EXÉCUTION SYMBOLIQUE ET TESTS STRUCTURELS

Projet - Introduction à la Vérification Formelle

Arnaud Convers

Janvier - Février 2018

ARCHITECTURE GENERALE

Pour l'ensemble des présentations et explications effectuées ici, le programme *Prog* proposé dans le sujet sera utilisé comme exemple.

Pour rappel: programme *Prog:*

```
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
```

Composition du dossier rendu

- Fichiers:
  - **rapport_convers.pdf**
  - **README.md** : un condensé du rapport, en anglais
  - **my_parser.py** : module contenant un ensemble de fonctions (non terminé) pour la tokenization et le parsing des programmes donnés en entrée.
  - **ast_tree.py** : ce module contient deux classes : la classe Node, décrivant la structure d'un abstract syntax tree et la classe GeneratorAstTree, utilisée pour écrire et programmer directement les programmes étudiés dans la structure AST.
  - **process_cfg_tools.py** : ensemble de functions - outils utilisées pour effectuer l'analyse de couverture de tests. Contient notamment la fonction principale process_value_test(), qui prend un entrée un CFG et des valeurs pour les variables du programme, parcourant le graphe à partir de ces variables et retournant les valeurs finales ainsi que le chemin parcouru.
  - **analysis_coverage.py** : ensemble de fonctions pour effectuer chacunes des analyses de couvertures, selon les critères demandés. Contient notamment la fonction principale calc_coverage(), qui pour un CFG et un ensemble de données de test, calcule la couverture des tests selons les différents critères.

- **symbolic_exec_tools.py** : ensemble de fonctions - outils utilisés pour effectuer la génération de tests. Contient notamment la fonction solve_path_predicate(), qui appelle le solveur python-constraint à partir d'un prédicat sur un chemin d'exécution.
- **generator.py** : module pour la génération de tests. Contient notamment la fonction principale generate_sets_tests() qui prend en argument un CFG et concatène les résultats de génération de test.
- **unit_tests.py** : quelques classes de unit tests pour s'assurer du fonctionnement au cours du développement de l'ensemble des modules écrits.
- **demo.py** : un fichier très simple, avec une fonction main permettant d'effectuer une analyse de couverture ainsi qu'une génération de tests pour deux programmes différents.

- Dossiers :
  - **sources_txt** : répertoire utilisé pour stocker les fichiers écrits en langage while, sur lesquels des tests de couverture ainsi que des générations de tests vont être effectué.
  - **sets_tests_txt** : répertoire utilisé pour stocker les fichiers de données de tests utilisés pour l'analyse de couverture.
  - **generated_tests** : répertoire destiné au stockage des fichiers de tests générés

Représentation des structures de données utilisées

- Abstract Syntax Tree

L'AST utilisé dans ce projet est un structure d'arbre très simple. Chaque noeud est une instance de la classe Node, ayant les propriétés suivantes :

- **Category** : la catégorie de la commande/type représentée par le noeud. Les catégories peuvent être: sequence, if, variable, constant, operation, assign, compare, while, logic.
- **Data** : la donnée portée par le noeud (s'applique pour les catégories : constant, variable, compare, operation)
- **Children** : la liste des enfants du noeud.

- Control Flow Graph

Le CFG utilisé dans ce projet est également codé assez simplement, en utilisant un dictionnaire python.

Les clefs sont les numéros des noeuds.

Les valeurs sont des listes qui contiennent les informations sur le noeud ainsi que sur les arêtes sortantes du noeud.

Composition de la liste descriptive d'un noeud :

- La première valeur est le type de la commande. Elle peut être : "assign", "while", "if" or "skip".

La liste des valeurs suivantes dépend du type de noeud.

- **noeuds "if" et "while":**
  - La seconde valeur est l'expression booléenne sur le noeud, écrit dans un format prédéfini, en Forme Normale Conjonctive : il s'agit d'une liste, avec une relation AND entre chaque élément ; chaque élément est également une liste de tuples séparés par une relation OR. Chaque tuple représente une condition primitive (type a < b)
    - tuple[0] est l'opérateur ("<=", ..., ">=")
    - tuple[1] est une liste de longueur 2, l'opérateur opérant entre les deux éléments.

