

HMIN306: Analyse Statique & Dynamique

Pascal Zaragoza
MAREL - LIRMM

pascal.zaragoza
@berger-levrault.com
@lirmm.fr



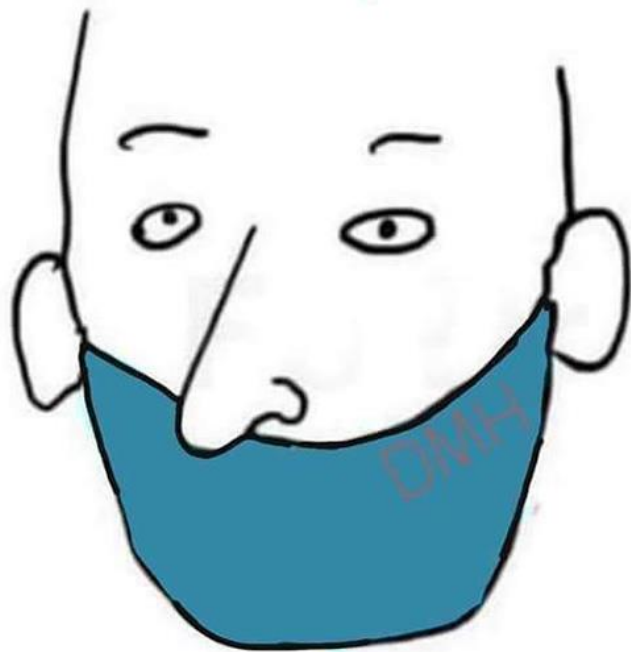
HMIN306: Analyse Statique & Dynamique

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

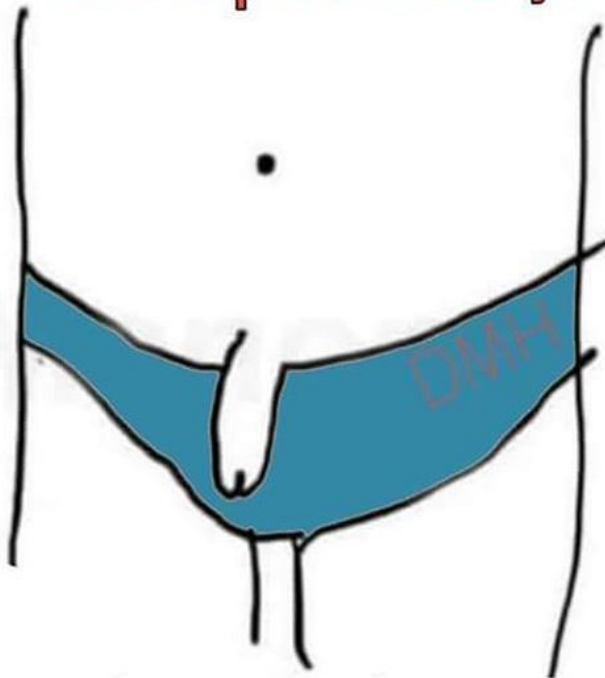
- Jonathan Aldrich Charlie Garrod : Principles of Software Construction: Objects, Design and Concurrency ; Static Analysis
- Plusieurs documents/cours rédigés par Bruno Duffour (notamment analyse dynamique)
- Rapport de synthèse de Jean-Yves Bouterre
- etc.



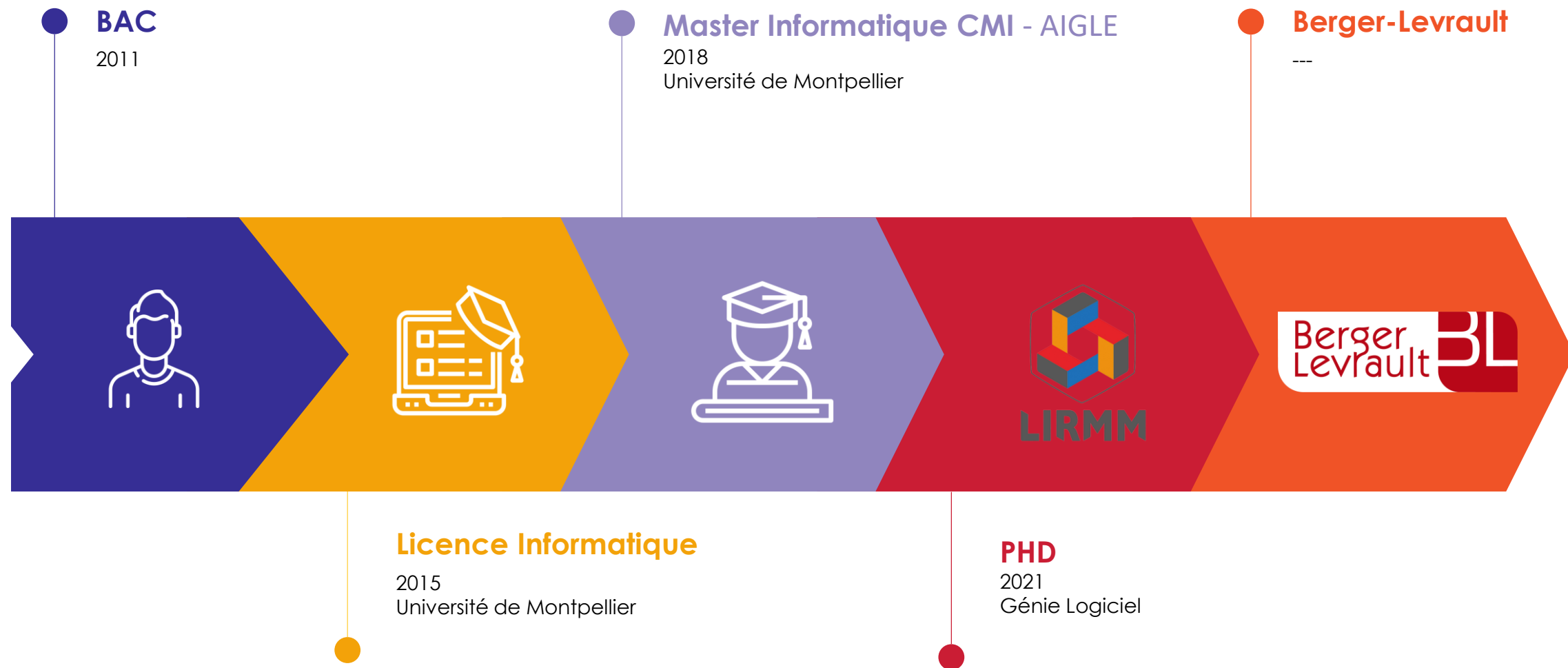
Porter son masque
comme ça...



C'est comme porter
son slip comme ça



- Port du masque obligatoire.
- Vous devez rester à votre place.
- Vous devez minimiser les contacts avec les autres.



Analyse de programme

● Introduction sur l'analyse

● Analyse Statique

● Analyse Dynamique

● Outils

● Q&As

Introduction



**Analyser le
comportement/propriétés d'un
programme automatiquement.**

Def. Analyse de programme

Pourquoi analyser?

01

Compréhension

Améliorer la vue sur un projet, pour faciliter la maintenance, l'évolution, etc.

02

Vérification

Pour vérifier le bon fonctionnement des programmes avant la livraison d'un logiciel.

03

Transformation

Pour faciliter la transformation et une éventuelle migration du projet.

04

Optimisation

Une analyse automatique permet de proposer au développeur des solutions optimisés.

