CAS 741, CES 741 (Development of Scientific Computing Software)

Fall 2019

10 Verification and Validation Continued

Dr. Spencer Smith

Faculty of Engineering, McMaster University

October 10, 2019

Verification and Validation Continued

- Administrative details
- Questions?
- Finish what started last day
- Nonfunctional software testing
- Theoretical foundations of testing
- Complete coverage principle
- White box testing
- Oracle problem
- SCS Specific Ideas
- Overview of template

Administrative Details

- As the GitHub repo owner
  - Add your reviewers as collaborators
  - When your project is ready for review
    - Assign your reviewers an issue for them to create issues
    - Assign the instructor to review
- As a GitHub reviewer
  - Assigned 2 colleagues (see `Repos.xlsx` in repo)
  - Provide at least 5 issues on their SRS
- Reading week, no 741 classes
- V&V template in repo
- Adding a V&V checklist to repo

Administrative Details: Report Deadlines

System VnV Plan	Week 08	Oct 28
MG + MIS	Week 10	Nov 25
Final Documentation	Week 14	Dec 9

- The written deliverables will be graded based on the repo contents as of 11:59 pm of the due date
- If you need an extension, please ask
- Two days after each major deliverable, your GitHub issues will be due
- Domain expert code due 1 week after MIS deadline

Administrative Details: Presentations

Syst. VnV Present	Week 07	Week of Oct 21
MG + MIS Syntax Present	Week 9	Week of Nov 4
MIS Semantics Present	Week 11	Week of Nov 18
Unit VnV or Impl. Present	Week 12/13	Week of Nov 28

- Informal presentations with the goal of improving everyone's written deliverables
- Domain experts and secondary reviewers (and others) will ask questions (listed in Repos.xlsx file)

Administrative Details: Presentation Schedule

- Syst V&V Plan Present
  - **Monday: Deema, Peter**
  - **Thursday: Sharon, Ao**
- MG + MIS Syntax Present
  - Monday: Deema, Bo
  - Thursday: Sasha
- MIS Syntax + Semantics Present
  - Monday: Zhi, Peter
  - Thursday: Sharon, Ao
- Unit VnV Plan or Impl. Present
  - Monday: Bo, Sasha
  - Thursday: Zhi, Peter, Ao

Questions?

- Questions about SRS?
- Questions about V&V?

Goals of Testing

- If our code passes all test cases, is it now guaranteed to be error free?
- Are 5000 random tests always better than 5 carefully selected tests?

Goals of Testing

- To show the presence of bugs (Dijkstra, 1972)
- If tests do not detect failures, we cannot conclude that software is defect-free
- Still, we need to do testing - driven by sound and systematic principles
  - Random testing is often not a systematic principle to use
  - Need a test plan
- Should help isolate errors - to facilitate debugging

Goals of Testing Continued

- Should be repeatable
  - Repeating the same experiment, we should get the same results
  - Repeatability may not be true because of the effect of the execution environment on testing
  - Repeatability may not occur if there are uninitialized variables
  - Repeatability may not happen when there is nondeterminism
- Should be accurate
  - Accuracy increases reliability
  - Part of the motivation for formal specification
- Is a successful test case one that passes the test, or one that shows a failure?

Test (V&V) Plan

- Given that no single verification technique can prove correctness, the practical approach is to use ALL verification techniques. Is this statement True or False?

Test (V&V) Plan

- Testing can uncover errors and build confidence in the software
- Resources of time, people, facilities are limited
- Need to plan how the software will be tested
- You know in advance that the software is unlikely to be perfect
- You need to put resources into the most important parts of the project
- A risk analysis can determine where to put your limited resources
- A risk is a condition that can result in a loss
- Risk analysis involves looking at how bad the loss can be and at the probability of the loss occurring

White Box Versus Black Box Testing

- Do you know (or can you guess) the difference between white box and black box testing?
- What if they were labelled transparent box and opaque box testing, respectively?

