Automatic Test-cases Generation

A Project Report

by

Sushant Shambharkar
(153050081)

under the guidance of

Prof. Amey Karkare

Indian Institute of Technology Bombay
Mumbai 400076 (India)

3 May 2016

Abstract

ATG (Automatic Test-cases Generation) is a tool which generates a minimal test case suite for a given C program. Minimal in the sense that, the suite will have some upper bound and will contain only test cases which maximize some measure of test case performance like line coverage, branch coverage, etc.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Test cases generation can also be considered as White box testing. White box testing is a method of test-cases generation where you can see the control flow of the given code. It basically comes down to exhaustive path search. Find all the paths in the program and create test-case for each path. State of art Klee[2] is one of such tool.

The problem with Klee and other tools like Klee is that they try to do an exhaustive search and explore all the paths in the given code. Klee uses pruning when there is path explosion but it is not that effective. In most of the cases where the code is small enough Klee is able to explore all paths in the program, but in the process, it generates many redundant test cases. Many time this test suite will be so big that all test-cases my not run in time. To overcome this problem we need to implement a test-case selection technique which will select the important test cases for the given program. This important feature can be user defined like line coverage, branch coverage, path coverage, line/branch ratio, etc.

1.1 Types of coverage

Consider following C program.

```
1 int find(int input, int c1, int c2, int c3) {
2   int x = input;
3   int y = 0;
4   if (c1)
5     x++;
6   if (c2)
7     x--;
```

```
8   if (c3)
9     y=x;
10  return y;
11 }
```

1.1.1 Line coverage

Also called as Statement coverage. Test-suite should be such that every line(Statement) of the code is touched. For the above code find(x, 1, 1, 1) will give 100% line coverage.

1.1.2 Branch coverage

Every if-statement has two branches. If there are no nested if-statements then only two test-cases can cover 100% branches. In above code find(x, 1, 1, 1) and find(x, 0, 0, 0) will cover all branches.

1.1.3 Path coverage

Visualize the above code in a form of a binary tree. You can see that the above two testcases cover only two paths 1-1-1 and 0-0-0. Other 6 paths are still uncovered. To get to 100% path coverage we need other 6 test cases.
find(x, 0, 0, 0)
find(x, 0, 0, 1)
find(x, 0, 1, 0)
find(x, 0, 1, 1)
find(x, 1, 0, 0)
find(x, 1, 0, 1)
find(x, 1, 1, 0)
find(x, 1, 1, 1)
These test-cases will cover all the paths in the given program.

This is what Klee does. It explores all the paths in the given code. But we can see that line coverage and branch coverage is also a good measure for test-cases selection.

1.2 Symbolic execution

Analyzing a program to find inputs which caused different part of a program to execute is called symbolic execution. An interpreter follows the program, assuming symbolic values

for inputs variables rather than concrete values (actual inputs). Through this process interpreter arrives at expressions in terms of those symbolic values. SAT problem can be reduced to this problem which means this problem is in NP-complete.

1.3 Path Explosion

Symbolically executing all paths in a program does not scale well to larger programs. The number of feasible paths in a program grows exponentially with an increase in branching statement. It can even be infinite in the case of programs with unbounded loop iterations. Solutions to the path explosion problem generally use either heuristics for path-finding to increase code coverage, reduce execution time by parallelizing independent paths, or by merging similar paths.[1]