05

Décompilation

Facilite la compréhension lors de la décompilation de code.

06

Obfuscation

Permet d'extraire des information du code existant pour réduire l'obfuscation naturelle créée lors du développement du logiciel.

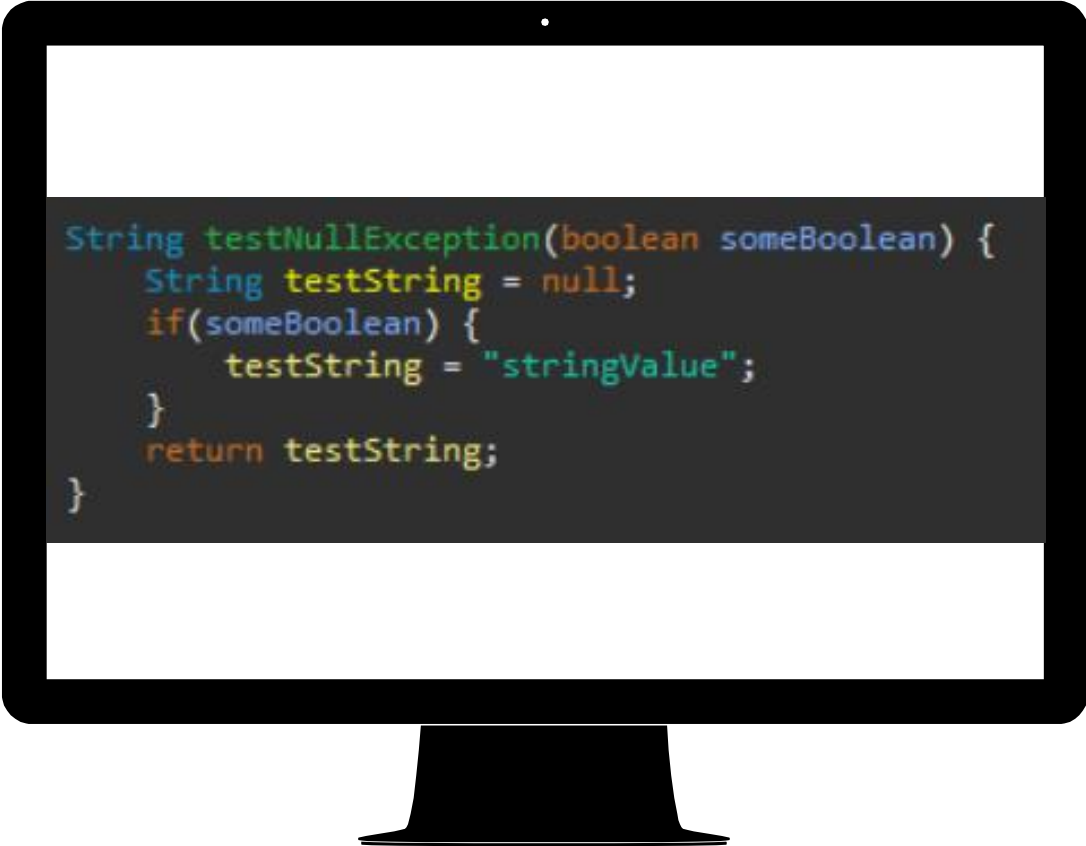
Pourquoi analyser : exemples

1. 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

2. Détection de code malicieux

- Exemple : Utilisation d'automates de sécurité



```
String testNullException(boolean someBoolean) {  
    String testString = null;  
    if(someBoolean) {  
        testString = "stringValue";  
    }  
    return testString;  
}
```

Pourquoi analyser : exemples

1. Détection l'exécution

• Exemples

- Accès nul
- Accès
- Cast ill
- Boucle
- Chemi
- Variab
- Synchr

2. Détection

- Exemple
sécurité

```

Problems @ Javadoc Declaration Console
<terminated> WindowHandles [JUnit] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe (Aug 15, 2019, 1:23:22 PM)
java.lang.NullPointerException
    at WindowHandles.test(WindowHandles.java:60)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
    at java.lang.reflect.Method.invoke(Unknown Source)
    at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:45)
    at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:15)
    at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:42)
    at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:20)
    at org.junit.internal.runners.statements.RunAfters.evaluate(RunAfters.java:30)
    at org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:263)
    at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:68)
    at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:47)
    at org.junit.runners.ParentRunner$3.run(ParentRunner.java:231)
    at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:60)
    at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:229)
    at org.junit.runners.ParentRunner.access$000(ParentRunner.java:50)
    at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:222)
    at org.junit.runners.ParentRunner.run(ParentRunner.java:300)
    at org.eclipse.jdt.internal.junit4.runner.JUnit4TestReference.run(JUnit4TestReference.java:86)
    at org.eclipse.jdt.internal.junit.runner.TestExecution.run(TestExecution.java:38)
    at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.runTests(RemoteTestRunner.java:459)
    at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.runTests(RemoteTestRunner.java:678)
    at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.run(RemoteTestRunner.java:382)
    at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.main(RemoteTestRunner.java:192)

```



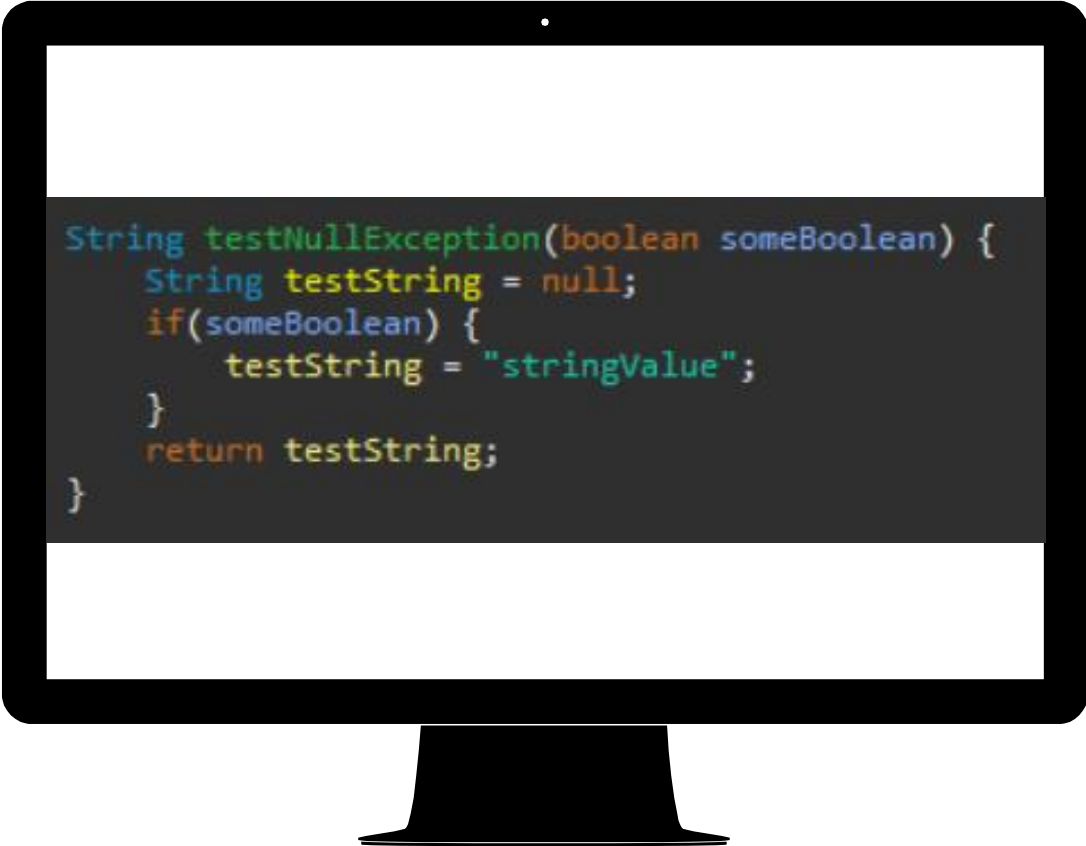
Pourquoi analyser : exemples

1. 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

2. Détection de code malicieux

- Exemple : Utilisation d'automates de sécurité



```
String testNullException(boolean someBoolean) {  
    String testString = null;  
    if(someBoolean) {  
        testString = "stringValue";  
    }  
    return testString;  
}
```

Pourquoi analyser ?

