A Basic Test Automation Set-up

Sujit Kumar Chakrabarti

July 18, 2024

Chapter 1

Introduction

1.1 Program under test (PUT) - Jugs

The program under test is a solution to a simple programming puzzle: Given two jugs of capacity c1 and c2, and a final volume f, the program should find the shortest sequence of steps to be followed to achieve a volume of f in either of the jugs the other being empty.

Chapter 2

Test Execution Automation

2.1 Instructions for Use

To compile the Jugs program, run:

```
make jugs
```

To run the Jugs program, run:

```
./jugs c1 c2 f
```

For example, running:

```
./jugs 9 15 3
```

will yield:

```

(0, 0)
(9, 0)
(0, 9)
(9, 9)
(3, 15)
(3, 0)
```

each line showing the volume of water in each jug in each subsequent step.

2.2 Testing Jugs

We have implemented a basic test automation system as a warm up exercise.

To compile the test program, run:

```
make test_jugs
```

To run the test, run:

```
./test_jugs
```

The output of executing the test will appears somewhat as follows:

```
Coverage:
Testcase1 : , 0, 2, 4, 5
Testcase2 : , 0, 1, 2, 4, 5, 2, 4, 5
Testcase3 : , 0, 1, 19, 20, 22, 23, 19, 21, 24, 2, 4, 5, 2, 4, 5
Testcase4 : , 0, 30, 31, 32, 31, 32, 33, 36, 37, 2, 4, 5
covered locations : , 0, 1, 2, 4, 5, 19, 20, 21, 22, 23, 24, 30, 31, 32, 33, 36, 37
```

This shows that four test cases were run: Testcase1, Testcase2, Testcase3 and Testcase4. Against each test case, the program locations that were covered by that test case are list. The last line of the output shows the list of all locations that were covered by all the test cases.

2.3 Cleanup

Run:

```
make clean
```

If required, to clean the directory of binaries to begin the building process afresh.

Chapter 3

Test Harness Design

3.1 Architecture

3.2 Instrumentation

The program under test jugs.cpp has been instrumented for labelling the program locations to be covered with numbers and inserting logging instruction at these locations. The instrumented version of jugs.cpp is `jugs_instrumented.cpp`. Testcases and Test harness: The key part of the test automation is implemented in the file tester.cpp and tester.h. This module implements two classes: TestHarness provides the capability of running test cases in sequence, recording and outputing the coverage report. Testcase is the test case class.

3.3 Test case Design

As this is the first cut attempt in getting some hands on experience with unit testing, we have written the test cases automatically. But, we have followed a somewhat systematic approach in this. We have drawn a function call graph (doc/dep.dot). This gives us an idea about the functions which are independent and easier to test and about those which make calls to other functions. We have chosen to first write test cases for testing the functions which are lower down in the dependency graph (i.e. they are independent).

3.4 Test Cases for Statement Coverage

Significant statement coverage can be obtained by designing approximately one test case per PUT method. To go further, we need to consider the control flow graphs (CFGs) of the methods to be tested. CFGs are a good starting point to achieve more extensive statement coverage.

3.4.1 Control Flow Graphs

Consider `Process::min` method shown in Fig.3.4.

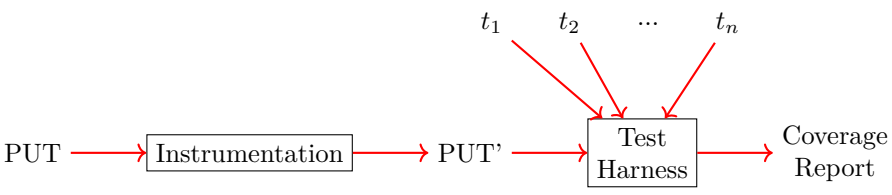

Figure 3.1: Dataflow architecture

3.5 Conclusion

3.5.1 What has been achieved?

Automated execution of test cases and recording coverage.

