Final Project Report

Course: Computer Aided Verification

Project Title: Automatic Assertion Generation using Execution Traces

Name: Ameer Hamza

Problem Statement:

Running the program in order to explore the properties of the program (whether we want to find bugs or error states in the program or to find the invariants of the program) is a tedious and time-consuming task. Trying to find invariants for the program can be a time consuming task if we use dynamic analysis tools for that. This project attempts to find likely invariants of a program by looking at a few traces obtained from a dynamic analysis tool. A few traces do not need a long time to obtain. Hence, doing so would reduce the overhead of dynamic analysis tools to find invariants.

Tools used:

Daikon [1] is an implementation of dynamic detection of likely invariants of a program. This tool implements the same functionality as is required for this project. However, I am only interested in some intermediate information that I need for my projects. Daikon tool contains Kvasir. It is the Daikon's front end for C/C++ which executes C/C++ programs and creates data trace (.dtrace) files of variables and their values by examining the operation of the binary at runtime. It is built upon the Fjalar dynamic analysis framework for C and C++ programs. The ".dtrace" file contains information of the variables and their values at various points during the runtime. I believed that the tool can help me find the traces that I need to run my likely invariant detection algorithm on. I did not study any other techniques since I planned to implement my own algorithm.

Implementation:

I planned to use my implementation for C programs. The implementation itself is written in python language. Given different execution traces of a program, the algorithm analyses values of variables at various points during the execution. Using the acquired information, the algorithm tries to guess likely invariants for the program by looking at

variable values and relationships within the variable values and among variables. It then reports a set of likely invariants for the program. Provided is the pseudocode/outline for the algorithm:

1. Get traces of program
2. Parse the traces to get the values of the variables at each point in execution
3. Find invariants
4. Output found information to respective files

Step 1 is done using Daikon's front end for C/C++, kvasir. There is a limitation in this tool that it only provides the values of the variables at function start and end but nowhere in between. It does not even register any changes made to the variables within the function body (as far as my knowledge of this tool is). This was a problem for me since I needed values at every point in the execution. However, I decided to stick with this tool and make it work somehow. To make it work, I created dummy functions in my C code that did nothing and passed the variables to them. I called these dummy functions wherever I needed to check the values of the variables. The effect it had was that it reported the values of variables at the start of those dummy functions and thus I was able to get the required information.

Step 2 is trivial since I only have to read the trace files reported by Kvasir and store the information in my program to run the analysis on. However, ".dtrace" trace files have a specific format and looking at it does not readily give the variable values. As I run my algorithm, I report the traces in a readable format in "trace.txt" file.

Step 3 is finding invariants using the trace information. I use predefined templates to check for the invariants. I look for the following invariants:

- Single variables:
    1. If variable values remain the same throughout different runs of the programs, such invariant is reported. For example, if the value of variable x always remains 0, the invariant reported is 'x == 0'.
    2. If there are a few values any variable acquires, such invariant is reported. "Few" is defined as 5 i.e. if variable has less than equal to 5 distinct values. For example, if x has values {1, 2, 3}, it is reported as 'x == 1 || x == 2 || x == 3'.
    3. The range of the values of variables is reported. For example, if the values of variable x remain greater than equal to 0 and less than equal to 100, it is reported as 'x >= 0' and 'x <= 100'.
    4. If the values of a variable are either non-positive or non-negative, the invariant is reported. For example, if the range of values of variable x is -100 <= x <= 0, the invariant is reported as 'x <= 0'.

- Multiple variables:
    1. Multiple relations between every pair of variables are checked. These relations include greater than, less than, greater than equal to, less than equal to and equal. For example, if the values of variable x are greater than the values of variable y at each point in the execution of the program, the invariant is reported as ‘x > y’. The same idea extends to remaining relations, which are reported as ‘x < y’, ‘x >= y’, ‘x <= y’ and ‘x == y’ respectively.

Step 4 then outputs the found invariants in the form of assertion statement in the file “assertions.txt” in the respective benchmark folder. The invariants are shown on standard output.

Benchmarks:

I chose to select some of the benchmarks available at a public github repository, named Collection of Verification Tasks [2]. I chose the C programs available in that repository. The chosen benchmarks are simple, contain variable declarations, simple manipulations including additions, subtractions, etc and simple loops. The benchmarks do not contain any functions (except for dummy variables that I added) since my algorithm does not find invariants for programs involving the functions. I made small modifications to the benchmarks, such as changing non-determinate values to rand methods calls in C’s stdlib.h library. As stated, I chose these benchmarks for the simplicity of the task they implement and since these programs give me enough invariants to analyse the programs.

Limitations:

I have a few limitations in the implementation, each of which are discussed below:

1. **Dummy functions:** I have already explained the reason for using dummy functions. This is the limitation of the tools, Daikon, that I used. This puts a limit to the degree of automation in this tool since all other parts of the algorithm are computed automatically. I had an option of choosing any tool other than Daikon, however I could not find a good tool that would have given me proper execution traces of the program. I decided to go with Daikon since it was taking me a long time to look for a new option. I had LLVM in mind and I thought of using LLVM to instrument the program to obtain the variable values during execution but installation and learning overhead of LLVM would have been immense, since I

have little familiarity with LLVM. I had already spent a lot of time on the AFL (American fuzzy lop) tool before choosing Daikon. Hence, I chose to stick with Daikon and using dummy functions.

2. **Float values:** My implementation works with floating point values. Even if the program has integers, the implementation considers them floats. This limitation is justified since the floats are more generic than integers. I can remove this limitation, if required, since I only have to parse another data file, called declaration file (extension: decls), provided by Daikon. I decided not to spend time to remove this limitation and decided to focus more on actual implementation.

File naming conventions and program execution information:

I have placed a bash script named “run.sh”. This script compiles the C code using gcc, with the flags required by Daikon to obtain the traces. Also, it runs Kvasir to obtain the traces, which are stored in the .dtrace files. The bash script requires the benchmark name (corresponds with the directory names) and the number of traces you want to run the program for. It automatically creates the specified number of trace files in the respective benchmark directories. Since anyone wanting to run the script might not have Daikon and Kvasir implementation available, I have already run it for all the benchmarks for 20 trace runs each benchmark. These traces reside in “daikon-output” directory in the respective benchmark directory. You can only run the python implementation to get the invariants from those traces.

As stated earlier, the implementation is done using python language because of my familiarity with the language. Specifically, I use python3. The implementation is included in the file “InvariantDetection.py”. This file requires the name of the benchmark and number of traces you want to run the algorithm for. This will only run for at most 20 traces that I have already included in the project. As an example, if you want to run the benchmark “sum” for 10 traces, specify following command on the commandline:

```
python3 sum 10
```

Running above command would show the invariants found in the benchmark to the standard output. It will also create a file “assertions.txt” in the benchmark directory which contains assertions to be added to the benchmark. Another file created is “traces.txt”, which shows a simplified version of the all traces since .dtrace file is not easy to read. Some benchmarks might take a bit long to run but the algorithm works and it will give the invariants (and assertions) for the program. In the end, there is another C file in each benchmark directory, containing “Orig” keyword. These are the original C programs that I used without the dummy functions added.

References:

1. https://plse.cs.washington.edu/daikon/
2. https://github.com/sosy-lab/sv-benchmarks