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

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

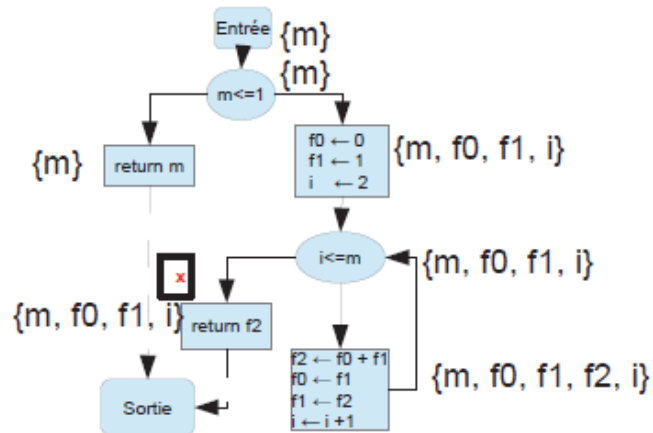
Error: may have null pointer on line 12

```

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

Pourquoi analyser ?

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



```

1 int fib(int m){
2   if(m <= 1){
3     return m;
4   }else{
5     int f0= 0, f1= 1, f2, i;
6     for(i=2; i<m;i++){
7       f2= f0 + f1;
8       f0= f1;
9       f1= f2;
10    }
11    return f2;
12  }
13 }
    
```

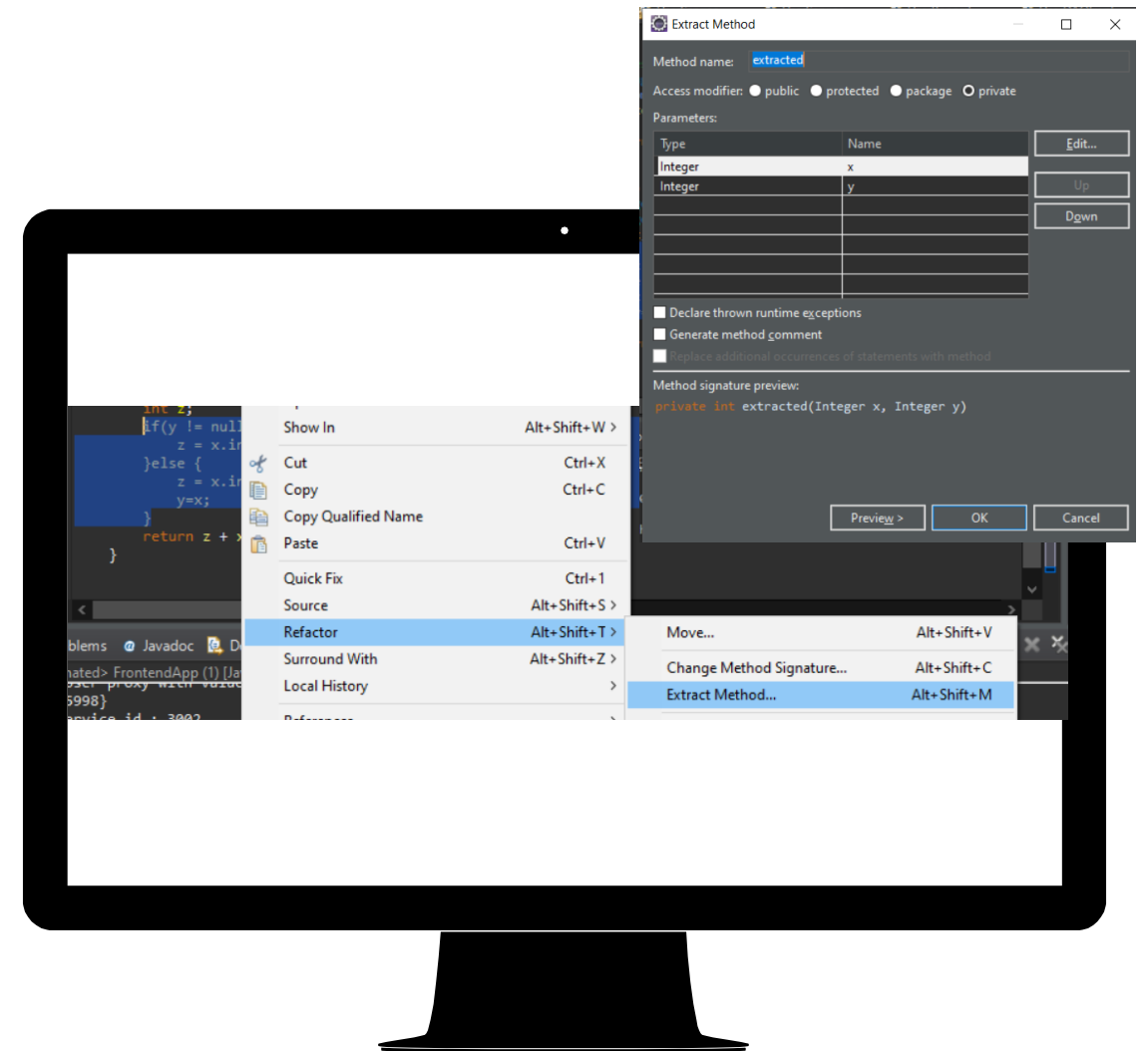
Pourquoi analyser ?

1. Détection d'erreurs pouvant survenir à l'exécution.
 - A null pointer checker : Vérification des pointeurs null basée sur l'analyse du flot de données
2. Détection d'erreurs pouvant survenir à l'exécution
 - Variable non initialisée
3. Transformation et compréhension
 - Calculer certaines métriques sur le code source d'un programme



Transformation et compréhension : Découpage (slicing)

1. Technique utilisée dans le but de faire ressortir certaines instructions d'un programme en relation avec une propriété.
2. Le résultat du découpage est un sous ensemble du programme.
3. Technique utile à la réutilisation du code. (voir exemple)
4. Technique utile à la compréhension
 - Séparer un programme complexe en partie de code moins compliquées.



Pourquoi analyser? OK
Mais comment?

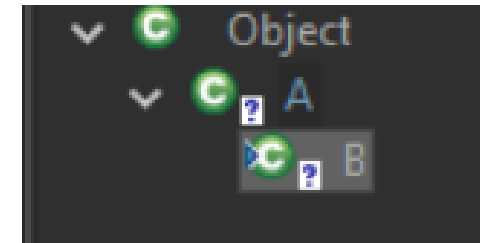
Analyse statique

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

Analyse dynamique

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

1. Pas d'exécution, donc pas de dommages et pas de temps d'exécution.
2. Par contre, il n'est pas possible d'être certain de certaines propriétés avec l'analyse statique à cause du problème de l'indécidabilité.