White Box Versus Black Box Testing

- White box testing is derived from the program's internal structure
- Black box testing is derived from a description of the program's function
- Should perform both white box and black box testing
- Black box testing
  - Uncovers errors that occur in implementing requirements or design specifications
  - Not concerned with how processing occurs, but with the results
  - Focuses on functional requirements for the system
  - Focuses on normal behaviour of the system

White Box Testing

- Uncovers errors that occur during implementation of the program
- Concerned with how processing occurs
- Evaluates whether the structure is sound
- Focuses on abnormal or extreme behaviour of the system

Dynamic Testing

- Is there a dynamic testing technique that can guarantee correctness?
- If so, what is the technique?
- Is this technique practical?

Dynamic Versus Static Testing

- Another classification of verification techniques, as previously discussed
- Use a combination of dynamic and static testing
- Dynamic analysis
  - Requires the program to be executed
  - Test cases are run and results are checked against expected behaviour
  - Exhaustive testing is the only dynamic technique that guarantees program validity
  - Exhaustive testing is usually impractical or impossible
  - Reduce number of test cases by finding criteria for choosing representative test cases

Static Testing Continued

- Static analysis
  - Does not involve program execution
  - Testing techniques simulate the dynamic environment
  - Includes syntax checking
  - Generally static testing is used in the requirements and design stage, where there is no code to execute
  - Document and code walkthroughs
  - Document and code inspections

Manual Versus Automated Testing

- What is the difference between manual and automated testing?
- What are the advantages of automated testing?
- What is regression testing?

Manual Versus Automated Testing

- Manual testing
  - Has to be conducted by people
  - Includes by-hand test cases, structured walkthroughs, code inspections
- Automated testing
  - The more automated the development process, the easier to automate testing
  - Less reliance on people
  - Necessary for regression testing
  - Test tools can assist, such as Junit, Cppunit, CuTest etc.
  - Can be challenging to automate GUI tests
  - Test suite for Maple has 2 000 000 test cases, run on 14 platforms, every night, automated reporting

Continuous Integration Testing

- What is continuous integration testing?

Continuous Integration Testing

- Information available on Wikipedia
- Developers integrate their code into a shared repo frequently (multiple times a day)
- Each integration is automatically accompanied by regression tests and other build tasks
- Build server
  - Unit tests
  - Integration tests
  - Static analysis
  - Profile performance
  - Extract documentation
  - Update project web-page
  - Portability tests
  - etc.
- Avoids potentially extreme problems with integration when the baseline and a developer's code greatly differ

Continuous Integration Tools

- Gitlab
  - Example at Rogue Reborn
  - Drasil
    - Details of Travis CI steps in .travis.yml file
    - Automated case study documentation, code and gen code documentation
    - Automated build of dependency graphs (bottom of page)
- Jenkins
- Travis
- Docker
  - Eliminates the “it works on my machine” problem
  - Package dependencies with your apps
  - A container for lightweight virtualization
  - Not a full VM

Sample Nonfunctional System Testing

- Stress testing - Determines if the system can function when subject to large volumes
- Usability testing
- Performance measurement

Sample Functional System Testing

- Parallel: Determines the results of the new application are consistent with the processing of the previous application or version of the application

Theoretical Foundations Of Testing: Definitions

- P (program), D (input domain), R (output domain)
  - P: D $\rightarrow$ R (may be partial)
- Correctness defined by OR $\subseteq$ D $\times$ R
  - P(d) correct if $\langle$ d, P(d) $\rangle \in$ OR
  - P correct if all P(d) are correct
- Failure
  - P(d) is not correct
  - May be undefined (error state) or may be the wrong result
- Error (Defect)
  - Anything that may cause a failure
    - Typing mistake
    - Programmer forgot to test "x=0"
- Fault
  - Incorrect intermediate state entered by program

Definitions Questions

- A test case t is an element of D or R?
- A test set T is a finite subset of D or R?
- How would we define whether a test is successful?
- How would we define whether a test set is successful?

