donné dans la section 3.4.

3.3.2 Comportement d'un futur

Dans nos modèles nous préservons et reproduisons la caractéristique de synchronisation par *rendez-vous* via les futurs. D'abord au niveau du modèle abstrait, dans le dMCG un futur est qualifié soit par son point de définition, un nœud de type $call(ro.m)$, soit par son point d'utilisation du résultat, un nœud de type $use(ro.m)$.

Comme nous associons à chaque point *First_ Use* (Définition 7) un processus futur, nous construisons systématiquement au nœud correspondant dans dMCG un LTS caractérisant l'activité de ce futur (voir Figure 3.8).

Le choix des nœuds d'utilisation pour la génération du LTS, au lieu des points de définition, est délibéré. En effet, le nombre de points de définition de futurs peuvent être supérieur au nombre points d'utilisation. Dans le cas où, le résultat d'un appel n'est pas utilisé le modèle du proxy correspondant ne sera pas généré ; ce qui réduit la taille du modèle global.

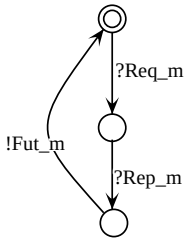


FIG. 3.8 – LTS de l'objet Futur

Le comportement d'un futur (Figure 3.8) se traduit par la réception d'une requête $?Req\_m$, émise par le body de l'objet auquel il correspond (et reçue simultanément par l'objet destination) ; puis la réception de la réponse, $?Rep\_m$, le message envoyé par l'objet destination autant qu'action $!Rep\_m$. Finalement, le renvoi de cette réponse au body comme étant la valeur du futur, $!Fut\_m$.

Notons que dans cette modélisation la valeur du futur n'est utilisée qu'une seule fois à chaque point du programme (donc sans recyclage), c'est à dire à une définition correspond une utilisation.

Les actions $?Req\_m$ et $!Fut\_m$ assurent la synchronisation de cet objet futur avec le body de l'objet auquel il correspond (Section 3.3.3) et l'action $?Rep\_m$ la synchronisation avec l'objet serveur (distant).

3.3.3 Règles de l'activité

Le comportement d'un objet actif est programmé dans le code de la méthode *runActivity* ; cette méthode détermine comment les requêtes affluant dans sa *queue de requêtes* sont servies ainsi que la manière dont sont adressées des requêtes à d'autres objets.

Les techniques d'analyse statique permettent d'établir pour chaque application :

- un graphe d'appel de méthodes étendu, dMCG, pour chaque classe d'objet actif,
- un prédicat, noté $\mathcal{L}oc(O)$, permettant de différencier les objets locaux (l'objet actif courant et ses objets passifs) des objets distants.

La construction du LTS de l'activité du corps de l'objet actif se fait par le parcours du dMCG de l'application, en partant de la méthode principale *runActivity* et par dépliage (inlining) de toutes les méthodes locales (de l'objet lui même et celles de ses objets passifs)

appelées depuis celle-ci. À chaque nœud du dMCG sera appliquée une règle de construction ; la terminaison de la procédure de dépliage est obtenue par détection des appels à des méthodes (ou mutuellement) récursives modulo une abstraction finie de la pile d'appel (voir Section 3.5).

La fonction de construction, notée $\rightsquigarrow_A$, décrivant le parcours du dMCG de la classe d'un objet actif et la création du LTS est définie par un ensemble de règles de transitions décrites dans un style SOS :

$$\frac{\{Premisse\}*}{< v = pattern,\ n,\ \mathcal{A},\ \mathcal{M},\ \mathcal{S}_c,\ \mathcal{S}_m > \ \rightsquigarrow_A\ < v',\ n',\ \mathcal{A}',\ \mathcal{M}',\ \mathcal{S}'_c,\ \mathcal{S}'_m >}$$

où

- $v = pattern$ est le nœud du dMCG en cours d'analyse et sa valeur,
- n est le dernier nœud créé du LTS,
- $\mathcal{A}$ le LTS en cours de construction. Le LTS vide est noté par $\emptyset$, et le LTS réduit à un seul nœud est désigné par le nœud lui même,
- $\mathcal{M}$ un mapping liant les nœuds du dMCG déjà visité à leurs homologues du LTS,
- $\mathcal{S}_c$ une pile de continuations ; une continuation est un couple (nœud du dMCG, nœud du LTS),
- $\mathcal{S}_m$ une pile d'appels de méthodes ; une pile vide est notée par [ ] et l'empilement d'un élément par $v : \mathcal{S}_m$.

Avant de définir les règles de construction, nous avons besoin de définir auparavant certaines constructions auxiliaires.

Définition 20 Opérateur de connexion. *Soient deux LTSs $\mathcal{A}$ et $\mathcal{A}'$ tels que $S \cap S' = \{s'_0\}$. La connexion de $\mathcal{A}$ et $\mathcal{A}'$ au nœud s'_0, notée $\mathcal{A} \triangleleft \mathcal{A}'$, donne le LTS $\mathcal{A}''$ défini par $S'' = S \cup S'$, $L'' = L \cup L'$, $s''_0 = s_0$, $\rightarrow''= (\rightarrow \cup \rightarrow')$*

Ce constructeur est utilisé dans la plupart des règles sous forme $\mathcal{A} \triangleleft (n \xrightarrow{x} n')$ pour relier une transition nouvellement créée au LTS en cours de construction.

Définition 21 Substitution. *L'opération consistant à remplacer un nœud n par un nœud m, i.e. remplace toute $n \xrightarrow{x} n'$ par $m \xrightarrow{x} n'$ et toute $n' \xrightarrow{x} n$ par $n' \xrightarrow{x} m$, est notée $\mathcal{A}[m/n]$.*

Cette opération est utile dans le cas des règles Loop et Join pour reproduire des boucles.

Donnons également deux fonctions : $fresh(n)$ et $init(n)$; désignant respectivement la fonction qui crée et retourne un nœud n (du pLTS) et la fonction qui transforme le nœud

n en un initial.

Les règles de construction du modèle du body (LTS de l'activité) d'un objet actif débutent à partir de l'état :

$$< v = root,\ \emptyset,\ \emptyset,\ \emptyset,\ [\,],\ [\,] >$$

root étant le nœud d'entrée de la méthode *runActivity*.

Notons que dans la suite, chaque nœud du dMCG visité est inséré dans le mapping $\mathcal{M}$ avec son homologue du LTS. Et pour améliorer la lisibilité les champs d'un tuple non modifiés après application d'une règle apparaissent en gris.

$$\frac{v_1 \rightarrow^T v_2 \quad init(fresh(n))}{\begin{array}{l}< v_1 = root,\ \emptyset,\ \emptyset,\ \emptyset,\ [\,],\ [\,] > \\ \quad \rightsquigarrow_A < v_2,\ n,\ n,\ \mathcal{M} \cup \{v_1 \mapsto n\},\ [\,],\ runActivity : [\,] >\end{array}} \text{(Ent\_Run)}$$

La première méthode analysée est *runActivity* ; à son nœud d'entrée, *root*, le nœud *initial* du LTS est alors créé, le nom de la méthode est empilé sur la pile d'appel et l'analyse se poursuit au nœud suivant (règle Ent_Run).

$$\frac{v_1 \neq root \quad v_1 \notin \mathcal{M} \quad v_1 \rightarrow^T v_2}{< v_1 = ent(m),\ n,\ \mathcal{A},\ \mathcal{M},\ \mathcal{S}_c,\ \mathcal{S}_m > \rightsquigarrow_A < v_2,\ n,\ \mathcal{A},\ \mathcal{M} \cup \{v_1 \mapsto n\},\ \mathcal{S}_c,\ m : \mathcal{S}_m >} \text{(Ent)}$$

En revanche à l'entrée d'une méthode quelconque autre que *runActivity* aucun nœud n'est créé ; mais le nom de celle ci est empilé sur la pile d'appel et nous poursuivons son analyse (règle Ent).

$$\frac{v_1 \notin \mathcal{M} \quad v_1 \rightarrow^C v_2 \quad v_1 \rightarrow^T v_2' \quad \mathcal{L}oc(O) \quad fresh(n')}{\begin{array}{l}< v_1 = call(O.m),\ n,\ \mathcal{A},\ \mathcal{M},\ \mathcal{S}_c,\ \mathcal{S}_m > \\ \quad \rightsquigarrow_A < v_2,\ n',\ \mathcal{A} \triangleleft (n \xrightarrow{m} n'),\ \mathcal{M} \cup \{v_1 \mapsto n\},\ (v_2', \perp) : \mathcal{S}_c,\ \mathcal{S}_m >\end{array}} \text{(L\_Call)}$$

À un appel à une méthode locale (définie comme étant observable) nous créons une transition étiquetée par le nom de la méthode invoquée (règle L_Call) ; le corps de cette méthode sera ensuite déplié (la partie du dMCG correspondant à cette méthode sera analysée). Le couple dont le premier élément est le nœud de continuation v_2' et le second la valeur $\perp$ est sauvegardé sur la pile de continuation pour une analyse ultérieure.

La valeur particulière $\perp$ représente un nœud du LTS présentement inconnu. Ce nœud non encore construit sera le dernier nœud de la méthode dépliée.

Le prédicat $\mathcal{L}oc(O)$ différencie les objets locaux (l'objet actif courant ainsi que ses objets passifs) des objets distants.

$$\frac{v_1 \notin \mathcal{M} \quad v_1 \rightarrow^T v_2 \quad v_1 \rightarrow^T v_3 \quad \ldots \quad v_1 \rightarrow^T v_k}{\begin{array}{l} < v_1 = seq,\ n,\ \mathcal{A},\ \mathcal{M},\ \mathcal{S}_c,\ \mathcal{S}_m > \\ \quad \rightsquigarrow_A < v_2,\ n,\ \mathcal{A},\ \mathcal{M} \cup \{v_1 \mapsto n\},\ (v_3, n) : \ldots : (v_k, n) : \mathcal{S}_c,\ \mathcal{S}_m > \end{array}} \text{(SEQ)}$$

Les nœuds séquence se rapportent aux points d'instructions séquentiels et de branchement. Donc aucune transition n'est créée. Dans le cas du branchement des couples de continuation, d'autres nœuds associés au nœud courant du LTS, sont empilés sur la pile de continuation pour un examen ultérieur (règle SEQ).

$$\frac{v_1 \notin \mathcal{M} \quad v_1 \rightarrow^T v_2}{\begin{array}{l} < v_1 = use(m),\ n,\ \mathcal{A},\ \mathcal{M},\ \mathcal{S}_c,\ \mathcal{S}_m > \\ \quad \rightsquigarrow_A < v_2,\ n,\ \mathcal{A} \triangleleft (n \xrightarrow{?Fut\_m} n'),\ \mathcal{M} \cup \{v_1 \mapsto n\},\ \ \mathcal{S}_c,\ \mathcal{S}_m > \end{array}} \text{(FUT)}$$

Le résultat promis (futur) d'un appel distant est requis au point d'utilisation de celui ci ; ainsi au nœud de type *use(m)* une transition étiquetée $?Fut\_m$, réception d'un futur, est alors créée (règle FUT).

$$\frac{n' = \mathcal{M}(v_1) \quad n'' \neq \perp}{< v_1,\ n,\ \mathcal{A},\ \mathcal{M},\ (v, n'') : \mathcal{S}_c,\ \mathcal{S}_m > \rightsquigarrow_A < v,\ n'',\ \mathcal{A}[n'/n],\ \mathcal{M},\ \mathcal{S}_c,\ \mathcal{S}_m >} \text{(LOOP)}$$

La règle LOOP est appliquée à tous types de nœuds ayant été déjà visités : dans le cas d'un appel récursif de méthode, dans une séquence en boucle ou un branchement en jointure. Le nœud courant du LTS, n, sera remplacé par n', le nœud déjà mappé avec v_1, de façon à former également une boucle dans le LTS en construction.

$$\frac{v_1 \neq ret \quad v_1 \rightarrow^T v_2 \quad n' = \mathcal{M}(v_1)}{< v_1,\ n,\ \mathcal{A},\ \mathcal{M},\ (v, \perp) : \mathcal{S}_c,\ \mathcal{S}_m > \rightsquigarrow_A < v_2,\ n',\ \mathcal{A}[n'/n],\ \mathcal{M},\ (v, \perp) : \mathcal{S}_c,\ \mathcal{S}_m >} \text{(JOIN)}$$

Cette règle s'applique à l'analyse de la dernière branche d'une jointure : la pile de continuation ne renferme plus de nœuds (en attente) appartenant à la méthode en cours d'analyse mais l'analyse de cette dernière n'est pas achevée ; la procédure se poursuit donc au prochain nœud séquence (règle JOIN).

$$\frac{v_1 \notin \mathcal{M} \quad n' \neq \bot}{< v_1 = ret,\ n,\ \mathcal{A},\ \mathcal{M},\ (v,n') : \mathcal{S}_c,\ \mathcal{S}_m > \rightsquigarrow_A < v,\ n',\ \mathcal{A},\ \mathcal{M} \cup \{v_1 \mapsto n\},\ \mathcal{S}_c,\ \mathcal{S}_m >} \text{ (RET1)}$$

$$\frac{}{< v_1 = ret,\ n,\ \mathcal{A},\ \mathcal{M},\ (v,\bot) : \mathcal{S}_c,\ m : \mathcal{S}_m > \rightsquigarrow_A < v,\ n,\ \mathcal{A},\ \mathcal{M} \setminus \mathcal{D}(m),\ \mathcal{S}_c,\ \mathcal{S}_m >} \text{ (RET2)}$$

À chaque nœud de retour, nous examinons si dans la pile de continuation demeurent des nœuds appartenant à la méthode en cours d'analyse (des nœuds d'alternatives). Si c'est le cas nous poursuivons en traitant au préalable ces derniers (règle RET1). Sinon, nous dépilons la méthode courante de la pile d'appel $\mathcal{S}_m$ et nous vidons du mapping $\mathcal{M}$ tous les nœuds de celle ci avant de poursuivre l'analyse au point de retour (règle RET2).

Remarquons que ces règles ne peuvent pas être appliquées si la pile de continuation est vide ; en effet, ce cas termine la procédure de construction du LTS.

$$\frac{v_1 \notin \mathcal{M} \quad v_1 \rightarrow^T v_2 \quad v_2 = call(m) \quad fresh(n')}{\begin{array}{l} < v_1 = serv,\ n,\ \mathcal{A},\ \mathcal{M},\ \mathcal{S}_c,\ \mathcal{S}_m > \\ \quad \rightsquigarrow_A < v_2,\ n',\ \mathcal{A} \triangleleft (n \xrightarrow{!Serv\_m} n'),\ \mathcal{M} \cup \{v_1 \mapsto n\},\ \mathcal{S}_c,\ \mathcal{S}_m > \end{array}} \text{ (SERV)}$$

Le nœud *serv*, suivi immédiatement par un nœud d'appel local *call(m)*, se rapporte à un point de *service* de la méthode m e.g. la primitive *serveOldest(m)*. Nous construisons une transition étiquetée $!Serv\_m$, émission de service, et nous poursuivons au nœud suivant (traitement de l'appel de la méthode) (règle SERV).

$$\frac{v_1 \notin \mathcal{M} \quad v_1 \rightarrow^T v_2 \quad fresh(n')}{\begin{array}{l} < v_1 = rep(m),\ n,\ \mathcal{A},\ \mathcal{M},\ \mathcal{S}_c,\ \mathcal{S}_m > \\ \quad \rightsquigarrow_A < v_2,\ n',\ \mathcal{A} \triangleleft (n \xrightarrow{!Rep\_m} n'),\ \mathcal{M} \cup \{v_1 \mapsto n\},\ \mathcal{S}_c,\ \mathcal{S}_m > \end{array}} \text{ (REP)}$$

Lorsque le résultat d'un appel de méthode est différent du *void*, le dMCG présente à la suite de la séquence *serv ; call(m)* le nœud de type *rep*. En ce point de retour d'un résultat une transition étiquetée $!Rep\_m$, émission de réponse à l'appelant distant, est alors créée (règle REP).

$$\frac{v_1 \notin \mathcal{M} \quad v_1 \rightarrow^T v_2 \quad \neg\mathcal{L}oc(O) \quad fresh(n')}{\begin{array}{l} < v_1 = call(O.m),\ n,\ \mathcal{A},\ \mathcal{M},\ \mathcal{S}_c,\ \mathcal{S}_m > \\ \quad \rightsquigarrow_A < v_2,\ n',\ \mathcal{A} \triangleleft (n \xrightarrow{!Req\_m} n'),\ \mathcal{M} \cup \{v_1 \mapsto n'\},\ \mathcal{S}_c,\ \mathcal{S}_m > \end{array}} \text{ (R\_CALL)}$$

Dans le cas d'un appel à une méthode d'un objet distant, nous créons une transition étiquetée $!Req\_m$, émission de requête vers l'objet distant (règle R_CALL) et l'analyse se poursuit au nœud séquence qui suit le nœud courant du dMCG (vu que ce nœud ne comporte pas d'arc d'appel $\rightarrow^C$ vers la méthode m).

Notons par $\mathcal{R}$ l'ensemble des règles précédentes. L'algorithme de construction du comportement de l'objet est décrit dans la Figure 3.9 par la procédure $\mathcal{P}$ qui prend en entrée le dMCG et $\mathcal{R}$ et retourne en sortie le LTS correspondant à l'activité du body.

Procedure $\mathcal{P}(dMCG,\ \mathcal{R}) : LTS$
(1) $v = root$;
(2) **While** il existe nœud à analyser **do**
choisir une $R_i \in \mathcal{R}$ à appliquer à v ;
avancer au nœud suivant **od** ;
(4) **return**(LTS) ;

FIG. 3.9 – Procédure de Construction

En partant du nœud initial de la méthode principale, *root*, la procédure parcourt tout le dMCG : applique à chaque nœud visité une règle (qui est unique voir Section 3.5) puis avance au nœud suivant : le successeur du nœud courant ou un nœud récupéré à partir de la pile de continuation.

3.3.4 Modélisation de la queue de requêtes

La queue de requêtes d'un objet actif fonctionne indépendamment de son corps ; elle accepte continuellement les requêtes (correctement typées) qui lui parviennent. Grâce à des primitives offertes par la librairie *ProActive* (Section 2.3) l'utilisateur peut choisir une politique de sélection et de service de ces requêtes.

Lorsqu'une requête est servie, elle est retirée d'abord de la queue puis la méthode correspondante est exécutée. Cette méthode peut à son tour faire appel à des méthodes d'autres objets actifs avant de retourner le résultat qui peut être un futur, une valeur donc possiblement non déterminée. Nous considérons que les événements de service et de réponse sont consécutifs si aucun appel à un objet distant ne survient dans le body de la méthode courante.

Généralement la queue de requêtes n'est pas bornée ce qui constitue la principale difficulté quant à sa modélisation : nous ne pouvons pas en donner un modèle suffisamment générique et approprié pour analyser tout type de propriétés. Néanmoins, moyennant des

abstractions, des approximations finies et les informations extraites à partir du code de l'application pour optimisation, nous obtenons des modèles praticables.

Le modèle de la queue est un LTS représentant le comportement de celle-ci à savoir : tous les ordres possibles des arrivées et les événements de service. Il se synchronise avec le corps du même objet par les actions de service $Serv\_m$ (règle SERV), et avec les objets distants par les actions de requête Req_m (règle R_CALL).

La construction de ce LTS se fait de manière générique en utilisant la notion les grammaires de graphe [54] en ayant l'ensemble des méthodes pouvant aboutir dans la queue et la taille de celle ci.

Par exemple, une queue de taille k recevant n méthodes : $m_1,\ m_2, \ldots, m_n$ et servant ces requêtes selon la primitive *FlushingServeOldestRequest* : sert la plus ancienne requête et supprime les autres, sera représentée par un LTS $(S, s_0, L, \rightarrow)$ défini par :

- $s_0 = \epsilon$,
- $S = \bigcup_{0<i\leq k} \{m_1,\ m_2, \ldots,\ m_n\}^i$,
- $L = \{?Req\_m_i,\ ?Serv\_m_i\}_i$,
- $\rightarrow = \{\bigcup_i (\omega \xrightarrow{?Req\_m_i} \omega m_i) | \omega \in S\} \cup \{m_i\omega \xrightarrow{?Serv\_m_i} \epsilon | \omega \in S\}$.

La figure 3.10 illustre une queue de taille 3 recevant deux méthodes $m1$ et $m2$, et servant indifféremment $m1$ or $m2$ dans l'ordre FIFO et puis en supprimant les autres.

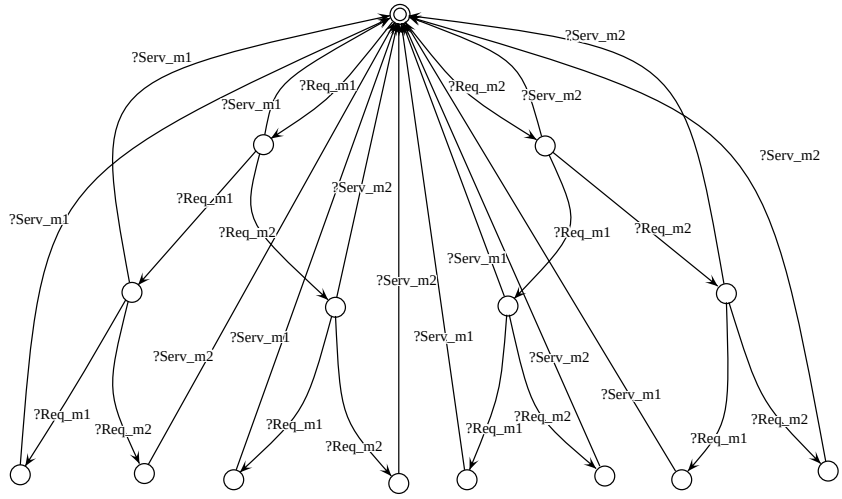


FIG. 3.10 – Une queue de taille 3 recevant 2 requêtes servie avec la primitive *FlushingServeOldestRequest*

Une queue de taille k recevant n requêtes : $m_1,\ m_2, \ldots, m_n$ est donnée par un LTS $(S, s_0, L, \rightarrow)$ défini par :

- $s_0 = \epsilon$
- $S = \bigcup_{0<i\leq k} \{m_1,\ m_2, \ldots,\ m_n\}^i$
- $L = \{?Req\_m_i,\ ?Serv\_m_i\}_i$,
- $\rightarrow = \{\bigcup_i(\omega \xrightarrow{?Req\_m_i} \omega m_i) | \omega \in S\} \cup \{m_i\omega \xrightarrow{?Serv\_m_i} \omega | \omega \in S\}$.

La figure 3.11 illustre un autre exemple (pris de [42]) de queue de taille 3 recevant deux méthodes, $TakeG$ et $TakeD$, servies dans l'ordre FIFO.

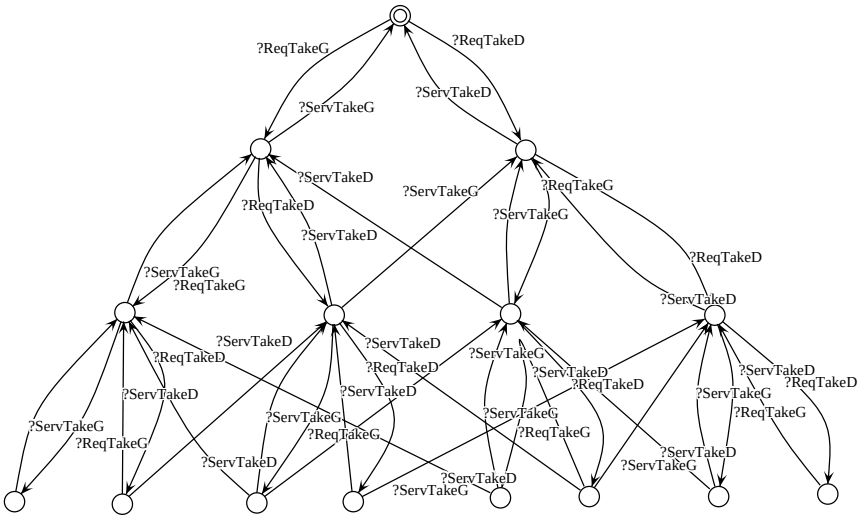


FIG. 3.11 – Une queue de taille 3 recevant 2 requêtes servie avec la primitive *ServeOldestRequest*

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.

À chaque fois qu'il est possible l'analyse statique fournisse des informations que les requêtes arrivant dans la queue sont traitées indépendamment dans le code l'utilisateur, par

les méthodes `serve{<name>}` : le comportement d'un objet actif est indépendant de l'ordre des arrivées des requêtes, nous adopterons alors une modélisation par queues séparées pour chaque requête.

Dans certains cas, et qui représentent une famille importante d'applications, il est également possible de prouver une propriété globale (d'une instantiation finie) du système, que chaque queue donnée est bornée par une taille fixe. Dans ces cas, nous construisons un modèle optimal des queues.

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 l'objet serveur, 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 du serveur.

La factorisation de la queue en queues séparées est la plus intéressante des optimisations ; elle est agencée grâce à l'analyse statique en collectant tous les services offerts par un objet actif.

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 3.13.

3.4 Exemple

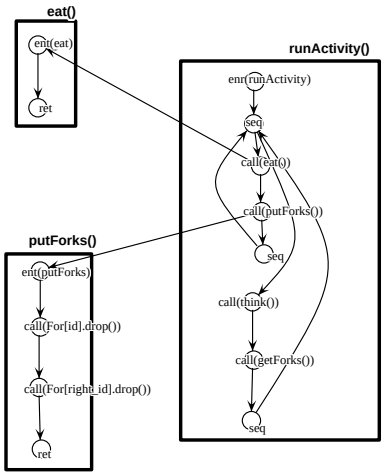
Nous illustrons notre travail par le problème classique du "dîner des philosophes", écrit dans une version *ProActive* distribuée, paramétrée par le nombre de philosophes que nous fixons ici à 3. Chaque philosophe et chaque fourchette est un objet actif, les philosophes communiquent avec les fourchettes par des appels de méthodes pour l'acquisition et la libération. À partir du code source de chaque classe d'objet et grâce à l'outil d'analyse nous obtenons le graphe d'appel de sa méthode principale, puis par application des règles sémantiques, à ce graphe, nous générons le modèle comportemental de cet objet.

3.4.1 L'objet Philosophe

Un philosophe 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.