```
private IA returnsTypeIA(boolean someBoolean) {  
    if(someBoolean)  
        return new B();  
    else  
        return new A();  
}
```

Figure. Exemple simple du problème de l'indécidabilité.

1. Permet d'obtenir des résultats plus précis pour une ou plusieurs exécutions concrètes.
2. Permet d'obtenir de l'information de nature temporelle à propos de l'exécution
 - Ex: Temps de performance, temps resté sur une partie du programme...
3. Permet d'obtenir de l'information sur la fréquence ou l'importance de certains événements.

```
for(Event tmpEvent : result) {  
    int tmpid = tmpEvent.getHostId();  
    User tmpuser = userService.getUserById(tmpid);  
    username.add(tmpuser.getUsername());  
    System.out.println("host:" + tmpuser.getUsername());  
}
```

Figure. Quel est la fréquence d'appel de cette méthode?

Inconvénients de l'analyse dynamique

1. L'étendu de l'analyse dépend de l'étendu des scénarios d'exécution.
 - *Besoin de scénarios qui couvrent l'ensemble du code à analyser.*
2. Possibilité de dommage en cas d'analyse pour des raisons de sécurité ou de performance.
 - Ex: Temps de performance, temps resté sur une partie du programme...
3. Dépendance par rapport au temps d'exécution.

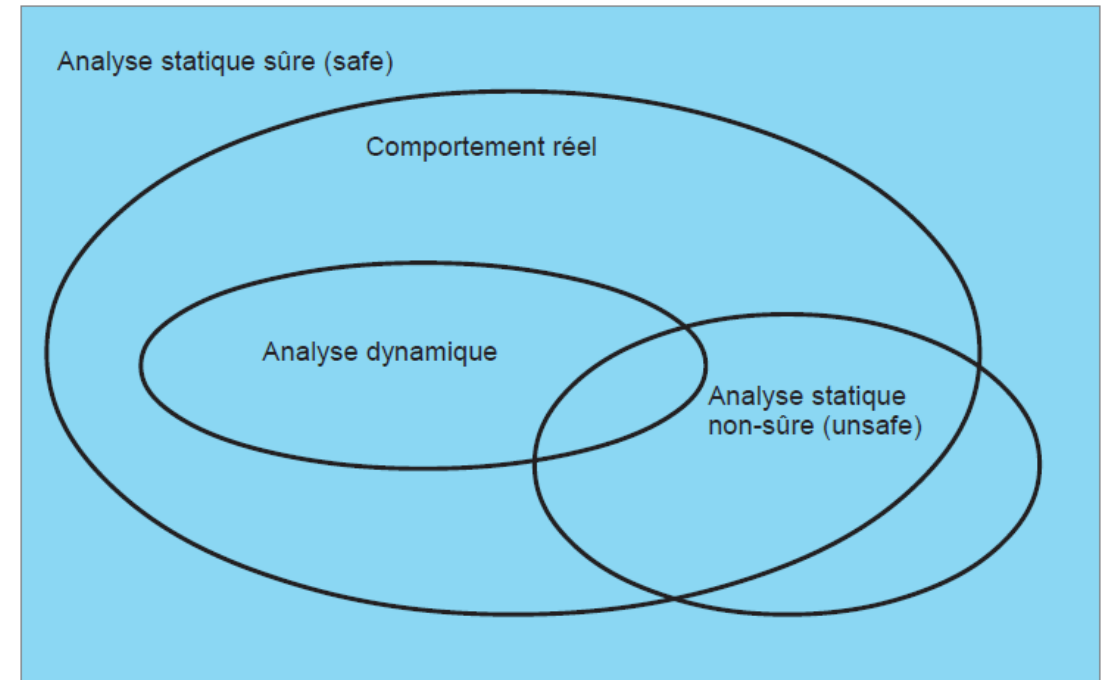


Figure. Couverture d'une analyse dynamique versus statique.

Analyse Statique

1. Code source (haut niveau)
2. Code 3-adresses (niveau intermédiaire) (TAC ou 3AC)
3. Code compilé (bas niveau)

```
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

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

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

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

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

    char one_char;
    cout << "\nEnter u caratère: ";
    cin >> one_char;
}
```

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

```
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

1. Code source (haut niveau)
2. **Code 3-adresses (niveau intermédiaire) (TAC ou 3AC)**
3. Code compilé (bas niveau)

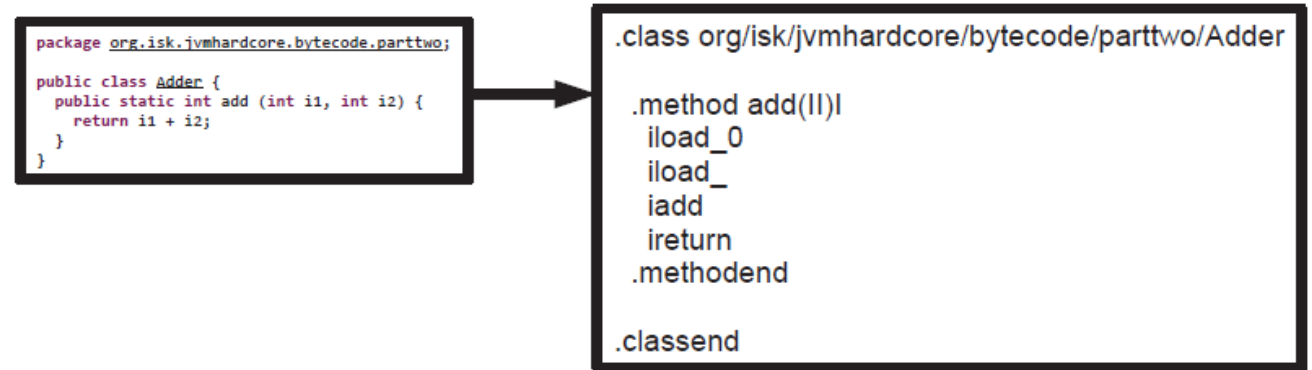
Calculate one solution to the [[quadratic equation]].

$x = (-b + \sqrt{b^2 - 4*a*c}) / (2*a)$

t1 := b * b
t2 := 4 * a
t3 := t2 * c
t4 := t1 - t3
t5 := sqrt(t4)
t6 := 0 - b
t7 := t5 + t6
t8 := 2 * a
t9 := t7 / t8
x := t9

