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

Fall 2017

09 Verification and Validation

Dr. Spencer Smith

Faculty of Engineering, McMaster University

October 3, 2017

Verification and Validation

- Administrative details
- Questions?
- 741 workflow
- Testing from SE perspective
- Testing from SC perspective
- V&V template
- V&V examples
  - SWHS
  - Mesh Gen
  - Rogue Reborn

Administrative Details

- SRS Presentation grades on Avenue
- Apology that unable to review full SRS
- GitHub issues for colleagues
  - Assigned 1 colleague (see `Repos.xlsx` in repo)
  - Provide at least 5 issues on their SRS
  - Grading
    - Not enough issues, or poor issues 0/2
    - Enough issues, but shallow 1/2
    - Enough issues and deep (not surface) 2/2
  - Due by Tuesday, Oct 10, 11:59 pm
- Reading week next week, no 741 classes
- V&V template updated in repo

Administrative Details: Deadlines

SRS	Week 05	Oct 4
SRS Issues	Reading week	Oct 10
V&V Present	Week 06	Week of Oct 16
V&V Plan	Week 07	Oct 25
MG Present	Week 08	Week of Oct 30
MG	Week 09	Nov 8
MIS Present	Week 10	Week of Nov 13
MIS	Week 11	Nov 22
Impl. Present	Week 12	Week of Nov 27
Final Documentation	Week 13	Dec 6

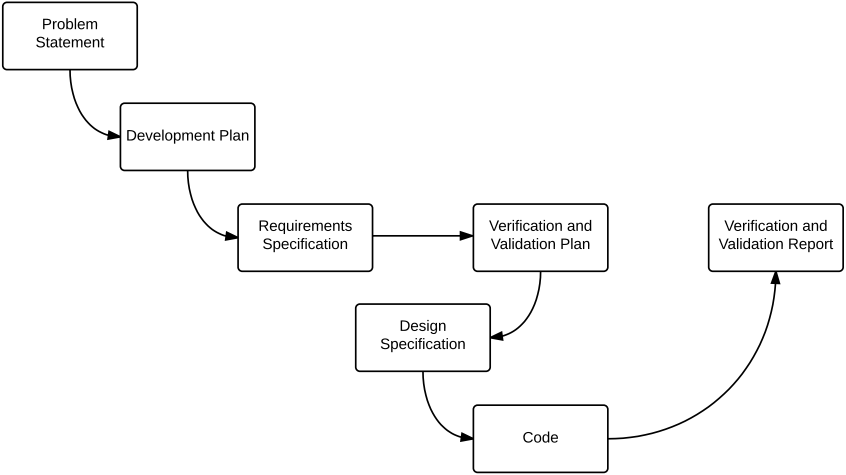
Administrative Details: Presentation Schedule

- V&V Present
  - **Tuesday: Steven, Alexandre P., Alexander S.**
  - **Friday: Geneva, Jason, Yuzhi**
- MG Present
  - Tuesday: Xiaoye, Shusheng, Devi, Keshav, Alex P, Paul
  - Friday: Yuzhi, Jason, Geneva, Alex S, Isobel, Steven
- MIS Present
  - Tuesday: Isobel, Keshav, Paul
  - Friday: Shusheng, Xiaoye, Devi
- Impl. Present
  - Tuesday: Alexander S., Steven, Alexandre P.
  - Friday: Jason, Geneva, Yuzhi

Questions?

- Questions about SRS?

"Faked" Rational Design Process



Outline of Verification Topics

- What are the goals of verification?
- What are the main approaches to verification?
  - What kind of assurance do we get through testing?
  - Can testing prove correctness?
  - How can testing be done systematically?
  - How can we remove defects (debugging)?
- What are the main approaches to software analysis?
- Informal versus formal analysis

Incorrect Version of Delete

Using `s = new T[MAX_SIZE]`, for some type `T`

```
public static void del(int i)
{
  int j;

  for (j = i; j <= (length - 1); j++)
  {
    s[j] = s[j+1];
  }

  length = length - 1;
}
```

- What is the error?
- What test case would highlight the error?