Listing 3.1 – Comportement d'un objet Philosopher et le dMCG correspondant

```
public class Philosopher implements RunActive {
protected hasBothForks=false;
...
public void getForks()
  { ProActive.waitFor(Fork[id].take());
    ProActive.waitFor(Fork[right_id].take());
    hasBothForks = true;
    }

public void putForks()
  { Fork[id].drop();
    Fork[right_ind].drop();
    hasBothForks = false;
    }

public void runActivity(Body b)
  { while (true) {
        if (hasBothForks) {
                eat();
                putForks();
        } else {
                think();
                getForks();
        }
}}
```



L'activité d'un philosophe est encapsulée dans la méthode principale *runActivity* et représentée par un graphe d'appel (Listing 3.1) ; celle-ci est une boucle infinie *while (true)* entre les appels aux méthodes : `eat`, `putForks` , `think` et `getForks`.

Cette activité est décrite dans le modèle (Figure 3.12) par la boite (englobante) `Philosopher` contenant trois boites : $\mathcal{A}$, $\mathcal{F}_1$ et $\mathcal{F}_2$ correspondant respectivement au body, et les deux objets futurs associés aux points d'appels aux fourchettes (droite et gauche).

Le body de l'objet philosophe (LTS de la boite $\mathcal{A}$) décrit les actions : émission des actions visibles (`eat`, `think`, `putForks` et `getForks`) vers le monde extérieur ; émission

de la requête `!Req_*.take` aux objets fourchette (droite et gauche) ; réception du résultat (futur) de sa requête `?Fut_*.take` correspondante ; et émission de la requête `!Req_*.drop` 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.

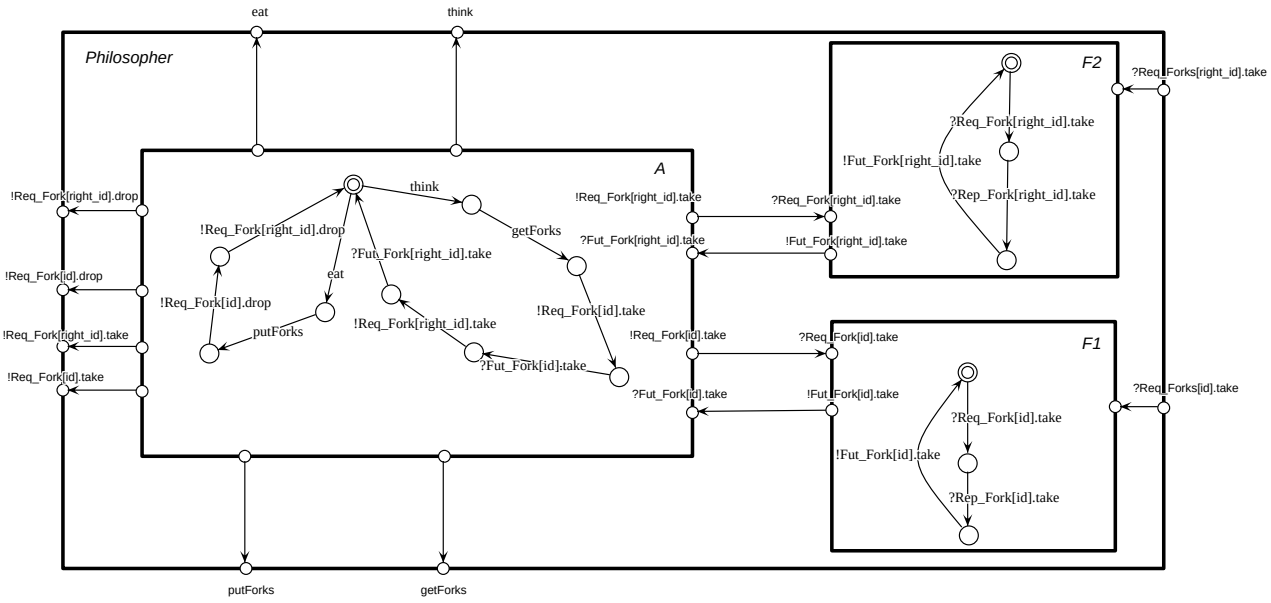


FIG. 3.12 – Comportement d'un Philosophe : LTSs des Futurs et de runActivity

Comme cela est spécifié dans le code, les réponses aux requêtes `take` sont attendues par les philosophes dans un état bloquant ; aucune action autre que `?Rep_*.take` n'est possible immédiatement après l'action `!Req_*.take`.

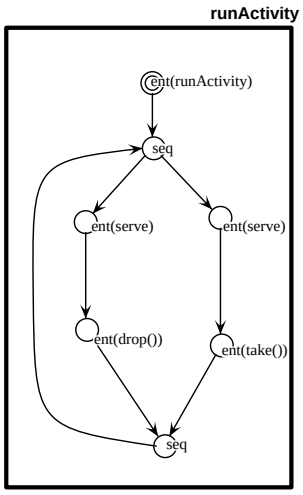
Le comportement des futurs associées aux points d'émission de requêtes (LTSs des boites $\mathcal{F}_1$ et $\mathcal{F}_2$) décrivent la réception de la requête `?Req_*.take` du body ; puis réception du résultat des objets fourchettes `?Rep_*.take` ; enfin, émission de celui ci `!Fut_*.take` vers le body.

3.4.2 L'objet Fourchette

L'objet fourchette reçoit les requêtes `take` et `drop` des philosophes. Son comportement est un service successif de ces requêtes, service contrôlé par la variable d'état `FreeFork`. Ainsi l'activité de la fourchette est également une boucle infinie entre l'appel à la primitive de service `serveOldestWithoutBlocking` de la requête `take` ou `drop`.

Listing 3.2 – La classe de l'objet Fork et le dMCG correspondant

```
public class Fork implements RunActive {
protected FreeFork=true;
...
public   ObjectForSynchro take()
  {      FreeFork=false;
         layout.updateFork(id,4);
         return new ObjectForSynchro()
  }

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

public void runActivity(Body myBody)
  {      while(true){
         if (this.FreeFork)
              myBody.serveOldestWithoutBlocking("take");
         else
              myBody.serveOldestWithoutBlocking("drop");
 }}
```



La Figure 3.13 présente le modèle de chaque objet fourchette : boite du body ($\mathcal{A}$ et boite de la queue ($\mathcal{Q}$. Notons que la requête `take` est dissociée en `takeR` et `takeL` selon qu'elle émane du philosophe gauche ou du philosophe droit. Les requêtes `take*` et `drop` ne sont pas traitées de la même manière (LTS de la boite $\mathcal{A}$) ; en effet, après son service `Serv_take*` une requête `take*` retourne un résultat `!Rep_take*` qui se synchronise avec l'attente d'un philosophe, tandis que `drop` ne l'est pas et son exécution est modélisée uniquement par l'action `Serv_drop` de la fourchette.

Par ailleurs, l'analyse statique produit des informations intéressantes pour la modélisation complète de la queue des requêtes des fourchettes ; les requêtes `Req_take*` et `Req_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 `Req_take*` et `Req_drop`. Par conséquent, nous pouvons utiliser un modèle dans lequel les queues séparées des requêtes `take` et `drop` sont simplement mises en parallèle (boite $\mathcal{Q}$_take et boite $\mathcal{Q}$_drop).

En outre, nous donnons un modèle (LTSs des boites $\mathcal{Q}$_*) dans lequel nous faisons l'hypothèse qu'à un moment donné, pas plus de deux requêtes `take*` peuvent se trouver dans la queue $\mathcal{Q}$_take* et pas plus d'une requête `drop` dans la queue $\mathcal{Q}$_drop . Cette hypothèse n'est correcte que si le système fonctionne convenablement, et nous prouverons qu'en effet c'est le cas.

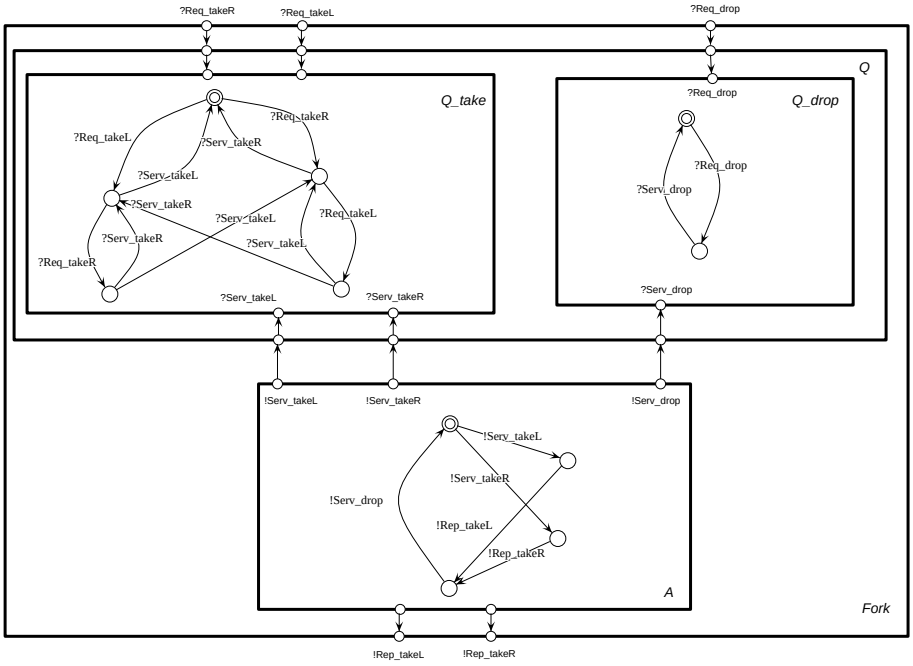


FIG. 3.13 – Comportement d'une fourchette : LTSs du body et de la Queue (factorisée)

3.4.3 Table du Dîner

Listing 3.3 – Classe de création des objets et initialisation de l'application

```
public class DinnerLayout{
...
for (int n = 0; n < 3; n++) {
                ...
Phils[n] = (Philosopher)ProActive.newActive(Philosopher.class.getName(), args);
                ...
Fks[n] = (Fork)ProActive.newActive(Forks.class.getName(), args);
                ...
    }
```

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 :

Les objets actifs (3 philosophes et 3 fourchettes) sont représentés par des boites hiérarchiques avec des ports et la communication entre eux représentée par des liaisons entre ces ports. La synchronisation de ces derniers est modélisée par le réseau, figure 3.14 ; chaque philosophe i communique avec les fourchettes voisines i et $i+1$ (gauche et droite) : par les messages pour les appels aux méthodes d'acquisition de fourchettes `Req_*.take` qui se synchronisent sur leur retour `Rep_*.take`, et les messages `Req_*.drop` qui eux

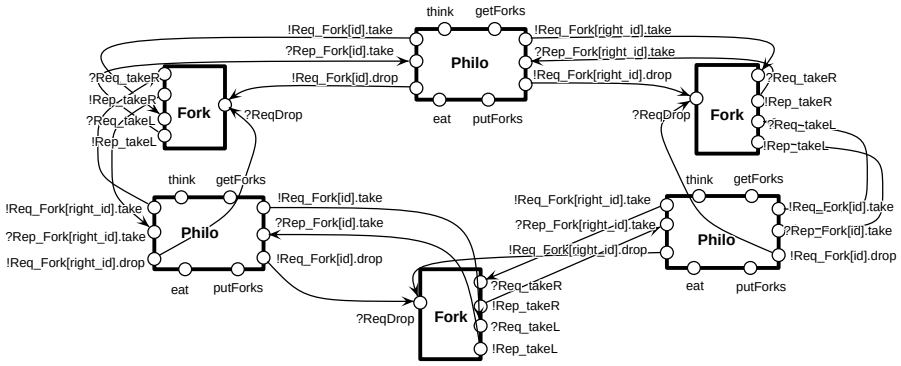


FIG. 3.14 – Réseau de synchronisation Philosophes-Fourchettes

n'attendent pas de retour. Apparaissent également les actions `think`, `eat`, `getForks` et `putForks` qui sont simplement des événements observables, sans synchronisation.

3.4.4 Vérification de propriétés d'une application

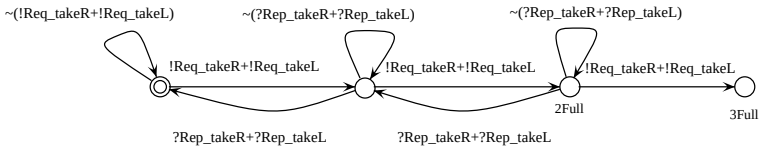
À présent nous avons spécifié les éléments de notre système. Dans la pratique, tous les automates et le réseau de ce chapitre ont été dessinés en utilisant l'éditeur graphique ATG ; celui-ci produit aussi des fichiers FC2 de chaque élément. L'ensemble des éléments, automates et réseau, sont *linkés* (liés) entre eux en un seul fichier FC2 représentant tout l'arbre. Et en utilisant des outils de vérification fc2tools et CADP nous avons vérifié certaines propriétés sur le modèle global.

Détection d'inter-blocages : les outils détectent en effet un deadlock et produisent également un chemin menant de l'état initial à l'état bloqué, état d'où ne sort aucune transition, et qui peut être visualisé à l'aide d'ATG.

Preuve de complétude : Nous montrons 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'abstrac-

tion 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, ce qui implique que notre propriété est prouvée. Nous obtenons le même résultat avec la queue des `drop`, qui n'a besoin, elle, que d'une seule place.

3.5 Propriétés de la procédure de construction

Dans cette section, nous formalisons des propriétés structurelles des modèles générés et des propriétés de la procédure de construction ; d'abord nous prouvons que si tous les domaines des objets sont finis alors la topologie du réseau est finie. Ensuite, nous montrons que la procédure de construction est déterministe et termine.

3.5.1 Vocabulaire

Afin de simplifier la lecture des propriétés et de leurs preuves, rappelons et définissons plus précisément le vocabulaire des notions manipulées par les règles de construction.

Le tuple $< v, n, \mathcal{A}, \mathcal{M}, \mathcal{S}_c, \mathcal{S}_m >$ introduit par toutes les règles est composé de :

- v le nœud courant du dMCG,
- n le nœud (généré) courant du LTS ,
- $\mathcal{A}$ le LTS en cours de construction,
- $\mathcal{M}$ le mapping des nœuds,
- $\mathcal{S}_c$ la pile de continuations,
- $\mathcal{S}_m$ la pile d'appels.

La procédure de construction utilise :

- un dMCG $(id, V, \rightarrow^T, \rightarrow^C)$; la procédure démarre à partir de la méthode d'activation *runActivity*, $id = C.runActivity$.
- $\mathcal{D}(id)$ l'ensemble des nœuds d'une méthode nommée id.
- $\mathcal{R}$ l'ensemble des règles.

Nous utiliserons également dans la suite les notions suivantes :

- une *méthode en cours d'analyse* pour désigner une méthode demeurant dans la pile de méthodes.
- une *nœud analysé* pour désigner le nœud sur lequel une règle a déjà été appliquée et sa méthode est *en cours d'analyse* ; un nœud analysé dont la méthode est complètement analysée est considéré non analysé.

Pour établir la preuve de finitude de la topologie et la terminaison de la procédure, nous avons besoin des notions suivantes :

3.5.2 Propriétés de la pile d'appel

Notre pile d'appel $\mathcal{S}_m$ est une abstraction de la pile d'appel de méthodes concrète de runtime Java ; nous limitons la manière dont les appels de méthodes apparaissent dans la pile. Avant de prouver sa finitude, commençons par montrer quelques invariants.

Invariant 1

$\forall m, \forall v \in \mathcal{D}(m)$ *nous avons :*

$v \in \mathcal{M} \Rightarrow v$ *est analysé* *(1)*

$m \in \mathcal{S}_m \Leftrightarrow ent(m) \in \mathcal{M}$ *(2)*

Preuve. Nous montrons les deux invariants séparément :

1. Découle directement à partir des règles. Dès qu'un nœud est examiné, toutes les règles différentes de Loop, Join et Ret2, insèrent explicitement celui-ci dans le mapping $\mathcal{M}$ $(v \mapsto n)$; quant aux règles Loop et Join, elles garantissent que v y est déjà (prémisse $\mathcal{M}(v)$), par contre, la règle Ret2 soustrait de $\mathcal{M}$ tous les nœuds de la méthode (parmi eux v) et termine l'analyse de celle-ci. Par conséquent, v demeure dans $\mathcal{M}$ jusqu'à la fin de l'analyse de la méthode à laquelle il appartient.
2. ($\Rightarrow$) $m \in \mathcal{S}_m$ implique que la règle Ent_Run ou bien la règle Ent a été appliquée au nœud $ent(m)$ (nœud d'entrée de m) ; en effet, ce sont les seules règles qui poussent m dans $\mathcal{S}_m$ et ces deux règles insèrent explicitement $ent(m)$ dans $\mathcal{M}$. De plus, l'unique règle retirant m de $\mathcal{S}_m$ et soustrayant $ent(m)$ de $\mathcal{M}$ est la règle Ret2 à la fin d'analyse.
   ($\Leftarrow$) Analogue au précédent. Étant donné que, l'opération d'insertion du nœud $ent(m)$ dans $\mathcal{M}$ et de la méthode m dans $\mathcal{S}_m$ sont simultanées. □

Notons par $\mathcal{U}$ l'ensemble des méthodes du dMCG. Durant l'analyse, nous avons :

Invariant 2 *À chaque étape, la séquence $\mathcal{S}_m$ est un sous-ensemble (sans répétition) de $\mathcal{U}$, donc la pile d'appel de méthodes est de taille finie.*

Preuve. Supposons que $m \in \mathcal{S}_m$, alors $v = ent(m) \in \mathcal{M}$ (Invariant 1 (2)) et donc v analysé (Invariant 1 (1)). Si m est rappelée une seconde fois au cours de son analyse (appel récursif), la règle L_Call sera donc appliquée. Puis, le prochain nœud à examiner sera $ent(m)$ (d'après la règle L_Call et la définition 18) d'où la règle à appliquer sera Ent. Or d'après l'hypothèse $ent(m) \in \mathcal{M}$, la prémisse ($v_1 \notin \mathcal{M}$) de la règle Ent est non vérifiée donc cette règle ne peut être appliquée et par conséquent m ne peut être empilée une fois de plus. □

3.5.3 La procédure de construction

Les règles de construction d'un LTS sont appliquées aux nœuds d'un dMCG supposé bien-formé (Définition 18). Bien entendu, cette hypothèse ne peut être vraie que si le dMCG est obtenu à partir d'un code source correct. Si tel est le cas, nous montrons que la terminaison de la procédure de construction du LTS est alors garantie.

Déterminisme

Soit $< v,\ n,\ \mathcal{A},\ \mathcal{M},\ (v', n') : \mathcal{S}_c,\ \mathcal{S}_m >$ un état quelconque d'un tuple à un moment donné de l'analyse. Selon la valeur des champs du tuple une règle de l'ensemble $\mathcal{R}$ est appliquée.

Nous récapitulons dans le tableau de la figure 3.5.3 l'ensemble des prémisses des règles.

L'ensemble des règles $\mathcal{R}$ vérifient deux propriétés.

Propriété 1 *L'ensemble des règles $\mathcal{R}$ est exhaustif.*

Preuve. Puisque tous les types de nœuds sont représentés par au moins une règle et par conjonction des prémisses nous constatons que tous les cas sont adressés.

Propriété 2 *Les règles sont exclusives deux à deux.*

Preuve. En examinant l'ensemble des règles $\mathcal{R}$, nous constatons que pour :

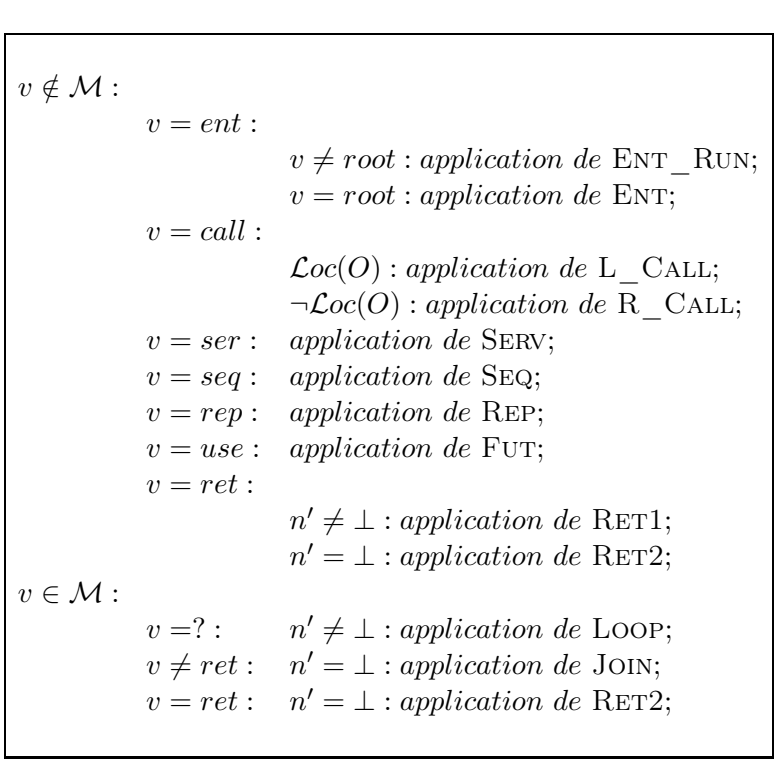


FIG. 3.15 – Ensemble des prémisses des règles

1. les règles filtrant sur le type du nœud, il est immédiat de voir qu'elles sont exclusives, nous avons une règle par type. À l'exception des nœuds de type *ret* ayant deux règles RET1 et RET2. Toutefois, grâce aux prémisses de ces règles les différents cas sont distingués de manière univoque :

   (a) si $v \notin \mathcal{M}$ première analyse de v et selon la valeur du nœud de continuation :
      i. si $n' \neq \perp$ alors la règle RET1 est appliquée (d'autres nœuds d'une alternative de la méthode attendent dans la pile de continuation) ;
      ii. sinon, $n' = \perp$ la règle RET2 est appliquée (fin d'analyse de la méthode).

   (b) sinon, $v \in \mathcal{M}$ cas d'un branchement sur v et :
      i. si $(n' = \perp)$ alors
         A. si $v \neq ret$ alors alors la règle JOIN est appliquée ;
         B. sinon, la règle LOOP est appliquée ;
      ii. sinon, $(n' \neq \perp)$ la règle RET1 est appliquée (d'autres nœuds d'une alternative de la méthode attendent dans la pile de continuation).

2. les règles sans filtre (la règle LOOP et la règle JOIN), elles s'appliquent à tous les types de nœuds. Cependant,

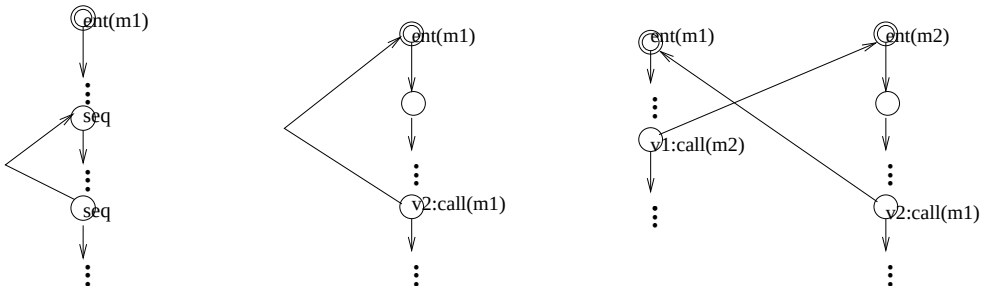
(a) la prémisse $n' = \mathcal{M}(v_1)$ (càd, $v_1 \in \mathcal{M}$) les disjoignent des règles de $\mathcal{R} \setminus \{\text{Loop}, \text{Join}, \text{Ret2}\}$.

(b) En outre,

i. la règle Loop est exclusive des règles Ret2 et Join grâce à la prémisse $n'' \neq \perp$;

ii. la règle Join est exclusive avec la règle Ret2 grâce à la prémisse $v_1 \neq ret$. □

En supposant un dMCG bien formé (Définition 18) et en se basant sur ces propriétés (1 and 2), nous pouvons énoncer que l'ensemble des règles $\mathcal{R}$ décrivent un système de règles dont l'évolution peut être déterminée exactement.



(a) Boucle (b) Récursion (c) Récursion mutuelle

Fig. 3.16 – Différents cas de cycles dans un graphe d'appel de méthodes

Terminaison

Un dMCG peut contenir des cycles, des boucles ou des appels récursifs, (voir Figure 3.16), donc la procédure peut analyser indéfiniment un nœud.

Pour montrer la terminaison de la procédure de construction et la finitude du modèle généré, nous montrons, préalablement, quelques résultats utiles pour la suite.

Définition 22 Chaine d'appels. *Désignons par chaine d'appels σ une séquence ordonnée $m_0^{v_0}.m_1^{v_1}\ldots m_n^{v_n}$ de noms de méthodes m_i attachés à des nœuds du dMCG. Pour toute méthode m_i de la chaine tel que $i < n$, v_i attaché à celle-ci est le nœud d'appel à la méthode m_{i+1}).*

En fait, une chaine d'appels est le chemin parcouru depuis la méthode principale m_0 de toutes les méthodes appelées associées, à leurs points d'appel, jusqu'à un point de programme (et sa méthode). Nous utilisons la notion de chaine d'appels pour représenter les nœuds du dMCG, au lieu du nœud lui-même, afin de distinguer tous les chemins d'accès à un nœud donné qui sont possiblement multiples.

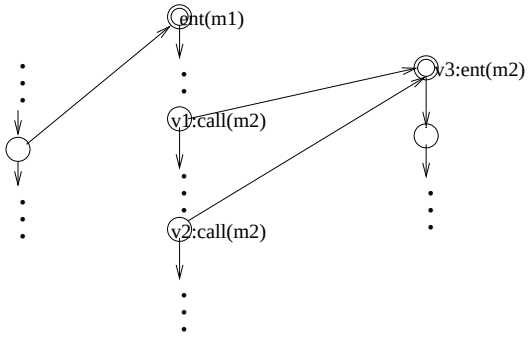


FIG. 3.17 – Appels multiples à une même méthode

Ceci étant dit, un nœud peut être analysé plusieurs fois, autant de fois que la méthode à laquelle il appartient est dépliée ; par exemple, dans la figure 3.17 le nœud v_3 (et tous les nœuds de m_2) sont atteints deux fois par la séquence $\sigma.m_1^{v_1}.m_2^{v_3}$ et par $\sigma.m_1^{v_2}.m_2^{v_3}$. Ainsi, le cardinal l'ensemble des σs collectés durant toute l'analyse est supérieur au cardinal de l'ensemble des nœuds du dMCG.

Dans le tableau de la figure 3.18 nous donnons, comment calculer une chaine d'appels σ à chaque analyse d'un nœud. À chaque application d'une règle $< v_k,\ n,\ \mathcal{A},\ \mathcal{M},\ \mathcal{S}_c,\ \mathcal{S}_m > \rightsquigarrow_A < v'_k,\ n',\ \mathcal{A}',\ \mathcal{M}',\ \mathcal{S}'_c,\ \mathcal{S}'_m >$, une chaine courante σ' est calculée à partir de la précédente σ selon le type de cette règle.

