Christophe Massol
Chris.massol@free.fr

IUP MIAGE Nice
Année 2002/2003

Outils d'analyse statique et de vérification pour les applications Java distribuées

Rapport de stage

INRIA – Projet OASIS
2004 route des Lucioles
BP 93
06902 Sophia Antipolis

Enseignant Responsable:
Olivier Dalle
Responsable du stage dans l'entreprise :
Eric Madelaine

REMERCIEMENTS

Je remercie tout particulièrement l'INRIA qui m'a permis d'effectuer un stage intéressant et enrichissant.

Je remercie également Monsieur Eric Madelaine ainsi que Madame Rabéa Boulifa pour toute leur aide apportée sur la compréhension du sujet ainsi que leurs précieuses explications.

Je remercie Monsieur Olivier Dalle pour son accompagnement tout au long du stage.

Enfin, je tiens à remercier le personnel de l'INRIA pour son accueil et sa sympathie.

SOMMAIRE

INTRODUCTION

Afin de développer mes connaissances avec profit et m'initier à la vie d'entreprise, l'enseignement universitaire professionnel de la MIAGE propose un stage de fin d'étude de 25 semaines (du 3 Mars au 31 août).

Parmi les nombreuses propositions, mon choix s'est porté sur l'INRIA.

Cet institut, spécialisé dans la recherche informatique, est toujours à la pointe de la technologie intégrant nombre de chercheurs de diverses nationalités.

J'ai évolué dans le projet OASIS, projet dont le but est d'offrir des outils pour la construction, l'analyse, la vérification et la maintenance d'applications réparties Java. J'ai été intégré dans l'équipe Vercors.

1 Présentation de l'entreprise

L'INRIA est un institut publique de recherche en informatique et automatique créé en 1967 par les ministères de la recherche, de l'économie, des finances et de l'industrie.
Les buts de cet institut sont de réunir les compétences du dispositif de recherche français afin de développer les nouvelles technologies qui sont un des moteurs important de notre économie, partager ses connaissances dans un cadre international, assurer la diffusion de son savoir, contribuer à la normalisation ainsi que de réaliser des expertises dans le domaine informatique. Cet institut doit également réaliser des programmes de coopération avec d'autres structures pour le développement des nouvelles technologies et réaliser des systèmes expérimentaux.

Cette vocation envers la haute technologie se traduit par des publications de concepts et de connaissances dans la littérature scientifique mais également par des logiciels et prototypes expérimentaux novateurs (comme exemple, nous pouvons citer la librairie ProActive – développée par le projet OASIS sur le site de Sophia Antipolis – qui facilite la programmation répartie Java entre plusieurs machines virtuelles). Pour mener à bien tous ses projets, l'INRIA met en place des partenariats avec le CNRS, les écoles d'ingénieurs, les universités, les industries ainsi que des relations européennes et internationales.

Cet établissement est partagé en 6 sites indépendants qui lui permettent de renforcer sa présence sur le territoire et de multiplier les partenariats. J'ai été intégré dans l'unité de recherche de Sophia Antipolis dirigée par Michel Cosnard. Cette unité a été créée en 1982 dans la plus grande technopole européenne. Elle intègre aujourd'hui 455 personnes formant 28 équipes de recherche. Ces équipes couvrent des sujets très variés de l'informatique et de l'automatique avec quatre thèmes principaux :

- Réseaux et systèmes ;
- Génie logiciel et calcul symbolique ;
- Interaction homme-machine, images, données et connaissances ;
- Simulation et optimisation des systèmes complexes.

Une équipe de recherche est organisée autour de 10 à 25 scientifiques. Elle intègre des chercheurs permanents de l'INRIA mais elle peut également intégrer des ingénieurs, des étudiants en thèse ainsi que des stagiaires ponctuels venant de diverses écoles. Elle est dirigée par un directeur de projet (personne habilitée à diriger des projets) et sa durée de vie est d'au plus douze ans.

2 Le projet OASIS

Le projet OASIS (Ref 15), créé par l'initiative d'Isabelle Attali en juin 1999, est un projet commun entre l'INRIA, le CNRS et l'UNSA. Son but est de proposer dans un contexte d'applications réparties des principes, techniques et outils pour la construction, l'analyse, la vérification et la maintenance de systèmes.

Le projet a été créé dans le contexte de l'explosion de l'Internet, du partage de l'information et de l'avènement du langage Java. La problématique du projet est exprimée par la difficulté liée à la programmation et à la maintenance d'applications réparties Java. Les objectifs sont alors de définir une plate-forme d'aide à la programmation ainsi qu'une plate-forme de développement, d'analyse et de vérification d'applications réparties sur un réseau local (LAN), sur un système de clusters ou sur le réseau Internet.

2.1 La librairie ProActive

La librairie Java ProActive (Ref 14) est la solution de l'équipe OASIS face à l'absence de bibliothèque basée sur la programmation de systèmes répartis. Elle est basée sur la JVM (Java Virtual Machine) Java de SUN et utilise le protocole *meta-objet* structuré sur la librairie RMI pour assurer le transport de l'information.

Concrètement, ProActive propose des primitives permettant de simplifier la programmation d'applications distribuées en Java et d'en garantir la sécurité (Figure 1). Cette librairie introduit la notion d'objets actifs.

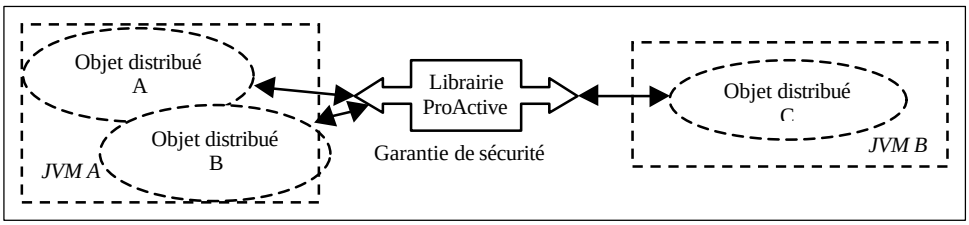

Figure 1: Communication ProActive

Un objet actif est un objet distribué par la librairie ProActive sur un réseau. Il peut alors effectuer des appels distants vers d'autres objets actifs sur le réseau ainsi que réceptionner des appels distants provenant d'autres objets actifs et y répondre. (**Figure 2**).

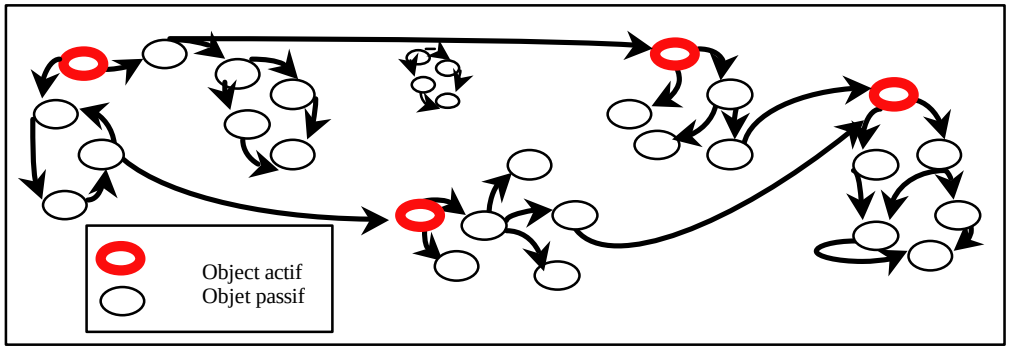

Figure 2 : Communication distribuée

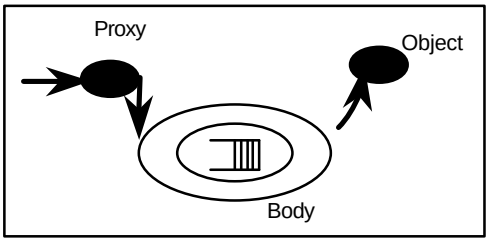

Figure 3 : Objet Actif

Un objet actif dispose d'une queue de requêtes (appels passés par l'intermédiaire de la librairie ProActive) représentée sur la Figure 3 par l'objet *Body*.

