Projet Adabu

8INF958 - Spécification, test et vérification

Imen Doudech
David Levayer
Corentin Ricou

Introduction

- Complexité croissante des systèmes
- Importance du monitoring
- Création d'un modèle basé sur l'observation
- Connaître le comportement normal permet de détecter les écarts
- Importance des contraintes temporelles

Sommaire

Contexte d'utilisation

Comportement attendu vs. Réalité

- Surveiller l'appel de méthodes n'est pas suffisant
- Permet de trouver des *bugs…* mais pas assez !
- La génération de tests à partir des contraintes possèdent une faiblesse : les contraintes elle-mêmes
- Parfois, certaines contraintes sont mal exprimées voire implicites
- Contraintes temporelles : Appel(B) seulement après Appel(A)
- Beaucoup d'exemples : liste, pile, map, etc.

Présentation d'Adabu

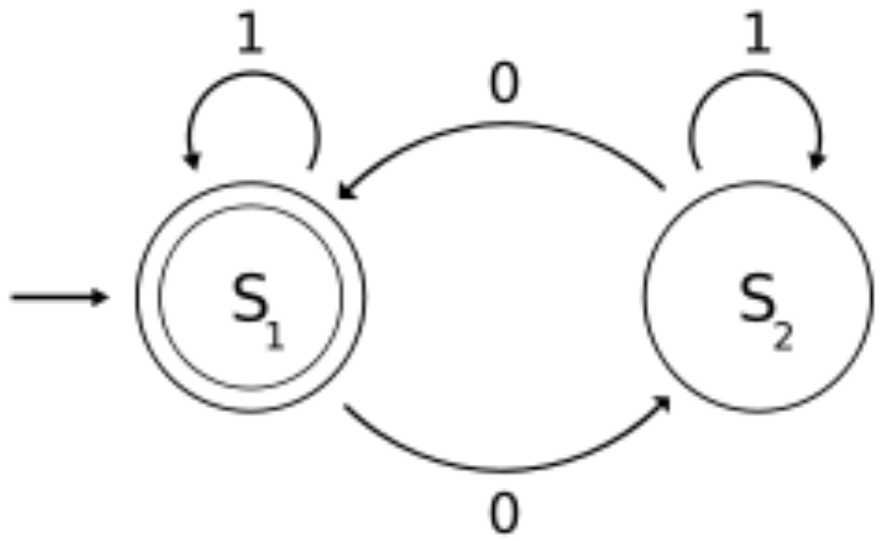
Généralités

- Outil open-source programmé en Java
- Forage de données : construire un modèle à partir de données observées
- Utilisation de machines à états (automate)
- Dernière mise à jour : 2011
- Créé par un groupe d'ingénieur de l'université de Saarland (Allemagne)

Présentation d'Adabu

Rappel sur les automates

- Un automate est une machine abstraite permettant de reconnaitre l'appartenance d'un mot à un langage
- Dans notre cas :
  - un mot correspond à une exécution possible d'un programme
  - le langage correspond à l'ensemble des exécutions considérées comme « normales »
- On modélise généralement les automates de manière graphique



Présentation d'Adabu

Principes clés de l'outil

- Deux types de méthodes
- les inspecteurs : fonctions utilisées pour récupérer une partie de l'état de l'objet (doit renvoyer un type différent de void, ne prendre aucun paramètre et ne pas modifier l'objet)
- les manipulateurs : toutes les autres méthodes (notamment celles qui modifient les attributs de l'objet)

Présentation d'Adabu

Principes clés de l'outil

- Adabu réutilise le projet de A. Salcianu et M. Rinard
- Une méthode est pure si elle ne modifie aucun attribut/valeur d'aucun objet (pas de différence entre avant l'appel et après)
- Ce projet permet également d'obtenir des précisions sur les méthodes appelées (type de retour, paramètres read-only, etc.)
- L'outil analyse les modifications de variables et d'attributs ; les structures créées à l'intérieur de la fonction ne sont pas concernées

Présentation d'Adabu

Instrumentation

- Pour l'instrumentation, Adabu utilise Javassist
- Ce framework permet de manipuler facilement du bytecode
- De cette façon, on peut ajouter des appels à des méthodes juste avant l'exécution
- Similarité avec AspectJ quant au résultat obtenu
- Les méthodes servent à construire l'automate notamment en récupérant l'état de l'objet
- Sert à construire un fichier de traces

