Software Engineering 4 (F22HX3)

Dynamic Analysis Techniques Part 1

Andrew Ireland

School of Mathematical & Computer Sciences
Heriot-Watt University
Edinburgh

Outline: Parts 1 & 2

- A strategy for dynamic testing
- Test case design techniques
- Built-in test

Dynamic Analysis

- Dynamic testing and analysis involves system operation, *i.e.* code execution.
- Dynamic testing that does not exploit the internal structure of the code is known as **functional**, black-box, behavioural or requirements based.
- Dynamic testing that does exploit the internal structure of the code is known as **structural**, white-box, glass-box or coverage based.
- Dynamic analysis will typically involve the construction of additional code to facilitate the testing process.

A Classification of Test Case Design Techniques

Functional	Structural	Hybrid
Equivalence partitioning Boundary value analysis State based testing	Statement testing Decision testing Condition testing Decision/condition testing Data definition-use testing ...	Error guessing Built-in test

The aim of design techniques is to provide a systematic basis for developing **effective** test cases, *i.e.* test cases that will provide a high yield in terms of defect detection.

A Strategy For Dynamic Analysis

1. Use functional testing and error guessing techniques to design initial test cases.
2. Use coverage metrics to determine the effectiveness of the initial test cases.
3. Use structural testing techniques to increase coverage if judged necessary.

Both positive (desirable behaviour) and negative (undesirable behaviour) test cases should be applied.

Obtaining Structural/Functional Balance

- While internal knowledge can greatly simplify the task of dynamic testing — over reliance on the actual code should be avoided, *i.e.* risk of designing test cases that demonstrate that the code does what the code does!
- The primary source of test cases should be the functional specification. The process of designing tests based upon functional specification will typically expose many defects before coding gets underway.

Equivalence Partitioning

- A technique that partitions the space of possible program inputs/outputs into a finite set of **equivalence classes**.
- An equivalence class defines a set of data values for which our program will perform the same computation.
  - If one test case in an equivalence class identifies a defect then all the other test cases in the equivalence class would also identify the same defect;
  - Conversely, if a test case did not identify a defect we would not expect any other test case in the equivalence class to identify defects.
- Equivalence partitioning can significantly prune the space of possible test cases — assuming the equivalence partitioning is performed correctly!

Guidelines for Partitioning

Ranges: if a requirement specifies a range then three equivalence classes are required, one valid and two invalid. Note that there are always implementation dependent boundaries, *e.g.* 16-bit integer gives rise to a partition bounded by 32767 and -32768.

Numbers: if a requirement specifies a number then three equivalence classes are required, one valid and two invalid.

Sets: if a requirement specifies a set then two equivalence classes are required, one valid and one invalid.

Note: the ordering of input parameters may need to be taken into consideration. Note also that partitioning can and should be applied to both input and output conditions.

An Example of Input Partitioning

"... the first and last page numbers for a paper which appears within a journal or conference proceedings should be specified ... the total number of pages for the text must be in the range 1 *to* 1000 *..."*

Input Condition	Valid Equivalence Classes	Invalid Equivalence Classes
first to last	first $\leq$ last TC1	first $>$ last TC6
size of first	first $>$ 0 TC2	first $\leq$ 0 TC7
size of last	last $\leq$ page total TC3	last $>$ page total TC8
page total (pt)	1 $\leq$ pt $\leq$ 1000 TC4	$<$ 1 TC9, $>$ 1000 TC10
No. of page nos	2 TC5	$<$ 2 TC11, $>$ 2 TC12

Test Cases

Test Case	Test Data	Expected Result
TC1	first = 22 last = 37	Accept
TC2	first = 22	Accept
TC3	last = 37 total = 150	Accept
⋮	⋮	⋮
TC6	first = 37 last = 22	Reject
TC7	first = -21	Reject
TC8	last = 160 total = 150	Reject
⋮	⋮	⋮
TC12	5, 22, 37, 50	Reject

Boundary Value Analysis

- In general test cases that explore boundary values give a higher yield of defect detection.
- Boundary value analysis extends equivalence partitioning by focusing attention on equivalence class boundaries, *e.g.* consider test case construction for an equivalence class of the form $(X > 0)$ and $(X \leq 99)$:

	EP	EP+BV
Valid	42	1, 99
Invalid	-5, 105	0, 100

EP = Equivalence Partitioning; BV = Boundary Value analysis.

State Based Testing

- Applicable when software has been designed as a state machine.
- Also applicable where requirements are specified in terms of state machines, *e.g.* washing machine control software.
- State transitions provide strong guidance in designing both positive and negative test cases.
- Syntax directed testing is strongly linked to state transition testing.

Test Case Effectiveness

- When have we tested enough?
- Coverage metrics provide one of the most widely used approaches to judging the effectiveness of testing.
- A coverage metric (CM) represents the ratio of metric items executed at least once (EM) to the total number of metric items (TM):

  $$\mathrm{CM} = \mathrm{EM}/\mathrm{TM}$$
- See Beizer (1990) for detailed analysis of coverage metrics and there relative merits.
- Coverage metrics are strongly related to structural or coverage test case design techniques ...