    Par exemple, l'expression

    ```
    ( x <= 0 ) and ( ( y == 3 ) or ( y < 0 ) )
    ```

    est représentée par

    ```
    [[('<=',['x',0])],[('==', ['y', 3]),('<',['y, 0])]]
    ```

  - La troisième valeur contient une liste de deux entiers. Ils représentent les deux noeuds suivants possible : Le premier si l'expression booléenne est vraie, le second sinon.
- **noeuds "assign":**
  - La seconde valeur est un dictionnaire {variable: nouvelle valeur},

par exemple {‘x’: ‘y+4’}
        - La troisième valeur est le numéro du noeud suivant
    - noeuds “skip”:
        - La seconde valeur est le numéro du noeud suivant.

NB: L’entier zéro est utilisé pour représenter l’absence de noeud suivant.

- Prédicats

Dans les phases de génération de tests, les prédicats sur des variables sont décrits sous la forme d’une liste. Par exemple :

```
['(x1 <= 0)', 'x2 = 0-x1', '(x4 == 1)']
```

est un prédicat, représentation des conditions sur la variables x aux étapes i données.

CHOIX DE CONCEPTION

Durant la rédaction de ce projet, la volonté a été d’écrire un maximum de choses par moi même et d’éviter au plus le recours à des librairies extérieures. Par conséquence, le code est parfois assez dense, même si d’importants efforts ont été faits pour augmenter sa concision et clarté.

I. Conversion d’un AST en CFG

Pour la conversion de AST à CFG, une classe ast_to_cfg a été créée. On l’instancie en lui passant un AST ; elle a une propriété step qui va servir à numéroter les différents noeuds du graphe. Cinq principales fonctions ont été définies : *treat_seq_node*, *treat_while_node*, *treat_if_node*, *treat_composed_boolean_expr*, *treat_assign_node*. Un AST étant toujours un node de type séquence, on traite de manière itérative chacun de ses enfants, selon son type et ce de manière récursive en appelant selon chaque cas l’une de ces fonction. Tout le long du traitement, la variable “step” de l’objet est incrémentée de façon à numéroter correctement chacun des noeuds du CFG.

Par exemple, le graphe de sortie du programme Prog est le suivant :

```
graph_prog = {
            1: ['if', [[('<=', ["x", 0])]], [2, 3]],
            2: ['assign', {'x': '0-x'}, [4]],
            3: ['assign', {'x': '1-x'}, [4]],
            4: ['if', [[('==', ["x", 1])]], [5, 6]],
```

```
        5: ['assign', {'x': '1'}, [0]],
        6: ['assign', {'x': 'x+1'}, [0]]
    }
```

II. Exécution d'une donnée de test sur un CFG

Pour pouvoir effectuer les analyses de couvertures de test, un ensemble de fonctions permettant de suivre l'évolution de variables selon les valeurs dans le graphe ont été mises en place. Pour cela, on utilise une variable de suivi prenant à chaque étape comme valeur le numéro du noeud actuel. L'exécution s'arrête lorsque la variable est à 0 (noeud de sortie) ou lorsque le nombre d'itération a dépassé un certain seuil (signe d'une boucle infinie dans le programme). A chaque étape, dans le cas d'une assignation, les valeurs des variables sont actualisées, et dans le cas d'une évaluation booléenne (commandes while et if), les valeurs des variables sont utilisées pour déterminer le noeud suivant. A la fin de l'exécution, les noeuds parcourus (chemin) ainsi que les valeurs des variables sont renvoyées.

III. Implémentation des critères

1) Critère "toutes les affectations"

L'implémentation de ce critère est relativement simple. Une première étape de parcours des différents noeuds du graphe est effectuée ; on stocke alors les noeuds à visiter (soit les noeuds de type "assign").

On process alors la valeur de test proposée sur le CFG du programme. A la sortie, on compare le chemin parcouru avec nos valeurs de test par l'objectif ; on peut ainsi en déduire la couverture ou non selon le critère TA.

2) Critère "toutes les décisions"

On procède de la même manière pour la vérification de ce critère. Les noeuds retenus sont les noeuds suivant un noeud de type "décision" (if ou while).

3) Critère "tous les k-chemins"

Dans le cas de ce type de critère, l'idée est de stocker une liste de chemins cibles, et de vérifier après exécution du programme si ces chemins ont été parcourus.

Pour choisir les chemins, on récupère l'ensemble des chemins possibles dans le graphe

de longueur inférieure ou égale à k.

4) Critère "toutes les i-boucles"

Pour ce critère, on définit une liste de noeuds cibles (premier noeuds après une boucle while si sa condition est vrai). Après avoir exécuté les valeurs tests sur le CFG, on vérifie ensuite les conditions sur la visite du noeud ainsi que sur le nombre de fois que le noeud a été visité.

