IA159 Formal Verification Methods

Software Testing

Jan Strejček

Department of Computer Science
Faculty of Informatics
Masaryk University

Focus and sources

Focus

- software testing is not a typical formal method
- we focus on three formal parts of software testing
  - control flow coverage criteria
  - dataflow coverage criteria
  - model-based testing

Sources

- Chapter 9 of
  D. A. Peled: Software Reliability Methods, Springer, 2001.
- Model-based testing.
  `http://www.goldpractices.com/practices/mbt`

Basic classification

Testing can be divided according to

- the level of the tested parts

  unit (module) testing - the lowest level of testing, where one tests small pieces of code separately

  integration testing - testing that different pieces of code work well together

  system testing - testing the system as a whole

- approach to the source code

  white box testing (aka transparent box testing)- based on inspecting the source code
  suitable for unit and integration testing

  black box testing - does not use the source code (which may be inaccessible or unknown)
  suitable for system testing

Basic terminology

- **execution path** - a path in the flowchart of the tested code, i.e., it is a sequence of control points and instructions appearing in the tested code
- **test case** - a sequence of inputs, actions, and events accompanied with expected response of the system
- **test suite** - a set of test cases

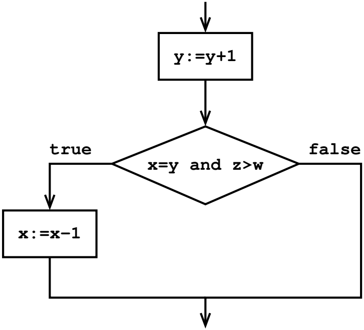
White box testing

- a typical program has a large or unbounded number of execution paths
- it is not feasible to examine all of them
- need for a reasonably small test suite with a high degree of probability of finding potential errors
- code coverage criteria are metrices saying whether a given test suite covers a given code
- testers aim to find the smallest test suite with the highest coverage
- two kinds of code coverage criteria
  - control flow coverage criteria
  - dataflow coverage criteria

White box testing

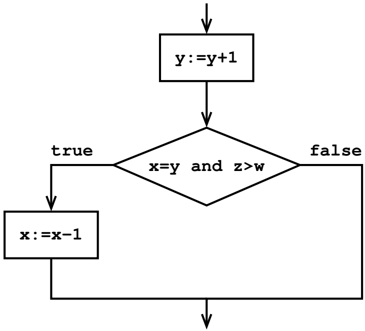
Control flow coverage criteria

Control flow coverage criteria



When is this code covered?

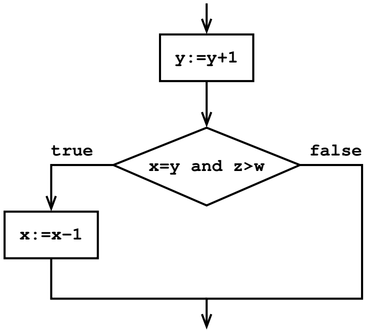
Control flow coverage criteria



statement coverage

- each executable statement (e.g. assignments, input, test, output) appears in at least one test case
- covering test case: $(x = 2, y = 2, z = 4, w = 3)$

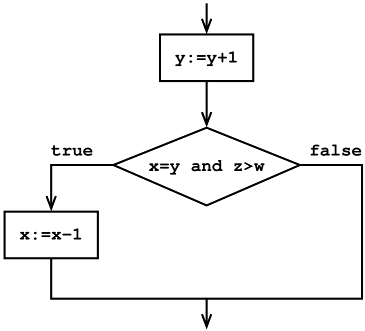
Control flow coverage criteria



edge coverage

- each execution edge of the flowchart appears in some test case
- two covering test cases: $(x = 2, y = 2, z = 4, w = 3)$, $(x = 3, y = 3, z = 5, w = 7)$

