Projet d’Introduction à la Vérification Formelle - Documentation

[Architecture générale du projet 2](#_Toc506757739)

[Instructions d’utilisation du code 2](#_Toc506757740)

[Limitations adoptées 2](#_Toc506757741)

[Parseur 2](#_Toc506757742)

[Génération de test 2](#_Toc506757743)

[Critère tous les DU-chemins 3](#_Toc506757744)

[Critères toutes les conditions/ toutes les décisions 3](#_Toc506757745)

[Principaux choix de conception 3](#_Toc506757746)

[Classe : graphe de contrôle 3](#_Toc506757747)

[Plusieurs programmes pour vérifier tous les critères de couverture 4](#_Toc506757748)

[Génération de sets de test 6](#_Toc506757749)

[Critères de couverture : mise en place et relations 6](#_Toc506757750)

[Toutes les affectations 6](#_Toc506757751)

[Toutes les décisions 6](#_Toc506757752)

[Tous les k-chemins 7](#_Toc506757753)

[Toutes les i-boucles 7](#_Toc506757754)

[Toutes les définitions 9](#_Toc506757755)

[Toutes les utilisations 10](#_Toc506757756)

[Tous les DU-chemins 11](#_Toc506757757)

[Toutes les conditions 12](#_Toc506757758)

[Relations entre les critères 14](#_Toc506757759)

*08/02/2018*

*Solène Duchamp - Charles Jacquet*

# Architecture générale du projet

├── main.py : module pour lancer la vérification de tous les programmes

│

├── pgrm1.txt : fichier contenant le programme 1

├── pgrm2.txt : fichier contenant le programme 2

├── pgrm3.txt : fichier contenant le programme 3

│

├── model\_graph.py : module contenant la classe graphe\_controle()

│

├── programme\_1.py : module testant les critères sur le programme donné dans le sujet

├── programme\_2.py: module testant les critères sur un programme composé d’une boucle

│ simple

├── programme\_3.py: module testant les critères sur un programme composé d’une boucle

│ while suivie d’une décision IF.

│

└── test\_set\_generation.py: module contenant une classe de générateur de test sets

# Instructions d’utilisation du code

Il suffit de lancer le module **main.py**

A l’intérieur, les 3 fonctions test\_programme\_X acceptent des jeux de test en entrée pour faire des tests personnalisés.

# Limitations adoptées

## Parseur

Nous avons décidé de ne pas créer de parser de code source mais plutôt d’assigner directement à chaque arête une instruction du programme.

## Génération de test

D’autre part, nous n’avons pas eu le temps de créer un outil de génération de tests généralisé.

Par conséquent nous avons hardcodé pour le programme 1 et les critères “toutes les décisions” et “toutes les affectations” la résolution du problème, ce qui nous permet tout de même de trouver un test set minimal pour ces deux critères.

Si nous avions eu plus de temps, nous aurions réutilisé l'algorithme de parcours de graphe afin de récupérer les fonctions d’affectation présentes sur les arêtes et les aurions ajoutées comme contraintes dans le système de résolution de problème linéaire avec contraintes.

## Critère tous les DU-chemins

Dans ce critère, nous avons considéré que dans tous les cas, nous ne devions pas parcourir les boucles plus d’une fois (même si la boucle est avant ou après le couple D-U).

## Critères toutes les conditions/ toutes les décisions

Dans notre modélisation, si une expression booléenne est une union ou intersection de conditions portant sur différentes variables, nous séparons cette expression en conditions élémentaires que nous assignons ensuite à différentes arêtes. Par conséquent les critères toutes les conditions et toutes les décisions sont très similaires.

# Principaux choix de conception

## Classe : graphe de contrôle

Module : **Model\_graph.py**

Nous avons créé une classe graphe\_controle() qui modélise un graphe de contrôle pour un programme donné. Cette classe se base sur la classe DiGraph() de la librairie networkx.

Nous avons créé plusieurs méthodes afin d’ajouter les opérations (affectation ou décision) au graphe et d’effectuer les tests sur ce graphe. Nous avons décidé de dissocier les fonctions d’ajout d’arêtes d’affectation et de d’arêtes de décision afin de nous faciliter la tâche pour les tests.

