Projet : Exécution symbolique et tests structurels

Ayaz Badouraly & Venceslas Danguy des Déserts

Enseignante : Mme Pascale Le Gall

Table des matières

Chapitre 1

Architecture du projet

1.1 Pipe de traitement

Le code se divise en trois grandes parties : parsing du programme dans le langage WHILE, vérification de critères sur ce programme, génération des tests pour les critères.

1.1.1 Parsing

Le parsing consiste, à partir d'un programme écrit en langage WHILE, à obtenir un graphe de contrôle (CFG) représentant le dit programme. On passe pour cela par une étape intermédiaire : un arbre de syntaxe abstraite, ou AST. L'AST est une représentation du code du programme sous forme d'arbre, tandis que le CFG représente vraiment le flux du programme.

Génération de l'AST

Nous utilisons pour ce faire la librairie Python ply, qui fournit une implémentation en Python du duo lex/yacc. Lex tokenize le programme en entrée en fonction de la grammaire définie, tandis que yacc prend l'entrée tokenizée et fournit l'AST. Nous utilisons la librairie *anytree* pour manipuler les arbres.

Conversion de l'AST au CFG

Il faut ensuite traduire cet AST en CFG. On utilise pour cela un algorithme récursif qui va parcourir l'arbre et construire le CFG. Chaque exécution de la fonction récursive renvoie un morceau de graphe ainsi que des arêtes sortantes à relier.

Ainsi, la conversion d'un arbre ne contenant en tout et pour tout qu'une assignation va renvoyer un noeud labellisé et une arête sortante de ce noeud portant une condition *True* et une commande *Assign*.

La conversion d'un arbre contenant un noeud While va relier les arêtes sortantes du graphe du sous-arbre au noeud parent de façon à former une boucle, etc.

On ajoute enfin un tag sur les noeuds correspondant à des instructions *If* ou *While* pour faciliter la vérification et la génération des tests.

Les graphes sont gérés par la librairie python *networkx*.

1.1.2 Vérification de critères

Nous sommes donc désormais capables d'obtenir le CFG associé à un programme écrit en langage WHILE. Nous voulons maintenant *vérifier* des critères de tests pour ce programme, c'est à dire, étant donné un jeu de tests (données de départs), dire si les tests couvrent un ensemble de cas (chemins d'exécution) précis.

Il faut donc commencer par exécuter le programme avec le jeu de test. Ceci se fait assez facilement, il suffit de parcourir le graphe de contrôle en maintenant un état (ensemble des variables et de leurs valeurs) et en choisissant, à chaque branchement, l'arête dont la condition est vérifiée par l'état.

On obtient ainsi un ensemble de chemins d'exécution (un par valuation de départ).

Parallèlement, on a un critère de test, c'est à dire un ensemble de chemins par lesquels nous voulons passer. Ainsi le critère *Toutes les affectations* demande-t-il qu'à la fin de l'exécution de tous les tests, on soit passé au moins une fois par chaque arête portant une commande *Assign*.

Chaque critère possède son propre algorithme de génération d'éléments à vérifier en fonction du programme (bien que l'on puisse toujours se ramener à un ensemble de chemins, il était parfois plus simple de considérer un ensemble de noeuds ou d'arêtes par lesquels il faut passer).

Il suffit ensuite de faire l'intersection des deux ensembles *{Chemins exécutés lors de la phase du test}* et *{Chemins à vérifier imposés par le critère}*.

1.2 Choix de conception

1.3 Limitations adoptées

1.4 Organisation du code

Le code est organisé en modules.

- **astree :** Contient notre structure d'AST avec les commandes et opérations définies par la sémantique opérationnelle de WHILE.
- **cfgraph :** Code utilisé pour exécuter un test sur le programme. Le module utils, en particulier, contient toutes les fonctions du graphe nécessaires pour générer les éléments à vérifier pour chaque critère.
- **syntax :** Contient le code du lexeur et du parseur ainsi que leur configuration
- **tests :** Ce dossier contient le coeur du projet. Le module *testor* contient la logique de vérification des critères : récupération des éléments à vérifier, analyse des chemins d'exécutions des tests pour y trouver les éléments. Le module *generator* gère la génération des jeux de tests en faisant appel au solveur *Z3*.
- **utils :** Contient notamment la conversion de l'AST vers le CFG.
- **input :** Contient des exemples de programmes sur lesquels on peut vérifier les critères.
- **output :** Exemples de graphes (images).