Statement Coverage

- **Definition:** Every statement is executed at least once.
- **Also known as:** line coverage, segment coverage, C1 coverage and basic block coverage.
- **Advantage:** easy to understand and relatively simple to achieve 100% coverage.
- **Disadvantage:** Even with 100%, statement coverage is weak *e.g.*

```
int* ptr = NULL;
if (condition)
   ptr = &result;
*ptr = 666;
```

100% statement coverage only requires that the execution path including `condition` set to `true` is achieved. Note that the `false` branch gives rise to code failure.

Decision Coverage

- **Definition:** Every statement and every decision (`if`-statement, `while`-statement etc) is executed at least once.
- **Also known as:** branch coverage, all-edges coverage, basis path coverage, C2 coverage, decision-decision path testing.
- **Advantage:** easy to understand and relatively simple to achieve 100% coverage and overcomes the deficiency of statement coverage.
- **Disadvantage:** 100% coverage does not mean that the deeper logical structure of the decision point will be adequately explored, *e.g.*

```
if (condition1 || condition2)
   statement1;
else
   statement2;
```


Condition/Decision Coverage

- **Definition:** Every statement and every branch is executed at least once, with each condition within each branch taking on all possible values at least once.
- **Advantage:** Addresses the deficiency of decision coverage to some extent, *i.e.* explores the deeper structure of the decision point to some extent.
- **Disadvantage:** Can still leave untested combinations of conditions, *e.g.*

```
if (condition1 && (condition2 || condition3)) ...
```

`case`	`condition1`	`condition2`	`condition3`	`branch`
1	True	True	True	True
2	False	False	False	False

Multiple Condition Coverage

- **Definition:** Every statement is executed at least once, all combinations of values for each condition are explored.
- **Advantage:** Addresses the deficiency of condition/decision coverage to some extent, *i.e.* explores the deeper structure of the decision point to some extent.
- **Disadvantage:** Expensive to develop, *i.e.* 2^N test cases where N is the number of boolean operands, *e.g.* ...

Multiple Condition Coverage

```
if (condition1 && (condition2 || condition3)) ...
```

`case`	`condition1`	`condition2`	`condition3`	`branch`
1	True	True	True	True
2	True	True	False	True
3	True	False	True	True
4	True	False	False	False
5	False	True	True	False
6	False	True	False	False
7	False	False	True	False
8	False	False	False	False

Modified Condition/Decision Coverage

- **Definition:** Every condition within a decision is executed to shown that it can independently effect the outcome of the decision.
- **Advantage:** A compromise requiring fewer cases than multiple condition coverage, *i.e.* minimum of $N + 1$ and a maximum of $2N$ cases where N denotes the number of boolean operands involved.
- **Disadvantage:** No real gains when N is small.

Modified Condition/Decision Coverage

```
if (condition1 && (condition2 || condition3)) ...
```

`case`	`condition1`	`condition2`	`condition3`	`branch`
2	True	True	False	True
3	True	False	True	True
4	True	False	False	False
6	False	True	False	False

Test cases 2 and 6 show independence of `condition1`
Test cases 2 and 4 show independence of `condition2`
Test cases 3 and 4 show independence of `condition3`

autograde Revisited

```
typedef enum gradetype {A, B, C, F};

gradetype autograde(int student[], int model[], int num){
int score; gradetype grade;
int correct = 0; int index   = 0;
while (index < num){
      if (student[index] == model[index]) correct++;
      index++;}
     score = (correct * 100) / num;
     if (score >= 80) grade = A;
     else
        if (score >= 60) grade = B;
        else
           if (score >= 50) grade = C;
           else
              grade = F;
     return grade;}
```


Decision Coverage: Test Matrix

Decision	Student/Model Answers	Num	Test
WHILE-1	*	> 0	TC1
WHILE-1	*	≤ 0	TC1
IF-1	% correct > 0	> 0	TC1
IF-1	% wrong > 0	> 0	TC2
IF-2	% correct ≥ 80	> 0	TC1
IF-2	% correct < 80	> 0	TC3
IF-3	% correct ≥ 60 & < 80	> 0	TC3
IF-3	% correct < 60	> 0	TC4
IF-4	% correct ≥ 50 & < 60	> 0	TC4
IF-4	% correct < 50	> 0	TC5

Test Cases

Test	`student`	`model`	`num`
TC1	1 1 1	1 1 1	3
TC2	0 1 1	1 1 1	3
TC3	1 1 0	1 1 1	3
TC4	1 0	1 1	2
TC5	1 0 0	1 1 1	3

Note: a single TC will typically cover multiple paths through a computation. Warning: can lead to problems, *e.g.* what if `num` is 0?

Data Definition-Use Testing

- Strong similarities with data flow analysis (see later static analysis lecture).
- Test cases are designed to exercise definition-use paths (*du*-paths).
- Greater focus – consider paths that:
  - effect control - predicate uses (*p*-uses)
  - are purely computational - (*c*-uses)
- Path finding can be automated, *e.g.* dependence tracking techniques such as used in program slicing. Creating actual test data, *i.e.* path sensitization is more creative.

Summary

- A strategy for dynamic testing.
- Survey of key functional (black-box) and structural (white-box) testing techniques.
- The complementary roles of functional and structural testing techniques within dynamic analysis.

References

- "Software Testing Techniques", Beizer, B. International Thompson Computer Press, 1990.
- "Black Box Testing", Beizer, B. Wiley & Sons, 1995.
- "Applicability of modified condition/decision coverage to software testing", Chilensky, J.J. and Miller, S.P. Software Engineering Journal, 1994.