3.5.2 What next?

- Automatic test generation
- Automated instrumentation
- Automated function call graph generation

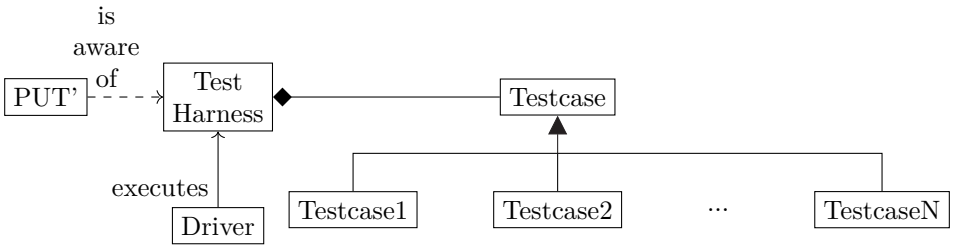

Figure 3.2: Class diagram

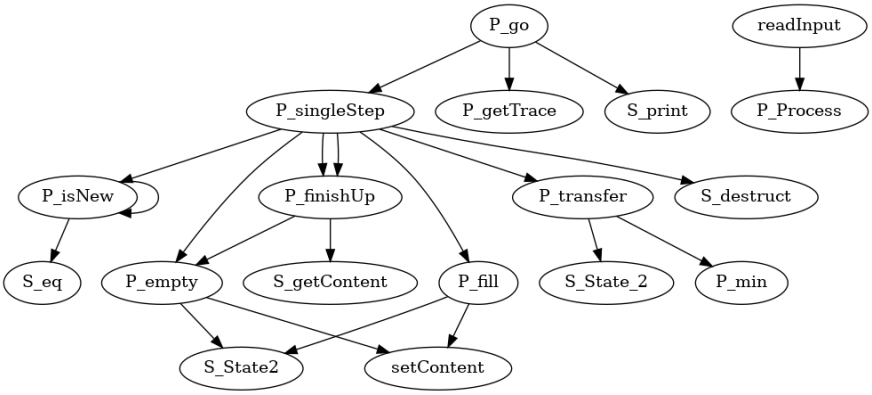

Figure 3.3: Function call graph

```
unsigned int Process::min(unsigned int a,
    unsigned int b) {
  // 50
  g_testharness.log(50);
  if(a < b) {
    // 51
    g_testharness.log(51);
    return a;
  }
  // 52
  g_testharness.log(52);
  return b;
}
```

