Impact Analysis - Dependency Analysis and Ripple Effect

Isaac Griffith

CS 4423 and CS 5523
Department of Computer Science
Idaho State University

Outcomes

After today's lecture you will:

- Have an understanding of and be to conduct a Dependency Analysis using both
  - Call graph based techniques
  - Dependency graph based techniques
- Be able to extract either a static or dynamic slice from a program dependency graph
- Have an understanding of and be able to evaluate the ripple effect of a change

Dependency-Based Impact Analysis

CS 4423/5523

ROAR

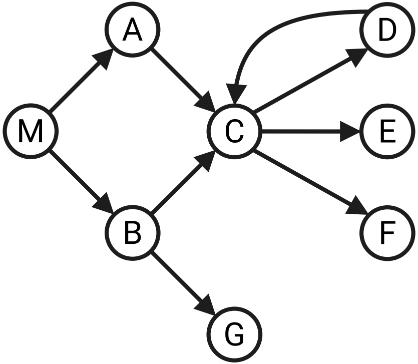
Dependency Analysis

- Source code objects are analyzed to obtain vertical traceability information.
- Dependency based impact analysis techniques identify the impact of changes by analyzing syntactic dependencies
  - Syntactic dependencies are likely to cause semantic dependencies.
- Two traditional impact analysis techniques:
  1. Call graph techniques
  2. Dependency graph techniques

Call Graph

- **Call graph** a directed graph, where:
  - **Nodes:** represents a function, a component, or a method.
  - **Edges:** between two nodes, A and B, indicates that A may invoke B

- Call graphs are used to understand the potential impacts that a software change may have
- Let:
  - P be a program
  - G be the call graph obtained from P
  - p be some procedure in P
- **Assumption:** some change in p has the potential to impact changes in all nodes reachable from p in G

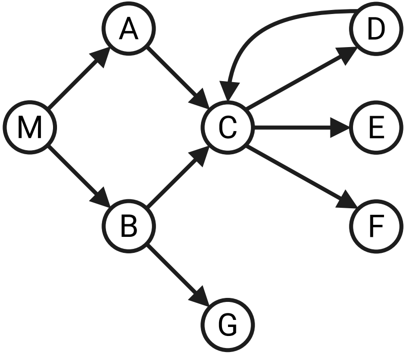


Call Graph

Call graph techniques have several disadvantages:

- Can produce an imprecise impact set
  - **Ex:** We cannot determine the conditions that cause impacts to propagate from M to other procedures
- Does not capture propagation due to procedure returns
  - **Ex:** Suppose that, `E` is modified and control returns to `C`.

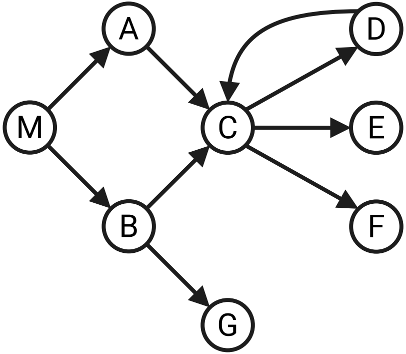
    When `C` returns, we cannot infer whether impacts of changing `E` propagates into none, both, `A`, or `B`.



Call Graph

- Consider the execution trace: `M B r A C D r E r r r r x`
  - Where `r` and `x` represent function returns and program exits

- The impact of the modification of `M` is computed by forward searching in the trace to find:
  - procedures indirectly or directly invoked by `E`; and
  - procedures invoked after `E` terminates
- We can identify procedures `E` returns into using a backward search in the trace
- In this trace, `E` does not invoke other entities, but returns into `M`, `A`, and `C`
- A modification in `E` has a potentially impact set: `{M, A, C, E}`



