JUnitMe2.0 : JUnit tests generation with Alloy and CodeModel

Auteurs :
Salla DIAGNE
Anis TELLO

16th December 2015

Table of contents

Introduction

"Every program is guilty until proven innocent".

During the development phase, developers spend lots of time writing tests for code they have written. But these tests can be very so tedious and repetitive that developers sometimes botch or simply skip this very important part of software development.

But what if developers didn't have to spend hours writing tests to find bugs and have a good code coverage of their program? What if there was a magic stick that can generate a bunch of unit tests?

Since Java is one of the most used programming languages nowadays, we implemented a solution that would generate automatically Java unit tests. This solution would save time for developers, and give the time and the energy to focus on working on the business layer.

This project is an extended version of an application developed by Valentin Lefils and Quentin Marrecau [1] [2] . The first version of the application treated the same problems we are facing, but only on small examples of Java programs.

In this project, we present our tool: JUnitMe2.0. Our tool can generate automatically Java unit tests for any Java open source application. From a description of specification, our tool generate all instances that covere the data specifications, then generate the unit tests corresponding to these instances.

Chapter 1

Technical work

1.1 Goal

The goal of this tool is to generate a large amount of Java unit tests automatically so that we can achieve a good coverage of code with no time or effort spent. Therefore developers can focus on developing the business layer without worrying about testing the code they have written. Our tool can be used to verify that no bugs will occur when objects of the program interact with each other. In other words, our tool can verify that there no actual error exists for an application.

1.2 Overview

The main idea behind our project is to define the structure of a given Java program and to export collection of constraints that describes this structure. After analyzing these constraints, if possible, we generate all instances that satisfy the constraints.

By translating these instances to Java and then generating Java unit tests, we can obtain a good code coverage automatically. The guiding principle of the program is the following :

1.3 Algorithm

1.3.1 Java to Alloy

Generating Alloy source code

Base model The base model is the Alloy meta-model. It is the base of the generated java to Alloy program. The model is composed of five major signatures: types, objects, contructor calls, methods and methods calls.

Types A type is an abstract signature representing the type of an object. All the types of the program to be analysed will extend this signature.

```
abstract sig Type{
}
```

Objects An object represents an instance in the Java program. Every object has a type and a signature.

```
sig Object{
    type : Type,
    constructor : ConstructorCall
}
```

Methods, constructors & calls A very important thing in order to simulate an execution trace, is to have the trace of execution with successive methods calls and not only instances of objects of simple type declaration. Successive methods calls can be presented in a linked list, so later on, simply by getting the head of the list we can browse a gernerated instance and we will have an access to method calls. The following Alloy code represents how we have represented methods in Alloy model.

```
abstract sig Method {
    paramTypes : seq Type,
    receiverType : Type
}

abstract sig Call{
}

one sig Begin extends CallWithNext{
}
```

```
one sig End extends Call{
}

sig ConstructorCall{
    paramTypes: seq Type
}

abstract sig CallWithNext extends Call{
    nextMethod : Call
}

sig MethodCall extends CallWithNext{
    receiver : Object,
    method : Method,
    params : seq Object
}
```

Constraints Methods in every instance generated later on should respect numerous constraints, such like a method not being able to call itself. The following Alloy code represents the constraints used in the Alloy model.

```
------ Method Constraints-------

-- There should be one and only method.next= end
fact{
    one m : MethodCall • m.nextMethod in End
}

-- Method call can't be linked to itself
fact{
    all mc : MethodCall • mc.nextMethod≠mc
}

-- Reciever has the right type
fact{
    all mc : MethodCall •  mc.receiver.type=mc.method.receiverType
}

-- Call can't be done twice
fact{
all mc: MethodCall • no mc2 : MethodCall •  (mc2 in mc.^nextMethod) ∧ (mc2.nextMethod=mc)
}

-- All methods calls has been called
fact{
    all mc : MethodCall • one c : CallWithNext • c.nextMethod=mc
}

-- Object which calls a method has the right type
fact{
    all mc:MethodCall • mc.receiver.type=mc.method.receiverType
```

```
}
```