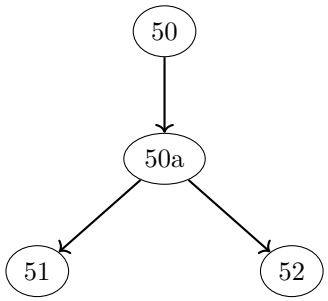

Figure 3.4: Control flow graph: `Process::min`

Figure 3.5: Statement coverage: `Process::min`

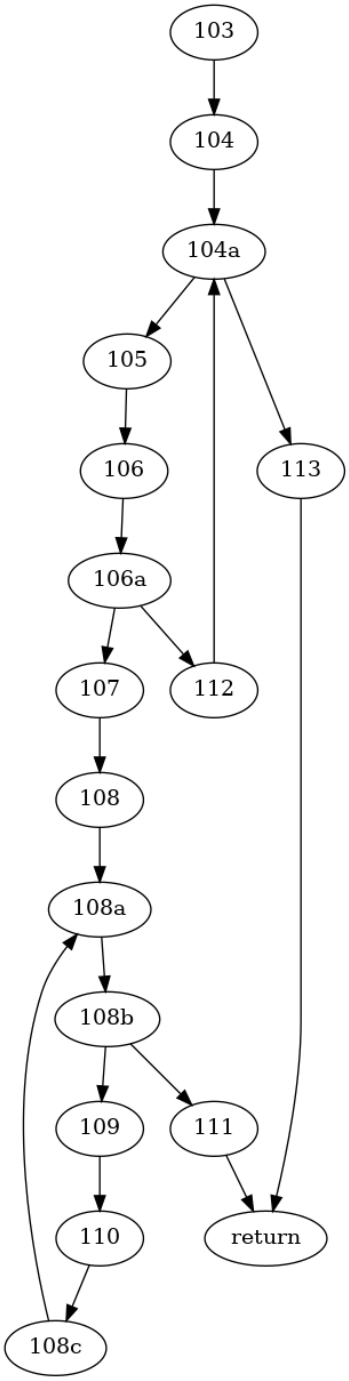

Figure 3.6: Control flow graph: `Process::go`

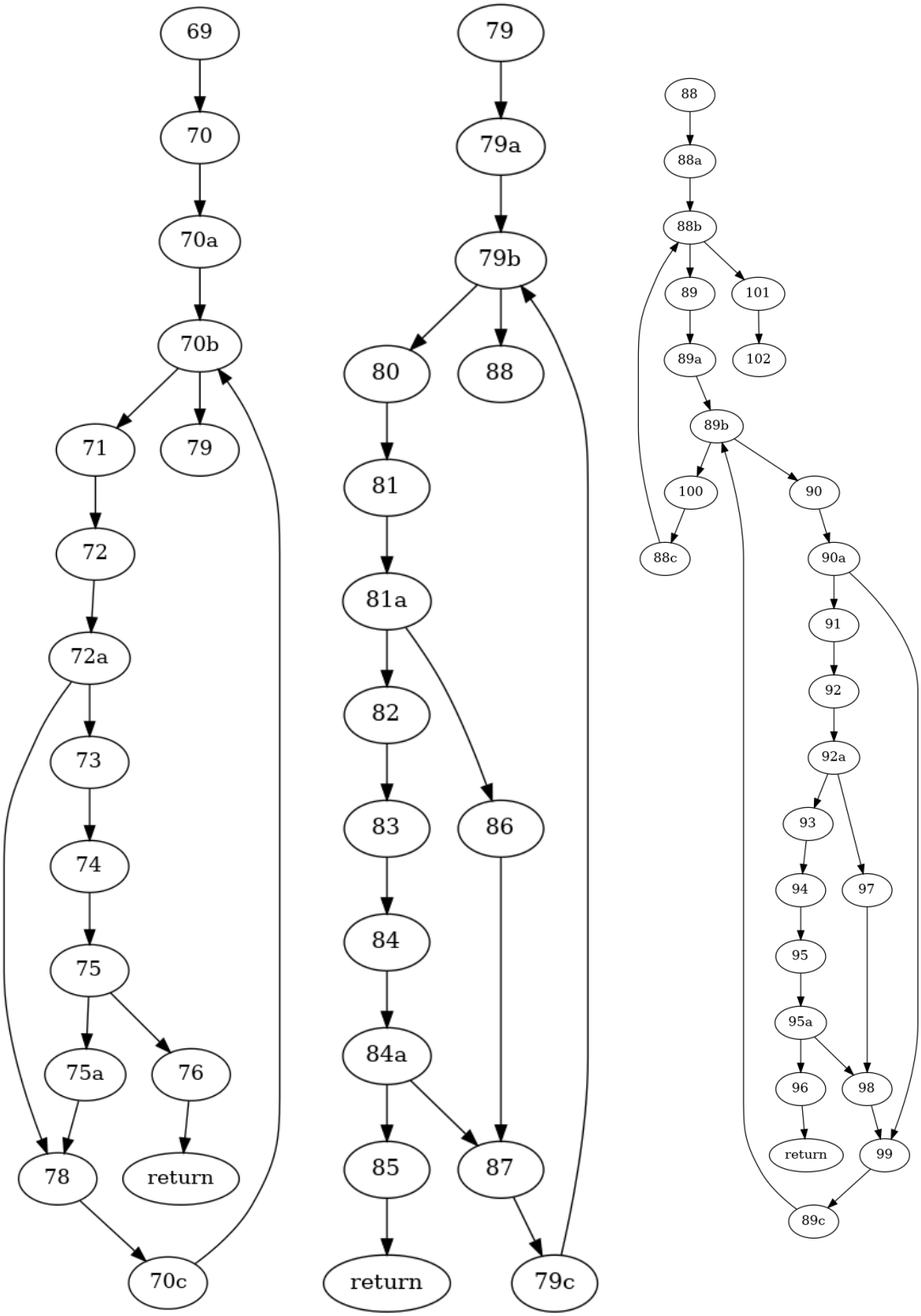

Figure 3.7: Control flow graph: `Process::singleStep`

Chapter 4

Automated Test Generation

4.1 Instructions for Use

To compile the Jugs program, run:

```
make testgen
```

To run the Jugs program, run:

```
./testgen
```

For example, at the time of writing this section, running the above command will yield the following output:

```
PTC_min1 : , 0, 2, 4, 5, 30, 31, 32, 33, 36, 37, 50, 51
PTC_min1 : , 0, 2, 4, 5, 30, 31, 32, 33, 36, 37, 50, 51
PTC_min1 : , 0, 2, 4, 5, 30, 31, 32, 33, 36, 37, 50, 51
PTC_min1 : , 0, 2, 4, 5, 30, 31, 32, 33, 36, 37, 50, 51
PTC_min1 : , 0, 2, 4, 5, 30, 31, 32, 33, 36, 37, 50, 52
test input = , 35005211, 521595368, 294702567, 1726956429, 336465782
test input = , 424238335, 719885386, 1649760492, 596516649, 1189641421
```