La librairie ProActive permet à l'utilisateur de définir le comportement des objets actifs. Le comportement de ces objets est caractérisé par la politique de traitement vis-à-vis des appels distants. Il est défini par l'utilisateur par la surcharge sur tout objet actif de la méthode *runActivity* liée à la librairie ProActive.

2.2 Environnement d'analyse et de vérification

L'équipe OASIS développe également un environnement d'analyse et de vérification de programmes dans le cadre de systèmes Java distribués : la plate-forme VERCORS (VERification de modèles pour COmposants Répartis communicants, surs et Sécurisés). Cette analyse repose sur la définition de modèles comportementaux définissant le comportement temporel des objets de l'application. En utilisant des outils de vérification basés sur ces

modèles comportementaux, des analyses peuvent être conduites afin de s'assurer du bon comportement (recherche d'inter-blocages, propriétés de sûreté).

Ces développements reposent sur les travaux des membres de l'équipe (Ref 1): définition d'une sémantique pour ProActive, d'un modèle intermédiaire (système d'automates paramètrés) pour exprimer les comportements, analyse statique permettant de déterminer certaines propriétés des applications.

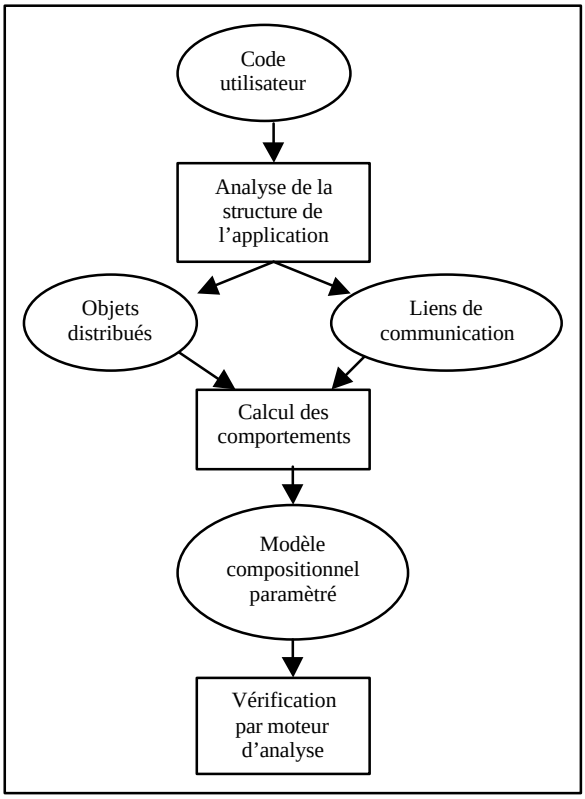

Figure 4 : Plate-forme Vercors

L'architecture de Vercors (Figure 4) repose sur la source Java du code utilisateur interprétée dans un langage intermédiaire proche du Bytecode, facile à analyser. La plate-forme va analyser la structure du programme distribué. Il est alors important de collecter les informations liées à la librairie ProActive :

- Les objets actifs créés par l'application ;
- Les points de communication entre objets actifs.

Nous calculons le comportement de l'application et le traduisons en modèle comportemental compositionnel. Les modèles comportementaux sont des automates représentant les actions et les états des objets de l'application.

Un moteur d'analyse va devoir valider les propriétés définies par l'utilisateur (recherche d'inter-blocage, test de sûreté, preuve de vivacité).

3 Objectifs du stage

3.1 Présentation du stage

Le cœur de la plate-forme Vercors est basé sur des outils d'analyse existants :

- Bandera (projet de l'université de Santos, Kansas)

Ce logiciel (Ref 3, Ref 5) permet d'analyser un code source Java et de l'interfaçer avec des vérificateurs de modèles comportementaux connus. Son analyse est basée sur un code intermédiaire permettant une analyse facilitée.

- Lande (projet IRISA, Rennes)

Les outils du projet Lande (Ref 9) permettent d'affiner l'analyse du code et de produire une analyse de classes afin d'établir un graphe précis d'appels de méthode.

- Soot

La plate-forme Soot (**Ref 10**) propose un langage intermédiaire entre le Java et le Bytecode permettant d'optimiser et d'analyser un code utilisateur. Cette plate-forme propose notamment l'outil Spark qui permet une analyse statique des pointeurs d'un programme

Bandera est incapable aujourd'hui de vérifier un code utilisateur Java distribué. Nous devons donc modifier son traitement et comprendre son architecture afin de le rendre compatible avec les traitements complexes de la librairie ProActive.

L'objectif pour l'équipe Vercors est de fournir - à l'aide des modules Bandera, Soot et Lande - un ensemble d'informations contenues dans le code utilisateur permettant la construction des modèles paramètrés compositionnels :

- l'ensemble des objets actifs créés dans le code ;
- l'ensemble des liens de communication entre objets actifs.

Ceci permettra de construire les automates représentant la structure du code utilisateur et de vérifier leur comportement grâce aux méthodes de génération de modèles finis dédiés aux systèmes distribués et aux outils de vérification fournis par l'équipe.

3.2 Plan de travail prévisionnel

Après une prise en main des outils à mettre en œuvre et une phase de prise de connaissances sur l'environnement scientifique du stage, nous avons émis un plan de travail prévisionnel :

- Comprendre le mécanisme utilisé par Bandera pour collecter l'ensemble des fichiers/classes utiles, et modifier cette procédure pour la limiter à ce qui nous intéresse (par exemple, ne pas analyser les classes internes de ProActive, ni les classes générées au vol) ;
- Interfaçer la plate-forme Vercors avec les outils proposés par le projet Lande ;
- Collecter les objets actifs ;
- Collecter les liens de communication vers les objets actifs ;
- Adapter le fonctionnement des modules proposés par Bandera pour une analyse ProActive ;
- Connecter les fonctions de génération de modèles du projet OASIS avec Bandera.

4 Les logiciels Bandera et Spark

4.1 Description de Bandera

Bandera est un logiciel permettant de gérer l'analyse de propriétés de programmes Java. Son architecture est basée sur un langage intermédiaire entre le code source Java et le Bytecode : le langage Jimple.

Bandera intègre plusieurs modules (Figure 5) : slicing, abstraction, vérification. Ils permettent d'optimiser le code utilisateur pour une génération d'automate optimal. Il permet également la gestion d'assertions et de propriétés, celle-ci étant spécifiées dans le code source utilisateur Java.

Bandera produit des modèles compatibles avec plusieurs moteurs de vérification de propriétés. Ces moteurs effectuent une analyse basée sur les propriétés utilisateur et fournissent des résultats permettant de visualiser pas-à-pas la propriété recherchée.

