WShabtiPict - Guida Sviluppatore

Introduzione

WShabtiPict è un semplice tool per:

1. Generare combinazioni di input per i test case dati un insieme di parametri e vincoli
2. Esplicitare gli oracoli per l'insieme delle combinazioni generate
3. Generare classi di test compatibili con JUnit 5

WShabtiPict offre due diverse modalità di utilizzo: interattiva e automatica.

In modalità **interattiva**, il tool guida l'utente nelle fasi di:

- Creazione del modello da testare (parametri, vincoli e grado delle combinazioni)
- Definizione dell'oracolo (umano)
- Creazione della classe di test compatibile con JUnit 5

In modalità **automatica**, il tool offre la possibilità di fornire all'eseguibile dei parametri di ingresso e di accedere singolarmente alle fasi di:

- Creazione di casi di test a partire da un modello pre-esistente fornito tramite file
- Creazione guidata dell'oracolo a partire da casi di test pre-generati forniti tramite file
- Creazione della classe di test a partire dai casi di test completi di oracolo forniti tramite file

Building

Building su Linux, OS/X, *BSD, etc.

In Linux e sistemi Unix-like (come OS/X) la compilazione va eseguita tramite clang++ o g++.

Su Linux è necessario l'utilizzo della libreria libstdc++ offerta da gcc 5 e l'utilizzo di c++2a offerto da g++-9.

Tramite il comando *make* si può effettuare la compilazione automatica.