In the above, we are trying to test the function `Process::min`. This function has three statements to be covered which are 50, 51 and 52 (see source file `testgen_main.cpp` to see how this is specified). The first 5 lines show that 5 sets of test inputs were generated before the required coverage was attained. However, only two of these are included in the final output. This is because the test generation algorithm tries to select only those test cases which give us efficient incremental coverage, and throws away those which give no incremental coverage (see Section 4.2.2 for details).

4.2 Random Generation

Our first attempt at automated test generation is through random generation. The idea is simple, we keep generating test inputs randomly until the desired coverage is achieved. The various components of this problem are as follows:

1. **Test input generation:** We create a simple class for this called `TestGenerator` that currently generates simple unsigned int values randomly.
2. **Test setup:** A method/function may require a setup to be tested. This setup has to be done systematically. We assign this responsibility to a abstract class called `ParameterisedTestCase`. The primary objective of this class is to abstract away the actual way the input values are fed to the method under test from the `TestGenerator` class.
3. **Coverage data collation:** To keep track of the coverage attained at any point: this problem has already been taken care of in the test harness. We will tweak it a bit and reuse it.

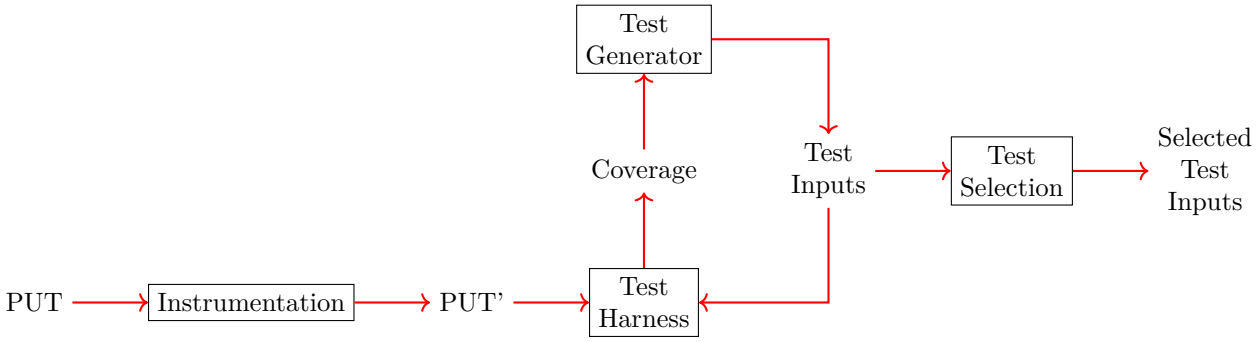

Figure 4.1: Architecture: Random Test Generation

4. **Test case selection:** The test generator finally tries to come up with an optimised set of test inputs.

4.2.1 Architecture

4.2.2 Test Case Selection

The total number of tests run before the required coverage is attained may be vastly larger than the actual number of test cases needed to achieve that coverage. Unfortunately, the problem of computing this optimal set of tests is NP-complete. For more details, see set cover problem [Kar72]. Here, we use a simple heuristic to compute this, shown in Algorithm 1. The algorithm goes through all the test inputs generated and their corresponding coverage. It then iteratively the test input that gives the best incremental coverage of target statements w.r.t. the test inputs selected so far. Please note that this heuristic is a greedy algorithm that does not promise to compute the optimal set. But, this may suffice in many practical cases since having the optimal test input set is neither essential nor feasible in most cases.

Algorithm 1 A basic test selection algorithm

```
procedure SELECTTESTCASE(T, targets)
    T' ← SORT(T)                          ▷ Sort in descending order of incremental coverage.
    T'' ← {}
    for all i ∈ {0, ..., |T'| − 1} do
        if ct < targets then              ▷ ct – targets covered
            add T'_i to T''
    return T''
```

Bibliography

[Kar72] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.