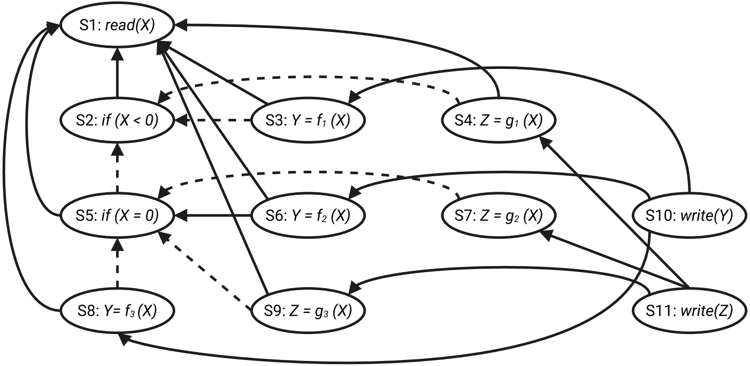
Program Dependency Graph

- **Program dependency graph (PDG)** a directed graph:
  - Two types of nodes:
    1. Simple statement
    2. Predicate expression
  - Two types of edges in a PDG
    1. Data dependencies
    2. Control dependencies
- Let v_i and v_j be two nodes in a PDG.
  - A data dependency edge from v_i to v_j indicates that
    - the computations performed at v_i directly depend upon the results of computations performed at v_j.
  - A control dependency edge from v_i to v_j indicates that
    - v_i may execute based on the result of evaluation of a condition at v_j.

Program Dependency Graph

- Solid Edges -> Data dependencies
- Dashed Edges -> Control dependencies

```
     begin
S1:      read(X)
S2:      if(X < 0)
         then
S3:          Y = f1(x);
S4:          Z = g1(x);
         else
S5:          if (X = 0)
             then
S6:              Y = f2(X);
S7:              Z = g2(X);
             else
S8:              Y = f3(X);
S9:              Z = g3(X);
             endif;
         endif;
S10:     write(Y);
S11:     write(Z);
     end
```

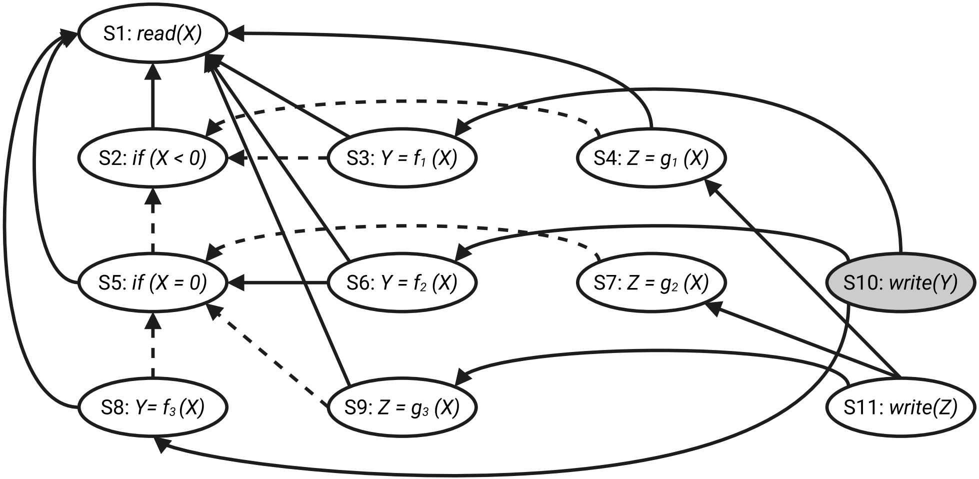


Static Slice

- A static program slice is extract from a PDG, as follows:
  - for a variable var at node n, identify all reaching definitions of var.
  - find all nodes in the PDG which are reachable from those nodes.
  - The visited nodes in the traversal process constitute the desired slice.