ENT *or* ENT_RUN :	σ	$\rightarrow$	$\sigma' = \sigma.top(\mathcal{S}'_m)^{v_k}$	
RET1 :	$\sigma = \bar{\sigma}.m_i^{v_j}$	$\rightarrow$	$\sigma' = \begin{cases} \bar{\sigma} & si\ v_j = ret \\ \sigma & sinon \end{cases}$	(1) (2)
DEFAULT :	$\sigma = \bar{\sigma}.m_i^{v_j}.m_{i'}^{v_{j'}}$	$\rightarrow$	$\sigma' = \begin{cases} \bar{\sigma}.m_i^{v_k} & si\ v_{j'} = ret \\ \bar{\sigma}.m_i^{v_j}.m_{i'}^{v_k} & sinon \end{cases}$	(1) (2)

FIG. 3.18 – Procédure de calcul des σs

avec *top* une fonction qui retourne l'élément au sommet d'une pile.

Remarquons que les règles ENT et ENT_RUN sont les seules qui augmentent la longueur de σ. En revanche, celle-ci décroît lors de l'application d'une règle quelconque qui suit l'application d'une règle retour (RET1 ou RET2). En fait, après apllication de la règle RET2 : σ décroît si le dernier nœud de la séquence est de type *ret* or la règle RET1 ne

modifie pas la valeur des nœud d'un séquence donc la modification est réalisée par la règle RET2.

Remarquons également que la règle RET1, contrairement aux autres, ne modifie pas les nœuds de la séquence.

En fait, à chaque application d'une règle la longueur de la séquence σ générée est égale à celle de la pile d'appel $\mathcal{S}_m$ à cet instant.

Lemme 1 *$\mathcal{S}_m$ est une projection de σ sur les v_is.*

Preuve. En effet, σ est une version décorée de la pile $\mathcal{S}_m$ à un instant donnée. □

Notons par Σ l'ensemble des σs collectés après analyse de tous les nœuds du dMCG et application des règles appartenant à $\mathcal{R} \setminus \{\text{LOOP}, \text{JOIN}, \text{RET2}\}$; l'ensemble des règles susceptibles de construire un nœud du LTS.

Nous avons :

Lemme 2 *Σ est un ensemble fini.*

Preuve. Considérons l'arbre des chaines d'appel (Figure 3.19). Il est borné en profondeur par la taille maximale de la pile d'appel de méthodes, et à chaque nœud, on a un branchement borné par le nombre de nœuds de la méthode courante. L'ensemble des nœuds d'un dMCG, notamment les nœuds d'appel de méthodes, étant fini, la pile d'appel de méthodes est également bornée (Invariant 2) d'où Σ est fini. □

Pour établir la preuve de terminaison nous utilisons à la fois l'ensemble des chaines d'appels Σ et la pile de continuation $\mathcal{S}_c$.

Notons par, Σ^- et $\mathcal{S}_c^-$ (Σ^+ et $\mathcal{S}_c^+$) l'état de Σ et $\mathcal{S}_c$ avant (après) application d'une règle.

Nous définissons un ordre lexical sur le produit cartésien $\Sigma \times \mathcal{S}_c$:

Définition 23 $\Sigma^+; \mathcal{S}_c^+ \prec \Sigma^-; \mathcal{S}_c^- \Leftrightarrow \begin{cases} \Sigma^+ \supsetneq \Sigma^- & si\ \Sigma^+ \neq \Sigma^- \\ \mathcal{S}_c^+ \subsetneq \mathcal{S}_c^- & sinon \end{cases}$

Lemme 3 *$\prec$ est bien fondé.*

Preuve. $\Sigma^+ \supsetneq \Sigma^-$ est bien fondé parce que Σ est finie (lemme 2). $\mathcal{S}_c^+ \subsetneq \mathcal{S}_c^-$ est bien fondé par ce que la longueur de $\mathcal{S}_c$ est décroissante. □

D'autre part nous avons,

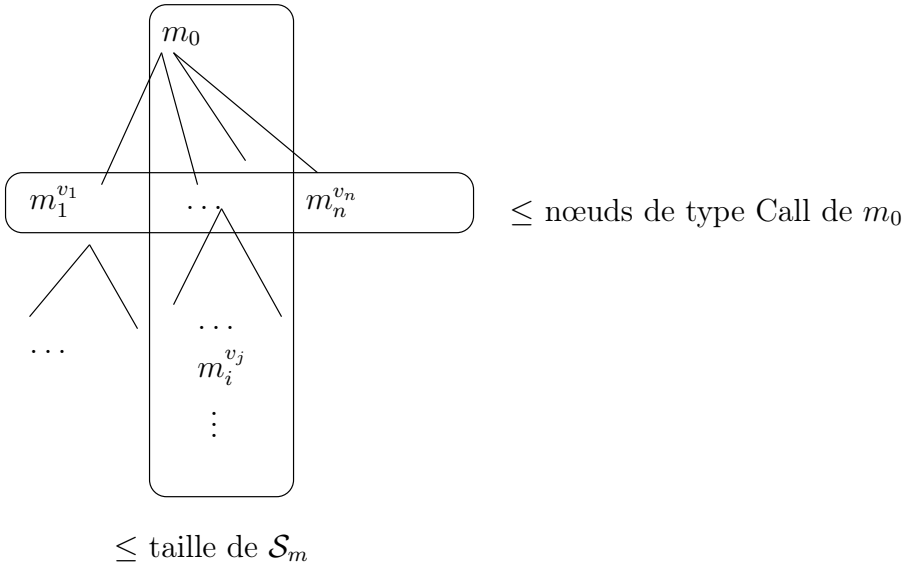


FIG. 3.19 – Nombre maximal de σs

Lemme 4 *À chaque application de règle, l'ordre lexical est strictement décroissant.*

Preuve. Σ étant fini et croissant, donc à chaque application de règle appartenant à $\mathcal{R} \setminus \{\text{RET1}, \text{LOOP}\}$, nous avons $\Sigma^+ \supsetneq \Sigma^-$. Par contre, pour les règles RET1, LOOP et JOIN même si $\Sigma^+ = \Sigma^-$, nous avons $\mathcal{S}_c \subsetneq (v, n) : \mathcal{S}_c$. □

Nous déduisons alors que les règles sont appliquées un nombre fini de fois, i.e., l'instruction de boucle (2) de la procédure $\mathcal{P}$ (voir Figure 3.9) termine.

Proposition 1 *$\mathcal{P}$ termine.*

Preuve. Découle directement des lemmes 3 et 4, étant donné que le nombre de nœud à analyser pendant la construction du LTS est décroissant, par conséquent ces nœuds sont analysés un nombre fini de fois. □

3.5.4 Le LTS construit

À présent, considérons le LTS construit après examen de tous les nœuds du dMCG de l'application étudiée.

Lemme 5 *Le LTS généré, à la fin de la procédure de construction, est fini.* □

Preuve. Étant donné que la procédure de construction termine (Proposition 1) et qu'à chaque application de règle au plus un nœud (du LTS) est créé (voir règles) donc le nombre de nœuds est nécessairement fini. □

Considérons maintenant le LTS global de l'application, obtenu par composition de l'ensemble des LTSs de différents unités : le LTS associé au body, les LTSs des proxies et le LTS de la queue.

Nous avons :

Lemme 6 *Le résultat de la composition d'un nombre n de LTSs finis,* $(S_i, s_{o_i}, L_i, \rightarrow_i)$, $i = 1 \ldots n$, *est un LTS fini* $(S, s_0, L, \rightarrow)$.

Preuve. Trivial, $S = \Pi_{i=1}^{n}(S_i)$. □

Partant, de la définition des LTSs des proxies et de l'hypothèse de la queue de requêtes comme étant finis. Nous en alors déduisons :

Proposition 2 *Le LTS global est fini.*

Preuve. Conséquence directe des lemmes 5 et 6. □

3.6 Bilan

Les résultats obtenus dans cette partie [44] de notre travail sont intéressants ; nous proposons des modèles comportementaux pour la vérification d'applictions concurrentes et distribuées, en l'occurrence d'applications *ProActive*. Toutefois, l'approche adoptée dans ce travail et les formalismes choisis sont également applicables à d'autres technologies telles que CORBA/RMI voire à des frameworks de spécifications de langages telles que la plate-forme, CREOL.

En outre, l'analyse de nos applications, contrairement aux travaux faits dans ce sens [58, 12] examine même des programmes renfermant des méthodes récursives. Cependant, cette analyse est assez limitée, d'une part, parce que nous posons des hypothèses assez fortes et pas toujours praticables, à savoir, la nécessité d'une analyse statique assez fine pour classer et différencier précisément les objets, et fournir une énumération finie d'instances pour chaque objet actif de l'application. D'autre part, en raison des abstractions fortes que nous appliquons à l'alphabet des actions, l'analyse n'est pas assez fine pour des systèmes "value-passing", systèmes dont les processus échangent des messages portant des données.

Nous allons résoudre ces contraintes dans les chapitres qui suivent.

Chapitre 4

Modèles comportementaux paramétrés

Abstraction : *C'est une simplification, en présence de l'objet concret infiniment complexe, qui consiste à considérer un élément de l'objet comme isolé et comme constant. ALAIN, Gallimard, 1961.*

En réalité les applications étudiées ne sont pas toujours des systèmes finis, de plus nous souhaitons à chaque fois que c'est possible représenter toute une famille (éventuellement infinie) de processus (généralement finis) par un modèle unique. De sorte qu'à la vérification automatique une (ou des) instantiation(s) de cette représentation sera exploitée.

Pour représenter un système infini, il est nécessaire de le *paramétriser*, de le représenter en fonction de *paramètres* tels que le nombre permis d'exécution d'une action, la taille de certaines données comme le nombre de messages à transmettre par les différentes composantes du système, etc. Ces paramètres dont le domaine est généralement infini sont souvent liés entre eux et obéissent à des contraintes. Par contre, comme il n'est pas toujours possible de raisonner sur les comportements d'un système pour toutes les valeurs possibles de ses paramètre, il est nécessaire d'*instancier* celui-ci, d'en examiner un nombre fini d'occurrences.

Dans ce chapitre, nous proposons donc une modélisation du comportement des systèmes (possiblement infini) par des modèles dits *paramétrés* qui sont, en fait, une extension de ceux définis dans le chapitre précédent. Dans la Section 4.1 rappelons la technique d'abstraction de variables ; puis 4.2 nous verrons, ce que c'est un paramètres et le statut d'une donnée dans nos modèles ; avant de définir dans les Sections 4.3 et 4.4 la syntaxe de leurs formalismes mathématiques et graphiques.

4.1 Abstraction vs instantiation

L'abstraction est une technique très générale, utilisée dans le domaine de la vérification afin de réduire l'espace d'états. Cette méthode a pour but de réduire et de simplifier un modèle en ignorant certains aspects de celui-ci par exemple en supprimant des variables ou en changeant leur domaines. Il est possible de démontrer mathématiquement que la satisfaction d'une propriété par le système abstrait implique sa satisfaction par le système de départ. La méthode d'abstraction représente une famille de techniques. Parmi elle, nous trouvons l'abstraction sur les variables (données).

L'abstraction sur les variables peut être réalisée de différentes façons. Par suppression simplement des variables à abstraire. Ce qui amène à rendre les gardes moins fortes et à faire disparaître certaines relations entre les variables. Cependant, souvent les propriétés sont directement ou indirectement liées à la valeur des variables. La suppression de ces variables rend les événements moins forts et fait disparaître certaines relations entre variables ce qui peut entraîner des comportements erronés qui n'existent pas dans le système réel. Toutefois, ce type de méthode s'applique bien dans les cas d'indépendance de données et dans les autres cas peut être mené en concert avec l'examen de dépendance entre les données grâce aux techniques d'analyse de flot de données.

Une autre approche consiste à maintenir toutes les variables mais de redéfinir leurs domaines de manière à conserver l'information nécessaire à la vérification d'une propriété donnée. Cette technique fait partie de l'interprétation abstraite [61, 60] qui est une théorie visant à définir un calcul mathématique approché, par abstraction, de propriétés de programme. Cette abstraction implique une perte d'information, qui ne permet pas de répondre à toutes les questions. Cependant toutes les réponses données sur un système abstrait sont justes dans le système détaillé, mais certaines questions ne peuvent trouver de réponses exactes dans le système abstrait. Cette technique est largement utilisée dans le domaine de la vérification, voire pour résoudre des problèmes industriels.

Pour la construction de modèles paramétrés nous utilisons fortement la technique d'abstraction, nous devons garantir la préservation des propriétés. Cleavland et Riely dans leurs travaux [57] assurent qu'une modélisation d'une abstraction finie d'un domaine de données simples conservait les propriétés de sûreté et de vivacité. En revanche, leur résultats s'appliquent à des systèmes "value-passing" [90] ; ce qui s'accorde bien, dans notre cas, aux paramètres échangés dans messages entre les processus. Mais qu'il faudra étendre aux paramètres de processus.

Une opération inverse de l'abstraction est l'*instantiation*. L'instantiation est obtenue par affectation à chaque élément abstrait d'une valeur concrète. Cette opération est, dans le cas général, nécessaire pour établir la vérification de propriétés d'un systèmes paramétré ; en effet, étant donnée qu'il n'existe pas d'outils de vérification permettant la vérification de systèmes paramétré, l'instantiation consiste à remplacer les valeurs symboliques (paramètres) par un nombre fini d'occurrences de leurs valeurs et ainsi pouvoir vérifier pour une (ou des) instance donnée.

4.2 Données Simples

Un des buts de l'utilisation des paramètres, est le besoin de raisonner à un certain niveau d'abstraction sur des variables et des structures de données dont les domaines de valeurs sont non bornés, par exemple les variables entières ou les canaux de communication (files d'attente) de taille arbitrairement grande, etc.

Lorsque les paramètres d'un modèle ont un domaine suffisamment simple (dénombrable), les travaux mentionnés à la section 4.1 nous permettent d'abstraire ces domaines par des domaines abstraits finis, et d'en déduire une abstraction finie du modèle du système. Cette approche nous permettra d'utiliser des outils de vérification de modèles classiques tout en préservant une large classe de propriétés de sûreté et de vivacité.

En pratique, nous nous restreignons à une notion de "types simples" permettant d'utiliser cette approche. Il n'est pas dans notre intention ici de discuter des limites concrètes d'applicabilité de l'approche dans le cadre, par exemple, des types disponibles dans les bibliothèques Java. Ce point sera important bien sûr, lorsqu'il s'agira de mettre en place des outils logiciels traitant des applications réalistes.

Définition 24 *Types de données simples*[1] *:*

- *Booléens.*
- *Énumérations finies.*
- *Caractères et Chaînes de caractères de longueur fixe.*
- *Entiers (*`Nat` *et* `Int`*) ou Intervalles (ouverts et fermés) sur les entiers.*
- *Tableaux d'objets de type simple, de taille entière fixe.*
- *Objets structurés ayant un nombre fini de champs nommés, chacun étant de type simple.*

Dans nos systèmes, et donc dans nos modèles, nous avons deux types de variables : les variables classiques représentant les valeurs de données manipulées par le programme et portées par les messages, semblables à celles des théories des systèmes dits "value-passing" tels que [90, 85, 57] ; et les variables représentant les “identificateurs” de processus qui elles ont un autre statut ; elles encodent la topologie du système distribué, permettent d'identifier les différentes instances des objets, de coder l'adressage des messages, mais aussi la “dynamicité” de la topologie comme dans les théories de channel-passing, tel que le π-calcul [89, 99] et process-passing, tel que le calcul des Ambients [48]. Bien que nous nous restreignons à des topologies (paramétrées) statiques, il est nécessaire de distinguer ces deux types de variables, du fait qu'elles ont des rôles différents dans le comportement et notamment, lors de l'application de la technique d'instantiation.

[1] Les cas des tableaux et des structures ne seront pas traités dans ce mémoire.

4.3 Modèles Paramétrés

À présent nous souhaitons exprimer une infinité de configurations dans une représentation symbolique ; les événements à observer sur nos modèles seront donc des actions portant des données symboliques : les arguments des messages, les valeurs des variables d'états et les paramètres de processus. Par conséquent, toutes les notions et les modèles définis préalablement sont étendus à des notions et modèles dits *paramétrés*, de façon à tenir compte des données et leurs domaines. La sémantique des modèles étendus que nous définissons ici est fortement liée à celle des STGAs (graphes de transitions symboliques avec affectation) [85].

4.3.1 Systèmes de transitions étiquetées paramétrées

Puisque les systèmes de transitions modélisent tout à fait le comportement d'un processus nous maintenons ces structures pour représenter également un ensemble (infini) de processus ; par contre, il est nécessaire d'étendre leur définition de façon à pouvoir représenter des *actions paramétrées*.

La première définition étend les actions de la section 3.1.1, par des expressions contenant des variables :

Définition 25 Expressions et Actions paramétrées. *Soient Var un ensemble dénombrable de variables, typées par des types simples, Σ_{Expr} une signature contenant les constantes et les opérateurs de nos types simples, $Expr$ l'algèbre de termes (bien typés) construite à partir de $\Sigma_{Expr} \cup Var$, $BExpr \subset Expr$ l'ensemble des expressions booléennes.*

L'ensemble des actions paramétrées $pAct$ est une algèbre de termes obtenue en étendant $Expr$ par un type $Action$ et par une signature (contenant les constructeurs d'actions) dépendant du domaine d'application.

Nous montrerons en page 80 (Définition 33) comment spécialiser cette définition dans le cas des actions modélisant les applications ProActive.

Définition 26 Actions Gardées avec Affectation *Une "action gardée avec affectation" $ga \in GAct$ est un tuple $(b, \alpha(\overrightarrow{x}), \overrightarrow{e})$ où b est une expression booléenne, $\alpha(\overrightarrow{x}) \in pAct$ une action paramétrée, et $\overrightarrow{e}$ est une expression de la forme $\overrightarrow{v := x}$ un ensemble fini d'affectations avec $x \in Expr$.*

Un *système de transitions étiquetées paramétré*, noté *pLTS*, est un LTS dont les labels sont bien entendu des actions paramétrées (des événements et des expressions éventuellement gardées) et dont les états sont munis d'un ensemble de variables d'état :

Définition 27 pLTS. *Un système de transitions étiquetées paramétré (pLTS comme parameterized labelled transition system) est un tuple :*

$$pLTS \stackrel{def}{=} (K, S, s_0, L, \rightarrow)$$

où :

1. $K = \{k_i\}$ *est un ensemble fini de paramètres.*
2. S *est l'ensemble des états ; à chaque état* $s \in S$ *est associé un ensemble fini de variables, notées* $\overrightarrow{v_s}$.
3. $s_0 \in S$ *est l'état initial.*
4. L *est un ensemble d'étiquettes* $(b, \alpha(\overrightarrow{x}), \overrightarrow{e}) \in GAct$. *Dans chaque affectation* $\overrightarrow{v := x}$, v *est une variable de l'état cible de la transition, et* x *une expression contenant des variables de l'état source de la transition.*
5. $\rightarrow$ *est l'ensemble des transitions :* $\rightarrow \subseteq S \times L \times S$.

De fait, un pLTS définit toute une famille de LTSs semblables (autant de LTSs que d'éléments dans les domaines des paramètres K).

4.3.2 Réseau de synchronisation paramétré

Nous définissons la composition parallèle d'une famille de processus (possiblement infinie) et leurs communications par ce que nous appelons un *réseau de synchronisation paramétré*, noté *pNet*. Un pNet a un nombre fini d'arguments, chacun pouvant être paramétré, donc représenter une famille de processus arguments. Par exemple, on peut définir un anneau de processus comme un pNet a un seul argument, dont le paramètre représente l'indice des processus dans l'anneau. Pour chacun des arguments, on spécifie sa sorte, c'est à dire ses moyens de communication avec le reste du réseau.

Un pNet comprend un ensemble de contraintes de synchronisation entre objets paramétrés, étendant la notion de vecteur de synchronisation d'Arnold [28]. Pour représenter des changements de topologies dans un réseau, nous introduisons une notion d'état de ce dernier. Un pNet est alors un transducteur opérant sur les systèmes de transition de ses arguments, à la manière des "Lotomatons" de A. Lakas pour les expressions ouvertes de processus Lotos [81].

Définition 28 Sorte paramétrée. *Une sorte paramétrée est un ensemble, noté* pI, *d'actions paramétrées,* $pI \subseteq pAct$.

Définition 29 pNet. *Un pNet est un tuple :*

$$pNet \stackrel{def}{=} < pA_G, H, T >$$

où

1. *$pA_G \subset pAct$ est l'ensemble des actions globales du réseau.*
2. *$H = \{pI_i, K_i\}_{i=1..n}$ est un ensemble fini de trous (arguments).*
3. *le transducteur T est un pLTS $T = (K_G, S_T, s_{0_T}, L_T, \rightarrow_T)$, tel que $\forall \overrightarrow{v} \in L_T, \overrightarrow{v} =< l_t, \alpha_1^{k_1}, \ldots, \alpha_n^{k_n} >$ où $l_t \in pA_G$, $\alpha_i \in pI_i \cup \{*\}$ et $k_i \in K_i$.*

Notons que le K_G du transducteur T est l'ensemble des paramètres globaux du pNet. Chaque trou dans le pNet a une sorte (un ensemble d'actions locales) pI_i et un ensemble fini de paramètres K_i.

4.4 Structures graphiques

À l'instar des modèles finis, les modèles ainsi étendus pour prendre en compte les paramètres et les variables sont décrits graphiquement. Pour ce faire, nous étendons aussi notre langage de spécification graphique.

4.4.1 pLTS graphique

La représentation graphique d'un pLTS est semblable à celle d'un LTS, auquel il faut : ajouter une liste de paramètres, ajouter les variables d'état, et enrichir les labels des transitions qui sont désormais des actions paramétrées. La syntaxe abstraite d'un pLTS graphique est :

Définition 30 pLTS graphique. *Un pLTS graphique, noté $\widehat{pLTS}$, est un triplet :*

$$\begin{array}{lll} \widehat{pLTS} & \stackrel{def}{=} & (Titre,\ \mathcal{R},\ \mathcal{L}) \\ Titre & \stackrel{def}{=} & (Nom, \mathcal{X}) \end{array}$$

avec

- *$Titre$ est l'identifiant du pLTS, constitué par son nom, qui est une chaîne de caractères, et un ensemble de paramètres, noté $\mathcal{X}$, qui sont des noms de variables suivis éventuellement de leurs domaines.*
- *$\mathcal{R}$ est un ensemble de ronds, dotés éventuellement de variables, et dont certains peuvent être marqués.*
- *$\mathcal{L}$ est un ensemble de liens entre les ronds, chaque lien est de la forme $R1 \xrightarrow{l} R2$ avec $l \in GAct$.*

Exemple, la Figure 4.1 est le graphique d'un pLTS représentant le comportement d'un buffer dont le titre *Buffer(Max : int)* a un seul paramètre *Max* de type *int*. Ce pLTS a un seul état (initial) muni d'une variable N également de type *int*, et deux liens orientés étiquetés par des actions gardées avec affectation écrites dans le style des modèles d'actions ProActive. Par exemple l'action $[0 < x <= Max - x]?P(p).put(x) \rightarrow N := N + x$ décrit

l'événement de réception de la requête de dépôt de x items par le producteur $P(p)$ et la mise à jour de la variable N.

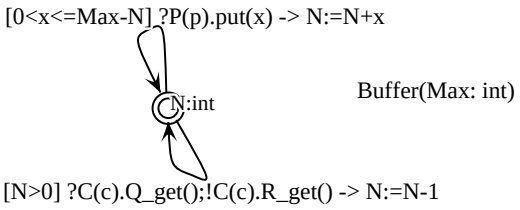


FIG. 4.1 – pLTS du Buffer

Ce simple graphique représente, en fait toute une famille de différents pLTSs, buffers de capacité $Max = 1 \ldots N$.

Dans la suite quand $\alpha(\overrightarrow{x})$ (ou $\overrightarrow{e}$) est vide le label est simplement noté par $[b]\overrightarrow{e}$ (ou $[b]\alpha(\overrightarrow{x})$).

4.4.2 pNet graphique

De même que dans le cas fini, nous nous limitons ici aux configurations statiques de réseaux (le transducteur n'a qu'un seul état).

Un pNet statique est décrit par un ensemble de boites paramétrées, dotées donc de variables définissant leurs instances ainsi que de variables locales et dont les ports et les liens sont étiquetés par des actions paramétrées (appartenant à $pAct$).

Le titre d'une boite ne se réduit pas au nom, il inclut des paramètres et des variables. La syntaxe du titre d'une boite B est :

$$Name(B) \stackrel{def}{=} (Nom, \mathcal{X}, \mathcal{Y})$$

Nous distinguons graphiquement les paramètres $\mathcal{X}$ (qui identifient les instances des processus) des variables $\mathcal{Y}$ (qui identifient les instances des états des pLTS). Dans nos dessins nous avons choisi de représenter entre crochets les paramètres et entre parenthèses les variables des boites.

La Figure 4.2 montre la représentation graphique d'un pNet représentant une partie du système Producteur/Consommateur décrit dans [36]. Le buffer représenté par la boite $Buf[](Max)$, l'ensemble des paramètres est vide (un seul buffer dans le système), par contre il a une variable Max. La boite $Cons[c]()$ encode un ensemble de processus (consommateurs) défini par le domaine des valeurs du paramètre c.

Chaque consommateur réclame un objet du buffer ($!Buf.Q\_get()$) et attend une

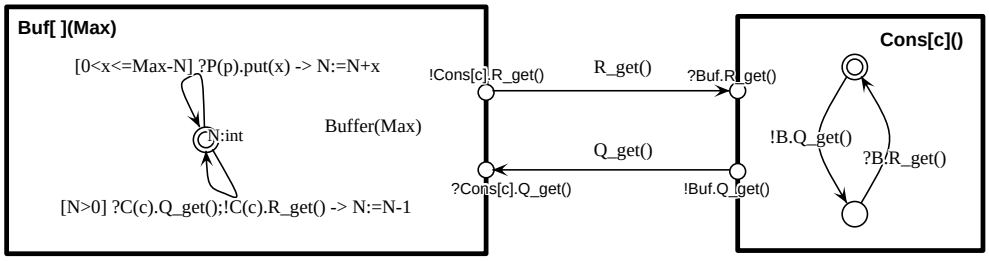


FIG. 4.2 – Exemple d'un pNet graphique

réponse ($?Buf.R\_get()$). Par ailleurs, le buffer reçoit des requêtes de toute une famille de consommateurs ($?Cons[c].Q\_get()$) et y répond ($!Cons[c].R\_get()$).