Definitions Continued

- Test case t: An element of D
- Test set T: A finite subset of D
- Test is successful if P(t) is correct
- Test set successful if P correct for all t in T

Theoretical Foundations of Testing

- Desire a test set T that is a finite subset of D that will uncover all errors
- Determining and ideal T leads to several undecideable problems
- No algorithm exists:
  - To state if a test set will uncover all possible errors
  - To derive a test set that would prove program correctness
  - To determine whether suitable input exists to guarantee execution of a given statement in a given program
  - etc.

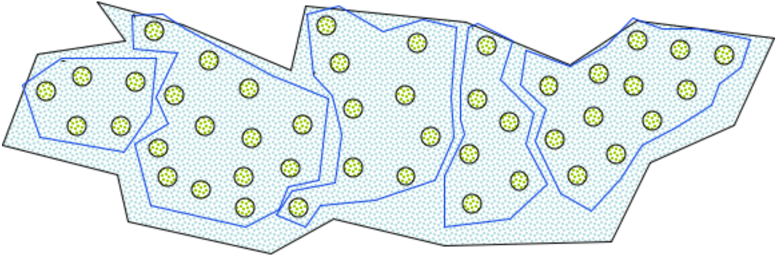
Empirical Testing

- Need to introduce empirical testing principles and heuristics as a compromise between the impossible and the inadequate
- Find a strategy to select significant test cases
- Significant means the test cases have a high potential of uncovering the presence of errors

Complete-Coverage Principle

- Try to group elements of D into subdomains D_1, D_2, ..., D_n where any element of each D_i is likely to have similar behaviour
- $D = D_1 \cup D_2 \cup ... \cup D_n$
- Select one test as a representative of the subdomain
- If $D_j \cap D_k = \emptyset$ for all $j \neq k$, (partition), any element can be chosen from each subdomain
- Otherwise choose representatives to minimize number of tests, yet fulfilling the principle

Complete-Coverage Principle



White-box Testing

- Intuitively, after running your test suites, what percentage of the lines of code in your program should be exercised?

White-box Coverage Testing

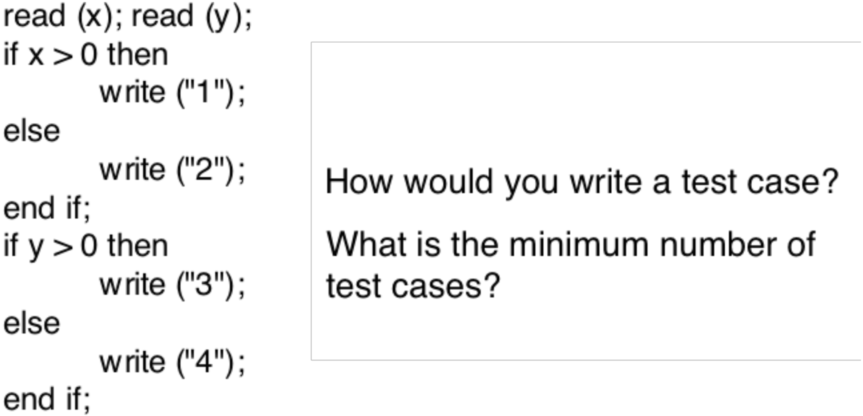
- (In)adequacy criteria - if significant parts of the program structure are not tested, testing is inadequate
- Control flow coverage criteria
  - Statement coverage
  - Edge coverage
  - Condition coverage
  - Path coverage

Examples that follow are from [1]

Statement-Coverage Criterion

- Select a test set T such that every elementary statement in P is executed at least once by some d in T
- An input datum executes many statements - try to minimize the number of test cases still preserving the desired coverage