Static Slice

- Consider variable Y at S10

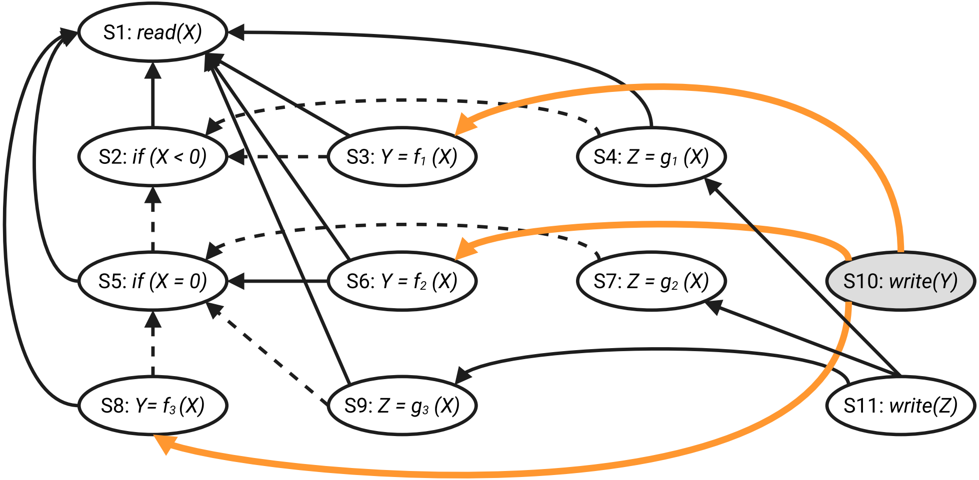


Select var Y in node S10

Idaho State University Computer Science

Static Slice

- Consider variable Y at S10

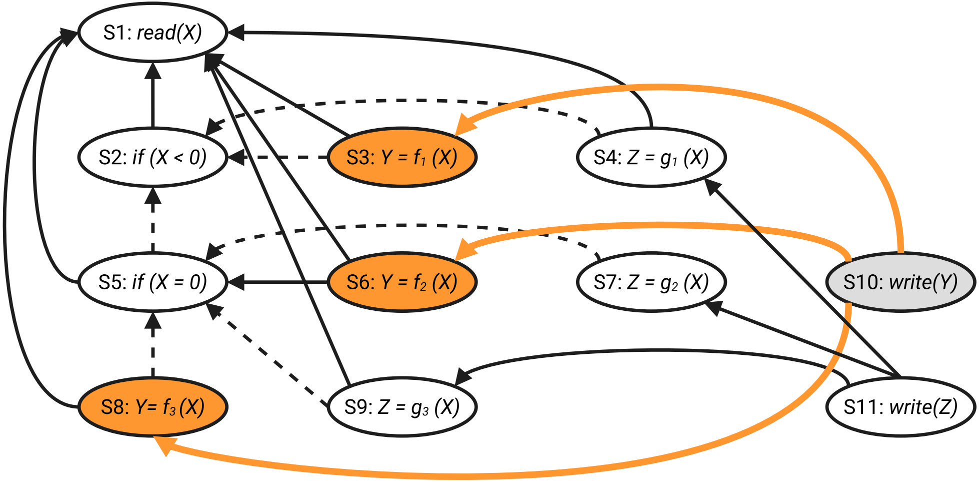


Find all reachable Nodes

ROAR

Static Slice

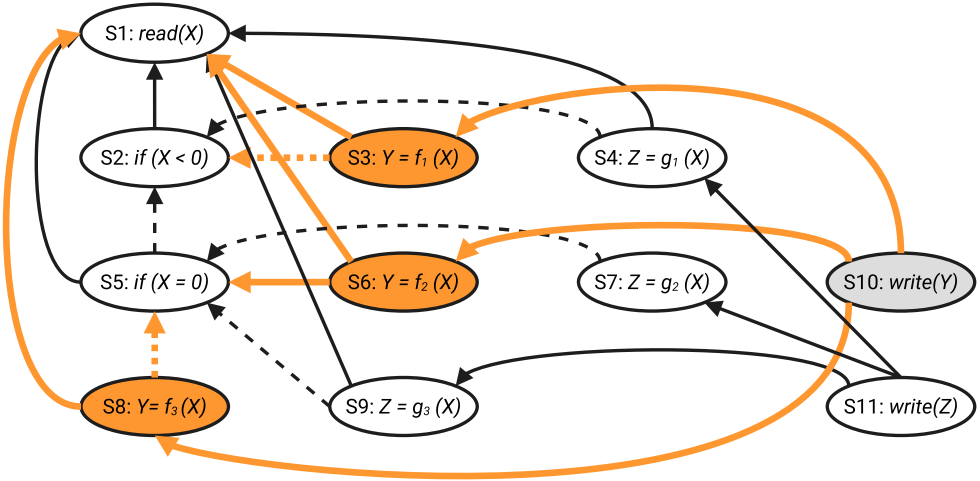
- Consider variable Y at S10



Initial Impact Set: {S3, S6, S8}

Static Slice

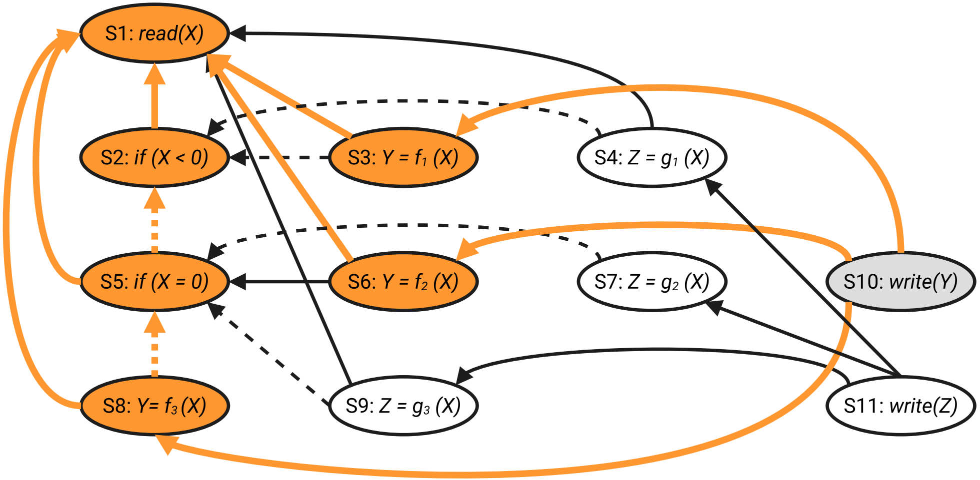
