CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Lecture 10: Testing Near and Post-Release
(and completion of Data Flow Analysis)

Gregory Gay
DIT635 - February 21, 2020

Today's Goals

- Finish Data Flow Analysis (from last time)
- Testing activities as we near release (and after):
  - Acceptance Testing: Giving users the system
  - Regression Testing: Ensuring re-designed code still works.

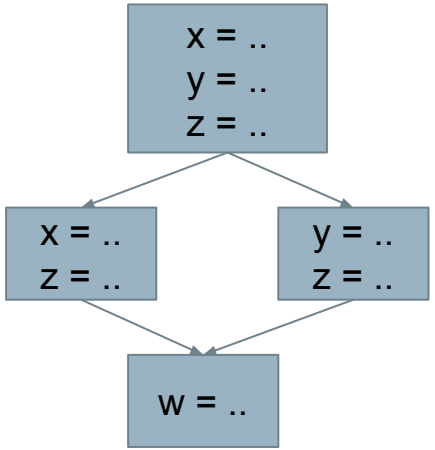
Data Flow Analysis

Reachability

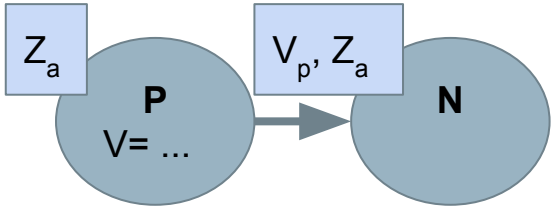
- Def-Use pairs describe paths through the program's control flow.
  - There is a (*d,u*) pair for variable *V* only if at least one path exists between *d* and *u*.
  - If this is the case, a definition V_d **reaches** *u*.
    - V_d is a *reaching definition* at *u*.
  - If the path passes through a new definition V_e, then V_e *kills* V_d.

Computing Def-Use Pairs

- One algorithm: Search the CFG for paths without redefinitions.
  - Not practical - remember path coverage?
- Instead, summarize the reaching definitions at a node over all paths reaching that node.



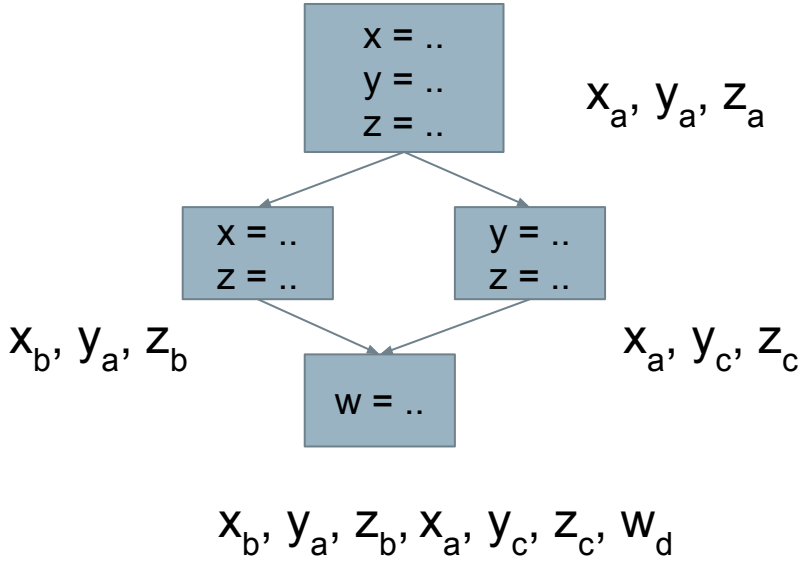
Computing Def-Use Pairs



- If we calculate the reaching definitions of node *n*, and there is an edge (*p, n*) from an immediate predecessor node *p*.
  - If p can assign a value to variable *v*, then definition v_p reaches *n*.
    - v_p is *generated* at *p*.
  - If a definition v_d reaches *p*, and if there is no new definition, then v_d is *propagated* from *p* to *n*.
    - If there is a new definition, v_p kills v_d and v_p propagates to n.