Example

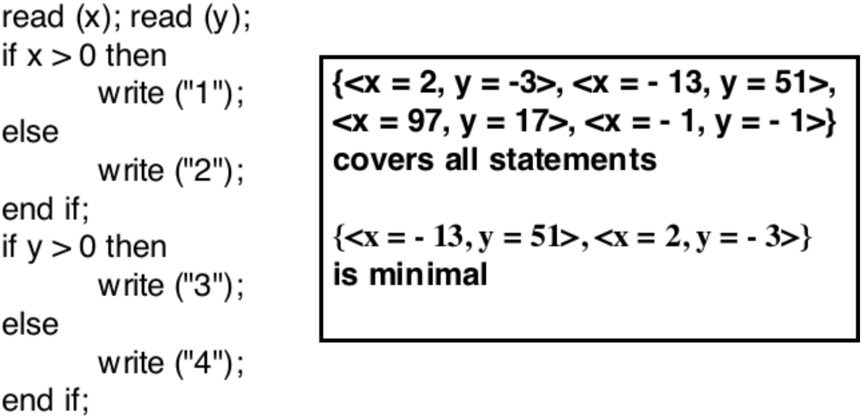


```
read (x); read (y);
if x > 0 then
        write ("1");
else
        write ("2");
end if;
if y > 0 then
        write ("3");
else
        write ("4");
end if;
```

How would you write a test case?

What is the minimum number of test cases?

Example

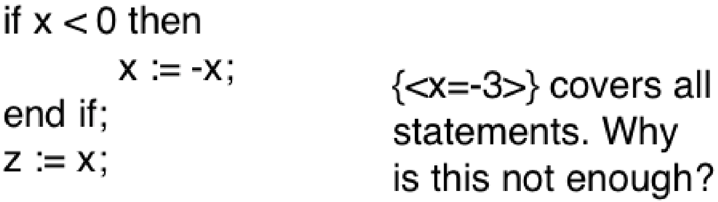


```
read (x); read (y);
if x > 0 then
        write ("1");
else
        write ("2");
end if;
if y > 0 then
        write ("3");
else
        write ("4");
end if;
```

{<x = 2, y = -3>, <x = - 13, y = 51>, <x = 97, y = 17>, <x = - 1, y = - 1>} covers all statements

{<x = - 13, y = 51>, <x = 2, y = - 3>} is minimal

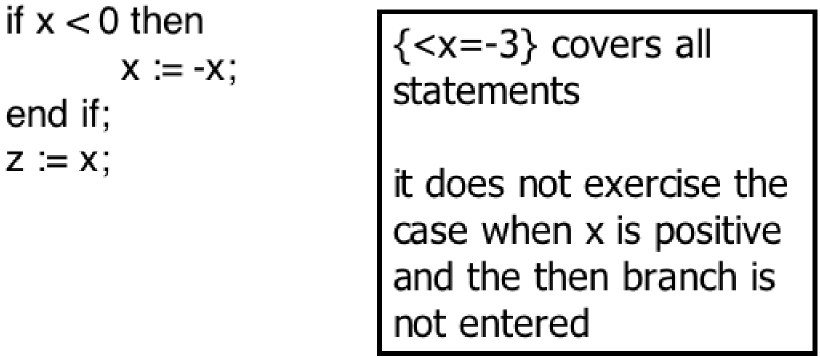
Weakness of the Criterion



```
if x < 0 then
        x := -x;
end if;
z := x;
```

{<x=-3>} covers all statements. Why is this not enough?