Chapter 2

Implementation

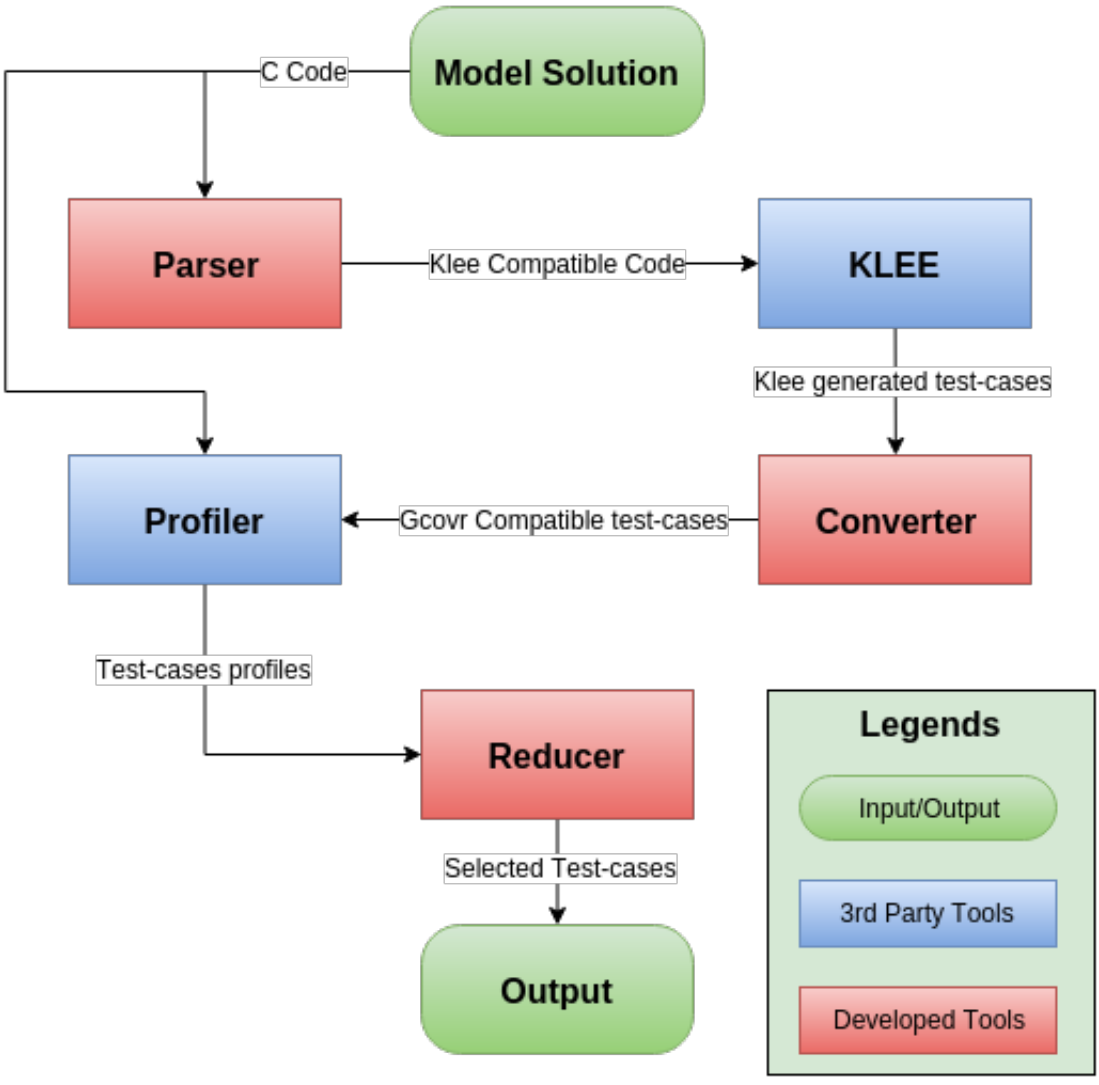

Figure 2.1: ATG Flowchart

We are going to take a simple example and walk it through the different parts of the project.

```
#include<stdio.h>
int get_sign(int x) {
  if (x == 0)
    return 0;
  if (x < 0)
    return -1;
  else
    return 1;
}
int main() {
  int a;
  scanf("%d",&a);
  printf("%d", get_sign(a));
}
```

2.1 Model Solution

This is C code for which test cases needs to be generated. The code needs to take input from the console (command line arguments are currently not supported). The code can be as simple as FindOddEven.c or can be complex as SudokuSolver.c, but when it comes to the generation of the test case, the time required will depend on a number of independent paths in the code.