Chapitre 2

Vérification des critères

2.1 Toutes les affectations

On va travailler sur le programme suivant :

```
1:  if (X <= 0) {
2:      X := -X;
    } else {
3:      X := 1 - X;
    }

4:  if (X == 1) {
5:      X := 1;
    } else {
6:      X := X + 1;
    }
```

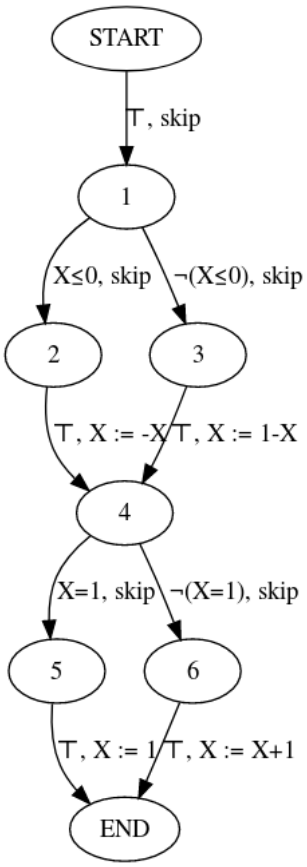

FIGURE 2.1 – CFG associé

Le critère *Toutes les affectations* demande à ce que tous les labels d'affectations (ici : $2, 3, 5, 6$) apparaissent au moins une fois dans l'un des chemins d'exécution correspondant aux données de tests.

Si on prend pour jeu de test :

$$\{\{X : -1\}, \{X : 1\}\}$$

On obtient les chemins d'exécutions suivants :

$\{X : -1\}$ **:** $1, 2, 4, 5$

$\{X : 1\}$ **:** $1, 3, 4, 6$

On est donc passé au moins une fois par chaque label associé à une affectation : le critère est vérifié sur ce programme et ce jeu de test.

Si, en revanche, on s'était contenté d'un unique test $\{X : -1\}$, on ne serait pas passé par les noeuds 3 et 6, et le critère n'aurait pas été vérifié.

Dans la pratique, étant donné un CFG, on récupère l'ensemble des noeuds (labels) tels qu'une arête sortante de ce noeud porte une affectation. Ceci se fait très facilement en parcourant l'ensemble des arêtes (sans notion de chemin ici). Il suffit ensuite de regarder lesquels de ces noeuds sont (ou non) dans les chemins des tests.

2.2 Toutes les décisions

On travaille toujours sur le programme défini en 2.1.

Le critère *Toutes les décisions* demande à ce que l'on passe au moins une fois par toutes les arêtes associées à une expression booléenne. Il s'agit, en bref, de toutes les arêtes sortantes des noeuds *if* et *while*, ce qui est équivalent à dire qu'il faut passer par les noeuds de destination de ces arêtes.

Ici, il faut donc passer par les noeuds $2, 3, 5, 6$. On retombe dans ce cas très simple sur les même éléments que le critère *Toutes les affectations*, et le même jeu de tests vérifie donc les deux critères.

2.3 Tous les k-chemins

Ce critère demande que chaque chemin d'exécution de longueur inférieure ou égale à k soit exécuté.

On commence donc par implémenter un générateur qui va renvoyer tous les chemins correspondants. Pour ce faire, on réalise un parcours en profondeur du graphe, où l'on s'arête lorsqu'on à un chemin complet (noeud *END* atteint) ou lorsque l'on a atteint la longueur maximale k.

Une fois que l'on a l'ensemble des chemins à vérifier, il suffit de comparer cet ensemble avec celui des chemins exécutés par les tests.