Correct Version of Delete

```
public static void del(int i)
{
  int j;

  for (j = i; j < (length - 1); j++)
  {
    s[j] = s[j+1];
  }

  length = length - 1;
}
```

Avoids potential ArrayIndexOutOfBoundsException Exception

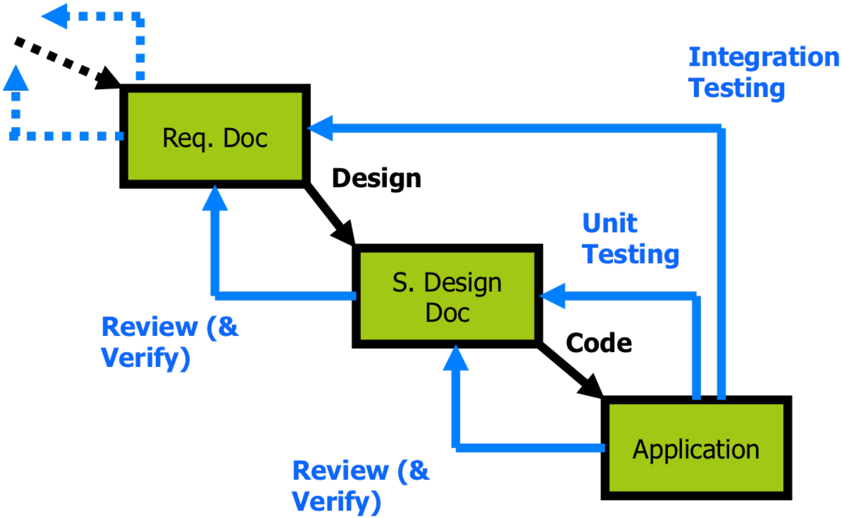
Verification Versus Validation

- What is the difference between verification and validation?

Verification Versus Validation

- Verification - Are we building the product right? Are we implementing the requirements correctly (internal)
- Validation - Are we building the right product? Are we getting the right requirements (external)
- According to Capability Maturity Model (CMM)
  - Software Verification: The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. [IEEE-STD-610]
  - Software Validation: The process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements. [IEEE-STD-610]
- We will focus on verification

Verification Activities



Testing Phases

1. Unit testing
2. Integration testing
3. System testing
4. Acceptance testing

Need for Verification

- Designers are fallible even if they are skilled and follow sound principles
- We need to build confidence in the software
- Everything must be verified, every required functionality, every required quality, every process, every product, every document
- For every work product covered in this class we have discussed its verification
- Even verification itself must be verified

Properties of Verification

From [1]

- May not be binary (OK, not OK)
  - Severity of defect is important
  - Some defects may be tolerated
  - Our goal is typically acceptable reliability, not correctness
- May be subjective or objective - for instance, usability, generic level of maintainability or portability
  - How might we make usability objective?
- Even implicit qualities should be verified
  - Because requirements are often incomplete
  - For instance robustness, maintainability
- What is better than implicitly specified qualities?

Approaches to Verification

- What are some approaches to verification?
- How can we categorize these approaches?

Approaches to Verification

- Experiment with behaviour of product
  - Sample behaviours via testing
  - Goal is to find "counter examples"
  - Dynamic technique
  - Examples: unit testing, integration testing, acceptance testing, white box testing, stress testing, etc.
- Analyze product to deduce its adequacy
  - Analytic study of properties
  - Static technique
  - Examples: Code walk-throughs, code inspections, correctness proof, etc.

Does our Engineering Analogy Fail?

- If a bridge can hold 512 kN, can it hold 499 kN?
- If our software works for the input 512, will it work for 499?

Verification in Engineering

- Example of bridge design
- One test assures infinite correct situations
- In software a small change in the input may result in significantly different behaviour
- There are also chaotic systems in nature, but products of engineering design are usually stable and well-behaved

Modified Version Works for 512, but not 499

```
procedure binary-search (key: in element;
                  table: in elementTable; found: out Boolean) is
begin
   bottom := table'first; top := table'last;
   while bottom < top loop
      if (bottom + top) rem 2 ≠ 0 then
         middle := (bottom + top - 1) / 2;
      else
         middle := (bottom + top) / 2;
      end if;
      if key ≤ table (middle) then
         top := middle;
      else
         bottom := middle + 1;
      end if;
   end loop;
   found := key = table (top);
end binary-search
```

if we omit this the routine works if the else is never hit! (i.e. if size of table is a power of 2)

Testing and Lack of “Continuity”

- Testing samples behaviours by examining “test cases”
- Impossible to extrapolate behaviour of software from a finite set of test cases
- No continuity of behaviour - it can exhibit correct behaviour in infinitely many cases, but may still be incorrect in some cases

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
- 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

- Requirements: Determines if the system can perform its function correctly and that the correctness can be sustained over a continuous period of time
- Error Handling: Determines the ability of the system to properly process incorrect transactions
- Manual Support: Determines that the manual support procedures are documented and complete, where manual support involves procedures, interfaces between people and the system, and training procedures
- 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.