4.4.3 Spécifications graphiques paramétrées vs modèles paramétrés

L'étape suivante consiste à donner la traduction de nos objets graphiques paramétrés dans les modèles de la section précédente.

Définissons d'abord la traduction d'un LTS graphique, $< (nom, params), \mathcal{R}, \mathcal{L} >$ ayant un seul rond marqué dit initial, en un LTS $< K, S, s_0, L, \rightarrow >$ par la fonction $[\![.]\!]_{pL}$ définie par :

$$[\![< (nom, params), \mathcal{R}, \mathcal{L} >]\!]_{pL} =< K, S, s_0, L, \rightarrow >$$

tel que

$$\begin{array}{lcl} K & = & params \\ S & = & R \\ s_0 & = & R_0 \text{ étant le rond initial} \\ L & = & \{l \in \mathcal{L} / R \xrightarrow{l} R'\} \\ \rightarrow & = & \{s \xrightarrow{ga} s' / \exists R \xrightarrow{ga} R' \in \mathcal{R} \wedge s = R, s' = R'\} \end{array}$$

Avant de définir la fonction $[\![.]\!]_{pN}$ traduisant un réseau graphique $(B_g < B_1, \ldots, B_n >, \mathcal{L})$ en son homologue $< A_G, I, T = (S_T = \{s_0\}, s_0, L_T, \rightarrow_T) >$, définissons des fonctions d'extraction de données.

Notons par $Params$ la fonction qui extrait les paramètres d'un objet : les paramètres d'une boite $Params(B) = \{\mathcal{X} / B \in \mathcal{B}\}$ et les paramètres attachés aux ports. Par exemple, dans figure 4.2, $Params(Cons[c]()) = \{c\}$, par contre, $Params(Buf[](Max)) = \emptyset$ et $Params(!Cons[c].R\_get()) = \{c\}$.

Notons également par Act la fonction qui extrait l'action (sans le processus) d'un label. Ainsi, $Act((!Cons[c].R\_get()) =!R\_get()$.

La fonction $[\![.]\!]_{pN}$ est définie par :

$$[\![(B_g < B_1, \ldots, B_n >,\ \mathcal{L})]\!]_{pN} =< A_G, I, T = (S_T = \{s_0\}, s_0, L_T, \rightarrow_T) >$$

tel que

$$\begin{array}{lll}
A_G & = & \{l \ / \ \exists B_i.p \xrightarrow{l} B_g.p' \in \mathcal{L}\} \\
\forall i / B_i \neq B_g, \quad H_i & = & \left\{ \begin{array}{lll} pI & = & \{Actions(B_i)\} \\ K & = & \mathcal{X}_i \cup \mathcal{Y}_i \ / \ Name(B_i) = (Name_i, \mathcal{X}_i, \mathcal{Y}_i) \end{array} \right. \\
L_T & = & \{\overrightarrow{v} / \forall B_i \in \mathcal{B} \ tel \ que \ \exists B_i.p \xrightarrow{l} B_j.p' \in \mathcal{L} \quad \overrightarrow{v} =< l, v_1, \ldots, v_n >, \\
 & & v_i = Act(Label(p))_i^{k_i} \ avec \ k_i = Params(B_i), \quad v_j = Act(Label(p'))_j^{k_j} \\
 & & avec \ k_j = Params(Label(p)) \ et \ \forall k \neq i, j \ v_k = *\} \\
\rightarrow_T & = & (s_0, \overrightarrow{v}, s_0) \quad \forall \overrightarrow{v} \in L_T
\end{array}$$

Notons qu'on pourrait assez simplement étendre notre syntaxe graphique pour prendre en compte les réseaux non statiques, en représentant explicitement le transducteur comme un pLTS graphique attaché à la boite englobante, et en indiquant les changements d'état sur les liens entre les boites internes.

Exemple. Voyons explicitement cette traduction sur un exemple ; soit le réseau de la figure 4.3 qui décrit la communication entre une famille de producteurs (processus $Prod$) et une famille de consommateurs (processus $Cons$). La boite $Prod$ est paramétrée par le nombre (p) d'instance du processus producteur et par la variable (Max) désignant le nombre maximal d'éléments produit par ce processus. La boite $Cons$ est paramétrée par le nombre (c) d'instance processus consommateur.

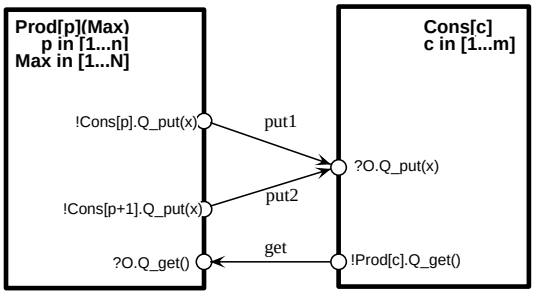


FIG. 4.3 – Un autre exemple d'un pNet graphique

Le modèle correspondant à cette figure serait $< A_G, I, T = (S_T = \{s_0\}, s_0, L_T, \rightarrow_T$

$) >$ défini par :

$$
\begin{array}{lcl}
A_G & = & \{put1, put2, get\} \\
H & = & \left\{ \begin{array}{lcl} pI & = & \{\{?Q\_get(), !Q\_put(x)\}, \{!Q\_get(), ?Q\_put(x)\}\} \\ K & = & \{\{p\ in\ [1 \ldots n], Max\ in\ [1 \ldots N]\}, \{c\ in\ [1 \ldots m]\}\} \end{array} \right. \\
L_T & = & \{< put1, !Q\_put^p(x), ?Q\_put^p(x) >, < put2, !Q\_put^p(x), ?Q\_put^{p+1}(x) >, \\
 & & < get, ?Q\_get^c(), !Q\_get^c() >\} \\
\rightarrow_T & = & (s_0, \overrightarrow{v}, s_0) \quad \forall \overrightarrow{v} \in L_T
\end{array}
$$

Notons que les processus $Prod[p]$ et $Cons[c]$ occupent respectivement les positions 1 et 2 dans les vecteurs de synchronisation (éléments de L_T).

4.5 Bilan

Nous venons de voir une extension des formalismes vus dans le chapitre précédent afin de prendre en compte les paramètres et les données.

Ces formalismes étendent également les formalismes proposés dans la littérature. En effet, la notion de système de transitions paramétré est introduite par Arnold (Définition 2) mais celui-ci ne donne pas une définition formelle à part que d'exprimer l'ajout de paramètres au niveau des états et des transitions ; ces paramètres sont définis par des ensembles finis, dans le cas de nos modèles paramètres sont des variables appartenant à des ensembles énumérables.
Il en est de même pour ce qui est des réseaux paramétrés, bien qu'il existe des outils comme Promela [21] permettant la description de processus paramétrés, il n'existe pas de description formelle de ces derniers.

Remarquons que la traduction d'un pLTS graphique en son homologue formel est trivial, ce qui n'est pas le cas des réseaux du fait des paramètres des instances des processus ; en effet, les paramètres des vecteurs de synchronisation sont construits à partir de ceux des boites, pour identifier le processus, et ceux de leurs ports, pour identifier l'instance.

Bien entendu, il a été nécessaire d'étendre également le langage intermédiaire *fc2* afin de faciliter la spécification des modèles ainsi enrichis et de pouvoir générer automatiquement des instantiations finies et utiliser des outils de vérification supportant la syntaxe de ce langage.

Chapitre 5

Modèles paramétrés des applications *ProActive*

Paramètre : *Variable en fonction de laquelle on exprime chacune des variables d'une équation.*
Le Petit Robert, ed 1999

Nous nous intéressons maintenant à l'utilisation de nos modèles paramétrés pour la vérification d'applications *ProActive* [35]. Nous décrivons dans ce chapitre une méthode pour construire les modèles hiérarchiques paramétrés capturant le comportement d'applications *ProActive* distribuées. Plus précisément, nous décrivons cette construction pour un noyau de la librairie *ProActive*, contenant les notions de base du modèle de programmation distribué (objets actifs, appels asynchrones, attente par nécessité, etc.). Nous discuterons dans la suite de la thèse certaines extensions à ce noyau, en particulier la prise en compte des mécanismes de communication de groupe.

La prise en compte des données dans les modèles comportementaux nous oblige naturellement à modifier les principes utilisés dans le cas finitaire. La principale concerne la représentation des méthodes récursives : en l'absence de données, nous pouvions les représenter comme de simples boucles dans le modèle comportemental. En présence de données, nous modélisons chaque méthode par un processus, et chaque appel de méthode par une communication entre processus ; il n'est plus besoin (et plus possible dans le cas général) de déplier les appels de méthodes pour construire le modèle. La structure obtenue est beaucoup plus proche de la structure du code source que dans le cas finitaire, et la preuve de finitude de la procédure devient triviale. En même temps, nous gagnons beaucoup en terme de taille du modèle généré : le nombre de processus élémentaires est linéaire en fonction du nombre de méthodes dans le code source, et le LTS de chaque méthode est linéaire dans la taille du code.

Tout comme pour les modèles finis la génération des modèles comportementaux

paramétrés d'une application se fait à partir du graphe d'appel de celle-ci ; cette structure abstraite doit donc fournir l'information nécessaire pour élaborer les paramètres : contexte et données. Dans la section 5.1, nous définissons donc une extension du dMCG de façon à prendre en compte les données manipulées et utiles dans ces exécutions, c'est à dire *le flot de données*. Nous verrons également dans la section 5.2 que la prise en compte des données induit nécessairement des transformations dans la topologie du réseau et par conséquence dans la formalisation des applications.

La construction des modèles paramétrés se fait également de manière hiérarchique et par composition de (sous-)modèles. En fait la structure est ici plus finement décomposée que dans notre construction finitaire, du fait du codage séparé de chaque méthode. La structure des modèles est donc plus profonde, et il sera d'autant plus important de pouvoir profiter de cette structuration dans les outils de vérification.

Nous terminerons ce chapitre par une modélisation paramétrée de notre exemple des philosophes 5.4.

5.1 Graphe d'appel de méthodes paramétré

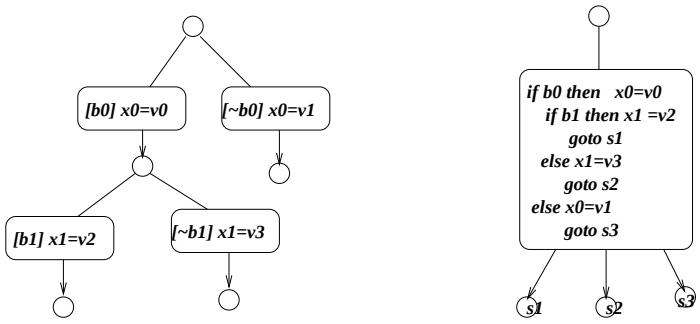
La structure de graphe d'appel de méthode telle qu'elle est utilisée dans le chapitre 3 reste un outil important pour décrire les résultats de l'analyse de classe et de flot de contrôle. Nous l'enrichissons ici par des informations supplémentaires afin de calculer les paramètres. Ces informations qui sont essentiellement des données sont collectées directement à partir du code source grâce aux techniques d'analyse de flot de données [108, 91].

Le graphe d'appel ainsi enrichi sera appelé *graphe d'appel de méthodes paramétré* et noté *pMCG* (de l'anglais parameterized MCG) ; il est doté au niveau de ses nœuds et de ses arcs de données et d'opérations : paramètres d'appels de méthodes (expressions contenant des variables), affectations, tests et branchements, instructions, etc.

Méta-transitions Si l'on ajoute des morceaux d'information séquentielle au niveau des arcs du graphe d'appel, on risque d'une part de devoir dupliquer ces morceaux de code (en particulier des tests) à plusieurs endroits dans le LTS représentant une méthode, et d'autre part d'augmenter le nombre de noeuds dans ce LTS. Lorsqu'ensuite on voudra calculer des produits de synchronisation de ces systèmes, on sera confronté au phénomène d'explosion du nombre de nœuds et de transitions du graphe.

Pour réduire cette complexité nous modifions la structure d'une transition en permettant qu'un arc (transition) ait un nœud source et plusieurs nœuds cibles. Une telle transition s'appellera une *méta-transition* (voir exemple Figure 5.1). Une méta-transition ne contient pas d'événements "observables" au sens de la sémantique comportementale, c'est à dire pas de communication locale ou distante.

Les labels d'une méta-transition sont des programmes séquentiels simples contenant



(a) (b)

FIG. 5.1 – Branchements (a) versus méta-transition (b)

exclusivement des expressions arithmétiques, des affectations et des branchements. On l'obtient simplement à partir du code source de la méthode (ou plus précisément d'une séquence du code intermédiaire Jimple) [107, 106] en remplaçant dans ce code les instructions non séquentielles (invocations, retour, etc.) par des "goto" vers les états de sortie de la méta-transition.

Définition 31 Méta-Transition (notée MT). *Le code d'une méta-transition est un programme :*

Prog	:=	*Stmt ; Prog*
Stmt	:=	*var = Expr*
		| **if** *Expr* **then** *Stmt* **[else** *Stmt* **]**
		| **goto** *lab*
Expr	:=	*opExpr { Expr } **
		fieldValue.

où var est une variable ou un champ d'un objet, lab dénote un nœud cible, opExpr une opération prédéfinie de Jimple, telle qu'une opération arithmétique, et fieldValue un accès à un champ d'un objet.

Nous pouvons maintenant donner la définition formelle d'un graphe d'appel de méthodes paramétré, pour un objet actif :

Définition 32 pMCG. *un pMCG est un tuple :*

$$pMCG \stackrel{def}{=} \left\langle M, m_0, V, \xrightarrow{calls}_C, \xrightarrow{succs}_T \right\rangle$$

où :

1. *M est un ensemble de noms de méthodes,*

2. $m_0 \in M$ *le nom de la méthode principale (initiale),*
3. V *est l'ensemble des nœuds, dont les types sont :*

   - *ent* $(m, args)$ *nœud d'entrée de la méthode* $m \in M$*, avec les paramètres* $args$*,*
   - *call* $(calls)$ *nœud d'appel de méthode (locale, distante ou statique),*
   - *pp* (lab) *un point programme étiqueté* lab*,*
   - *ret* (val) *un point de retour avec la valeur du résultat* val*,*
   - *serve* $(calls, mset, pred, mode)$ *la sélection d'une requête* $m \in mset$ *à partir de la queue, telle que* $pred(m) = true$*.*
   - *use* (fut, val) *un point d'utilisation d'une valeur future.*

4. $\xrightarrow{calls}_C$*, l'ensemble des transitions d'appel (pour les nœuds de type call ou serve), avec* $\forall c \in calls(n), \exists n'. < n, c, n' > \in \xrightarrow{calls}_C$ *et chaque appel* c *est soit :*

   - *Remote* $(o, m, args, var, fut)$ *pour un appel de méthode* m *d'un objet distant* o *via le proxy* fut*,*
   - *Local* $(o, m, args, var)$ *pour un appel de méthode* m *d'un objet local* o*,*
   - $Unknown$ $(o, m, args, var, fut)$ *quand l'analyse statique ne peut déterminer si l'appel est local ou distant,*
   - *Static* $(m, args, var)$ *pour un appel à une méthode* m *statique de la bibliothèque Java ou* ProActive*.*

5. $\xrightarrow{succs}_T$*, l'ensemble des méta-transitions. Tous les nœuds (sauf ret) sont munis d'une unique méta-transition,* $< succs(n) =< MT, N >$*, tel que* $< n, MT, N > \in \xrightarrow{succs}_T$*.*

Commentaires :

- Le pMCG est construit après analyse de classe, qui a calculé exactement les différentes valeurs possibles des types des objets, ainsi que des méthodes appelées à chaque point d'invocation. Les noms de méthodes sont donc des noms complets (incluant la classe et les types des arguments). La structure des noeuds *call* prend en compte l'imprécision de cette analyse : ils ont autant d'arcs sortants que de valeurs possibles pour le type de la méthode appelée.
- Les transitions ont naturellement une structure enrichie par les valeurs (symboliques) qu'elles portent, de même que les nœuds. Nous supposons qu'à chaque méta-transition, nous pouvons distinguer si celle-ci renferme une variable globale, accédé en lecture ou en écriture, ou non.

$$\begin{array}{rcl} MT & := & Access(var); Code \\ & | & Update(var, exp); Code \\ & | & Code \end{array}$$

Ces informations seront utilisées pour construire les actions paramétrées des modèles.

- Il y a des cas où l'analyse statique ne peut pas déterminer statiquement si un objet est local ou distant. C'est le cas par exemple pour un appel dans une boucle à une méthode d'un objet actif paramétré `P[k]`, lorsque l'objet courant fait partie de ce tableau d'objets. On utilise dans ce cas un appel "Unknown" ; cette imprécision restera présente dans le modèle généré, et ne sera résolue que lors de l'instanciation du modèle.
- Les appels aux méthodes statiques nous obligent à nous poser une question de principe : pour préserver la sémantique de *ProActive*, il faut respecter la règle de non-partage de données entre objets actifs. Ceci pourrait être violé par un usage imprudent de champs ou de méthodes statiques [1].

5.2 Topologie et Communication

Nos modèles finitaires comprenaient un processus pour chaque instance d'objet actif dans une application distribuée. Nous procédons ici de manière similaire, mais chaque processus (paramétré) représentant maintenant une classe d'objets actifs, le réseau global de notre application sera donc constitué d'autant de processus paramétrés qu'il y a de classes actives définies dans le code.

La structure interne de ces processus, nous l'avons dit en introduction, est plus complexe que dans le cas finitaire. Pour chaque activité, nous retrouverons un processus paramétré représentant la queue de requêtes, éventuellement factorisé en plusieurs queues indépendantes. Mais nous trouverons en plus un processus paramétré pour représenter le "proxy" de chacune des variables futures utilisée dans le code, et un processus paramétré pour chaque méthode de l'application.

Nous décrivons dans cette section la structure et la construction des réseaux de synchronisation constituant cette hiérarchie.

5.2.1 Structure d'une application

Une application est un ensemble d'objets actifs modélisés chacun par un processus. Chacun est décomposé en un certain nombre de processus internes, représentant les différents aspects de l'objet actif : un processus pour chacune des méthodes de la classe active et des classes auxiliaires qu'elle utilise (regroupés dans un processus "Body"), un processus représentant la queue de requêtes, éventuellement décomposée en un certain nombre de queues indépendantes, un processus représentant le proxy gérant chacune de variables "futures" présentes dans le code de l'activité. A chaque niveau, ces processus communiquent entre eux par échanges de messages. La construction du modèle global de l'application requiert donc le calcul d'une énumération finie de chaque groupe de processus :

[1] cette règle n'est pas contrôlée par la version courante de la bibliothèque

1. Calcul de $\mathcal{O} = \{O_i\}$, ensemble de classes d'objets actifs, défini à partir de l'ensemble des points de création et des paramètres de création de ces objets actifs.
2. Pour chaque objet actif O_i calculer :
   (a) l'ensemble des classes (non actives) utilisées par cet objet actif.
   (b) $\mathcal{M}_i = \{M_k\}$ ensemble des méthodes locales (utiles) de la classe de cet objet et des classes attachées.
   (c) $\mathcal{F}_i = \{F_k\}$ ensemble d'objets futurs, défini par l'ensemble des points de création de ces futurs : points d'appel de méthodes distantes avec un type de retour non vide.
   (d) l'ensemble des méthodes publiques de la classe active, l'ensemble des appels à des méthodes de *service* avec les prédicats de sélection et les modes de services correspondants.

Les liens entre ces processus sont également calculés à partir du code : à partir des points d'appels locaux et distants, les points d'utilisation de futurs, les points de service.

5.2.2 Messages et Notations

La communication entre les processus se fait par échange (envoi et réception) de messages : les requêtes (!∗) et les réponses (?∗). Ces messages incluent ceux définis précédemment à savoir : les appels distants, le service de requêtes à partir de la queue et les messages de disponibilité de futurs, mais de format différent puisqu'ils comportent les paramètres (Figure 5.2). En outre, nous définissons deux autres types : messages d'activation des méthodes locales et messages de retour de résultat d'exécution d'une méthode locale.

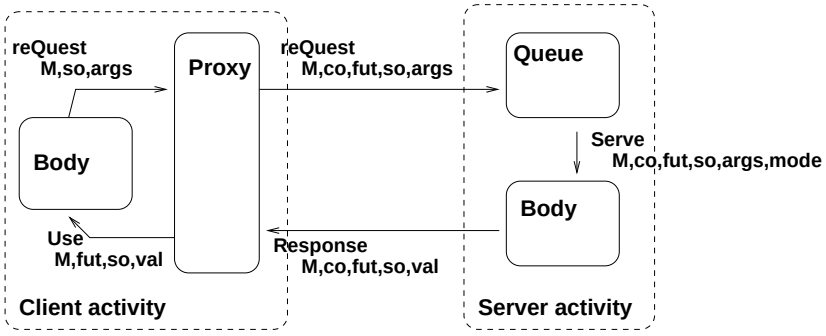


FIG. 5.2 – Communication (paramétrée) entre deux objets actifs

Un message m désignera désormais le nom (qualifié) d'une méthode, l'objet destination (paramétré) et les arguments d'appels (abstraits), son format général est :

$$Message \stackrel{def}{=} [< dir >< adr >]. < type > \_ < meth > (< args >)$$

avec

- $< dir >$ la direction du message envoi (!) ou réception (?).
- $< adr >$ l'identification du correspondant, c'est à dire du destinataire pour une émission, de l'émetteur pour une réception.
- $< type >$ le type de message, noté par : Q pour une requête (appel distant), R pour une réponse, S pour un service, U pour l'utilisation d'un futur, *Call* pour un appel local et *Ret* pour un retour de résultat local.
- $< meth >$ le nom de la méthode invoquée.
- $< args >$ les arguments du message : l'adresse de l'appelant, les paramètres d'appel, les variables et les valeurs.

Les messages échangés entre les processus auront donc la forme : $!x.*$ ou $?x.*$, où x dénote l'adresse respectivement du récepteur ou de l'émetteur. Toutefois, dans le cas d'un message de réception ($?x.*$) x est une variable liée qui ne sera remplacée par l'adresse de l'émetteur que lors de la synchronisation.

Par souci de concision et de clarté des messages, nous avons remplacé la notation des types *Req*, *Rep*, *Serv* et *Fut*, définis dans le chapitre 3 respectivement par les notations Q, R, S et U.

Pour la représentation des méthodes locales et des variables globales, nous complétons cet ensemble de types par trois autres types, notés *Call*, *Check*, *Update* et *Ret*, représentant respectivement l'invocation d'une méthode, l'accès à, la modification d'une valeur d'une variable, et le retour d'un appel de méthode ou d'une valeur de variable.

Enfin, nous avons besoin d'une notation pour les actions que l'utilisateur veut observer (et utiliser dans les propriétés temporelles), qui peuvent être soit des passages à des points de programme spécifiques (noeuds PP du MCG), soit l'observation d'une synchronisation réussie (passage d'un message dans le réseau). Une action observable m, avec les paramètres $args$, sera notée par $Obs\_m(args)$.

5.2.3 Structure des méthodes

Dans nos modèles finis, les paramètres effectifs d'appels de méthodes ne sont pas pris en compte ; un appel récursif d'une méthode est donc considéré comme une boucle, ce qui nous a permis de déplier totalement les comportements, tout en gardant la finitude du modèle. Ceci ne peut être le cas dans la version paramétrée (du moins si le paramètre en question est "observable" dans les propriétés comportementales qui nous intéressent). Par exemple, dans la méthode de calcul de la factorielle, l'appel à *Fact(n)* devrait être bien différencié de l'appel à *Fact(n-1)*. Pour prendre en compte cette distinction au niveau des modèles, nous considérons chaque méthode comme étant un processus indépendant et les appels entre méthodes seront des communications (paramétrées) entre ces processus (figure 5.3).

Lorsqu'un processus de ce type sera instancié, les domaines de ses paramètres (d'appel et de retour) seront abstraits de manière finie, et ces domaines abstraits comporteront au moins une valeur couvrant "*le reste du domaine concret*", et jouant le rôle d'indicateur de "*dépassement du domaine d'instanciation*".

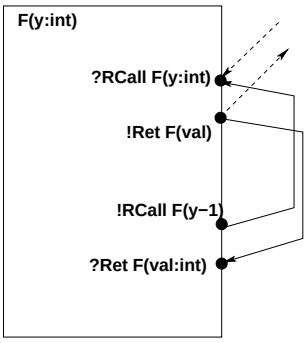


FIG. 5.3 – Méthode Récursive

Représenter les méthodes récursives comme des processus paramétrés, et les appels comme des synchronisations locales entre différentes instantiations est un mécanisme très expressif, même en se limitant à des domaines de valeurs dénombrables. Dans [101], De Simone montrait comment des constructions similaires (avec des paramètres dans des domaines semi-linéaires) dans l'algèbre de processus Meije étaient turing-expressives.

5.2.4 Objets et Stores

Pour représenter proprement le modèle paramétré d'un objet actif, il est important de considérer la manière dont seront reproduits tous les attributs de cet objet lors de l'instanciation.

En Java une instance d'objet possède un certain nombre d'"éléments" : champs et méthodes auxquels elle peut accéder.

La sémantique de *ProActive* suppose que chaque objet actif, donc activité, dispose de son propre domaine (store) où sont rangés ses champs (variables d'instance) propres, le principe même de la distribution. Par conséquent, en représentant un modèle paramétré d'une classe d'un objet, nous devons représenter le store donc ces données qui lors de l'instanciation du modèle, seront également instanciées. Ainsi, nous calculons une énumération de ces éléments par analyse statique. Dans le cas où celle-ci peut déterminer précisément le nombre de ces objets nous l'utilisons ; sinon, nous indexons l'objet par un entier désignant son rang dans la création.

Notre représentation est adéquate dans le cas où les objets actifs sont à des localités différentes (différentes JVMs), par contre ce n'est plus le cas dès que deux objets actifs sont

sur la même JVM. En effet, en Java donc en *ProActive* il existe trois sortes de variables :

- variable d'instance, attribut d'une instance de la classe (1 par instance/objet).
- variable locale déclarée dans une méthode.
- variable de classe attribut de la classe (1 seule pour toute la classe).

Cette représentation du store demeure vraisemblable, pour les variables locales, liées au processus d'une méthode, et les variables d'instance, liées au processus d'un objet, qui seront dupliquées en autant que le nombre d'instances de processus associés. Par contre, ce qui ne peut être le cas des variables de classe dont les valeurs sont communes à toutes les instances. Nous proposons alors deux solutions : soit astreindre le programmeur à ne pas utiliser des variables de classe. Contrainte non sévère, les variables de classe compromettent le principe de distribution et de mobilité ; ou encore, créer un objet actif (un processus) qui comportera toutes ces variables de classe et leur manipulations (lecture ou écriture) seront des communications vers cet objet.
Le problème est semblable pour la représentation des méthodes de classe.