Arguments Primitve types has been included in the base model. All other used types in a given program are generated at runtime.

```
------- Param Constraints ------

-- Types verification
fact{
    all mc : MethodCall •validParam[mc.method,mc]
}

pred validParam[method : Method, call : MethodCall]{
        call.params.type=method.paramTypes
        #call.params=#method.paramTypes
    all  pt : method.paramTypes.elems • all p : call.params.elems • call.params.idxOf[p]=
        method.paramTypes.idxOf[pt] implies pt=p.type
}

-------- Primitive Types ------------

one sig Gen_Double extends Type{}
one sig Gen_Integer extends Type{}
one sig Gen_Float extends Type{}
one sig Gen_Boolean extends Type{}
one sig Gen_Byte extends Type{}
one sig Gen_Character extends Type{}
one sig Gen_Long extends Type{}
one sig Gen_Short extends Type{}
```

Generating Alloy Code To generate Alloy instances for an existing Java program we used Spoon(1.4). Spoon uses an AST (Abstract syntax tree)[3] to browse the structure of a specified program. With informations collected in the AST, we are able to generate the code of Alloy program corresponding to the initial Java program. The generated code is then grafted to the base model. The result is a complete Alloy model inside the file *FinalGen.als*.

Data: Java source code
Result: Alloy model
foreach *class c* **do**
  Modify the name of class from *package.class* to *package_class*;
  Add *ClassName* as a type to Alloy file;
  foreach *public method m in c* **do**
    Create method name as: $className\_MethodName\_NumberOflineInJavaFile$;
    Add a *Sig* extending $Method\{\}$ representing the method m into Alloy model;
    Add to Alloy model number of method parameters as a $fact$;
    foreach *Parameter p in m* **do**
      Add to Alloy model the type of p as a $fact$;
    end
    Add to Alloy model the type of *receiver* as a $fact$;
  end
  foreach *public constructor ct in c* **do**
    Add a *Sig* extending $ConstructorCall\{\}$ representing the constructor *ct* in to Alloy model;
    Add to Alloy model number of constructor parameters as a $fact$;
    foreach *Parameter p in ct* **do**
      Add to Alloy model the type of p as a $fact$;
    end
  end
end

Algorithm 1: How to transfer Java code to Alloy model

An example of the results output for a method:

```
sig package_myClass extends Type{}
sig package_myClass_method_1 extends Method{}
fact{
  #package_myClass_method_1.paramTypes=3
  package_myClass_method_1.paramTypes[0]=typeArg1
  package_myClass_method_1.paramTypes[1]=typeArg2
  package_myClass_method_1.paramTypes[2]=typeArg3
  package_myClass_method_1.receiverType=X
}
```

and for a constructor:

```
one sig ftp_client_FTPRequestHandler_1_36 extends ConstructorCall{}
fact{
  #ftp_client_FTPRequestHandler_init1_36.paramTypes=3
  ftp_client_FTPRequestHandler_init1_36.paramTypes[0]=ftp_shared_FTPDatabase
  ftp_client_FTPRequestHandler_init1_36.paramTypes[1]=ftp_shared_FTPServerConfiguration
  ftp_client_FTPRequestHandler_init1_36.paramTypes[2]=ftp_server_command_FTPCommandManager
}
```

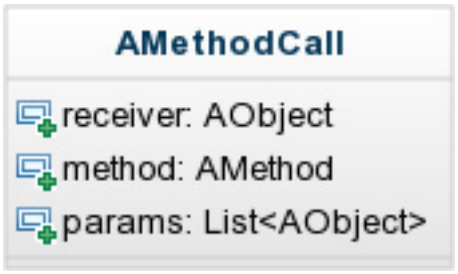
1.3.2 Alloy to Java

Generating Java unit tests

Translate Alloy signatures to Java classes To make Alloy solutions browsing easier, we translated every signature in the Alloy model into Java classes. For example, for the signatures *Method* and *MethodCall*, we have the following classes:

```
abstract sig Method {
    paramTypes : seq Type,
    receiverType : Type
}
```

```
sig MethodCall extends CallWithNext{
    receiver : Object,
    method : Method,
    params : seq Object
}
```



Modelizing Alloy instances in Java Using Alloy Analyzer we execute the generated Alloy source code. Alloy Analyzer generates all possible instances. Each instance is a solution, we can obtain a solution using A4Solution object. A4Solution object has a method *satisfiable* to check if the solution is valid and a method *next* to go the next possible solution.