Présentation d'Adabu

Construction du modèle

- Une fois la trace obtenue, on peut générer le modèle
- Approche naïve : états concrets
  - variable *int* : on génère un grand nombre d'états !
- Solution : états abstrait
  - variable *int* : a < 0, a = 0 et a >0
- Traces d'un objet : correspond à la liste des transitions valides de l'automate : sous la forme t = { (s1, c1, s1'), (s2, c2, s2') }
- Les états abstraits forment les modèles de chaque objet analysé
- Le modèle final n'est autre que la fusion des modèles des objets

Présentation d'Adabu

Utilisation de l'outil

- Installation facile : il suffit de décompresser le projet
- Utilisation plus délicate
  - Utilise Java 1.6 : erreur avec les JDK supérieurs
  - Problème avec certaines variables des scripts
- Plusieurs étapes clés
  - compilation des classes à analyser (commande javac)
  - Création d'un fichier de traces (script traceur.sh)
  - Création de l'automate (script adaburun.sh)

Exemples d'utilisation

Cas simple : programme *dummy*

- Exemple fourni par le site d'Adabu
- Pas vraiment pertinent : cas trop simple et sans contrainte temporelle
- Permet de tester les paramètres de l'outil

```
package test;

public class Foo {
  int state = 0;
  public Foo() {
    state = 1;
    System.out.println("Foo constructor.");
  }

  public void bar() {
    state = 2;
    System.out.println("Foo.bar");
  }
}
```

Exemples d'utilisation

Cas simple : pile

```java
public class Pile {

    private Stack<String> st;

    public Pile(){
        st = new Stack<String>();
    }

    public boolean add( String elem ){
        st.push(elem);
        return true;
    }

    public String remove(){
        return st.pop();
    }

    public boolean clear(){
        st.clear();
        return true;
    }

    public boolean isEmpty(){
        return st.empty();
    }
}
```

- Notre exemple
- Présence d'une contrainte temporelle (appel à remove)

```java
public class TestPile {
      public static void main(String[] args) {

          Pile p = new Pile();
          p.clear();
          p.add("Bonjour");
          p.remove();
          p.add("Je");
          p.add("m'appppel");
          p.remove();
          p.add("m'appele");
          p.add("David");
          p.remove();
          p.clear();

      }
}
```

