Analyses Statiques et Dynamiques des Programmes

Abdelhak-Djamel Seriai

Nota :
Le contenu de ce cours a été préparé en se basant sur plusieurs sources.
Parmi ces sources :

- Jonathan Aldrich Charlie Garrod : Principles of Software Construction: Objects, Design and Concurrency ; Static Analysis
- Plusieurs documents/cours rédigés par Bruno Duffour (notamment analyse dynamique)
- Rapport de synthèse de Jean-Yves Bouttery
- Etc.

Analyse de programme

Définition

Analyser le comportement/propriétés d'un programme automatiquement

Techniques d'analyse

- Analyses statiques
  - Basées sur l'analyse du code source
  - Considèrent toutes les exécutions possibles
  - Calculs complexes mais sans impact sur l'exécution

- Analyses dynamiques
  - Basées sur l'analyse d'une ou plusieurs exécutions du code
  - Considèrent certaines exécutions concrètes
  - Impact sur l'exécution proportionnel à la quantité d'information recueillie

Techniques d'analyse

- Avantages de l'analyse statique
  - Pas d'exécution, donc pas de dommages et pas de temps d'exécution.

- Inconvénients de l'analyse statique
  - Il n'est pas possible d'être certain de certaines propriétés avec l'analyse statique à cause du problème de l'indécidabilité.
    - Exemple,
      - Dans un programme assembleur, une instruction indique de sauter à une adresse contenue dans un registre quelconque.
        - Puisqu'il n'y a pas d'exécution, on ne connaît pas cette adresse et les possibilités de valeurs du registre peuvent être très grandes. On ne pourra donc pas étudier statiquement tous les scénarios d'exécution possibles du programme.

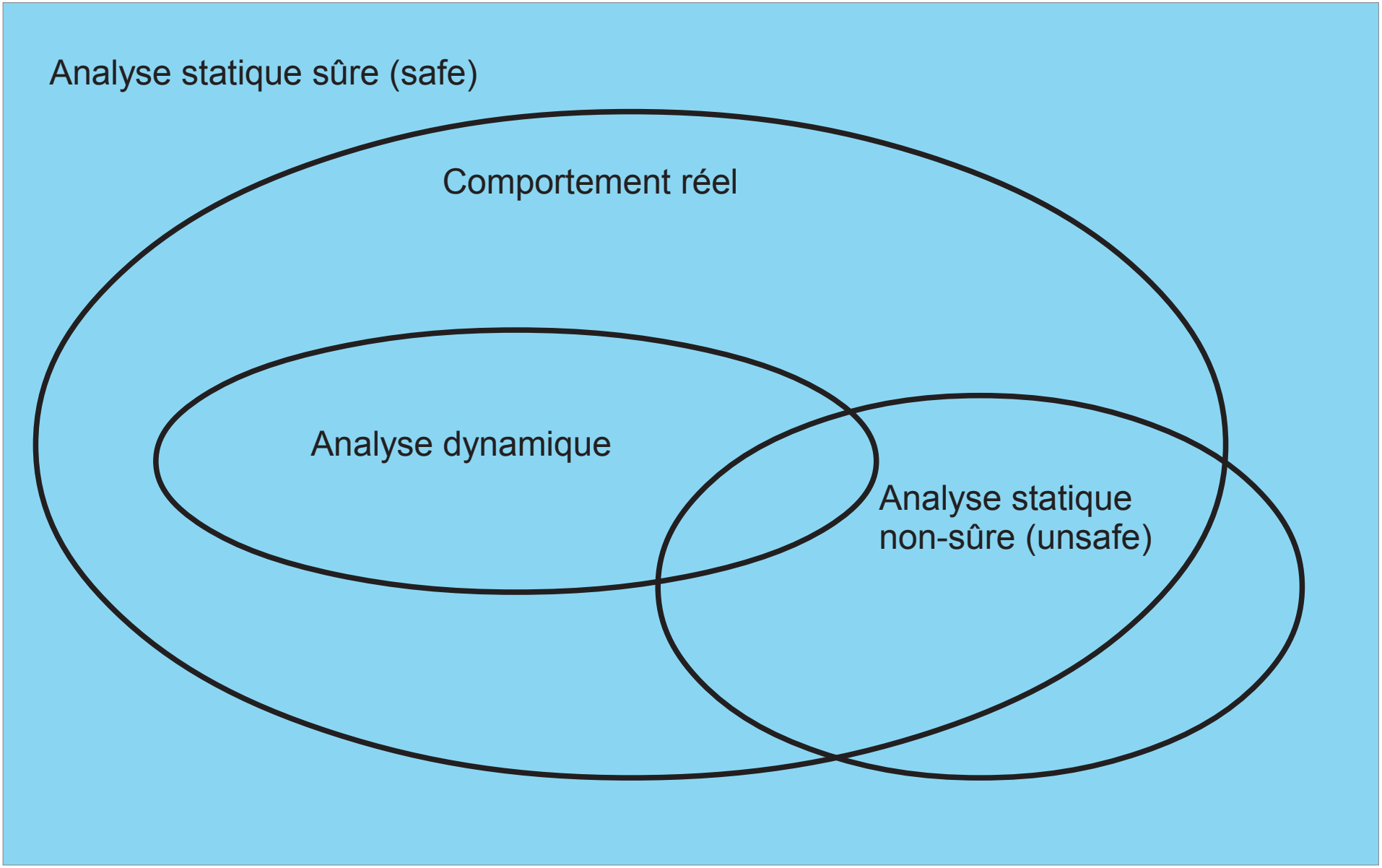
Techniques d'analyse

- Avantages de l'analyse dynamique
  - Permet d'obtenir des résultats plus précis pour une ou plusieurs exécutions concrètes
  - Permet d'obtenir de l'information de nature temporelle à propos de l'exécution
    - Temps d'exécution/performance, temps resté sur une partie du programme
  - Permet d'obtenir de l'information sur la fréquence ou l'importance de certains événements
    - Exemple une méthode est appelée très souvent, beaucoup de tableaux sont créés, etc.

Techniques d'analyse

- Inconvénients de l'analyse dynamique
  - L'étendu de l'analyse dépend de l'étendu des scénarios d'exécution
    - Besoin de scénarios qui couvrent l'ensemble du code à analyser
  - Possibilité de dommage en cas d'analyse pour des raisons de sécurité (comportement malicieux)
  - Dépendance par rapport au temps d'exécution

Approximation du comportement



Analyse de programme

Objectif

Vérification

Compréhension

Transformation

Décompilation

Obfuscation

Optimisation

Etc.

Pourquoi Analyser ?

- Sûreté
  - Détection de code malicieux
    - Exemple
      - Utilisation d'automates de sécurité.
        - Les transitions correspondent aux instructions du programme
        - Un ou plusieurs états ne doivent pas être atteints :
          - s'ils le sont, cela signifie que la politique n'a pas été respectée
        - Si un scénario d'exécution du programme possède une suite d'instructions, donc de transitions de l'automate, qui fait atteindre un état tel que la politique n'est pas respectée, le programme est considéré comme malicieux

Pourquoi Analyser ?

- Détection d'erreurs pouvant survenir à l'exécution
  - Exemples
    - Accès à une variable à partir d'un pointeur nul
    - Accès à un tableau à l'extérieur de ses bornes
    - Cast illégal
    - Boucle infinie
    - Chemin sans retour
    - Variables non initialisées
    - Synchronisation inconsistante
  - Exemple d'outil : FindBugs
    - Outil Standalone ou plugin Eclipse, etc.
    - Vérifie plus que 250 patrons d'erreurs
    - http://findbugs.sourceforge.net/

Pourquoi Analyser ?

- Détection d'erreurs pouvant survenir à l'exécution.
  - A null pointer checker
    - Vérification des pointeurs null basée sur l'analyse du flot de données

```
1. int foo() {
2. Integer x = new Integer(6);
3. Integer y = bar();
4. int z;
5. if (y != null)
6. z = x.intVal() + y.intVal();
7. else {
8. z = x.intVal();
9. y = x;
10. x = null;
11. }
12. return z + x.intVal();
13. }
```

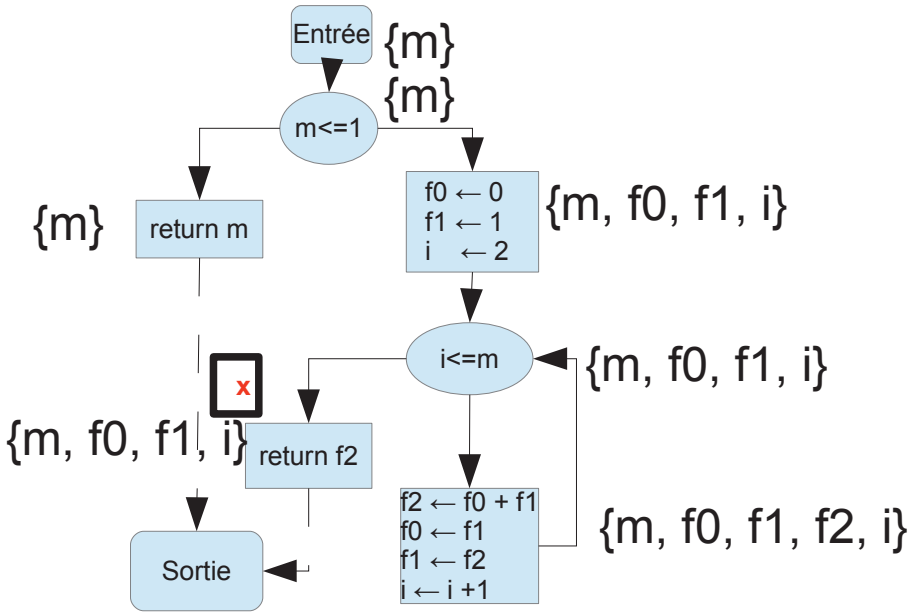
```
3: x not-null
4: x not-null, y maybe-null
5: x not-null, y maybe-null
6: x not-null, y not-null
8: x not-null, y null
9: x not-null, y null
10: x not-null, y not-null
12: x maybe-null, y not-null
```

Error: may have null pointer on line 12

Pourquoi Analyser ?

- Détection d'erreurs pouvant survenir à l'exécution
  - Variable non initialisée

```
int fib(int m) {
        if (m <= 1) {
                return m ;
        } else {
          int f0 = 0, f1 = 1, f2, i ;
          for (i=2 ; i <= m ; i++) {
                f2 = f0 +  f1 ;
                f0 = f1 ;
                f1 = f2 ;
        }
        return f2 ;
        }
}
```



Pourquoi Analyser ?