Computing Def-Use Pairs

- The reaching definitions flowing out of a node include:
  - All reaching definitions flowing in.
  - Minus definitions that are killed.
  - Plus definitions that are generated.



Flow Equations

- As node *n* may have multiple predecessors, we must merge their reaching definitions:
  - ReachIn(n) = $\bigcup_{p \in pred(n)}$ ReachOut(p)
- The definitions that reach out are those that reach in, minus those killed, plus those generated.
  - ReachOut(n) = (ReachIn(n) \ kill(n)) $\cup$ gen(n)

Computing Reachability

- Initialize
  - *ReachOut* is empty for every node.
- Repeatedly update
  - Pick a node and recalculate *ReachIn*, *ReachOut.*
- Stop when stable
  - No further changes to *ReachOut* for any node
  - Guaranteed because the flow equations define a *monotonic* function on the finite *lattice* of possible sets of reaching definition.

Iterative Worklist Algorithm

- Initialize the reaching definitions flowing out to G
- Keep a *worklist* of nodes to be processed.
- At each step remove an element from the *worklist*
- Calculate the flow equations.
- If the recalculated value is different for the node add its successors to the worklist.

```
for(n ∈ nodes){
    ReachOut(n) = {};
}
workList = nodes;
while(workList != {}){
    n = a node from the workList;
    workList = workList \ {n};
    oldVal = ReachOut(n);
    ReachIn(n) = ⋃_{p∈pred(n)} ReachOut(p);
    ReachOut(n) = (ReachIn(n) \
                   kill(n)) ⋃ gen(n);
    if(ReachOut != oldVal){
        workList = workList ⋃ succ(n);
    }
}
```

Can this algorithm work for other analyses?

- ReachIn/ReachOut are flow equations.
  - They describe passing information over a graph.
  - Many other program analyses follow a common pattern.
- Initialize-Repeat-Until-Stable Algorithm
  - Would work for any set of flow equations as long as the constraints for convergence are satisfied.
- Another problem - expression availability.

Available Expressions

- When can the value of a subexpression be saved and reused rather than recomputed?
  - Classic data-flow analysis, often used in compiler.
- Can be defined in terms of paths in a CFG.
- An expression is ***available*** if - for all paths through the CFG - the expression has been computed and not later modified.
  - Expression is *generated* when computed.
  - … and *killed* when any part of it is redefined.

Available Expressions

- Like with reaching, availability can be described using flow equations.
- The expressions that become available (gen set) and cease to be available (kill set) can be computed simply.
- Flow equations:
  - $\text{AvailIn}(n) = \bigcap_{p \in \text{pred}(n)} \text{AvailOut}(p)$
  - $\text{AvailOut}(n) = (\text{AvailIn}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$

Iterative Worklist Algorithm

- Input:
  - A control flow graph G = (nodes, edges)
  - pred(n)
  - succ(n)
  - gen(n)
  - kill(n)
- Output:
  - AvailIn(n)

```
for(n ∈ nodes){
    AvailOut(n) = set of all expressions
    defined anywhere;
}
workList = nodes;
while(workList != {}){
    n = a node from the workList;
    workList = workList \ {n};
    oldVal = AvailOut(n);
    AvailIn(n) = ∩_{p∈pred(n)} AvailOut(p)
    AvailOut(n) = (AvailIn(n) \ kill(n)) ∪
                  gen(n);
    if(AvailOut != oldVal){
        workList = workList ∪ succ(n);
    }
}
```

Analysis Types

- Both reaching definitions and expression availability are calculated on the CFG in the direction of program execution.
  - They are *forward* analyses.
  - Other analyses backtrack from exit to entrance (backwards analyses).

Analysis Types

- Definitions can reach across *any path*.
  - The in-flow equation uses a union.
  - This is a *forward*, *any-path* analysis.
- Expressions must be available on *all paths*.
  - The in-flow equation uses an intersection.
  - This is a *forward, all-paths* analysis.