- Consider variable Y at S10



Find all reachable Nodes

Static Slice

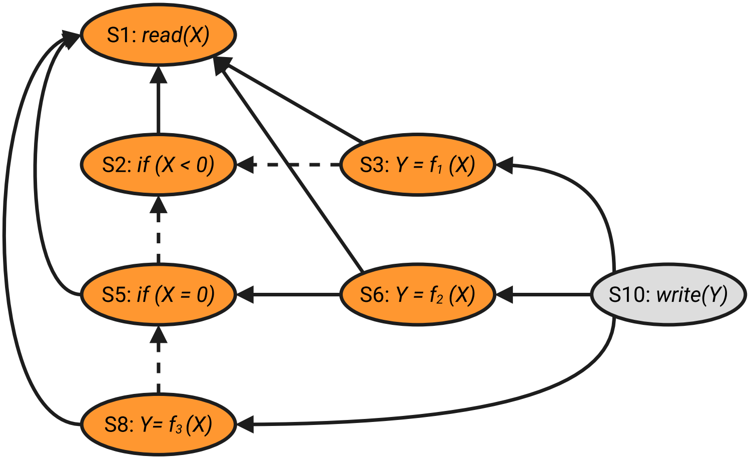
- Consider variable Y at S10



Final Impact Set: {S1, S2, S3, S5, S6, S8}

Static Slice

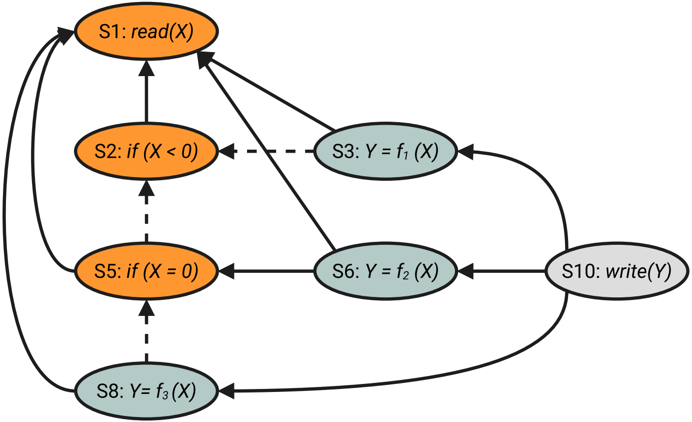
- Consider variable Y at S10



The Static Slice of PDG for Y at S10

Dynamic Slice

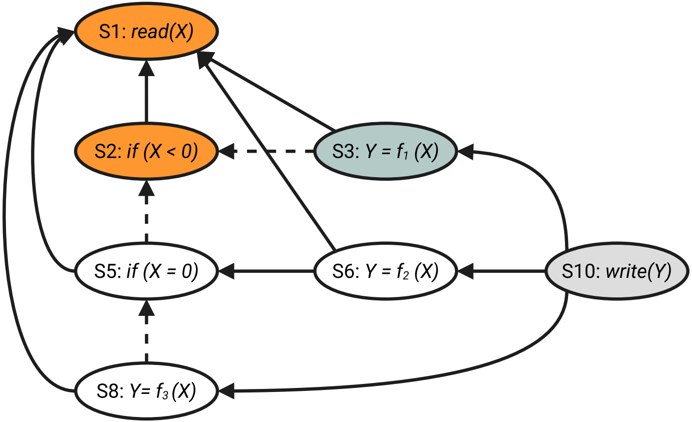
- A dynamic slice is typically more useful in localizing defects than static slices