Parser module is still in development phase hence, when writing the code, special guidelines need to be followed. Please refer user manual for the guidelines. In the above code, we need to identify statements which are used for taking input from the console. Ones identified, we need to move these statements into a function. Name of the function is customizable.

```
#include<stdio.h>
void isolate(int *a, int size, char name[]){
    scanf("%d",a);
}
int get_sign(int x) {
  if (x == 0)
    return 0;
  if (x < 0)
```

```
9     return -1;
10  else
11    return 1;
12 }
13 int main() {
14   int a;
15   isolate(&a, sizeof(a), "a");
16   printf("%d", get_sign(a));
17 }
```

2.2 Parser

The parser does the job of converting C code to Klee compatible code. This module is still in development hence, it requires some user support. The input to Parser in C code and output is Klee compatible C code. Parser finds the function which we used for isolating the scanfs and replace the function with Klee symbolic function which makes concrete variables symbolic.

```
1 #include<stdio.h>
2 int get_sign(int x) {
3   if (x == 0)
4     return 0;
5   if (x < 0)
6     return -1;
7   else
8     return 1;
9 }
10 int main() {
11   int a;
12   Klee_make_symbolic(&a, sizeof(a), "a");
13   printf("%d", get_sign(a));
14 }
```

2.3 Klee

KLEE is a symbolic virtual machine built on top of the LLVM compiler infrastructure, and available under the UIUC open source license. Klee does an exhaustive search on given code and generates test cases. If the path tree is small, Klee will be able to generate

test case of every path. If the path tree is not fully searchable the Klee will apply some pruning methods or time limited DFS/BSF. The test cases generated by KLEE are written in files with extension .ktest. These are binary files, which can be read with the ktest-tool utility. The test cases generated by Klee for above program is following.

```
$ ktest-tool --write-ints Klee-last/test000001.ktest
ktest file : 'Klee-last/test000001.ktest'
args       : ['get_sign.o']
num objects: 1
object    0: name: 'a'
object    0: size: 4
object    0: data: 1

$ ktest-tool --write-ints Klee-last/test000002.ktest
...
object    0: data: -2147483648

$ ktest-tool --write-ints Klee-last/test000003.ktest
...
object    0: data: 0
```

2.4 Converter

This module extracts the actual input for the program from generated ktest files. ktest files generated by Klee contains much more information than just program input. This module takes important information into consideration and creates test files. The converted test cases for above program is following.

```
$ cat test000001
1
$ cat test000002
-2147483648
$ cat test000003
0
```

2.5 Profiler

This module uses Gcovr[5] for program profiling. Profiling is a form of dynamic program analysis that measures, code coverage of a program by the particular test case, the usage

of particular instructions, or the frequency and duration of function calls. Gcovr is one of these profiling tools which provides analysis of a test case on given program in XML format which is easy to parse. This module takes the original code as input and applies every test case given by Converter and create a profile of each test case. This profile contains information like line coverage, branch coverage, line hit rate, etc. Following is the profile of code when test000001 given as input.

```
<?xml version="1.0" ?>
<!DOCTYPE coverage
  SYSTEM 'http://cobertura.sourceforge.net/xml/coverage-03.dtd'>
<coverage branch-rate="0.5" line-rate="0.777777777778" timestamp="
    1462317863"
 version="gcovr 3.2">
 <sources>
  <source>.</source>
 </sources>
 <packages>
  <package branch-rate="0.5" complexity="0.0" line-rate="0.777777777778
   "
   name="">
   <classes>
    <class branch-rate="0.5" complexity="0.0" filename="get_sign.c"
     line-rate="0.777777777778" name="get_sign_c">
     <methods/>
     <lines>
      <line branch="false" hits="1" number="3"/>
      <line branch="true" condition-coverage="50% (1/2)" hits="1"
    number="4">
       <conditions>
        <condition coverage="50%" number="0" type="jump"/>
       </conditions>
      </line>
      <line branch="false" hits="0" number="5"/>
      <line branch="true" condition-coverage="50% (1/2)" hits="1"
    number="7">
       <conditions>
        <condition coverage="50%" number="0" type="jump"/>
       </conditions>
      </line>
      <line branch="false" hits="0" number="8"/>
```

```
30      <line branch="false" hits="1" number="10"/>
31      <line branch="false" hits="1" number="13"/>
32      <line branch="true" condition-coverage="50% (1/2)" hits="1"
    number="15">
33       <conditions>
34        <condition coverage="50%" number="0" type="jump"/>
35       </conditions>
36      </line>
37      <line branch="false" hits="1" number="16"/>
38     </lines>
39    </class>
40   </classes>
41  </package>
42 </packages>
43 </coverage>
```