Fonctions de création

* Add\_variables : Fonction permettant d’ajouter les variables utilisées dans ce programme
* Add\_arete\_decision : Fonction permettant d’ajouter des fonctions de décision sur les arêtes
* Add\_arete\_affectation : Fonction permettant d’ajouter des fonctions d’affectation sur les arêtes
* Is\_loop : Fonction renvoyant un booléen en fonction de l’existence de cycles dans ce graphe

Fonctions support

* Def\_function : Fonction renvoyant les variables qui sont définies sur les arêtes sortantes de u
* Ref\_function : Fonction renvoyant les variables qui sont utilisées sur les arêtes sortantes de u
* Parcourir : Fonction permettant de parcourir le graphe, en fonction d'un dictionnaire d’état initial
* Travel\_with\_path : Idem que parcourir mais renvoyant une liste de string représentant le chemin et non une liste de listes de tuples
* Parcourir\_boolean : Fonction permettant de parcourir le graphe, en fonction d'un état initial et de renvoyer les arêtes de décision évaluées à vrai et à faux
* Parcours\_tous\_chemins (j) : Parcours tous les chemins partant du nœud racine jusqu'au nœud final. Le chemin contiendra exactement j tours de chaque boucle (s'il y en a).
* Parcours\_tous\_chemins\_string (j) : Idem que parcours\_tous\_chemins mais renvoie une liste de string représentant les chemins et non une liste de listes de tuples
* Nodes\_between : Fonction donnant les chemins partiels entre u et v
* Chemins\_partiels : Idem que nodes\_between mais avec u et v dans le chemin
* Show\_graph : Affiche le graphe dans une nouvelle fenêtre

Fonctions critères

* Toutes\_affectations : Fonction vérifiant le critère "toutes les affectations”
* Toutes\_decisions : Fonction vérifiant le critère "toutes les décisions"
* Toutes\_boucles : Fonction vérifiant le critère "toutes les i-boucles"
* Tous\_k\_chemins : Fonction vérifiant le critère "toutes les k-chemins"
* Toutes\_les\_def : Fonction vérifiant le critère "toutes les définitions
* Toutes\_les\_utilisations : Fonction vérifiant le critère "toutes les utilisations"
* Tous\_les\_DU\_chemins : Fonction vérifiant le critère "tous les DU-chemins"
* Toutes\_les\_conditions : Fonction vérifiant le critère "toutes les conditions"

Avoir une classe nous permet de réutiliser de manière efficace les fonctions pour à la fois créer un graphe de contrôle pour chaque programme, mais aussi pour vérifier tous les critères de couverture.

## Plusieurs programmes pour vérifier tous les critères de couverture

Comme nous ne parsons pas le code de nos programmes, mais que nous ajoutons directement les instructions sur les arêtes de nos graphes de contrôle, nous avons besoin d’un module par programme. Dans chacun des modules, nous testons tous les critères de couvertures suivants:

Nous testons pour ce programme les critères suivants:

* Toutes les affectations
* Toutes les décisions
* Tous les k-chemins
* Toutes les i-boucles
* Toutes les définitions
* Toutes les utilisations
* Tous les DU-chemins
* Toutes les conditions

Programme 1:

|  |
| --- |
| 1 : if X <= 0  then 2 : X := -X  else 3 : X = 1 - X;  4 : if X = 1  then 5 : X := 1  else 6 : X = X + 1 |

Programme 2:

|  |
| --- |
| while X <= 0 :  then 2 : X := X + 1  3 : then : return X |

Programme 3:

|  |
| --- |
| 1 : while X <= 0 :  then 2 :  if Y <= 0 :  then 3 : X := X - Y  else 4 : X := X + Y  5 : then : return X, Y |

## Génération de sets de test

Le module **test\_set\_generation.py** a pour but de trouver des sets de test pour chaque programme et chaque critère de couverture.