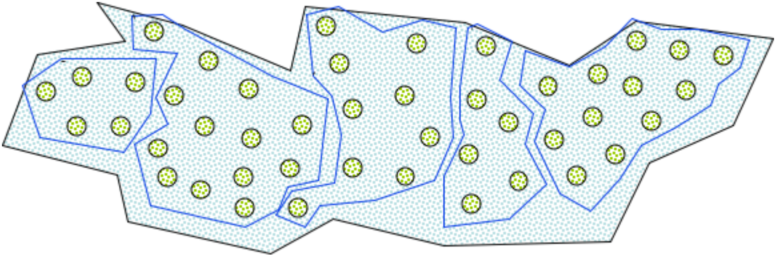
Empirical Testing

- 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

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;
```

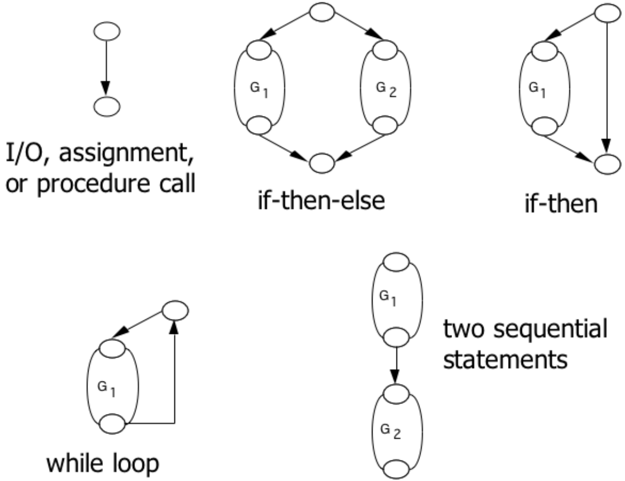
{<x=-3} covers all statements

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

Edge-Coverage Criterion

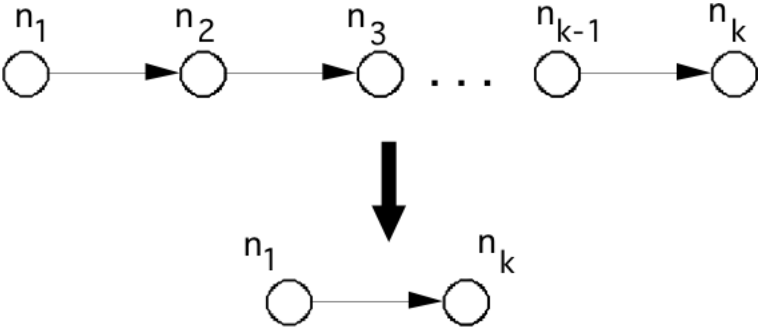
- Select a test set T such that every edge (branch) of the control flow is exercised at least once by some d in T
- This requires formalizing the concept of the control graph and how to construct it
  - Edges represent statements
  - Nodes at the ends of an edge represent entry into the statement and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

Control Graph Construction Rules



Simplification

A sequence of edges can be collapsed into just one edge



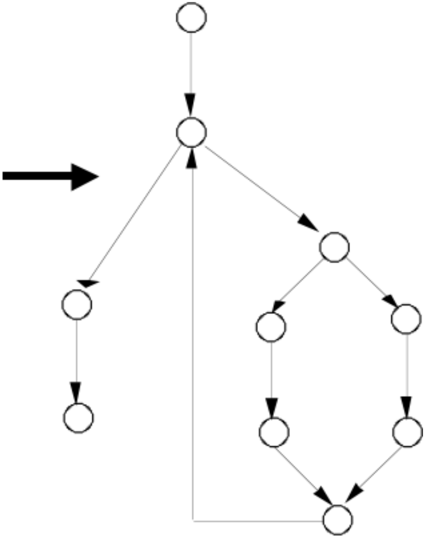
Example: Euclid's Algorithm

```
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 ≠ 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

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
  - 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)
  - etc.

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 Hook and Kelly, 2009

Analysis of Units

- Dynamic testing of units is not the only option
- Static testing (analysis) includes the following
  - Informal inspection
  - Systematic inspection
  - Code walkthroughs, data flow analysis
  - Correctness proofs (for instance using pre and post conditions)
  - Complexity measures

Challenges Specific to Scientific Computing

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

Validation Testing Report for PMGT

- Prepared by Wen Yu
- 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 validation 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

References I

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