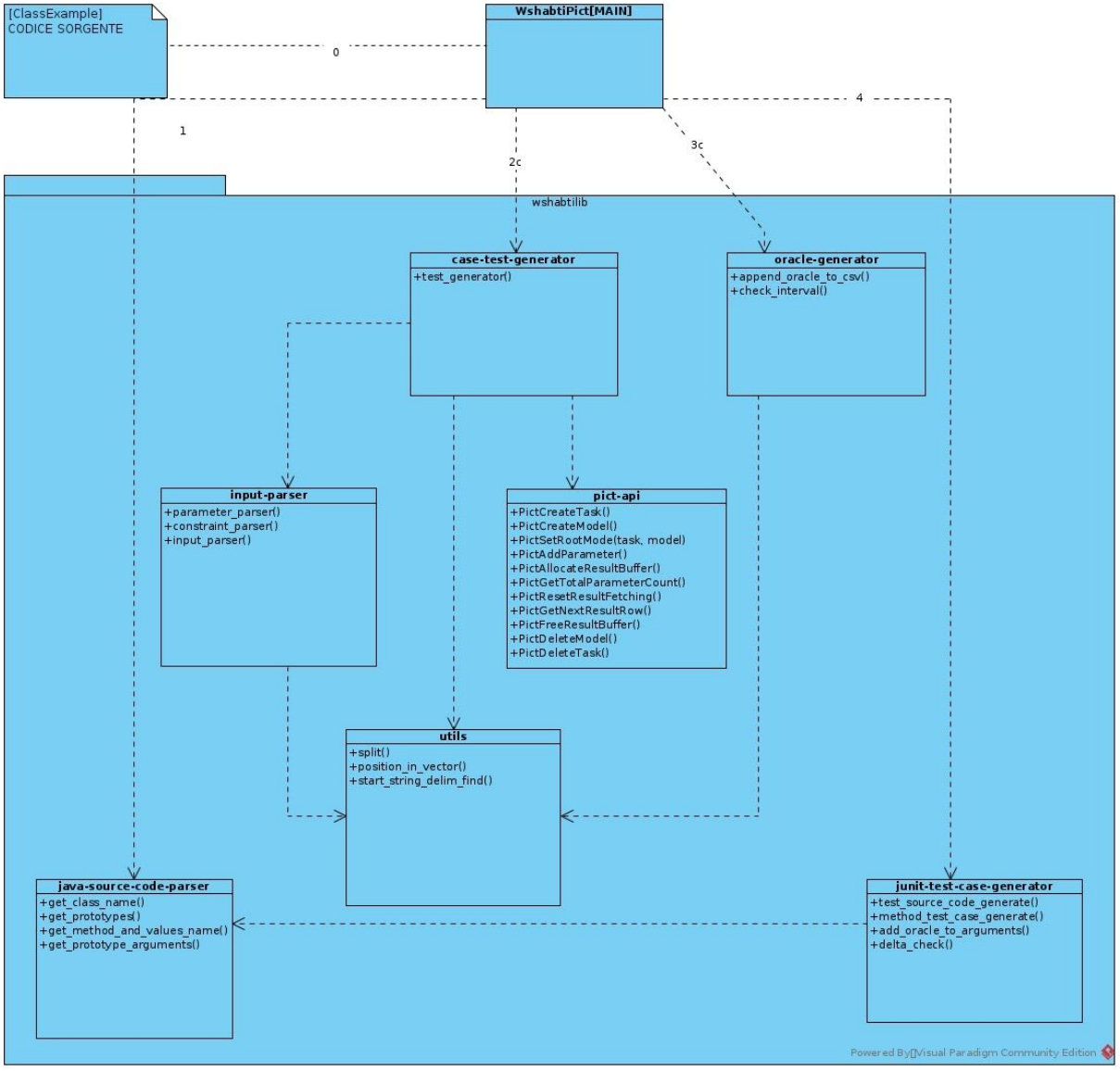
Building su Windows

Ci sono due possibilità su Windows per compilare ed eseguire il progetto:

- Installare *cygwin* con gli add-ons *make* e *build-essential*, ciò permette di ottenere su Windows un sottosistema *Linux*. A questo punto si potrà procedere come su *Linux* avviando la compilazione tramite il comando make
- Utilizzare *WSL* (*Windows Subsystem for Linux*) per eseguire in ambiente *GNU/Linux* senza installare *cygwin*. Per compilare il progetto è necessario soddisfare le seguenti dipendenze che vanno installate manualmente, ad esempio su derivate di *Debian* (come *Ubuntu*) i comandi sono:

```
sudo apt update
sudo apt install build-essential
sudo apt install make
sudo apt upgrade
```

Struttura



Nel diagramma (dove i collegamenti rappresentano relazioni di utilizzo) il *main* sfrutta una serie di funzioni messe a disposizione dai moduli della *wshabtilib*. Esso è quindi tranquillamente sostituibile con un altro *main* coordinatore che può richiamare i moduli a piacimento.

L'implementazione fornita rispetta il seguente flusso di esecuzione:

1. Parsing della classe .java attraverso il modulo ***java-source-code-parser***.
   In questa fase, dal codice sorgente, vengono estratti i prototipi ed il nome della classe.

2. A partire da un metodo si generano i casi di test attraverso il modulo ***case-test-generator***.
   Esso, dato l'insieme dei parametri, eventuali vincoli ed il grado delle combinazioni, fornisce un file .csv contenente le combinazioni per i casi di test (senza oracolo).
   Attraverso il modulo ***input-parser*** costruisce:

   - *Matrice dei parametri*
   - *Matrice dei vincoli*

   Tali matrici vengono utilizzate per generare le combinazioni per i casi di test tramite ***pict-api***.

3. Una volta ottenuto il file .csv contenente le combinazioni per i casi di test, si aggiungono gli oracoli attraverso il modulo ***oracle-generator*** che li aggiunge al file.
   L'operazione di definizione dell'oracolo consiste in una semplice aggiunta del valore atteso che, eventualmente, potrà esser seguito da un valore *delta* tramite cui definire un intervallo entro cui il valore atteso dovrà trovarsi.