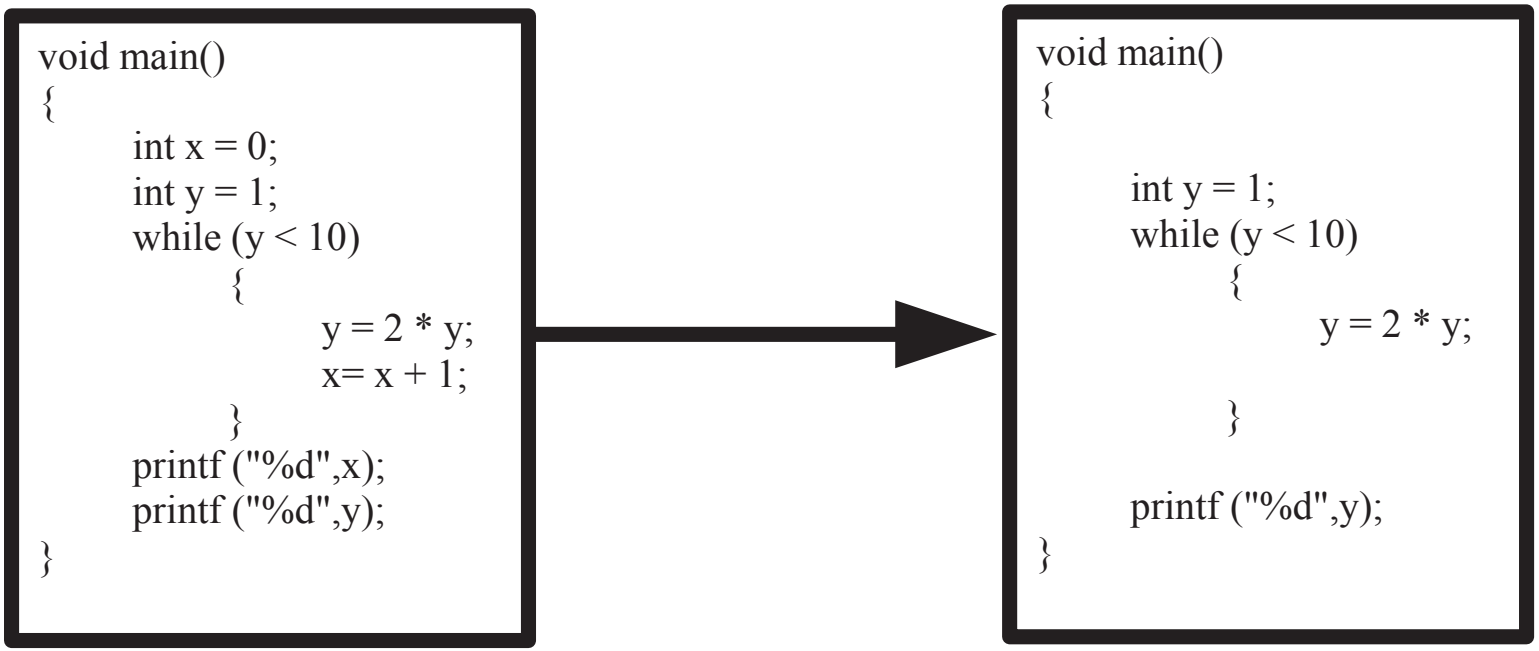
- Transformation et compréhension
  - Calculer certaines métriques sur le code source d'un programme.
    - Exemples :
      - La proportion de code à l'intérieur de boucles dans le programme
      - Le nombre de scénarios d'exécution possibles
      - Le taux de commentaires dans le programme
      - Le couplage des classes/des méthodes
      - La cohésion des méthodes d'une classe/de l'implémentation d'une interface

Pourquoi Analyser ?

- Transformation et compréhension
  - Découpage (sclicing)
    - Technique utilisée dans le but de faire ressortir certaines instructions d'un programme en relation avec une propriété
    - Le Résultat du découpage est un sous ensemble du programme.
    - Technique utile à la réutilisation du code.
      - Exemple : fragmentation du code
        - Extraire le chemin de contrôle d'un résultat partiel d'un programme qui manipule un ensemble de données sans relation avec le reste des données manipulées
    - Technique utile à la compréhension
      - Séparer un programme complexe en parties de code moins compliquées

Pourquoi Analyser ?

- Transformation et compréhension
  - Découpage (sclicing)
    - Exemple



```
void main()
{
        int x = 0;
        int y = 1;
        while (y < 10)
                {
                        y = 2 * y;
                        x= x + 1;
                }
        printf ("%d",x);
        printf ("%d",y);
}
```

```
void main()
{

        int y = 1;
        while (y < 10)
                {
                        y = 2 * y;

                }

        printf ("%d",y);
}
```

Analyse Statique

Représentations des programmes analysés

- Code source (haut niveau)

```java
public class CallingMethodsInSameClass
{
	public static void main(String[] args) {
		printOne();
		printOne();
		printTwo();
	}

	public static void printOne() {
		System.out.println("Hello World");
	}

	public static void printTwo() {
		printOne();
		printOne();
	}
```

JAVA

```c
#include <stdio.h>
void main (void)
{
char prenom[50]="";
printf("Quel est votre prénom ?\n");
scanf("%s",prenom);
printf ("Bonjour %s !\n",prenom);
}
```

C

```cpp
#include <float.h>
void main(void)
{
	char nomm[20];
	int age;

	cout << "Tapez votre nom: ";
	cin >> nomme;			// saisie d'une chaine de caractère

	cout << "Tapez votre age: ";
	cin >> age;  // saisie d'un entier

	cout << "votre nom est: " << name

	char one_char;
	cout << "\nEnterez u caratère: ";
	cin >> one_char;

}
```

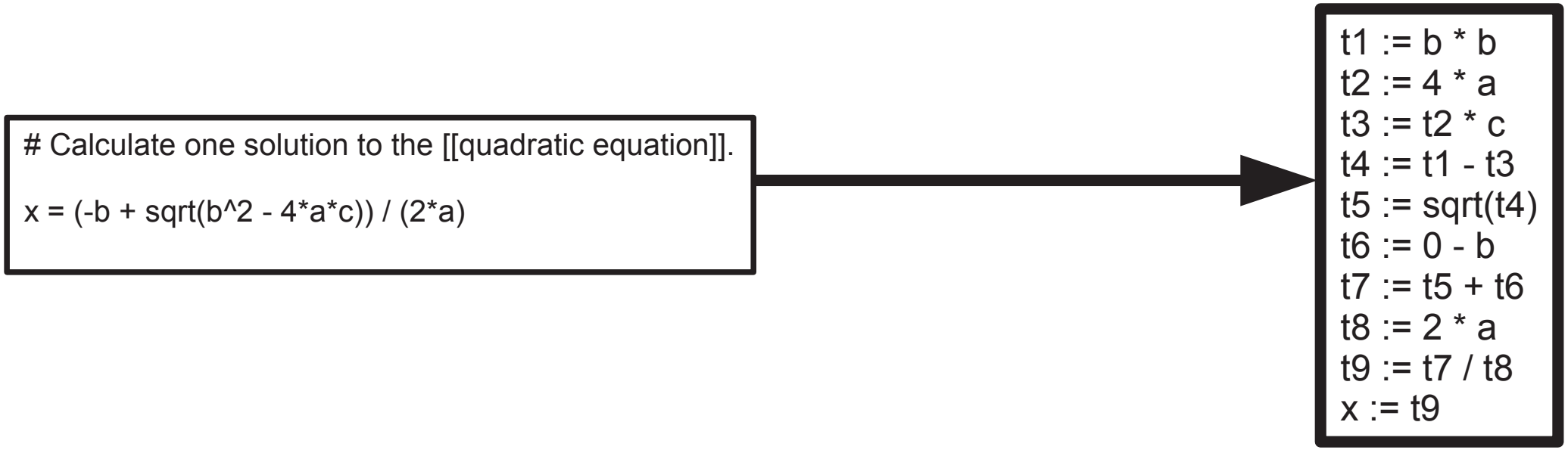
C++

```ada
with Ada.Integer_Text_io;
with Ada.Text_io;
procedure exemple1 is
maNote:Natural;
begin
Ada.Integer_Text_io.get( maNote );
maNote:=maNote+2;
Ada.Text_io.put( "Nouvelle note :" );
Ada.Integer_Text_io.put( maNote );
end exemple1;
```

ADA

Représentations des programmes analysés

- Code 3-adresses (niveau intermédiaire) (TAC ou 3AC)
  - Est un code intermédiaire utilisé pour l'optimisation du code
  - Chaque instruction TAC a au plus trois parties/éléments :
    - Généralement une combinaison de variables et un opérateur binaire
    - Résultat ← <opérande1> <opérateur> <opérande2>



Représentations des programmes analysés

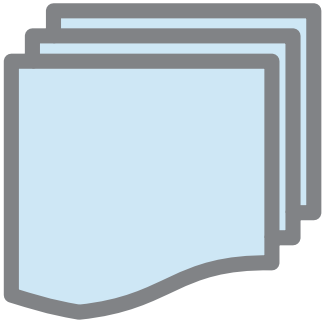
- Code compilé (bas niveau)
  - Bytecode
    - Signifiant en anglais, « code octal », est un code intermédiaire entre les instructions machines et le code source
    - Il n'est pas directement exécutable

```
package org.isk.jvmhardcore.bytecode.parttwo;

public class Adder {
  public static int add (int i1, int i2) {
    return i1 + i2;
  }
}
```

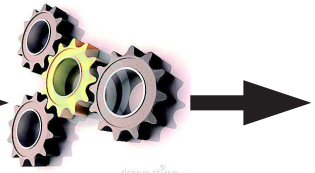
```
.class org/isk/jvmhardcore/bytecode/parttwo/Adder

 .method add(II)I
   iload_0
   iload_
   iadd
   ireturn
 .methodend

.classend
```

Phases d'analyse

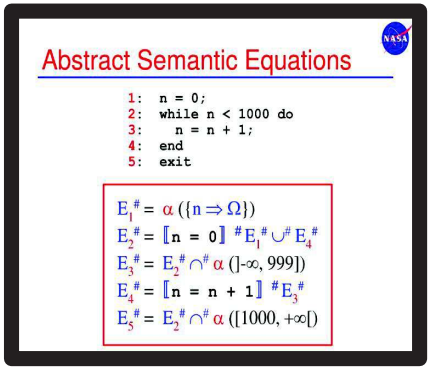
- Deux phases de l'analyse
  1) Extraction d'un modèle de représentation du code source
     - Modèle exacte versus modèle approximatif
     - Représentation du Flot de contrôle versus flot de données versus graphe d'appels, …