Pour trouver des valeurs de test pour nos valuations de départ, nous utilisons dans ce module la librairie **constraint** qui permet de trouver des solutions à des problèmes de programmation linéaire avec contraintes.

# Critères de couverture : mise en place et relations

## Toutes les affectations

Pour toutes les affectations, nous récupérons d’abord toutes les arêtes d’affectations.

Puis nous parcourons le graphe avec notre jeu de valuations de test.

Une fois tous les chemins récupérés, nous vérifions si toutes les arêtes d’affectations sont bien dans l’ensemble des arêtes visitées. Si ce n’est pas le cas, la fonction renvoie un pourcentage de couverture.

**Prog 1**

|  |  |  |
| --- | --- | --- |
| Jeu de test | Couverture | Chemin(s) manquant(s) |
| [{'x': -1}, {‘x’: 1}] | 100% |  |

## Toutes les décisions

## 

Pour toutes les décisions, nous récupérons d’abord toutes les arêtes de décision.

Puis nous parcourons le graphe avec notre jeu de valuations de test.

Une fois tous les chemins récupérés, nous vérifions si toutes les arêtes de décision sont bien dans l’ensemble des arêtes visitées. Si ce n’est pas le cas, la fonction renvoie un pourcentage de couverture et les arêtes manquantes.

**Prog 1**

|  |  |  |
| --- | --- | --- |
| Jeu de test | Couverture | Chemin(s) manquant(s) |
| [{'x': -1}, {‘x’: 1}] | 100% |  |

## Tous les k-chemins

Pour tous les k-chemins, nous récupérons tous les chemins possibles grâce à la fonction Parcourir().

Puis nous parcourons le graphe avec notre jeu de valuations de test.

Une fois les tests faits, nous vérifions que tous les chemins possibles font partie des chemins parcourus. Si ce n’est pas le cas, la fonction renvoie un pourcentage de couverture et les chemins manquants.

**Prog 1**

k = 2

|  |  |  |
| --- | --- | --- |
| Jeu de test | Couverture | Chemin(s) manquant(s) |
| [{'x': -1}, {‘x’: 1}] | 100% |  |

k = 4 (tous les chemins)

|  |  |  |
| --- | --- | --- |
| Jeu de test | Couverture | Chemin(s) manquant(s) |
| [{'x': -1}, {‘x’: 1}] | 50% | [[(1, 3), (3, 4), (4, 5), (5, 7)], [(1, 2), (2, 4), (4, 6), (6, 7)]] |
| [{'x': -9}, {'x': -1}, {'x': 1}] | 75 % | [(1, 3), (3, 4), (4, 5), (5, 7)] (°) |

(°) Explication : pour avoir x = 1 (arête (4, 5)), avec x = 1-x (arête (3, 4)), alors x doit être <= 0, or l’arête (1, 3) oblige x > 0. Le chemin est impossible à parcourir.

## Toutes les i-boucles

Pour les k compris entre 1 et i, on génère les chemins faisant respectivement k tours de boucles grâce à parcourir.

Puis on vérifie que nous passons bien par ces chemins avec le jeu de test.

**Prog 1**

Il n’y a pas de boucle, on prend donc i= 1 et le critère correspond alors à parcourir tous les chemins (ci-dessus).

**Prog 2**

i=1

|  |  |  |
| --- | --- | --- |
| Jeu de test | Couverture | Chemin(s) manquant(s) |
| [{'x': 0}, {‘x’: 1}] | 100% |  |
| [{'x': -1}, {‘x’: 1}] | 50% | [[(1, 2), (2, 1), (1, 3)]] |

i=2

|  |  |  |
| --- | --- | --- |
| Jeu de test | Couverture | Chemin(s) manquant(s) |
| [{'x': 0}, {‘x’: 1}] | 67% | [[(1, 2), (2, 1), (1, 2), (2, 1), (1, 3)]] |
| [{'x': -1}, {‘x’: 0}, {‘x’: 1}] | 100% |  |