Weakness of the Criterion



```
if x < 0 then
        x := -x;
end if;
z := x;
```

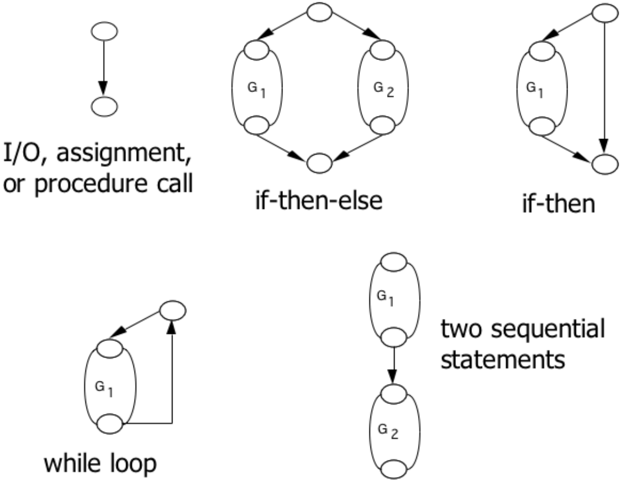
{<x=-3} covers all statements

it does not exercise the case when x is positive and the then branch is not entered

Edge-Coverage Criterion

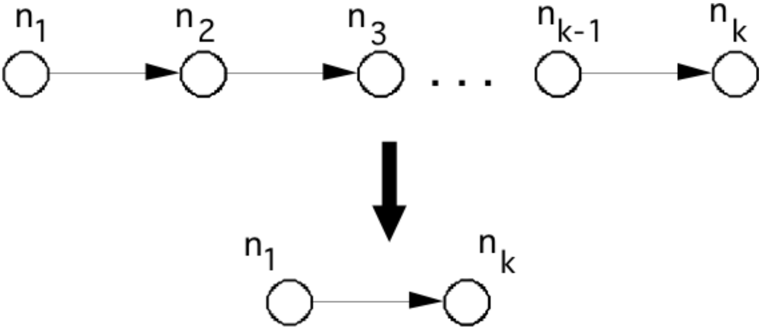
- Select a test set T such that every edge (branch) of the control flow is exercised at least once by some d in T
- This requires formalizing the concept of the control graph and how to construct it
  - Edges represent statements
  - Nodes at the ends of an edge represent entry into the statement and exit