Generating tests In order to generate the code of Java unit tests, we used CodeModel, which allows us to generate Java classes in a simple way.
We browse Java modelized solution and generate the variables used to call methods and the variables passed as methods parameters.
For each solution we use the execution trace. Firstly, we initalize all the necessary types for the receiver method, then all the veriables that will be used in parameters.
If an exception occurs during the execution of the tests, its stack trace is logged in a file, and

the number of different exceptions thrown is written in the file.

Data: Alloy instance modelized in Java
Result: Java unit test
foreach *Alloy Instance modelized in Java* **do**
  Get the head of method calls list;
  while *methods calls are not finished* **do**
    Create method signature;
    Add *org.Junit.Test* annotation ;
    Add *ThrowException* to method signature;
    Create a *trycatch* block;
    Inside the *try* block:{
    - Declare all variables
    - Call constructors
    - Add method calls with the necessary parameters
    };
    Inside the *catch* block:{
    - Log the exception in a file
    - Rethrow it
    };
  end
end

Algorithm 2: How to translate an Alloy instance modelized in Java into Java unit tests

An example of the results output:

```
@BeforeClass
public static void initExceptionBuilder() throws IOException {
    ExceptionLogger.initLogFile();
}

@Test
public void test0() throws Exception {
  try {

  }catch(java.lang.Exception x){
    ExceptionLogger.logException(x);
    throw(x);
  }
```

```
}

@AfterClass
public static void closeExceptionBuilder() throws IOException {
    ExceptionLogger.closeLogFile();
}
```

1.4 Implementation

In order to realise our project we have used four different techonologies: Spoon, Alloy, Alloy Analyzer and CodeModel.
The main parts of our project is generating Alloy source code from a given Java program, then generating Alloy instances. Thereafter modelizing these instances in Java to, finally, convert these modelized instances to Java unit tests.

Spoon

In order to analyze and transform source code, we needed an efficient and powerful library. We have chosen Spoon, a high-quality open-source library created and maintained by INRIA (French Institute for Research in Computer Science and Automation (French: Institut national de recherche en informatique et en automatique)). Providing a complete and fine-grained JAVA metamodel, Spoon enables us to perform effortless treatments in differents parts of code. These treatments are performed by processors, which are able to browse, modify, or even add any program element (class, method, field, statements, expressions...).[4][5] Spoon can also be used on validation purpose, to ensure that your programs respect some programming conventions or guidelines, or for program transformation, by using a pure-JAVA template engine.[6]

Alloy

Alloy is a language for describing structures and a tool for exploring them. It has been used in a wide range of applications from finding holes in security mechanisms to designing telephone switching networks. An Alloy model is a collection of constraints that describes (implicitly) a set of structures, for example: all the possible security configurations of a web application, or all the possible topologies of a switching network.[7]

Alloy Analyzer

Alloy Analyzer, is a solver that takes the constraints of a model and finds structures that satisfy them. It can be used both to explore the model by generating sample structures, and to check

properties of the model by generating counterexamples.[7]

CodeModel

CodeModel is a Java library for code generators; it provides a way to generate Java programs in a way much nicer than PrintStream.println(). With CodeModel, we can build the java source code by first building AST (Abstract syntax tree)[3], then writing it out as text files that is Java source files.[8]

Chapter 2

Evaluation

2.1 Performance

The performance is the main drawback of our tool. On a small project (i.e. $<= 5000 LineOfCode$), JunitMe2.0 behaves pretty well, generating thousands of tests in less than one minute. But it can be very slow on bigger projects, execution taking several minutes.
This can't be avoided, because many instances are taken in consideration by the Alloy solver.