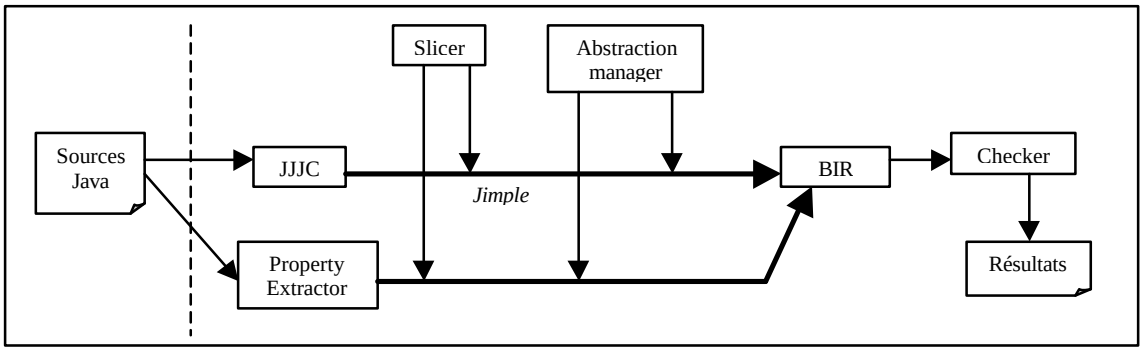

Figure 5 : Architecture de Bandera

4.1.1 Module de transformation Java -> Jimple (JJJC)

Ce module transforme un code source Java en langage intermédiaire Jimple (Ref 11). Ce langage intermédiaire entre le Java et le Bytecode facilite l'analyse d'un code utilisateur Java et le structure très précisément. Il construit une structure précise du code comme le Bytecode mais propose des variables typées ainsi qu'une optimisation du code (le Bytecode est très lourd en nombre de lignes de code). Le Jimple est produit par la plate-forme Soot.

4.1.2 Module d'analyse statique de classes

Ce module permet de réaliser un graphe d'appels de méthode pour l'ensemble du code d'une application. Ce graphe va nous renseigner sur le type de l'objet sur lequel les appels de méthode sont effectués.

```
Class A extends B {
      Public String toString(){ return « A »; }
}

Class B {
  Public String toString(){ return « B »; }
}

Class C {
  Public static void main(String[] argv){
      B b;
      B = new A();
     System.out.println(b.toString());
  }
}
```

Figure 6 : Analyse de classes

Dans le code Figure 6, *b.toString()* est un appel vers l'instance *b* de la méthode *toString()* du type *B.* L'analyse du module d'analyse de classe permet ici d'affiner l'information et de nous dire que l'instance *b* fait ici appel à la méthode *toString()* de l'objet *A.*

Dans cet exemple simple, il est facile de le définir mais dans un contexte plus complexe avec des instructions conditionnelles, de boucle ou interprocédurales, l'analyse devient très vite compliquée.

4.1.3 Module d'extraction de propriétés (Property Extractor)

Ce module (Ref 7) permet de relever toutes les assertions déclarées par l'utilisateur dans le programme afin d'être validées. L'utilisateur a la possibilité, par l'interface graphique proposée par Bandera, de rendre actif des assertions et de déclarer des propriétés à vérifier.

4.1.4 Module de découpage (Slicing)

Le slicing (découpage de code) est une méthode (Ref 4) permettant de supprimer les parties d'un code utilisateur afin de diminuer l'espace d'états à analyser (la taille de l'arbre

d'analyse). On va alors conserver seulement la partie du code sur laquelle agit la propriété qu'on souhaite vérifier et ne pas traiter de code superflu.

4.1.5 Module abstraction

A partir d'un code source Java, nous n'avons pas accès aux données dynamiques qui constituent le programme (Ref 6). Nous devons alors approximer les valeurs de celles-ci afin de pouvoir les traiter statiquement et garantir le comportement du programme accompagné de cette approximation.
Concrètement, l'abstraction permet de remplacer l'espace inconnu accordé à une variable du programme par un espace défini respectant le comportement du programme. Cela permet, par exemple, d'éviter l'explosion d'états dans les automates générés (gain de temps et de mémoire).

On peut par exemple sur une variable de type « Integer » appliquer une abstraction sur trois états : {négatif, zéro, positif}. Il faut bien sûr vérifier au préalable que l'abstraction soit pertinente et adaptée à la propriété à vérifier.

4.1.6 Module BIR (Bandera Intermediate Representation)

Ce module permet de construire un modèle de comportement du programme. Ce modèle est ensuite traduit en langage d'entrée correspondant au moteur de validation sélectionné.

4.1.7 Module de vérification (Model Checking)

Ce module vérifie le comportement du code utilisateur face aux propriétés déclarées par l'utilisateur. Il va permettre de le valider.
Bandera interface des moteurs de vérification connus comme SPIN, SMV et JPF en leur proposant en entrée les modèles obtenu à partir du module BIR.

4.2 Fonctionnement de Bandera

Bandera utilise pour son analyse les fichiers sources Java d'une application et y extrait les propriétés et assertions spécifiées dans le code utilisateur (Ref 2, Ref 5). Parallèlement, il va construire la représentation interne Jimple par le module JJJC. Ce module effectue

également une analyse statique de classes afin d'affiner la représentation interne du code source.

Sur cette représentation, Bandera va y appliquer son module de slicing ainsi que son module d'abstraction.

Le résultat obtenu est un code Jimple épuré et optimisé afin d'analyser les spécifications définies par l'utilisateur en écartant le code obsolète à cette spécification. Celui-ci est traité dans le module BIR qui calcule le modèle comportemental et qui le convertit en langage d'entrée du checker demandé.

Le checker effectue alors son analyse et retourne les résultats à Bandera qui les interprète de façon graphique.

4.3 Limites du logiciel Bandera

Bandera est dédié à l'analyse de programmes séquentiels ou concurrents et ne permet pas de traiter des applications réparties, applications construites sur plusieurs points pouvant être lancées sur différents systèmes. Bandera est également incompatible avec nos outils car il est basé sur une vision monolithique. En effet, le module BIR produit un code global pour l'ensemble de l'application, ce qui est incompatible avec nos outils de vérification compositionnels. De plus, Bandera ne peut traiter un code Java basé sur la librairie ProActive, notamment à cause de la génération de classes à la volée.

4.4 Module Abstraction proposé par Bandera

4.4.1 Définition de l'abstraction

L'abstraction permet de traiter un code utilisateur présentant des spécificités de nombres d'objets illimités, ou de nombres d'instructions illimitées traduisant une non finitude des automates. En effet, l'abstraction permet de réaliser un mapping d'un univers donné vers un univers plus simple à traiter et garantissant la finitude de l'automate. Un exemple simple peut être réalisé sur le domaine des entiers :

```
For(int x=-4 ; x<200){
      If(x==0)
            Foo() ;
      If(x>0)
            Foo2() ;
}
```

Figure 7 : Exemple d'abstraction sur « Integer »

Ici, nous pouvons réaliser une abstraction sur l'entier *x*. La plus appropriée ici est l'abstraction sur le signe de l'entier. Nous allons attribuer à *x* le domaine S={NEG, ZERO, POS} qui ne comporte plus que trois états.

4.4.2 Module Abstraction dans Bandera

4.4.2.1 Le langage BASL

Bandera utilise un langage BASL (Bandera Abstraction Specification Language) afin de définir des abstractions de type. Ce langage permet de définir des Tokens permettant d'établir un domaine (dans l'exemple ci-dessus, les tokens sont *{NEG, ZERO, POS}*) et de les mapper avec le code existant (Figure 8) grâce á la fonction *abstract (Type type).* Dans notre exemple, la fonction de mapping est :

```
Abstract(n)
      Begin
            n<0 -> (NEG) ;
            n==0  -> (ZERO) ;
            n>0  -> (POS) ;
End
```

Figure 8 : Mapping

Ce langage permet également de réaliser la définition des abstractions sur les opérateurs (+,-…). Dans notre exemple :

```
Operator + add
      Begin
            (NEG, ZERO) -> (NEG) ;
            ….
      End
```

Figure 9 : Définition d'opérateurs lors d'une abstraction

Le langage BASL permet de réaliser des abstractions sur les types de base (*int*, *double*…) mais également sur les classes ainsi que sur les tableaux (Figure 9). L'abstraction sur les classes permet de réaliser des abstractions plus poussées mais doit être défini au cas par cas (elles sont liées à la classe utilisateur qui la définie). L'abstraction sur les tableaux comprend deux types d'abstractions :

- L'abstraction de l'index du tableau ;
- L'abstraction du type d'objet contenu dans le tableau.