Sur le programme de la partie 2.1, pour le critère *Tous les 10-chemins*, il faut vérifier les chemins suivants (en fait, tous, car on a pris k grand) :

$$\{[START, 1, 2, 4, 5, END],$$
$$[START, 1, 2, 4, 6, END],$$
$$[START, 1, 3, 4, 5, END],$$
$$[START, 1, 3, 4, 6, END]\}$$

On note que le chemin $[START, 1, 3, 4, 5, END]$ n'est pas faisable. Malheureusement, cela ne peut se détecter qu'à l'étape de génération des tests : d'où l'importance de signaler à l'utilisateur quels sont les chemins manquants afin qu'il puisse décider de lui-même si ou non la couverture peut-être améliorée en complétant le jeu de tests.

Le jeu de tests suivant permet d'atteindre la couverture maximale :

$$\{\{X : -1\}, \{X : 1\}; \{X : 3\}, \}$$

2.4 Toutes les i-boucles

On cherche ici tous les chemins pour lesquels les boucles while sont exécutées au plus i fois. Attention, les "compteurs" des boucles imbriquées internes doivent être remis à zéro à chaque itération de la boucle externe. Le critère peut aussi se formuler "tous les chemins pour lesquels les boucles whiles sont exécutées au plus i fois **à la suite**.

Sur le même principe que les k-chemins, on réalisé un générateur de chemins d'intérêt basé sur un parcours en profondeur. On maintient cette fois un compteur par boucle, et on veille à réinitialiser les compteurs des boucles imbriquées internes lorsque c'est nécessaire.

Il suffit ensuite de comparer les chemins obtenus avec ceux des tests.

Voici un programme comportant une boucle :

```
1:  if (x >= 0) {
2:      x := x + 1;
    } else {
3:      x := x - 1;
4:      y := 3;
5:      z := y + 1;
    }

6:  while (z <= 0) {
7:      z := x + 1;
    }

8:  print(x);
```

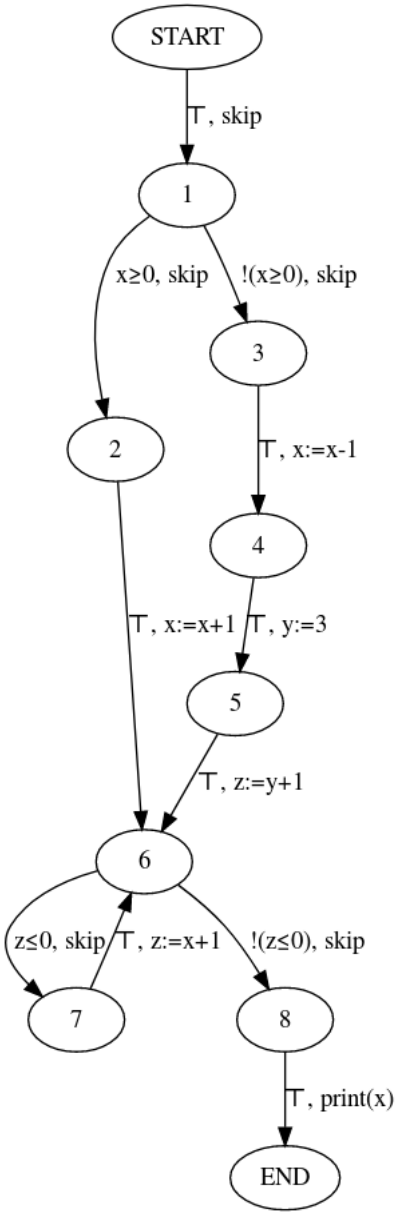

FIGURE 2.2 – CFG associé

Les chemins à parcourir pour *Toutes les 1-boucles* sont :

$$\{[START, 1, 2, 6, 8, END],$$
$$[START, 1, 2, 6, 7, 6, 8, END],$$
$$[START, 1, 3, 4, 5, 6, 8, END],$$
$$[START, 1, 3, 4, 5, 6, 7, 6, 8, END] -> Infaisable\}$$

Le jeu de test suivant vérifie le critère :

$$\{\{X:-1,Z:0\},\{X:0,Z:0\};\{X:0,Z:1\},\}$$

2.5 Toutes les définitions

Le critère *Toutes les définitions* demande que, pour chaque variable X, pour chaque noeud u affectant X, on ait un chemin $..u...v..$ tel que v utilise X, et qu'il n'y ait pas de réaffectation de X entre u et v.