**Prog 3**

i=1

|  |  |  |
| --- | --- | --- |
| Jeu de test | Couverture | Chemin(s) manquant(s) |
| [{'x': -4, 'y': -2},  {'x': -4, 'y': 2}] | 33% | [[(1, 2), (2, 4), (4, 1), (1, 5)],  [(1, 2), (2, 3), (3, 1), (1, 5)]] |
| [{'x': 4, 'y': -2},  {'x': -2, 'y': 4},  {'x': -2, 'y': -4}] | 100% |  |

i=2

|  |  |  |
| --- | --- | --- |
| Jeu de test | Couverture | Chemin(s) manquant(s) |
| [{'x': 4, 'y': -2},  {'x': -2, 'y': 4},  {'x': -2, 'y': -4}] | 67% | [[(1, 2), (2, 4), (4, 1), (1, 2), (2, 4), (4, 1), (1, 5)],  [(1, 2), (2, 3), (3, 1), (1, 2), (2, 3), (3, 1), (1, 5)]] |
| [{'x': 4, 'y': -2},  {'x': -2, 'y': 4},  {'x': -2, 'y': 2},  {'x': -2, 'y': -2},  {'x': -2, 'y': -4}] | 100% |  |

## Toutes les définitions

Il s’agit de vérifier que nous utilisons au moins une fois toutes les variables que nous définissons.

Nous considérons alors que le nœud 1 définie toujours toutes les variables (puisqu’on lui donne un dictionnaire d’état (une autre manière de faire aurait été de rajouter un nœud 0), et le nœud de fin utilise systématiquement toutes les variables (car il renvoie le dictionnaire d’état).

Pour cela :

- Nous récupérons pour chaque variable var les nœuds node 1 tel que var appartient à def(node 1), et les nœuds node 2 tels que var appartient à ref(node 2)

- Pour tous les nœuds de type node 1, on va chercher une donnée du jeu de test passant par node 1 puis node 2, avec aucun nœud de type node 1 entre les deux nœuds (pas de redéfinition de var)

- Une fois toutes les variables parcourues, on regarde s’il reste des nœuds de type node 1 qui n’ont pas pu être utilisés.

**Prog 1**

|  |  |  |
| --- | --- | --- |
| Jeu de test | Couverture | Noeud(s) manquant(s) |
| [{'x': 1}] | 60% | {'x': [2, 5]} |
| [{'x': -1}, {'x': 1}] | 100% |  |

**Prog 2**

|  |  |  |
| --- | --- | --- |
| Jeu de test | Couverture | Noeud(s) manquant(s) |
| [{'x': 1}] | 50% | {'x': [2]} |
| [{'x': 0}] | 100% |  |

**Prog 3**

|  |  |  |
| --- | --- | --- |
| Jeu de test | Couverture | Noeud(s) manquant(s) |
| [{'x': -4, 'y': 4}] | 75% | {'x': [3], 'y': []} |
| [{'x': -4, 'y': -2},  {'x': -4, 'y': 4}] | 100% |  |

## Toutes les utilisations

Pour ce critère, nous considérons également que le nœud 1 définie toutes les variables du graphe, et que le dernier nœud du graphe est un nœud d’utilisation de chaque variable.

Ici, nous cherchons à vérifier que toutes les utilisations accessibles après une définition sont exécutées au moins une fois.

Pour cela :

- Pour chaque variable var les nœuds node 1 tel que var appartient à def(node 1), et les nœuds node 2 tels que var appartient à ref(node 2)

- Pour chaque couple (u, v) nous stockons tous les chemins de u à v sans redéfinition de var

- Puis pour chaque chemin entre u et v, nous vérifions que nous avons bien une de nos donnée test qui passe par ce chemin

**Prog 1**

|  |  |  |
| --- | --- | --- |
| Jeu de test | Couverture | Chemin(s) manquant(s) |
| [{'x': -1}, {'x': 1}] | 88% | {'x': [(2, 6)]} |
| [{'x': 1}, {'x': -1}, {'x': -2}] | 100% |  |

**Prog 2**

|  |  |  |
| --- | --- | --- |
| Jeu de test | Couverture | Chemin(s) manquant(s) |
| [{'x': 1}] | 33% | {'x': [(1, 2), (2, 1)]} |
| [{'x': 0}] | 100% |  |