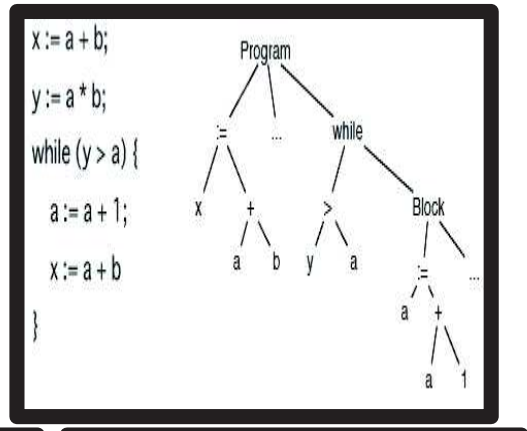


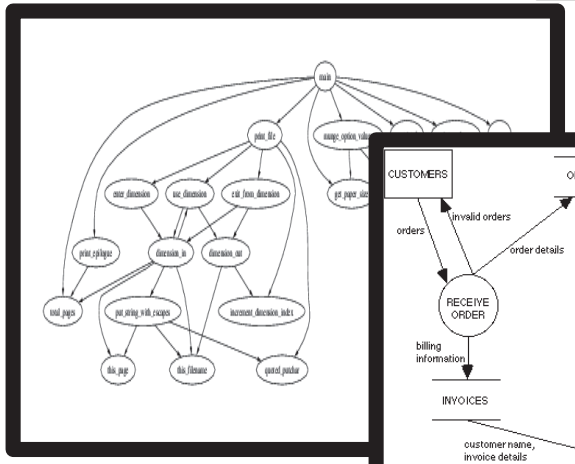
Code source

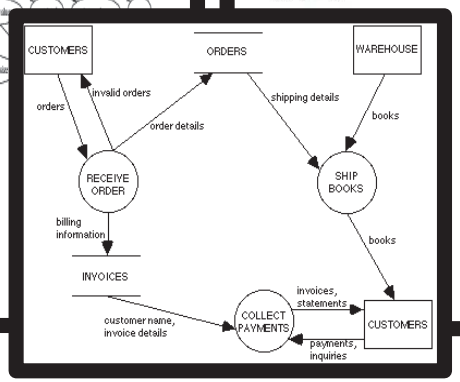


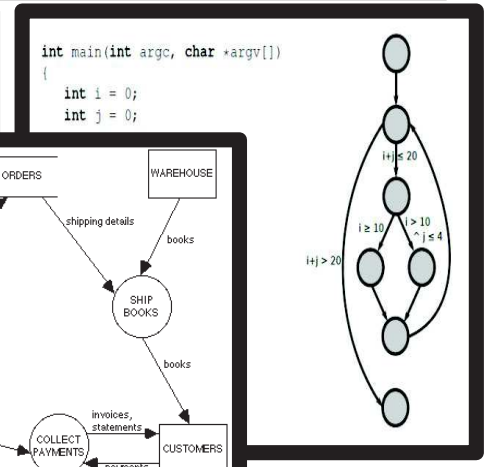
Extraction d'un modèle de représentation du code source





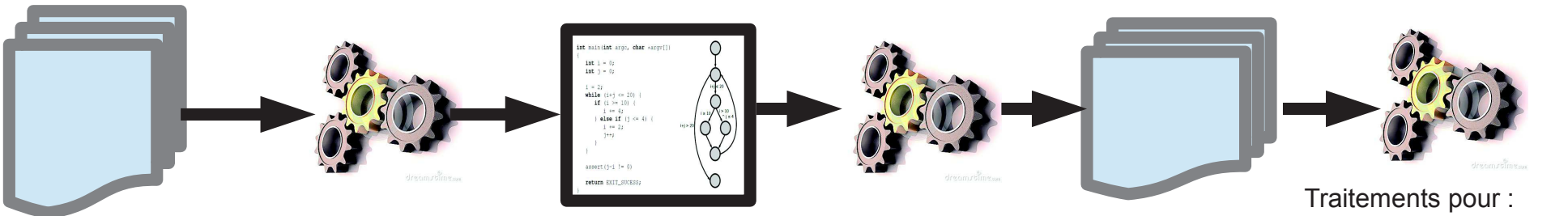






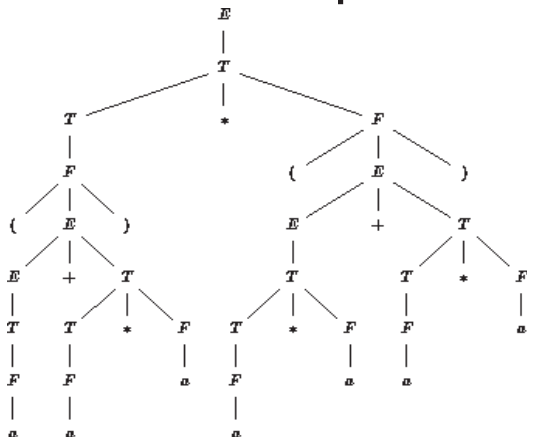
Phases d'analyse

- Deux phases de l'analyse
  2) Analyse (identification) des propriétés basées sur les modèles extraits
     - Exemples
       - Analyses classiques pour la vérification et optimisation
       - Analyses qui concerne la précision du graphe d'appels
       - Analyse de propriétés de performances
       - Calcul de métriques (pour la compréhension, la transformation, …)
       - Etc.



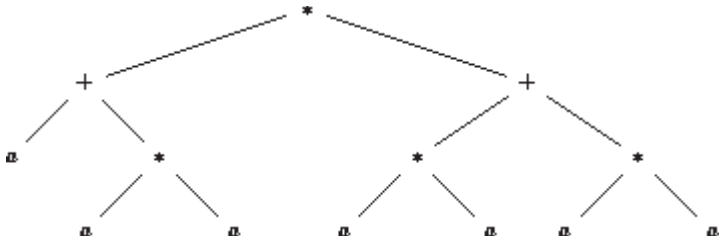
Informations extraites à partir de l'analyse statique

- Arbre syntaxique abstrait (Abstract Syntax Tree, AST)
  - Est un arbre dont les nœuds internes sont des opérateurs et dont les feuilles (ou nœuds externes) représentent les opérandes de ces opérateurs.
    - Généralement, une feuille est une variable ou une constante.
  - Diffère d'un arbre de dérivation par l'omission des nœuds et des branches qui n'affectent pas la sémantique
    - Exemple : Omission des parenthèses



Arbre de dérivation

(a+a*a) * (a*a +a*a)

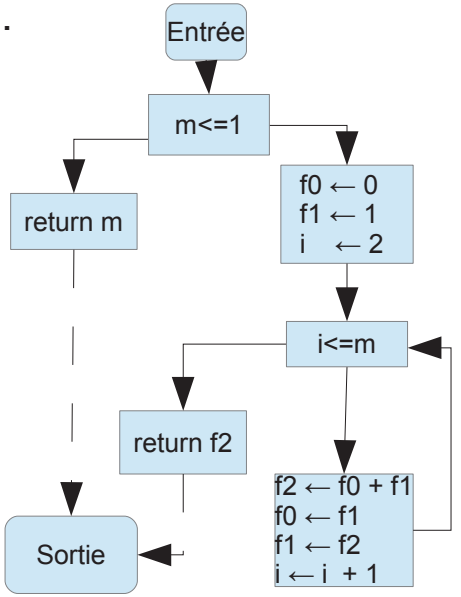


AST

Informations extraites à partir de l'analyse statique

- Graphe flot de contrôle-GFC (Control Flow Graph-CFG)
  - Est une représentation sous forme de graphe de tous les chemins qui peuvent être suivis par un programme durant son exécution.
    - Les sommets du graphe représentent un bloc de base :
      - Bloc de base : un bout de code d'un seul tenant sans sauts ni cibles de sauts.
      - Les cibles de sauts marquent le début d'un bloc de base, tandis que les sauts en marquent la fin.
    - Les arcs représentent les sauts dans le flot de contrôle.
    - La plupart des représentations d'un CFG comprennent deux blocs spéciaux :
      - le bloc d'entrée, par lequel on entre dans le graphe de flot de contrôle
      - le bloc de sortie, par lequel on le quitte.

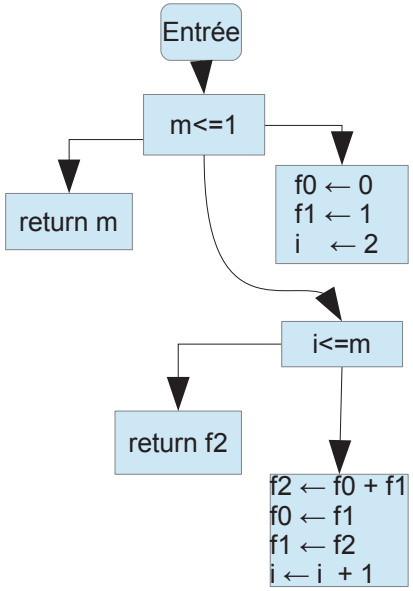
```
int fib(int m) {
        if (m <= 1) {
                return m ;
        } else {
          int f0 = 0, f1 = 1, f2, i ;
          for (i=2 ; i <= m ; i++) {
                f2 = f0 +  f1 ;
                f0 = f1 ;
                f1 = f2 ;
        }
        return f2 ;
        }
}
```



Informations extraites à partir de l'analyse statique

- Le graphe de dépendance de contrôle
  - Montre quelles instructions seront exécutées en fonction de la valeur d'une expression dans le programme.
  - Les noeuds du graphe sont les mêmes que ceux du graphe de flot de contrôle.
    - Pour deux noeuds p et q, un arc va de p vers q si la valeur de l'expression p a un impact sur le fait que l'instruction q soit exécutée ou non.

```
int fib(int m) {
        if (m <= 1) {
                return m ;
        } else {
          int f0 = 0, f1 = 1, f2, i ;
          for (i=2 ; i <= m ; i++) {
                f2 = f0 +  f1 ;
                f0 = f1 ;
                f1 = f2 ;
        }
        return f2 ;
        }
}
```



Informations extraites à partir de l'analyse statique

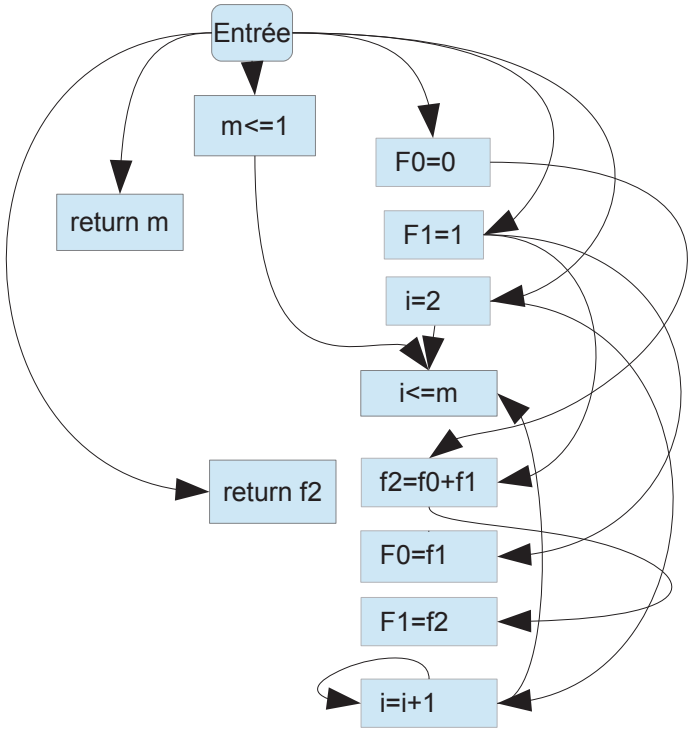
- Flot de données
  - Avoir de l'information sur l'utilisation des variables dans le temps
  - Exemple,
    - L'ensemble des variables utilisées et celui des variables modifiées pour chaque instruction du programme.

```
int fib(int m) {
        if (m <= 1) {                          Utilise= {m} ; Définit = {}
              return m ;                       Utilise= {m} ; Définit = {}
        } else {
          int f0 = 0, f1 = 1, f2, i ;          Utilise= {} ; Définit = {f0, f1}
          for (i=2 ; i <= m ; i++) {           Utilise= {m, i} ; Définit = {i}
              f2 = f0 +  f1 ;                  Utilise= {f0, f1} ; Définit = {f2}
              f0 = f1 ;                        Utilise= {f1} ; Définit = {f0}
              f1 = f2 ;                        Utilise= {f2} ; Définit = {f1}
        }
        return f2 ;                            Utilise= {f2} ; Définit = {}
        }
}
```

