Test Case Generation For Closed-Loop Controllers

Project Documentation

Nikolaos Vasilakis

nvas@seas.upenn.edu

Version 0.8, January 22, 2012

Contents

1 Introduction

In this report we document the use of different approaches in an effort to conclude towards reachability identification and guided test-case generation for closed-loop controllers. In particular, we are interested in concluding if specific paths are reachable, and, if positive, generate test-cases that are able to reach these paths. An underlying goal is to keep the whole process completely automated.

We make use of the two following tools: KLEE [CDE08], a symbolic execution tool, capable of automatically generating tests that achieve high coverage on a diverse set of programs. BLAST [BHJM07], a software model checker for C programs, which is able to check that the software satisfies specific behavioral properties. Apart from the use of these tools, we implemented a number of automation scripts, that significantly aid building, running the tools and reporting.[*]

We divide our report in the following sections: Section 2 covers the development environment setup and installation of the tools. Section 3 summarizes the steps required to reason about a program manually and gives information on some useful command-line options. Section 4 covers the use of automation scripts for a number of programs and documents numerous attributes of these scripts. Section 5 reports on the comparison of the results. Section 6 gives some hints on possible future directions as well as information on issues we did not manage to overcome yet (and possible solutions).

[*]The use of different languages for different tasks was a conscious choice: we used shell scripting for the building and running the tools; and python for comparing and reporting statistics.

2 Installation

The software was run and tested on Ubuntu Linux (11.04, 2.6.38-13-generic) with gcc version 4.5.2.

The easiest way to install KLEE, is by downloading the self-contained package. After downloading the package, you extract it and insert the bin directory to the PATH, so that you can execute all of the *.cde programs by invoking only their names.

```
wget http://keeda.stanford.edu/~pgbovine/klee-cde-package.v2.tar.bz2
tar -jxvf !!:$:t
klee_path="`pwd`/klee-cde-package/bin"
export PATH=$klee_path:$PATH
```

Regarding BLAST, two versions of the tool are of interest. While the latest version is version 2.5, we ended up using version 2.0 which incorporates test vector generation among other things (e.g., GUI).[†] Note that these features were not maintained after v.2.0. To get both version 2.5 and version 2.0 you run:

```
mkdir blast2.5
wget http://mtc.epfl.ch/software-tools/blast/down\
load/blast-2.5_linux-bin-x86.zip
unzip !!:$:t -d blast2.5

wget http://mtc.epfl.ch/software-tools/blast/down\
load/blast-2.0_linux-bin.tgz
tar -zxvf !!:$:t
mv blast_linux_exe blast2.0
```

More importantly, we need to add the solvers in the 2.0 version:

```
cp -v blast2.5/!(pblast.opt|spec.opt) blast2.0/
```

Now you need to add only one version of BLAST in your path (you can still run the other if you specify the full path to the executable). Here, we add BLAST 2.0 to path:

```
blast_path="`pwd`/blast2.0/bin"
export PATH=$blast_path:$PATH
```

In case you want to switch to BLAST 2.5, you run the following commands:

```
PATH=$(echo $PATH | sed -e "s;:\?$blast_path;;")
PATH=$(echo $PATH | sed -e "s;^:;;")
blast_path="/media/w7/Projects/blast-2.5_linux-bin-x86";
export PATH=$blast_path:$PATH
```

[†]We made extensive use of both version before settling down with version 2.0.

Lastly, in order to grab the latest revision of the automation scripts, you only need to clone the repository with the following command:

```
git clone git@github.com:nvasilakis/glucose.git
```

3 Working with one instance

Here we document how you would work with a simple program.

3.1 KLEE

In this example we are using the following simple program:

```
int test_me(int x) {
  if (x == 0)
 return 0;
  if (x < 0)
 return -1;
  else if ((x < 2) && (x>0)) {
 return 1;
  }
}

int main() {
  int x;
  klee_make_symbolic(&x, sizeof(x), "x");
  test_me(x);
}
```

To compile the program to LLVM bitcode:

```
llvm-gcc.cde --emit-llvm -c -g demo1.c
```

To run KLEE on the bitcode:

```
klee.cde -write-cvcs demo1.o
```

To convert klee results from binary to ascii:

```
ktest-tool.cde klee-last/test*.ktest  > results.txt;
```

Converting cvc results to branch statements:[‡]

[‡]Supposing that the "pathconditions" folder with the java classes is in the same folder, you just change directory in to "pathconditions" with cd before running the java command. The "pathconditions" folder is part of the automation tools found in the git repository.

```
java CreateAssertion ../klee-last/test*.cvc > ../assertions.txt;
```

Now we can view the assertions by issuing:

```
cat -n assertions.txt
```

which gives:

```
 1
 2  ===================================
 3  FILE: ../klee-last/test000001.cvc
 4  ((0!=x) && (x > 1))
 5
 6  ===================================
 7  FILE: ../klee-last/test000002.cvc
 8  ((0!=x) && (x <= 1) && (x >= 0))
 9
10  ===================================
11  FILE: ../klee-last/test000003.cvc
12  ((0!=x) && (x < 0))
13
14  ===================================
15  FILE: ../klee-last/test000004.cvc
16  ((0==x))
```

3.2 BLAST

One can find examples to run BLAST upon, at the BLAST examples page [BH+08]. We are going to use the same program as before; however, now we are going to check if line 7 is reachable:

```
 1 int test_me(int x) {
 2   if (x == 0)
 3  return 0;
 4   if (x < 0)
 5  return -1;
 6   else if ((x < 2) && (x>0)) {
 7  ERROR: goto ERROR;
 8  return 1;
 9   }
10 }
11
12 int main() {
13   int x;
14   test_me(x);
15 }
```

Among many options [B+08], some command line options of particular interest to BLAST are the following:

- -test: Generates test vectors.
- -noblock: Old counterexample analysis
- -nocache: Do not cache theorem prover calls. (less memory)
- -debug: Extensive debugging output
- -traces: Dumps trace for diagnostic purposes.
- -trace-file: Specify the name of the file containing trace information.