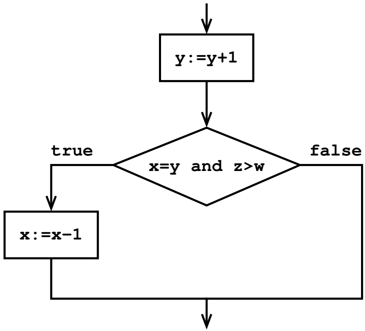
Control flow coverage criteria



condition coverage

- each decision predicate is a Boolean combination of element conditions, e.g. $\mathrm{x} < \mathrm{y}$ or $\mathrm{even}(\mathrm{x})$
- each of these element conditions appears in some test case where it is calculated to TRUE and in another test case where it is calculated to FALSE (if possible)

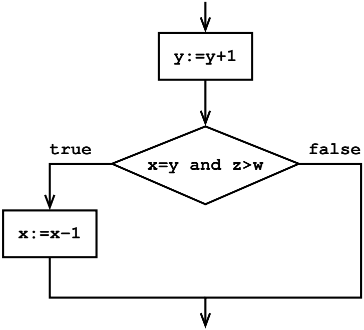
Control flow coverage criteria



condition coverage

- two covering test cases: $(x = 3, y = 3, z = 5, w = 7)$, $(x = 3, y = 4, z = 7, w = 5)$
- in both cases, the decision predicate is evaluated to FALSE…

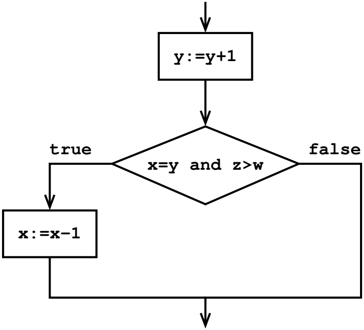
Control flow coverage criteria



edge/condition coverage

- executable edges as well as conditions has to be covered
- three covering test cases: $(x = 2, y = 2, z = 4, w = 3)$, $(x = 3, y = 3, z = 5, w = 7)$, $(x = 3, y = 4, z = 7, w = 5)$

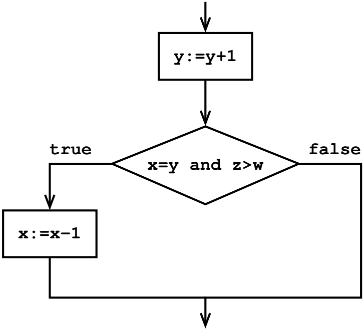
Control flow coverage criteria



multiple condition coverage

- similar to condition coverage
- each Boolean combination of TRUE/FALSE values that may appear in any decision predicate during some execution of the program must appear in some test case

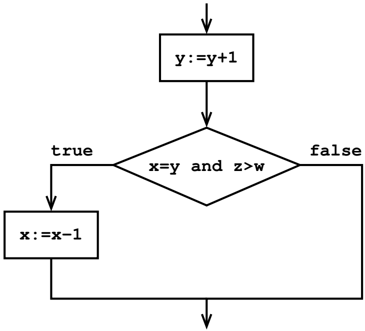
Control flow coverage criteria



multiple condition coverage

- four covering test cases: $(x = 2, y = 2, z = 4, w = 3)$, $(x = 3, y = 3, z = 5, w = 7)$, $(x = 3, y = 4, z = 7, w = 5)$, $(x = 3, y = 4, z = 5, w = 6)$
- disadvantage: an explosion of the number of test cases

Control flow coverage criteria



path coverage

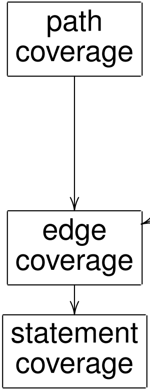
- every executable path must be covered by a test case
- the number of paths can be enormous (for example, loops may result in infinite or an unfeasible number of paths)

Hierarchy of control flow coverage criteria

- a criterion A subsumes a criterion B, denoted A $\rightarrow$ B, if guaranteeing the coverage A also guarantees B

Hierarchy of control flow coverage criteria

- a criterion A subsumes a criterion B, denoted A $\rightarrow$ B, if guaranteeing the coverage A also guarantees B



Note

It can happen due to a lucky selection of the test cases, that a less comprehensive coverage will find errors that a more comprehensive approach will happen to miss.

