|  |
| --- |
|  |
| OPL - PathImpact |
|  |
| Analyseur automatique d’impacts de Code Source |

**Université Lille 1 – Sciences et Technologies**

13 février 2017

Créé par : Étienne WATTEBLED

OPL - PathImpact

Analyseur automatique d’impacts de Code Source

Table des matières

[I. Introduction 4](#_Toc474081874)

[II. PathImpact 6](#_Toc474081875)

[a. Stacktraces 6](#_Toc474081876)

[b. Sequitur 7](#_Toc474081877)

[c. DAG 13](#_Toc474081878)

[III. Implémentation 14](#_Toc474081879)

[a. Structures principales 14](#_Toc474081880)

[Stacktrace 14](#_Toc474081881)

[Grammaire 14](#_Toc474081882)

[b. Stacktrace 15](#_Toc474081883)

# Introduction

De jour en jour, afin d’améliorer la productivité ou l’efficacité, les métiers évoluent provoquant des besoins fonctionnels qui ne cessent d’évoluer même lorsque les applications sont opérationnelles et utilisées par les utilisateurs. Les applications possèdent, par conséquent, de plus en plus de fonctionnalités et deviennent de plus en plus complexes. Très vite, la maintenance devient alors très onéreuse, difficile à gérer et préoccupante. Il n’est pas rare qu’une ou plusieurs fonctionnalités soient modifiées, ou même supprimées durant le développement d’un logiciel, ce qui peut provoquer de multiples modifications qui se répercutent.

On appelle ces modifications des « impacts ».

Ces derniers peuvent devenir rapidement catastrophiques, de par l’apparition de régressions ou de code simplifiable voire mort qui peut persister pendant des années. De plus, à cause de la complexité et de la taille de certaines applications, certaines modifications peuvent s’avérer extrêmement coûteuses à mettre en place, si bien que, le client pourrait refuser de procéder à une évolution.

Il est alors indispensable et pouvoir évaluer avec précision, les impacts provoqués par une modification donnée d’une méthode, ne serait-ce que pour estimer le coup ou conserver le code sans parasites. Cependant, l’analyse manuelle peut s’avérer complexe et longue, et si l’évolution est annulée, ce temps utilisé aura été inutilement perdu.

Une solution existe : il existe des algorithmes capables de détecter automatiquement les impacts, plus précisément, de lister toutes les méthodes susceptibles d’être « touchées » par la modification d’une méthode. Ils sont utilisés pour faire des chiffrages, pour gagner du temps à mettre à jour une STD (Spécifications Techniques Détaillées), ou encore, lister avec précision les modifications à apporter avant de procéder au développement pour ne rien oublier.

Certains analyseurs d’impacts sont plus précis que d’autres mais les algorithmes permettant de parvenir à des résultats extrêmement précis sont très consommateurs en termes de puissance de calculs.

Ce rapport présente une grande partie de l’implémentation d’un analyseur d’impacts appelé « PathImpact » qui fait partie des analyseurs qui consomment le plus de ressources, mais il dispose d’une excellente précision. Seule la dernière étape permettant de déterminer les impacts n’est pas été traitée dans ce projet.

Avant tout, ce document explique comment fonctionne PathImpact et détaille par la suite son implémentation. La troisième partie concerne les tests unitaires JUnit, suivie d’un chapitre traitant l’évaluation. Et enfin, une phase de critiques et de limitations précède la conclusion.

# PathImpact

PathImpact est un analyseur d’impacts fonctionnant en trois étapes principales :

1. **Détermination et concaténation des stacktraces**
2. **Compression des stacktraces via un algorithme appelé Sequitur**
3. **Construction d’un arbre appelé DAG, et détermination des impacts.**

Ces étapes sont expliquées dans les parties ci-dessous.

## Stacktraces

L’algorithme PathImpact a besoin des différentes stacktraces en entrée pour déterminer les impacts.

Afin de les déterminer, il est nécessaire d’exécuter le programme dans tous ses états possibles, autrement dit, tester tous les cas de paramètres en entrée. Ainsi, si le programme a juste besoin d’un entier « n » pour fonctionner, il faut tester toutes les valeurs de « n » représentables (de Integer.MIN\_VALUE jusque Integer.MAX\_VALUE en Java).

La stacktrace est représentée par une collection ordonnée d’éléments représentant les méthodes.  
Ainsi, « A B » signifie « la méthode A appelle la méthode B ». Cependant, pour pouvoir faire la différence entre : « A appelle B, B retourne une valeur, puis A appelle C » et « A appelle B, puis B appelle C », il est nécessaire d’intégrer un nouvel élément que l’on appelle « r » et qui signifie « return ».  
Ainsi, « ABrCDrrr » signifie : « A appelle B, B retourne une valeur, A appelle C, C appelle D, D retourne une valeur, C retourne une valeur, puis A retourne une valeur ».