Informations extraites à partir de l'analyse statique

- Le graphe de dépendance de données
  - Est un graphe dont les nœuds sont les mêmes que celui du graphe de flot de contrôle
  - Dans ce graphe, un arc va de p vers q s'il est possible que la valeur d'une des variables modifiées à l'instruction p soit utilisée à l'instruction q sans qu'elle ne soit modifiée entre temps.

```
int fib(int m) {
        if (m <= 1) {
                return m ;
        } else {
          int f0 = 0, f1 = 1, f2, i ;
          for (i=2 ; i <= m ; i++) {
                f2 = f0 +  f1 ;
                f0 = f1 ;
                f1 = f2 ;
        }
        return f2 ;
        }
}
```



Interprétation abstraite

- **Définition :**
  - Est une théorie d'approximation de la sémantique de programmes.
- **Objectif :**
  - Réaliser une exécution partielle d'un programme pour obtenir des informations sur sa sémantique, par exemple sur sa structure de contrôle ou sur son flot de données, sans avoir à en faire le traitement complet.
- **Comment :**
  - Un programme dénote une série de traitements dans un univers donné de valeurs ou d'objets.
  - L'interprétation abstraite consiste à utiliser cette dénotation pour décrire (réécrire) ces traitements dans un autre univers d'objets abstraits.
  - Les résultats de l'interprétation abstraite fournissent des informations sur les calculs concrets du programme.

Interprétation abstraite

- Exemple

- Prouver des propriétés de sûreté (débordements arithmétiques, divisions par 0, ..)
  - État indésirable = Point de programme + Valeurs des variables
  - Idée : Déterminer toutes les valeurs prises par les variables
  - Problème : Trop complexe (théorie)
  - Solution : Approximation (Abstraction)

Calculer une intervalle de valeurs possibles pour chaque variable

```
// x ∈ [2, 5]
y = 2 * x;
// y ∈ [4, 10]
if (y <= 4) {
    z = y - 2;
    // z ∈ [2, 2]
    } else {
      z = y + 2;
      // z ∈ [7, 12]
}
// z ∈ [2, 12]
```

Étendue de l'analyse statique

- Locale (« peephole ») :
  - L'analyse porte sur quelques instructions
- Intraprocédurale
  - L'analyse porte sur le code d'une seule méthode
- Interprocédurale
  - L'analyse porte sur un programme complet ou un fragment de programme

Analyses Locales

- Requiert uniquement la séquence des instructions du programme
  - Utiliser par exemple pour la transformation du code pour l'optimisation
    - Constant folding : Evaluation et remplacement des sous-expressions constantes

      - I = 320 * 200 * 32 → i = 2048000
      - "abc" "def" → "abcdef".

    - Strength reduction : Remplacer les opérations lentes par des opérations plus rapides
      - Le remplacement d'une multiplication dans une boucle avec une addition

```
c = 7;
for (i = 0; i < N; i++)
{
    y[i] = c * i;
}
```

→

```
c = 7;
k = 0;
for (i = 0; i < N; i++)
{
    y[i] = k;
    k = k + c;
}
```

Analyses Locales

- Requiert uniquement la séquence des instructions du programmes
  - Utiliser par exemple pour l'optimisation du code par transformation du code.
    - Constant folding – évaluation et remplacement des sous-expressions constante
    - Strength reduction
    - Null sequences : Supprimer les opérations non utilisées
    - Combine Operations : remplacer plusieurs opération par une équivalente
    - Algebraic Laws : utiliser les lois algébriques pour simplifier ou réordonner les instructions
    - Special Case Instructions : Utilisation d'instructions conçues pour des cas particuliers d'opérandes.
    - Etc

Analyses intra-procédurales

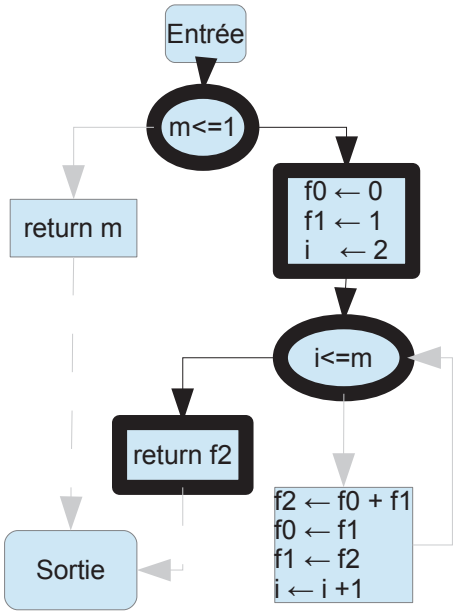
- Requiert un graphe de flot de contrôle pour la méthode/fonction analysée

- Exemples
  - Présence de valeurs de retour sur tous les chemins
  - Code non-atteignable
  - Cohérence des types
  - Débordement de la pile

Analyses intra-procédurales

- Exemple
  - Code non-atteignable
    - Si un bloc ou une portion du graphe de contrôle n'est pas connecté au bloc d'entrée, ce bloc ne peut jamais être atteint durant l'exécution, et il s'agit de code mort qui peut être supprimé

```
int fib(int m) {
      if (m <= 1) {
            return m ;
      } else {
        int f0 = 0, f1 = 1, f2, i ;
        for (i=2 ; i <= m ; i++) {
            f2 = f0 +  f1 ;
            f0 = f1 ;
            f1 = f2 ;
      }
      return f2 ;
      }
}
```



M>1

i=2

i>m

m>1 & i=2 & i>m
Equiv. m>1 & m<2

N'admet aucune solution entière, donc cette exécution est impossible

Analyse interprocédurales

- Requiert un graphe d'appel
- Exemples
  - Analyse de pointeurs/références
  - Analyse d'échappement (escape analysis)
  - Analyse de tainte (taint analysis)

Analyse interprocédurales

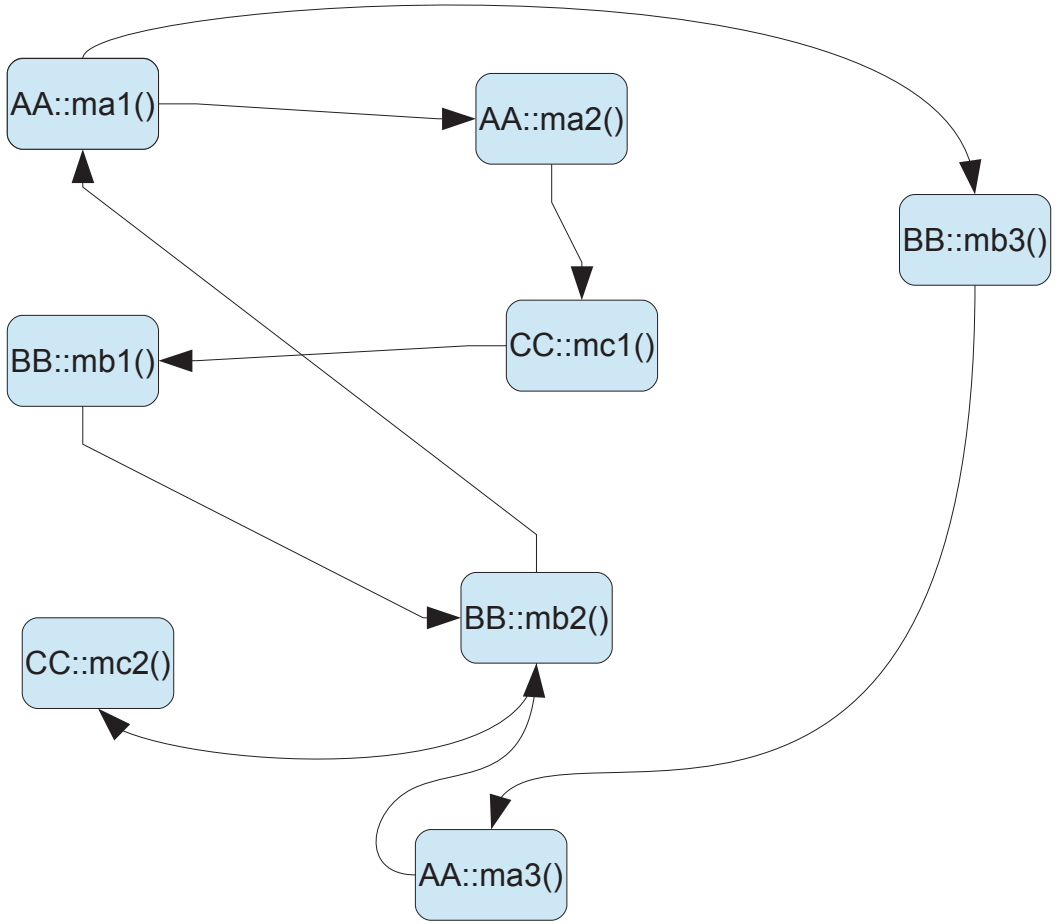
- Un graphe d'appel est un graphe orienté qui représente les relations d'appel entre les procédures/méthodes d'un programme.
  - Chaque nœud représente une procédure
  - Chaque arc qui va du nœud f au nœud g indique que la procédure f appelle la procédure g.
  - Un cycle dans un graphe d'appel indique des appels récursifs.

- Un graphe d'appel peut être statique ou dynamique
  - Un graphe d'appel dynamique est une représentation des appels enregistrés durant certaines exécutions du programme.
  - Un graphe d'appel statique représente tous les appels possibles durant d'exécution d'un programme.