Pour vérifier ce critère, nous commençons par implémenter les fonctions *def* et *ref* telles que définies dans l'énoncé.

On construit ensuite un dictionnaire *definitions* qui associe à chaque variable X l'ensemble des noeuds u tels que $X \in def(u)$.

Pour chaque variable X, on regarde ensuite chaque chemin. Si on trouve un noeud u intéressant, c'est à dire $X \in def(u)$, on suit le chemin jusqu'à trouver v tel que $X \in ref(v)$. On s'assure bien sûr qu'entre u et v, il n'y a pas de noeud u' tel que $X \in ref(u')$. Si on trouve v, on supprime u de *definitions*$[X]$.

Si à la fin, *definitons* est vide, alors le critère est vérifié.

Le programme de la partie 2.1 ne pourra jamais vérifier ce critère, quel que soit le jeu de test, car les affectations 5 et 6 ne sont jamais utilisées. Si on rajoute une commande print qui affiche une variable, ainsi qu'une nouvelle variable Y, on peut le réécrire :

```
1:  if (X <= 0) {
2:      X := -X;
3: Y := 5;
    } else {
4:      X := 1 - X;
    }

5:  if (X == 1) {
6:      X := 1;
    } else {
7:      X := X + 1;
8: print(Y);
    }
9: print(X);
```

Le dictionnaire *definitions* s'écrit alors :

$$\{X:[2,4,6,7],Y:[3]\}$$

Le jeu de tests suivant ne vérifie pas le critère :

$$\{\{X = 1, Y = 1\}, \{X = -1, Y = 1\}\}$$

(Pas de chemin entre 3 et 8)
Contrairement à cet autre jeu de tests :

$$\{\{X = 1, Y = 1\}, \{X = -2, Y = 1\}\}$$

2.6 Toutes les utilisations

Le critère *Toutes les utilisations* demande que toutes ls utilisations acessibles pour chaque définitions soient exécutées au moins une fois.

On crée en parcourant le graphe un dictionnaire *usages* qui pour chaque variable X, associe à chaque noeud u définissant X ($X \in def(u)$) l'ensemble des noeuds v successeurs de u tels que v utilise X ($X \in ref(v)$) tels qu'il n'y ait pas de noeud u' qui redéfinisse X sur le chemin de u à v.

On modifie le programme pour mieux montrer ce critère :

```
1:  if (X <= 0) {
2:      X := -X;
    } else {
3:      X := 1 - X;
    }

4:  if (X == 1) {
5:      X := 1;
    } else {
6:      X := X + 1;
7:  print(X);
    }
```

Le dictionnaire *usages* s'écrit :

$$\begin{aligned} usages = \{X : \{ & \\ & 2 : 4, \\ & 3 : 4, \\ & 6 : 7, \}\} \end{aligned}$$

Un jeu de test vérifiant ce critère pour ce programme est alors :

$$\{\{X:0\},\{X:1\}\}$$

Notons que contrairement au critère *Toutes les définitions*, le critère *Toutes les utilisations* n'impose pas l'existence d'une utilisation !

2.7 Tous les DU-chemins

Tous les chemins simples entre une définition de X et une utilisation de X sans redéfinition doivent être exécutés au moins une fois.

Le critère *Toutes les utilisations* demandait simplement qu'au moins un chemin entre une paire (utilisation, définition) soit exécuté, ce critère-ci demande que tous les chemins simples le soient.

On récupère tous les chemins simples entre chaque noeud de définition ($def(u) \neq \emptyset$) et END grâce à la librairie *networkx*.

On regarde ensuite ces chemins à la recherche de paire (u, v) telles que u définit X, v utilise X, et il n'y a pas de redéfinition de X entre u et v.

Lorsqu'on trouve une telle paire, on stocke le chemin entre u et v dans un dictionnaire *du_paths*.

Il suffit ensuite de chercher ces morceaux de chemins dans les chemins d'exécutions correspondants aux tests.