Bandera propose, suite à une définition d'abstraction, sa compilation afin d'être utilisable par le logiciel.

4.4.2.2 L'interface utilisateur

Le module graphique du logiciel permet de parcourir graphiquement le code utilisateur afin d'abstraire très simplement les variables du programme.

4.4.2.3 Résultats de l'abstraction

Le module abstraction va modifier le code utilisateur par un nouveau code Jimple valide utilisant les abstractions définies. Dans notre exemple ci-dessus, l'entier *x* va être remplacé par un objet de type *Sign*, les actions sur cet entier vont être également remplacées par les actions de type *Sign* correspondantes au mapping utilisateur.

4.4.3 Utilisation du module d'abstraction pour un programme utilisant la librairie ProActive

Le module abstraction produit un code Jimple valide. Il va donc pouvoir être analysé par notre plate-forme en aval de Bandera sans aucune modification de traitement de notre plate-forme.

4.5 Spark : outil d'analyse statique de pointeurs sur langage Jimple

L'analyse statique est une méthode d'étude de programme basée sur le code source et n'a donc pas accès aux données réelles dynamiques représentant le programme. Cela se traduit par une approximation et des expressions sur les données garantissant le comportement du programme.

Nous avons besoin, pour notre analyse du comportement d'un objet basé sur la librairie ProActive, d'étudier les pointeurs d'objets du programme utilisateur afin d'identifier leur appartenance éventuelle à la librairie ProActive. Un tel outil, travaillant sur une représentation Jimple, est présent dans la plate-forme Soot : Spark (Ref 13 et Ref 12).

Spark construit un graphe d'affectation de pointeur (PAG – Pointeur Assignment Graph) et associe, à chaque appel à un pointeur, son point de création. Nous allons présenter un exemple simple (Figure 10):

```
Public void m1(){
  P p ;
  P = new P() ;
  p.m() ;
}
```

Figure 10 : Analyse de pointeurs

Spark nous permet (Figure 10) d'identifier le pointeur lors de l'opération *p.m()*. Ici son analyse nous permet d'identifier l'instance *p* par son point de création *new P()* contenu dans la ligne 3.
Cela nous permet de définir un objet précisément par son point de création.

La principale limite de l'analyse de Spark est qu'elle traite un objet par un point de création dans le code statique. En effet, un point de création dans le code peut aussi bien définir un objet unique ou un ensemble d'objets. Spark peut alors nous donner un point de création pour un ensemble d'objets ayant un même point de création. L'analyse devient alors imprécise.

```
Public void m2(){
  P[] p = new P[10];
  for( i= ; i<5 ; i++){
    p[i]= new P();
  }
  p[4].m();
```

```
  p[3].m();
}
```

Figure 11 : Point de création

Ici, les pointeurs *p[3]* et *p[4]* (Figure 11) sont caractérisés par le même point de création dans l'analyse de Spark *new P()* contenu dans la ligne 4.

L'autre limite de Spark est liée au principe même de l'analyse statique : Cette analyse traite le comportement du programme sans le lancer. Il y a donc des approximations qui sont utilisées dans les cas complexes. Un comportement complexe dans le programme pour une analyse statique peut être un ensemble de traitements conditionnels comme des instructions « If » ou des boucles « while ».

```
Public void m3(int i) {
  Object o;
  If(i > 3)
    o = new P();
  else
    o = new Q() ;
  o.m();
}
```

Figure 12 : Ensemble de points de création

L'objet *o* peut prendre deux formes suivant l'attribut *i* : *P* ou *Q* (Figure 12).
Lors de l'analyse de l'appel *o.m()* à la ligne 7, Spark nous retournera un ensemble de points de création possible pour le pointeur *o* : {*new P()* ; *new Q()*}
Cela se traduit par les types possibles de l'objet *o* lors de l'appel *o.m()* dans la ligne 7.

4.5.1 Utilisation de Spark dans notre analyse

Spark convient très bien dans notre analyse car notre analyse est basée sur le modèle paramètré. En effet, un modèle paramètré construit un seul modèle pour un type d'objet ayant un même comportement et y applique des paramètres. Dans notre code, un type d'objet actif est caractérisé par son point de création. Il caractérise un ensemble d'objets actifs d'un même type (même Type Java, même comportement, mêmes attributs). Un point de création sera donc lié à un objet paramètré dans notre modèle. L'analyse de Spark convient tout à fait pour notre analyse puisqu'il nous donne un lien vers un ensemble d'objets actifs de même type.

5 Travaux effectués

La solution retenue est d'utiliser Bandera avec ses modules de parsing en code Jimple, de slicing, d'abstraction et de spécification. Ces modules sont intéressant à conserver car ils travaillent sur un code Java (ce qui est très rare) et sont particulièrement efficaces. Le module graphique de Bandera est lui aussi particulièrement intéressant car il permet à un néophyte d'utiliser des méthodes de Model Checking, ce qui constitue un des objectifs de l'équipe Vercors.

Nous allons lui intégrer les outils du projet Lande permettant d'analyser plus finement le fichier source pour produire un graphe d'appel précis. Puis, nous allons collecter les informations importantes pour la construction du modèle compositionnel : les objets actifs créés dans le programme ainsi que leurs points de communication. Nous allons ensuite appliquer des règles sémantiques permettant de construire un modèle comportemental fini, permettant son analyse par un model-checker. Ceci constitue le cœur de la plate-forme d'analyse Vercors (Figure 13).