Analyse interprocédurales

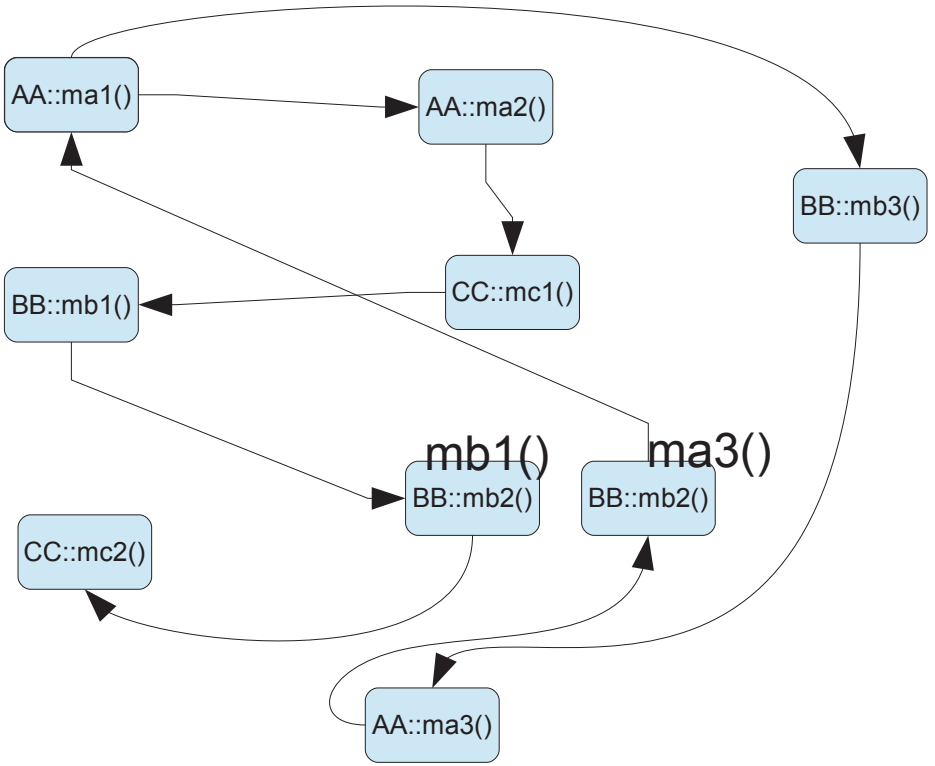
Graphe d'appels

```
class AA {
      public BB atta1;
      public CC atta2;
      public int atta3;

      public void ma1(){
            ma2();
            if (atta3> 5) {atta1 = new BB();}
            else atta1.mb3(); }

      public void ma2(){atta2.mc1();}
      public void ma3(){atta1.mb2();}
}

class BB {
      public AA attb1;
      public CC attb2;

      public void mb1(){mb2(); }
      public void mb2(){
            attb2.mc2();
            attb1.ma1(); }
public void mb3(){attb1.ma3();}
}

class CC{
       public void mc1(){
            BB b = new BB();
            b.mb1(); }

       public void mc2(){
                  System.out.println(''fin'') ;}
}
```



Analyse interprocédurales

- Graphe d'appels contextuel
  - En général, un nœud du graphe d'appel représente toutes les invocations possibles d'une même méthode
  - Il est possible de représenter la même méthode par plusieurs nœuds pour conserver plus de contexte.



Analyse des références

- L'analyse de références (reference analysis) permet de déterminer, pour chaque variable (ou attribut) dans le programme, un ensemble d'objets auxquels elle peut pointer à l'exécution.

- Cas d'analyse de références
  - Appel statique
    - Cible connue à la compilation
    - Pas de receveur
  - Appel spécial (méthodes privées, super)
    - Cible connue à la compilation
    - Receveur explicite
  - Virtuel/interface
    - Cible dépend du type dynamique du receveur

Analyse des références

- Résolution des appels

```
public class A {

 public static void sm() {System.out.println("je suis la méthode statique 'sm' de A ");}

 //public  void sm() {System.out.println("je suis la méthode statique 'sm' de A ");}


 private void m() { System.out.println("je suis la méthode 'm' de A ");}


 public static void main (String[] args){
             A ref = new A();

              ref.m();

             ref = new B();

             ref.m();

             ref.sm();

       }
}

class B extends A {
       public void m() { System.out.println("je suis la méthode 'm' de B ");}

        public static void sm() {System.out.println("je suis la méthode statique 'sm' de B ");}
}
```

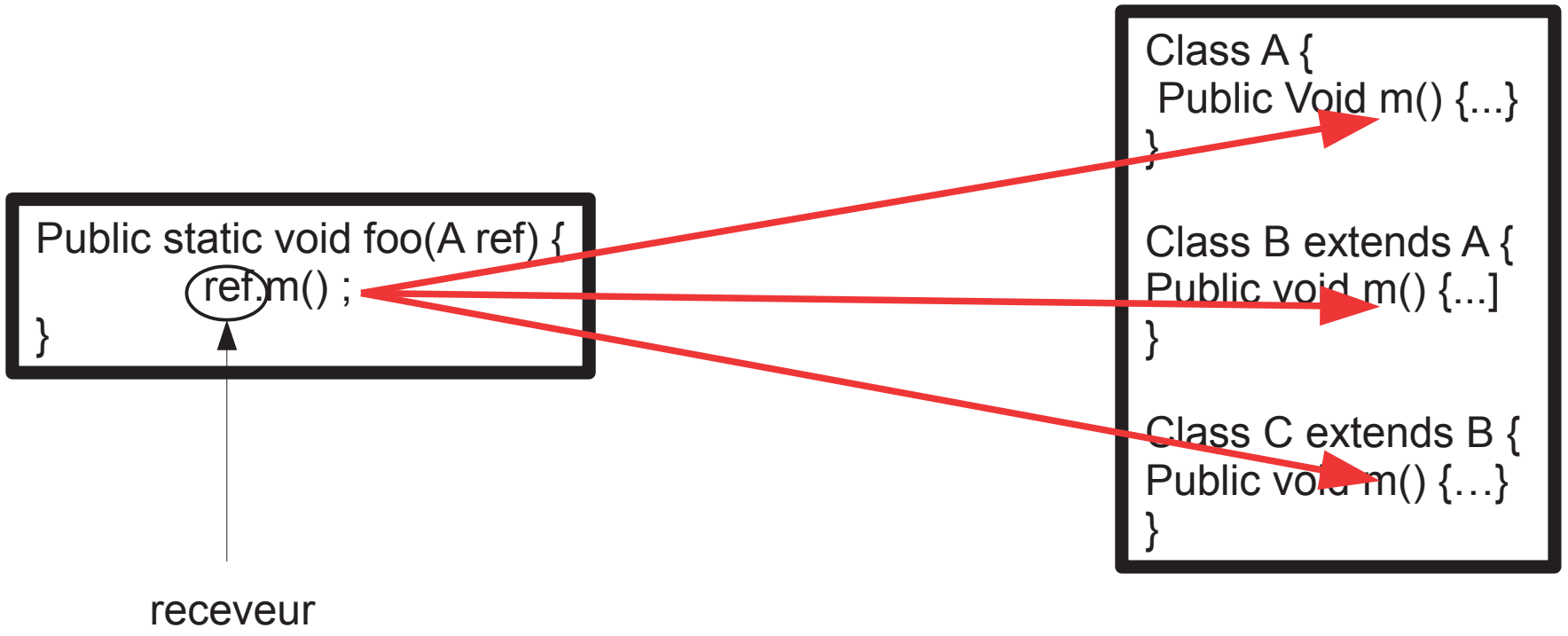
Analyse des références

- Résolution des appels

```
public class A {

 public static void sm() {System.out.println("je suis la méthode statique 'sm' de A ");}

 //public  void sm() {System.out.println("je suis la méthode statique 'sm' de A ");}


 private void m() { System.out.println("je suis la méthode 'm' de A ");}


 public static void main (String[] args){
             A ref = new A();

             //je suis la méthode 'm' de A
             ref.m();

             ref = new B();

             //je suis la méthode 'm' de A
             ref.m();

            //warning The static method sm() from the type A should be accessed in a static way
            //je suis la méthode statique 'sm' de A
             ref.sm();

       }
}

class B extends A {
       public void m() { System.out.println("je suis la méthode 'm' de B ");}

        public static void sm() {System.out.println("je suis la méthode statique 'sm' de B ");}
}
```

Analyse des références

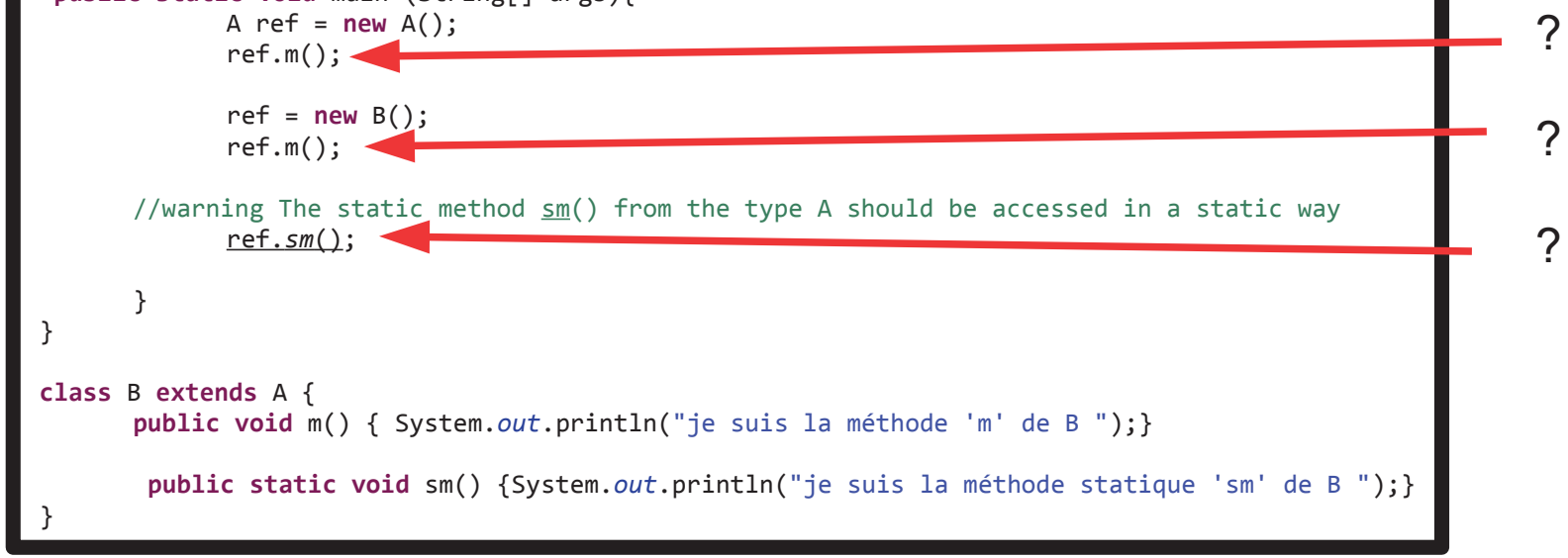
- Résolution des appels
  - Appels virtuels
    - Dans les langages OO, la cible d'un appel virtuel dépend du type de l'objet receveur à l'exécution