4. I passi 2,3 si ripetono ciclicamente per tutti i metodi desiderati

5. Si genera la classe di test contenente l'insieme dei casi di test per i metodi scelti attraverso il modulo ***junit-test-case-generator***.
   Questo modulo sfrutta il modulo ***java-source-code-parser*** per fare un parsing più accurato e ottenere, oltre al nome della classe e i prototipi, anche gli argomenti dei metodi

case-test-generator

Responsabilità: produce le combinazioni per i casi di test

Prototipo: *void test_generator(int k_degree, string input_file, string output_file)*
Parametri di ingresso: • **k_degree: ordine del test combinatoriale** • **input_file: path del file INPUT_VALUES_FILE** • **output_file: path di uscita per il file TEST_INPUT_FILE**
Parametri di uscita: test_case.csv
Commento: la funzione genera in uscita un file .csv contenente le combinazioni generate. Il valore di *k_degree* deve essere minore o uguale del numero dei parametri.

input-parser

Responsabilità: analisi del documento testuale per costruire le matrici di parametri e vincoli

Prototipo: *void input_parser(string input_file_path, vector<vector<string>> &parameter_matrix, vector<vector<string>> &constraint_matrix)*
Parametri di ingresso: • **input_file_path: path del file INPUT_VALUE_FILE** • **&parameter_matrix: matrice che conterrà i parametri** • **&constraint_matrix: matrice che conterrà i vincoli**
Parametri di uscita: parameter_matrix, constraint_matrix
Commento: la funzione sfrutta le due sottofunzioni parameter_parser e constraint_parser per costruire le matrici. In particolare, input_parser, crea un file stream che passa sequenzialmente alle 2 funzioni (prima al parameter_parser e poi al constraint_parser così da "spezzettare" la navigazione delle righe del file in 2 step).

Prototipo: *void constraint_parser(ifstream &input_file_stream, vector<vector<string>> &constraint_matrix)*
Parametri di ingresso: • **&input_file_stream: flusso del file in input** • **&constraint_matrix: matrice che conterrà i vincoli**
Parametri di uscita: constraint_matrix
Commento: la funzione viene richiamata dall'input_parser dopo il parameter_parser. Infatti il parameter_parser si interromperà appena trova un # (constraints presenti) o la fine del file (constraints non presenti).

Prototipo: *void parameter_parser(ifstream &input_file_stream, vector<vector<string>> &parameter_matrix)*
Parametri di ingresso: • **input_file_stream: flusso del file in input** • **&parameter_matrix: matrice che conterrà i parametri**
Parametri di uscita: parameter_matrix
Commento: la funzione viene richiamata dall'input_parser. Il parameter_parser si interromperà appena trova un # o la fine del file.

java-source-code-parser

Responsabilità: analizza il codice sorgente della classe Java e permette l'estrazione dei metodi e il nome della classe

Prototipo: *string get_class_name(string source_code_path)*
Parametri di ingresso: • **source_code_path: path del codice sorgente della classe Java**
Parametri di uscita: class_name
Commento: la funzione estrapola dal codice sorgente fornito il nome della classe.

Prototipo: *vector<string> get_prototypes(string source_code_path)*
Parametri di ingresso: • **source_code_path: path del codice sorgente della classe Java**
Parametri di uscita: prototypes
Commento: la funzione estrapola dal codice sorgente fornito un vettore contenente le firme di tutti i prototipi delle funzioni membro in esso contenute.

Prototipo: *vector<string> get_method_and_values_name(string method_prototype)*
Parametri di ingresso: • **method_prototype: firma del metodo in esame**
Parametri di uscita: method_and_values_name
Commento: la funzione genera in uscita un vettore contenente sotto forma di stringhe i valori di un metodo e il nome del metodo stesso nella prima posizione.