**Prog 3**

|  |  |  |
| --- | --- | --- |
| Jeu de test | Couverture | Chemin(s) manquant(s) |
| [{'x': -4, 'y': 4}] | 67% | {'x': [(1, 3), (3, 1)], 'y': [(1, 3)]} |
| [{'x': -4, 'y': 4},  {'x': -4, 'y': -2}] | 100% |  |

## Tous les DU-chemins

Pour ce critère, pour chaque variable, on considère tous les couples définition-utilisation (node1, node2). Pour chaque couple, nous allons chercher à savoir si tous les chemins simples sans redéfinition de var sont exécutés

De la même manière que les critères précédents, on commence par stocker pour chaque variable var les nœuds de type node 1 où var est définie et les nœuds de type node 2 où var est utilisée.

Puis :

- On stocke les chemins simples partiels entre u et v

- On enlève ceux où var est redéfinie

- On génère les chemins produits par le jeu de test

Et on vérifie que pour chaque chemin partiel :

- Il existe une donnée de test passant par ce chemin, sans être passé plusieurs fois par une boucle intermédiaire

**Prog 1**

|  |  |  |
| --- | --- | --- |
| Jeu de test | Couverture | Chemin(s) manquant(s) |
| [{'x': -1}, {'x': 1}] | 88% | {'x': ['246']} |
| [{'x': 1}, {'x': -1}, {'x': -2}] | 100% |  |

**Prog 2**

|  |  |  |
| --- | --- | --- |
| Jeu de test | Couverture | Chemin(s) manquant(s) |
| [{'x': 0}] | 50% | {'x': ['13']} |
| [{'x': 1}, {'x': 0}] | 100% |  |

**Prog 3**

|  |  |  |
| --- | --- | --- |
| Jeu de test | Couverture | Chemin(s) manquant(s) |
| [{'x': -4, 'y': 4},  {'x': -2, 'y': -4}] | 33% | {'x': ['15', '41'], 'y': ['12415', '15']} |
| [{'x': -4, 'y': 5},  {'x': -2, 'y': -4},  {'x': 2, 'y': 4} ] | 100% |  |

## Toutes les conditions

Ce critère cherche à vérifier que chaque arête de décision a été exécutée au moins une fois à vrai et une fois à faux.

Pour cela, on utilise la fonction parcourir\_boolean, qui, à partir d’un dictionnaire d’état initial, nous renvoie les arêtes de décision exécutées à vrai et celles exécutées à faux.

Notons que nos graphes sont construits tels que chaque arête de décision est une condition (c’est-à-dire une expression élémentaire), nous n’avons donc pas à parser la fonction de l’arête.

Ainsi, pour chaque donnée test, nous récupérons les arêtes de décision exécutées à vrai et celles exécutées à faux.

Puis, nous comparons ces deux listes avec la liste des arêtes de décision.

**Prog 1**

|  |  |  |
| --- | --- | --- |
| Jeu de test | Couverture | Arêtes manquantes |
| [{'x': 1}] | 50% | Non évaluée(s) à vrai : {(1, 2), (4, 5)}  Non évaluée(s) à faux : {(1, 3), (4, 6)} |
| [{'x': -1}, {'x': 1}] | 100% |  |

**Prog 2**

|  |  |  |
| --- | --- | --- |
| Jeu de test | Couverture | Chemin(s) manquant(s) |
| [{'x': 1}] | 50% | Non évaluée(s) à vrai : {(1, 2)}  Non évaluée(s) à faux : {(1, 3)} |
| [{'x': 0}] | 100% |  |

**Prog 3**

|  |  |  |
| --- | --- | --- |
| Jeu de test | Couverture | Chemin(s) manquant(s) |
| [{'x': -4, 'y': 4}] | 75% | Non évaluée(s) à vrai : {(2, 3)}  Non évaluée(s) à faux : {(2, 4)} |
| [{'x': -4, 'y': 4},  {'x': -4, 'y': -2}] | 100% |  |

# Relations entre les critères