Loop coverage

- all mentioned criteria (except of path coverage) do not care about number of loop iterations
- ad hoc strategies for testing loops
  - check the case where the loop is skipped
  - check the case where the loop is executed once
  - check the case where the loop is executed some typical number of times (but what is typical?)
  - if the bound n on the number of iterations of the loop is known, try executing it $n-1$, n, and $n+1$ times
- testing loops become even more difficult when nested loops are involved

White box testing

Dataflow coverage criteria

Dataflow coverage criteria

There may be an execution path in which some variable is set to some value for a particular purpose, but later the value is misused. Control flow criteria do not ensure that such an execution path is included in test suite.

$$\Downarrow$$

Dataflow coverage criteria [Rapps-Weyuker, 1985]

Auxiliary sets of nodes

for each program variable x we define the following sets of flowchart nodes

$$
\begin{aligned}
\mathit{def}(x) &= \text{nodes where some value is assigned to } x \\
p{-}\mathit{use}(x) &= \text{nodes where } x \text{ is used in a predicate (e.g. in if or while statements)} \\
c{-}\mathit{use}(x) &= \text{nodes where } x \text{ is used in some expression other than a predicate}
\end{aligned}
$$

for each $s \in \mathit{def}(x)$ we further define the sets

$$
\begin{aligned}
\mathit{dpu}(s, x) &= \text{nodes } s' \in p{-}\mathit{use}(x) \text{ such that there is a path from } s \text{ to } s' \text{ going only through nodes not included in } \mathit{def}(x) \\
\mathit{dcu}(s, x) &= \text{nodes } s' \in c{-}\mathit{use}(x) \text{ such that there is a path from } s \text{ to } s' \text{ going only through nodes not included in } \mathit{def}(x)
\end{aligned}
$$

Dataflow coverage criteria

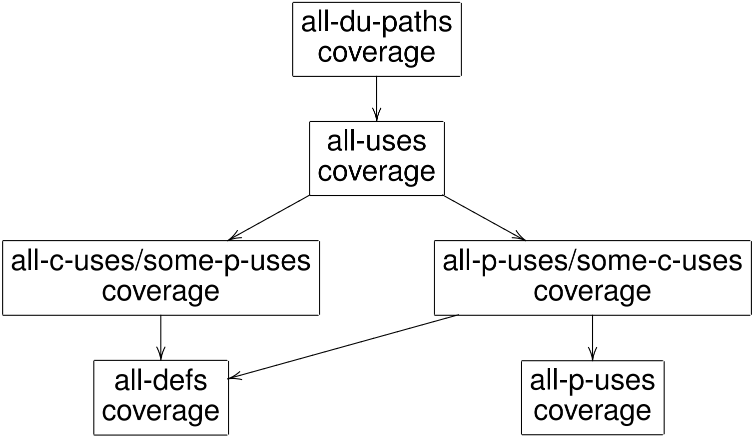
For each program variable x and each node $s \in \mathit{def}(x)$, the test suite should include the following paths starting in s and going only through nodes not included in $\mathit{def}(x)$, as subpaths:

- **all-defs:** include one path to some node in $\mathit{dpu}(s, x)$ or in $\mathit{dcu}(s, x)$.
- **all-p-uses:** include one path to each node in $\mathit{dpu}(s, x)$.
- **all-c-uses/some-p-uses:** include one path to each node in $\mathit{dcu}(s, x)$, but if $\mathit{dcu}(s, x)$ is empty, include at least one path to some node in $\mathit{dpu}(s, x)$.
- **all-p-uses/some-c-uses:** include one path to each node in $\mathit{dpu}(s, x)$, but if $\mathit{dpu}(s, x)$ is empty, include at least one path to some node in $\mathit{dcu}(s, x)$.
- **all-uses:** include one path to each node in $\mathit{dpu}(s, x)$ and to each node in $\mathit{dcu}(s, x)$.
- **all-du-paths:** include all the paths to each node in $\mathit{dpu}(s, x)$ and to each node in $\mathit{dcu}(s, x)$.