Only one of S3, S6, or S8 may be executed for any value of X

Dynamic Slice

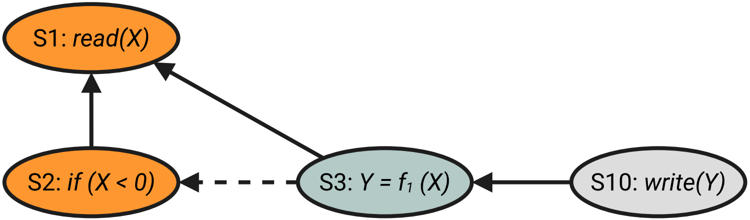
- A dynamic slice is typically more useful in localizing defects than static slices



If the input value for X is -1, only S3 is executed.

Dynamic Slice

- A dynamic slice is typically more useful in localizing defects than static slices



The final slice contains only S1, S2, and S3

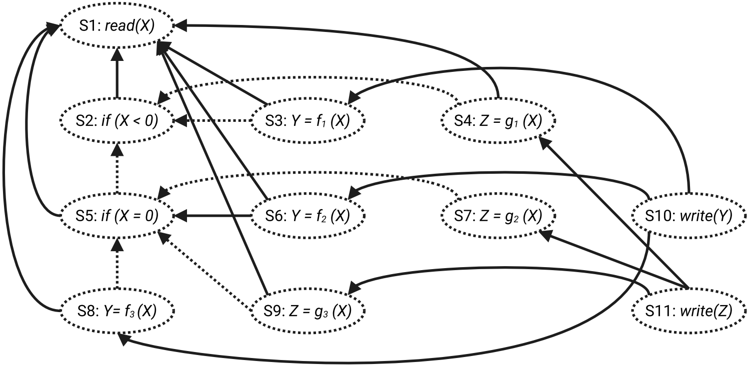
- For −1 as the values of X, if the value of Y is incorrect at S10, one can infer that either f_i is erroneous at S3 or the “if” condition at S2 is incorrect.

Dynamic Slicing

- Obtaining a dynamic slice
  - Inputs: a test and a PDG
  - We represent the execution history, of the program as a sequence of vertices $\langle v1, v2, ..., vn \rangle$.
  - **Execution history** ($hist$): of a program P for a test case, $test$, and a variable var is the set of all statements in $hist$ whose execution had some effect on the value of var as observed at the end of the execution.
  - Then:
    1. for the current test, mark the executed nodes in the PDG.
    2. traverse the marked nodes in the graph.

Example

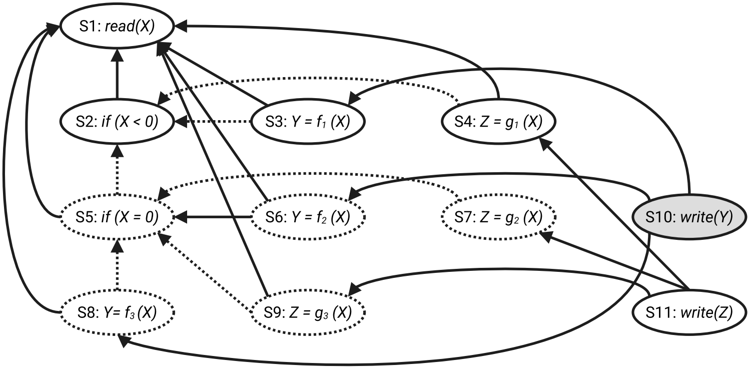
- **Goal:** Dynamic Slice for `Y` at end of execution
- **Input:** `X = -1`
- **History:** $\langle S1, S2, S3, S4, S10, S11 \rangle$