2.6 Reducer

This is a major part of the ATG (Automatic test cases Generation). The input to Reducer is profiles of every test case. Using these profiles, Reducer creates a matrix who have the following form.

$$M_{ij} = \begin{cases} 1, & \text{if line i is covered by test case j} \\ 0, & \text{otherwise} \end{cases}$$

For the profile generated in the last section, we get the following matrix. Column represents line number in code where a row is the test case.

In above example, all test cases need to be selected if we want 100% line coverage.

2.7 Output

Finally, we get output as in selected test cases. We get other information too like, lines which are not covered, etc.

```
1 cat output.txt
2 Total number of lines : 14
3 Number of lines not covered : 0
4 Lines which are not covered : []
5 Total number of test cases : 3
```

```
6 Number of test cases selected : 3
7 test cases selected : [0,1,2]
```

Chapter 3

NP-Hardness of ATG

3.1 Set Cover Problem

Given a universal set U of n elements, a collection of subsets of U, S = S1, S2...,Sm where every subset Si has an associated cost. Find a minimum cost sub-collection of S that covers all elements of U.

```
Example:

   U = {1,2,3,4,5,6}
   S = {S1,S2,S3,S4}

   S1 = {4,1,3},     Cost(S1) = 5
   S2 = {2,5},       Cost(S2) = 10
   S3 = {1,4,3,2},   Cost(S3) = 3
   S3 = {1,4},       Cost(S3) = 15

Output: Minimum cost of set cover is 13 and
        set cover is {S2, S3}

There are two possible set covers {S1, S2} with cost 15
and {S2, S3} with cost 13.
```

There is no polynomial time solution available for this problem as the problem is a

known NP-Hard problem. There is a polynomial time Greedy approximate algorithm, the greedy algorithm provides a Log(n) approximate algorithm.

3.1.1 Approximate Greedy Algorithm

Let U be the universe of elements, S1, S2, ... Sm be the collection of subsets of U and Cost(S1), C(S2), ... Cost(Sm) be costs of subsets.

```
1) Let I represent set of elements included so far.
    Initialize I = {}

2) Do following while I is not same as U.
    a) Find the set Si in {S1, S2, ... Sm} for which
    Cost(Si) / |Si - I| is minimum.
    b) Add elements of above picked Si to I, i.e.,
    I = I U Si

Example:
Let us consider the above example to understand Greedy Algorithm.

First Iteration:
I = {}

The per new element cost for S1 = Cost(S1)/|S1 - I| = 5/3

The per new element cost for S2 = Cost(S2)/|S2 - I| = 10/2

The per new element cost for S3 = Cost(S3)/|S3 - I| = 3/4

The per new element cost for S4 = Cost(S4)/|S4 - I| = 15/2

Since S3 has minimum value S3 is added, I becomes {1,4,3,2}.

Second Iteration:
I = {1,4,3,2}
```

```
The per new element cost for S1 = Cost(S1)/|S1 - I| = 5/0
Note that S1 doesn't add any new element to I.

The per new element cost for S2 = Cost(S2)/|S2 - I| = 10/1
S2 adds only 5 to I.

The per new element cost for S3 = Cost(S3)/|S3 - I| = 15/0
S3 doesn't add any new element to I.

The greedy algorithm provides the optimal solution for above
example, but it may not provide optimal solution all the time.
Consider the following example.

S1 = {1, 2}
S2 = {2, 3, 4, 5}
S3 = {6, 7, 8, 9, 10, 11, 12, 13}
S4 = {1, 3, 5, 7, 9, 11, 13}
S5 = {2, 4, 6, 8, 10, 12, 13}
S6 = {1, 2, 3}

Let the cost of every set be same.

The greedy algorithm produces result as {S3, S2, S1}

The optimal solution is {S4, S5}
```