2.2 Ease of use

JunitMe2.0 is very easy to use. Only three parameters (the last one is optional) are needed for the execution of the program. The command line to execute is the following :
$java - jar JunitMe2.0.jar < projectPath >< nbTests >< alloyRunCount >$
Therefore, the program will analyse the project located at $projectPath$, execute the resulting Alloy model with a run count fixed to $alloyRunCount$ (if this parameter isn't specified, its default value will be 8), and generate $nbTests$ tests for this project.

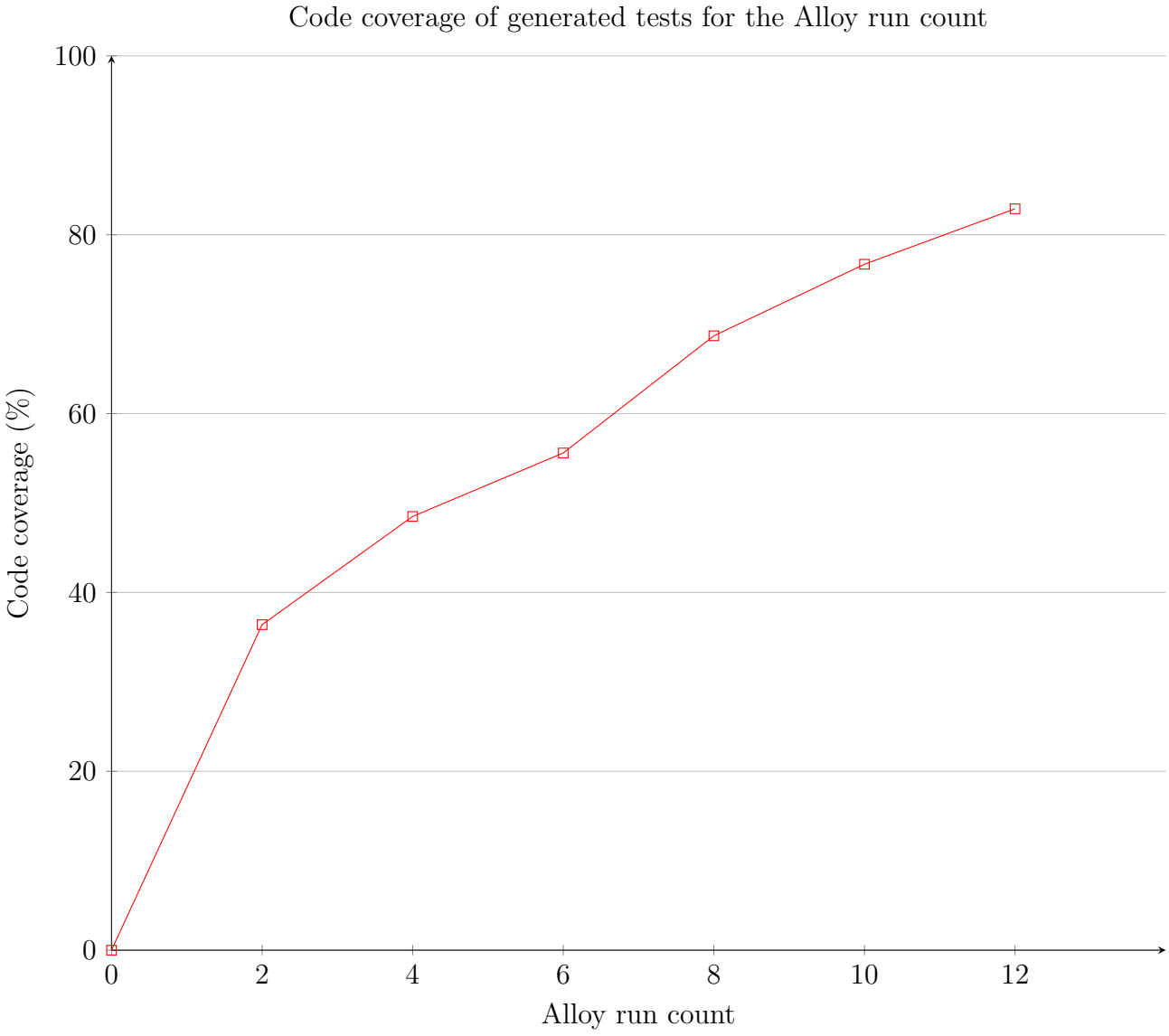
2.3 Validation

Our project has been tested on three projects from the SF100 benchmark [9] : Jipa [10], a simple interpreter in Java, Inspirento [11], a nice journal maker, and Newzgrabber [12], an application used for retrieving file attachments to newsgroup articles. We fixed the number of generated tests to 10000, and measured the code coverage by varying the alloy run count. The tool used to calculate the coverage is EclEmma.[13]

Project name	Coverage percentage
Jipa	82.9%
Inspirento	78.0%
Newzgrabber	87.2%

Figure 2.1: Code coverage of generated tests with the Alloy run count fixed to 8 and the number of tests fixed to 10000

The increase of the Alloy run count improved the tests code coverage. For example, the relation between the Alloy run count and the percentage of covered code for the project Jipa (which is a small project) is represented in the following plot :



2.4 Limitations

Despite its efficiency and reliability, our tool has some limitations, mainly in the area of types that Alloy model covers.

At the time of writing, JunitMe2.0 cannot instantiate generic, parameterized types and arrays in the tests. Interfaces and abstract types also cannot be instanciated in the tests, so the tool just skips them during the construction of the Alloy model.

Fortunately, these functionalities can be implemented easily in the future, the design of the code being generic enough to allow so.