Forward, All-Paths Analyses

- Encode properties as tokens that are generated when they become true, then killed when they become false.
  - The tokens are “used” when evaluated.
- Can evaluate properties of the form:
  - “G occurs on all execution paths leading to U, and there is no intervening occurrence of K between G and U.”
  - Variable initialization check:
    - G = variable-is-initialized, U = variable-is-used
    - K = *variable-is-uninitialized* (kill set is empty)

Backward Analysis - Live Variables

- Tokens can flow backwards as well.
- Backward analyses are used to examine what happens *after* an event of interest.
- “Live Variables” - analysis to determine whether the value held in a variable may be used.
  - A variable may be considered live if there is any possible execution path where it is used.

Live Variables

- A variable is live if its current value may be used before it is changed.
- Can be expressed as flow equations.
  - $\text{LiveIn}(n) = \bigcup_{p \in \text{succ}(n)} \text{LiveOut}(p)$
    - Calculated on successors, not predecessors.
  - $\text{LiveOut}(n) = (\text{LiveIn}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$
- Worklist algorithm can still be used, just using successors instead of predecessors.

Backwards, Any-Paths Analyses

- General pattern for backwards, any-path:
  - “After D occurs, there is at least one execution path on which G occurs with no intervening occurrence of K.”
    - D indicates a property of interest. G is when it becomes true. K is when it becomes false.
    - Useless definition check, D = variable-is-assigned, G = variable-is-used, K = variable-is-reassigned.

Backwards, All-Paths Analyses

- Check for a property that must inevitably become true.
- General pattern for backwards, all-path:
  - “After D occurs, G always occurs with no intervening occurrence of K.”
  - Informally, “D inevitably leads to G before K”
    - D indicates a property of interest. G is when it becomes true. K is when it becomes false.
    - Ensure interrupts are reenabled, files are closed, etc.

Analysis Classifications

	Any-Paths	**All-Paths**
Forward (pred)	**Reach** *U* may be preceded by G without an intervening *K*	**Avail** *U* is always preceded by G without an intervening *K*
Backward (succ)	**Live** *D* may lead to *G* before *K*	**Inevitability** *D* always leads to *G* before *K*

Crafting Our Own Analysis

- We can derive a flow analysis from run-time analysis of a program.
- The same data flow algorithms can be used.
  - Gen set is “facts that become true at that point”
  - Kill set is “facts that are no longer true at that point”
  - Flow equations describe propagation

Monotonicity Argument

- **Constraint**: The outputs computed by the flow equations must be monotonic functions of their inputs.
- When we recompute the set of “facts”:
  - The gen set can only get larger or stay the same.
  - The kill set can only grow smaller or stay the same.

Let’s Take a Break

“Final” Testing Stages

- All concerned with behavior of the system as a whole, but for different purposes.
- Acceptance Testing
  - Validation against the user’s expectations.
- Regression Testing
  - Ensuring that the system continues to work as expected when it evolves.

Verification and Validation

Activities that must be performed to consider the software “done.”

- **Verification:** The process of proving that the software conforms to its specified functional and non-functional requirements.
- **Validation:** The process of proving that the software meets the customer’s true requirements, needs, and expectations.

System and Acceptance Testing

- System Integration Testing
  - Checks system against specification.
  - Performed by developers and professional testers.
  - Verifies correctness and completion of the product.
- Acceptance Testing
  - Checks system against user needs.
  - Performed by customers, with developer supervision
  - Validates usefulness and satisfaction with the product.

Regression Testing

- Systems continue to evolve post-release.
  - Patches to newly-discovered faults.
  - New features.
  - Adaptations to new hardware/software (OS).
- Rechecks test cases passed by previous production systems.
- Guards against unintended changes.

Acceptance Testing

Acceptance Testing

Once the system is internally tested, it should be placed in the hands of users for feedback.