Toutes les stacktraces sont par la suite concaténées les unes après les autres. Par conséquent, un nouvel élément « x » est utilisé et signifie « fin du programme ».

En considérant l’exemple ci-dessous, la stacktrace générée doit alors être :

« main, m, r, r, x, main, m2, m3, r, m4, r, r, r, x »

public class Test {

public static void main(String args[]) {

int i = 0;

i = Integer.parseInt(args[i]) ;

if (i == 0) {

m();

} else {

m2();

}

}

public static void m() {

...

}

public static void m2() {

m3();

m4();

}

public static void m3() {

...

}

public static void m4() {

...

}

}

## Sequitur

Sequitur est un algorithme de compression utilisant la théorie des langages, plus précisément, la grammaire pour parvenir à ses fins.

Une grammaire est un ensemble de règles de production qui permet de générer les mots d’un langage.  
Toute grammaire possède une règle de départ (axiome) que l’on appelle la plupart du temps « S ».

Voici par exemple, la grammaire des expressions arithmétiques, dans laquelle l’axiome est « exp » :

exp → exp + exp | exp × exp | (exp) | num  
num → chiffre num | chiffre  
chiffre → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

**Wikipédia (Grammaire formelle)**

L’algorithme de Sequitur est basé sur deux propriétés fondamentales :

1. Il ne doit pas y avoir de doublons de digramme.  
   *Définition de digramme : assemblage de deux signes (généralement deux lettres de l’alphabet).*
2. Si une règle de production n’est plus utilisée ou n’est utilisée qu’une seule fois, elle doit être « désappliquée » puis supprimée.

Ainsi, la grammaire G définit par les règles :

S → aaaabb1p  
1 → cde  
2 → fgh

déroge les deux propriétés (1 et 2) car :

1. Le digramme « aa » est présent deux fois.  
   Solution : créer une nouvelle règle 3 → aa
2. La règle 1 n’est utilisée qu’une seule fois.  
   La règle 2 n’est pas du tout utilisée.  
   Solution : supprimer la règle 2 et « désappliquer » la règle 1.

La même grammaire respectant les deux propriétés est alors :

S → 33bbcdep  
3 → aa

Étant donné que la grammaire vide respecte les deux propriétés, celle-ci devient le point de départ de Sequitur et la règle de production « S » va grandir petit à petit.

Dès qu’une règle est dérogée, Sequitur va utiliser les solutions associées aux propriétés (ci-dessus) pour rectifier la grammaire.

Le pseudo algorithme de Sequitur est donc :

En paramètre, une simple liste l d’éléments

. Créer une grammaire g vide.  
. Ajouter le 1er élément de la liste l à la règle S de la grammaire g.

. Pour chaque élément e de la liste l (à partir du 2ème élément)

. Ajouter l’élément e dans la règle S de la grammaire (à la fin)

. Tant que les deux derniers éléments de S forment un bigramme déjà rencontré

. S’il existe déjà une règle r qui produit ces deux éléments

. La récupérer

Sinon

. Créer la règle r qui produit ces deux éléments

Fin si  
 . Appliquer la règle r sur toute la grammaire

. Désappliquer puis supprimer toutes les règles qui ne sont pas utilisées au moins deux fois

Fin tant que

Fin pour

Ce pseudo algorithme est très simplifié, principalement sur la notion de « bigramme déjà rencontré ».

Exemple : On considère les éléments suivants :

« D A B A B C D A B E B C D A B A B »

S →

S → D

S → D A

S → D A B

S → D A B A

S → D A B A B [le bigramme A B existe, on crée une règle]

S → D 1 1  
1 → A B

S → D 1 1 C  
1 → A B

S → D 1 1 C D  
1 → A B

S → D 1 1 C D A  
1 → A B

S → D 1 1 C D A B [la règle AB existe déjà, on l’utilise]  
1 → A B

S → D 1 1 C D 1 [le bigramme D1 est en double, on crée une règle]  
1 → A B

Tant que les deux derniers éléments sont en double, aucun élément ne doit être ajouté à « S ».

