www.ijecs.in
International Journal Of Engineering And Computer Science ISSN:2319-7242
Volume 5 Issue 8 August 2016 Page No. 17662-17666

Automatic Generation of Coverage tests for System Programs

Chaturvedula Pratyusha[1], Naveen Kumar[2], K. Rajasekhar[3]

[1]M. Tech (Software Engineering), School of Information Technology, Jntuh

Email:pratyushach518@gmail.com

[2] Assistant Professor,School of Information Technology, Jntuh

[3] Scientist' G', Defense Research and Development Laboratory, kanchanbagh.

Abstract:

Software testing is one of the critical activity for the organizations who spend lot of time and cost for the improvement of software quality. Programming testing is characterized as the way toward executing a project with the plan of discovering mistakes. Testing would ensure the correctness and produces reliable software. In order to achieve the quality software, sufficient number of test cases should be designed and tested.To calculate the integrity of test cases and identify that there are no unexpected functionalities, the structural code coverage should be measured like, statement or decision coverage.

Numerous strategies and procedures are advanced in era of test information for different experiments and is considered as one of the vital components of examination region in the organizations. In this study, the automatic generation for test cases are explored. This papermainly focuses on a symbolic execution tool used to generate the optimized test cases. The basic aim of the project is to generate the test cases and to achieve 100% coverage of given structural code including statement coverage, decision coverage and path coverage.

1. Introduction

Most of the IT Organizations generate test cases manually. This is a slow process and the intervention of man power leads to the generation of ineffective test cases and lastly the quality is affected.

Hence, without trying to run the code manually, we run it on the symbolic input. When the program executes based on symbolic input it produces the optimized test cases maintaining set of constraints used for the code coverage. Growing more lines of code and getting accumulated accurately is a major test and immense sum is spent on confirmation and approval.
Code coverage [9] is the cadent to decide the completeness of test cases and make sure that there is no dead code or uncovered code. Hence code coverage is an important test criteria.
Some of the code coverage metrics include:

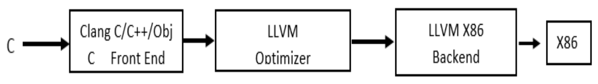
i. Statement coverage: Each statement of the code must be covered with at least one test case.
ii. Decision coverage (also branch coverage): Each conditional statement must be executed in its true and false condition.
iii. Path coverage: Every possible route through the code must be operated by at least one test case.

In this paper I present a symbolic execution tool called KLEE, which is used for obtaining optimized test casesand a coverage tool called GCOV used for representing the lines of code covered.

KLEE:

Klee is a robust optimization tool used to represent the program states completely and obtain the high code coverage. Klee uses simple and straight forward approachto accord with independent conditions. These features improve the performance of Klee.

Klee runs on top of LLVM. LLVM (Low Level Virtual Machine) is a just in time compiler which is used to generate and execute machine code. Llvm is intended for run time, arrange time advancement of programs written in various programming dialects. Llvm is also used to design the compiler front end and back end. Front end is responsible for parsing and validating the errors in input code. Then it translates the parsed code to llvm IR (Intermediate Representation). Klee is implemented with the help of this llvm IR.



LLVM is a research based project to provide modern compilation strategy to support static and dynamic compilation of different programming languages. Hence, LLVM has grown as an umbrella project comprising of subprojects used as open source projects that can be used for academic research. The sub-projects of LLVM are:

i. LLVM Core Libraries: - Provides a target-independent optimizer.
ii. Clang: - Tool that finds bugs in the code and provides to build source level tools.
iii. OpenMP: - Provides runtime for the implementation in clang.
iv. Klee: - Produces a test case in an event to detect a bug. Klee implements a symbolic virtual machine to evaluate dynamic paths and to prove the properties of functions.

Observations on Klee:

1. Klee automatically generates the test cases. These test cases cover the total lines of the program to maximum when run on the coverage tools.
2. Klee gets more code coverage than the sustained manual effort.
3. Klee finds important errors in heavily tested code.
4. Klee is not limited to low level programming errors. It also finds out the functional errors and other inconsistencies.
5. Klee can also be applied to non-application code.

GCOV:

Gcov is a testcoverage tool. Gcov gives the code coverage analysis. Gcov helps in understanding:

- How regularly each line of code executes.
- What lines of code are really executed.
- How much processing time each section of code employs.