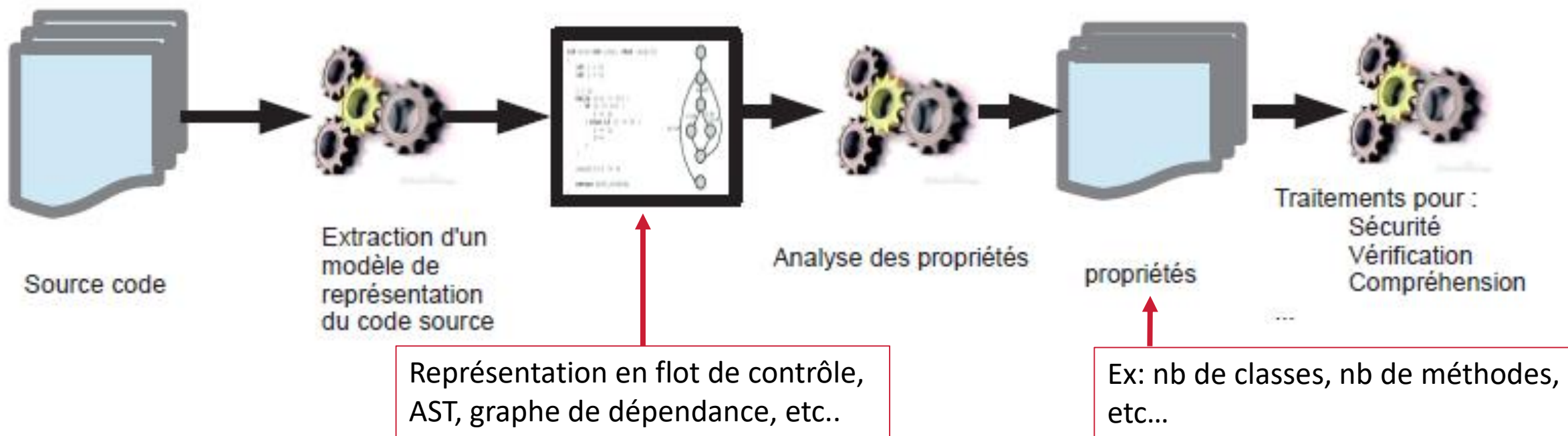
- Est un code intermédiaire utilisé pour l'optimisation du code.
- Chaque instruction TAC a au plus trois parties/éléments :
 - Généralement une combinaison de variables et un opérateur binaire.
 - Résultat <- <opérande1> <opérateur> <opérande2>

1. Code source (haut niveau)
2. Code 3-adresses (niveau intermédiaire) (TAC ou 3AC)
3. **Code compilé (bas niveau)**



1. Deux phases de l'analyse:

- Extraction d'un modèle de représentation du code source
- L'analyse (identification) des propriétés basées sur les modèle extraits



- Est un arbre dont les nœuds internes sont des opérateurs et dont les feuilles (ou nœuds externes) représentent les opérandes de ces opérateurs.
- Diffère d'un arbre de dérivation par l'omission des nœuds et des branches qui n'affectent pas la sémantique.
 - *Exemple: Omission des parenthèses*



```

graph TD
    Root["*"] --> L["+"]
    Root --> R["+"]
    L --> a1["a"]
    L --> M1["*"]
    R --> M2["*"]
    R --> d1["d"]
    M1 --> a2["a"]
    M1 --> b["b"]
    M2 --> c["c"]
    M2 --> d2["d"]
  
```

AST

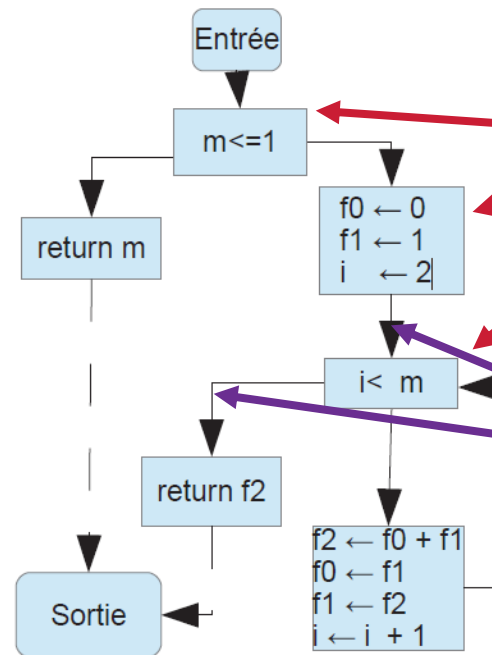
Graphe flot de contrôle (Control Flow Graph, CFG)

- Est une représentation sous forme de graphe de tous les chemins qui peuvent être suivis par un programme durant son exécution.

```

1 int fib(int m) {
2   if(m <= 1) {
3     return m;
4   } else {
5     int f0= 0, f1= 1, f2, i;
6     for(i=2; i<m; i++) {
7       f2= f0 + f1;
8       f0= f1;
9       f1= f2;
10    }
11    return f2;
12  }
13 }

```



Bloc de base : un bout de code d'un seul tenant sans sauts ni cibles de sauts (les sommets).

cible de saut : Les arcs représentent les sauts dans le flot de contrôle

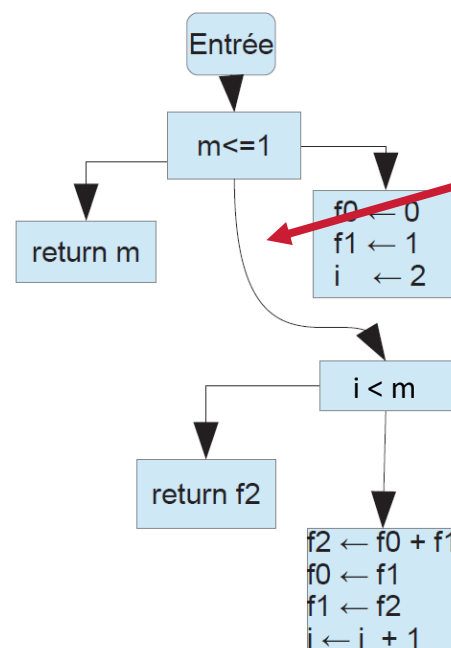
Les graphe de dépendance de contrôle

- Montre quelles instructions seront exécutées en fonction de la valeur d'une expression dans le programme.
- Les nœuds du graphe sont les mêmes que ceux du CFG.

```

1 int fib(int m) {
2   if (m <= 1) {
3     return m;
4   } else {
5     int f0 = 0, f1 = 1, f2, i;
6     for (i = 2; i < m; i++) {
7       f2 = f0 + f1;
8       f0 = f1;
9       f1 = f2;
10    }
11    return f2;
12  }
13 }

```



Pour deux nœuds 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.

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.

```
1 int fib(int m) {
2     if(m <= 1){ // Utilise = {m}; Définit = {}
3         return m; // Utilise = {m}; Définit = {}
4     }else{
5         int f0= 0, f1= 1, f2, i; // Utilise = {}; Définit = {f0, f1}
6         for(i=2; i<m;i++){ // Utilise = {m,i}; Définit = {i}
7             f2= f0 + f1; // Utilise = {f0,f1}; Définit = {f2}
8             f0= f1; // Utilise = {f1}; Définit = {f0}
9             f1= f2; // Utilise = {f2}; Définit = {f1}
10        }
11        return f2; // Utilise = {f2}; Définit = {}
12    }
13 }
```