Prototipo: *string get_prototype_arguments(string method_prototype)*
Parametri di ingresso: • **method_prototype: firma del metodo in esame**
Parametri di uscita: prototype_arguments
Commento: la funzione genera in uscita una stringa contenente gli argomenti del metodo, inclusi i loro tipi.

junit-test-case-generator

Responsabilità: genera una classe di Test (.java) utilizzabile con JUnit 5

Prototipo: *`void test_source_code_generate(string input_file_path, string output_file_path, string test_class_name, vector<string> prototypes)`*
Parametri di ingresso: • **`input_file_path: path del file TEST_WITH_ORACLES_INPUT_FILE`** • **`output_file_path: path di uscita per il file .java`** • **`test_class_name: nome della classe in test`** • **`prototypes: vettore dei prototipi testati`**
Parametri di uscita: `<test_class_name>Test.java`
Commento: la funzione genera nell'output_file_path un file .java utilizzabile con la suite di test JUnit5.

oracle-generator

Responsabilità: aggiunge l'oracolo alle combinazioni dei casi di test

Prototipo: *void append_oracle_to_csv(vector<string> oracle, string input_file_path, string output_file_path)*
Parametri di ingresso: • **oracle: vettore contenente l'oracolo per ogni caso di test** • **input_file_path: path del file TEST_INPUT_FILE** • **output_file_path: path di uscita per il file TEST_WITH_ORACLES_INPUT_FILE**
Parametri di uscita: TEST_WITH_ORACLES_INPUT_FILE
Commento: la funzione aggiunge ad ogni riga di file il rispettivo oracolo (e delta se usato).

junit-test-case-generator

Responsabilità: genera il file .java formattato per l'utilizzo con JUnit 5

Prototipo: *void test_source_code_generate(string input_file_path, string output_file_path, string test_class_name, vector<string> prototypes)*
Parametri di ingresso: • **input_file_path** • **output_file_path** • **test_class_name** • **prototypes**
Parametri di uscita: <test_class_name>Test.java
Commento: la funzione genera nell'output_file_path un file .java compatibile con la suite di test JUnit5.

pict-api

È possibile trovare la documentazione qui.

Sintassi dei file

INPUT_VALUES_FILE

Il file che specifica i possibili valori di un parametro, rispetta la seguente sintassi:

```
<nome_parametro1>:<valore_1>,<valore_2>,<valore_3>,...
<nome_parametro2>:<valore_1>,<valore_2>,<valore_3>,...
...
#CONSTRAINTS
<valore_p1_x>,<valore_p2_x>,<valore_p3_x>,...
...
```

Per specificare dei vincoli esclusivi, è sufficiente aggiungere in seguito alla specifica di tutti i parametri, una riga separatrice che inizia con # *(è possibile aggiungere qualunque commento a seguito del cancelletto).*
Dalla riga successiva a quella che inizia con il cancelletto, ciascuna indicherà una combinazione da escludere.
Tramite l'utilizzo di un ***** si può indicare che il parametro a quella posizione può assumere qualunque valore (è quindi possibile specificare più esclusioni in una sola riga)

Ad esempio:

```
filesystem:ext4,ntfs,fat32
os:windows,linux
dimensione_cluster:512,1024,2048,4096
#CONSTRAINTS
ext4,windows,*
```

permette di escludere la combinazione ext4,windows ovvero qualunque test che contiene ext4 e windows, a prescindere dal valore del parametro dimensione_cluster.

Nel caso si volesse utilizzare un intervallo di valori continuo, è possibile utilizzare la sintassi:

```
<nome_parametro>:<start>:<stop>:<step>,...
```

Dove start indica il primo valore assunto, stop è l'ultimo valore - incluso - e step specifica l'avanzamento. *E' possibile combinare entrambe le notazioni*
Ad esempio:
`1:6:5` viene espanso in *{1,2,3,4,5,6,7,8,9,10}*
`1,2,5:14:2` viene espanso in *{1,2,5,7,9,11,13}*

TEST_INPUT_FILE

Il file contenente i casi di test generati rispetta la seguente sintassi:

```
<valore_p1>,<valore_p2>,...
```

Ogni test generato occupa una riga del file, ciascuna riga è composta dai valori di ogni parametro.