Control Graph Construction Rules

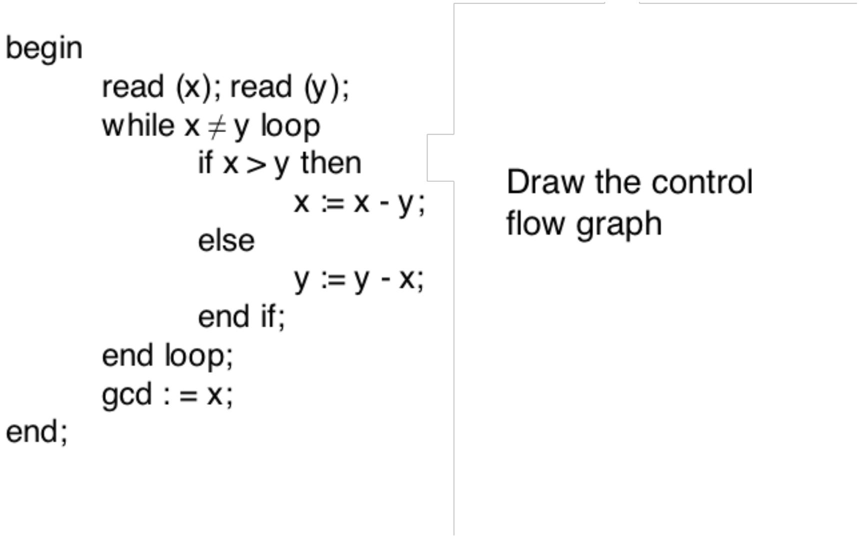


Simplification

A sequence of edges can be collapsed into just one edge



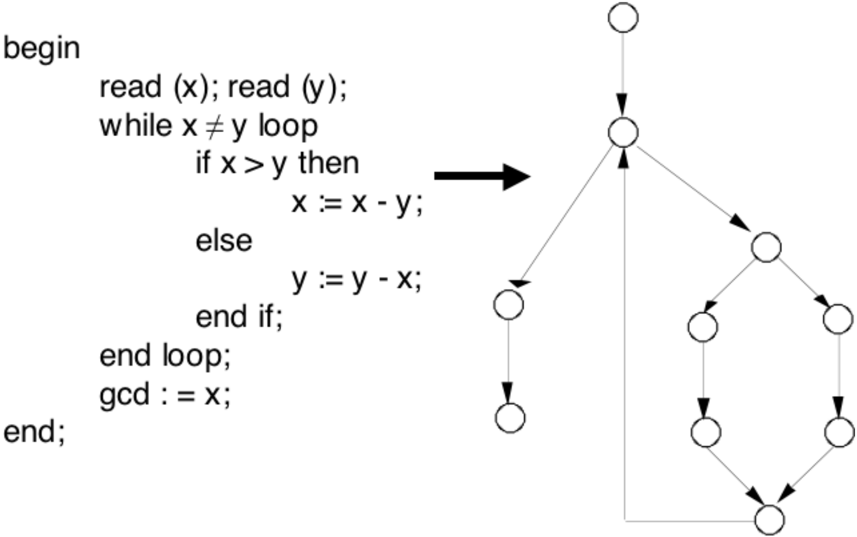
Example: Euclid's Algorithm



```
begin
    read (x); read (y);
    while x ≠ y loop
        if x > y then
            x := x - y;
        else
            y := y - x;
        end if;
    end loop;
    gcd : = x;
end;
```

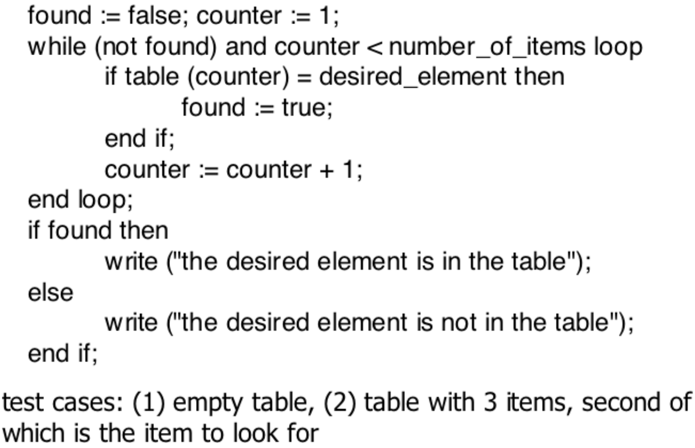
Draw the control flow graph

Example: Euclid's Algorithm



```
begin
        read (x); read (y);
        while x ≠ y loop
                if x > y then
                        x := x - y;
                else
                        y := y - x;
                end if;
        end loop;
        gcd : = x;
end;
```

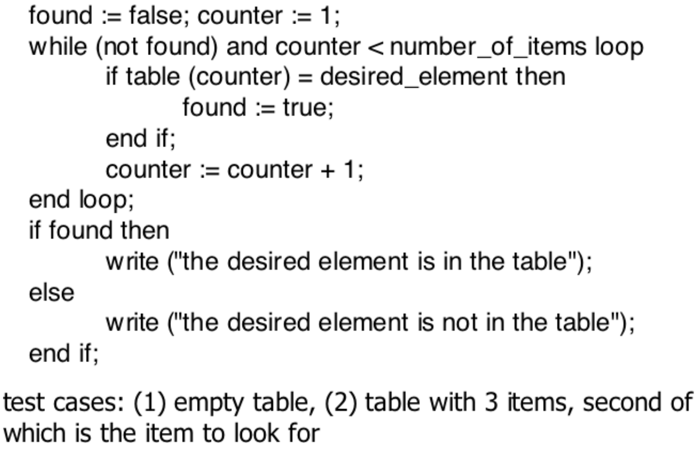
Weakness



```
found := false; counter := 1;
while (not found) and counter < number_of_items loop
        if table (counter) = desired_element then
                found := true;
        end if;
        counter := counter + 1;
end loop;
if found then
        write ("the desired element is in the table");
else
        write ("the desired element is not in the table");
end if;
```

test cases: (1) empty table, (2) table with 3 items, second of which is the item to look for