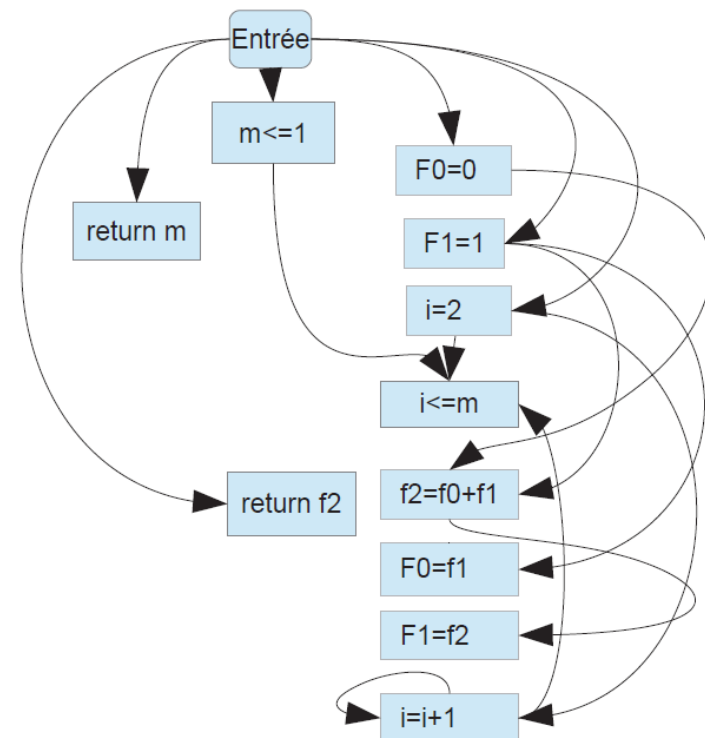
Le graphe de dépendance de données

- 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 q sans qu'elle ne soit modifiée entre temps.

```

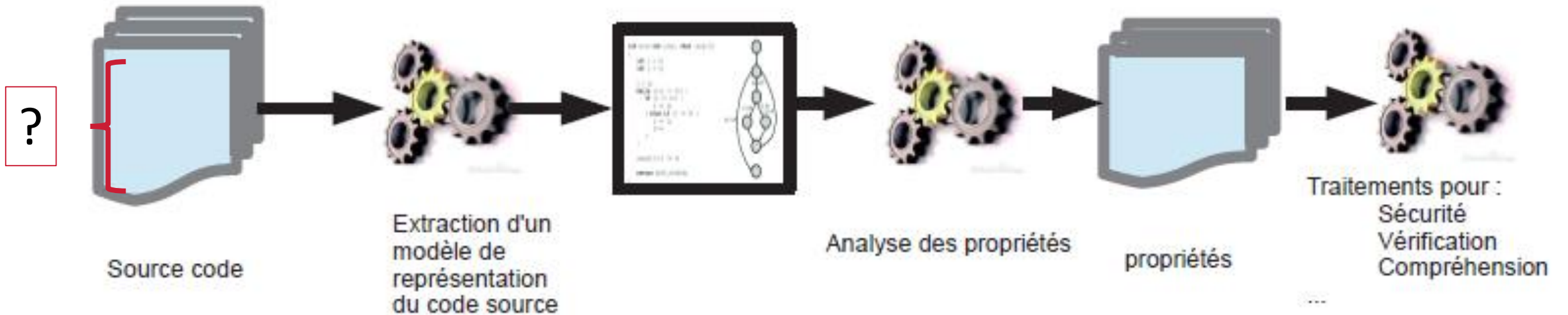
1 int fib(int m) {
2     if(m <= 1) {
3         return m;
4     } else {
5         int f0= 0, f1= 1, f2, i;
6         for(i=2; i<m; i++) {
7             f2= f0 + f1;
8             f0= f1;
9             f1= f2;
10        }
11        return f2;
12    }
13 }

```



On peut analyser du code à plusieurs niveaux :

1. Locale (« peephole ») : l'analyse se porte sur quelques instructions.
2. Intra-procédurale : l'analyse porte sur le code d'une seule méthode.
3. Inter-procédurale : l'analyse porte sur un programme complet ou un fragment de programme.



Requiert uniquement la séquence des instructions du programme

- Utiliser par exemple pour la transformation du code pour l'optimisation
- Ex:

constant folding : évaluation et remplacement des sous-expressions constantes

- $i = 320 * 200 * 32 \rightarrow i = 2048000$
- "abc" "def" \rightarrow "abcdef".

strength reduction : remplacer les opérations lentes par des opérations plus rapides
(remplacer une multiplication par une addition).

```
int c = 7;
int[] y = new int[11];
for(int i=0; i<10; i++)
    y[i] = c * i;
```

```
int c = 7;
int[] y = new int[11];
int k = 0;
for(int i=0; i<10; i++)
    y[i] = c * i;
    k = k + c;
```


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

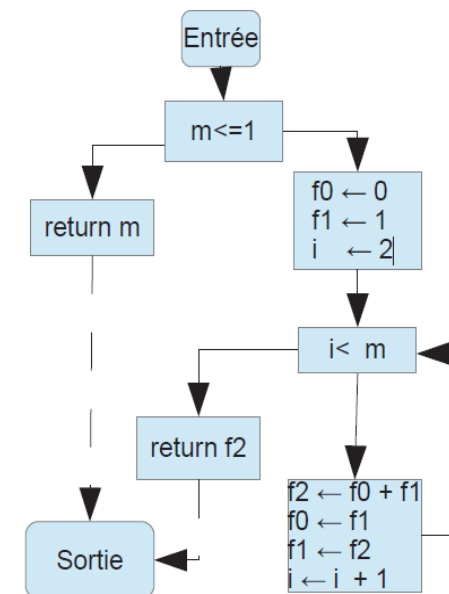
Exemples:

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

```

1 int fib(int m) {
2   if(m <= 1) {
3     return m;
4   }else{
5     int f0= 0, f1= 1, f2, i;
6     for(i=2; i<m;i++){
7       f2= f0 + f1;
8       f0= f1;
9       f1= f2;
10    }
11    return f2;
12  }
13 }

```



Examples :

- Analyse de pointeurs/références
- Analyse d'échappement (escape analysis)
- Analyse de tainte (taint analysis)

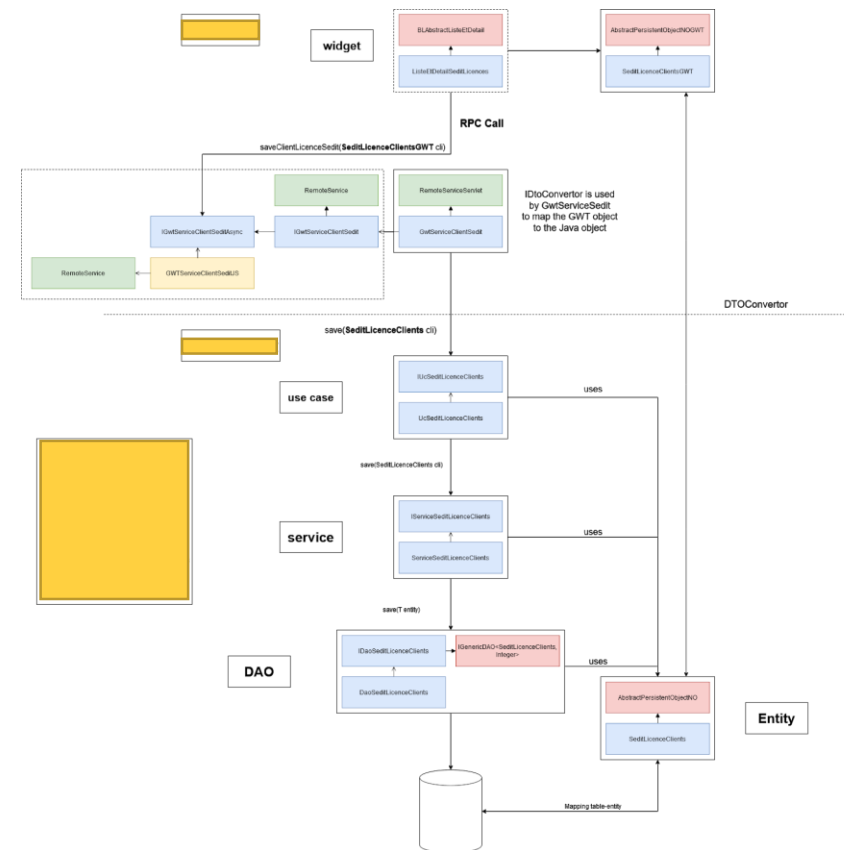


Figure. Analyse de l'architecture d'un code

Graphe d'appel

- Est une représentation sous forme de graphe orienté qui représente les relations d'appel entre les procédures/méthodes d'un programme.
 - Chaque nœud représente une procédure/méthode.
 - Chaque arc représente une procédure qui appelle une autre.
- Un graphe d'appel peut-être statique ou dynamique
 - Un graphe d'appel dynamique est une représentation des appels enregistrés durant certaines exécutions du programmes.
 - Un graphe d'appel statique représente tous les appels possibles durant d'exécution d'un programme.

