Planchais Christophe, Benkebir Chakib et Youssouf Roudani

RAPPORT

PROJET VV HCC/NPE

CONSTRUCTION D'UN GRAPHE DE FLOT DE CONTRÔLE POUR CALCUL HCC / DÉTECTION NPE : DÉVELOPPEMENT AGILE

INTRODUCTION :

Nous avons considéré la recherche HCC comme un test pour notre programme. La recherche réussi en calculant le nombre de parcours différents parmi les méta nœuds. L'expansion des méta-nœuds « MetaNode.java » représente un changement de scope de variable (une branche de IF, un corps de While ou un appel de fonction). Un nouveau méta-nœud n'implique pas forcément une augmentation de la complexité cyclomatique (ex. un appel de fonction). On prendra pour la HCC, le maximum des CC suivant chaque entrée *main* du package.

UTILISATION DE SPOON :

class_X_Hydrator.java :

L'utilisation de *Spoon* n'est pas simple, ainsi nous avons dédié tout notre « front-end » à Spoon, tandis que notre « back-end » s'occupe du flot de contrôle, le « front-end » étant logiquement le client du « back-end ». Lors de notre développement agile, nous avons suivi l'API Spoon en constatant qu'un parsing à la main était parfois nécessaire pour profiter d'informations telles que : variable appelant une fonction ou classe élémentaire (le résultat de getshortRepresentation).

La classe NODE.java encapsule les CtElements Spoon et propose des champs destinés à remonter dans l'AST.

```
public class NODE {
    X_CtMethodImpl node_method;
    X_CtConstructorImpl node_constructor;
    X_CtClassImpl node_class;
    X_CtInterfaceImpl node_interface;
    X_CtPackageImpl node_package;
    X_CtPackageImpl node_root;
```

La difficulté nous a paru grande, ainsi Class_X_Hydrator.java concentre tout le code spoon et nous a permis d'avancer sans savoir ce dont spoon serait vraiment capable.

```
static void hydrate_x (NODE entry, NODE root) {
    entry.node_string=entry.i_element.toString();
```

entry.node_root=(X_CtPackageImpl) root;

hydrate_switch(entry,root); **-->** appel au filtre statique de construction des enfants du NODE x

for (NODE x : entry.children)

Class_X_Hydrator.hydrate_x(x, root); **-->** appel (récursif) au filtre "hydrate_x" pour chacuns des nouveaux enfants de x

}

A l'heure de rendre ce rapport, nous sommes d'ailleurs encore attachés à combler les manques de notre « front-end ».

```
static void hydrate_switch(NODE nX, NODE root) {
    if(nX instanceof X_CtPackageImpl) {
        X_CtPackageImpl n=(X_CtPackageImpl) nX;
        Set<CtPackage> pcks= n.i_element.getPackages();
        for (CtPackage pck : pcks) {
            NODE x = X_.create(pck, null, null,null, null, n, n); --> appel au filtre statique de
            construction X_.java (wrapping)
            n.children.add(x);
        }
```

[...]

SOLUTION DE DESIGN : CREATION DES NOEUDS DE WRAPPING

Toutes les classes de wrapping X_Ct-Impl héritent de NODE.java; la méthode « expand » de ces classes est destinée au calcul du flot de contrôle du programme.

L'entrée static main de PackageAnalysis.java : entrée normale du programme. Elle utilise le processeur :

"class _PackageProcessor extends AbstractProcessor<CtPackage> " qui est uniquement utilisé pour détecter le package racine :

CtPackage rootpck= packages.get(packages.size()-1);

xpackage = (X_CtPackageImpl) X_.create(rootpck, null,null,null,null,null,null); **-->** et on wrapp le rootpck.

Ensuite, on appelle le filtre de construction pour le package racine.

Class_X_Hydrator.hydrate_x(xpackage, xpackage);

Enfin, on cherche les méthodes main du package avec une méthode statique,

M_Linearizer.scan_for_main (xpackage, M_Linearizer.meth_main_list);

for (X_CtMethodImpl m : M_Linearizer.meth_main_list) {

new M_Linearizer(m); **-->** et on lance M_Linearizer pour chaque entrée main;

}

M_Linearizer lance build_control_flow() qui lance la methode "expand_x" récursive (le "_x" dans le nom). Comme paramètre on trouve le meta noeud de base "this.main_Node".