Weakness



```
found := false; counter := 1;
while (not found) and counter < number_of_items loop
        if table (counter) = desired_element then
                found := true;
        end if;
        counter := counter + 1;
end loop;
if found then
        write ("the desired element is in the table");
else
        write ("the desired element is not in the table");
end if;
```

test cases: (1) empty table, (2) table with 3 items, second of which is the item to look for

Do not discover the error ($<$ instead of $\leq$)

```
if c1 and c2 then
   st;
else
   sf;

// equivalent to

if c1 then
   if c2 then
     st;
   else
     sf;
else
   sf;
```

Condition-Coverage Criterion

- Select a test set T such that every edge of P's control flow is traversed and all possible values of the constituents of compound conditions are exercised at least once
- This criterion is finer than edge coverage

Weakness

```
if $x \neq 0$ then
        y := 5;
else
        z := z - x;
end if;
if $z > 1$ then
        z := z / x;
else
        z := 0;
end if;
```

{<x = 0, z = 1>, <x = 1, z = 3>} causes the execution of all edges, but fails to expose the risk of a division by zero

Path-Coverage Criterion

- Select a test set T that traverses all paths from the initial to the final node of Ps control flow
- It is finer than the previous kinds of coverage
- However, number of paths may be too large, or even infinite (see while loops)
- Loops
  - Zero times (or minimum number of times)
  - Maximum times
  - Average number of times

The Infeasibility Problem

- Syntactically indicated behaviours (statements, edges, etc.) are often impossible
- Unreachable code, infeasible edges, paths, etc.
- Adequacy criteria may be impossible to satisfy
    - Manual justification for omitting each impossible test case
    - Adequacy “scores” based on coverage - example 95 % statement coverage

Further Problem

- What if the code omits the implementation of some part of the specification?
- White box test cases derived from the code will ignore that part of the specification!

Testing Boundary Conditions

- Testing criteria partition input domain in classes, assuming that behavior is “similar” for all data within a class
- Some typical programming errors, however, just happen to be at the boundary between different classes
  - Off by one errors
  - $<$ instead of $\leq$
  - equals zero

Criterion

- After partitioning the input domain D into several classes, test the program using input values not only “inside” the classes, but also at their boundaries
- This applies to both white-box and black-box techniques
- In practice, use the different testing criteria in combinations
- Use testing tools for coverage metrics

The Oracle Problem

When might it be difficult to know the "expected" output/behaviour?

The Oracle Problem

- Given input test cases that cover the domain, what are the expected outputs?
- Oracles are required at each stage of testing to tell us what the right answer is
- Black-box criteria are better than white-box for building test oracles
- Automated test oracles are required for running large amounts of tests
- Oracles are difficult to design - no universal recipe

The Oracle Problem Continued

- Determining what the right answer should be is not always easy
  - Scientific computing
  - Machine learning
  - Artifical intelligence

The Oracle Problem Continued

What are some strategies we can use when we do not have a test oracle?

Strategies Without An Oracle

- Using an independent program to approximate the oracle (pseudo oracle)
- Method of manufactured solutions
- Properties of the expected values can be easier than stating the expected output
    - Examples?

Strategies Without An Oracle

- Using an independent program to approximate the oracle (pseudo oracle)
- Method of manufactured solutions
- Properties of the expected values can be easier than stating the expected output
  - Examples?
  - List is sorted
  - Number of entries in file matches number of inputs
  - Conservation of energy or mass
  - Expected trends in output are observed (metamorphic testing [5, 4, 6])
  - etc.