2.8 Toutes les conditions

Nous n'avons pas implémenté ce critère faute de temps. Il est toutefois possible de tirer parti de notre structure d'AST : on peut envisager qu'au moment de la conversion entre l'AST et le CFG on transforme chaque décision en un ensemble de noeuds *If* correctement hiérarchisés. On pourrait ensuite vérifier le critère *Toutes les décisions* sur ce nouveau CFG plus détaillé.

2.9 Relations entre critères

Pour deux critères C_1 et C_2, on dit que C_1 est *plus fort* que C_2 si tout jeu de test satifaisant C_1 satisfait aussi C_2. On notera cette relation $C_1 \succ C_2$.

Ici, nous avons les relations suivantes :

$$\textit{Tous les DU-chemins} \succ \textit{Toutes les utilisations}$$
$$\textit{Toutes les conditions} \succ \textit{Toutes les décisions}$$

Chapitre 3

Génération des tests

3.1 Implémentation

On adopte la stratégie générale suivante pour générer les tests associés à un critère donné :

- générer un ensemble de chemins par lesquels on veut passer — en fonction du critère, on voudra passer par **tous** les chemins, ou par **au moins** un chemin
- pour chaque chemin, faire une exécution symbolique — on notera l'utilisation de symboles **uniques** associés à chaque variable explicite ou implicite (*ie.* les registres `_reg`, qui enregistrent les calculent intermédiaires)
- chaque exécution symbolique fournit des contraintes entre les symboles : donner ces contraintes au solveur Z3
- obtenir un test associé à un chemin si le problème est satisfiable

L'exécution symbolique à partir d'un chemin sur le CFG ainsi que la résolution du problème de contraintes sont effectuées dans le fichier `src/tests/solver.py`.

La génération des chemins est propre à chaque critère et est détaillée ci-dessous. Sauf explicitement dit, lorsque l'on dira que l'on génére tous les chemins entre deux nodes, on sous-entendra *tous les chemins simples*. Il s'agit d'un choix tout à fait discutable, mais étant donné l'indécidabilité du problème de la génération des tests, il faut de toute manière se fixer une limite raisonnable.

3.1.1 Toutes les affectations

Le critère *toutes les affectations* est un critère sur les nodes du CFG. Pour chaque node `SAssign`, on va générer tous les chemins jusqu'à ce node, en partant du node `START`.

On obtient un test pour un node lorsque l'on trouve au moins un chemin généré faisable.

3.1.2 Toutes les décisions

Le critère *toutes les décisions* est un critère sur les edges du CFG. On souhaite passer par chaque edge sortant d'un node de type `SIf` ou `SWhile`. On peut donc se ramener à un critère d'accessibilité au node de décision (`SIf` ou `SWhile`). On génère tous les chemins jusqu'au node, puis pour chaque chemin, on génère deux chemins : un chemin où l'on a rajouté l'edge associé à la décision `true`, un autre chemin où l'on a rajouté l'edge associé à la décision `false`.

On obtient un test pour un edge lorsque l'on trouve au moins un chemin généré faisable.

3.1.3 Tous les k-chemins

Le critère *tous les k-chemins* est un critère sur les chemins du CFG. On utilise donc directement le générateur de `k-path` du fichier `src/cfgraph/utils.py`. Il faut trouver un test pour chaque chemin. Si le solveur ne trouve pas de test admissible, le chemin n'est pas faisable.

3.1.4 Toutes les i-boucles

Le critère *toutes les i-boucles* est un critère sur les chemins du CFG. On utilise donc directement le générateur de `i-loop` du fichier `src/cfgraph/utils.py`. Il faut trouver un test pour chaque chemin. Si le solveur ne trouve pas de test admissible, le chemin n'est pas faisable.

3.1.5 Toutes les définitions

Le critère *toutes les définitions* est un critère sur les nodes de type *SAssign*. Pour chaque node `ndef`, on cherche les utilisations associées `nref` (*ie.* sans redéfinition intermédiaire.) On forme ainsi, pour chaque node `ndef`, un ensemble de paires de nodes `(ndef, nref)`. Pour chaque paire, on génère tous les chemins entre `START` et `ndef` et entre `ndef` et `nref`. On concatène deux sous-chemins pour obtenir un chemin pour la paire `(ndef, nref)`.