Furthermore, for some Java open source projects, Alloy could reach its limit and a *Translation capacity exceeded* exception would happen even for a relatively small number of tests and a small Alloy run count.

Conclusion

Using Spoon Java library to analyze and trasform source code, Alloy a language and tool for relational models, Alloy Analyzer a solver that takes the constraints of a model and finds structures that satisfy them and CodeModel a Java library for code generators we have succeeded in extending a tool capable of generating Java unit tests a given Java program. Our tool can verify that there no actual error exists for an application. This tool can be extended in the future to be able to treat a bigger variety of Java programs. Today, our tool is able to generate Java unit tests that can achieve up to 88% of code coverage for 10000 tests.

Bibliography

[1] Valentin Lefils and Quentin Marrecau. Génération de tests junit avec alloy. http://static.monperrus.net/iagl/2014/rendu3-alloy-test-data-generation/01_Junit_Generation_Marrecau_Lefils/Rapport.pdf, 2015.

[2] Valentin Lefils and Quentin Marrecau. Junitme (github). https://github.com/user4me/JunitMe.

[3] Wikipedia. Abstract syntax tree. https://en.wikipedia.org/wiki/Abstract_syntax_tree.

[4] INRIA. Spoon - source code analysis and transformation for java. http://spoon.gforge.inria.fr/.

[5] Martin Monperrus, Renaud Pawlak, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of java source code. https://hal.inria.fr/hal-01078532v2/document.

[6] Spoon - javasource. http://java-source.net/open-source/code-analyzers/spoon.

[7] MIT. Alloy: Official website. http://alloy.mit.edu/alloy/.

[8] SUN. Codemodel. https://codemodel.java.net/.

[9] SF100 Benchmark. Sf100 benchmark. http://www.evosuite.org/subjects/sf100/.

[10] Jipa. A simple interpreter in java. http://sourceforge.net/projects/jipa.

[11] Inspirento. A nice journal maker. http://inspirento.sourceforge.net/.

[12] Newzgrabber. An applic ation used for retrieving file attachments to newsgroup articles. http://newzgrabber.sourceforge.net/.

[13] EclEmma. Eclemma, java code coverage for eclipse. http://eclemma.org/.