To run BLAST 2.5 against our demo2.c program, we issue:

```
pblast.opt demo2.c > bl-results2.5.txt
```

In the output we recognize "Error found! The system is unsafe :-(", which means that the path is reachable.

Therefore, now we switch to BLAST 2.0, in order to (try to) identify the input that can bring the program to the required state (path).[§] We redirect the output to bl-results2.0.txt.

```
pblast.opt -test -noblock -nocache demo2.c > bl-results2.0.txt
```

Please also note the following lines in the output (or bl-results2.0.txt); we assume that this line outputs information regarding the test cases.

```
Test case:
Next round of solving Linprog
```

The files Blast generates, apart from bl-results2.0.txt, are the following:

- demo2.tst: A binary file we have not yet interpreted.
- test.cfa: A file that contains the graph representation/code with some (css-like) styles.
- test.cfa-tests: A file which seems to contain the possible paths in the graph.
- test.cfa.txt: An empty file.
- demo2.abs: A binary file.
- There is also the file containing the traces.

[§]For information on how to switch, refer to section 2.

4 Working with a set of instances

In order to automate the whole run-and-report process, we have written some scripts. In particular, there is a bash script named generateTests.sh that automates the building and running process, while the python script, report.py, aids in the comparison and reporting of the results.

4.1 generateTests.sh

Though the script may seem complicated, it actually does three things. At the beginning it has a function to handle the PATH variable accordingly, while adding and removing tools. The second part adds the ability to handle multiple input and flags (in order to add and remove tools from the command line). The third part is where the actual computation is done: It runs KLEE against a program (or set of programs), identifies path conditions, converts them to assertions and runs BLAST against this set of assertions, to conclude if these are reachable or not (and notify us via a notify-send, if such a system exists, as the computation can take several minutes if not hours!)

In order to do this, after the script writes the assertion set to a file, it reads line by line and augments the original file (provided it finds a #BLAST# tag in the original file, where the user wants to add the assertions).

It is important to distinguish between the dependent (original) and the independent (modified) file. The underlying idea has to do with treating the dependent variables as independent in order to run KLEE fast (this is the input file to the script) and then run BLAST on the original file that treats the variables dependently again. *This means that both files need to exist in the workspace (and the dependent file name must be ⟨name⟩_dependent.c)*

For instance, in order to run the script for the controller program in the repository, you just issue:

```
./generateTests.sh controller.c
```

provided there is a controller_independent.c file in the same directory (it can also be a copy of the same file).

If the "generateTests.sh" script is sourced with -a or -d flags then it only alters the path (without generating the tests) even for when the script exists. This is particularly useful if someone wants to generate selected tests without the use of the script. For instance:

```
source ./generateTests.sh -a
```

Please take care in changing the hard-coded paths to your tools installation (i.e., KLEE, BLAST).

5 Comparing Results

The process to extract and compare results is carried out mainly by report.py. Report is a script responsible for the following:

1. given a set of source files with hardcoded assertions, it extracts these assertions and compares them against the assertions dynamically generated from the generateTests.sh script; and

2. given the output from BLAST on which paths are reachable or not, it generates simple statistics on this set.

Before running the script, one would need to update lines 152,153 for (1) above and line 157 for (2). Line 152 is where the folder with the programs containing the hardcoded assertions exists; line 153 is the file containing the assertions from KLEE (see Section 3); and line 157 is the file containing the results from BLAST (see Section 3) .

To run the report script for extraction:

```
./report.py -m extract # (or --mode)
common: \ldots
Only in 1st set: \ldots
Only in 2nd set: \ldots
```

and for comparison:

```
./report.py -m compare  # (or --mode)
reachable paths:       35   46 %
unreachable paths:     29   38 %
unknown reachability:  12   15 %
total paths:           76
```

6 Future work

The main issue we did not manage to identify was how to retrieve the test-vectors from BLAST 2.0. As documented in several papers [BCH$^+$04] [BHJM07], it seems possible to get a set of inputs that leads program to a specific state, if such a state is indeed reachable. However, after *extensive* research we have not yet managed to identify how to extract such input vectors. We hope that, if such vectors are computable with the use of BLAST 2.0, it should not be very hard to extract them.

In the case that such vectors are not computed by BLAST 2.0, another solution could be to extract the test vectors from trace formulas we get from BLAST 2.0, with the use of an external LP Solver.

References

[B$^+$08] D. Beyer et al., *BLAST - Infrastructure for C Program Verification*, `http://www.sosy-lab.org/~dbeyer/blast_doc/blast005.html`, 2008, [Online; accessed 15-Jan-2012].

[BCH$^+$04] D. Beyer, A.J. Chlipala, T.A. Henzinger, R. Jhala, and R. Majumdar, *Generating tests from counterexamples*, Proceedings. 26th International Conference on Software Engineering (2004), 326–335.

[BH$^+$08] D. Beyer, Thomas A. Henzinger, et al., *BLAST Examples*, `http://mtc.epfl.ch/software-tools/blast/download/examples/`, 2008, [Online; accessed 19-Jan-2012].

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar, *The software model checker Blast*, International Journal on Software Tools for Technology Transfer (2007), no. 5-6, 505–525.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler, *KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs*, USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008) (2008).