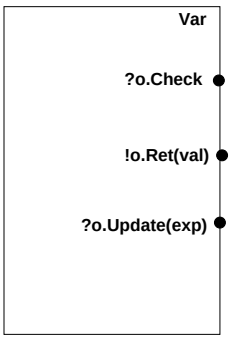


FIG. 5.4 – Processus d'une variable dans le Store

Une variable v dans le store est représentée par un processus (Figure 5.4) ; Celui-ci est activé soit par un événement de lecture de valeur de la variable ($?o.Check$), puis le retour de cette valeur $!o.Ret(val)$) ou par une mise à jour de son contenu ($?o.Update(exp)$).

5.3 Construction des modèles

De même que dans les modèles comportementaux bornés, les modèles paramétrés sont construits de manière compositionnelle et hiérarchique. Nous décrivons dans les sections suivantes la construction :

- du réseau global de l'application,
- du réseau de chaque activité,
- des automates des queues, des proxies de futurs et des variables globales,

- de l'automate de chaque méthode.

Le lecteur trouvera à partir de la figure 5.11 des illustrations de ces constructions sur un exemple complet.

5.3.1 Actions

Commençons par définir le domaine des *actions paramétrées*, qui sont les constituants de base de nos comportements. Cette définition est une instance de la définition générique du chapitre précédent :

Définition 33 Action paramétrée. *Une action paramétrée est une expression de la forme* $[b]\mathcal{A}$ *telle que* b *est une expression booléenne et* $\mathcal{A}$ *est une action : soit* τ *une action non-observable,* $\mathcal{P}$ *un programme séquentiel (avec des affectations de variables), ou encore* $\mathcal{E}$ *un élément de communication :* $?P.m(\overrightarrow{x})$ *réception du message d'appel à la méthode* m *(avec les arguments* $\overrightarrow{x}$*) à partir du processus* P *, ou* $!P.m(\overrightarrow{x})$ *appel à une méthode* m *(avec les arguments* $\overrightarrow{x}$*) d'un processus distant* P*. Les booléens* b *et les arguments* x *sont des expressions construites à partir de variables et de paramètres, ainsi que des constantes et des opérateurs de nos types simples. L'ensemble des actions paramétrées est noté pAct.*

5.3.2 Réseau global de l'application

Le réseau modélisant l'application est un pNet ayant une boite argument pour chaque point de création d'objet actif dans le code de l'application. Dans les cas simples, cela correspond à un argument pour chaque classe active de l'application, mais en toute généralité il nous faut distinguer les instances créées en des points différents, chacune avec un domaine spécifique pour ses paramètres.

Chacun de ces arguments a un ensemble K_i de paramètres correspondant aux paramètres de son point de création ; le domaine de chaque paramètre est un type simple, calculé le plus précisément possible par analyse statique. Notons $B_i(S_i, K_i = \{p_{i_k} : dp_{i_k}\})$ la boite paramétrée du i-ème argument.

Pour déterminer la sorte S_i d'un argument, c'est à dire les ports de sa boite paramétrée, nous avons le choix entre utiliser la sorte associée à la classe active correspondante (l'ensemble de ses méthodes publiques, donc des requêtes acceptables par sa queue), ou d'utiliser les informations collectées par analyse statique pour nous restreindre aux méthodes effectivement utilisées par l'application.

Les ports de chaque boite et les liens associés sont construits parallèlement sur toutes les boites par analyse du code source (ou du pMCG) de toute l'application :

1. L'ensemble des points d'appel de l'objet O_i (appel de la forme $O_j[p_{i_k}].m(args)$) à une méthode distante d'un objet distant $O_j[p_{i_k}]$ définissent l'ensemble des "requêtes émises" par cet objet O_i et "servies" par l'objet O_j. Leur correspondent les ports de

communication "envoi et réception de requêtes" respectivement de B_i et de B_j, notés $!O_j[p_{i_k}].Q\_m(args, fut)$ et $?o.Q\_m(args, fut)$.

Si cette communication doit être observée dans le comportement global, il lui correspond un port $Obs\_m(O_i, O_j[p_{i_k}], args)$ de la boite englobante.

Si de plus la méthode invoquée (m) retourne un résultat nous associons également à ces boites des ports "émission et réception de réponse", $!o.R\_m(val, fut)$, $?x.R\_m(val, fut)$ et $Obs(o, x, val)$.

2. L'ensemble des événements observables du code d'un l'objet O_i (méthodes pertinentes pour la présente preuve), détermine l'ensemble des actions locales (sans communication) ; à chaque événement observable $e(args)$ est associée sur sa boite B_i un port $Obs\_e(args)$.

5.3.3 Réseau de l'activité d'un objet

S'il peut y avoir dans le réseau global d'une application plusieurs arguments correspondant à des points de création différents d'objets de même classe active, il nous suffit par contre de générer un seul modèle par classe C_i d'objet actif. Pour chacune nous construisons :

1. Une boite, notée $B(\mathcal{Q}_i)$, associée à sa queue de requêtes ;
2. Un ensemble de boites, notée chacune $B(F_k)$ et associée à chaque objet futur de $\mathcal{F}_i = \{F_k\}_k$;
3. Un ensemble de boites, notée chacune $B(M_k)$ et associée à chaque méthode de $\mathcal{M}_i = \{M_k\}_k$. L'une de ces boites est celle de la méthode $C_i.runactivity$ qui définit le comportement de l'activité ;
4. une boite, notée $B(\mathcal{S}_i)$, associée au store de la classe.

Les ports et les liens entre ces boites sont construits de la manière suivante :

Appels locaux : pour chaque boite $B(M_k)$ représentant une méthode $m(args)$ sont associés respectivement les ports "activation et retour de méthodes", notés $?k.Call\_m(args)$ et $!k.Ret\_m(args)$.

Pour chaque point d'appel de méthode locale sur un objet passif x on ajoute à la boite de la méthode courante les ports $!x.Call\_m(args)$ et $?x.Ret\_m(args)$.

Requêtes et services : pour chaque méthode publique q de la classe C, la boite englobante et la queue $B(Q_i)$ ont un port "réception de requêtes", $!o.Q\_q(args, [fut])$. Pour chacune, la queue $B(Q_i)$, et les méthodes dont le code contient le noeud de service correspond ont des ports $?/!x.S\_q(args, f, mode)$, et lorsque cette requête rend un résultat, la méthode correspondante et la boite englobante ont un port $!o.R\_q(val, f)$.

Appels distants et futurs : pour chaque point de création d'un futur (appel distant $o.m(args)$), on a un port $!f.Q\_m(args, o)$ sur la boite de la méthode appelante, deux ports $!o.Q\_m(args, f)$, $?y.R\_m(v)$ sur la boite englobante, et deux ports $?x.Q\_m(args, o)$ et $?y.R\_m(v)$ sur la boite du proxy du futur.

Pour chaque point d'utilisation d'un futur f, on a un port $!o.U\_m(val)$ sur la boite du proxy, et un port $?o.U\_m(val)$ sur la boite de la méthode qui utilise.

Variables : pour chaque accès en lecture à une variable v, on ajoute à la boite de la méthode courante les ports $!v.Check$ et $?x.Ret(val)$. Et pour chaque mise à jour, on ajoute un port $!v.Update(exp)$.

Pour chaque paire d'actions opposées $!m$-$?m$, nous construisons un lien orienté entre les ports correspondants et, lorsqu'elle est observable, le lien étiqueté par l'action $Obs\_m(args)$.

5.3.4 Comportement des Queues

La queue de requêtes d'un objet s'exécute indépendamment de son corps, elle accepte toutes les requêtes arrivantes. L'arrivée de ces requêtes diverses et variées est codée par les actions $?o.Q\_m(\ldots)$. Nous avons vu (Section 2.3) qu'il existe plusieurs primitives et différents modes de sélection dans la queue.

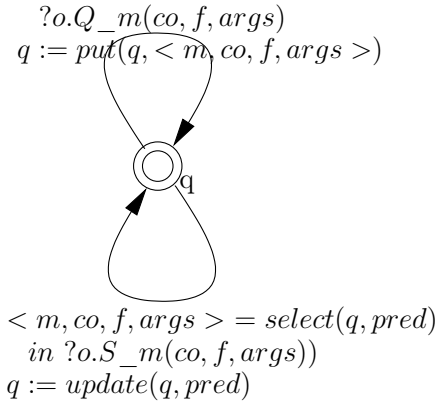


FIG. 5.5 – Automate d'une Queue de Requêtes

Nous supposons que par analyse statique nous pouvons déterminer un élément de la partition finie des requêtes arrivant dans une queue ; et nous pouvons également déterminer tous les modes de sélection utilisés dans le code de l'objet actif correspondant, à partir de l'ensemble :

$$Modes = \{serve, serveAndFlush\} \times \{Oldest, Newest, Nth\}$$

Nous représentons la queue, dans la mesure du possible, par un produit indépendant de processus, de façon que chaque processus désigne une partition et modélise un mode approprié. Le modèle de chaque partition est construit de manière automatique, comme étant une instantiation du modèle générique la Figure 5.5, dans lequel $m, args, pred$ seront alors remplacés par les valeurs possibles correspondantes.

En fait, cette factorisation de la queue en queues séparées à partir de la partition des primitives servies par à un objet donné, est une optimisation considérable de celle-ci ; en effet, nous évitons de calculer leur produit indépendamment du contexte. Le comportement de ces queues est coordonné avec le comportement du body (méthode runActivity) de l'objet correspondant grâce à l'action de service $?o.S\_m(...)$.

5.3.5 Proxies de Futurs

L'automate d'un objet futur est également représenté de manière générique (Figure 5.6), il décrit le comportement de celui-ci : réception à partir du client d'une requête ($?O.Q\_m(\ldots)$) destinée au serveur, puis réception de la réponse du serveur ($?O'.R\_m(\ldots)$) et enfin émission de celle-ci vers le client ($!O.U\_m(\ldots)$). Dans le cas où l'analyse statique garantit qu'un résultat est consommé à un point donné du programme, l'objet futur correspondant peut alors être recyclé. Ainsi un seul automate est utilisé pour représenter toute une famille d'objets futurs, indexés par le nombre de leurs occurrences dans le store.

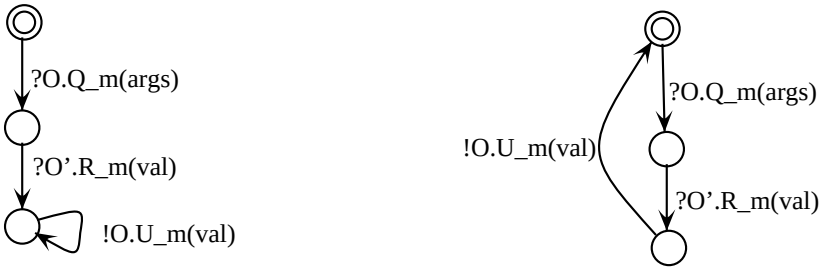


FIG. 5.6 – Automate d'un Futur (avec ou sans Recyclage)

Les requêtes distantes sont interceptées par le proxy de l'émetteur qui créé les objets futurs.

5.3.6 Comportement d'une variable dans le Store

L'automate d'une variable dans le store décrit l'évolution de celui-ci. Pour des variables *entières*, nous avons un store indépendant (non indexé) par variable ou un store indépendant (indexé) pour chaque point d'allocation. Par contre, pour les structures (par

exemple tableaux), le modèle sera plus complexe est en cours d'étude (nous donnons des pistes dans un exemple 6.3).

L'automate d'une variable (Figure 5.7) reçoit soit des requêtes d'inspection de sa valeur ($?O.Check$), renvoie donc celle-ci ($!O.Ret(val)$) ou encore des requêtes de modification de la valeur par une exp ($?O.Update(exp)$) et selon que l'expression contient la valeur de la variable ou non, la fonction $update(v_i, exp)$ définit l'implémentation de l'opération de mise à jour.

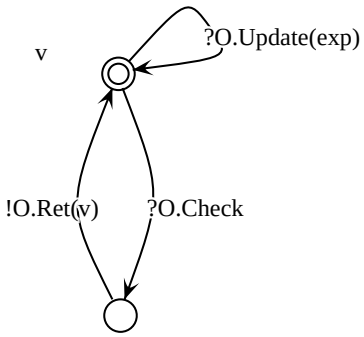


FIG. 5.7 – Automate d'une variable dans le Store

Nous verrons dans la section 5.4 un exemple d'utilisation du store.

5.3.7 Comportement des méthodes

Pour une meilleure lisibilité, nous exprimons la construction du pLTS modélisant une méthode par un algorithme plutôt que par des règles SOS, mal adaptée à des structures complexes comme nos pMCGs.

Notons par :

- s_i un nœud du pLTS,
- n_i un nœud du pMCG,
- Aut le pLTS en cours de construction ; le pLTS vide est noté par $\emptyset$, et le pLTS réduit à un seul nœud est désigné par le nœud lui même,
- Map le mapping liant chaque nœud du pMCG déjà visité à son homologue dans le pLTS,
- $ToDo$ ensemble de couples (nœud du pMCG, nœud du pLTS).

Nous utilisons également des fonctions auxiliaires à savoir :

- la fonction $fresh$ qui créée et retourne un nouveau nœud d'automate,
- et les fonctions de manipulation des structures : $Aut.init$ créée le nœud initial

de l'automate Aut ; et $Aut.add$ créée et ajoute une transition à l'automate Aut ; $Aut.replace$ réalise l'opération de substitution (Définition 21) , et $ToDo.choose$ sélectionne et retourne un élément quelconque depuis l'ensemble $ToDo$.

Algorithme de Construction : La construction se fait par parcours du pMCG en appliquant la procédure principale ci-dessous `Method-Behav` à chaque méthode rencontrée (en partant de la méthode *runActivity*).

$$\texttt{Method-Behav}(m, ent(m), pMCG)$$

où m est une méthode, $ent(m)$ son nœud d'entrée et $pMCG$ est le graphe d'appel complet l'application. Et tels que les structures Map et $ToDo$ sont vides. Dans la suite, tous les successeurs d'un nœud analysé du pMCG sont insérés sont insérés dans $ToDo$. Et chaque nouveau nœud construit du pLTS est inséré dans Map.

Nous notons $calls(n)$ l'ensemble des transitions d'un nœud *call* et $succs(n)$ la fonction qui calcule la méta-transition et l'ensemble des nœuds successeurs d'un nœud quelconque.

```
1  Method-Behav (m, n, pMCG) :
2    Aut.init = fresh s0 ; Map = {n ↦ s0} ;  ToDo = {< n, s0 >}
3    while ToDo ≠ ∅
4      ToDo.choose < n, s >
5      if Map(n) then DO-LOOP-JOIN
6      else
7        select n in
8          Ent(m,args)                              : DO-ENTRY
9          Call(calls(n))                           : DO-CALL
10         PP(lab)                                  : DO-PP
11         Serve(calls(n),mset,pred,mode)           : DO-SERVE
12         Use(fut,val)                             : DO-FUTURE
13         Ret(val)                                 : DO-RETURN
14     unless n=Ret
15        let MT,N = succs(n) in
16          match MT with
17             Access(v);Code          : DO-CHECK
18           | Update(v,exp);Code      : DO-UPDATE
19           | Code                    : DO-NOTHING
20        foreach n_i in N do
21           fresh s_i ; ToDo.Add < n_i, s_i >
22        Aut.add s1 --Code--> S = {s_i}_i
```

Règle principale : La procédure de construction parcourt le graphe de la méthode par un algorithme résiduel : l'ensemble $ToDo$ mémorise les nœuds à traiter plus tard, le mapping Map mémorise ceux déjà traités.

Si le nœud (du pMCG) en cours d'analyse est nouvellement visité une procédure dépendant du type du nœud est alors appliquée (lignes 8-13). Dans le cas contraire, la procédure `Do-Loop-Join` sera appliquée. Pour chaque nœud analysé dont le type est différent de `Ret`, si la méta-transition correspondante une (ou des) variable globale des règles d'accès au store sont alors activées, avant d'associer à l'ensemble de ses successeurs ($\mathcal{N} \geq 1$) un nombre égal de nœuds du pLTS $\mathcal{S}$ sont insérés dans $ToDo$ pour une analyse ultérieure ; et une transition correspondante étiquetée par le code séquentiel intra-procédural (Méta-Transition) est également créée et jointe au pLTS en cours de construction (lignes 20-22).

```
DO-ENTRY(m, args) =
   fresh s_1 ;  Map = Map ∪ {n ↦ s_1}
   Aut.add s  --?o.Call_m(args)-->  s_1
```

Initialisation : Lors de l'analyse du nœud de type *ent* (nœud d'entrée d'une méthode), une transition de réception du message d'activation de la méthode ($?o.Call\_m(args)$) est créée. Le transitions intra-procedurales seront traitées par le code de la partie commune de la procédure Method-Behav (lignes 15-22).

```
DO-PP(lab) =
   if observable(lab) then Aut.add s  --Obs(lab)-->  (fresh s_1), Map = Map ∪ {n ↦ s_1}
   else s_1 = s
```

Nœuds séquentiels : Les nœuds `PP` du pMCG correspondent aux points de programme sans événement de synchronisation mais dont la visibilité est requise, par exemple un label du code source associé à une boucle ou jointure dans un programme, ou encore un point particulier désigné par l'utilisateur. Donc à chaque un nœud `PP` considéré observable nous créons une transition portant le label de ce nœud.

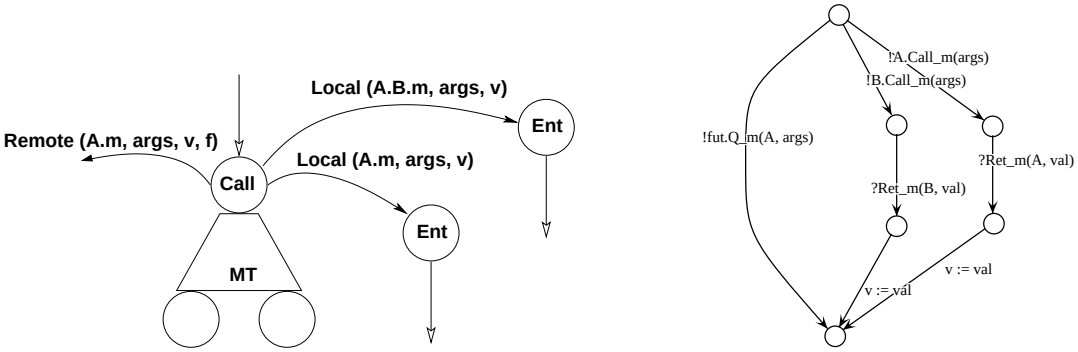
```
DO-CALL(calls(n))  =
  fresh s_1, Map = Map ∪ {n ↦ s_1}
  foreach call in calls(n)
    match call with
      "Remote(o,m,args,var,fut)" :  Aut.add s  --!fut.Q_m(o,args)-->  s_1
      "Static(o,m,args,var)" :      Aut.add s  --!Call_Stat_m(o,args)-->  s_1
      "Local(o,m,args,var)" :       Aut.add s  --!x.Call_m(args)-->  (fresh s_2)
      "Unknown(o,m,args,var,fut)" : Aut.add s  --[Mix]!fut.Q_m(o,args)-->  s_1
                                    Aut.add s  --[Mix]!x.Call_m(args)-->  (fresh s_2)
       if local-or-static-or-unknown ∩ void-result(m) then
         Aut.add s_2  --?o.Ret_m(o,val)-->  s_1
 else  if local-or-static-or-unknown ∩ non-void-result(m) then
         Aut.add s_2  --?o.Ret_m(o,val)-->  (fresh s_3)  --var := val-->  s_1
```

Nœuds d'appels : Un appel de méthode d'un objet sera traité différemment selon que l'analyse de classe est précise ou approximative, que l'objet de la méthode appelée est local ou distant (voire tous les deux), ou encore que la méthode est une méthode utilisateur ou une méthode Java (statique). En somme, d'un nœud de type `call` peut en sortir une ou plusieurs transitions de type $\rightarrow_C$ (Figure 5.8 (a)) ; chacune d'elle porte le type d'appel.

À chaque type d'appel dans le pMCG, nous créons alors un type de transition correspondante dans le pLTS étiquetée par le type d'appel. Pour un appel :

- à une méthode distante une requête est envoyée à l'objet futur, $!fut.Q\_m(\ldots)$;
- à une méthode locale, un message d'activation est envoyé au processus représentant celle-ci, $!x.Call\_m(\ldots)$;
- à une méthode statique l'événement $!Call\_Stat\_m(\ldots))$ est généré ;
- indéfini, nous générons les deux types d'appels, et munissons chacune d'une garde $[Mix]$ qui ne sera résolue qu'au moment de l'instanciation du modèle paramétré.

Après tout appel qui n'est pas distant, le message de fin de méthode doit être reçu avant de poursuivre ; une transition de retour à l'appelant est créée, $\xrightarrow{?o.Ret\_m(...)}$. Si la méthode exécutée (localement) retourne un résultat alors nous mettons à jour la variable destinée à recevoir celui-ci ($var := val$).



(a) Nœud de type call

(b) pLTS correspondant

FIG. 5.8 – Cas d'un appel de méthode indéfinie

À la suite du traitement de toutes les branches d'appel de méthode (Figure 5.8 (b)), il y aura un unique état de "jointure" dans le pLTS (le nœud `fresh` s_1 dans la procédure `Do-Call`).

```
DO-RETURN(val) =
   Aut.add s  --!o.Ret_m(val)-->  (fresh s1)
```

Nœuds de retour : À un point de retour d'une méthode un message et une valeur résultat (éventuellement vide) est retourné à l'appelant dont l'identification a été liée à la variable o à la réception du message d'activation. Notons que ce nœud n'est pas marqué dans le mapping car il caractérise la fin de l'analyse de la méthode.

```
DO-SERVE(calls(n),mset,pred,mode) =
 fresh s1, Map = Map ∪ {n ↦ s1}
 foreach call in calls(n)
   match call with "Local(o,m,args,var)"
   if m ∈ mset do
     fresh s2, s3
     Aut.add s  --[pred(argrs)]!Queue.S_m(f,o,args,mode)-->  s2  --!m.Call_m(this,args)-->  s3
     if null-result(m) then Aut.add s3  --?o.Ret_m(val)-->  s1
     else                   Aut.add s3  --?o.Ret_m(val)-->  (fresh s4)  --!x.R_m(f,this,val)-->  s1
```

Nœuds de Service : Le nœud de type `serve` est semblable au nœud de type `call`, il est doté de deux types d'arcs (arc de transfert et arcs d'appel) (cf. Définition 32). Les arcs d'appel caractérisent les méthodes locales servies en ce point. Pour chaque méthode servie dont le nom est accepté par l'objet (appartenant à $mset$), nous générons l'événement $[pred]!Queue.S\_m(\dots);!m.Call\_m(\dots)$. Celui-ci caractérise l'envoi, vers la queue, du message de service de la méthode m gardée éventuellement par le prédicat $pred(argrs)$ puis l'envoi d'un message d'activation à cette méthode.

Si le type de retour de la méthode est *void*, nous créons uniquement une transition (étiquetée $?o.Ret\_m(val)$) de retour de la méthode vers la méthode "activante" *runActivity*. Dans le cas contraire, nous créons également une transition (étiquetée $?x.R\_m(...)$) émission de la réponse au client.

```
DO-JOIN-LOOP ()  =
    s1 = Map(n)
    Aut.replace(s, s1)
```

Boucles : La règle Do-Loop-Join s'applique à tous les types de nœuds du pMCG déjà visités. Cette règle utilise l'opération de substitution (Définition 21) du nœud courant du pLTS par un nœud récupéré du Map.

```
DO-FUTURE (fut,val) =
  Aut.add s --?o.U_m(fut,val)--> (fresh s1)
  Map = Map ∪ {n ↦ s1}
```

Utilisation de la valeur d'un futur : Un point d'utilisation d'un futur est un point où le résultat d'un appel est requis, nous créons donc en ce point une transition réception du futur $?o.U\_m(\dots)$ qui permet la synchronisation avec l'automate futur (Figure 5.6).

```
DO-CHECK(var) =
  Aut.add s --!var.Check(--> (fresh s2) --?o.Ret(val)--> (fresh s1)
  Code.Replace(var,val)
  Map = Map ∪ {n ↦ s1}
```

Lecture d'une variable : À un point de lecture d'une variable, des transitions étiquetées par la requête d'inspection de la variable puis la réception de la valeur lue, sont jointes à l'automate (qui synchronise avec l'automate 5.4). La variable lue est alors remplacée, dans le code de la méta-transition, par la valeur retournée.

```
DO-UPDATE(var, exp) =
  Aut.add s --!var.Update(exp)--> (fresh s1)
  Map = Map ∪ {n ↦ s1}
```

Modification d'une valeur de variable : Lors de la modification d'une variable, une action de mise à jour de celle-ci, avec la nouvelle valeur, est envoyée à son processus.

Minimisation :

Notons que chaque règle construit au moins deux nœuds du pLTS, et alterne une transition "comportementale" avec une méta-transition "séquentielle". Si le code analysé est suffisamment abstrait beaucoup de méta-transitions peuvent être réduites à des transitions τ, et nous pouvons minimiser le pLTS obtenu par équivalence observationnelle.

5.4 Exemple

Afin d'illustrer l'aspect des spécifications paramétrées d'une application *ProActive* et la procédure de construction de ces derniers. Nous revenons sur l'exemple du dîner des philosophes du chapitre 3 pour en générer cette fois des modèles paramétrés. Pour ce faire, il a été nécessaire de développer et d'adapter d'une part, les outils d'analyse statique afin de construire les pMCGs et d'autre part, les outils graphiques afin de dessiner les modèles (pLTS et pNet) et d'en générer un format *fc2* également paramétré.

5.4.1 Le réseau de l'application

Le réseau d'une application de n philosophes et n fourchettes est représenté (Figure 5.9) par uniquement deux boites $Fork[f]$ et $Philo[p]$. Ces boites sont paramétrées respectivement par les indices p et f appartenant à $[1 \dots n]$, et caractérisant le nombre d'instances de chacun des processus, et par la boite englobante, nommée $Table$.

Un philosophe p peut demander l'acquisition des fourchettes droite et gauche, indexées p et $p+1$, par les requêtes $!Fork[p].Q\_take$ et $!Fork[p+1].Q\_take$; reçoit donc la réponse $?O.R\_take$. De la même manière, après avoir mangé, ce philosophe libère les fourchettes, par les requêtes : $!Fork[p].Q\_drop$ et $!Fork[p+1].Q\_drop$.