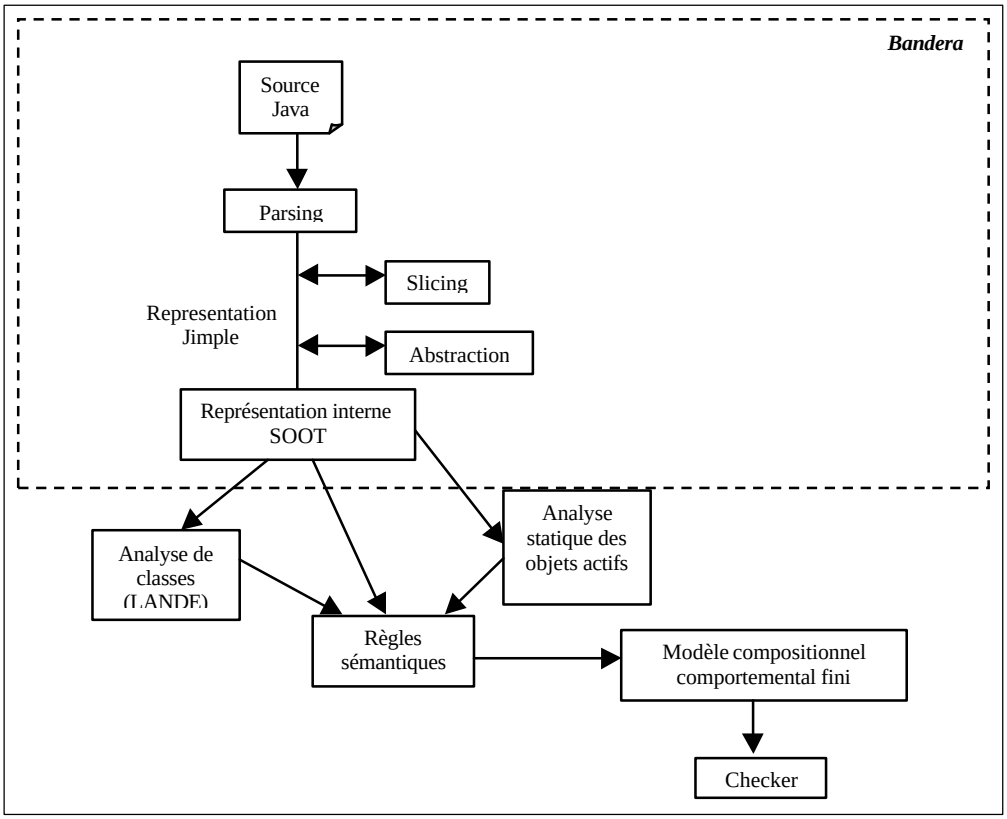

Figure 13 : Architecture Vercors

5.1 Adaptation de Bandera à ProActive

Bandera, dans sa version initiale, ne reconnaît ni les applications réparties, ni les traitements effectués par la librairie ProActive. En effet, la librairie ProActive créée des classes à la volée non analysables par Bandera. De plus, le traitement de la librairie ProActive est à écarter dans notre approche. En effet, nous souhaitons analyser les objets actifs ainsi que leurs communications et non pas le traitement interne des primitives fournies par ProActive dont le rôle est d'assurer la transmission et l'organisation des données (Figure 14).

La première étape a été de permettre à Bandera d'analyser le code utilisateur d'un programme utilisant ProActive en écartant l'analyse de la librairie ProActive. Nous pouvons alors simplifier le dialogue entre deux objets actifs par l'envoi de requêtes entre ces deux objets et le traitement effectué par chaque objet (en occultant la liaison avec l'API ProActive). Nous analysons donc le code utilisateur ainsi que le comportement des objets actifs vis-à-vis des requêtes qu'ils reçoivent en identifiant les primitives fournies par la bibliothèque ProActive qui décrit ce comportement.

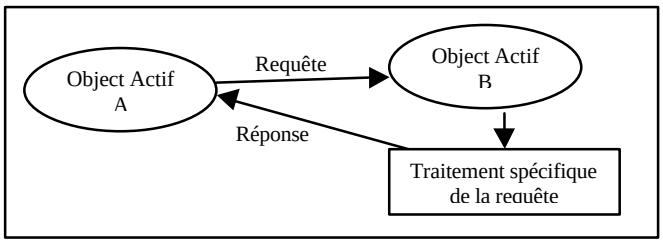

Figure 14 : Communication ProActive

Ce traitement a été pris en compte en modifiant le module JJJC du logiciel Bandera.

5.1.1 Modification du module JJJC

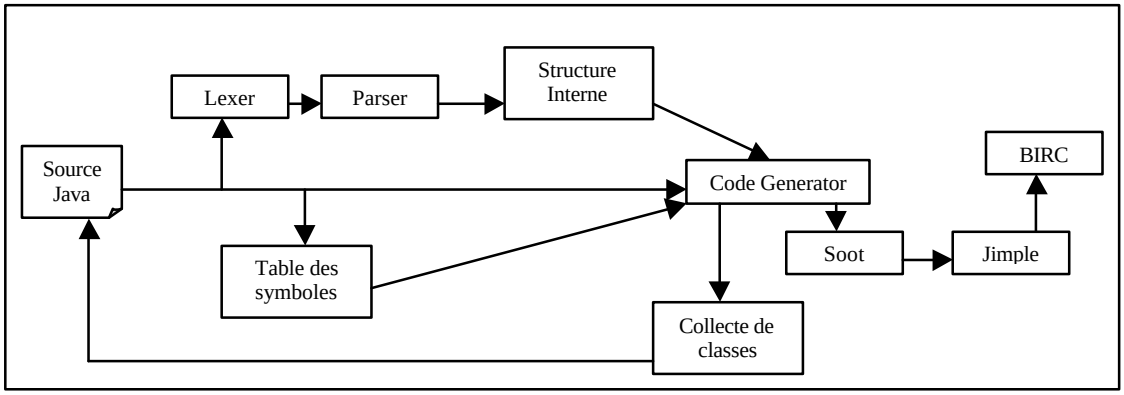

Figure 15 : Module JJJC

Le JJJC (**Figure 15**) commence par parser le fichier source contenant la méthode *main* grâce au module Lexer afin de produire une structure interne en forme d'arbre Java. En parallèle, le JJJC construit la table des symboles caractérisant les variables du code source. Ceci est alors traité par le module *code generator* qui parcourt l'arbre interne afin de produire la structure Jimple à l'aide de l'outil SOOT. Lors du traitement de l'arbre, Bandera détecte toute déclaration d'instance de classe et les collecte afin de traiter tout l'ensemble des classes du programme utilisateur. Une fois la collecte effectuée dans le fichier source, le processus JJJC est relancé pour chaque classe déclarée afin d'analyser l'ensemble des classes du programme.

5.1.2 Prise en compte de ProActive

Pour ignorer le traitement des classes ProActive (pour les raisons exprimées au-dessus), nous allons agir sur le processus de collecte de classes. Lorsque le *code generator* souhaite ajouter une classe à traiter, nous testons le nom de cette classe qui est caractérisé par le nom de package Java puis le nom de la classe.

Si le nom du package Java correspond au package Java ProActive (*org.objectWeb.proActive*), nous annulons l'ajout de celle-ci au processus de collecte de classe. De cette façon, le module JJJC est capable de convertir un code source d'un programme basé sur la librairie ProActive en structure interne Jimple.

5.2 Intégration des outils d'analyse de classe Lande

L'analyse de classes Java proposée par le projet Lande produit, à partir de Bytecode Java, un graphe d'appel au flot de contrôle. Cet outil est basé sur la plate-forme SOOT et le langage JIMPLE tout comme Bandera. Nous utilisons ces outils pour avoir un renseignement précis du flot de contrôle.

L'intégration des outils Lande va s'appliquer au cœur du logiciel Bandera après optimisation du code par ses modules de slicing et d'abstraction.

Plusieurs problèmes sont apparus lors de l'intégration des deux outils :

- Différence importante de version entre le SOOT-Bandera et le SOOT-Lande ;
- Différence entre les versions du langage JIMPLE des deux représentations.

Plusieurs choix se sont présentés :

- Modifier le code de l'outil Lande pour le rendre compatible avec l'architecture Bandera

  Cette solution présente l'avantage de garder intacte l'architecture de Bandera, nous permettant d'intégrer facilement des mises à jour de celui-ci.

  Cette solution nous rend dépendants de la version actuelle de Lande car, reporter les modifications dans une version future, pourrait être une tâche très lourde.

- Modifier le code de Bandera pour le rendre compatible avec l'outil Lande

  Cette solution nous rend dépendants de la version actuelle de Bandera et rend difficile l'intégration de futures versions de celui-ci.

- Utiliser le code Jimple produit par Bandera et son architecture interne pour appeler l'outil Lande.

  Ici l'outil Lande peut facilement reconstruire sa représentation interne propre en parsant le source Jimple.

  Cette solution nous paraissait bonne car, dans celle-ci, nous sommes seulement dépendant du code Jimple.

Cette solution n'a pas pu être retenue car le langage Jimple lui-même a évolué et n'est pas compatible avec nos deux versions (Jimple-Bandera, Jimple-Lande).

- Utiliser Bandera pour traduire sa représentation interne en source Java, pour ensuite les compiler et les fournir à l'outil Lande qui permet de travailler à partir de classes compilées.

  Cette solution est facile à mettre en œuvre mais elle utilise des traitements lourds comme la compilation de fichiers sources ainsi que le parsing de Bytecode Java. De plus, elle effectue un traitement redondant.