Initially, nodes have dashed lines
If executed, the node is made solid

Example

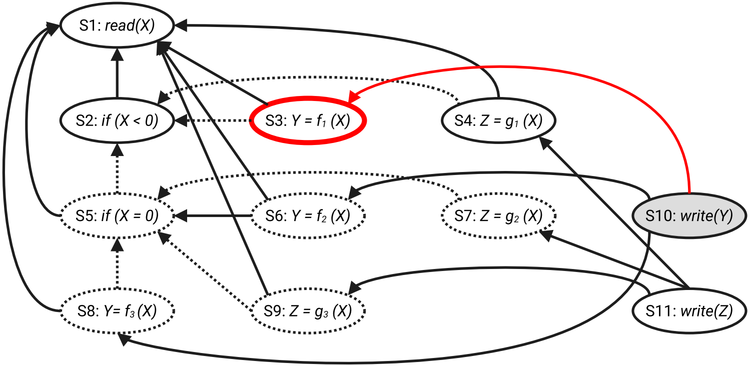
- **Goal:** Dynamic Slice for `Y` at end of execution
- **Input:** `X = -1`
- **History:** $\langle S1, S2, S3, S4, S10, S11 \rangle$



The executed nodes are made solid

Example

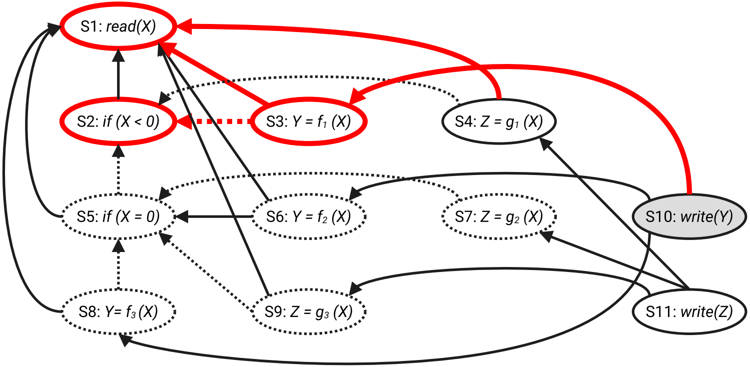
- **Goal:** Dynamic Slice for `Y` at end of execution
- **Input:** `X = -1`
- **History:** $\langle S1, S2, S3, S4, S10, S11 \rangle$



Start at S3, as this is where Y is defined

Example

- **Goal:** Dynamic Slice for `Y` at end of execution
- **Input:** `X = -1`
- **History:** $\langle S1, S2, S3, S4, S10, S11 \rangle$



Final slice containing S1, S2, S3

Ripple Effect

CS 4423/5523

ROAR

Ripple Effect

- Brief History
  - Early 1970s: Haney introduced the concept of ripple effect
  - 1978: Yau, Collofello and McGregor define ripple effect from the functional and performance perspectives
    - Viewed ripple effect as a complexity measure to compare changes to source code.
    - Computed by means of error flow analysis.

Error Flow Analysis

- In this analysis, program variables involved in a change are considered to be potential sources of errors
- Inconsistency can propagate from those sources to other variables in the program.
- The other sources of errors are successively identified until error propagation is no more possible.
- This work was extended to include stability measure
  - Stability reflects the resistance to the potential ripple effect which a program would have when it is changed.
  - Stability analysis and impact analysis differ as follows:
    - stability analysis considers the total potential ripple effects rather than a specific ripple effect caused by a change.

Design Stability

- **Design stability** - Yau and Collofello
  - Developed an algorithm to compute stability based on design documentation.
  - One counts the number of assumptions made about shared global data structures and module interfaces.
- Difference between design level stability and code level stability:
  - Design level stability does not consider change propagation within modules.

Computing Ripple Effect

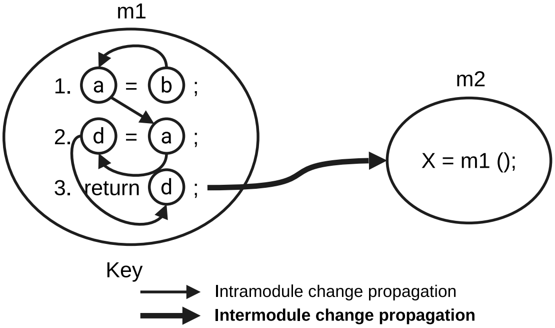
- The basic idea is to identify the impact of a change to one variable on the program.
- We will consider two types of change propagation
  - Intramodule change propagation
  - Intermodule change propagation

Intramodule Change Propagation

Intramodule change propagation: the propagation of changes from one source code line in a module to another source code line within the same module

Example:

- Consider module m1 containing the three lines of code referring to variables `a`, `b`, `d`.
  - A change in the value of `b` will impact `a` in line (1)
    - The change will propagate to `a` in line (2).
  - Variable `a` affects variable `d` in line (2)
    - This will propagate to variable `d` in line (3).