Analyse des références

- Appels virtuels

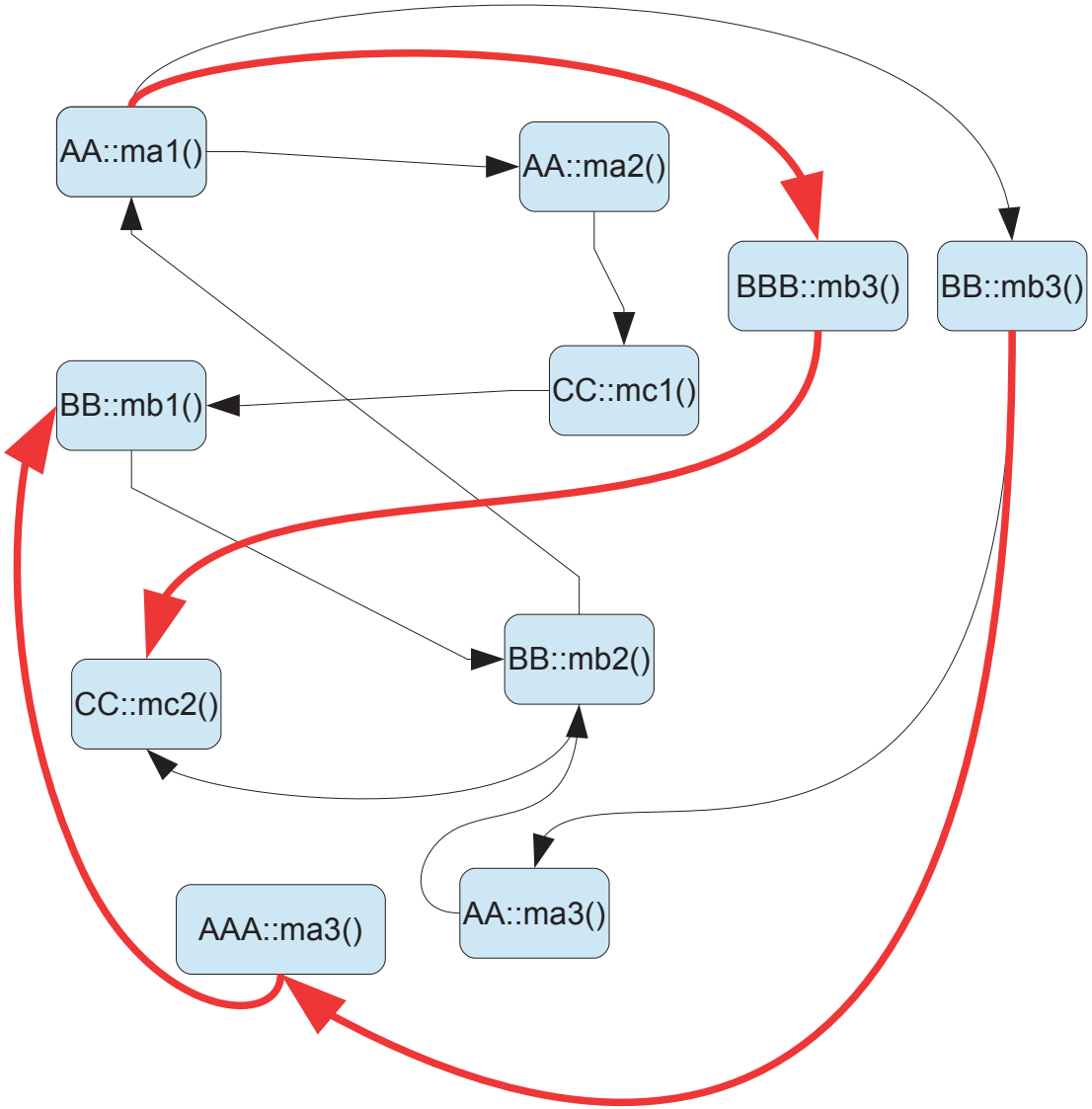
```
public class A {

 public static void sm() {System.out.println("je suis la méthode statique 'sm' de A ");}

 //public  void sm() {System.out.println("je suis la méthode statique 'sm' de A ");}


 private void m() { System.out.println("je suis la méthode 'm' de A ");}


 public static void main (String[] args){
             A ref = new A();
             ref.m();

             ref = new B();
             ref.m();

       //warning The static method sm() from the type A should be accessed in a static way
             ref.sm();

       }
}

class B extends A {
       public void m() { System.out.println("je suis la méthode 'm' de B ");}

        public static void sm() {System.out.println("je suis la méthode statique 'sm' de B ");}
}
```



Analyse des références

- Résolution des appels virtuels

```
class AA {
      public BB atta1;
      public CC atta2;
      public int atta3;

      public void ma1(){
            ma2();
            if (atta3> 5) {atta1 = new BB();}
            else atta1.mb3(); }

      public void ma2(){atta2.mc1();}
      public void ma3(){atta1.mb2();}
}

class AAA extends AA {

      public void ma3(){atta1.mb1();}
}
class BB {
      public AA attb1;
      public CC attb2;

      public void mb1(){mb2(); }
      public void mb2(){
            attb2.mc2();
            attb1.ma1(); }
      public void mb3(){attb1.ma3();}
}

class BBB extends BB {

      public void mb3(){attb2.mc2();}
}
```



Analyse des références

- Résolution des appels virtuels
  - Beaucoup de techniques d'analyse statique ont été proposées pour établir une analyse des références
    - Différentes par rapport aux
      - Algorithmes utilisés
      - Présentations des programmes utilisées
      - La précision qui peut être obtenue
      - Le coût engendré

Analyse des références

- Résolution des appels virtuels
  - Types des techniques d'analyse de références
    - Analyses des classes (class analyses) :
      - Utilisent un objet abstrait par classe (avec ou sans attribut).
    - Analyses Points-to :
      - Groupent les instances selon un mécanisme donné (ex : site de création)
    - Les analyses peuvent utiliser autres informations
      - Informations qui concernent le flot de données/contrôle : Flow sensitivity
      - Informations qui concerne le conetxte d'appel : Context sensitivity

Analyse de la hiérarchie d'héritage (Class hierarchy analysis - CHA)

- Première analyse de référence proposée

  - J. Dean, D. Grove, C. Chambers, Optimization of OO Programs Using Static Class Hierarchy, ECOOP'95

- Concept clé :
  - Inspecte la hiérarchie de classes pour déterminer quelles sont les classes qui peuvent être concernées par une référence déclarée de type A
    - Sous-arbre d'héritage dont la racine est A
  - Identifie les méthodes qui peuvent être appelées sur un site virtuel
  - Fait l'hypothèse que le programme entier est atteignable
  - Ignore le flot de contrôle
  - Utilise un objet abstrait par classe et un attribut abstrait par classe

Analyse de la hiérarchie d'héritage (Class hierarchy analysis - CHA)

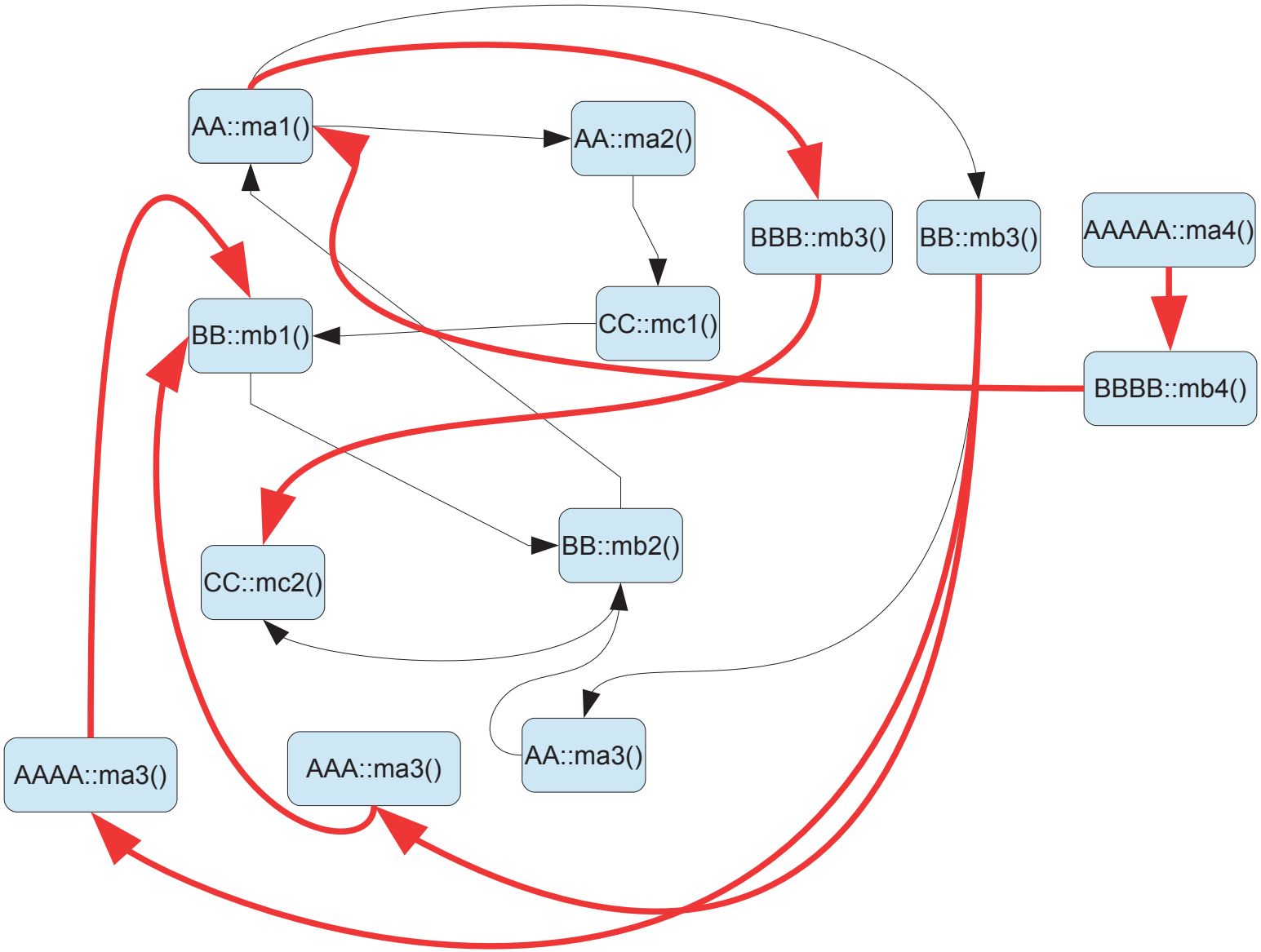
- Résolution des appels virtuels

```
class AA {
      public BB atta1;
      public CC atta2;
      public int atta3;

      public void ma1(){
            ma2();
            if (atta3> 5) {atta1 = new BB();}
            else atta1.mb3(); }

      public void ma2(){atta2.mc1();}
      public void ma3(){atta1.mb2();}
}

class AAA extends AA {

      public void ma3(){atta1.mb1();}
}

class AAAA extends AA {

      public void ma3(){atta1.mb1();}
}

class AAAAA extends AA {

      public void ma4(){atta3 = 1 ;
                        (new BBB()).mb4();
      }
}
```

```
class BB {
      public AA attb1;
      public CC attb2;

      public void mb1(){mb2(); }
      public void mb2(){
            attb2.mc2();
            attb1.ma1(); }
      public void mb3(){attb1 =new AAA() ;
                        attb1.ma3();}
}

class BBB extends BB {

      public void mb3(){attb2.mc2();}
}

class BBBB extends BB {

      public void mb4(){attb1.ma1();}
}
```