S → 2 1 C 2  
1 → A B  
2 → D 1

S → 2 1 C E  
1 → A B  
2 → D 1

S → 2 1 C E B  
1 → A B  
2 → D 1

S → 2 1 C E B C  
1 → A B  
2 → D 1

S → 2 1 C E B C D  
1 → A B  
2 → D 1

S → 2 1 C E B C D A  
1 → A B  
2 → D 1

S → 2 1 C E B C D A B [la règle AB existe]  
1 → A B  
2 → D 1

S → 2 1 C E B C D 1 [la règle D1 existe]  
1 → A B  
2 → D 1

S → 2 1 C E B C 2  
1 → A B  
2 → D 1

S → 2 1 C E B C 2 A  
1 → A B  
2 → D 1

S → 2 1 C E B C 2 A B [la règle AB existe]  
1 → A B  
2 → D 1

S → 2 1 C E B C 2 1 [le bigramme 21 est en double, on crée une nouvelle règle  
1 → A B  
2 → D 1

S → 3 C E B C 3  
1 → A B  
2 → D 1   
3 → 2 1

La règle 2 n’est utilisée qu’une seule fois, il faut la « désappliquer »

S → 3 C E B C 3  
1 → A B  
3 → D 1 1

Sequitur fonctionne donc sur des bigrammes et parvient parfois à créer des règles de plus de deux éléments. Mais comment parvient-il à réutiliser une règle de taille supérieure à 2 ?

En réalité, Sequitur va former une nouvelle règle qui va progressivement grandir pendant que la règle qui existait va progressivement diminuer de taille jusqu’à n’avoir plus que deux éléments. À partir de là, puisque cette règle n’aura plus que deux éléments, elle sera détectée et la dernière règle créée ne sera plus utilisée qu’une seule fois, elle sera donc supprimée.

Pour illustrer ce phénomène, voici un exemple :

« ABCDABCDABCD »

S → A

S → A B

S → A B C

S → A B C D

S → A B C D A

S →A B C D A B [création règle AB]

S → 1 C D 1  
1 → A B

S → 1 C D 1  
1 → A B

S → 1 C D 1 C [création règle 1C]  
1 → A B

S → 2 D 2  
1 → A B [règle 1 utilisée une seule fois]  
2 → 1 C

S → 2 D 2  
2 → A B C

S → 2 D 2 D [création règle 2D]  
2 → A B C

S → 3 3  
2 → A B C [règle 2 utilisée une seule fois]  
3 → 2 D

S → 3 3  
3 → A B C D

S → 3 3 A  
3 → A B C D

S → 3 3 A B [AB est en doublon même s’il est dans une règle : on crée une nouvelle règle]  
3 → A B C D

S → 3 3 4  
3 → 4 C D  
4 → A B

La règle 3 va progressivement diminuer de taille, pendant que la règle 4 va grandir.

S → 3 3 4 C [4C est en doublon, création d’une nouvelle règle]  
3 → 4 C D  
4 → A B

S → 3 3 5  
3 → 5 D  
4 → A B [la règle 4, utilisée une seule fois]  
5 → 4 C

S → 3 3 5  
3 → 5 D  
5 → A B C

S → 3 3 5 D [la règle 5 D existe]  
3 → 5 D  
5 → A B C

S → 3 3 3  
3 → 5 D  
5 → A B C [la règle 5 n’est utilisée qu’une seule fois

S → 3 3 3  
3 → A B C D

On constate que Sequitur a bien réussi à réutiliser une règle possédant plus de deux éléments, et ce, grâce à la fois à la création de nouvelles règles de deux éléments et aux simplifications.

Afin d’éviter les règles en multiples exemplaires, il est donc important de vérifier si la bigramme à la fin de S n’est pas non plus présent dans une des règles (en doublon).

De plus, il est impossible d’obtenir des doublons de bigramme, car, que ce soit dans « S » ou dans une règle, si le bigramme existe déjà alors une règle est obligatoirement créée.

Sequitur est donc un algorithme qui fonctionne correctement dans tous les cas et la compression est très efficace.

## DAG

Le DAG est une représentation de la grammaire de Sequitur sous forme d’arbre de profondeur 2.

La racine contient tous les éléments de « S », la couche du dessous contient toutes les règles de production, et les feuilles sont les autres éléments (x, r, et les méthodes). Chaque élément de la racine est, soit rattaché à la couche du dessous s’il s’agit d’une règle, soit rattaché à une feuille. Tous les éléments des règles sont aussi rattachés de la même manière.

Une fois de DAG créé, si une méthode est modifiée, il est possible de déterminer les impacts.  
Pour cela, l’algorithme commence sur la feuille de la méthode modifiée et parcourt le DAG en avançant tant que l’élément « x » n’est pas présent. En procédant de cette manière, PathImpact détermine toutes les méthodes appelées après la méthode modifiée, et donc, susceptibles d’être affectées par le changement.

# Implémentation

## Structures principales

### Stacktrace

La stacktrace est représentée sous la forme d’une liste chaînée d’éléments.