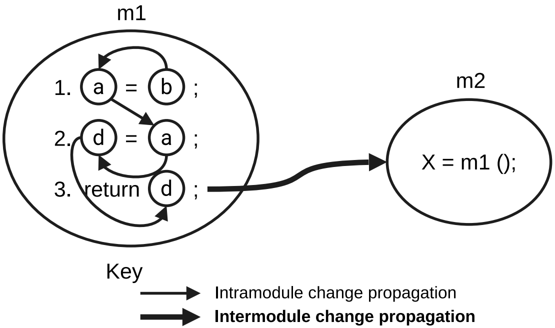


Intermodule Change Propagation

- **Intermodule change propagation:** Propagation of values of variables in one module to variables in a different module
  - Intermodule change propagation of values of a variable w occurs in the following ways:
    - If `w` is a global variable, then a change made to `w` by one module is seen by another module accessing `w`.
    - If `w` is an input parameter in a call to a second module, then values of w are propagated from the caller to the callee.
    - If `w` is an output parameter, then its value propagates from the module that makes an output to the module that accepts the output.

Example

- Variable `d` propagates to any module that calls `m1`, because `d` appears in the return statement.
- If variable `a` is global, its appearance on the left-hand-side of an assignment statement causes its value to be propagated to any module that uses variable `a`.



Computing Ripple Effect

- A matrix V_m is used to represent the initial starting points for intramodule change propagation.
- The matrix records the following five conditions of the module's variable x for all x:
  - x is defined in an assignment statement.
  - x is assigned a value in a read input statement.
  - x is an input to an invoked module.
  - x is an output from an invoked module.
  - x is a global variable.
- In V_m, variable definitions are uniquely identified.
- Variable occurrences are provide a value in the matrix
  - occurrence meets one of the above conditions -> value = 1
  - otherwise -> value = 0
- x_i^d means that the variable x has been **defined** at line (i).
- x_i^u means that the variable x has been **used** at line (i).

Computing Ripple Effect

- In module `m1`, variable `a` is global and it is considered to be defined. Matrix V_{m1} for the lines of code in `m1` is expressed as:

$$V_{m1} = \begin{array}{c} \begin{array}{ccccc} a_1^d & d_1^u & d_2^d & a_2^u & d_3^u \end{array} \\ \begin{pmatrix} 1 & 0 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

- A zero-one (0–1) matrix Z_m indicates values of what variables propagate to other variables in the same module.
- Individual occurrences of variables are denoted by rows and columns of Z_m.
- The source code of module m1 results in the following matrix:

$$Z_{m1} = \begin{array}{c|c} & \begin{array}{ccccc} a_1^d & d_1^u & d_2^d & a_2^u & d_3^u \end{array} \\ \begin{array}{c} a_1^d \\ d_1^u \\ d_2^d \\ a_2^u \\ d_3^u \end{array} & \begin{pmatrix} 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

- It is easy to observe that Z_{m1} is both reflexive and transitive. The reflexive property implies that every variable propagates to itself. whereas transitivity means that if v_1 propagates to v_2 and v_2 propagates to v_3 then v_1 also propagates to v_3.

Idaho State University Computer Science

Computing Ripple Effect

- Suppose that module m_1 is called by m_2, a is a global variable, and m_2 and m_3 use a.
- If values of the variable corresponding to row i propagate to the module corresponding to column j, then the (i, j)th entry of the zero-one matrix is set to 1.
- For all the variables of a module m1, propagation of their values to other modules is captured by an X matrix, denoted by X_{m1} as follows:

$$X_{m1} = \begin{array}{c} \\ a_1^d \\ d_1^u \\ d_2^d \\ a_2^u \\ d_3^u \end{array} \begin{array}{c} \begin{array}{ccc} m_1 & m_2 & m_3 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{array}$$

The intermodule change propagation for variables occurring in m_1 is obtained by means of the Boolean product of the two matrices Z_{m1} and X_{m1}, as follows:

$$Z_{m1}X_{m1} = \begin{pmatrix} 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

In the Boolean product $Z_{m1}X_{m1}$, the “1” in row 2, column 3 indicates change propagation from b_1^u to m_3; similarly, the “0” in row 3, column 2 indicates no change propagation from d_2^d to m_2.