  Il est à noter qu'il est plus difficile alors de maintenir les liens de référence vers le code source.

- Créer un package permettant de traduire la représentation Soot de Bandera en représentation interne Soot-Lande (un package intermédiaire).

  Cette solution a un intérêt principal : on ne modifie ni le code de Lande, ni le code de Bandera. Nous pourrons donc très facilement utiliser de futures versions de ces deux logiciels. Ce package intermédiaire peut être facilement modifié au vu d'une mise à jour de la représentation interne d'un des deux outils. Il effectue un traitement beaucoup moins lourd que la solution précédente.

La dernière solution présentant des intérêts de performance et de compatibilité est retenue et implémentée. Différents tests ont été réalisés afin de valider la passerelle. Ces tests ont été faits en comparant les résultats de plusieurs codes par la passerelle avec les résultats obtenus directement par la dernière version de Soot (en occultant le traitement de Bandera).

5.3 Analyse statique des objets actifs

Pour analyser un programme écrit avec la librairie ProActive, le module de transformation en modèle paramètré a besoin de reconnaître les objets actifs présents dans le code utilisateur du programme Java à analyser.

Pour ce faire, et une fois que Bandera a construit sa représentation interne du code utilisateur et y a appliqué le Slicing et son module d'abstraction, une fonction parcourt cette représentation pour chercher les objets actifs.

L'analyse se porte donc sur les nœuds d'affectation de variable qui permettent de pouvoir rechercher les appels aux méthodes *turnActive* et *newActive* (méthode servant à créer un objet actif).

5.3.1 L'analyse des objets actifs

L'analyse rassemble les objets actifs dans une énumération d'objets pour pouvoir être identifiés et traités par le générateur de modèles paramètrés. Cette analyse parcourt l'arbre Jimple et collecte les objets actifs en analysant le code Jimple.

Cette collecte se fait sur la représentation interne du code Jimple par le logiciel SOOT dans laquelle nous allons rechercher des appels aux méthodes *turnActive* et *newActive* de l'API ProActive. Pour caractériser et identifier un objet actif, nous avons besoin de son type, son adressage mais nous avons également besoin des paramètres passés à la méthode d'instanciation de l'objet permettant de produire son comportement. Ces informations vont identifier de manière unique un objet actif.

Il faut alors analyser la structure Jimple du programme afin d'identifier un nœud de création d'objet.

5.3.2 La structure Jimple

Le Jimple est un langage proche du Bytecode Java. Il utilise des variables locales temporaires dans lesquelles il va stocker des informations et y appliquer des opérations. Ces variables locales sont assimilables à des registres. Le Jimple permet d'être proche du Bytecode tout en étant plus facile à analyser car beaucoup moins lourd (optimisé). Un autre avantage du Jimple est, qu'à la différence du Bytecode, ses variables sont typées.

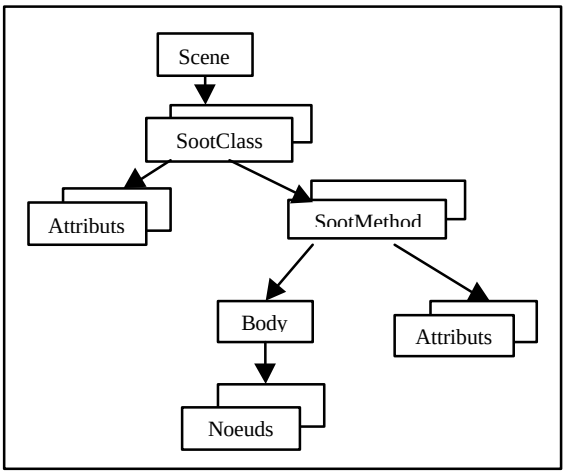

Figure 16 : Représentation Jimple

La structure interne représentant le code Jimple est constituée d'objets Java. L'objet *Scene* (Figure 16) est l'objet permettant d'accéder à toute la structure, c'est le point de départ pour parcourir un code Jimple. Il contient notamment une liste d'objets *SootClass* (Figure 16) caractérisant toutes les classes constituant le code utilisateur. Chaque classe contient une liste d'attributs et une liste de méthodes. Le plus intéressant se trouve dans l'objet *SootMethod* (Figure 16) : l'objet *Body* (Figure 16). Cet objet représente le corps d'une méthode et contient toutes les opérations effectuées par l'objet défini par l'utilisateur sous forme d'un arbre constitué de nœuds.

Un nœud peut prendre différentes formes : Une affectation, une instruction break, une définition, une instruction Goto, une instruction If, un appel de méthode, une instruction de retour, une levée d'exception. Tous ces nœuds contiennent différentes informations de type *Expression*. Nous allons nous intéresser au nœud d'affectation permettant d'identifier un appel de méthode de type :

objet = turnActive(…) ;
objet = newActive(…).

5.3.2.1 Jimple – Nœud Affectation

Un nœud d'affectation (Figure 17 et **Figure 18**) contient deux informations de type *Expression*. Nous nous intéressons ici à une certaine forme du nœud affectation :

- L'expression de gauche doit être soit un attribut de variable globale soit une variable locale ;
- L'expression de droite doit être un appel de méthode.

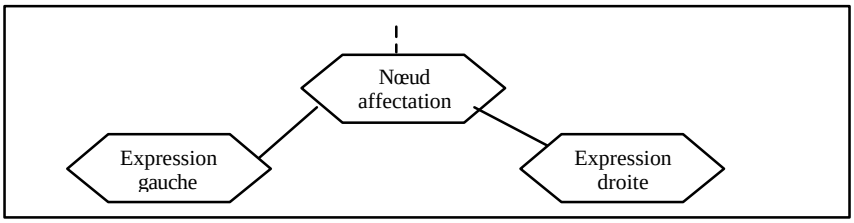

Figure 17 : Nœud affectation

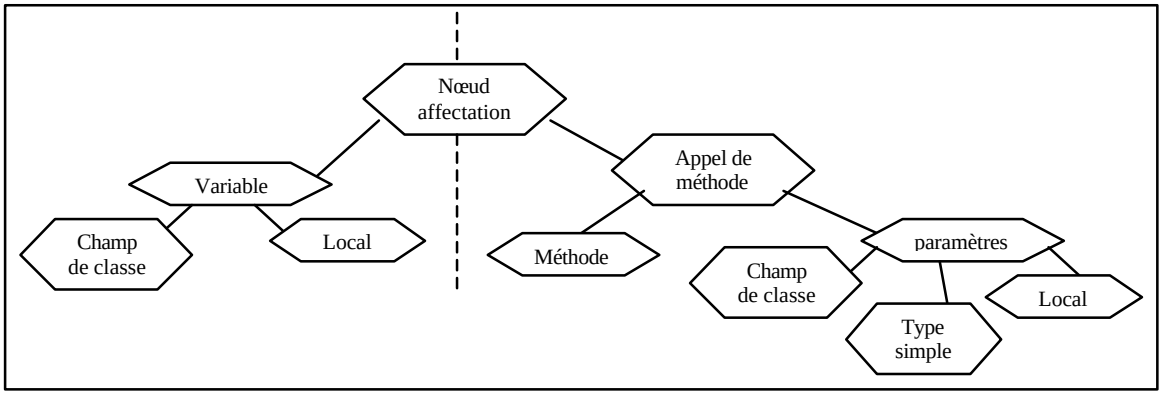

Figure 18 : Arbre Jimple

L'analyse de la variable contenu dans l'expression gauche doit pouvoir trouver le type de l'objet à affecter.

L'analyse de la variable contenu dans l'expression droite doit pouvoir identifier le nom de la méthode à appeler.