5) Critère "toutes les définitions"

Pour ce critère, on définit une liste de couples d'étapes à visiter, correspondant à la définition et à l'utilisation d'une variable. Après l'exécution des valeurs, pour chaque couple de noeuds, on regarde s'il existe des chemins qui contiennent les deux noeuds du couple considéré.

6) Critère "toutes les utilisations"

La démarche suivi est proche de la précédente, la différence étant que l'on stocke une liste de chemins à parcourir et non une paire definition - utilisation.

7) Critère "tous les DU-chemins"

Pour le critère tous les DU-Chemins, on effectue un travail proche du critère TDef ; on garde également un dictionnaire sur les étapes à l'intérieur des boucles while, permettant de vérifier après l'exécution si les chemins de résultats sont bien des chemins simples.

8) Critère "toutes les conditions"

Pour le critère toutes les conditions, on récupère tout d'abord l'intégralité des conditions depuis l'ensemble des expressions booléennes du CFG. On parcourt ensuite le graphe en gardant des informations sur les conditions exécutées (si elles ont été évaluées à True ou à False). Pour valider la couverture, on vérifie simplement que l'ensemble des conditions a été évaluées à la fois à True et à False.

IV. Génération de données de test à partir d'un chemin

Pour générer l'ensemble des tests, on se réfère toujours à un chemin que l'on veut parcourir. Par exemple, si l'objectif est d'arriver au noeud N, on calcule d'abord le chemin pour arriver à N puis on travaille sur ce même chemin.

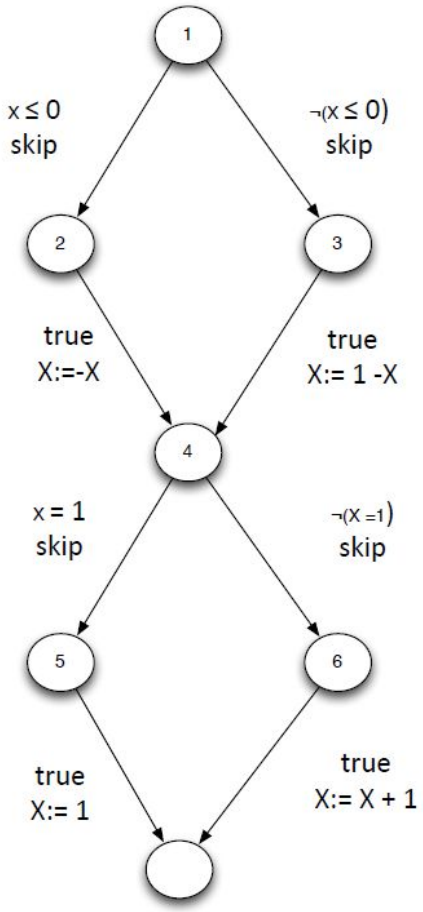
L'idée va être d'utiliser le solveur de contraintes python-constraint pour résoudre des

problèmes de programmation par contraintes définis depuis des prédicats que l'on va écrire à partir du chemin à parcourir.

Pour écrire un prédicat de chemin à partir d'un chemin donné, on parcoure dans le graphe les différentes étapes et on récupère les conditions ou assignations ; on numérote les variables avec le numéro du noeud, sauf dans le cas d'une assignation, où les variables référencées sont numérotées le numéro du noeud - 1, pour signifier le fait qu'on fait référence à la valeur de la variable définie précédemment.

Pour chaque variable x, on se retrouve ainsi avec un set de sous variables $\{x_1, ..., x_n\}$

Par exemple, pour le programme Prog :



Pour effectuer le chemin 1, 2, 4, 5, on a :

- étape 1: True
- étape 2: x <= 0
- étape 4: x = - x

On peut alors écrire le prédicat:

$T \wedge x_2 <= 0 \wedge x_4 == - x_3$

avec les conditions :

$x_2 == x_1$; $x_2 == x_3$

Le principe est donc de passer ces différentes contraintes au solveur, et de récupérer la ou les valeurs sur x_1 (valeur de la variable X à l'entrée du graphe).

V. Implémentation des critères

Par manque de temps et d'efficacité, la génération a été implémentée pour seulement trois critères :

1) Critère "Toutes les Affectations" et critère "Toutes les Décisions"

Comme lors de la vérification de couverture, on récupère d'abord la liste des noeuds à visiter. Pour chacun de ces noeuds, on calcule ensuite le chemin pour arriver jusqu'au noeud. A partir de ce chemin, on effectue la méthode décrite précédemment pour récupérer les valeurs. On concatène ensuite le résultat.