On obtient un test pour un node `ndef` lorsque l'on trouve au moins un node `nref` associé et un chemin, associé à la paire `(ndef, nref)`, faisable.

3.1.6 Toutes les utilisations

Le critère *toutes les utilisations* est un critère sur les nodes. Pour chaque variable et pour chaque node utilisant cette variable, noté `nref`, on cherche les définitions associées `ndef` (*ie.* dans redéfinition intermédiaire.) On forme ainsi des paires de nodes entre la définition d'une variable et son utilisation (sans redéfinition intermédiaire.) Pour chaque paire de nodes `(ndef, nref)`, on génère tous les chemins entre `START` et `ndef` et entre `ndef` et `nref`. On concatène deux sous-chemins pour obtenir un chemin pour la paire `(ndef, nref)`.

On obtient un test pour une paire `(ndef, nref)` lorsque l'on trouve au moins un chemin généré faisable.

3.1.7 Tous les DU-chemins

Le critère *tous les DU-chemins* est assez similaire au critère *toutes les utilisations*. De la même manière, on génère un ensemble de chemins pour une paire `(ndef, nref)` donnée.

La différence étant qu'il faut trouver un test pour chaque chemin. Et si le solveur ne trouve pas de test admissible, le chemin n'est pas faisable.

3.2 Résultats

Pour l'exemple `src/input/example3.imp` qui contient à la fois des branchement `SIf` et des boucles `SWhile`, nous obtenons pour $k = 10$ et $i = 1$:

```
$ python src/tests/generator.py src/input/example3.imp

Running gen_ta...
Feasibility of 100.00%
Generated test : [{'x': 0, 'z': 0}, {'x': -1, 'z': None},
                  {'x': 0, 'z': None}]
gen_ta took 47.37ms

Running gen_td...
Feasibility of 100.00%
Generated test : [{'x': -1, 'z': None}, {'x': 0, 'z': None}]
gen_td took 21.23ms

Running gen_ktc...
Feasibility of 100.00%
```

```
Generated test : [{'x': 0, 'z': 0}, {'x': -1, 'z': None},
                  {'x': 0, 'z': 1}]
gen_ktc took 25.48ms

Running gen_itb...
Path ['START', 0, 1, 2, 4, 5, 6, 7, 8, 7, 9, 'END'] is unfeasible
Feasibility of 75.00%
Generated test : [{'x': 0, 'z': 0}, {'x': -1, 'z': None},
                  {'x': 0, 'z': 1}]
gen_itb took 34.75ms

Running gen_tdef...
Feasibility of 100.00%
Generated test : [{'x': 0, 'z': 0}, {'x': 0, 'z': 1},
                  {'x': -1, 'z': None}, {'x': 0, 'z': None},
                  {'x': None, 'z': None}]
gen_tdef took 58.07ms

Running gen_tu...
Feasibility of 100.00%
Generated test : [{'x': 0, 'z': 1}, {'x': 0, 'z': 0},
                  {'x': -1, 'z': None}, {'x': 0, 'z': None},
                  {'x': None, 'z': None}]
gen_tu took 52.44ms

Running gen_tdu...
Simple path [4, 5, 6, 7, 8] is unfeasible
Simple path [4, 5, 6, 7, 8, 7, 9] is unfeasible
Feasibility of 84.62%
Generated test : [{'x': 0, 'z': 1}, {'x': 0, 'z': 0},
                  {'x': -1, 'z': None}, {'x': 0, 'z': None},
                  {'x': None, 'z': None}]
gen_tdu took 103.47ms
```

On notera deux choses. D'une part, certains critères ne peuvent pas être validés à 100%. Il se s'agit pas d'une erreur du solveur, mais bien d'une détection de chemins non traversables quelle que soit l'entrée utilisateur. D'autre part, le solveur peut attribuer une valeur `None`, ce qui signifie que la valeur de la variable peut-être quelconque pour ce test.