Specification Testing and Graph Coverage

Meenakshi D'Souza

International Institute of Information Technology Bangalore.

July 2017

Goals

- Understand what specifications are.
  - This lecture: Design specifications.
  - Next lecture: Requirements, use cases etc.
- What are graphs for specifications and how do graph coverage criteria work for specifications.

Design specifications

- A design specification describes aspects of what behaviour software should exhibit.
- Behavior exhibited by software need not mean the implementation directly.
- It could be a model of the implementation.
- For testing with graphs, we consider two types of design specifications.
  - Sequencing constraints on methods/functions.
  - State behaviour descriptions of software.

Sequencing constraints

- Sequencing constraints are rules that impose constraints on the order in which methods may be called.
- They can be encoded as preconditions or other specifications.
- They may or may not be given as a part of the specification or design.
- Testers need to derive them if they don't exist— they are considered another rich source of errors.

Sequencing constraints: An example

```
public int deQueue()
{
   // Pre:  At least one element must be on the queue.
   ...
   ...
public enQueue(int e)
{
   // Post:  e is on the end of the queue.
   ...
   ...
```

Sequencing constraints: Example, contd.

- Simple sequencing constraint:
  `enQueue()` must be called *before* `deQueue()`
- In the example code, it is implicitly given as pre and post conditions.
- Does not include the requirement that we must have at least as many `enQueue()` calls as `deQueue()` calls.
  - Can be handled by state behavior techniques.

Testing sequencing constraints

- Sequencing constraints may or may not be given explicitly, might not be given at all.
- Absence of sequencing constraints usually indicates more faults.
- Tests are created as sequences of method calls, testing if the sequence obeys the constraints.
    - Usually write tests to find errors in constraints or missing constraints.

Testing sequencing constraints: An ADT example

Consider a class `FileADT` that encapsulates operations on a file. Class `FileADT` has three methods:

- `open(String fName)`: Opens file with name `fName`.
- `close()`: Closes the file and makes it unavailable.
- `write(String textLine)`: Writes a line of text to the file.

What are natural sequencing constraints you would expect for this class?

Class `FileADT`: Sequencing constraints

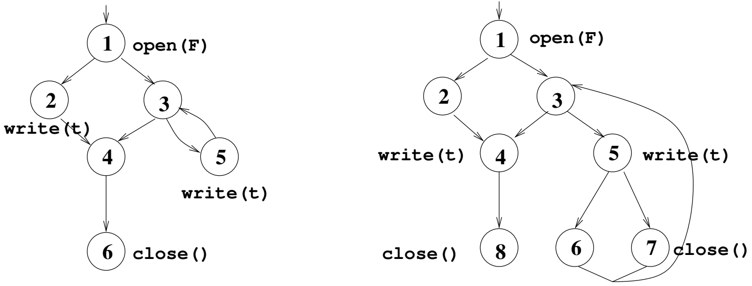
- An `open(f)` *must* be executed before every `write(t)`.
- An `open(f)` *must* be executed before every `close()`.
- A `write(f)` may not be executed after a `close()` unless there is an `open(f)` in between.
- A `write(t)` *should* be executed before every `close()`.

Note: Violation of a constraint with a *must* is a fault. Violation of a constraint with a *should* indicates a potential fault.

Testing constraints on class `FileADT`

- In every code that uses the methods of the class `FileADT`, the sequencing constraints should be satisfied.
- We consider the CFG of a code that uses these methods and find paths in the CFG that *violate* the sequencing constraints.
- These checks can be done in two ways: *statically* and *dynamically*.

Two CFGs that use class FileADT



Static testing sequence constraints of CFGs

- Proceed by checking each constraint.
- Constraint 1: Check whether paths exist from the `open(F)` at node 1 to `write(t)` at nodes 2 and 5.
- Constraint 2: Check whether a path exists from the `open(F)` at node 1 to the `close()` at node 6.
- Constraints 3 and 4:
  - Check if a path exists from node 6 to any of the nodes with `write(t)` and if a path exists from `open(F)` to `close()` without a `write(t)` in between.
  - Path [1,3,4,6] violates these constraints.

Dynamic testing sequence constraints of CFGs

- Goal again is to find test paths that violate sequencing constraints.
- For path [1,3,4,6]: It could be the case that edge (3,4) cannot be taken without going through the loop [3,5,3].
  - This cannot be checked statically, dynamic execution is necessary.
- We write test requirements that try to *violate* the sequencing constraints.
- Apply them to all programs that use this class.
- Such requirements are mostly infeasible, but, we still try to satisfy them to identify paths violating constraints.

TR for class `FileADT`

- Cover every path from the start node to every node that contains a `write()` such that the path does not go through a node containing an `open()`.
- Cover every path from the start node to every node that contains a `close()` such that the path does not go through a node containing an `open()`.
- Cover every path from every node that contains a `close()` to every node that contains a `write()`.
- Cover every path from every node that contains an `open()` to every node that contains a `close()` such that the path does not go through a node containing a `write()`.

Dynamic testing of class `FileADT`

- Dynamic testing that defines test paths for the TRs for `FileADT` reveal another error in the second CFG example.
- There is a path [1,3,5,7,4,8] which goes through two write comments in the file without a close in between.

Sequencing constraints: State behavior

- Not all sequencing constraints can be captured using simple constraints.
- Some of them need the notion of *memory* or *state* of a program.
  - Queue example: There must at least as many `enQueue()` calls as `deQueue()` calls.
  - This needs a count of each kind of call as the program executes.
- Finite state machines are useful models to describe state behaviour.

Credits

Part of the material used in these slides are derived from the presentations of the book Introduction to Software Testing, by Paul Ammann and Jeff Offutt.