2. Overview

This section explains the functionality of Klee. It illustrates problems common to the programs which relate to complexity and environmental dependencies. The code illustrates two additional common features.

i) If the input code has bugs, Klee finds out and generates test cases.

ii) Second, Klee helps to achieve good code coverage.

The objectives of Klee is to strike each line of executable code in the program and discover risky operations which could cause an error. Klee runs the programs symbolically by generating the constraints that accurately defines the set of values probable on a given path. When Klee discovers an error, Klee resolves the current path constraints to generate a test case that follows same path when run on unmodified version of the program.

Klee is aimed to generate the paths for an unmodified program which follows the same path Klee took. The capacity to run the tests outside of Klee in combination with standard tools is instrumental to diagnose the errors and validating the results.

2.1 Usage

Klee does not need any source modifications and the user can glance real programs with Klee in moments. First compile the code to byte code using LLVM compiler.

Command used for ‘ex’ source file:

```
$ llvm-gcc -emit-llvm -c -g ex.c
```

We get the llvm byte code saved as ex.o

We then run Klee on produced byte code. Command used:

```
$ klee ex.o
```

Klee runs through three paths from our written code where one value is equal to 0, one less than 0, and other greater than 0. It explores that it generated one test case for an each path explored. It generates output in the output directory and shows the number of instructions, completed paths and generated tests. Klee generates the actual test files with extension .ktest.

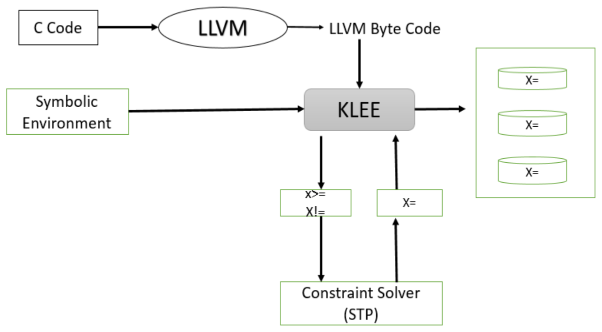
These binary files can be read and tested with the ktest-tool utility.

3. Klee Architecture

Klee is the remodel of a prior execution tool called EXE. Klee capacities as a crossover identifying with a working framework for typical procedure and a translator. Klee is represented as a state for symbolic process. Programs are compiled to llvm which is an instruction set. Klee analyzes this guideline set and maps these instructions to imperatives without evaluation.

Klee is apredictor loop which chooses a state to run and executes single instruction symbolically. This loop proceeds till there are no states left and client characterized timeout is come to.

Conditional branches yield Boolean expressions and changes the position of instruction pointer based on the conditions whether it is true or false. Klee forwards a query to the constraint solver (STP) whether the condition fulfills the present path or not. Now the instruction pointer will be updated to the applicable location. Klee maintains replica of the state to discover both paths, renewing instruction pointer and the path condition. For the risky operations, if an issue is recognized, Klee constructs a test case to incite the flaw and ends the state.



Architecture of Klee

With the other risky operations, load and store instructions produce checks. The upfront demonstration of memory used by checked code is a byte-array. [7] Unfortunately the constraint solver (STP) cannot solve the resultant constraints.

Hence Klee maps memory object in checked code to STP array. Klee is well-optimized for the programs that use symbolic pointers when a dereferenced pointer refers to many objects.Klee simplifies the constraint set by altering prior constraints when new parity constraints are added.

Klee is an instrumentational approach such that the symbolic execution is done at backend during the execution of normal program. It can be easily implemented in C. Llvm byte code or the symbolic executed code is important for Klee to generate the test cases. After the generation of test cases, they are run on the original binaries to describe an instance of the symbolic environment.

4. Experimental Results:

Code coverage is calculated using gcov tool for the statement and decision coverage using the test casesgenerated by Klee. The tool has reported with the 100% statement coverage.