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

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

    public void ma2(){atta2.mc1();}
    public void ma3(){attal.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"); }
}
```

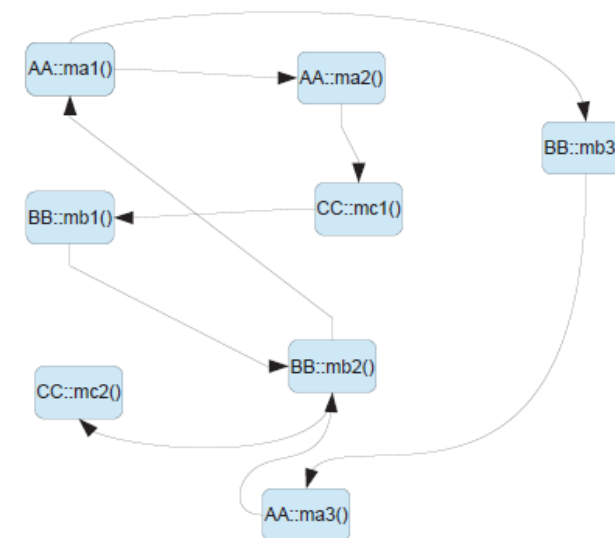


Figure. Trois classes avec son graphe d'appels.

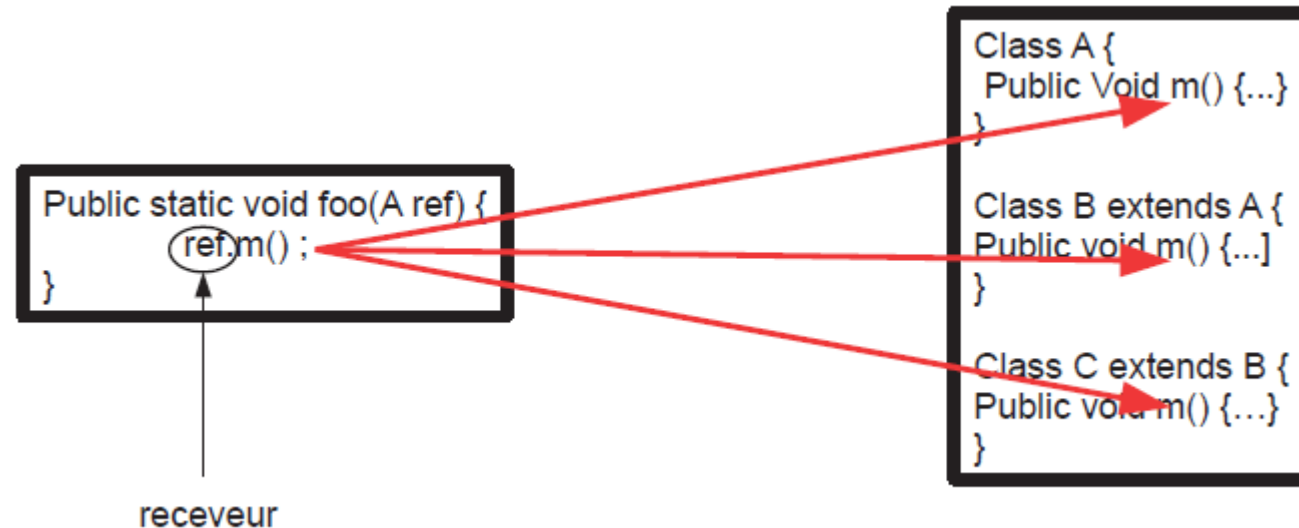
- L'analyse de références (reference analysis) permet de déterminer, pour chaque variable (ou attribut) dans le programme un ensemble d'objets auxquels elle peut pointer à l'exécution.
- Cas d'analyses de références:
 - Appel statique
 - Cible connue à la compilation
 - Appel spécial (méthodes privées, super)
 - Cible connue à la compilation
 - Virtuel/interface
 - Cible dépend du type dynamique (connue lors de l'exécution).

```
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 ");}  
}
```

- 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'analyses de références:
 - Appel statique
 - Cible connue à la compilation
 - Appel spécial (méthodes privées, super)
 - Cible connue à la compilation
 - Virtuel/interface
 - Cible dépend du type dynamique (connue lors de l'exécution).

```
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 B  
        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 ");}  
}
```

Résolution des appels pour les appels virtuels : dans les langages OO, la cible d'un appel virtuel dépend du type de l'objet receveur à l'exécution.



Il existe divers algorithmes pour générer un graph de dépendance :

1. CHA : Class Hierarchy Analysis
2. RTA : Rapid Type Analysis
3. XTA : Tip-Palsberg Class Analysis

Tip and Palsberg, « Scalable Propagation-based Call Graph Construction Algorithms », OOSLA'00



Idée :

- regarder **l'hierarchie des classes** pour déterminer quelles classes peuvent être référencées lors qu'on référence une classe mère A.
- Trouver les méthodes qui sont invoqués
- Prendre le pire cas et créer une référence vers toutes les méthodes de la classe mère et ses classes filles.

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

```
class AA {
    public BB attal;
    public CC attal2;
    public int attal3;

    public void ma1(){
        ma2();
        if (attal3 > 5) {attal = new BB();}
        else attal.mb3(); }

    public void ma2(){attal2.mc1();}
    public void ma3(){attal.mb2();}
}

class AAA extends AA {
    public void ma3(){attal.mb1();}
}

class AAAA extends AA {
    public void ma3(){attal.mb1();}
}

class AAAAA extends AA {
    public void ma4(){attal3 = 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();}
}
```



```
class AA {
    public BB attal;
    public CC attal2;
    public int attal3;

    public void ma1(){
        ma2();
        if (attal3 > 5) {attal = new BB();}
        else attal.mb3(); }