This greedy algorithm is Log(n) approximate.[4]

3.2 Reduction of Set Cover to ATG

In ATG, we have the problem of test-cases selection. In this problem, we have a set of test-cases. Each test-case covers some line of the code. We can consider all line in the code as universal set U, but there is a problem. A code can have unreachable parts in

it. Unreachable code is part of the program which can never be executed because there exists no control flow path to the code from the rest of the program.[3] So there will be no test-case which can touch these lines.

To tackle this problem, we will consider U as a union of all the test-cases. This way, we will avoid unreachable part of code. Now we can reduce Set Cover problem to ATG.

Universal Set = Union of lines covered by each test case =

Subsets = lines covered by each test case

$\bigcup_{i=1}^{N} T_i = U$

So, We have reduced Set Cover to ATG. The reduction is done in polynomial time (O(1)). Hence, ATG is in NP-Hard.

Chapter 4

Future Work

4.1 Parser

Parser requires some user support in order to convert c code to Klee compatible code. This user support can be removed. Parser just needs to parse the c code and find the parts in the program through which input is being taken. There can be for-loops for array input, while-loop for menu-driven input, if-else, etc. However, all of these can be replace by only one command.

```
klee_make_symbolic(&a, sizeof(a), "a");
```

4.2 Klee

Klee is 3rd party tool. If we want to use some other tool we can replace Klee. The main script is flexible and it will allow any other tool to be used.

4.3 Reducer

In the set cover problem, we augment each set with some cost which needs to be minimized. Similarly in ATG, each test-case have some cost. Current implementation considers factors like line hit count, execution time, etc. These factors can be tuned so that selected test-case set will not only cover maximum part of the code but also will the important ones.

Chapter 5

User Manual

5.1 Installation through Docker

Installation ATG is pretty easy. The best way to install ATG is through docker[**?**]. Please find a full guide for installation of docker at the following URL.

https://docs.docker.com/v1.8/installation/ubuntulinux/

To pull down the latest build of a particular Docker image run:

```
$ docker pull sam1064max/atg
```

If you want to use a tagged revision of KLEE you should instead run:

```
$ docker pull sam1064max/atg:<TAG>
```

Where <TAG >is one of the tags listed on the DockerHub. Typically this is either ”latest” or a version number (e.g. 1.0).

After installing Docker, pull the docker image of ATG. Please run the following command through root user.

```
docker pull sam1064max/atg
```

After pulling the image you have to start the image using the following command.

```
docker run -ti --name=atg_container --ulimit='stack=-1:-1' sam1064max
 /atg
```

This will create a docker container named atg_container. Next time you want to run the container, just run the following command.

```
docker start -ai atg_containe
```

Ones inside the container to start web service run following command.

```
service apache2 start
```

If you want to play with ATG, go to ATG folder and run ./Run.sh.

```
Usage: ./Run.sh [options] klee_code c_code
  -o      Use other test-cases generator (def: klee)
  -p      Use parser to make c code klee compatible (def: false)
  -d      Delete the temp files after completion (def: false)
```

kleecode and ccode folder contains some sample codes.

```
./Run.sh kleecode/maze.c ccode/maze.c
```

The Rest API follows following JSON format

Input :

```
{
    "env":<environment_of_the_program>,
    "assignment_id":<id_of_the_programming_assignment>,
    "code_id":<code_id_of_the_code_under_compilation>,
    "klee_code":<klee_compatible_c_code>,
    "c_code":<c_code>
}
```

Ouput:

```
{
    "err":<error>,
    "type":<type_of_output>,
    "output":<output_content_to_be_displayed>
}
```

5.2 Installation on native Linux

I will highly recommend 5.1 because it's very hard to build Klee on native Linux. To build Klee see the guidelines from the following site.

http://klee.github.io/build-llvm34/

After building Klee, clone the Github repository of ATG.

```
git clone https://github.com/sam1064max/ATG
```

Github version has no web container. Rest is same as 5.1.