Nous pouvons alors voir si la méthode appelée est une méthode d'instanciation d'objet actif et nous pouvons collecter tous les renseignements contenus dans cet appel de méthode servant à identifier un objet actif (paramètres de la méthode).

De plus, pour ajouter une précision supplémentaire afin de pouvoir être certain d'identifier de manière unique un objet actif dans le code utilisateur, nous allons stocker l'objet *Expression* constituant l'instanciation de cet objet. Cela va permettre dans le point suivant de reconnaître à chaque appel à un objet actif, l'expression ayant servi à le créer et permettant ainsi de l'identifier (grâce à l'analyseur statique de pointeurs présent dans Soot).

5.4 Analyse statique des liens de communication

La collecte des nœuds d'appels à des objets actifs distants va affiner encore plus les informations pour pouvoir générer les modèles paramètrés. Cela se traduit par la distinction, lors d'un appel, des objets non actifs et des objets actifs. Tout ceci exprimé par la différence d'un comportement entre un objet actifs distant et un objet non actif vis-à-vis d'une réception d'un appel de méthode. En effet, l'objet actif va traiter les appels de méthode par rapport à sa queue d'appels fournie par la librairie ProActive. Ce traitement est spécifié par l'utilisateur dans son code dans une méthode ProActive liée à l'objet actif. Ceci traduit la communication avec l'objet actif.

Pour effectuer cette analyse, nous devons utiliser des techniques d'analyse statique de pointeurs pour identifier le type de l'objet appelé. Notre analyse va s'appuyer sur la structure Jimple de Soot. Cette structure utilise, comme nous l'avons vu plus haut, des variables locales temporaires pour exécuter ses traitements. Il va donc falloir que l'analyse statique nous donne un renseignement fiable et suffisamment précis pour identifier l'objet sur lequel l'appel de méthode est effectué.

Afin d'implémenter l'analyse des points d'appel aux objets distants, nous allons parcourir l'arbre de structure Jimple et repérer les points d'appels de méthode. Nous allons ensuite identifier le pointeur sur l'objet appelé. Pour faire cette identification, nous allons avoir recourt à l'analyse de Spark qui va nous informer en retour de l'ensemble de points de création référencés par le pointeur. Ces points de création vont nous permettre de connaître la nature de l'objet (actif, non actif) et de l'identifier.

5.4.1 Adaptation de Spark pour une analyse d'un code basé sur la librairie ProActive

La structure Spark est basée sur tout le code utilisé par le programme, c'est-à-dire que Spark va analyser toutes les librairies dont le programme a besoin (et pour ce faire, les traduire en structure Jimple). Notre problème est que la librairie ProActive est non analysable dans ce contexte (classes crées à la volée).

L'idée est donc d'arrêter l'analyse de Spark lors d'appels aux méthodes de la librairie ProActive et de lui faire assimiler qu'un appel à la méthode *newActive* est équivalent à un appel Java « new » - Ceci pour identifier les points de création d'objets actifs.

Spark va parcourir l'arbre de structure Jimple pour retrouver un appel Java « new » qui garantirait le point de création d'un pointeur (lors d'une affectation de pointeur). Nous voulons rajouter lors d'un appel aux méthodes statiques *newActive* et *turnActive*, un nouveau point de création dans le graphe PAG. Ce nœud de création dans le graphe va pouvoir faire le lien entre un pointeur référençant celui qui a reçu le point de création et son point de création d'objet actif. Cet ajout dans la méthode de collecte de Spark des éléments Java « new » a été alors implémenté.

Cette omission du traitement de la librairie ProActive implique que toutes les actions induites par celle-ci et qui sont intéressantes pour notre analyse doivent être traitées. Notamment, nous devons prendre en compte l'appel au constructeur de l'objet ainsi que l'appel à la méthode *runActivity* (appels fait dans les sous-actions des primitives de la librairie ProActive).

5.4.1.1 Intégration de l'appel constructeur

Le traitement de l'appel au constructeur est primordial dans notre analyse pour plusieurs raisons :

Cet appel va, dans un premier temps, définir le comportement de l'objet. En effet, son comportement peut changer en fonction des actions contenues dans son constructeur.

Dans un deuxième temps, l'analyse de l'appel constructeur va permettre d'effectuer une analyse statique correcte du code utilisateur. En effet, lors d'un appel à la méthode *newActive*, Spark ne sait pas, ici, qu'un objet est créé. Il ne va pas alors analyser le constructeur correspondant à cet objet. Il y a ici une perte importante d'informations qui ne garanti pas le comportement de l'objet.

La solution apportée est la suivante : nous rajoutons dynamiquement, après l'appel *newActive* analysé par Spark, un nœud Jimple d'appel au constructeur. Nous simulons le comportement de la librairie ProActive.

Pour définir le constructeur à appeler, la librairie ProActive propose le squelette de la méthode *newActive* suivant :

Object newActive(String nomObjet, Object[] parametresConstructeurs)

Les objets qui définissent l'appel au constructeur sont contenus dans un tableau d'objets. ProActive analyse alors le type dynamique de l'objet afin d'identifier le constructeur à appeler. Ici une difficulté se rajoute : nous devons analyser statiquement le type des objets contenus dans un tableau d'objets pendant l'analyse de Spark pour appeler le constructeur correspondant et lui passer en paramètres les pointeurs d'objets contenus dans le tableau. Cette analyse est compliquée car, tant que Spark n'a pas fini son traitement, on ne peut avoir des informations complètes sur le PAG. Une longue phase d'analyse spécifique sur Spark a été nécessaire afin d'identifier ce problème.

5.4.1.2 Intégration de l'appel à la méthode *runActivity*

Cet appel est nécessaire dans notre analyse car, en ignorant les primitives ProActive, nous ignorons les actions engendrées par cette bibliothèque dont l'appel à cette méthode. L'analyseur statique Spark ne traitera donc pas cette méthode et nous perdrions des informations très importantes pour notre analyse.

Cet appel est simple à traiter car, contrairement à l'appel constructeur, il ne comporte aucune donnée importante dans notre analyse en paramètre. Nous rajoutons alors, au cours de l'analyse des nœuds Jimple par Spark, un nœud d'appel à *runActivity* à la suite d'une création d'objet actif.

5.4.2 Identification d'une communication vers un objet actif

Lors de notre analyse des objets actifs, nous avons identifié de manière unique des ensembles d'objets actifs. Nous avons notamment mémorisé leurs points de création – un point de création dans la structure Jimple est une instance de l'objet Java *NewExpr* du package Java *soot.jimple* -.

Pour identifier les appels vers un objet actif dans le code utilisateur, nous allons parcourir l'arbre de nœuds Jimple (figure 16) et identifier les nœuds d'appels de méthode. Nous allons ensuite utiliser Spark pour identifier la nature du pointeur sur lequel cet appel est

effectué. L'analyseur de pointeur va nous fournir un ensemble de points de création correspondants. Il suffit alors de comparer par adressage les différentes instances retournées par Spark avec les instances caractérisant les créations d'objets actifs.

5.4.3 Limites de l'analyse

Les limites sont liées aux limites de l'analyse statique et aux limites de Spark. Dans certains cas, cette analyse ne pourra nous offrir qu'un ensemble de points de création qui sont susceptibles d'être référencés par le pointeur suivant le déroulement des boucles. On ne pourra pas alors identifier de manière unique un objet référencé par un pointeur.

5.4.4 Identification d'une instance d'un objet actif à un point donné

```
Public void call(int j){
      activeObjectsArray[j].m();
}
```

Figure 19 : Identification d'une instance d'un programme

Notre analyse statique présentée dans les chapitres précédents permet seulement d'associer à un pointeur donné un ensemble d'instances associées liées à un point de création. Nous sommes donc incapable d'identifier précisément l'instance référencée par un pointeur.