Analyse de la hiérarchie d'héritage (Class hierarchy analysis - CHA)

- Résolution des appels virtuels



Rapid type analysis (RTA)

- Est une amélioration de CHA

- D. Bacon and P. Sweeney, " Fast Static Analysis of C Virtual Function Calls", OOPSLA'96

- Concept clé :
  - Ignorer les classes pour lesquelles aucun objet n'est créé
  - Une cible ne sera considérée possible que si un objet du type approprié a été préalablement créé dans une méthode atteignable
  - Construit le graphe d'appel à mesure que l'analyse avance (On-the-fly)
  - Ignore le flot de contrôle
  - Utilise un objet abstrait par classe et un attribut abstrait par classe

Rapid type analysis (RTA)

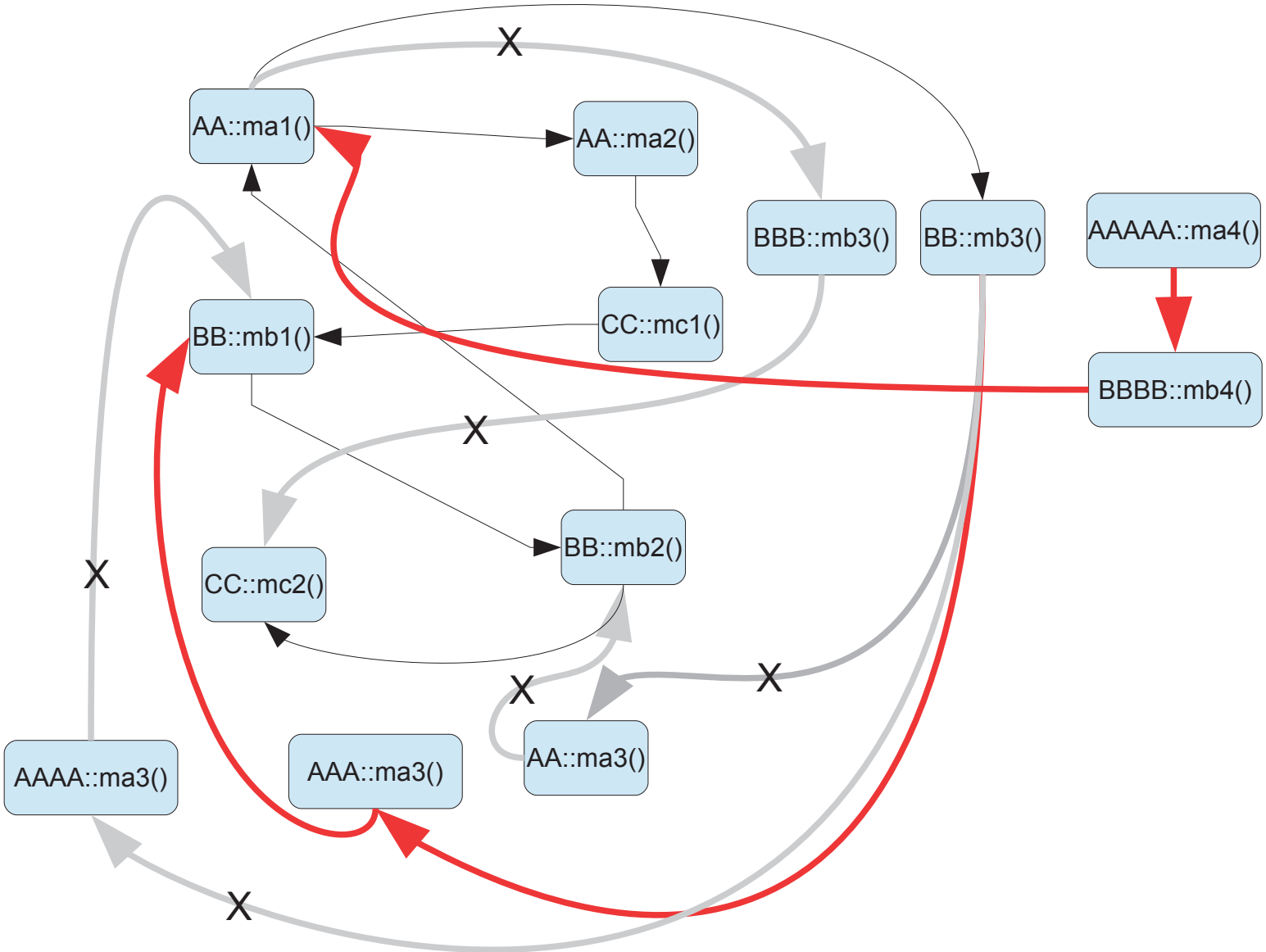
- Résolution des appels virtuels

```
class AA {
      public BB atta1;
      public CC atta2;
      public int atta3;

      public void ma1(){
            ma2();
            if (atta3> 5) {atta1 = new BB();}
            else atta1.mb3(); }

      public void ma2(){atta2.mc1();}
      public void ma3(){atta1.mb2();}
}

class AAA extends AA {

      public void ma3(){atta1.mb1();}
}

class AAAA extends AA {

      public void ma3(){atta1.mb1();}
}

class AAAAA extends AA {

      public void ma4(){atta3 = 1 ;
                        (new BBB()).mb4();
      }
}
```

```
class BB {
      public AA attb1;
      public CC attb2;

      public void mb1(){mb2(); }
      public void mb2(){
            attb2.mc2();
            attb1.ma1(); }
      public void mb3(){attb1.ma3();}
}

class BBB extends BB {

      public void mb3(){attb2.mc2();}
}

class BBBB extends BB {

      public void mb4(){attb1.ma1();}
}
```

Rapid type analysis (RTA)

- Exemple



Analyse des références basée sur l'analyse des classes

- Comparaison
  - Bacon-Sweeney, OOPSLA'96

```
class A {
public :
    virtual int foo(){ return 1; };
};

class B: public A {
public :
    virtual int foo(){ return 2; };
    virtual int foo(int i) { return i
1; };
};

void main() {
    B* p = new B;
    int result1 = p->foo(1);
    int result2 = p->foo( ) ;
    A* q = p;
    int result3 = q->foo( );
}
```

- CHA résout le cas de l'appel result2 à B.foo() parce que B n'a pas de sous classes

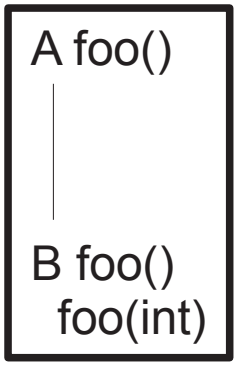
- CHA ne résout pas le cas de l'appel de result3.

- RTA résout le cas de l'appel result3 parce que B a été instanciée

Analyse des références basée sur l'analyse des classes

- Limitations liées à la sûreté des types
  - CHA et RTA fonctionnent dans un contexte où la sûreté des types est respectée

```
//#1
void* x = (void*) new B
B* q = (B*) x;              //a safe downcast
int case1 = q->foo()

//#2
void* x = (void*) new A
B* q = (B*) x;              //an unsafe downcast
int case2 = q->foo()  //probably no error

//#3
void* x = (void*) new A
B* q = (B*) x;                  //an unsafe downcast
int case3 = q->foo(666)  //runtime error
```



Les deux analyses ne distinguent pas ces trois cas

XTA

- Proposée par Tip & Parlsberg (OOPSLA'00)
- Amélioration de RTA
- Concept clé : utiliser un ensemble de types par méthodes et par variable plutôt qu'un ensemble global
  - Comme RTA, construit le graphe d'appels à mesure que l'analyse avance (on-the-fly)
- Filtre les types passés en paramètres / valeurs de retour en fonction des types déclarés
- Ignore le flot de contrôle
- Utiliser un objet abstrait par classe et attribut abstrait par classe dans chaque méthode

Tip and Palsberg, "Scalable Propagation-based Call Graph Construction Algorithms", OOPSLA'00

Analyse dynamique

Objectif

- Compréhension de programmes
- Détection de phases d'exécution
- Analyse de l'utilisation des ressources
  - Analyse de performance
  - Analyse de l'utilisation de la mémoire
- Optimisation JIT
- Débogage
- Analyse de tainte (sécurité)
- Analyse de couverture
- Détection d'invariants probables
- Etc.

Quelles préoccupations ?

1) Comment recueillir l'information ?
   - Transformations du code source / compilé (instrumentation)
   - Modifications de l'environnement d'exécution
   - Utilisation d'une interface de profilage

2) Comment éviter (minimiser) le surcoût ?

3) Comment s'assurer que les résultats ne sont pas affectés par les observations ?
   - ex: changements au comportement du système

4) Comment choisir les entrées / exécutions représentatives ?
   - Comment s'assurer de la validité des résultats?

Comment recueillir l'information ?

- Instrumentation
  - L'instrumentation consiste à ajouter des fragments de code (sondes) à un programme de façon à ajouter la génération des résultats d'analyse à son comportement initial
    - Le programme s'exécute dans son environnement normal
  - Au cours de l'exécution, les sondes sont exécutées en plus du programme d'origine et recueillent l'information nécessaire

Comment recueillir l'information ?


```
public static int[] append (int i, int[] a) {
Int[] r ;
Analysis.recordEntry("append") ;

        if (a != null) {
                Analysis.recordAlloc("int[]", a.length   1) ;

                r = new int [a.length  1] ;
                 Analysis.recordCall("System.arraycopy") ;
                System.arraycopy(a, 0, r, 1, a.length) ;
        } ele {
                Analysis.recordAlloc("int[]", 1) ;
                r= new int[1] ;
                }
        r[0] = i ;

        Analysis.recordExit("append") ;
        return r ;


}
```

Comment recueillir l'information ?

- Instrumentation
  - Types d'instrumentation
    - Code source (source-to-source)
      - transformations du code source (haut niveau)
    - Code compilé (ex: bytecode)
  - Stratégie
    - Hors-ligne : Instrumenter avant de débuter l'exécution
      - génère une nouvelle version du programme instrumenté
        - ex : AspectJ, Soot, etc.
    - En-ligne : Instrumenter durant l'exécution
      - Le code est transformé à mesure qu'il est chargé par l'environnement d'exécution
        - ex: valgrind, Steamloom, Dyko, etc.

Comment recueillir l'information ?

- Instrumentation en-ligne
  - Comment intercepter le chargement du code ?
  - En Java, plusieurs stratégies :
    - JVMTI : permet d'intercepter le chargement de toutes les classes (sauf les tableaux)
    - java.lang.Instrument : permet d'intercepter le chargement de certaines classes (au moins toutes les classes de l'application)
    - ClassLoader personnalisé : permet de charger des classes dans un ClassLoader spécifique et de les modifier avant le chargement
    - Modification de la JVM
    - Modifications du code d'entrée/sortie (I/O)

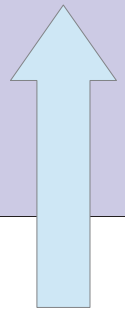
Comment recueillir l'information ?