Challenges Specific to Scientific Computing

- Unknown solution
- Approximation of real numbers
- Nonfunctional requirements
- Parallel computation

Mutation Testing for SC

- Generate changes to the source code, called mutants, which become code faults
- Mutants include changing an operation, modifying constants, changing the order of execution, etc.
- The adequacy of a set of tests is established by running the tests on all generated mutants
- Need to account for floating point approximations
- See [3]

Specific SC V&V Approaches

Summary of most points below in [10]

- Compare to closed-form solutions
- Method of manufactured solutions [8]
- Interval arithmetic [2]
- Convergence studies
- Compare to other program (parallel testing)
- Can also consider using code inspection
  - [7, 9]
  - Sample checklists

Specific SC V&V NonFunctional

- Installability, consider VMs
- Portability, consider VMs, Docker, CI
- Describe (rather than specify) impact of changing inputs
    - Accuracy
    - Performance
    - Relative comparison
- Usability
    - Fairly simple standard survey
    - Example

Validation Testing Report for PMGT

- Prepared by Wen Yu (here)
- Do not know the correct solution, but know properties of the correct solution
- Automated correctness validation tests
  - The area of each element is greater than zero
  - The boundary of the mesh is closed
  - Vertices in a clockwise order
  - $nc + nv - ne = 1$
  - ...
- Visual correctness verification tests
  - No vertex outside the input domain
  - No vertex inside a cell
  - No dangling edges
  - All cells connected
  - The mesh is conformal

Validation Testing Report for PMGT (Continued)

- List and description of test cases
- Test cases are labelled and numbered
- Traceability to SRS requirements
- Traceability to MG
- Summary of results
- Analysis of results
  - Focus on nonfunctional requirements
  - Speed

Test Plan From BlankProjectTemplate

- Add links to templates
- For Unit VnV plan mention tools
  - Linters
  - Coding standard checkers (like flake8)
  - unit testing frameworks
  - Performance testing (like Valgrind)

References I

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli.
Fundamentals of Software Engineering.
Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.

Timothy Hickey, Qun Ju, and Maarten H. Van Emden.
Interval arithmetic: From principles to implementation.
J. ACM, 48(5):1038–1068, September 2001.

Daniel Hook and Diane Kelly.
Testing for trustworthiness in scientific software.
In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, SECSE '09, pages 59–64, Washington, DC, USA, 2009. IEEE Computer Society.

References II

- U. Kanewala and J. M. Bieman.
  Techniques for testing scientific programs without an oracle.
  In *Software Engineering for Computational Science and Engineering (SE-CSE), 2013 5th International Workshop on*, pages 48–57, May 2013.

- Upulee Kanewala, James M. Bieman, and Asa Ben-Hur.
  Predicting metamorphic relations for testing scientific software: A machine learning approach using graph kernels.
  Software Testing Verification and Reliability, preprint, 2015.

References III

- Upulee Kanewala and Anders Lundgren.
  Automated metamorphic testing of scientific software.
  In Jeffrey C. Carver, Neil Chue Hong, and George Thiruvathukal, editors, *Software Engineering for Science*, Chapman & Hall/CRC Computational Science, chapter Examples of the Application of Traditional Software Engineering Practices to Science, pages 151–174. Taylor & Francis, 2016.

- Diane Kelly and Terry Shepard.
  Task-directed software inspection technique: an experiment and case study.
  In *CASCON '00: Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, page 6. IBM Press, 2000.

References IV

- Patrick J. Roache.
  Verification and Validation in Computational Science and Engineering.
  Hermosa Publishers, Albuquerque, New Mexico, 1998.

- Terry Shepard and Diane Kelly.
  How to do inspections when there is no time.
  In *Proceedings of the 23rd International Conference on Software Engineering*, page 718. IEEE Computer Society, 2001.

References V

W. Spencer Smith.
A rational document driven design process for scientific computing software.
In Jeffrey C. Carver, Neil Chue Hong, and George Thiruvathukal, editors, *Software Engineering for Science*, chapter Section I – Examples of the Application of Traditional Software Engineering Practices to Science, pages 33–63. Taylor & Francis, 2016.