2) Critère "Tous les k-chemins"

De la même façon que dans la vérification de couverture, on récupère la liste des chemins à parcourir ; on effectue ensuite la méthode présentée ci dessus pour récupérer les valeurs des variables.

LIMITATIONS

Plusieurs limitations ont étées rencontrées durant la conception de cet outil, de même que plusieurs hypothèses limitantes ont été introduites. Voici une rapide présentation de ces limitations.

1. Le parser n'ayant pas du tout été priorisé il est loin d'être prêt à être utiliser. Il est donc nécessaire de coder directement les structures en AST pour pouvoir les convertir en CFG et effectuer les analyses de couvertures.
2. Dans l'écriture des données de tests, dans le fichiers contenant les données, on suppose que chacune des variables référencées ou même **assignée** dans le programme a une valeur dans ce fichier.
3. Dans la vérification formelle pour certains tests, aucune fonctions n'a été introduite pour vérifier que certains chemins n'étaient pas impossible, ce qui peut pour le moment provoquer certains tests (all k-paths, …) à être toujours faux, et à rendre impossible la génération de valeur pour ces chemins particuliers.
4. Dans le critère "Toutes les i-boucles" : On vérifie la condition pour chaque boucle while ; il n'a pas été tenu compte des cas de boucles imbriquées.
5. Dans la génération, du fait de la façon dont les contraintes sont construites et écrites (rajout d'une chiffre à la fin du nom de la variable, via manipulation direct de string), il est important que chaque nom de variable soit de longueur égale à 1.
6. Enfin, et du fait des limitations précédentes, le générateur n'est pas robuste ni

mature ; il n'a pas pu être testé sur beaucoup de cas et par conséquent risque de poser problèmes lors de génération sur des programmes un peu plus complexes que sur ceux testés.

PROCÉDURES D'UTILISATION DU CODES

1. Prérequis

Le projet n'utilise pas de bibliothèques extérieures mis à part la bibliothèque python constraint. L'ensemble du projet est codé dans la version 3.6 de python.

```
$ pip install git+https://github.com/python-constraint/python-constraint.git
```

2. Démonstration

Une rapide démonstration est proposée, pour l'analyse de la couverture de test ainsi que pour la génération. Elle est effectuée sur le programme Prog ainsi que sur un programme factoriel (codé de façon incrémentale).

Le programme Fact:

```
n := 1
while x >= 1
n := n * x
x := x - 1
```

Pour exécuter la démonstration, une seule commande est nécessaire, depuis la racine du projet :

```
$ python demo.py
```

3. Analyse de couverture

Pour réaliser l'analyse de couverture d'un programme donné avec un fichier de test donné, il faut avoir écrit le fichier de test correspondant. Il est structuré de la manière suivante :

- chaque ligne représente un ensemble de valeurs pour les variables du programme.
- Dans chaque ligne, chaque variable est associée à sa valeur initiale par le symbole ":" ; chaque couple variables/valeur initiale est séparé de l'autre via une virgule.

Par exemple, un fichier de tests de valeurs pour le programme Fact peut être:

```
x:1,n:1
x:2,n:1
x:3,n:1
x:4,n:1
x:5,n:1
x:6,n:1
x:7,n:1
```

Le parseur n'étant pas terminé, le fichier du programme doit pour l'instant surtout être représenté en tant qu'AST (instance de la classe ast_tree) dans la classe statique GeneratorAstTree. (voir partie 5, ajout d'un programme).

Pour réaliser la couverture d'un programme donné avec un ensemble de test donné, la commande est alors :

```
$ python analysis_coverage.py path_prog.txt path_data_test.txt [-v]
```

Avec -v argument optionnel pour activer le mode "verbose".

4. Génération

Pour la génération, de même, le fichier du programme doit pour l'instant surtout être représenté en tant qu'AST (instance de la classe ast_tree) dans la classe statique GeneratorAstTree. Pour réaliser la génération de tests, la commande est alors :

```
$ python generator.py path_prog.txt
```

5. Utilisation des outils sur un nouveau programme.

Le dossier proposé vient avec deux programmes analysés : Prog et Factoriel. Néanmoins, pour pouvoir utiliser les outils proposés sur de nouveaux programmes, le parser n'étant pas terminé, il faut les coder à la main dans une structure d'AST puis les rajouter manuellement dans la liste des AST disponibles via la fonction get_ast_from_name() dans le module ast_tree.py.