TEST_WITH_ORACLES_INPUT_FILE

Il file contenente i casi di test con oracolo presenta la seguente sintassi

```
<valore_p1>,<valore_p2>,...,valore_atteso,delta
```

Il valore delta risulta necessario solo quando almeno uno dei valori attesi deve essere contenuto all'interno di un intervallo (estremi inclusi). In questo caso sarà necessario aggiungere un delta nullo ",0" accanto i rimanenti valori attesi esatti.

Ad esempio:
`10,3,3.3,0.1`
`5,2,2.5,0` {il valore atteso esatto 2.5 viene rinchiuso in un delta ,0}
`5,3,1.6,0.1`

Classe di test generata

Per l'insieme dei metodi di test viene generato un file .java per il testing in JUnit 5.
A questo punto, per utilizzare la classe di test per il testing di unità, non resta che importare nel progetto:

- La classe di test situata in ***"./JUnitTests/<nome della classe>/<nome della classe>Test.java"***
- I file .csv contenenti gli oracoli e le combinazioni da testare **per ogni metodo** situati in ***"./JUnitTests/<nome della classe>/<nome del metodo>_test.csv"***

Oracolo esatto

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvFileSource;

class CalcolatriceTest {

    @ParameterizedTest
    @CsvFileSource(resources = "./somma_test.csv")
    public void test_somma(int a, int b, Object oracle){
         Calcolatrice CalcolatriceTest = new Calcolatrice(/*constructor arguments*/);
         assertEquals(String.valueOf(oracle), String.valueOf(CalcolatriceTest.somma(a, b)),"somma FAILED");
    }
}
```

Il template per l'oracolo esatto prevede:

- import, @ParameterizedTest, @CsvFileSource
- nome della classe di test (viene aggiunto al nome della classe il suffisso Test)
- Riferimento al file .csv come "./<nome del metodo>_test.csv"
- Creazione del metodo "test_<nome del metodo>" con parametri:
    - Parametri del metodo da testare
    - Object oracle
- Istanziazione della classe sotto test con costruttore di default.
    - Per utilizzare un costruttore specifico occorre aggiungere manualmente i parametri a seguito della creazione del file di output "<nome della classe>Test.java"

- Utilizzo di assertEquals che effettua un confronto esatto tra valore di ritorno e valore atteso (entrambi convertiti in stringhe). Tramite il metodo .valueOf() è possibile ottenere anche il valore delle classi wrapper di Java (Integer, Float,...).
  Inoltre .valueOf() permette di confrontare anche classi con metodi che hanno un valore di ritorno non standard di Java, a patto che effettuino un override di tale metodo.

Oracolo non esatto

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvFileSource;

class CalcolatriceTest {

    @ParameterizedTest
    @CsvFileSource(resources = "./divisione_test.csv")
    public void test_divisione(int a, int b, double oracle, double delta){
        Calcolatrice CalcolatriceTest = new Calcolatrice(/*constructor arguments*/);
        assertEquals(oracle,CalcolatriceTest.divisione(a, b),delta, "divisione FAILED");
    }
}
```

Il template per l'oracolo **non esatto** prevede:

- *import, @ParameterizedTest, @CsvFileSource*
- nome della classe di test (viene aggiunto al nome della classe il suffisso Test)
- Riferimento al file .csv come **"*./<nome del metodo>_test.csv*"**
- Creazione del metodo **"test_*<nome del metodo>*"** con parametri:
    - Parametri del metodo da testare
    - double oracle
    - double delta
- Istanziazione della classe sotto test con costruttore di default
    - Per utilizzare un costruttore specifico occorre aggiungere manualmente i parametri a seguito della creazione del file di output **"<nome della classe>Test.java"**
- Utilizzo di assertEquals che verifica che il modulo della differenza tra il valore di ritorno e il valore atteso (oracolo) sia contenuto nell'intervallo [-delta,+delta].

Limitazioni

WShabtiPict, al momento, supporta solo i tipi standard di Java (incluse le classi Wrapper) e solo metodi che ritornano uno scalare.

Licenza