Assignment 7 (Theory and Tool)

Model Checking with Blast

17-654/17-754: Analysis of Software Artifacts
Jonathan Aldrich (jonathan.aldrich@cs.cmu.edu)

Due: Monday, April 10, 2006 (5:00 pm)

100 points total

You may do this assignment in pairs. For each pair, turn in ONE file named <username1>-<username2>-17654-A7.zip, where the usernames are your Andrew ids. The zip file should contain answers.{txt,doc,pdf} with your answers to free-form questions. Diagrams can be hand-drawn and turned in at class, or drawn electronically or scanned and included in the answers file above or in a separate file. Also include the .spc file for question 2.3, cmds.c for 2.2, the Blast-ready source files to the application for question 3, and one output file for each of 2.1, 2.3, and 3.1. In a comment at the top of answers, state both of your names, both Andrew ids, and how long you each spent on the assignment.

Assignment Objectives:

- Understand how CEGAR eliminates infeasible paths
- Specify properties and find bugs in real C code using the Blast model checker.

1. Simulating CEGAR (20 points)

Consider the following code fragment:

```
1  lock()
2  gotLock = 1;
3  while (*) {
4       if (*) {
5            if (gotLock == 1) {
6                 unlock();
7                 gotLock = 0;
             }
        }
8       if (gotLock != 1) {
9            lock();
10           gotLock = 1;
        }
     }
11  unlock();
12
```

Consider an initial analysis of this program, keeping track of only two predicates: lock==0 and lock==1. Assume lock==0 initially. Whenever a lock statement is executed, a variable lock is set to 1. Whenever an unlock statement is executed, lock is set to 0.

Consider using BLAST to find double lock or double unlock errors in this code. I.e., Blast will find an error if lock() is called when lock==1 or unlock() is called when lock==0.

Question 1.1.
Draw the initial control flow automaton (CFA) that Blast will build, up to the point where Blast finds the *first* error (in which case other parts of the CFA may still be incomplete), or else explores the entire search space and finds no error. In your CFA:

- Number the nodes to match the number of the line that's about to execute.
- Label each node with "L" or "U" to represent the "lock==1" or "lock==0" predicates, respectively (the initial node should be labeled "U" for unlocked/lock==0).
- Label each non-branching edge with the statement executed.
- For branches, label P(*pred*) where *pred* is the condition that must be true for that branch to be taken (thus if there are two edges one will be *pred* and one *!pred*). For example, P(gotLock==1) will go on one edge in the CFA. You need not use any label if the predicate is *.

Note that answers may vary slightly (e.g. the shape of the graph or, in the case of an error, which error path shows up first), depending on how you explore the CFA, but your answer should be consistent with the Blast algorithm given in the paper on the 654 web site.

Question 1.2.
If an error state is reachable in the CFA given above, does it represent a real bug? If so, prove it is a real bug, by showing using Hoare Logic that the precondition of the path is not false.

If the bug is not real, show using Hoare Logic that the precondition of the path is false.

You may use the technique shown in the class on Blast, renaming a variable each time it is reassigned, or you may use the Hoare Logic system described earlier in class.

Question 1.3.
If the bug was not real, come up with an interpolant that explains why the path is unreachable. Specifically, give a line in the program and a predicate, such that (1) the predicate is true in the (infeasible) path right before that program line, (2) the predicate describes the current state, but (3) the conjunction of that predicate with the weakest precondition of the rest of the path (after that line) is unsatisfiable (false).

Question 1.4.
Now, build the CFA again, this time considering the predicate you found in Question 1.3. Although Blast does not in fact apply this predicate to the whole program, please apply it to the whole program for simplicity in this assignment (otherwise you may find you need to add it again later).