Les classes et interfaces utilisées pour représenter une stacktrace sont les suivantes :

* **ElementItf** : interface représentant tous les éléments d’une stacktrace.   
  Seules les méthodes getNom() et print() sont présentes dans cette interface.
* **AbstractElement** : Classe abstraite implémentant ElementItf et ne possédant uniquement que le nom.
* **Evenement**: Type énuméré implémentant ElementItf. Les seules valeurs possibles sont « RETURN » dont le nom est « r » ainsi que « END\_OF\_PROGRAM » dont le nom est « x ».
* **Methode**: Classe héritant de AbstractElement.

### Grammaire

Afin d’implémenter Sequitur, une classe Grammaire a été mise en place.

Il est ainsi possible :

* De créer une grammaire
* D’ajouter un élément à « S »
* D’appliquer une règle à « S » ou à la grammaire entière.
* De désappliquer une règle à « S » ou à la grammaire entière.
* De récupérer toutes les règles de la grammaire
* De demander à la grammaire de simplifier toutes les règles.
* De savoir s’il existe une règle donnée (en lui passant une liste d’éléments)
* De demander à la grammaire de s’afficher (via une méthode print)
* De demander à la grammaire de créer une règle
* De demander à la grammaire de supprimer une règle (la règle sera alors automatiquement désappliquée sur toute la grammaire).
* De récupérer tous les éléments de « S » ou d’une règle.

Une classe Règle, permettant de représenter une règle a été créée et celle-ci hérite de la classe AbstractElement. Il a été choisi lors de l’implémentation que celle-ci soit une classe interne à Grammaire afin qu’aucune règle ne puisse, entre autres, être créée en dehors d’une grammaire. De plus, le fait que la classe Règle soit interne permet de ne rendre visibles certaines méthodes que pour la grammaire (exemple : le constructeur). De plus, il est plus juste de considérer le fait qu’il ne peut pas exister de règle sans grammaire. L’élément « S » (axiome) est en fait, représenté en tant que règle. Une fois l’instance de la règle obtenue par la grammaire, il est possible d’accéder aux éléments de celle-ci. Cependant, afin que les utilisateurs ne puissent pas modifier les éléments d’une règle sans passer par la grammaire, ou encore, supprimer une règle de la grammaire en récupérant la liste de toutes les règles sans que la grammaire en soit informée (et donc bypasser le système), toutes les listes sont retournées via une vue non modifiable (il existe des méthodes dans la classe Collections en Java qui permettent de retourner une vue non modifiable d’une liste, d’une map… etc.)

## Détermination des Stacktraces

Afin de récupérer les stacktraces, il est extrêmement complexe voire impossible de tester dans un temps raisonnable toutes les combinaisons de données possibles en paramètres et de déterminer toutes les stacktraces possibles du programme.

Une solution consiste à faire un parcours en profondeur et, une fois arrivée sur une feuille, de considérer le programme comme étant terminé. Il ne faut donc pas oublier de générer les éléments « x » correspondant ainsi l’élément « r » à la fin.

Ainsi, en reconsidérant la classe Test vue précédemment (page 7), la stacktrace sera :

« main m r r x main m2 m3 r r r x main m2 m4 r r r x »

au lieu de

« main, m, r, r, x, main, m2, m3, r, m4, r, r, r, x »

Pour générer correctement ces stacktraces et les concaténer, Spoon a été utilisé via un processor qui se déclenche sur chaque méthode (CtMethod). Si la fonction est une fonction main, alors elle est ajoutée à une liste dédiée aux fonctions mains. Par la suite, toutes les invocations présentes dans la méthode (tous les appels de méthode) sont enregistrées dans une Map qui, pour une méthode donnée, fournit une liste de méthode appelée.

Lorsque le processor a terminé, il ne doit y avoir qu’une seule et unique fonction main dans la liste des fonctions mains (sinon, une erreur survient). Il suffit ensuite de commencer de la fonction main et de faire un parcours en profondeur récursif.

Attention, deux cas particuliers sont à prévoir :

* Il faut ignorer les méthodes abstraites.
* Il faut gérer les appels récursifs dans le parcours en profondeur. Si l’élément, lors du parcours en profondeur, est déjà dans la stacktrace, alors il doit être ajouté dans la stacktrace et cette dernière doit être formée (et donc, il faut considérer que cela forme une exécution complète). Il faut ensuite continuer avec les prochains éléments…

En considérant la classe ci-dessous, la stacktrace est :

« main, a, a, r, r, r, x, main, a, c, r, r, r, x, main, b, b, r, r, r, x »

public class Test {

public static void main(String args[]) {

a();

b();

}

public static void a() {

a();

c();

}

public static void b() {

b();

}

public static void c() {

…

}

}

## Sequitur