Résultats

Adabu

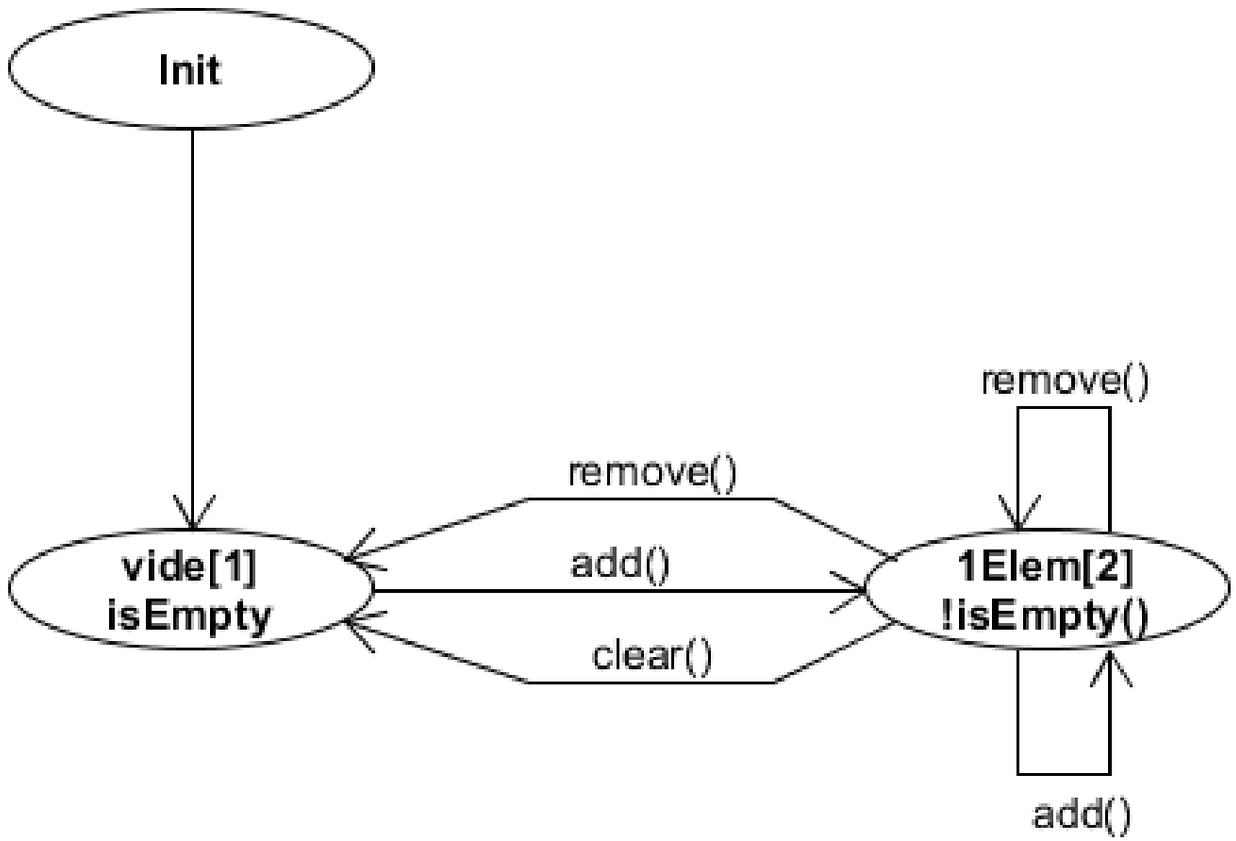
- Automate généré

```
digraph automaton {
	page="9,11";
	size="9,11";
	0 -> 1 [label="<init>()V{0=6694}"];
	1 -> 1 [label="clear()Z{0=6694}"];
	1 -> 2 [label="add(Ljava/lang/String;)Z{0=6694, 1=6704},add(Ljava/lang/String;)Z{0=6694, 1=6711}"];
	2 -> 1 [label="remove()Ljava/lang/String;{0=6694},clear()Z{0=6694}"];
	2 -> 2 [label="add(Ljava/lang/String;)Z{0=6694, 1=6726},remove()Ljava/lang/String;{0=6694},add(Ljava/lang/
String;)Z{0=6694, 1=6722},add(Ljava/lang/String;)Z{0=6694, 1=6715},remove()Ljava/lang/String;{0=6694}"];
	0 [label=""];
	1 [label="st: 1,st.capacityIncrement: 0,st.elementCount: 0,st.elementData: 1,st.modCount: 0"];
	2 [label="st: 1,st.capacityIncrement: 0,st.elementCount: 1,st.elementData: 1,st.modCount: 0"];
}
```

Résultats

Adabu

- Automate généré (version compréhensible…)



Utilisation des résultats d'Adabu

Runtime monitoring

- Utilisation de l'automate dans un aspect
- Permet de faire de la surveillance lors de l'exécution

```
  // Coupe toutes les méthodes des classes de myClass
pointcut myMethod(): myClass() && execution(* *(..));

before() : myMethod(){

    if(!activated)
        return;

    // Action réalisée à l'entrée des fonctions coupées par myClass
    Signature s  = thisJoinPointStaticPart.getSignature();
    String name = s.getName();
    if(name.equals("main")){
        initAutomata();
        return;
    }

    boolean transitionAccepted = automate.switchState(name);
    if(transitionAccepted)
        System.out.println("Transition: "+automate.getPreviousState()+" ("+name+") --> "+
                automate.getCurrentState());
    else{
        System.out.println("Echec: "+automate.getPreviousState()+" ("+name+") --> "+
                automate.getCurrentState());
        activated = false;
    }
```

Démonstration

- Time to work…

Conclusion

Toutes les bonnes choses ont une fin

- Connaître le fonctionnement « normal » est un premier pas pour détecter les erreurs
- Adabu permet d'analyser un programme dans l'optique d'en construire un modèle
- Ce modèle prend la forme d'un automate : il est ensuite possible de réaliser du *runtime monitoring*
- Adabu possède aussi plusieurs défauts : très peu de documentation, version ancienne de Java, communauté inexistante, automate non-déterministe
- Ouverture : concept à fort potentiel, mais avec un outil différent…

Questions

?

Imen Doudech
David Levayer
Corentin Ricou