4.1 Coverage Methodology:

The program is run by using the following command in terminal:

```
$ gcc -fprofile-arcs -ftest-coverage ex.c
```

When ./a.out is run, we give the Klee generated test cases as input. It compiles and when

$ gcovex.gcda command is run, it shows:

```
90.00% of 10 source lines executed in file ex.c
```

Further to get the clear idea of what lines are covered,we use the command:

```
cat ex.c.gcov
```

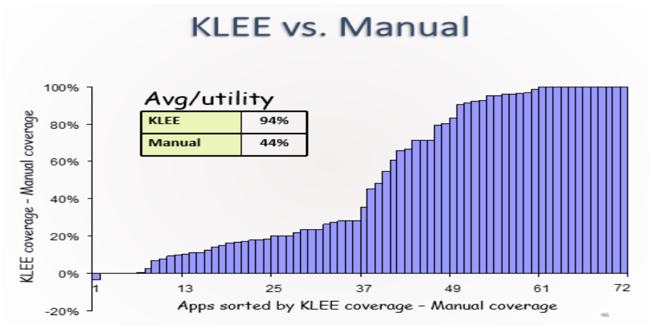
When –b option is used,

```
$ gcov -b ex.c
```

The result is:

- 90.00% of 10 source lines executed in file ex.c
- 80.00% of 5 branches executed in file ex.c
- 80.00% of 5 branches taken at least once in file ex.c
- 50.00% of 2 calls executed in file ex.c

The graph shows coverage of klee generated testcases versus manual test cases:



5.Related Work:

Apart from statement coverage, another aim was to achieve the path coverage. To achieve the path coverage, we need to generate a control dependency graph for the program so that the graph ensures that the program flows in an optimized manner. And when tested with the test cases, it satisfies the path coverage.

To generate this, we basically need C grammar rules and an open source tool called ANTLR. By developing code to accept all the elements of a c program, this tool generates tokens, parser code and lexer code when a C program is given as input. From this, it provides the nodes and links that can be used to generate the control dependency graph. This tool mainly uses the concept of stack, to reach every line of code and to acquire the dependencies for a given program.

Now, when the nodes and links are generated we generate the graph using a tool named Graphviz, which generates required graphs using Dot code.

Lastly, by using the Klee generated test cases, the graph is tested for path coverage of given input program.

A similar research is also done to give 100% results for the coverage. The path coverage gives more expected results through data dependency graph compared to control dependency graph. To generate the data dependency graph,Backtracking method is to be performed and breadth first search graph needs to be generated.

6.Future works:

Attaining path coverage for vast lines of code is not practicable. Hence more optimized testing techniques are to be used. Instrumentation of the code directly using random testing techniques can give feasible results.In future, the reports can be generated in html using clang static analyzer.

7. Conclusion:

The main aim is to give a C program as input, and get the maximum code coverage.Despite many techniques used for the automatic test case generation, Klee has given satisfactory results for the code coverage. The automated test cases are also tested for the path coverage by generating control dependency graph which has given good coverage.

References:

[1] About Klee. www.klee.github.io

[2] Getting started with llvm core libraries. (Bruno Cardoso lopes & Rafael Auler)

[3] Wikipedia : llvm URL : http://en.wikipedia.org/wiki/llvm

[4] Llvm compiler infrastructure project

[5] Architecture of open source applications.

[6] Automatic generation of test cases.

[7] Cristian Cadar,Daniel Dunbar, Dawson Engler – KLEE:

[8] http://www.slideshare.net/shauvik/symbolic-execution-and-klee.

[9] Automated Testcase Generation for Numerical Support functions in Embedded Systems.- Johann Schumann,Stefan-Alexander Schneider.

[10] Control flow graph generation.

[11] J. Bauer and A. Finger, “Test plan generation using formal grammars,” in Proc. 4th Int.

DOI: 10.18535/ijecs/v5i8.42

Conf Software Engineering, 1979, pp. 425-432.

[12] J. Benson, "Adaptive search techniques applied to software testing,"ACM Perform. Eval. Rev., vol. 10, no. 1, pp. 109- 116, Spring 1981.

[13] D. Bird and C. Munoz, "Automatic generation of random self-checking test cases," IBM Syst. J., vol. 22, no. 3, pp. 229-245, 1983.

[14] Www.antlr.org

[15] Bruce A. Cota "Control flow graph as representation language" Winter Simulation Conference, pp 556-559,1994.

[16] Jon Edvardsson "A Survey on automatic test data generation"

Author: Chaturvedula Pratyusha, M.Tech (Software Engineering)

Email: pratyushach518@gmail.com