- Pièges de l'instrumentation
  - Le code inséré par instrumentation ne doit pas appeler du code instrumenté sans une garde
    - Possibilité de causer de la récursion sans borne
      - Attention aux modifications dans java.lang.Object!

Comment recueillir l'information ?

- Pièges de l'instrumentation
  - Exemple : Récursion infinie

```
public class Object {
        public Object() {
                Analysis.recordEntry("Object.Object()") ;
        }
}

public class Analysis {
        public static void recordEntry(String method) {
                Node n = new Node (method) ;
                ...
        }
}
```



Appel récursif croisé infini : Object(), Node(), Object(), ...

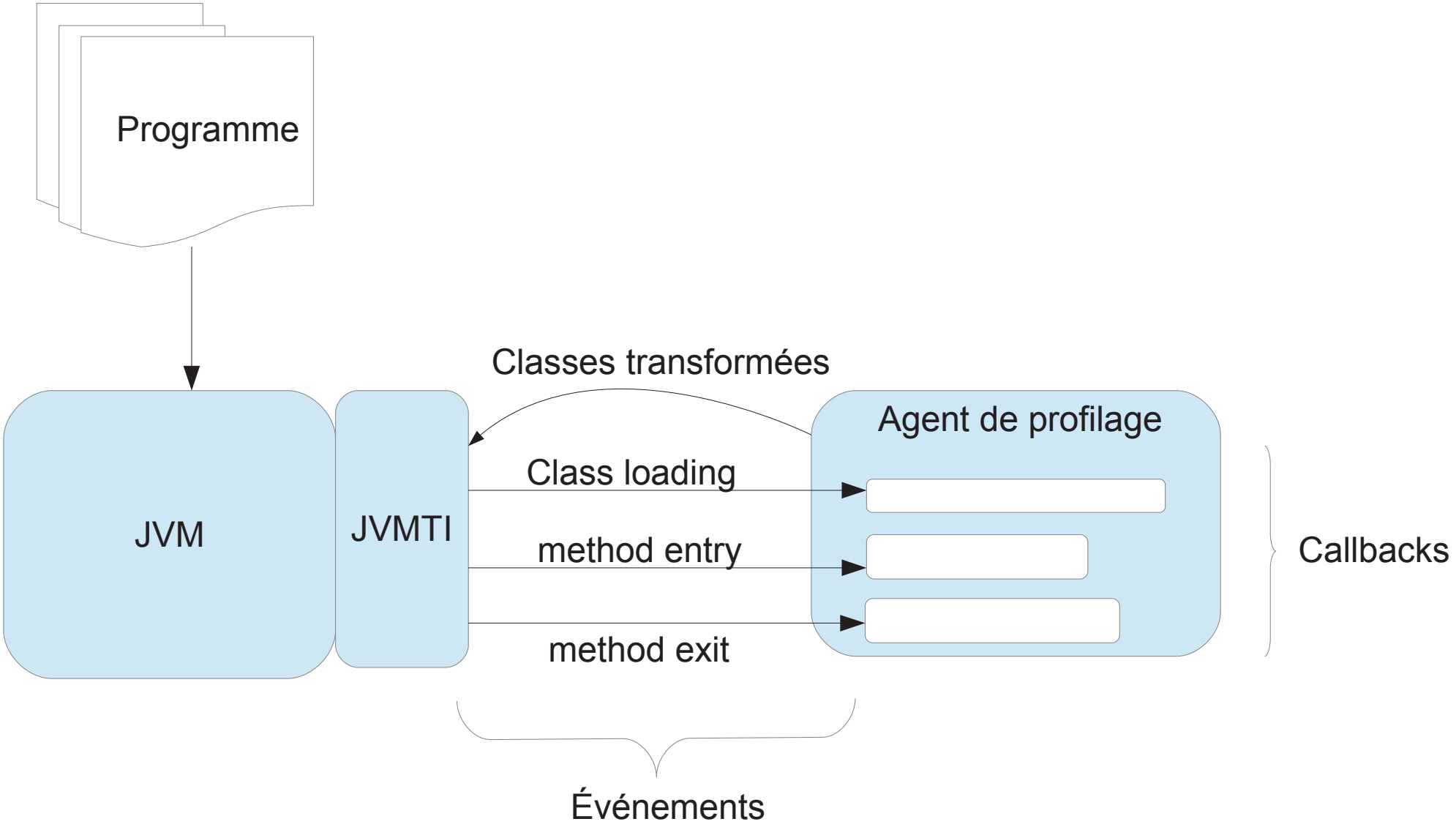
Comment recueillir l'information ?

- Pièges de l'instrumentation
  - Certaines classes sont difficiles à instrumenter
    - Java : Object, String, tableaux, etc.
  - Toutes les classes utilisées ne sont pas forcément disponibles à l'avance
    - Java : Proxy génère des classes à l'exécution
  - Certaines méthodes (ex: native) ne peuvent pas être instrumentées

Comment recueillir l'information ?

- Interface de profilage
  - Fournit un mécanisme d'inspection du programme en cours d'exécution
  - Permet de « se brancher » à une machine virtuelle pour intercepter des événements ou contrôler l'exécution
  - Exemples :
    - Java Virtual Machine Tool Interface (JVMTI)
    - Valgrind
    - Émulateurs
    - Program counters (CPU)
    - etc.

Comment recueillir l'information ?



Comment recueillir l'information ?

- Environnement d'exécution modifiés
  - Pour les langages interprétés, il est souvent possible de modifier un interprète existant afin de collecter l'information désirée
  - Exemples :
    - JikesRVM (aka Jalapeno VM) - JVM de recherche écrite en Java
      - aussi SableVM, Maxine, …
    - Modifications d'Interprète commerciaux (e.g. V8 pour JS, ...)

Comment recueillir l'information ?

	Interfaces	Modifications de l'environnement	Instrumentation
Information disponible	Limitée par l'interface	Complète	Accessible à l'application
Difficulté	Facile	Complexe	Moyenne
Impact	Moyen à élévé	Minimal	Minimal à moyen

Perturbations liées à l'exécution

- L'analyse dynamique peut perturber l'exécution de façon
  - Directe : Le comportement de code d'analyse peut directement affectée l'information recueillie
    - Ex : le temps d'exécution mesuré inclut le temps passé à exécuter les sondes
  - Indirecte : le code d'analyse affecte indirectement le comportement mesuré/observé du système
    - Ex : invalidation de la mémoire cache cause un ralentissement, ordonnancement différent des fils d'exécution (threads)

Types d'analyse dynamique

- En ligne (Online)
  - Les résultats sont calculés au cours de l'exécution
    - Des calculs complexes peuvent perturber l'exécution
    - Seulement l'information pertinente est enregistrée
- Hors ligne (Offline)
  - Les résultats sont calculés après l’exécution (post-mortem) à l'aide de traces d'exécution
  - L'impact du calcul sur l'exécution est diminue
  - La quantité d'information à enregistrer peut être énorme (et proportionnelle au temps d'exécution)

Outils

- Analyse statique de code java [source Wiki pédia : https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis]

  - Checkstyle – Besides some static code analysis, it can be used to show violations of a configured coding standard.
  - FindBugs – An open-source static bytecode analyzer for Java (based on Jakarta BCEL) from the University of Maryland.
  - IntelliJ IDEA – Cross-platform Java IDE with own set of several hundred code inspections available for analyzing code on-the-fly in the editor and bulk analysis of the whole project.
  - JArchitect – Simplifies managing a complex Java code base by analyzing and visualizing code dependencies, by defining design rules, by doing impact analysis, and by comparing different versions of the code.
  - Jtest – Testing and static code analysis product by Parasoft.
  - LDRA Testbed – A software analysis and testing tool suite for Java.
  - PMD – A static ruleset based Java source code analyzer that identifies potential problems.
  - SemmleCode – Object oriented code queries for static program analysis.
  - Sonargraph (formerly SonarJ) – Monitors conformance of code to intended architecture, also computes a wide range of software metrics.
  - Soot – A language manipulation and optimization framework consisting of intermediate languages for Java.
  - Squale – A platform to manage software quality (also available for other languages, using commercial analysis tools though).
  - SonarQube – is an open source platform for Continuous Inspection of code quality.
  - SourceMeter - A platform-independent, command-line static source code analyzer for Java, C/C++, RPG IV (AS/400) and Python.
  - ThreadSafe – A static analysis tool for Java focused on finding concurrency bugs.

Outils

- Analyse statique de code C++/C
  - Astrée – finds all potential runtime errors by abstract interpretation, can prove the absence of runtime errors and can prove functional assertions; tailored towards safety-critical C code (e.g. avionics).
  - BLAST – (Berkeley Lazy Abstraction Software verification Tool) – An open-source software model checker for C programs based on lazy abstraction (follow-on project is CPAchecker.[5]).
  - Cppcheck – Open-source tool that checks for several types of errors, including use of STL.
  - cpplint – An open-source tool that checks for compliance with Google's style guide for C++ coding.
  - Clang – An open-source compiler that includes a static analyzer (Clang Static Analyzer).
  - Coccinelle – An open-source source code pattern matching and transformation.
  - Cppdepend – Simplifies managing a complex C/C++ code base by analyzing and visualizing code dependencies, by defining design rules, by doing impact analysis, and comparing different versions of the code.
  - ECLAIR – A platform for the automatic analysis, verification, testing and transformation of C and C++ programs.
  - Eclipse (software) – An open-source IDE that includes a static code analyzer (CODAN).
  - Fluctuat – Abstract interpreter for the validation of numerical properties of programs.
  - Frama-C – An open-source static analysis framework for C.
  - Goanna – A software analysis tool for C/C++.
  - Klocwork Static Code Analysis – A static analysis tool for C/C++.
  - Lint – The original static code analyzer for C.
  - LDRA Testbed – A software analysis and testing tool suite for C/C++.
  - Parasoft C/C++test – A C/C++ tool that does static analysis, unit testing, code review, and runtime error detection; plugins available for Visual Studio and Eclipse-based IDEs.
  - PC-Lint – A software analysis tool for C/C++.
  - Polyspace – Uses abstract interpretation to detect and prove the absence of run time errors, Dead Code in source code as well as used to check all MISRA (2004, 2012) rules (directives, non directives).
  - PVS-Studio – A software analysis tool for C, C++, C++11, C++/CX (Component Extensions).
  - PRQA QA·C and QA·C++ – Deep static analysis of C/C++ for quality assurance and guideline/coding standard enforcement with MISRA support.
  - SLAM project – a project of Microsoft Research for checking that software satisfies critical behavioral properties of the interfaces it uses.
  - Sparse – An open-source tool designed to find faults in the Linux kernel.
  - Splint – An open-source evolved version of Lint, for C.