2 Running BLAST on GAIM (30 points)

In this section, you will learn how to use Blast by finding a bug and defining a protocol. First, download Blast.zip from the course web page and extract the contents to a directory path that does not contain any spaces. Verify that an "includes" directory was created that contains a number of .h files. Next, go to www.cygwin.org and install cygwin, making sure that gcc is selected under the development section. Finally, add "$cygwin path$\cygwin\bin" (e.g. "c:\cygwin\bin") to your system path, and either ensure the Blast directory is in your path or "." is in your path (and work from the Blast directory). If your installation was successful, entering "gcc" on the command prompt should result in "gcc: no input files" and not command not found.

Next, read through the BLAST tutorial at http://embedded.eecs.berkeley.edu/blast/doc/blast.pdf. You can skip section 2 on BLAST installation.

Question 2.1 (10 points)

Run the following two commands from the Cygwin shell (the DOS shell may not work) to build a .i file and run BLAST on the result:

gcc –E –I include cmds.c > cmds.i
pblast.opt cmds.i –main gaim_cmd_list

Turn in Blast's output as output21.txt.

Question 2.2 (10 points)

Describe the bug that causes the assertion to fail. Fix the bug and turn in your modified copy of cmds.c.

Question 2.3 (10 points)

Write a Blast .spc spec file to check the following protocol for ExampleProtocol.c:

1. DoA(); // at least 1 call
2. // at least 1 call of 2a or 2b

2a. DoB();
2b. DoC();

3. DoD(); // 0 or more calls
4. DoE(); // 1 call iff DoB() was called

Run Blast using ExampleProtocol.c and your .spc file on the main methods (1) GoodDriver, (2) BadDriver1, and (3) BadDriver2. Turn in Blast's output for each of these methods (output23a.txt – output23c.txt) and your .spc file.

3 Running BLAST on Source Code (50 points)

Run BLAST on a piece of source code. The code may not have been written specifically for this assignment, but you may use code from previous projects or from an open source project. Two good sites for finding open source applications are www.apache.org and www.sourceforge.net. Note that BLAST will only run on C or C++ code. Either (a) identify or insert at least ten assert statements (make sure you include Blast's assert.h) or (b) write a .spc file describing a protocol.

Some hints:

- You probably don't want to run BLAST on the whole application (but you may do so if you wish). Instead, you probably want to find a single .c file or small number of .c files. BLAST will begin running at a function you specify. For function calls for which you do not provide source (Note: you will still need the .h file), BLAST will replace it with a no-op if its return value is not used, or if there is a return value BLAST will replace it with an effect on the variable assigned to the function call's result.
- Running BLAST on a .c file will require a directory containing all of the .h files referenced in the .c file and all of the .h files referenced by those .h files. Pick an application or .c file where these are easy to find. In particular, be careful of applications that generate .h files during the build process, as you will either have to build the application or have lots of work to remove missing dependencies. If you find a project that you are having difficulty successfully compiling, give up and try a different project. Do not waste lots of time trying to get gcc to run successfully.

Question 3.1 (15 points)
Turn in a zip file containing

- All of the files you ran blast on (.c & .h files)
- The URL from which the code can be obtained and any additional instructions for downloading the code included in the answers.txt file
- A file output31.txt that contains the commands you used to run gcc and blast and the resulting output generated by BLAST

Question 3.2 (15 points)
If you chose option (a) list the assert statements you identified or added, with line number and file. If you chose option (b) list the contents of the .spc file you wrote.

Question 3.3 (5 points)
In a paragraph or less, describe the functionality provided by the code you ran BLAST on.

Question 3.4 (5 points)
Describe what conditions you checked by the asserts you added or the .spc file you wrote. Why should these conditions be true?

Question 3.5 (10 points)
Critique BLAST, and describe your experience using BLAST. Would you ever use BLAST on a project outside of this course? What are its strengths and weaknesses compared to (1) whitebox testing and (2) ESC/Java?

Question 3.6 (EXTRA CREDIT). (20 points)
If you think you found a real bug, you may receive 20 points of extra credit. Describe the bug, and justify why the program is wrong, not your assertions or specification.

If you submit the bug report to a developer and this submission results in the developer (who may not be you or anyone you know personally) making a change to fix the bug, you may receive additional extra credit (similar to in the previous assignment).