La fourchette, de son coté, reçoit par le même port $?O.Q\_take$, les requêtes d'acquisition de tous les philosophes. Par la synchronisation des événements, l'objet émetteur est substitué à l'adresse de l'objet de réception ; par exemple, la synchronisation de $?O.Q\_take$ et ($!Fork[p].take$ la variable liée O est remplacée par $Fork[p]$.

Les événements observables de l'extérieur, dans ce modèle, sont : l'action de manger ($Obs\_eat(p)$) et l'action de réfléchir ($Obs\_think(p)$).

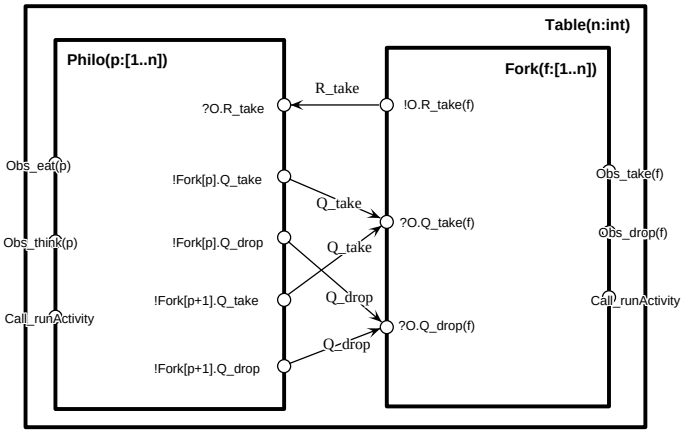


FIG. 5.9 – pNet de la table du dîner

5.4.2 L'objet Philosophe

À partir du même code source que celui donné dans le chapitre 3 (Listing 3.1) nous obtenons pour la classe `Philosopher` le pMCG de la Figure 5.10. Remarquons que la structure générale d'un pMCG est semblable à celle du dMCG (Chapitre 3) ; en effet, le philosophe boucle indéfiniment entre les appels aux méthodes : `think`, `getForks`, `eat` et `putForks`.

Par contre, le pMCG inclut le flot de données : la méthode `runActivity` inclut la variable `hasForks`, l'ajout de la méta-transition, une garde "`if(hasForks) goto 1 else goto 2`" et des expressions d'affectations "`hasForks=0`" et "`hasForks=1`" ;

Remarquons également la différence dans la représentation des appels de méthodes ; les méthodes distantes sont différenciées des méthodes locales grâce aux labels des arcs sortants des nœuds de type *call* qui désignent le type d'appel (local ou distant) : des arcs étiquetés *Local*(...) vers les points d'entrée des méthodes `think()`, `getForks()`, `eat()` et `putForks()` ; et des arcs étiquetés *Remote*(...) qui ne pointent nulle part ; en effet, les nœuds au bout de ces transitions ne sont pas des points de programmes, c'est juste des nœuds pour caractériser l'extrémité d'une transition d'appel distant.

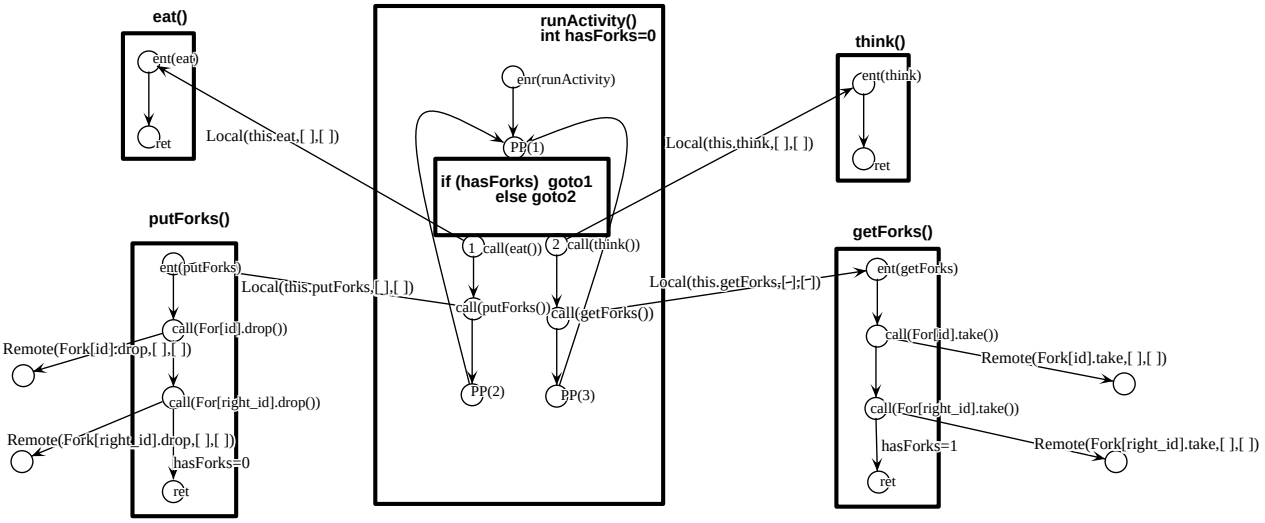


FIG. 5.10 – pMCG de la classe Philosopher

Le réseau global d'un objet de la classe `Philosopher`, dérivant de ce pMCG après, application de la procédure de construction, est présenté dans la figure 5.11. Il est constitué de huit boites. Par souci de clarté des dessins et de simplification de leur lecture, nous associons à chaque boite directement le nom du processus qu'elle représente ; les processus qui ne portent ni paramètres, ni variables, porteront juste le nom.

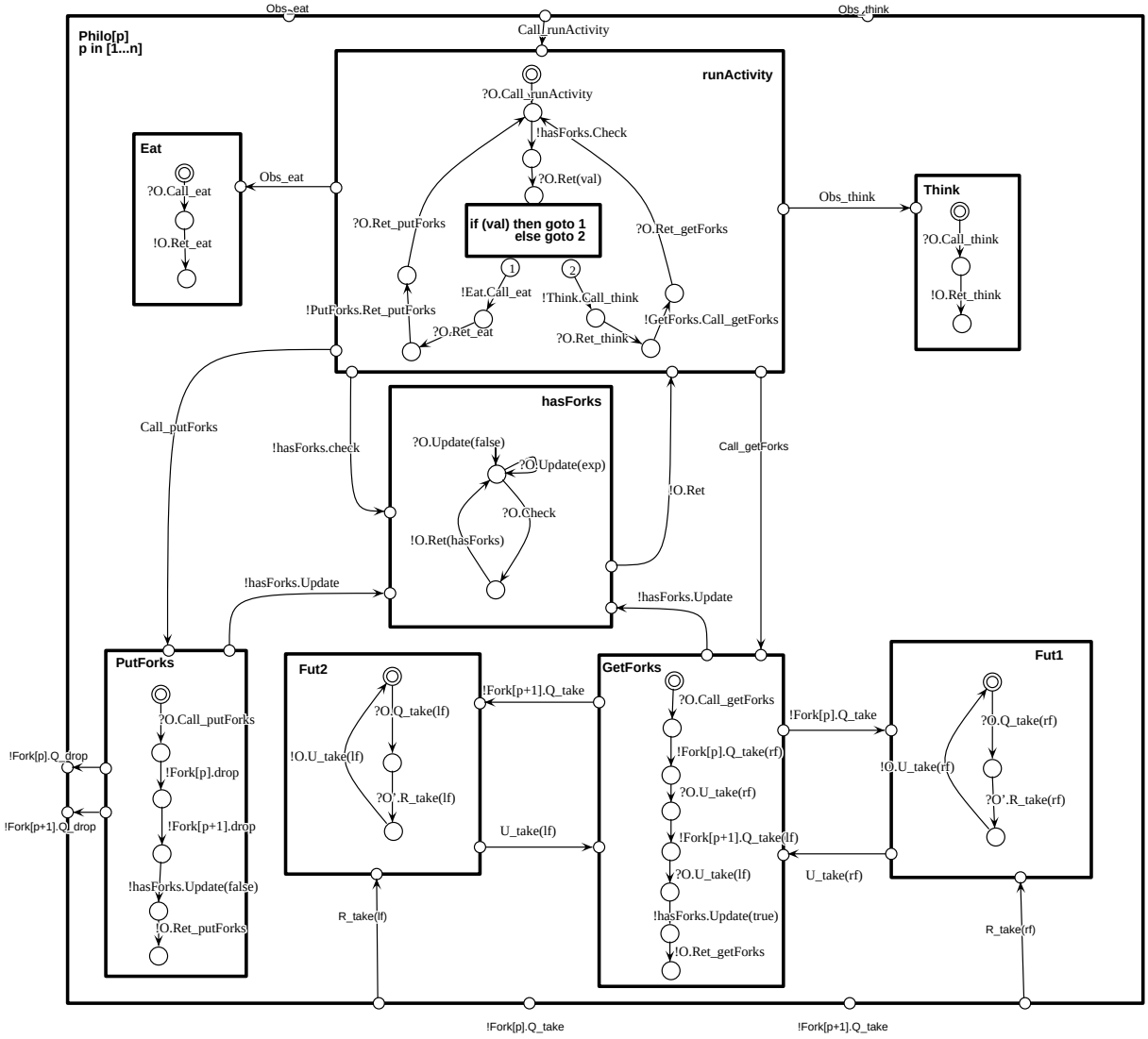


FIG. 5.11 – Le modèle global d'un objet Philosopher

Nous avons ainsi :

- une boite englobante, nommée `Philo[p]`, paramétrée par l'instance (`p`) de cet objet. Cette boite porte les ports des événements observables de l'extérieur : `Obs_eat`, `Obs_think`, et les ports des messages échangés avec des objets distants ; par exemple, les messages d'acquisition et de libération envoyés au processus `Fork[p]` (`!Fork[p].Q_drop` et `!Fork[p].Q_take`).
- une boite associée à la méthode principale, `runActivity()` ;
- quatre boites associées aux quatre méthodes locales ;
- une boite associée à la variable `hasForks` ;
- et deux boites des deux proxies (correspondant aux deux appels distants).

L'automate de la méthode principale reçoit, d'abord, de l'extérieur un message d'activation ; puis le comportement de celle-ci est représenté par deux boucles entre : la recherche de la valeur de la variable `hasForks` et l'appel selon la valeur retournée, soit de des méthodes locales `eat()` puis `putForks()`, ou des méthodes `think()` puis `getForks()`.

Après initialisation de la variable `hasForks` à `false`, son comportement est une boucle entre la réception des messages d'accès en lecture de cette variable ou d'accès écriture.

Les automates des méthodes `eat()` et `think()` ne représentent que la réception des messages d'activation de celles-ci et de retour vers l'appelant (le corps de ces méthodes ne comporte aucun code).

Les automates des méthodes `getForks()` et `putForks()` quant à eux représentent, en plus des messages d'activation et de retour, des requêtes d'appels distants : $!Fork[p].Q\_*$ et $!Fork[p+1].Q\_*$ et des messages de modification de la valeur de la variable `hasForks`.

L'automate des futur correspondant à un appel de méthode distante `getForks` (des fourchettes gauche ou droite), reçoit la requête $?O.Q\_take(rf)$ destinée l'objet $Fork[p]$ ou $Fork[p+1]$; puis reçoit, de l'objet appelé, en l'occurrence l'objet Fork[p], le résultat, $?O'.R\_take(rf)$; et enfin, retourne ce résultat à l'appelant (au processus `GetForks`), $!O.U\_take(rf)$.

5.4.3 L'objet Fourchette

Le pMCG donné dans la Figure 5.12 de l'objet fourchette (classe Fork) dérive également du code donné dans le chapitre 3 (Listing 3.2). Dans ce graphe nous trouvons également des paramètres : une garde "`if (freeFork) goto1 else goto 2`" et les affectations "`freeFork=false`" et "`freeFork=true`".

Le comportement de la fourchette est une boucle infinie ; selon la valeur de la variable de `freeForks`, définie dans la classe Fork, la requête `take` ou `drop` sera servie.

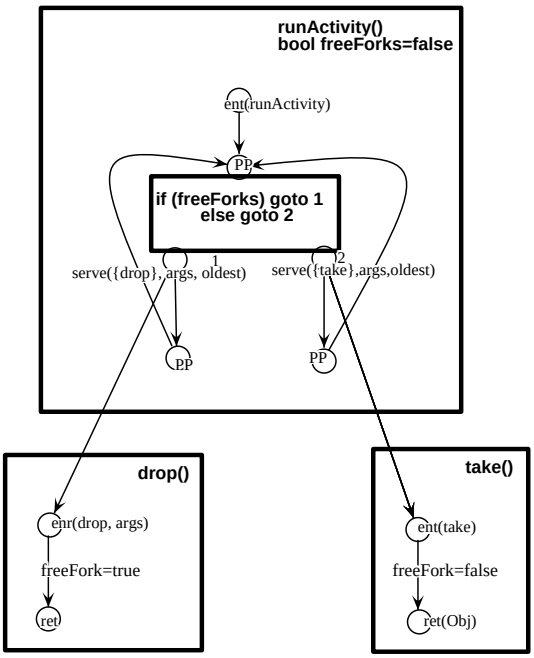


FIG. 5.12 – Le pMCG de la classe Fork

Le modèle paramétré généré à partir de ce pMCG est donné dans la figure 5.13 ; celui-ci représente le réseau global du comportement de l'objet Fork, et il également formé par un ensemble de boites :

- la boite englobante `Fork[f]` ;
- la boite de méthode principale, `runActivity()` ;
- les deux boites correspondant aux deux méthodes locales `drop()` et `take()` ;
- la boite correspondant à la variable `freeForks`,
- et enfin, la boite associée à la `Queue`.

De même que pour le philosophe dans chaque boite nous trouvons le modèle comportemental du processus qu'elle représente : le réseau dans le cas de la boite englobante et des automates dans les autres cas.
Bien entendu, comme dans la version finitaire il n'y a pas de processus futurs, en effet, le processus `Fork` n'émet aucun appel distant. La queue de requêtes reçoit toujours à partir de l'extérieur (des objets philosophes) les requêtes : `?O.Q_take` et `?O.Q_drop` et à partir de la méthode principale, des messages de service avec leur mode d'exécution ; par contre, elle est codée différemment.

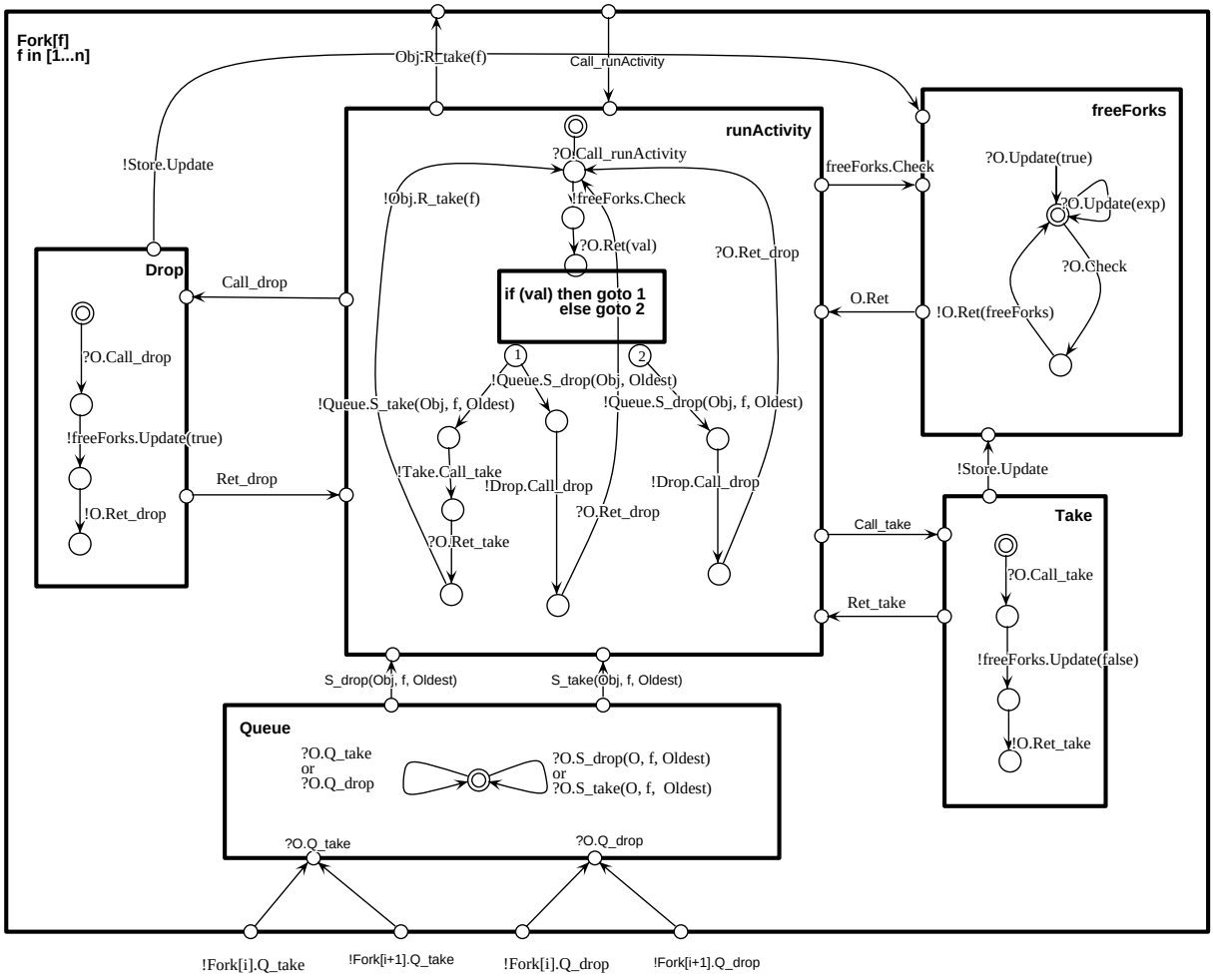


FIG. 5.13 – Le modèle global de la fourchette

5.4.4 Vérification de propriétés

L'intérêt de la construction de tels modèles est leur utilisation pour la vérification de propriétés comportementales sur les programmes modélisés. Les propriétés sont : la présence ou l'absence de deadlocks, les propriétés d'atteignabilité, de sûreté et/ou de vivacité.

La vérification de propriétés paramétrées sur des graphes symboliques (contenant des variables non instanciées, pouvant donc dénoter des espaces d'états infinis) est en général indécidable. Cependant, dans certains outils tels que CADP on peut exprimer et vérifier des propriétés avec quantificateurs, mais ces propriétés seront vérifiées sur des graphes ou tous les paramètres seront instanciés.

Par exemple la propriété donnée par l'automate 5.14 et exprimant qu'un "philosophe

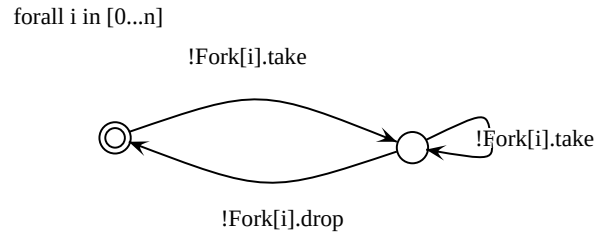


FIG. 5.14 – Un exemple de propriété paramétrée

ne peut demander la même fourchette que s'il ne l'a pas relâchée", peut être interprétée sur un graphe avec 5 philosophes, en supposant que les étiquettes des transitions ont la forme "take-Fork(0)", ..., "take-Fork(4)" et "drop-Fork(0)", ..., "drop-Fork(4) ") s'exprimerait en XTL [87] de la manière suivante :

```
forall philo :integer among { 0 ... 4 } in
  Box (take-Fork(philo),
    AG_A (not (drop-Fork(philo)),
      Box (take-Fork(philo), false)
    )
  )
end_forall
```

5.5 Bilan

La représentation d'applications réparties par des modèles symboliques a l'avantage, d'une part, de prendre en compte plus d'informations du système étudié et de représenter des systèmes (possiblement infinis) par un nombre réduit de processus. D'autre part, elle offre une modélisation plus compacte pour la preuve directe de propriétés d'atteignabilité ; Et plus générique, pour extraire différentes instantiations finies pour la vérification de différentes propriétés comportementales, sans avoir à reconstruire à chaque fois le modèle global. En outre, les propriétés à vérifier sur les instances portent désormais plus d'informations et de données, donc sont plus riches.

Toutefois, cette représentation, à l'instar des modèles finis, repose sur une abstraction des domaines de données, des valeurs des messages échangés et des processus adressés.

Notons que dans ce chapitre, nous n'avons pas eu à faire la preuve de terminaison de la procédure de construction ; en effet, nous associons un processus/méthode, le nombre de méthodes d'une classe étant fini donc le nombre de processus l'est également.

Chapitre 6

Exemples

Dans ce chapitre nous présentons trois exemples d'applications réelles, développées en Java. Ces exemples ont été choisis de façon à illustrer et à couvrir différents aspects de modélisation des applications. Pour chaque exemple, nous décrirons les caractéristiques de son architecture et celle de son code Java. Ensuite, nous donnerons les réseaux des processus constituant celle-ci ainsi que les modèles comportementaux correspondants dérivés par application de l'algorithme de construction de modèles paramétrés. Nous commenterons également les points cruciaux et les difficultés de chaque modèle. Enfin, nous donnerons une description simplifiée de l'implémentation.

6.1 Factorielle

Le premier exemple est l'application de calcul de la factorielle, le choix de cet exemple n'a pas pour but la représentation du caractère réparti d'une application – qui ne l'est pas – mais c'est simplement d'illustrer la manière de représenter les méthodes récursives par des modèles paramétrés.

Le calcul de la factorielle est effectué par la fonction définie par :

$$\begin{aligned} fact(0) &= 1 \\ fact(n) &= n * fact(n-1) \quad si\ n > 1 \end{aligned}$$

Cette fonction est implémentée dans le code Java du Listing 6.1, par la méthode récursive, *fact* qui multiplie successivement la donnée en entrée par le résultat de l'appel du calcul de la factorielle de l'antécédent de cette donnée jusqu'à atteindre l'entier 0 et retourne le résultat.

Listing 6.1 – La méthode Factorielle

```
int fact(int n){
  if(n <= 1)
      return 1;
  else
     return n * fact (n-1);
}}
```

La Figure 6.1 présente le modèle global de cette l'application simplement par un processus paramétré par les variables n et v qui sont respectivement le nombre dont on calcule la factorielle et le résultat retourné après calcul.

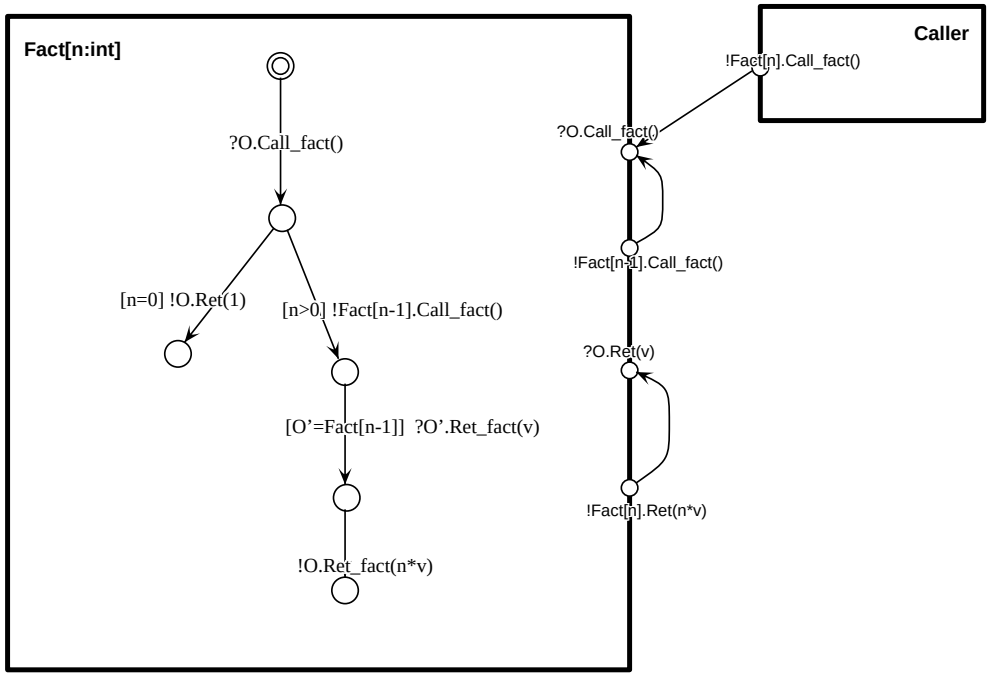


FIG. 6.1 – Modèle global de la méthode *fact*

Remarquons que les liens des ports de la boite `Fact(n,v)` illustrent bien l'aspect récursif de la fonction de calcul ; par exemple le lien partant du port $!Call\_Fact(n)$ vers le port $?Call\_Fact(n)$, désigne qu'un processus `Fact` envoie un message d'activation de méthode à un processus identique, `Fact`, c'est l'appel récursif.

Pour pouvoir faire de la vérification de propriétés, ce modèle doit être instancié ; La procédure d'instantiation déplie le modèle paramétré en un nombre fini, égal à n (nombre d'appel de la méthode *fact*) d'instances, reliées en cascade. Chacune de ces instances, calcule une partie du résultat : elle reçoit un nombre n, si celui-ci est 0 renvoie 1 ; sinon, envoie un message d'activation à l'instance suivante, après réception du résultat de celle-ci

envoie au serveur le résultat $v * n$. Bien entendu, cela suppose, la première instance de la chaîne est activée par message d'activation initial, extérieur qui est le premier appel à la méthode *fact* depuis la méthode principale `Caller`.

Bien entendu, cet exemple simple et non distribué (sans queue de requêtes, ni futur) n'offre pas un modèle élaboré pour prouver des propriétés comportementales intéressantes, à part, par exemple vérifier que la factorielle d'un nombre n est calculée après calcul de la factorielle du précédent.

Il illustre aussi le type de propriétés paramétrées que notre méthode d'abstraction finie permet d'aborder : toute propriété concernant des valeurs finies sera traitée par une instanciation adéquate. Nous ne pouvons par contre, prouver aucune propriété inductive.

6.2 Nombres de Fibonacci

Notre deuxième exemple est celui d'une application dont la topologie est complètement statique, dans le sens où les processus ne sont pas paramétrés mais dont les messages le sont et dont chaque message met à jour une variable du processus.