Par exemple, pour ajouter le programme Prog est codé en AST dans une fonction prog_tree() à la ligne 245 du module ast_tree.py. Il est ensuite référencé par la méthode get_ast_from_name() de la manière suivante :

```
if name == "prog_1":
      return GeneratorAstTree.prog_tree()
```

Le nouvel arbre peut ensuite être appelé par son nom depuis la ligne de commande :

```
$ python analysis_coverage.py nom_ast_tree path_data_test.txt [-v]
```

ou

```
$ python generator.py nom_ast_tree
```

RÉSULTATS SUR PROG ET FACT

Dans cette sections, quelques résultats d'analyse sur les programmes Prog1 et Fact sont présentés, avec respectivement les ensembles de tests suivants :

x:-3 x:-2 x:-1 x:0 x:1 x:2 x:3	x:1,n:1 x:2,n:1 x:3,n:1 x:4,n:1 x:5,n:1 x:6,n:1 x:7,n:1

Outputs (Couverture)

Pour Prog	Pour Fact
Starting analysis... ------ Criterion: all affectations We want the following nodes to be visited: [2, 3, 5, 6] TA: OK Coverage: 100% ------ Criterion: all decisions We want the following nodes to be visited: [1, 2, 3, 4, 5, 6] TD: OK Coverage: 100%	Starting analysis... ------ Criterion: all affectations We want the following nodes to be visited: [1, 3, 4] TA: OK Coverage: 100% ------ Criterion: all decisions We want the following nodes to be visited: [2, 3, 0]

```
 ------
Criterion: all k paths for k = 4
We want the following paths to be
taken: [[1, 3, 4, 5], [1, 2, 4, 6],
[1, 3, 4, 6], [1, 2, 4, 5]]
All k paths for k = 4 fails:
Paths [[1, 3, 4, 5]] were never
taken entirely.
Coverage: 75.0 %.

 ------
Criterion: all i loops
We want the following nodes [] to be
visited. (At must 2 times.)
2-TB: OK
Coverage: 100 %

 ------
Criterion: all definitions
for following variables, we want the
corresponding path to be taken:
{'x': [1, 2, 3, 4, 5, 6]}
TDef: OK
Coverage: 100 %

 ------
Criterion: all utilization
TU: Ok
Coverage: 100 %

 ------
Criterion: all du-paths
TDU: OK
Coverage: 100%

 ------
Criterion: all conditions
TC: OK
Coverage: 100%
End analysis coverage.
Tests are passing 7 criterion on 8
```

```
TD: OK
Coverage: 100%

 ------
Criterion: all k paths for k = 4
We want the following paths to be
taken: [[1, 2, 3, 4], [1, 2, 0]]
All k paths for k = 4: OK.
Coverage 100%

 ------
Criterion: all i loops
We want the following nodes [3] to
be visited. (At must 2 times.)
2-TB: OK
Coverage: 100 %

 ------
Criterion: all definitions
for following variables, we want the
corresponding path to be taken:
{'n': [1, 3], 'x': [2, 4]}
TDef: OK
Coverage: 100 %

 ------
Criterion: all utilization
TU: Ok
Coverage: 100 %

 ------
Criterion: all du-paths
TDU: OK
Coverage: 100%

 ------
Criterion: all conditions
TC: fails
Following conditions were not
evaluated entirely: ["[('>=', ['x',
1])]"]
coverage: 50.0%
```

	End analysis coverage. Tests are passing 7 criterion on 8

Quelques commentaires :

- Le critère tous les k-chemins ne sera jamais respecté pour: en effet, ici le chemin bloquant est un chemin impossible (voir limitation 3)
- Sur la couverture du programme Prog, on remarque que le critère TC est annoncé comme non respecté alors qu'il l'est. Ceci est dû au mauvais comportement de l'outil écrit pour l'exécution d'un programme ; il s'arrête lorsque le noeud suivant est le noeud de sortie. Hors, dès que la condition concernée dans Fact est évaluée à fausse, le noeud suivant est le noeud 0. La condition ('>=', ['x', 1]) est ainsi considérée comme n'étant jamais à False. On pourrait contourner ce problème en arrêtant le programme lorsque le noeud est 0, et non pas lorsque le noeud suivant est zéro.