ENTRÉE STATIC MAIN DE ClassBuilder.java :

Nous pensons que la construction automatique par ClassBuilder.java des classes de wrapping X_Ct-Impl, du filtre de construction X_ et du filtre d'hydratation class_X_Hydrator (le parcours récursif de l'AST) reste valable.

SOLUTION : DESIGN DES META NOEUDS DU FLOT DE CONTRÔLE

la classe MetaNode.java possède trois listes :

ArrayList<NODE> LQNodes ; **-->** les NODE considérés linéaires (pas de branchement ou de changement de scope des variables)

ArrayList<NODE> WQNodes ; **-->** les NODE en attente de traitement WaitingQueue

ArrayList<MetaNode> NQMetaNodes ; **-->** les MetaNodes qui suivent dans le graphe du flot de contrôle du programme

EVALUATION : DÉBOGAGE PAR MESSAGES

Le package analysé est imprimé de la manière suivante :

NPE->2 classes and 1 interfaces 1 sub packages

NPE.NPEPACK->1 classes and 0 interfaces 1 sub packages

NPE.NPEPACK.NPEPACK2->1 classes and 0 interfaces 0 sub packages

Toutes les classes analysées le sont de la manière suivante (on a les constructeurs, les paramètres et les méthodes) après le sign @@:

public static void main(String[] args){@@ CtMethodImpl

Tri2 tri=new Tri2("");@@ CtLocalVariableImpl CtConstructorCallImpl

tri.name="kristof";@@ CtAssignmentImpl CtFieldWriteImpl-tri.name CtLiteralImpl-"kristof"

int j=2+1;@@ CtLocalVariableImpl CtBinaryOperatorImpl CtLiteralImpl-2 CtLiteralImpl-1

int i = tri.calltri(j);@@ CtLocalVariableImpl CtInvocationImpl CtVariableReadImpl-j

if (j==2)@@ CtIfImpl CtBinaryOperatorImpl CtVariableReadImpl-j CtLiteralImpl-2

On lance une construction de type "itérative", certes inefficace en temps, à des fins de débogage : à chaque changement on recommence une nouvelle "passe" à partir de la racine tant que des changements existent.

M_Linearizer.java:

```
public void build_control_flow(MetaNode root){
    while (MetaNode.compute_changed( root) >0) {
    counter ++;
    System.out.println(counter +"eme pass ----------\n");
    MetaNode.reset_changed_x(root);
    expand_x (root,root);
```

Il nous est apparu important de visualiser la phase de construction des meta nœuds. Des erreurs concurrentielles d'accès aux listes LQ/WQNodes nous ont posés beaucoup de soucis. C'est à cause de cela, que nous avons opté pour une construction en profondeur « iterative ».

public void expand_x (MetaNode root, MetaNode entry) {

expand_WQ(entry); **-->** on commence par vider les noeuds, en attente, de la liste WQNodes

for (MetaNode n : entry.NQMetaNodes) { **-->** puis on lance la récursion

expand_x (root,n);

EXEMPLE : X_CtlfImpl.java:

public void expand(MetaNode parent) {

MetaNode mn_else = new MetaNode(parent);

MetaNode mn_then = new MetaNode(parent);

mn_then.WQNodes.add(this.then_block);

mn_else.WQNodes.add(this.else_block);

parent.WQNodes.remove(0);

parent.changed=true;

parent.NQMetaNodes.add(mn_then);

parent.NQMetaNodes.add(mn_else);

DISCUSSION

Le problème du NPE qui apparait lorsque l'on essaie de déréférencer une variable pointant sur le zéro-mémoire ne peut être résolu entièrement car toute propriété (telle que le programme plante-il?) "sémantique et non-triviale" est indécidable (Rice). Cependant, le challenge était intéressant, car nous avons découverts ce qu'est l'introspection de code java.

Notre programme, de par sa construction (génération automatique, etc..) affiche presque les caractéristiques d'un interpréteur. Nous pensons qu'avec peu de temps supplémentaires, nous aurions pu rendre ce programme opérationnel. La discussion sera donc orientée « software engineering » : Devons-nous construire des programmes nécessaires ou bien suffisants ?

Peut-être avons-nous voulu voir des problèmes là où c'était trivial. Pourtant nous nous sommes attachés à développer de façon agile (les 2 deadlines du 22 et 19 septembre). La question reste ouverte.