Sur la Figure 19, le tableau *activeObjectArray* référence des instances d'objets actifs de même type disposant d'une méthode *m()*. Nous devons identifier précisément l'instance sur laquelle est effectué cet appel à un instant donné afin d'avoir une information précise sur le comportement du programme. Aujourd'hui, nous ne sommes pas capable avec l'analyse fournie par Spark d'effectuer cette identification.

A la lecture de travaux antérieurs effectués sur d'autres langages comme le C (Ref 16), nous souhaitons pouvoir identifier toutes les dépendances liées à un pointeur afin de pouvoir générer un contexte d'environnement. Nous devons ensuite exprimer ce contexte par une expression qui, en fonction de paramètres, pourrait identifier finement l'instance appelée à un point donné.

5.5 Planning récapitulatif

Travail	*Durée (en semaines)*
Prise de connaissances sur l'univers technique du stage : • Model checking, slicing, abstraction, Bandera	1

Travail	*Durée (en semaines)*
Prise de connaissances sur l'univers technique du stage : • Jimple, Soot, ProActive	1

Travail	*Durée (en semaines)*
Identification du processus de collecte de classe du module JJJC de bandera • Ecartement de l'analyse de la librairie ProActive • Présentation des travaux à l'équipe	1

Travail	*Durée (en semaines)*
Travail sur l'intégration des outils Lande Rennes en collaboration avec un ingénieur de Rennes	1

Travail	*Durée (en semaines)*
Prise de connaissance plus spécifique sur la structure de Soot et implémentation de la reconnaissance statique des points de création des objets actifs	2

Travail	*Durée (en semaines)*
Prise de connaissance sur l'analyse statique de pointeurs et de l'outil Spark. Identification des besoins pour notre analyse • Présentation à l'équipe de l'outil Spark et de mes réalisations	2

Travail	*Durée (en semaines)*
Correction des bugs de la passerelle Soot beta4/soot 1.2.5 et ajouts de fonctionnalités	2

Travail	*Durée (en semaines)*
Intégration de Spark dans notre analyse • Reconnaissance des nœuds de création spécifiques à un objet actif • Intégration des actions de la librairie ProActive	6

Travail	*Durée (en semaines)*
Collecte des communications entre objets actifs • Collecte des points de sortie d'un objet actif • Collecte des points d'entrée	3

Travail	*Durée (en semaines)*
Etude du module abstraction	1

Travail	*Durée (en semaines)*
Interfaçage de Vercors à l'architecture Bandera	1

Travail	*Durée (en semaines)*
Préparation du compte-rendu • Ecriture du rapport écrit • Etude de la présentation orale • Pré-soutenance devant l'équipe OASIS	3

5.6 Avancement

5.6.1 Les fonctionnalités implémentées

- Bandera compile en code Jimple une application utilisant la librairie ProActive.
- Le code Jimple est correctement mis à jour par la passerelle afin de pouvoir utiliser Spark et les outils du projet Lande.
- Le module abstraction de Bandera peut être utilisé sans modifications de code.
- Les objets actifs sont identifiés et placés dans une énumération.

5.6.2 Les fonctionnalités non terminées

- L'ajout de l'appel constructeur et de la méthode *runActivity* (traitements internes de la librairie ProActive).

  La difficulté est que nous sommes en cours d'analyse. Spark ne possède pas assez d'informations sur les paramètres à passer aux constructeurs et nous sommes dans une impasse.

- Analyse des points d'appel distants :

  L'information sur les appels distants dépend du point précédent. Par contre, la technique d'analyse de ces appels a été bien implémentée.

5.6.3 Les problèmes ouverts

- Prise en compte de l'identification unique d'un objet actif dans le code utilisateur.

CONCLUSION

Ce stage était très intéressant car il m'a plongé dans le monde de la recherche informatique, monde à la pointe de l'avancée informatique. Cet univers m'a beaucoup apporté en connaissances informatiques dans des domaines non étudiés lors de la formation dispensée à l'IUP MIAGE. J'ai acquis les principes du model checking, de l'analyse statique de programmes et de l'analyse statique de pointeurs (ou d'alias),. J'ai pu également participer à nombres de colloques effectués par des invités prestigieux comme des chercheurs français où étrangers venant présenter leurs travaux dans les locaux de l'INRIA. Cela m'a permis d'élargir mes connaissances dans divers domaines pointus de l'informatique.

De plus, j'ai appris à m'intégrer dans une équipe déjà formée et en fonction, à effectuer une prise de connaissance rapide, enrichissante et conséquente sur les différents univers techniques abordés lors du stage (model checking, slicing, abstraction, analyse statique) grâce à des supports d'informations divers comme des présentations soutenues dans divers instituts de recherche, des rapports de recherche ou de la documentation technique disponible sur Internet. Toute cette documentation était bien sûr rédigée en anglais, ce qui m'a permis de développer mon Anglais technique.

Cet univers technologique m'a également aidé à exposer mes expérimentations aux membres du projet. Lors de ces présentations, j'ai pu consolider mon aisance pour exposer un travail mais également me remettre en question pour mieux progresser. En effet, lors de ces présentations, les remarques des membres du projet INRIA m'ont permis d'évoluer et de mieux comprendre certains points difficiles.

Cette expérience m'a également initiée à la vie en entreprise, liée aux engagements, à la tenue d'objectifs à long et court terme sous un rythme soutenu.

Pour finir, Je suis très fier d'avoir réaliser ce travail dans ce contexte intéressant, ardu et incontestablement riche qu'est l'INRIA.

REFERENCES

Ref 1 : Preuves de propriétés de comportement de programmes ProActive, Rabéa Bouliffa, Eric Madelaine, mai 2002

Ref 2 : The Bandera Tools for motel-checking Java source code : A user's manual, John Hatcliff, Oksana Tkachuk, march 2001

Ref 3 : Bandera : Extracting Finite-state Models from Java Source Code, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, Hongjun Zheng, 2000

Ref 4 : Slicing software for model construction, John Hatcliff, Matthew Dwyer, Hongjun Zheng, December 1999

Ref 5 : Using the Bandera tool set to model-check properties of concurrent Java software, John Hatcliff, Matthew Dwyer

Ref 6 : Tool-supported Program Abstraction for Finite-state Verification, Matthew Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina Pasareanu, Robby, Willem Visser, Hongjun Zheng, in Proceedings of the 23rd International Conference on Software Engineering, May, 2001

Ref 7 : Expressing Checkable properties of dynamic systems : The Bandera specification language, James C. Corbett, Matthew B. Dwyer, John Hatcliff, Robby, June 2001

Ref 8 : Constructing Compact Models of Concurrent Java Programs, James C. Corbett, In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), March, 1998.

Ref 9 : Class Analysis of Object-Oriented Programs through Abstract Interpretation, Thomas Jensen and Fauto Spoto

Ref 10 : Soot - a Java Optimization Framework**,** Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon and Phong Co, September 1999

Ref 11 : Jimple : Symplifying Java Bytecode for analyses and transformations, Raja Vallée-Rai, Laurie J. Hendren

Ref 12 : Spark Options, Ondrej Lhotak, February 2003

Ref 13 : Spark: A flexible points-to analysis framework for Java, Ondej Lhoták, February 2003

Ref 14 : Towards Seamless Computing and Metacomputing in Java

D. Caromel, W. Klauser, J. Vayssiere, pp. 1043--1061 in Concurrency Practice and Experience, September-November 1998

Ref 15 : OASIS : Objets actifs, Sémantique, Internet et sécurité, Juin 1999

Ref 16 : Automatic determination of communication topologies in mobile systems, A. Venet, 1998