- Users must ultimately approve the system.
- Many faults emerge when used in the wild.
  - Alternative operating environments.
  - More eyes on the system.
  - Wide variety of usage types.
- Acceptance testing allows users to try the system under controlled conditions.

User-Based Testing Types

- Alpha Testing
  - A small group of users work closely with development team to test the software.
- Beta Testing
  - A release of the software is made available to a larger group of interested users.
- Acceptance Testing
  - Customers decide whether or not the system is ready to be released.

Alpha Testing

- Users and developers work together.
  - Users can identify problems not apparent to the development team.
    - Developers work from requirements, users have their own expectations.
- Takes place under controlled conditions.
  - Software is usually incomplete or untested.
- “Power users” and customers who want early information about system features.

Beta Testing

- Early build made available to a larger group of volunteers and customers.
- Software is used under uncontrolled conditions, hardware configurations.
  - Important if the system will be sold to any customer.
  - Discovers interaction problems.
- Can be a form of marketing.
- Should not replace traditional testing.

Acceptance Testing

- Validation activity between developer and customer.
- Software is taken to a group of users that try a set of scenarios under supervision.
  - Scenarios mirror the typical system use cases.
  - Users provide feedback and decide whether the software is acceptable for each scenario.
- Users ultimately decide whether the software is ready for release.
  - Developers may negotiate with users.

Acceptance Testing Stages

- Define acceptance criteria
  - Work with customers to define how validation will be conducted, and the conditions that will determine acceptance.
- Plan acceptance testing
  - Decide resources, time, and budget for acceptance testing. Establish a schedule. Define order that features should be tested. Define risks to testing process.

Acceptance Testing Stages

- Derive acceptance tests.
  - Design tests to check whether or not the system is acceptable. Test both functional and non-functional characteristics of the system.
- Run acceptance tests
  - Users complete the set of tests. Should take place in the same environment that they will use the software. Some training may be required.

Acceptance Testing Stages

- Negotiate test results
  - It is unlikely that all of the tests will pass the first time. Developer and customer negotiate to decide if the system is good enough or if it needs more work.
- Reject or accept the system
  - Developers and customer must meet to decide whether the system is ready to be released.

Qualitative Process

- Results may vary based on the user surveyed and environmental factors.
  - Software may need to be accepted regardless of users' preferences if deadline is strict.
  - May be used as an "excuse" to reject a project.
- Users should be "typical"
  - Usually interested volunteers.
  - How users interact with beta may not match real system.
  - May not catch faults that normal users will see.

Usability

- A **usable** product is quickly learned, allows users to work efficiently, and can be used without frustration.
  - Must be evaluated through user-based testing.
  - Objective criteria:
    - Time and number of operations to perform tasks.
    - Frequency of user error.
  - Subjective criteria:
    - Satisfaction of users.
  - Can be evaluated throughout lifecycle.

Usability Testing Steps

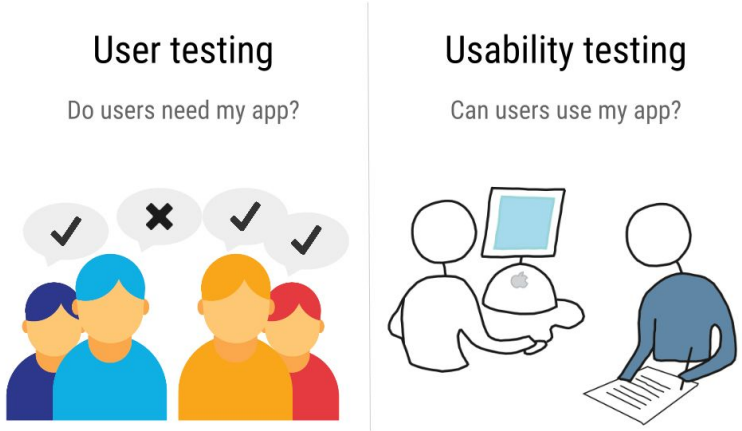
- Inspecting specifications:
- Testing early prototypes:
  - Bring in end users to:
    - Explore mental models (exploratory testing)
    - Evaluate alternatives (comparison testing)
    - Validate usability.
  - May involve mockup GUIs, not working software.
- System and Acceptance Testing:
  - Evaluate incremental builds, compare against competitors, check against compatibility guidelines.