Cette application est le calcul des nombres de Fibonacci décrit dans le calcul ASP [50] (Section 5.4) dans un style proche des "séquential process" et défini par la suite :

$$\begin{array}{lcl} fib(0) & = & 0 \\ fib(1) & = & 1 \\ fib(n) & = & fib(n-1) + fib(n-2) \quad avec\ n > 1 \end{array}$$

Pour décrire le programme qui calcule et affiche la valeur de $fib(n)$ selon la définition récursive précédente, cette application peut être simulée par des processus communiquants entre eux (Figure 6.2). Les processus *Cons1* (pour mémoire) et *Cons2* (pour fib(n)) représentent respectivement le nombre à l'ordre $n-1$ et le nombre à l'ordre n, le processus *Add* l'additionne deux nombres reçus en entrée et en délivre le résultat et enfin, *Display* est l'afficheur de nombres.

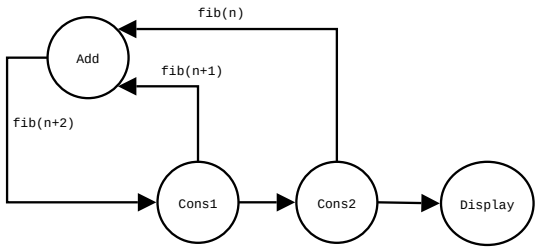


FIG. 6.2 – Calcul des nombres de Fibonacci

En partant d'une configuration où $cons1 = 0$ et $cons2 = 1$ le système évolue par

transfert de nombre entre processus. Cette évolution est décrite par trois objets actifs. La fonction de transfert de la première suite mémorise la valeur de la seconde suite à l'instant d'avant n1 -> n2. La fonction de transfert de la seconde suite reprend la récurrence ci-dessus n2 -> n1+n2. On part avec n1=0 et n2=1.

Cette application est implémentée en *ProActive*, de façon que chaque processus est simulé par un objet actif (Listing 6.2).

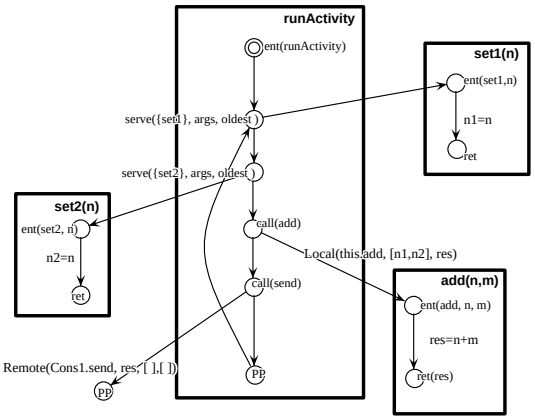
Listing 6.2 – Le code de la méthode principale

```
public static void main(String[] args) {

        ...

        Add add = (Add) ProActive.newActive(Add, null);
        Cons1 cons1 = (Cons1) ProActive.newActive(Cons1, null);
        Cons2 cons2 = (Cons2) ProActive.newActive(Cons2, null);
        ...
}
```

Le code de la classe de chaque objet actif et du pMCG associé sont décrits ci-dessous.

Listing 6.3 – La classe de l'objet Add et le pMCG correspondant

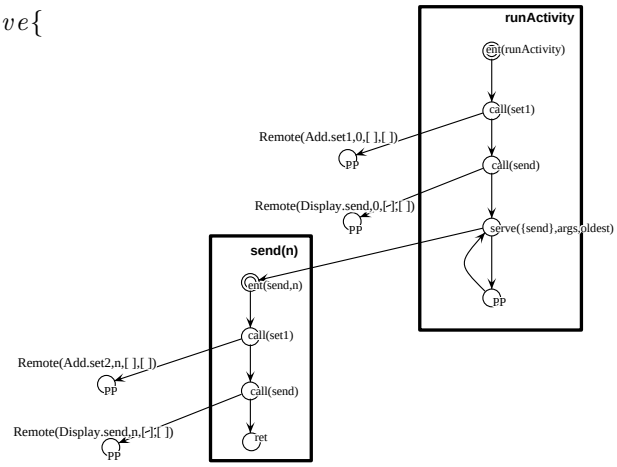
```
public class Add implements RunActive{

public void runActivity(Body body) {
  int n1, n2=0;

    while (true) {
          service.serveOldest("set1");
          service.serveOldest("set2");
          Cons1.send(add(n1,n2));
 }}

void set1 (int n) {n1=n; }

void set2 (int n) {n2=n; }

int add (int n, int m) {return (n+m);}
}
```



L'objet `Add` reçoit des requêtes `set1` et `set2` respectivement des objets `Cons1` et `Cons2` ; son comportement consiste à servir en boucle la première puis la seconde requête et ensuite à envoyer le résultat de l'addition des deux variables locales vers l'objet mémoire `Cons1`.

Listing 6.4 – La classe de l'objet Cons1 et le pMCG correspondant

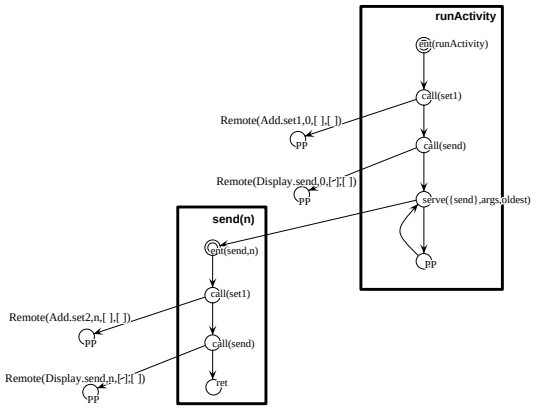
```
public class Cons1 implements RunActive{

public void runActivity(Body body) {
Add.set1(1);
Cons2.send(1);
while (true) {
  service.serveOldest("send");
}


void send (int n) {
Add.set1(n);
Cons2.send(n);
   }
}
```



L'objet actif `Cons1` quant à lui, envoie au commencement le nombre 1 vers l'objet `Add` (pour l'addition) puis vers `Cons2` (pour l'affichage) ; son comportement se poursuit par une boucle servant un seul type de requêtes provenant de `Cons1` et correspondant à l'appel de la méthode `send` ; le comportement de cette méthode est également l'envoi d'un nombre, cette fois n, vers l'objet `Add` puis vers l'objet `Cons2`.

Listing 6.5 – La classe de l'objet Cons2 et le pMCG correspondant

```
public class Cons2 implements RunActive{

public void runActivity(Body body) {
Add.set2(0);
Display.send(0); //Affichage de 0
while (true) {
  service.serveOldest("send");
 }


void send (int n) {
Add.set2(n);
Display.send(n);
 }
}
```



L'objet `Cons2` a un comportement semblable à celui de `Cons1` ; il envoie d'abord le nombre 0 vers `Add`, également pour l'addition, puis ce même nombre vers l'objet `Display` pour affichage ; ensuite boucle dans le service de la méthode `send` émanant de l'objet `Cons1` et dont le code est analogue à celui de la même méthode pour l'objet `Cons1`.

Le modèle global de cette application est donné dans la figure 6.3. Nous avons donc

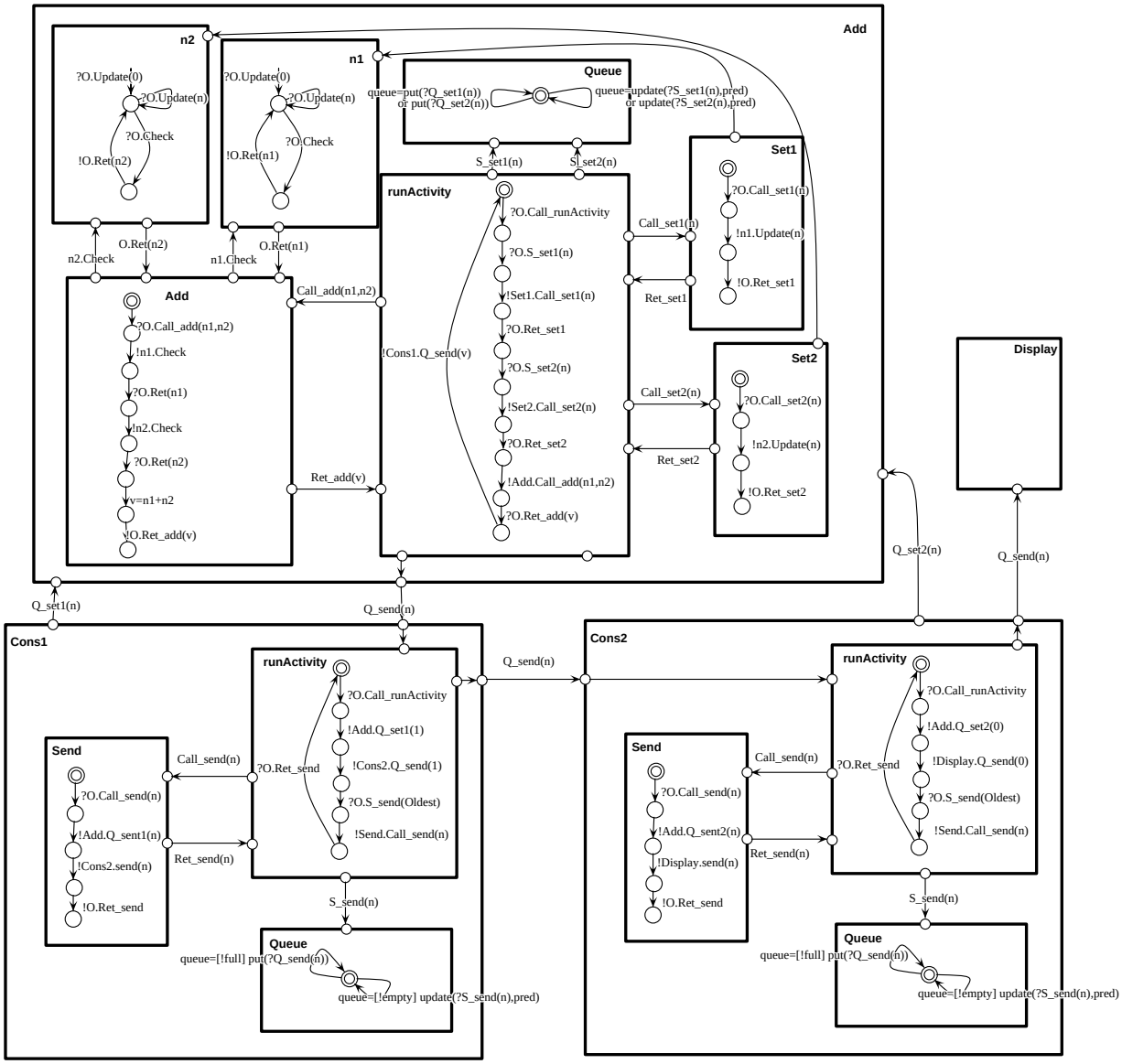


FIG. 6.3 – Modèle global de l'application Fibonacci

quatre boites associés aux objets actifs : `Add`, `Cons1`, `Cons2`, `Display` et nommées par les noms de ces derniers. Nous trouvons dans chacune des boites le modèle comportemental (réseau global) de chaque objet ;

Remarquons que les boites (`Add`, `Cons1` et `Cons2`) ne sont pas paramétrées, en effet, les objets actifs correspondants sont non paramétrés ; ils sont au nombre de trois à communiquer entre eux par envoi de messages. Par contre, les messages quant à eux sont paramétrés, chaque message porte une variable que l'objet récepteur mettra à jour ou affichera. La boite `Display` est vide car nous n'étudions pas ici le comportement de l'afficheur. Les réseaux des autres boites comportent toutes les boites : de la méthode *runActivity*, des méthodes locales et de la queue de requêtes ; la boite de `Add` comporte, en outre, le `Store`. Remarquons également que toutes les valeurs de retour des appels de méthodes distantes sont de type *void*, donc aucune valeur future n'est attendue après ces appels. Donc pas d'automates de proxies générés, ce qui réduit conséquemment le modèle global (moins d'entrelacement) .

Si nous instancions ce modèle avec les différentes valeurs prises par les variables portées par les messages et observons les messages arrivant au processus `Display`, nous obtenons alors la série de fibonacci.
En outre, nous pouvons prouver sur ce modèle des propriétés, par exemple, la propriété de bornitude des queues : "la queue de l'objet Cons1 peut être réduite à une seule cellule" ou "la queue de l'objet Cons2 reçoit au plus deux requêtes". Ce qui nous permet de représenter ces queues par des modèles finis.

6.3 Arbres binaires

Notre troisième exemple, disponible sur le site de *ProActive* [17], illustre la création d'objets (passifs et actifs), que nous représentons par un réseau paramétré reflétant directement la structure des instances et des liens figurant dans le code.

Un arbre binaire est une liste composée de nœuds ayant une valeur, un fils gauche (left) et un fils droit (right). Le pointeur racine (root) adresse le premier élément de l'arbre. Chacun des pointeurs gauche et droit adressent un "sous-arbre" plus petit. Un pointeur de valeur *null* est un arbre binaire sans éléments – un arbre vide.

Une telle structure peut être utilisée pour ordonner des éléments (ou ici des paires clef et valeur), en considérant que le sous-arbre gauche d'un nœud contient les éléments dont la clef est inférieure à celle de l'élément correspondant au nœud, tandis que le sous-arbre droit contient des éléments dont la clef est supérieure (figure 6.4).
La construction de l'arbre se fait à partir d'une suite de nombres fournis par l'utilisateur ; à chaque fois qu'un nombre est entré, en partant de la racine, la valeur de ce nombre est comparée à celle d'un nœud de l'arbre si elle est égale elle est remplacée, si elle est inférieur la comparaison se poursuit récursivement au sous-arbre gauche ; sinon le droit ; jusqu'à trouver l'endroit où sera inséré l'élément ou atteindre une feuille, auquel cas de

nouveaux fils (gauche et droit) sont créés.

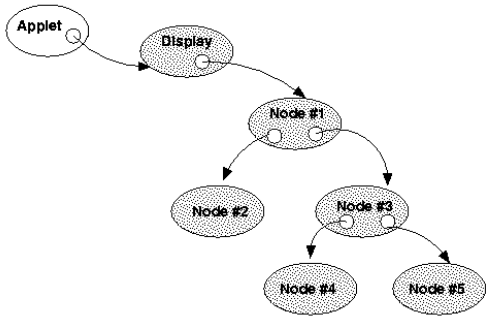


FIG. 6.4 – Arbre binaire

Le code Java décrivant cette construction est donné la classe `BinaryTree` (Listing 6.6). Cette classe comporte une méthode `put` qui recherche récursivement, à l'aide d'une clef (*key*), l'endroit où il sera inséré l'objet *Value* ; une méthode `get` qui elle recherche un objet désigné par sa clef et le retourne si celui-ci est trouvé, retourne *null* dans le cas contraire.

Listing 6.6 – La classe BinaryTree

```
public class BinaryTree {
  protected int key;
  protected Object value;
  protected boolean isLeaf;
  protected BinaryTree leftTree;
  protected BinaryTree rightTree;

  public BinaryTree() {
    this.isLeaf = true;  }

public void put(int key, Object value) {

      if (this.isLeaf) {
      this.key = key;
      this.value = value;
      this.isLeaf = false;
      this.createChildren();
    } else if (key == this.key) {
            this.value = value;
    } else if (key < this.key)
            this.leftTree.put(key, value);
        else
            this.rightTree.put(key, value);
    }
```

```
public ObjectWrapper get(int key) {
    if (this.isLeaf)
        return new ObjectWrapper ("null");
    if (key == this.key)
        return new ObjectWrapper(this.value);
    if (key < this.key) {
        ObjectWrapper res = this.leftTree.get(key);
        return res;  }
    ObjectWrapper res = this.rightTree.get(key);
    return res;
  }
```

Pour la construction d'un arbre dont les nœuds sont des objets passifs, nous ajoutons à cette classe la méthode `createChildren` (Listing 6.7) qui crée le fils gauche et le fils droit.

Listing 6.7 – Création de nœuds passifs

```
protected void createChildren() {
    this.leftTree = new BinaryTree();
    this.rightTree = new BinaryTree();
  }
```

Le processus de construction de l'arbre est lancé à partir de la méthode principale `main` (Listing 6.8) par l'objet actif *myTree*. Cette méthode insère successivement les nombres 1, 2, 3, 4 ; puis requièrt les objets ayant comme clé les nombres : 3, 4, 2, 1.

Listing 6.8 – Méthode principale

```
public static void main(String[] args) {
                ...
BinaryTree myTree = null;
                ...
myTree = (BinaryTree)newActive(ActiveBinaryTree.class.getName(), null);
            ...
        myTree.put(1, "one");
        myTree.put(2, "two");
        myTree.put(3, "three");
        myTree.put(4, "four");

        ObjectWrapper tmp1 = myTree.get(3);
        ObjectWrapper tmp2 = myTree.get(4);
        ObjectWrapper tmp3 = myTree.get(2);
        ObjectWrapper tmp4 = myTree.get(1);
        ...

}
```

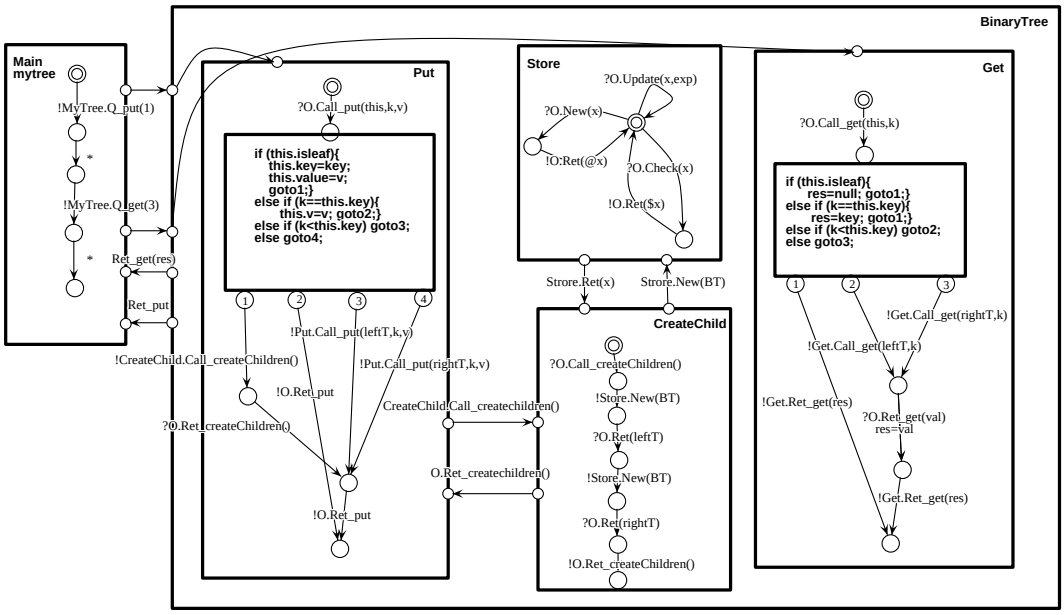


FIG. 6.5 – Le modèle de l'application Arbre binaire (avec objets passifs)

Dans la figure 6.5 nous trouvons le modèle global construit pour cette application. Une boite est associée à la méthode `main` et une autre à `BinaryTree`. La première boite englobe le proxy (que nous ne représentons pas) et le corps de la méthode (pour simplification nous représentons par $*$ les séquences de requêtes et de retour du résultat). La seconde boite englobe les boites `Put`, `Get` et `CreateChild`, des trois méthodes locales ainsi que la boite du store, `Store`.

Remarquons que les modèles des méthodes locales ne diffèrent pas de ce que nous avons vu précédemment une boite/méthode ainsi que l'automate (activation de la méthodes, appels à d'autres et retour).

Par contre, le modèle de l'automate décrivant le comportement du store diffère de celui donné dans la figure 5.7. Nous l'avons étendu, d'une part, de façon à regrouper les variables et les structures dans un processus ; d'autre part, de façon à prendre en compte la création dynamique des objets ; en effet, notons que les actions et $Check(x)$ et $Update(x, exp)$ prennent en paramètre le nom de la variable, et l'action de retour, $Ret(\$x)$, renvoie la valeur de celle-ci. Notons également, l'ajout des actions de création d'objet dans le store ($New(x)$), message correspondant à l'invocation du constructeur *new* ; ceci induit la création de cet objet dans le store puis le renvoie, à l'appelant, de l'adresse de ce nouveau élément ($Ret(@x)$).

Notons que n'avons pas représenté dans les automates des méthodes les messages d'accès aux valeurs des variables (*key*, *isLeaf*, *v*) dans le store.

Une alternative intéressante, par exemple pour un calcul réparti sur une grille, est d'allouer les nœuds de l'arbre dans des objets actifs, selon une stratégie à définir (par exemple jusqu'à une certaine profondeur de l'arbre).

Élargissons donc notre représentation de façon à considérer les nœuds de l'arbre comme étant des objets mixtes : des objets passifs et des objets actifs. Nous ajoutons donc à la classe la méthode de création des fils "actif".

```
  protected void createChildren() {
      ...

      this.leftTree = (BinaryTree) newActive(BinaryTree, null);
      this.rightTree = (BinaryTree) newActive(BinaryTree, null);
      ...
}
```

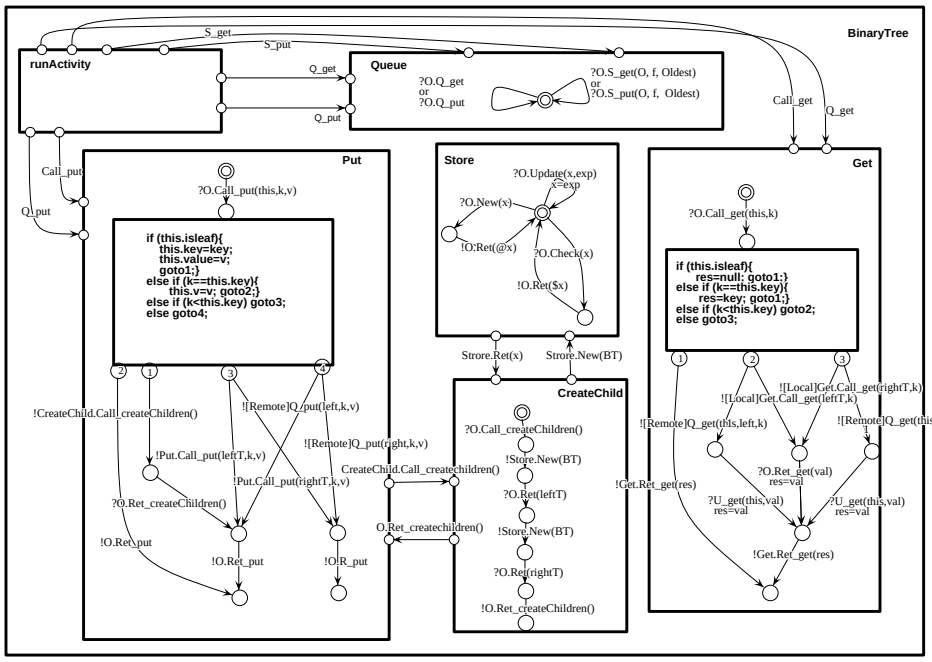


FIG. 6.6 – Arbre binaire avec des objets mixtes (actifs et passifs)

La représentation de l'application ainsi étendue serait alors un modèle "mixte" (Figure 6.6) dans lequel nous aurons un indéterminisme sur le type de requête distante ou activation de méthode (appel distant ou local). Cet indéterminisme du modèle général sera résolu à l'instanciation en choisissant une stratégie de création des nœuds de l'arbre (objets passifs ou objets actifs).

En fait, le comportement des objets actifs est semblable à celui des objets passifs

sauf que les premiers intègrent une queue de requêtes ; et c'est dans les messages de communication qu'est représentée la distinction d'adressage à un objet actif ou passif.

Notons que cet exemple est en cours d'étude, nous le donnons ici pour illustrer le pouvoir de représentation de nos modèles et repousser les limites de notre analyse.

6.4 Réalisation

L'ensemble des règles de calcul de modèles a été implémenté et testé par un prototype appelé *PAX* (de Parameterized Automata eXtractor), destiné à construire, à partir d'un graphe d'appel d'une application *ProActive*, des pLTSs. Cet extracteur de modèles est intégré dans une plate-forme nommée *Vercors* (acronyme de VERification de modèles pour COmposants Répartis communicants, sûrs et Sécurisés) ; une plate-forme dédiée à l'analyse de programmes distribués, notamment de programmes Java, et dont le schéma fonctionnel est donné par la Figure 6.7 ; Vercors prend en entrée le code source d'un programme Java, en l'occurrence *ProActive*, après application des techniques d'abstraction des données et analyse statique génère le modèle paramétré (pNet et pLTS) qui par instantiation produit un (ou plusieurs) modèles finis, utilisables par les outils de vérification.

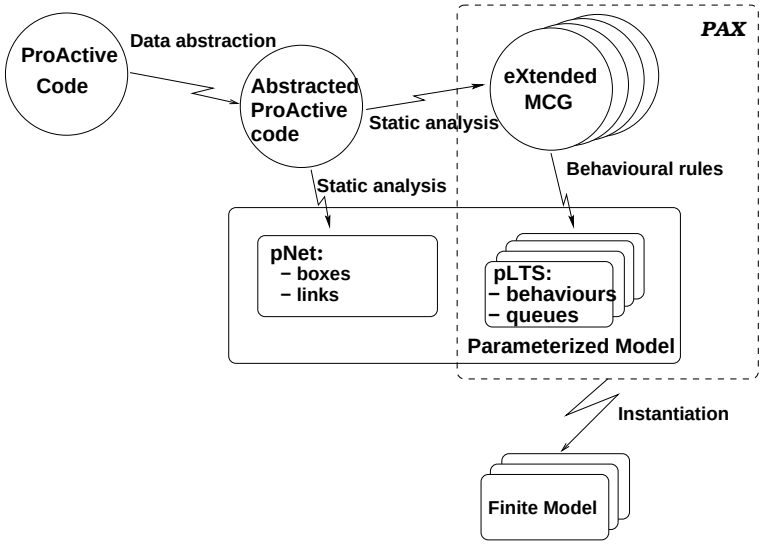


FIG. 6.7 – Architecture de la plate-forme Vercors

Les phases d'analyse statique du code *ProActive*, d'abstraction de données, et de production du MCG, sont en cours d'implantation dans un logiciel basé sur l'outil Bandera (travail de Christophe Massol).

Le prototype PAX a été implémenté dans le langage fonctionnel *OCaml*, un langage

dont les caractéristiques sont appropriées à la structure des règles, notamment le pattern matching et le typage.
Pour la génération de nos graphes nous avons utilisé la bibliothèque *Ocamlgraph* [16]. Cette librairie implémente les graphes par des foncteurs, i.e. indépendamment de la structure de leurs données, et fournit des primitives de haut niveau pour leur manipulation.

Par exemple, la procédure principale définie en section 5.3.7, qui calcule le pLTS d'une méthode, est traduite simplement en un programme OCAML (Listing 6.9).