ROAR

Computing Ripple Effect

The Boolean product of V_{m1} and $Z_{m1}X_{m1}$ indicates the variable definitions that propagate from m_1 to other modules:

$$V_{m1}Z_{m1}X_{m1} = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 3 \end{pmatrix}$$

Now, $V_{m1}Z_{m1}X_{m1}$ indicates that there are no change propagations to m_1, one change propagation to m_2, and three change propagations to m_3.

- Concerning the complexity of making changes, the more complex a module is, the more resources are needed to change the module.
- Therefore, a measure of complexity can be factored into the calculation of change propagation to obtain a measure of the complexity of modifying the definitions of variables.
- The well-known McCabe's cyclomatic complexity can be integrated with the ongoing computation of change propagation.

Computing Ripple Effect

- A C matrix of dimensions $1 \times n$ is chosen to represent McCabe's cyclomatic complexity, where n is the number of modules:

$$C = \begin{array}{c} m_1 \\ m_2 \\ m_3 \end{array} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

Because the complete codes for m_1, m_2, and m_3 have not been given, we assume their arbitrary complexity values for example purpose. The product of $V_{m1} Z_{m1} X_{m1}$ and C is:

$$V_{m1} Z_{m1} X_{m1} C = \begin{pmatrix} 0 & 1 & 3 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 4 \end{pmatrix}$$

The complexity-weighted total propagation of variable definitions for m_1 is represented by $V_{m1} Z_{m1} X_{m1} C$. The quantity $V_{m1} Z_{m1} X_{m1} C / |V_{m1}|$, where $|V_{m1}|$ represents the total member of variable definitions in m_1, represents the mean complexity-weighted propagation of variable definitions in m_1. In the aforementioned example, $|V_{m1}| = 3$, and it means that ripple in module m_1 is caused by three sources. For module m_1, the mean complexity-weighted propagation of variable is $4/3 = 1.33$. The general expression for calculating the ripple effect for a program (REP) is as follows:

$$REP = \frac{1}{n} \sum_{m=1}^{n} \frac{V_m \cdot Z_m \cdot X_m \cdot C}{|V_m|}$$

where m = module and n = number of modules

Computing Ripple Effect

- Sue Black examined some links between ripple effect and Lehman's Laws

Lehman's Laws	Relevance to ripple effect
I Continuing change	Compare versions of program Highlight complex modules Measure stability over time Highlight areas ripe for restructuring/refactoring
II Increasing complexity	Determine which module needs maintenance Measure growing complexity
III Self regulation	Helps measure rate of change of system Helps look at patterns/trends of behavior Determine the state of the system

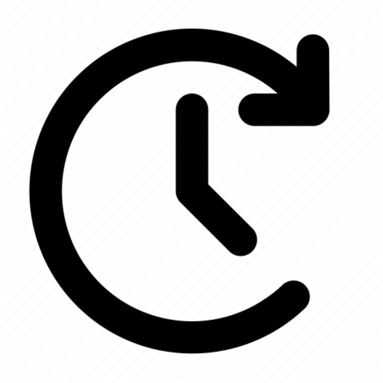
Computing Ripple Effect

- Sue Black examined some links between ripple effect and Lehman's Laws

Lehman's Laws	**Relevance to ripple effect**
IV Conservation of organizational stability	Not relevant
V Conservation of familiarity	Provide system change data
VI Continuing growth	Measure impact of new modules on a system Help determine which modules in use in a new version
VII Declining quality	Highlight areas of increasing complexity Determine which modules need maintenance Measure stability over time
VIII Feedback system	Provide feedback on stability/complexity of system

For Next Time

- Review EVO Chapter 6.3
- Read EVO Chapter 6.4 - 6.6 & 7.1 - 7.3
- Watch Lecture 16



Are there any questions?