Exploratory Testing

- Explore the mental model of end users.
  - Early in design stage, ask users how they would like to interact with the system.
- Look for common answers from users.
  - If conflicts, try to combine elements of answers.
  - Larger sample sizes will yield better results.
  - Consider all groups of stakeholders.
    - Some stakeholders will have different usage patterns from others.

Validation Testing

- Used to assess overall usability.
  - Identifies difficulties and obstacles encountered while using the system.
  - Measures error rate, clicks/time to perform a task.



Validation Testing

- Preparation phase:
  - Define objectives for the session, identify items to be tested, select population, plan actions.
- Execution phase:
  - Users monitored as they execute planned actions.
- Analysis phase:
  - Results evaluated, software changes planned.

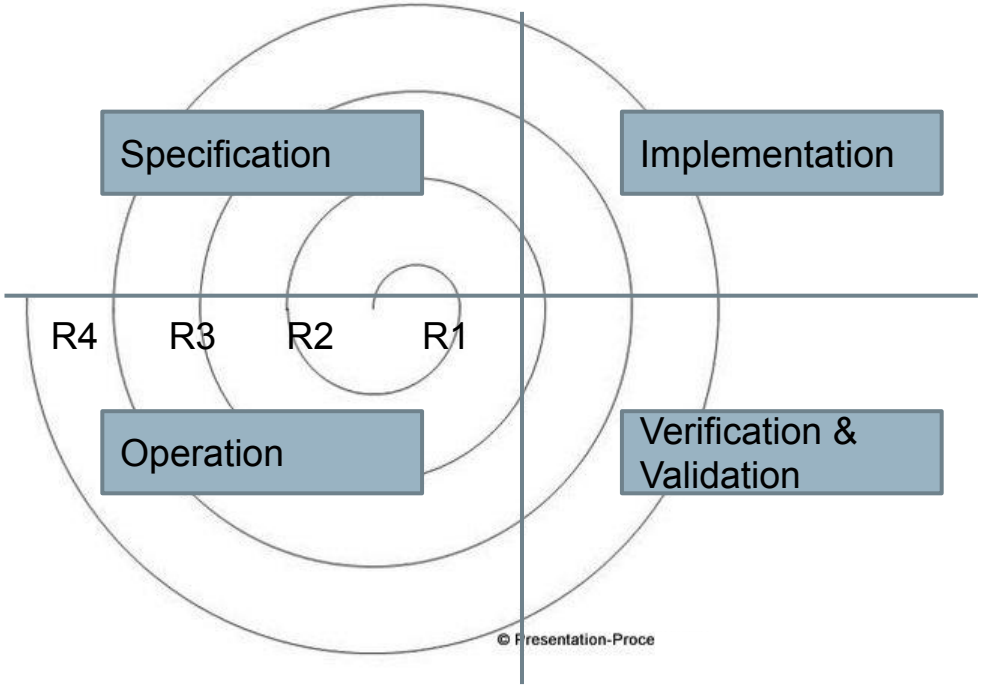
Validation Testing

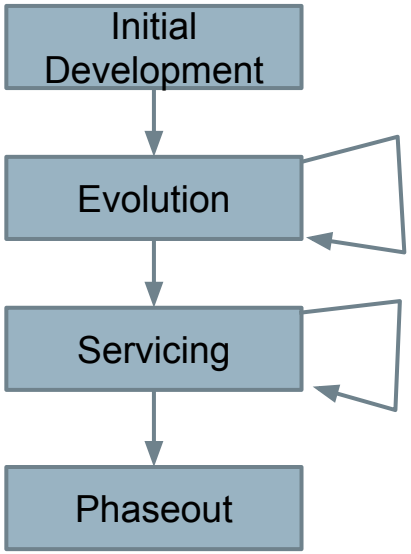
- Activities should be based on typical use cases of expected features.
  - Goal is to ensure “normal use” is optimal.
- Users should perform tasks independently.
  - Actions are recorded through tracking software.
  - Comments and impressions are collected with post-activity questionnaires.
- Consider accessibility needs.
  - Font size, color choices, audio guidance.