    public void ma2(){attal2.mc1();}
    public void ma3(){attal.mb2();}
}

class AAA extends AA {
    public void ma3(){attal.mb1();}
}

class AAAA extends AA {
    public void ma3(){attal.mb1();}
}

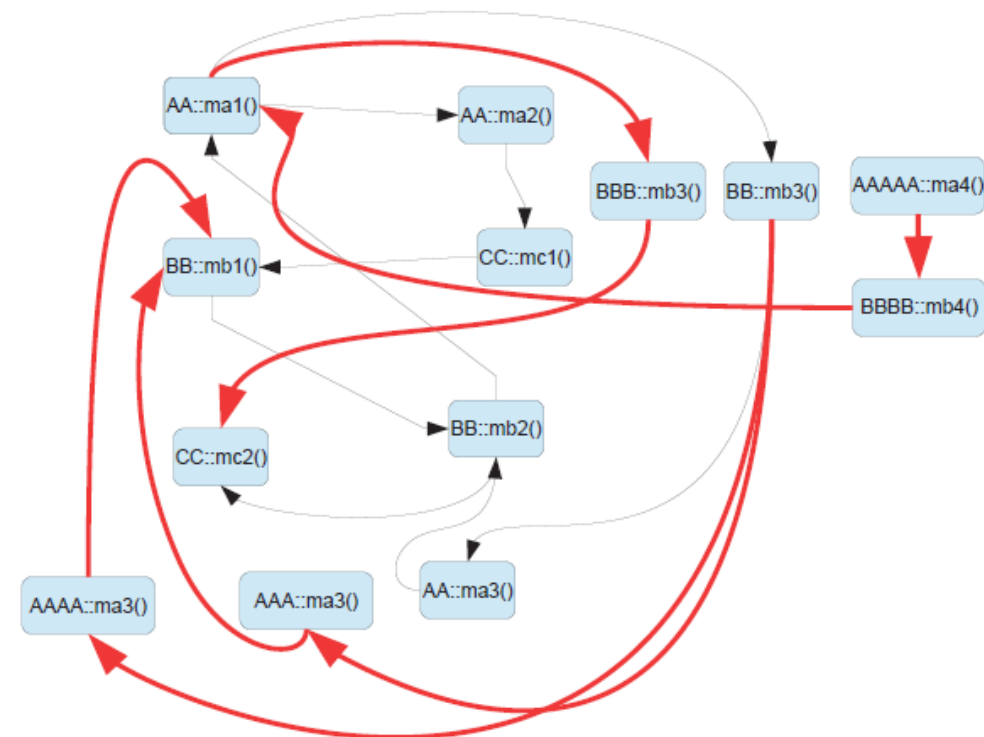
class AAAAA extends AA {
    public void ma4(){attal3 = 1 ;
        (new BBB()).mb4();
    }
}
```

```
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();}
}
```



Amélioration de CHA

Points clés:

- On ignore le classes pour lesquelles aucun objet n'est créé.
- Une cible (d'une référence) 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 la volet.
- Ignore le flot de contrôle

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



Class Hierarchy
Analysis

Rapid Type
Analysis

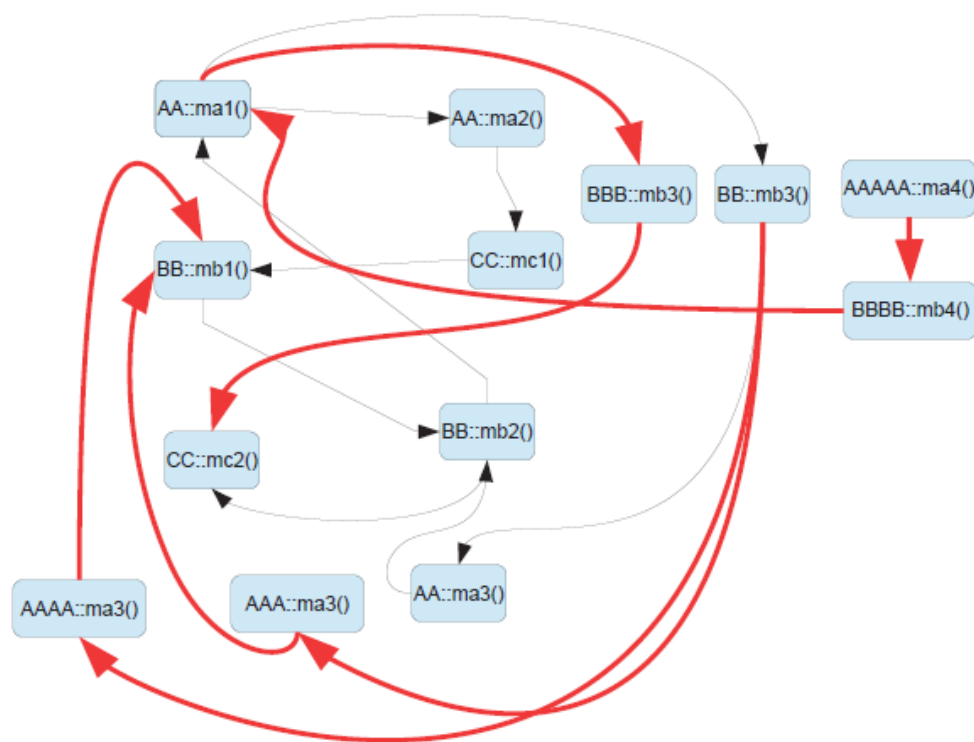


Figure. Graphe généré par technique CHA.

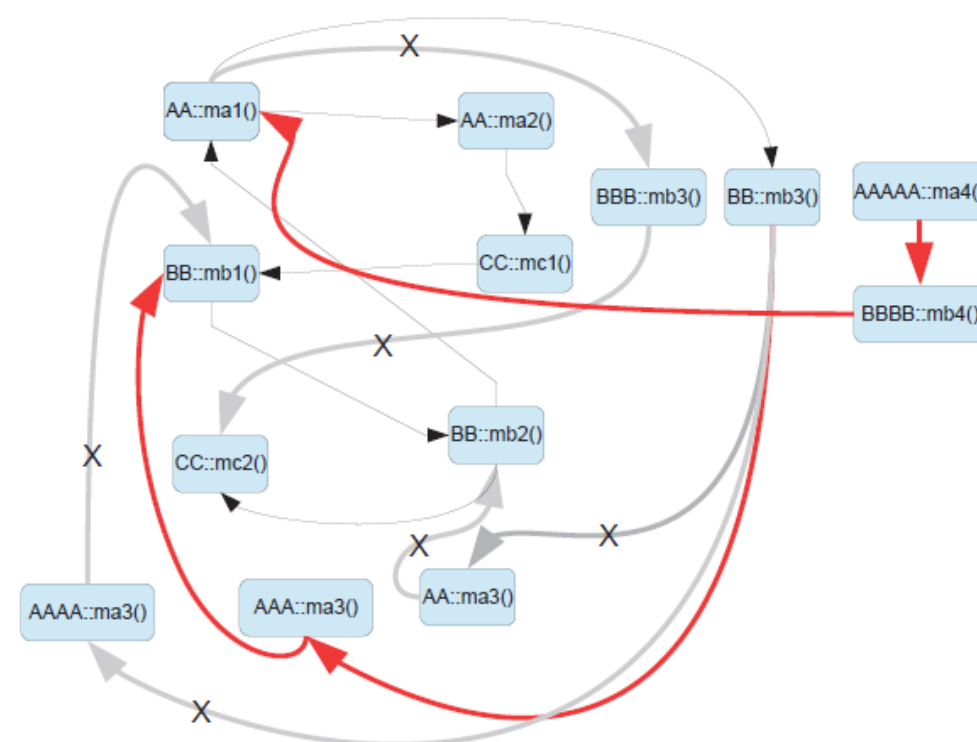


Figure. Graphe généré par technique RTA.

```
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( );
}
```

- Invocation pour result1 : CHA résout le cas de l'appel vers foo(1).
- Invocation pour result2 : CHA peut résoudre dans ce cas car la classe B n'a pas de sous-type.
- Invocation pour result3: CHA ne peut pas différencier, par contre RTA va pouvoir déduire grâce à l'instanciation de la classe B.

Limitation liées à la sûreté des types :
CHA et RTA fonctionnent dans un contexte où la sûreté des types est respectée.

Solution: XTA

- Amélioration sur RTA
- Utilise un ensemble de types par méthodes et par variable plutôt qu'un ensemble global.
- Filtre les types passées en paramètres / retour

Tip and Palsberg, « Scalable Propagation-based Call Graph Construction Algorithms », OOSLA'00

```
//#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
```

**Merci.
Des Questions ?**



berger-levrault.com

