Effective Unit Test Design and Automated Debugging

PVR Murthy
Corporate Technology
Siemens
Bangalore, India

***Abstract*— There exist several unit test tools such as JUnit, however, besides the suggestion that assertions can be specified to reflect properties that must be satisfied at different program points, there is no guidance to testers and developers about how to design tests effectively that can be used with such tools. This paper provides a systematic basis for testing a component or a function by defining each test to be a partitioned <pre-condition, postcondition> pair. Automated debugging is also possible by computing the actual program states in the forward direction and the hypothesized program states in the backward direction and identifying the statement where an error is present.**

Keywords- Assertions, trace, coverage, debugging

I. Introduction

There are many unit test tools available such as JUnit for Java and NUnit for C# [1,2]. The essential methodology here is to write and place assertions at desired points in the code of a function. During the run of a test, it can be verified whether the state of computation satisfies the specified assertions. However, there has not been much discussion on systematic methods to design tests for use with NUnit or JUnit to test components or methods.
The input domain of a function can be expressed as a union of predicates corresponding to different partitions , each predicate representing a set of values that the input parameters may assume in a partition. Whether the partitions are formed based on equivalence class partitioning, or, boundary value analysis , or some other method, each partitioning of the input domain represents a class of tests. Given a corresponding expected result condition, or test oracle, for each partitioning predicate, the specification of each test is available as a <pre-condition, postcondition> pair, where the pre-condition corresponds to the partitioning predicate and the postcondition corresponds to the expected result condition.
In this paper, each test designed is viewed as a <pre-condition, postcondition> pair. Coverage with respect to the input domain is said to be achieved, if the union of the pre-conditions is the entire input domain of the function under test. This paper illustrates how automated debugging can be carried out, if the trace of statements executed, for each test run, is emitted.

II. Paritioning and Automated Debugging

Some sample test cases where a,b,c > 0 are

```
1)precondition:   (a eq b) and (b eq c) and  (c eq a)
                  type = triangleType(a,b,c)
  postcondition:  ( type eq equilateral)

2 )precondition:  (a eq b) and (a ne c) and
                                       (a+b gt c)
                   type = triangleType(a,b,c)
   postcondition: ( type eq isosceles)
```

Consider the function max(a,b) defined as *if (a>b) then max = b else max = a*. Below are tests designed based on the partitioning of the input domain of the function max.
Test 1:Pre-condition: (a>b). Postcondition: (max equals a). *Test 2*:Pre-condition: (a<=b); Postcondition: (max equals b).
Consider Test 1 and compute actual program states in the forward direction.

```
     Pre-condition: (a>b)
          If (a>b) then
        State: (a>b)
             max = b
   State: (a>b) and (max == b)
```

Now computing hypothesized states in the backward direction, the postcondition (max == a) contradicts with the corresponding actual state (a>b) and (max == b). The hypothesized state, just preceding the statement max = b, is (a==b) which contradicts with the actual state (a>b). From this it can be concluded that the statement max = b, in the then branch of the if (a>b) statement, is erroneous or has a bug. Our method described above towards automated debugging is similar to the method reported in [3], however, there is a significant difference in that in [3] , the postcondition is a union of expected result conditions for different paths or scenarios. In [3], the postcondition is the expected result condition of the entire method or function. The trace of execution for a test corresponds to a unique single path and it has to be determined which disjunct in the postcondition in [3] actually corresponds to the trace and this requires trying out disjunct by disjunct until the relevant disjunct in the postcondition is picked up. These overheads are not present in our method as both the pre-condition and postcondition are corresponding partitions (of a test) that result in a unique trace of statements upon execution.

III. ALGORITHM

Automated Debugger Algorithm:

Step 1: Design tests for a function corresponding to each behavioural slice or a partitioned pre-condition, postcondition pair.

Step 2: Run each test designed in step 1 and store, for each failed test, the test execution trace in terms of the corresponding sequence of statements or branches executed.

Step 3: For a failed test, starting with the corresponding partitioned pre-condition, compute actual program state at each program point or statement in the corresponding trace in the forward direction.

Step 4: For a failed test, starting with the corresponding partitioned postcondition, compute backwards hypothesized state at each statement in the corresponding execution trace. If the actual program state does not imply the hypothesized program state at a program point, the location of a likely erroneous statement is detected as in [3].

For a trace S_1, S_2, …,S_n generated by the execution of the test <pre-condition, postcondition>, automated debugging may be carried out as shown below, if the test fails.

```
<pre-condition>
  S1
[actual program state,
 hypothesized program state]

  S2
[actual program state,
 hypothesized program state]

                    …

  Si              Evidence as
                  actual program state
                  and hypothesized state
                  contradict each other;
                  likely location of the
                  error at Sj, where j<=i.

                    …

  S(n-1)
[actual program state,
 hypothesized program state]

  Sn
<postcondition>
```

Concrete execution:

An instance of each <pre-condition, postcondition> pair may be created by assigning suitable values to variables or parameters and the function invoked with actual parameter values and the trace of statements executed is recorded. Actual program states and hypothesized program states are computed and remembered at each program point. The statement where a contradiction is encountered leads to an evidence or a clue about the possible location of the errorneous statement.

Symbolic execution:

Actual states can be computed in the forward direction based on symbolic execution as well. Hypothesized states can be computed backward starting from the postcondition. The statement where a contradiction is found between the actual state and hypothesized state leads to evidence. If symbolic execution is employed , the path in the function that corresponds to a test needs to be identified.

IV. CONCLUSIONS

This paper described a systematic basis for unit testing wherein each test case is represented by a <pre-condition, postcondition > pair. Each pre-condition is a partition of the input domain of the function or component under test. The union of all the pre-conditions (of the tests) must be the input domain of the function under test , if test coverage needs to be achieved. The paper also described a technique which is the basis for automated debugging for failed tests. The examples discussed in the paper deal with numeric variables , however, the methodology is equally applicable for variables of any data type as the key is to view each test as a <pre-condition, postcondition> pair.

REFERENCES

[1] http://www.nunit.org

[2] http://www.junit.org

[3] Haifeng He, Neelam Gupta,: "Automated Debugging Using Path-Based Weakest Preconditions," FASE 2004: pp 267-280.