Dataflow coverage criteria

The paths should not contain cycles except for the first and the last nodes, which may be the same (e.g. an assignment $\mathtt{x} := \mathtt{x} + \mathtt{1}$ is both in $\mathit{def}(x)$ and $c{-}\mathit{use}(x)$).

Hierarchy of dataflow coverage criteria



Notes

- we cannot expect that our test suite achieves the full coverage (for example, some instructions may be unreachable)
- we cannot even compute the maximal possible coverage as it is undecidable whether a given part of the code is reachable or not

- there are other approaches to evaluation quality of a test suite, e.g. mutation analysis [Budd-DeMillo-Lipton-Frederick, POPL'80]

Mutation analysis

Idea

A test suite is unlikely to be comprehensive enough if it gives the same results to two different programs.

- given a test suite and a code, one generates several mutations of the program (based on code inspections and structural changes)
- if some test case behaves differently on the original code and a mutation, then the mutation dies
- if a considerable number of mutations remain alive, the test suite is probably inappropriate

Notes (cont.)

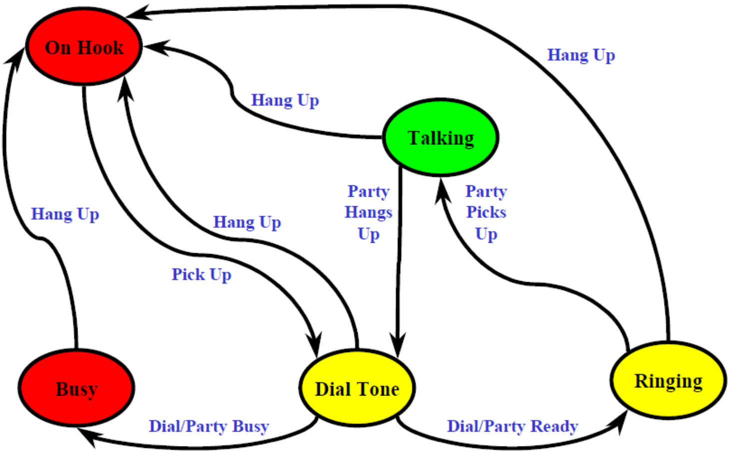
- addition of a code monitoring executions of test cases (e.g. a code measuring the coverage) can affect a behaviour of the system under test
- there are dedicated software packages for test case generation, coverage evaluation, test execution, and test management (maintaining different test suits, perform version control etc.)

Model-based testing

Black box testing: model-based testing

- to check the functionality of the system under test, we need to know the intended behaviour of the system
- the behaviour can be described by
  - simple text
  - message sequence charts
  - state machines
  - ...
- formal descriptions are called models of the system

Example: A model of a simple phone system



Model-based testing: coverage criteria

- models can be used to generate a test cases
- a test suite can be designed according to various coverage criteria
  - cover all edges
  - cover all the states
  - cover all the paths (usually impractical)
  - cover each adjacent sequence of *n* states
  - cover certain nodes at least/at most a given number of times in each test case
  - switch coverage - cover each pair of incoming and outcoming edge for all states
- after execution of the test suite and evaluation of the obtained results, we decide whether to
  - modify the model or
  - generate more tests or
  - stop testing

Probabilistic testing

- we may want to test primarily the typical executions in order to maximize minimal time to failure (MTTF)
- in this case we employ probabilistic testing
  - the system is modeled as a Markov chain
  - the test suite is then generated according to probabilities of transitions

Coming next week

Deductive software verification

- prehistory of formal verification: 40 years old technique!
- Does my program terminate? For all inputs?
- If yes, does it do what it is supposed to do?
- Can it be proven automatically?