Listing 6.9 – Procédure *Behav*

```
let method_behav g caller_obj (n0:mcgnode) =
   let s0 = LTS.V.create (fresh())
   and si= ref (LTS.V.create(-1))
   and aut = B.empty()
   and map = ref []
   in
   (
     Queue.add (n0,s0) todo;
     while ((Queue.is_empty todo)=false) do
      match choose() with
        (n,s) ->
                 if List.mem_assoc n !map then do_loop (n, s, map, aut)
                 else
               ((match (MCG.V.label n) with
                Ent (obj,meth,args)     ->
                      do_entry (s,obj,meth,args,map,si,n',aut)
              | Call (v,obj,d,meth,args) ->
                      do_call (v,obj,d,meth,args,map,si,n,s,aut,(MCG.succ_e g n'))
              | PP (_) -> do_pp (s,map,si,n)
              | Serve (ml,st1,mode,st2,l)  ->
                   do_serve (ml,st1,mode,st2,l,map,si,n,s,aut,(MCG.succ_e g n))
              | Ret (meth,v) -> do_return (caller,meth,v,s,map,si,n,aut)
              | Use (v,m) -> do_future (m,v,s,map,si,n,aut))
      ;
      let rec iter_succ list =
               match list with
                    [] -> ()
                  | e::l ->
               (let dest = MCG.E.dst e in
                (match MCG.E.label e with
                   T(mt) ->
                             let s= LTS.V.create (fresh()) in
                             let edge= LTS.E.create !si (string_of_action mt) s
                             in
                               LTS.add_edge_e aut edge;
                               Queue.add (dest,s) todo;
                 | C(_) -> ());
                 iter_succ l)

      in
         match (MCG.V.label n) with
         Ret(m,v) ->()
         | _ ->  iter_succ (MCG.succ_e g n))
   done;
   aut)
```

Notre prototype produit des pLTSs en format FC2 paramétré (format développé par

Tomàs Barros). L'outil d'instantiation permet de transformer un ensemble de fichiers Fc2 paramétré (pNets et pLTSs) en un fichier FC2 non paramétré utilisable par divers outils de vérification de modèles et d'équivalences.

6.5 Bilan

Nous venons de voir à travers quelques exemples des modèles pouvant être générés, à partir des règles définies dans le chapitre précédent, pour diverses applications et avec différents degrés de complexité. Ces règles qui jusqu'ici était définies conceptuellement sont expérimentées en les intégrant dans une plate-forme d'analyse de programmes Java.

Conclusion

Notre objectif initial était de nous attaquer au problème de vérification de propriétés comportementales d'applications distribuées, notamment d'applications Java. Précisément, il s'agissait de proposer et de calculer des modèles adéquats pour contrôler le comportement d'applications concurrentes. En conclusion générale de ce travail, nous pouvons dire que cet objectif est rempli, puisque les modèles générés à partir des applications *ProActive* nous permettent bien de vérifier des propriétés comportementales de ces applications. Mais ce serait là une réponse bien abrupte, faisant peu de cas de l'apport de ce travail.

En effet, la nature même des modèles choisis pour représenter les applications, les systèmes d'automates communiquants, semble limiter l'analyse à des systèmes assez simples, modélisant par exemple les protocoles de transfert de données d'un site à un autre. Cette classe n'est pas négligeable, cependant notre analyse adresse une classe plus large d'applications. En plus, des applications distribuées communiquants par échange de messages ("value-passing"), elle s'applique également à des applications dont la topologie est paramétrée voire dynamique, dont le nombre de processus n'est pas fixé préalablement et où la communication est symbolique.
En outre, bien que nous nous intéressons dans le cadre de ce travail au langage Java, les modèles et l'approche sont applicables à d'autres langages orientés objets voire à des frameworks de spécification de protocoles de communication distribuée, tels que *CREOL* [6].

Résumons notre approche. Nous représentons de manière finie des applications distribuées (finies ou infinies) par des réseaux de systèmes de transitions étiquetées. Ainsi le modèle global d'une application, construit de manière compositionnelle et hiérarchique, est un système de transition. Ce dernier est alors utilisé pour établir la preuve de propriétés comportementales du système modélisé, au moyen de model-checkers. Nous avons également présenté des arguments montrant que l'algorithme de construction de modèles termine et génère des structures finies.

Notre approche est semblable à celles adoptées dans [1, 23, 97] pour analyser et vérifier des programmes concurrents, elle modélise les interactions et les communications entre les threads ou objets ; les modèles sont construits à partir du code et l'analyse se base essentiellement sur les résultats d'analyse statique, notamment pour le calcul d'une énumération d'entités communicantes (threads, agents ou objets). Par contre, elle diffère

dans la démarche de construction : nos modèles sont construits de manière compositionnelle et hiérarchique, ce qui constitue un point fort. D'une part pour maîtriser le problème d'explosion d'états, l'opération de minimisation est appliquée à chaque étape de la composition ; d'autre part, pour l'économie de temps, le modèle d'un objet peut être réutilisé dans différentes constructions et remplacé par un autre objet pourvu que les deux objets aient la même interface (services offerts et services requis).
Par ailleurs, notre analyse, contrairement à la plate-forme Bandera [1] qui est la plus proche de ce travail par les applications et le langage qu'elle aborde, prend en compte la récursion des programmes ; les programmes avec des méthodes récursives sont également analysés et représentés de manière finie.

6.5.1 Contributions

L'apport de ce travail dans le domaine de vérification est la définition et la réalisation d'un générateur ouvert et générique pour les applications en général et les applications concurrentes en particulier. Ouvert dans le sens où il est applicable à différents langages ou protocoles de description d'applications. Générique car il fournit un modèle paramétré compact représentant toute une famille de modèles. Ce dernier est instancié à chaque vérification et les paramètres sont alors adaptés à chaque propriété à vérifier.

Notre contribution peut se résumer en plusieurs points :

- Nous avons défini une sémantique comportementale formelle sous forme de règles opérationnelles pour un noyau de *ProActive*. Cette sémantique nous a permis de construire des modèles comportementaux finis pour des programmes écrits dans ce noyau.
- Nous avons étendu la notion de systèmes de transitions et de réseaux de synchronisation à des homologues paramétrés (pLTSs et pNets) dont nous donnons nos propres définitions.
- Nous avons défini un langage de spécification graphique comportant des automates et des réseaux communiquants paramétrés, pour une utilisation intuitive par des développeurs non spécialistes ; nous avons défini également la traduction de la sémantique de ces spécifications graphiques dans celle des modèles. Ce qui nous a permis de comparer le pouvoir expressif des deux formalismes.
- Pour le même noyau à partir duquel nous avons défini des modèles finis, nous avons défini un système de règles de génération de modèles paramétrés.
- Nous avons implémenté ces règles ce qui a donné naissance à l'extracteur *PAX*.

6.5.2 Perspectives

Ce travail pourrait être étendu dans plusieurs directions.
Une première amélioration serait, bien sûr, d'étendre le noyau de *ProActive* étudié à tous

les aspects du langage, de façon à pouvoir analyser une famille plus grande d'applications et vérifier un nombre plus important de propriétés.

Une amélioration immédiate serait la prise en compte des exceptions, une notion liée au langage Java. Les programmes ne s'exécutant pas toujours de manière prévisible, il serait indispensable d'intégrer dans les modèles cette notion et de caractériser le comportement d'un objet après la levée d'une exception.

En fait, dans nos représentations de la topologie des applications les objets actifs ont une adresse logique. Le modèle est indépendant de la localité physique des objets, a priori, un objet actif peut se trouver à n'importe quel endroit. Ce qui rend les modèles applicables à toutes les configurations de topologies. Par contre, si nous voulons exprimer des propriétés de mobilité ou des propriétés de migration, par exemple, la propriété "un processus présent à une localité donnée, i, communique avec tous les autres processus". Il serait nécessaire d'ajouter à l'ensemble des paramètres, les paramètres symbolisant la localité physique. À l'évidence, la mobilité ne change pas, a priori, le comportement d'un objet ; cependant, les messages porteront davantage d'informations dont l'adresse physique et la topologie du réseau sera possiblement dynamique.

Nous pensons également à la communication de groupe, aujourd'hui, incontournable dans le cas des grilles de calcul. Cette communication de type un vers plusieurs s'applique typiquement à des applications parallèles où un certain nombre de processus coopèrent pour exécuter une seule tâche : quand un processus calcule un résultat, possiblement partiel, il le communique aux autres membres du système sans avoir à connaître le nombre, l'identité ou l'emplacement de chacun de ses membres. D'autant que, *ProActive* offre une abstraction orientée objet et des primitives [33] permettant de généraliser la communication point à point – objet à objet – à celle de groupe et de manipuler des groupes d'objets actifs aussi simplement que des objets actifs individuels. L'approche adoptée et les formalismes sélectionnés pour la construction des modèles pour la sémantique invocation point à point s'avèrent également appropriés et bien adaptés pour interpréter la sémantique de ce type de communication. En effet, la topologie du réseau et le comportement du serveur demeurent inchangés, il n'y a que le proxy du client (automate du futur) qui sera modifié de façon à coder l'émission vers (et la réception d') un ensemble d'objets.

Une autre piste de travail intéressante serait l'application des résultats de ce travail et des formalismes, mathématiques et graphiques, proposés pour la représentation d'entités plus générales que sont les composants (distribués).
La notion de composants va certainement devenir un des concepts clés pour la nouvelle génération d'architectures logicielles. En effet, les composants et les langages de description d'architecture favorisent la construction d'une application par réutilisation effective de modules logiciels existants en décrivant les interconnexions entre ceux-ci. Cette architecture et la conception des composants s'apparentent bien à notre approche ; visiblement dans nos réseaux graphiques une boite est un système fermé, permettant de décrire un composant offrant des services (la réception des requêtes) et requièrant d'autres (l'émission de requêtes).

Il est également souhaitable d'examiner les propriétés préservées par les différentes abstractions, notamment les abstractions des processus. En effet, la représentation symbolique des processus et le passage de ces derniers en paramètres entraînent des abstractions successives. Il serait donc intéressant de définir et d'étudier l'ensemble des propriétés qui sont garanties sur les instances. En se basant sur les travaux de Cleavland et Riely [57] nous pouvons dire que les propriétés de sûreté et de vivacité sont assurées pour le "value-passing". Évidemment, il reste à savoir les propriétés préservées par cette applicabilité au "process-passing".

Bibliographie

[1] Bandera. http ://bandera.projects.cis.ksu.edu/.

[2] Behave ! http ://research.microsoft.com/behave.

[3] Blast. http ://www-cad.eecs.berkley.edu/ rupak/blast/.

[4] CADP. http ://www.inrialpes.fr/vasy/cadp.html.

[5] CAV Tools. http ://www.cs.ccu.edu.tw/p̃ahsiung/courses/cav/resources/cav-tools.html.

[6] Creol. http ://www.ifi.uio.no/ creol/.

[7] Design/CPN. http ://www.daimi.au.dk/designCPN/.

[8] Estrel. http ://www-sop.inria.fr/meije/esterel/esterel-eng.html.

[9] Fc2Tools. http ://www-sop.inria.fr/meije/verification/doc.html.

[10] HyTech. http ://www-cad.eecs.berkeley.edu/ tah/HyTech/.

[11] JACK. http ://fmt.isti.cnr.it/fmt-tools.htm.

[12] Java pathfinder. http ://ase.arc.nasa.gov/people/havelund/jpf.html.

[13] Kronos. http ://www-verimag.imag.fr/PEOPLE/Sergio.Yovine/kronos.html.

[14] Modex/feaver. http ://cm.bell-labs.com/cm/cs/what/modex/.

[15] Murphi. http ://verify.stanford.edu/dill/murphi.html.

[16] Ocamlgraph. http ://www.lri.fr/ filliatr/ocamlgraph/.

[17] Proactive. http ://www-sop.inria.fr/oasis/ProActive/.

[18] Slam. http ://reearch.microsoft.com/slam.

[19] SMV. http ://www-2.cs.cmu.edu/ modelcheck/smv.html.

[20] Soot : a Java optimization framework. http ://www.sable.mcgill.ca/soot/.

[21] SPIN. http ://www.cs.washington.edu/research/projects/spin/www/.

[22] UPPAAL. http ://www.docs.uu.se/docs/rtmv/uppaal/.

[23] Zing. http ://research.microsoft.com/zing/.

[24] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., 1996. ISBN 0-38794-775-2.

[25] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers : Principles, Techniques, and Tools*. Addison Wesley, 1985. ISBN 0-201-10194-7.

[26] J. Alves-Foss and F.S Lam, editors. *Dynamic Denotational Semantics of Java*, volume 1523 of *LNCS*. Springer, 1999. ISBN 3-540-66158-1.

[27] K. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. In *Information Processing Letters*, 1986. 22(6) :307-309.

[28] A. Arnold. *Finite transition systems. Semantics of communicating sytems*. Prentice-Hall, 1994. ISBN 0-13-092990-5.

[29] A. Arnold. Nivat's processes and their synchronization. *Theor. Comput. Sci.*, 281(1-2) :31–36, 2002.

[30] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, Massachusetts, USA, 1996.

[31] 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.

[32] I. Attali, D. Caromel, and M. Russo. A formal and executable semantics for Java. In *Proceeding of Formal Underpinnings of Java*. OOPSLA'98 Workshop, Vancouver, 1998.

[33] L. Baduel, F. Baude, and D. Caromel. Efficient, flexible, and typed group communications in Java. In *Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 28–36, Seattle, 2002. ACM Press.

[34] T. Ball, S. Chaki, and S. K. Rajamani. Parameterized verification of multithreaded software libraries. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 158–173. Springer-Verlag, 2001.

[35] T. Barros, R. Boulifa, and E. Madelaine. Parameterized models for distributed Java objectss. In *International Conference on Formal Techniques for Networked and Distributed Systems FORTE'04*, 2004. LNCS 3235.

[36] T. Barros and E. Madelaine. Formalisation and proofs of the chilean electronic invoices system. Technical Report RR-5217, INRIA, June 2004.

[37] F. Baude, D. Caromel, F. Huet, and J. Vayssière. Objets actifs mobiles et communicants. *Technique et science informatiques*, 21(6–2002) :1–26, 2000.

[38] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *System and Software Verification : Model-checking Techniques and Tools*. Springer-Verlag Berlin and Heidelberg GmbH, Co. K, 2001. ISBN 3-540-41523-8.

[39] J.A. Bergstra, A. Ponse, and S.A. Smolka. *Handbook of Process Algebra*. North-Holland, 2001. ISBN 0-444-82830-3.

[40] E. Börger and W. Schulte. A programmer friendly modular definition of the semantics of Java. volume 1523 of *LNCS, Berlin*. In J. Alves-Foss, editor, Springer-Verlag.

[41] E. Börger and R. F. Stark. *Abstract State Machines*. Springer-Verlag, USA, 2003. ISBN 3-540-00702-4.

[42] R. Boulifa and E. Madelaine. Preuve de propriétés de comportement de programmes Proactive. Technical Report RR-4460, INRIA, may 2002. in french.

[43] R. Boulifa and E. Madelaine. Finite model generation for distributed Java programs. In *Workshop on Model-Checking for Dependable Software-Intensive Systems*, San-Francisco, June 2003. North-Holland, IEEE. ISBN 0-7695-1952-0.

[44] R. Boulifa and E. Madelaine. Model generation for distributed Java programs. In E. Astesiano N. Guelfi and G. Reggio, editors, *Workshop on scientiFic engIneering of Distributed Java applIcations*, Luxembourg, nov 2003. Springer-Verlag, LNCS 2952.

[45] G. W. Brams, C. Andre, G. Berthelot, C. Girault, G. Memmi, G. Roucairol, J. Sifakis, R. Valette, and G. Vidal-Naquet. *Reseaux de Petri : Theorie et Pratique. Tome 1 : Theorie et Analyse ; Tome 2 : Modelisation et Applications.* Editions Masson, September 1982.

[46] 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.

[47] M. Broy and E.-R. Olderog. Trace-oriented models of concurrency.

[48] L. Cardelli and A. D. Gordon. Mobile ambients. *Theor. Comput. Sci.*, 240(1) :177–213, 2000.

[49] 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.

[50] D. Caromel and L. Henrio. *A Theory of Distributed Objects.* Springer-Verlag, 2005. ISBN 3-540-20866-6.

[51] D. Caromel, L. Henrio, and B. Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages.* ACM Press, 2004.

[52] D. Caromel, L. Henrio, and B. Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages*, pages 123–134. ACM Press, 2004. ISBN 1-58113-729-X.

[53] 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.

[54] D. Caucal. On the regular structure of prefix rewriting. *Theor. Comput. Sci.*, 106(1) :61–86, 1992.

[55] P. Cenciarelli, editor. *Towards a Modular Denotational Semantics of Java.* ECOOP Workshops, 1999.

[56] E. Clarke, J.O. Grumberg, and D.A. Peled. *Model-Checking.* The MIT Press, 1999. ISBN 0-262-03270-8.

[57] R. Cleaveland and J. Riely. Testing-based abstractions for value-passing systems. In *International Conference on Concurrency Theory (CONCUR)*, volume 836 of *Lecture Notes in Computer Science*, pages 417–432. Springer, 1994.

[58] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, S. Laubach, and H. Zheng. Bandera : Extracting finite-state models from Java source code. *Int. Conference on Software Engineering (ICSE)*, 2000.

[59] J. C. Corbett. Constructing compact models of concurrent java programs. In *International Symposium on Software Testing and Analysis*, pages 1–10, 1998.

[60] P. Cousot. Interprétation abstraite. *Technique et science informatique*, 19(1-2-3) :155–164, January 2000.

[61] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[62] Patrick Cousot and Radhia Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM Press, 1977.

[63] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 77–101. Springer-Verlag, 1995. ISBN 3-540-60160-0.

[64] G. Delzanno, J.F. Raskin, and L. Van Begin. Towards the automated verification of multithreaded java programs. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 173–187. Springer-Verlag, 2002.

[65] C. Demartini and R. Sisto. Static analysis of java multithreaded and distributed applications. In *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, page 215. IEEE Computer Society, 1998.

[66] S. Drossopoulou. *Towards an abstract model of Java dynamic linking and verification.* ACM SIGPLAN Workshop on Types and Compilation, Canada, 2000.

[67] S. Drossopoulou and S. Eisenbach. *Towards an Operational Semantics and Proof of Type Soundness for Java.* Springer-Verlag, March 1998.

[68] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Pasareanu, R., H. Zheng, and W. Visser. Tool-supported program abstraction for finite-state verification. In *International Conference on Software Engineering*, pages 177–187, 2001.

[69] E.A. Emerson and K.S. Namjoshi. Automatic verification of parameterized synchronous systems. In R. Alur and T. Henziger, editors, *Information Processing Letters*, Rutgers, 1996. 8th International Conference on Computer Aided Verification, CAV'96. 22(6) :307-309.

[70] J. Feret. Dependency analysis of mobile systems. In *European Symposium on Programming (ESOP'02)*, number 2305 in LNCS. Springer-Verlag, 2002. © Springer-Verlag.

[71] W. Fokkink. *Introduction to Process Algebra*. Springer-Verlag, 2000. ISBN 3-540-66579-X.

[72] H. J. Genrich and K. Lautenbach. The analysis of distributed systems by means of predicate ? transition-nets. In *Proceedings of the International Sympoisum on Semantics of Concurrent Computation*, pages 123–147. Springer-Verlag, 1979.

[73] S. M. German and A. P. Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3) :675–735, 1992.

[74] P. H. Hartel and L. Moreau. Formalising the safety of Java, the Java Virtual Machine and Java Card. *ACM Computing Surveys*, 33(4) :517–558, Dec 2001.

[75] Paul Havlak. Interprocedural symbolic analysis. Technical Report TR94-228, 17, 1997.

[76] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight java : a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3) :396–450, 2001. ISSN 0-164-0925.

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

[78] T. Jensen and F. Spoto. Class Analysis of Object-Oriented Programs through Abstract Interpretation. In F. Honsell and M. Miculan, editors, *FOSSACS*, volume 2030 of *LNCS*, pages 261–275, Genova, Italy, April 2001. Springer-Verlag.

[79] J.B. Jorgensen and K.H. Mortensen. Modelling and analysis of distributed program execution in BETA using coloured petri nets.

[80] R.P. Kurshan and K.L. McMillan. A structural induction theorem for processes. *Inf. Comput.*, 117(1) :1–11, 1995.

[81] A. Lakas. *Les Transformations Lotomaton : une contribution à la pré-implémentation des systèmes Lotos*. PhD thesis, Univ. Paris VI, June, 1996.

[82] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 235–248. ACM Press, 1992.

[83] D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized linear networks of processes. In *24th ACM Symposium on Principles of Programming Languages, POPL'97*, Paris, January 1997.

[84] O. Lhoták. *SPARK : A flexible points-to analysis framework for Java*. PhD thesis, Université McGill, Montreal, Décembre 2002.

[85] Huimin Lin. Symbolic transition graph with assignment. In U. Montanari and V. Sassone, editors, *CONCUR '96*, Pisa, Italy, 26–29 August 1996. LNCS 1119.

[86] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1996.

[87] R. Mateescu. *Vérification des propriétés temporelles des programmes parallèles*. PhD thesis, Institut National Polytechnique de Grenoble, Avril, 1998.

[88] W.M. McLendon and R.F Vidale. Analysis of an ada system using coloured petri nets and occurrence graphs. In *Proceedings of 13th International Conference on Application and Theory of Petri Nets, Sheffield, UK*, pages 384–388. Springer-Verlag, 1992.

[89] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Inf. Comput.*, 100(1) :1–40, 1992. ISSN. 0890-5401.

[90] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989. ISBN 0-13-114984-9.

[91] G. Naumovich, G. S. Avrunin, and L. A. Clarke. Data flow analysis for checking properties of concurrent java programs. In *Proceedings of the 21st international conference on Software engineering*, pages 399–410. IEEE Computer Society Press, 1999.

[92] T. Nipkow and D. von Oheimb. Javalight is type-safe-definitely. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 161–170. ACM Press, 1998. ISBN 0-89791-979-3.

[93] ISO : Information Processing Systems Open Systems I nterconection. Lotos - a formal description technique based on the temporal ordering of observational behaviour. ISO 8807, Aug 1998.

[94] H. D. Pande and W. Landi. Interprocedural def-use associations in C programs. In *Proceedings of the symposium on Testing, analysis, and verification*, pages 139–153. ACM Press, 1991.

[95] I. Pechtchanski and V. Sarkar. Dynamic optimistic interprocedural analysis : a framework and an application. *SIGPLAN Not.*, 36(11) :195–210, 2001.

[96] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[97] S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages*, pages 123–134. ACM Press, 2004. ISBN 1-58113-729-X.

[98] V. Roy and R. de Simone. Auto and autograph. In *Workshop on Computer Aided Verification, New-Brunswick*, pages 65–75. LNCS 531, Spring-Verlag, June 1990. also available as RR-4460.

[99] D. Sangiorgi and D. Walker. *PI-Calculus : A Theory of Mobile Processes*. Cambridge University Press, 2001.

[100] S.Graf and H.Saidi. Construction of abstract state graphs with PVS. In *Computer Aided Verification (CAV'97)*, Haifa, Israel, June 1997. Springer-Verlag, LNCS.

[101] Robert de Simone. Higher-level synchronizing devices in MEIJE-SCCS. *Theoretical Computer Science (TCS)*, 37 :245–267, 1985.

[102] R. F. Stark, E. Börger, and J. Schmid. *Java and the Java Virtual Machine : Definition, Verification, Validation*. Springer-Verlag New York, Inc., 2001. ISBN 3-540-42088-6.

[103] Inc. Sun Microsystems. RPC : Remote Procedure Call protocol specification version 2. RFC 1057, Network Working Group, June 1988.

[104] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 264–280. ACM Press, 2000. ISBN 1-58113-200-X.

[105] D. Syme. Proving Java type soundness. volume 1523 of *LNCS, Berlin*. In J. Alves-Foss, editor, Springer-Verlag, 1999.

[106] R. Vallé-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[107] R. Vallee-Rai and L. J. Hendren. Jimple : Simplifying Java bytecode for analyses and transformations. Technical Report RR-5217, Sable Research Group, McGill University, July 1998.

[108] W. E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 83–94. ACM Press, 1980.

[109] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Proceedings of the international workshop on Automatic verification methods for finite state systems*. Springer-Verlag New York, Inc., 1990. ISBN 0-387-52148-8.

Index

Résumé

Dans notre travail nous nous sommes intéressés à la vérification automatique de propriétés comportementales d'applications réparties par des méthodes fondées sur les modèles. En particulier, nous étudions le problème de génération de modèles à partir de programmes Java répartis et représentés par des systèmes de transitions communiquants.

Pour ce faire, nous définissons une sémantique comportementale de programmes ProActive, une librairie Java pour la programmation parallèle, distribuée et concurrente. À partir de cette sémantique nous construisons des modèles comportementaux pour des abstractions finies d'applications écrites dans ce langage. Ces modèles sont basés sur la sémantique des algèbres de processus et peuvent donc être construits de manière compositionnelle et hiérarchique. La construction de modèles finis n'est pas toujours possible. Pour pouvoir traiter des problèmes prenant en compte des données, ainsi que des problèmes concernant des topologies non bornées d'objets répartis, nous définissons une nouvelle notion de modèleles hiérarchiques, à base de systèmes de transitions paramétrés et de réseaux de synchronisation paramétrés. Moyennant des abstractions ces modèles permettent de spécifier des applications possiblement infinies par des représentations expressives, finies, et plus proche de la structure du code. Par ailleurs, nous définissons un système de règles sémantiques permettant de générer automatiquement ces modèles (finis ou paramétrés) à partir d'une forme intermédiaire, obtenue par analyse statique, des programmes analysés. Les modèles ainsi générés sont exploitables directement ou après instantiation par des outils de vérification.

Abstract

In this work, we aim at developing automatic methods for verification of behavioural properties of distributed applications by methods based on models. Specifically, we study the question of building models from the source code of the Java applications. The models are labelled transition systems.

Therefore, we define a behavioural semantics for ProActive, a Java library for concurrent, distributed, and mobile computing. From this semantics we build behavioural models for finite abstractions of applications. These models are based on process algebra semantics, so they can be built in a compositional manner. Building the finite models is not always possible. In order to deal the problems that take into account the data as well the problems concerning topologies with infinite objects, we define the notion of hierarchical models, based on parameterized transition systems and parameterized synchronisation networks. By means of abstractions these models can depict infinite applications by expressive and finite representations. On the other hand, we define a system of semantics rules for building the (finite or parameterized) models from an intermediate form of programs obtained by static analysis. The models generated this way are used directly or after instantiation, standard by verification tools.