Chapter 6

Developer Manual

Read 5.1 for the installation of ATG. Most of the scripts are fully developed, however, because I'm continuously fixing the bugs, actual scripts in the on-line repository can be slightly different. If you feel confused, please check the revision history.

6.1 Run.sh

This the main script. All other parts of ATG are invoked by this script in a sequential manner.

```
#!/usr/bin/env bash

set -o errexit
set -o pipefail
set -o nounset

TEST_GENERATOR="klee"
PARSER="false"
DELETE="false"

while getopts 'o:p:d:' opt ; do
  case $opt in
    o) TEST_GENERATOR=$OPTARG ;;
    p) PARSER=$OPTARG ;;
    d) DELETE=$OPTARG ;;
  esac
done

```

```
19 if [ $# -lt 2 ]; then
20   echo "Usage: $0 [options] klee_code c_code"
21   echo "   -o      Use other test-cases generator (def: $TEST_GENERATOR)
       "
22   echo "   -p      Use parser to make c code klee compatible (def:
       $PARSER)"
23   echo "   -d      Delete the temp files after completion (def: $DELETE)
       "
24   exit 1
25 fi
26
27 KLEE_CODE=$1
28 C_CODE=$2
29
30 if [ $TEST_GENERATOR != "klee" ]; then
31   echo "$TEST_GENERATOR is currently not available. Please use <klee>"
32   exit 1
33 fi
34
35 if [ $PARSER = "true" ]; then
36   echo "klee code converter is currently under process. Please convert
       the code manually."
37   exit 1
38 fi
39
40 #Klee-------------------------------------
41 mkdir -p log
42 clang -emit-llvm -g -c $KLEE_CODE -o temp.bc
43 klee --libc=uclibc --posix-runtime temp.bc > consol_output.txt
44 mv -t log/ consol_output.txt temp.bc
45 #Klee-------------------------------------
46
47 #Converter
48 ./klee2test.sh "klee-out-0/*.ktest" "cases"
49
50 #Profiler
51 ./cases2xml.sh $C_CODE "cases/*" "xml"
52
53 #Reducer
54 python3 Reducer.py "xml/*" > output
```

```
55
56 cat output
57
58 if [ $DELETE = "true" ]; then
59   rm -fr cases/ xml/ klee-*
60 fi
```

You can get the usage of the script by running,

```
1 ./Run.sh
```

In this script, TEST_GENERATOR is the tool we want to use of test-case generation. Currently, we have only Klee. If you want to use some other tool, replace the klee part of the script. Respectively, you have to write the converter. Rest of the script you can understand by reading comments.

6.2 cases2xml.sh

This is the Profiler script. It uses gcovr for profiling each test-case.

```
1 mkdir -p $3
2 for f in $2;
3 do
4     g++ -fprofile-arcs -ftest-coverage $1
5     ./a.out < $f > log/gcovr_consol_output.log
6     gcovr -r . --xml > "xml/$(basename $f).xml"
7     mv -t log/ a.out *.gc*
8 done
```

This script creates a folder "xml". Inside the folder, there is a profile for each test case in XML format.

6.3 Reducer.py

This script has two part. First part handles the parsing of profile XML. Second part solves set cover problem with the greedy approximation. Command line arguments taken by the scripts are folder path of profiles and cost function. The cost function is used for calculating the cost of each test-case which we either have to minimize or maximize.

References

[1] P. Boonstoppel, C. Cadar, and D. Engler. Rwset: Attacking path explosion in constraint-based test generation. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–366. Springer, 2008.

[2] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[3] S. K. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming languages and Systems (TOPLAS)*, 22(2):378–415, 2000.

[4] U. Feige. A threshold of ln n for approximating set cover. *Journal of the ACM (JACM)*, 45(4):634–652, 1998.

[5] B. J. Gough. foreword by richard m. stallman. *An Introduction to GCC-for the GNU compilers gcc and g+*, 2005.

Appendix A

Supporting Material