Let’s Take a Break

Regression Testing

Software Lifecycle





Software Maintenance

- Fault Repairs
  - Changes made in order to correct coding, design, or requirements errors.
- Environmental Adaptations
  - Changes made to accommodate changes to the hardware, OS platform, or external systems.
- Functionality Addition
  - New features are added to the system to meet new user requirements.

Maintenance is Hard

It is harder to maintain than to write new code.

- Must understand code written by another developer, or code that you wrote long ago.
- Creates a “house of cards” effect.
- Developers tend to prioritize new development.

New code must be tested. Existing code must also be *retested*.

System Regression

- System evolution may change existing functionality in unforeseen ways.
- When a new version no longer works as expected, it *regresses* with respect to tested functionality.
  - A basic quality requirement is that new versions are *non-regressive* - if we tested it and it works, it should continue to work.
- Regression testing used to detect regressive code.

Regression Testing

- Basic idea: when changes have been made, re-execute tests used to verify the original code.
- Not as simple as it sounds:
  - When do you execute regression tests?
    - On check-in? Before patch is publicly released?
  - Can you afford to execute all tests?
    - The number will grow as the system expands.
  - Can you actually execute all tests?
    - Do you need to?
    - Are some tests obsolete?

Test Case Maintenance

- Test suites must be maintained over time.
- Obsolete tests should be removed.
  - Tests involving requirements, features, classes, or interfaces that no longer exist or have been modified.
- Redundant tests should be identified.
  - Tests that cover the same structural elements, input partitions, other test goals.
  - May be introduced to test changed code, or by concurrently-working testers.
  - Can still be executed, but may not be needed.

Regression Test Selection

- The number of tests to reexecute may be very large (and grows over time).
- Not all tests *need* to be re-executed.
  - Regression testing costs can be reduced by *prioritizing* the set of test cases.
  - Select a subset of tests relevant to the changes.
    - Techniques based on code and specifications.
  - Choose a cut-off based on testing budget.

Code-Based Test Selection

- Select a test case for execution if it exercises a portion of the code modified.
- Control-based selection:
  - Maintain a record of CFG nodes executed by each test.
  - Compare the structure of the old and new versions.
  - Tests that exercise added, modified, or deleted elements are prioritized.
  - Can be based on control or data flow.

Example

Version 1:

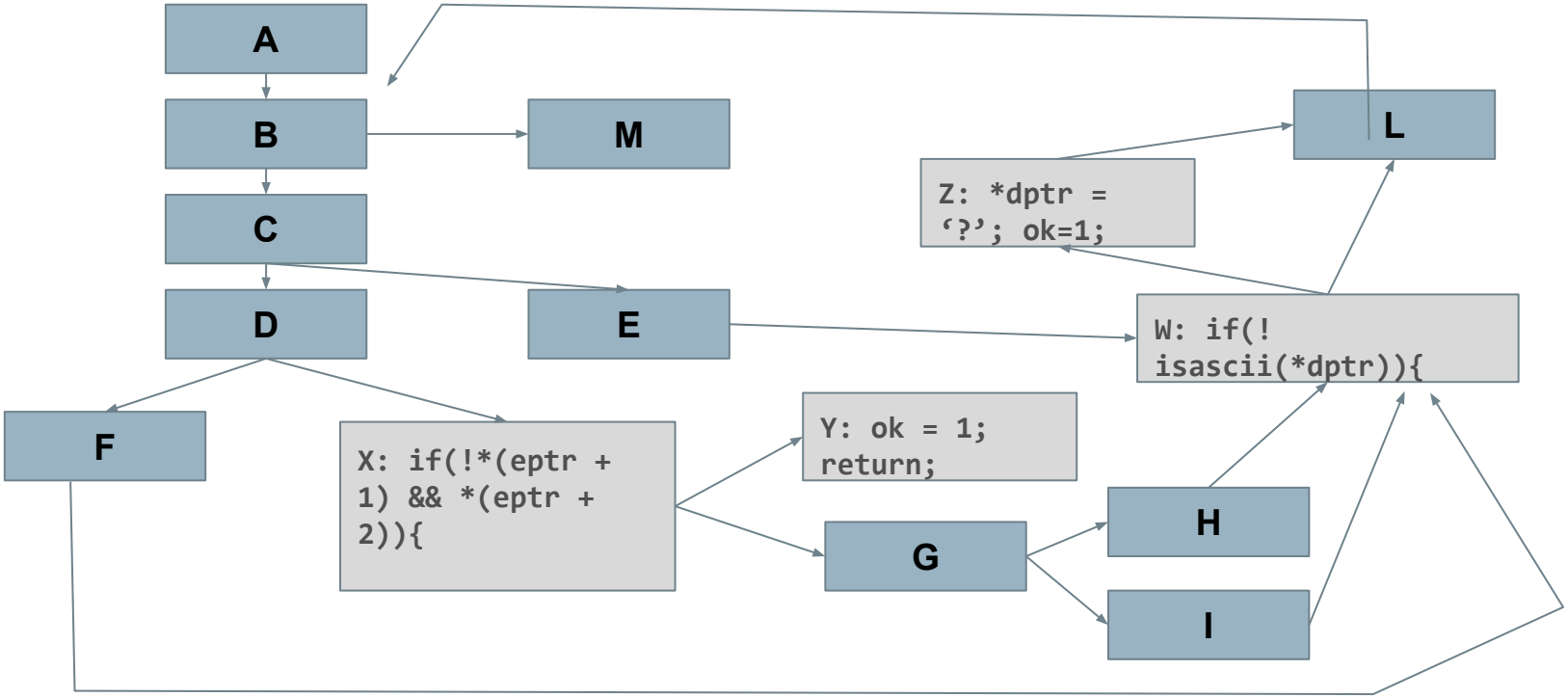
```
} else if (c == ‘%’){
     int digit_high = ..
}
 …

++dptr;
++eptr;
}
```

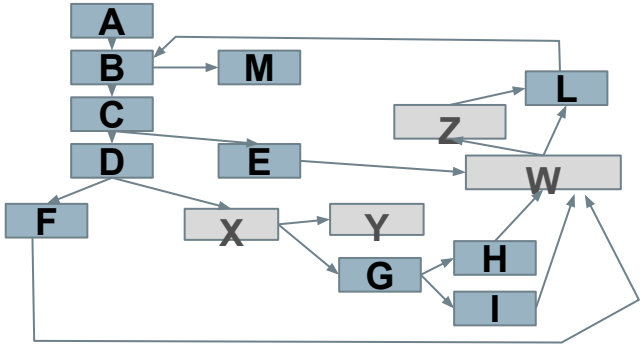
Version 2:

```
} else if (c == ‘%’){
     if(!*(eptr + 1) && *(eptr + 2)){
          ok = 1; return;
     }
     int digit_high = ..
}
 …
if(! isascii(*dptr)){
          *dptr = ‘?’; ok=1;
}
++dptr;
++eptr;
}
```

Example


A
B
M
C
D
E
F
X: if(!*(eptr + 1) && *(eptr + 2)){
Y: ok = 1; return;
G
H
I
Z: *dptr = '?'; ok=1;
L
W: if(! isascii(*dptr)){

Example



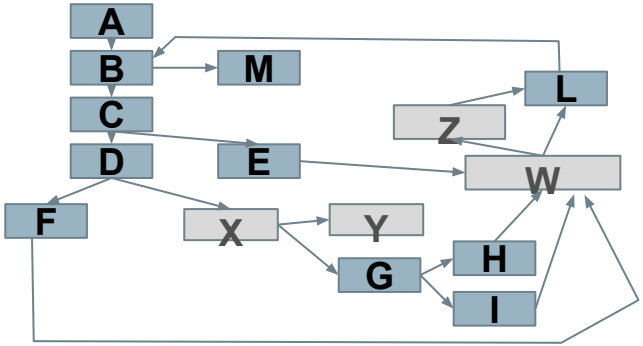
Corrective Changes Only: Ignores new features, and only considers corrective patches.

ID	Input	Path
1	" "	A B M
2	"test+case%1Dadequacy"	A B C D F L … B M
3	"adequate+test%0Dexecution%7U"	A B C D F L … B M
4	"%3D"	A B C D G H L B M
5	"%A"	A B C D G I L B M
6	"a+b"	A B C D F L B C E L B C D F L B M
7	"test"	A B C D F L B C D F L B C D F L B M
8	"+%0D+%4J"	A B C E L B C D G I L … B M
9	"first+test%9Ktest%K9"	A B C D F L … B M

Data-Based Test Selection

- New code can introduce new DU pairs and remove existing pairs.
- Re-execute test cases that execute DU pairs in the original program that were deleted or modified in the revised program.
  - Also select test cases that execute a conditional statements modified in the revision.
    - Changed predicates can affect DU paths.

Example



Variable	Definitions	Uses
*eptr		**X**
eptr		**X**
*dptr	**Z**	**W**
dptr		**Z, W**
ok	**Y, Z**	

Selective Execution

- When a regression suite is too large, we must reduce the number of tests executed.
- Techniques predict “usefulness” of tests:
  - Elements covered, history of effectiveness.
- High priority tests selected more than low priority.
  - Eventually, all tests will be selected.
  - However, at varying frequencies.
  - Efficient rotation in which the cases most likely to reveal faults will be selected more often.

Selective Execution Schema

- Execution History Schema:
  - Simple strategy.
  - Recently executed tests are given low priority.
  - Cases not recently executed are given high priority.
  - Often used along with correlation to changed elements.
- Fault-Revealing Priority Schema:
  - Test cases that recently revealed faults are prioritized.
  - Faults are not evenly distributed, but tend to cluster around particular functionality/units in the code.
  - Not all faults may have been fixed.

Selective Execution Schema

- Structural Priority Schema:
  - Weight tests by the number of elements covered.
    - Statements, branches, conditions, etc.
  - Weight each element by when it was last executed.
  - Prioritize tests that cover a large number of elements that have not recently been executed.
  - Ensures that all structural elements are eventually recovered, especially if they have not recently been tested.

We Have Learned

- Control-flow and data-flow both capture important paths in program execution.
- Analysis of how variables are defined and then used and the dependencies between definitions and usages can help us reveal important faults.
- Many forms of analysis can be performed using data flow information.

We Have Learned

- Analyses can be *backwards* or *forwards.*
  - … and require properties be true on *all-paths* or *any-path*.
  - Reachability is forwards, any-path.
  - Expression availability is forwards, all-paths.
  - Live variables are backwards, any-path.
  - Inevitability is backwards, all-paths.
- Many analyses can be expressed in this framework.

We Have Learned

- Late-stage testing techniques are concerned with behavior of the system as a whole, but for different purposes.
- Acceptance Testing
  - Validation against the user's expectations.
- Regression Testing
  - Ensuring that the system continues to work as expected when it evolves.

Next Time

- Exercise Session - Structural Testing
  - Bring laptops, download Meeting Planner code.
- Integration Testing and Testing of OO Systems
  - Reading: Pezze and Young, Chapters 15, 21, 22.2
- Assignment 2
  - Due March 1!

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY