Faculty of Computers and Information
BSc. Software Engineering Program

Dr. Lamia Abo Zaid

د. لمياء أبوزيد

Software Evolution : TOC

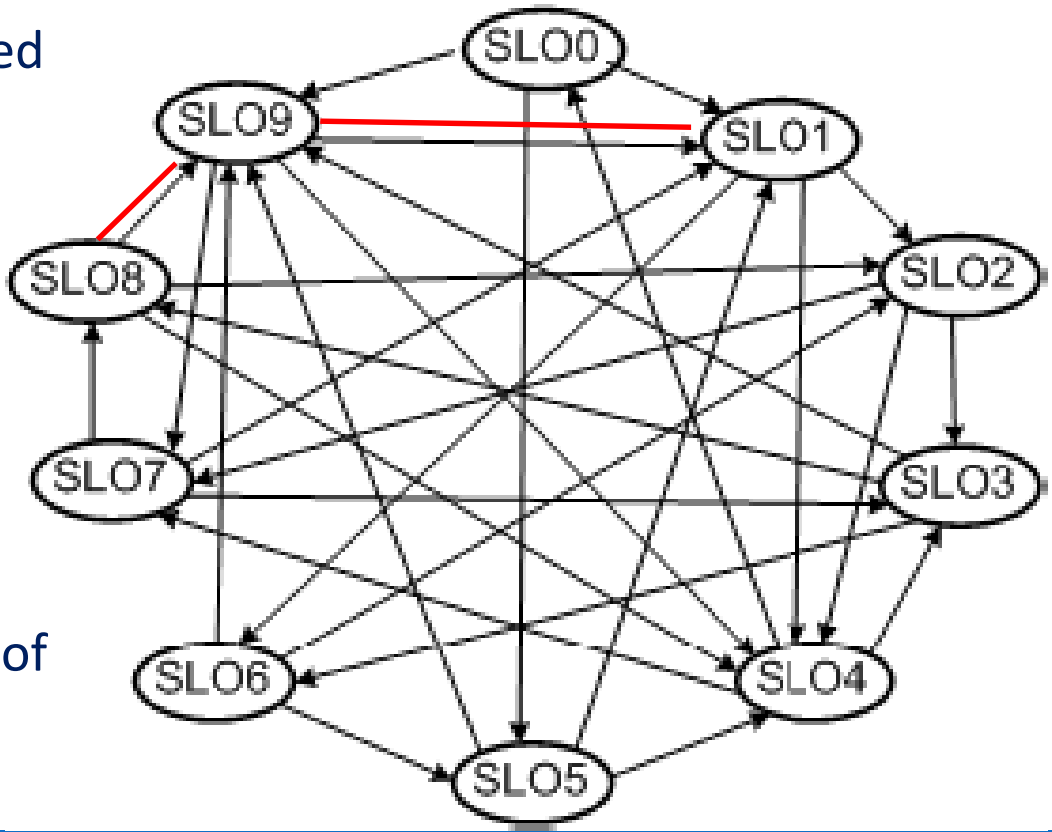
1. Introduction to Software Evolution
2. Taxonomy of Software Maintenance and Evolution
3. Evolution and Maintenance Models
4. Reuse and Domain Engineering
5. Program Comprehension
6. Impact Analysis
7. Refactoring
8. Reengineering
9. Legacy Information Systems

Identifying the Candidate Impact Set

- The next step of the impact analysis process is defining CIS (Candidate Impact Set)
- Changes in one part of the software system may have direct impacts or indirect impacts on other parts
- The SIS is augmented with software lifecycle objects (SLOs) that are likely to change because of changes in the elements of the SIS.
- The change could be of direct impact and/or indirect impact.
  - **Direct impact:** A direct impact relation exists between two entities, if the two entities are related by a fan-in and/or fan-out relation.
  - **Indirect impact:** If an entity A directly impacts another entity B and B directly impacts a third entity C, then we can say that A indirectly impacts C. A→B , B→C ➔ A→C

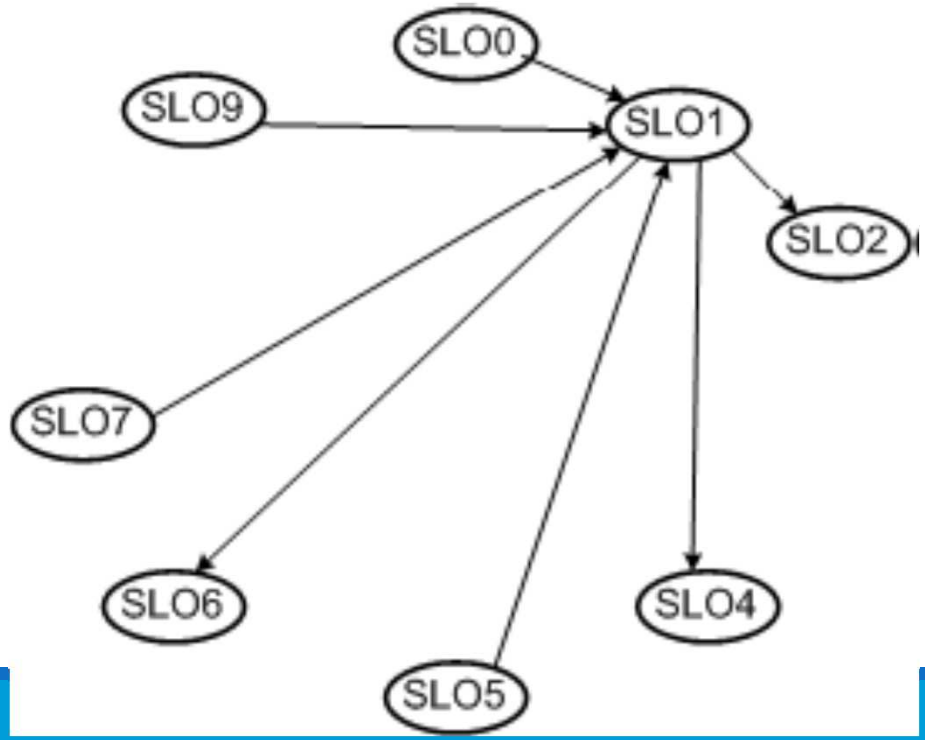
Identifying the Candidate Impact Set

- software lifecycle objects (SLOs) form a directed graph. SLO represents a software artifact connected to other artifacts.
- Dependencies among SLOs are represented by arrows.
- In the example, SLO1 has a direct impact from SLO9 and an indirect impact from SLO8.
- The in-degree of a node i reflects the number of known nodes that depend on i.



Identifying the Candidate Impact Set

- The four nodes – SLO0, SLO5, SLO7 and SLO9 – are dependent on SLO1, and the in-degree of SLO1 is four.
- The out-degree of SLO1 is three.



Identifying the Candidate Impact Set

- The connectivity matrix is constructed by considering the SLOs and the relationships shown in the SLOs directed graph (previous slide)
- A reachability graph can be easily obtained from a connectivity matrix.
- A reachability graph shows the entities that can be impacted by a modification to a SLO, and there is a likelihood of over-estimation.

	SLO0	SLO1	SLO2	SLO3	SLO4	SLO5	SLO6	SLO7	SLO8	SLO9
SLO0		x				x				x
SLO1			x		x		x			
SLO2				x	x			x		
SLO3							x		x	x
SLO4	x			x				x		
SLO5		x			x					x
SLO6			x			x				x
SLO7		x		x					x	
SLO8			x		x					x
SLO9		x			x			x		

Identifying the Candidate Impact Set

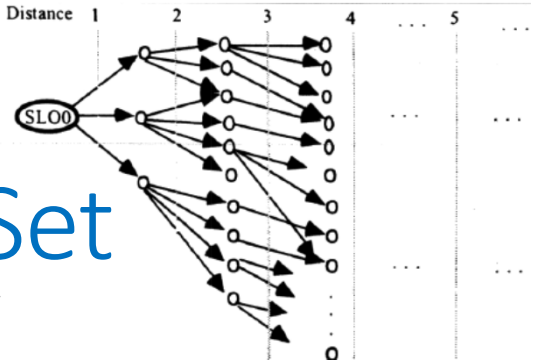
- The dense reachability matrix has the risk of over-estimating the CIS.
- To minimize the occurrences of false positives, one might consider the following two approaches:
  - Distance based approach.
  - Incremental approach.

	SLO0	SLO1	SLO2	SLO3	SLO4	SLO5	SLO6	SLO7	SLO8	SLO9
SLO0		x	x	x	x	x	x	x	x	x
SLO1	x		x	x	x	x	x	x	x	x
SLO2	x	x		x	x	x	x	x	x	x
SLO3	x	x	x		x	x	x	x	x	x
SLO4	x	x	x	x		x	x	x	x	x
SLO5	x	x	x	x	x		x	x	x	x
SLO6	x	x	x	x	x	x		x	x	x
SLO7	x	x	x	x	x	x	x		x	x
SLO8	x	x	x	x	x	x	x	x		x
SLO9	x	x	x	x	x	x	x	x	x	

Identifying the Candidate Impact Set

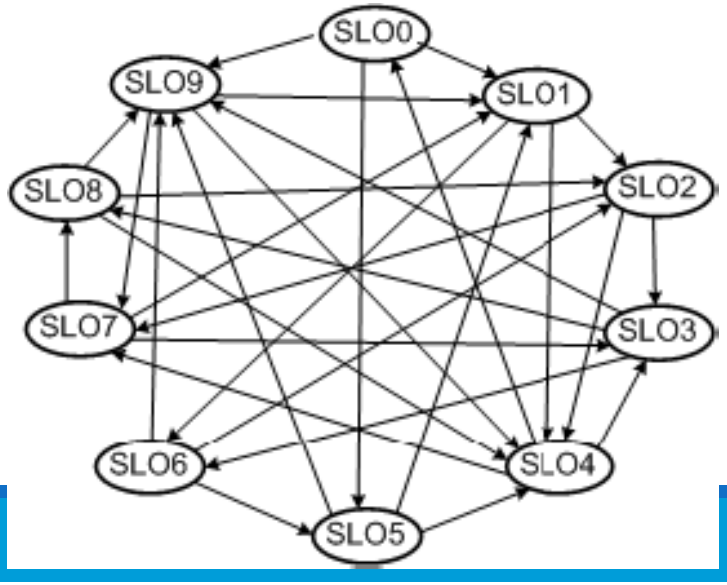
- **Distance based approach:** In this approach, SLOs which are farther than a threshold distance from SLO i are not be considered to be impacted by changes in SLO i.
- **Incremental approach:** In this approach, the CIS is incrementally constructed. For every SLO in the SIS, one considers all the SLOs interacting with it, and only SLOs that are actually impacted by the change request are put in the CIS.

	SLO0	SLO1	SLO2	SLO3	SLO4	SLO5	SLO6	SLO7	SLO8	SLO9
SLO0		1	2	3	2	1	2	2	3	1
SLO1	2		1	2	1	2	1	2	3	2
SLO2	2	2		1	1	3	2	1	2	2
SLO3	3	2	2		2	2	1	2	1	1
SLO4	1	2	3	1		2	2	1	2	2
SLO5	2	1	2	2	1		2	2	3	1
SLO6	3	2	1	2	2	1		2	3	1
SLO7	3	1	2	1	2	3	2		1	2
SLO8	3	2	1	2	1	3	3	2		1
SLO9	2	1	2	2	1	3	2	1	2	



Identifying the Candidate Impact Set

- The connectivity matrix is constructed by considering the SLOs and the relationships shown in the SLOs directed graph (previous slide)
- A reachability graph can be easily obtained from a connectivity matrix.



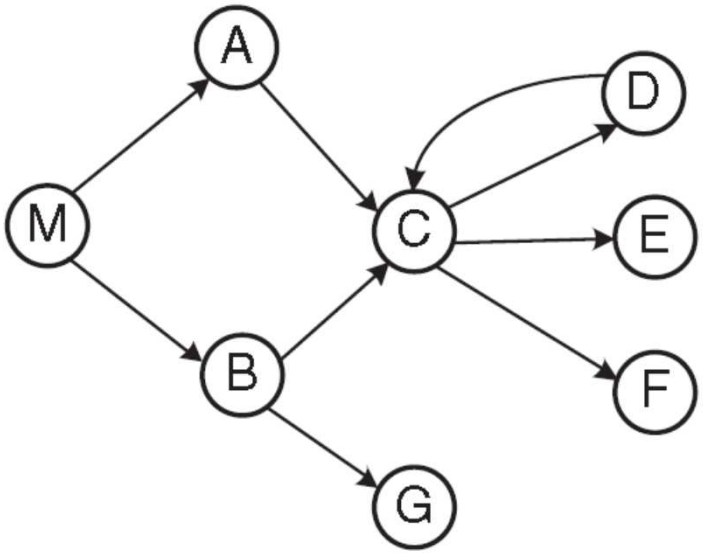
	SLO 0	SLO 1	SLO 2	SLO 3	SLO 4	SLO 5	SLO 6	SLO 7	SLO 8	SLO 9
SLO0		1	2	3	2	1	2	2	3	1
SLO1	2		1	2	1	2	1	2	3	2
SLO2	2	2		1	1	3	2	1	2	2
SLO3	3	2	2		2	2	1	2	1	1
SLO4	1	2	3	1		2	2	1	2	2
SLO5	2	1	2	2	1		2	2	3	1
SLO6	3	2	1	2	2	1		2	3	1
SLO7	3	1	2	1	2	3	2		1	2
SLO8	3	2	1	2	1	3	3	2		1
SLO9	2	1	2	2	1	3	2	1	2	

Dependency-based Impact Analysis

- In general, source code objects are analyzed to obtain vertical traceability information.
- Dependency based impact analysis techniques identify the impact of changes by analyzing syntactic dependencies, because syntactic dependencies are likely to cause semantic dependencies.
- Two traditional impact analysis techniques are explained:
  1. Based on call graph.
  2. Based on dependency graph.

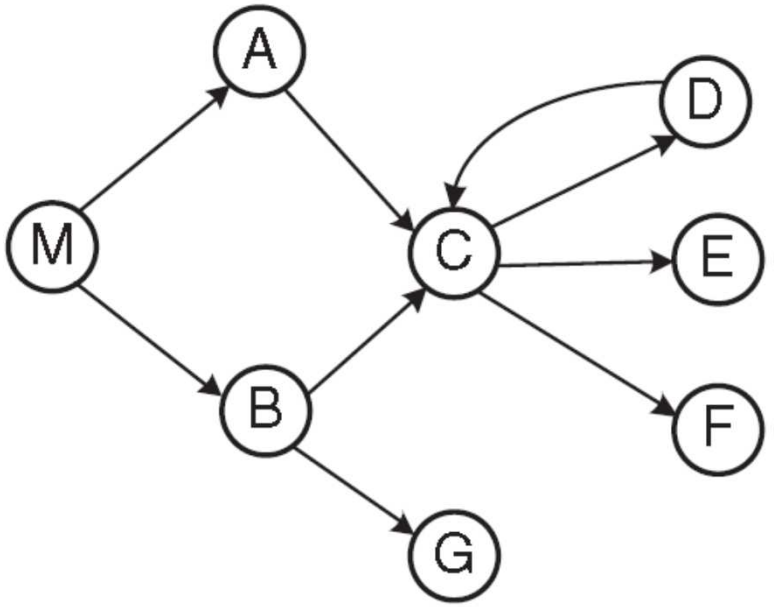
Dependency-based Impact Analysis - Call Graph

- A call graph is a directed graph in which a node represents a function, a component, or a method.
- An edge between two nodes A and B means that A may invoke B
- Programmers use call graphs to understand the potential impacts that a software change may have.
- Let P be a program, G be the call graph obtained from P, and p be some procedure in P → A key assumption in the call graph-based technique is that some change in p has the potential to impact changes in all nodes reachable from p in G.



Dependency-based Impact Analysis - Call Graph

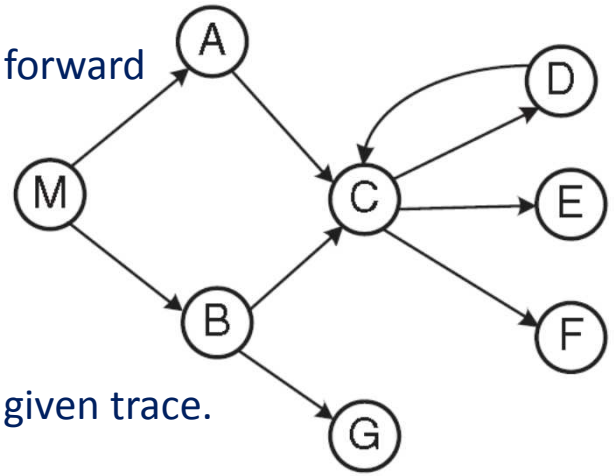
- The call graph-based approach to impact analysis suffers from many disadvantages as follows:
  - impact analysis based on call graphs can produce an imprecise impact set. For example, one cannot determine the conditions that cause impacts of changes to propagate from M to other procedures.
  - impact propagations due to procedure returns are not captured in the call graph-based technique. Suppose that *E* is modified and control returns to C. Now, following the return to C, it cannot be inferred whether impacts of changing E propagates into none, both, A, or B.



Dependency-based Impact Analysis - Call Graph

- Let us consider an 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 with respect to the given trace is computed by forward searching in the trace to find:
  - procedures that are indirectly or directly invoked by E; and
  - procedures that are invoked after E terminates.
- identify the procedures into which E returns by performing backward search in the given trace.
- For example, in the given trace, E does not invoke other entities, but it returns into M, A, and C.
- Due to a modification in E, the set of potentially impacted procedures is {M,A,C, E}.

Dependency-based Impact Analysis - Program Dependency Graph

- In the program dependency graph (PDG) of a program:
  - each simple statement is represented by a node, also called a vertex;
  - each predicate expression is represented by a node.
- There are two types of edges in a PDG: data dependency edges and control dependency edges.
- Let v_i and v_j be two nodes in a PDG.
- If there is a data dependency edge from node v_i to node v_j , then the computations performed at node v_i are directly dependent upon the results of computations performed at node v_j .
- A control dependency edge from node v_i to node v_j indicates that node v_i may execute based on the result of evaluation of a condition at v_j .

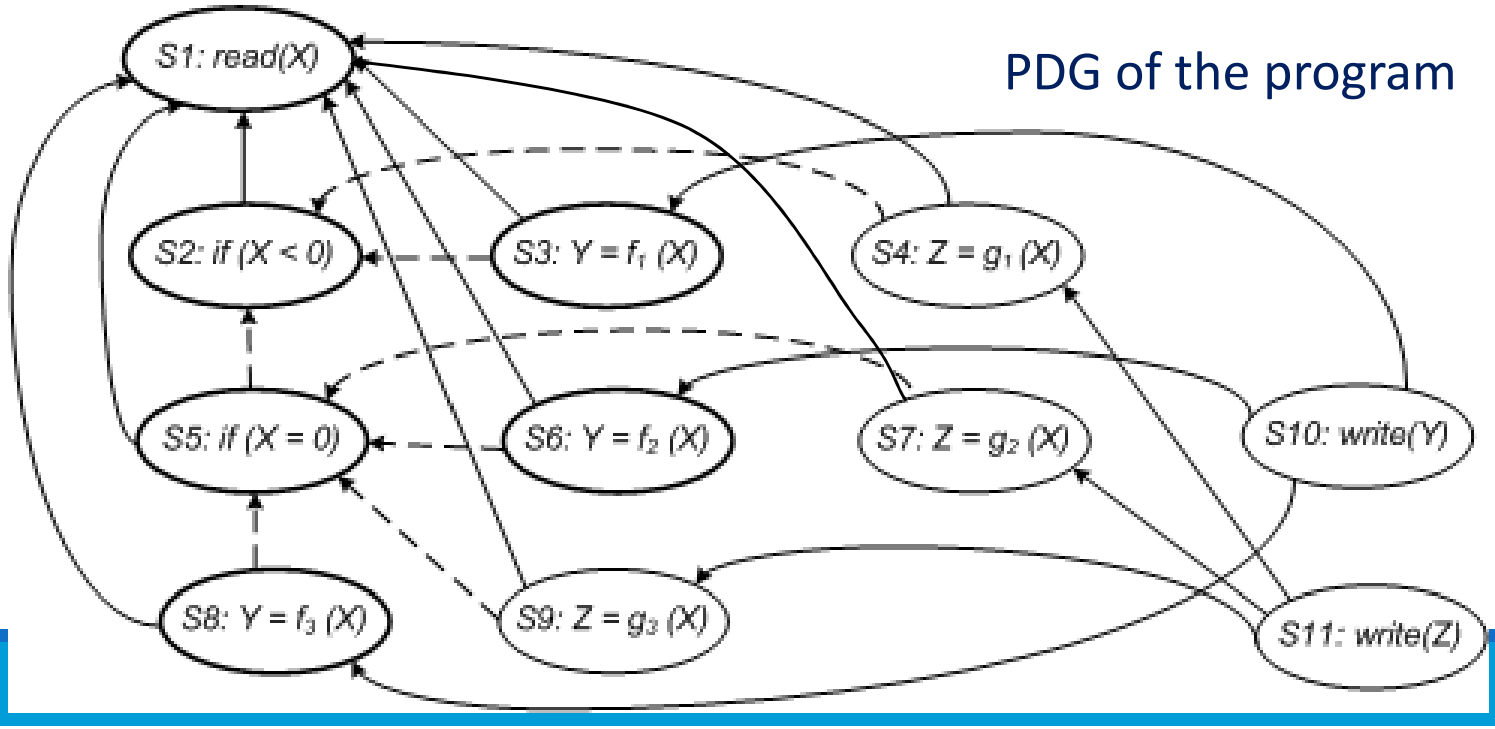
Dependency-based Impact Analysis - Program Dependency Graph

Program Code

```
        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);
            end_if;
        end_if;
S10 :   write(Y);
S11 :   write(Z);
    end
```

- Data dependencies are shown as solid edges, whereas control dependencies are shown as dashed edges.

PDG of the program

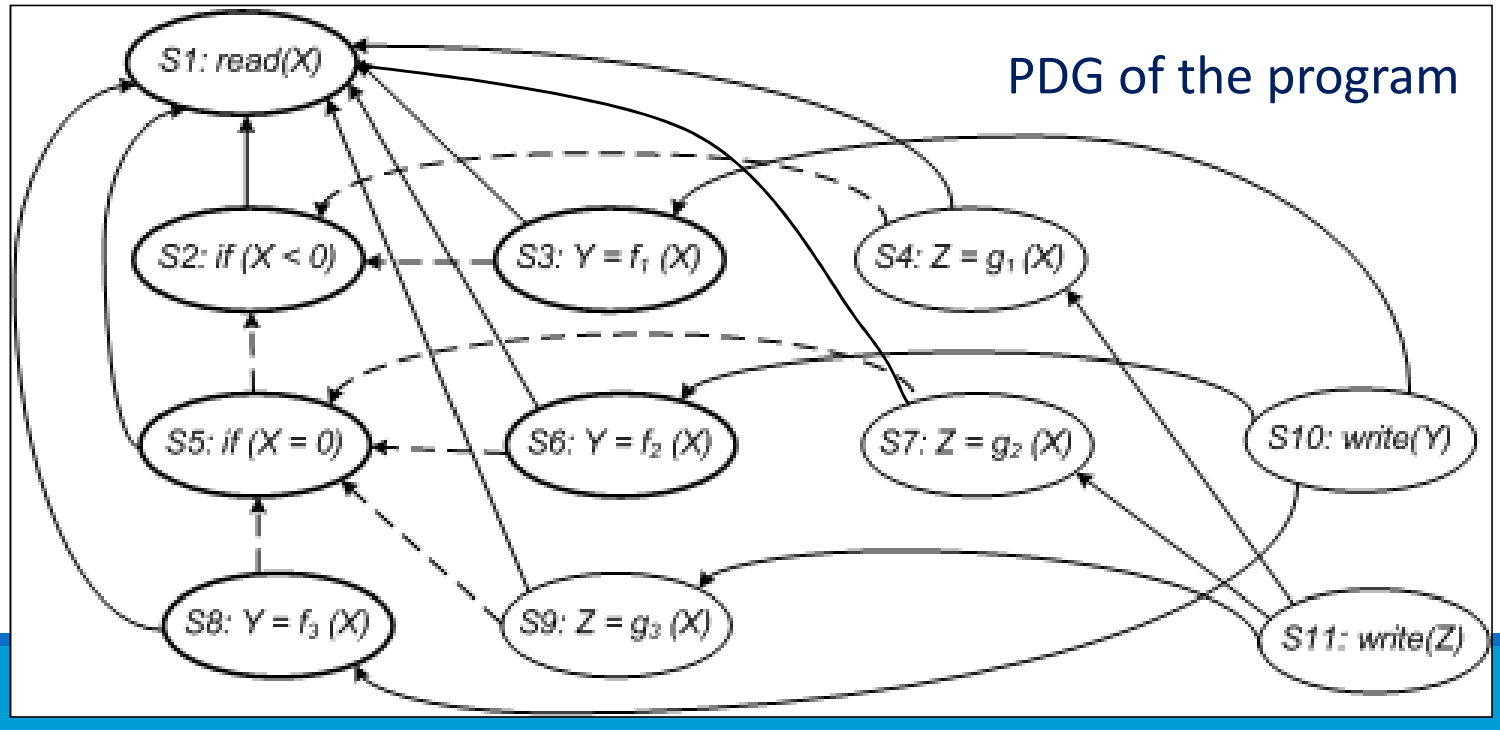


Dependency-based Impact Analysis - Program Dependency Graph

Program Code

```
        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);
            end_if;
          end_if;
S10 :     write(Y);
S11 :     write(Z);
        end
```

- Data dependencies are shown as solid edges, whereas control dependencies are shown as dashed edges.

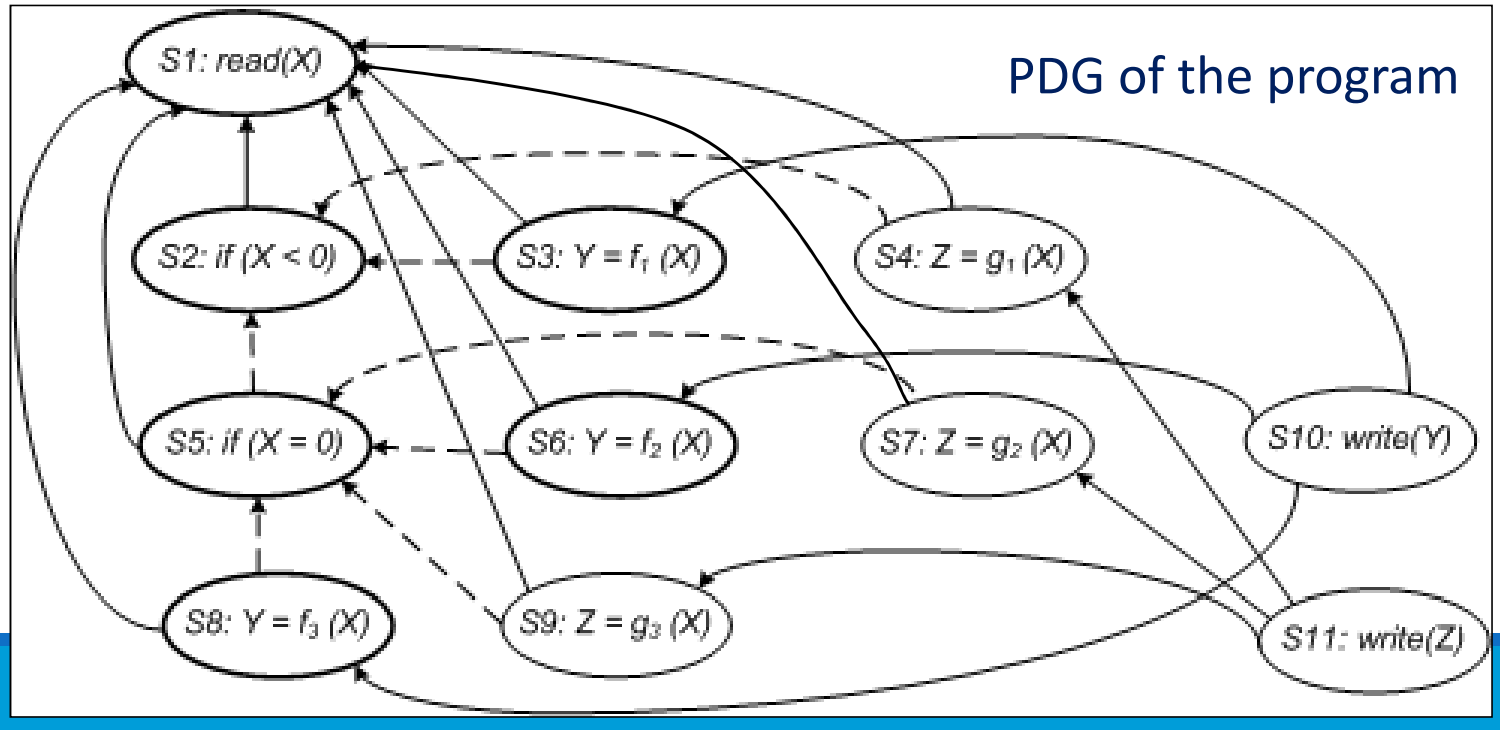


Dependency-based Impact Analysis - Program Dependency Graph

Program Code

```
        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);
                end_if;
            end_if;
S10 :       write(Y);
S11 :       write(Z);
        end
```

- Data dependencies are shown as solid edges, whereas control dependencies are shown as dashed edges.

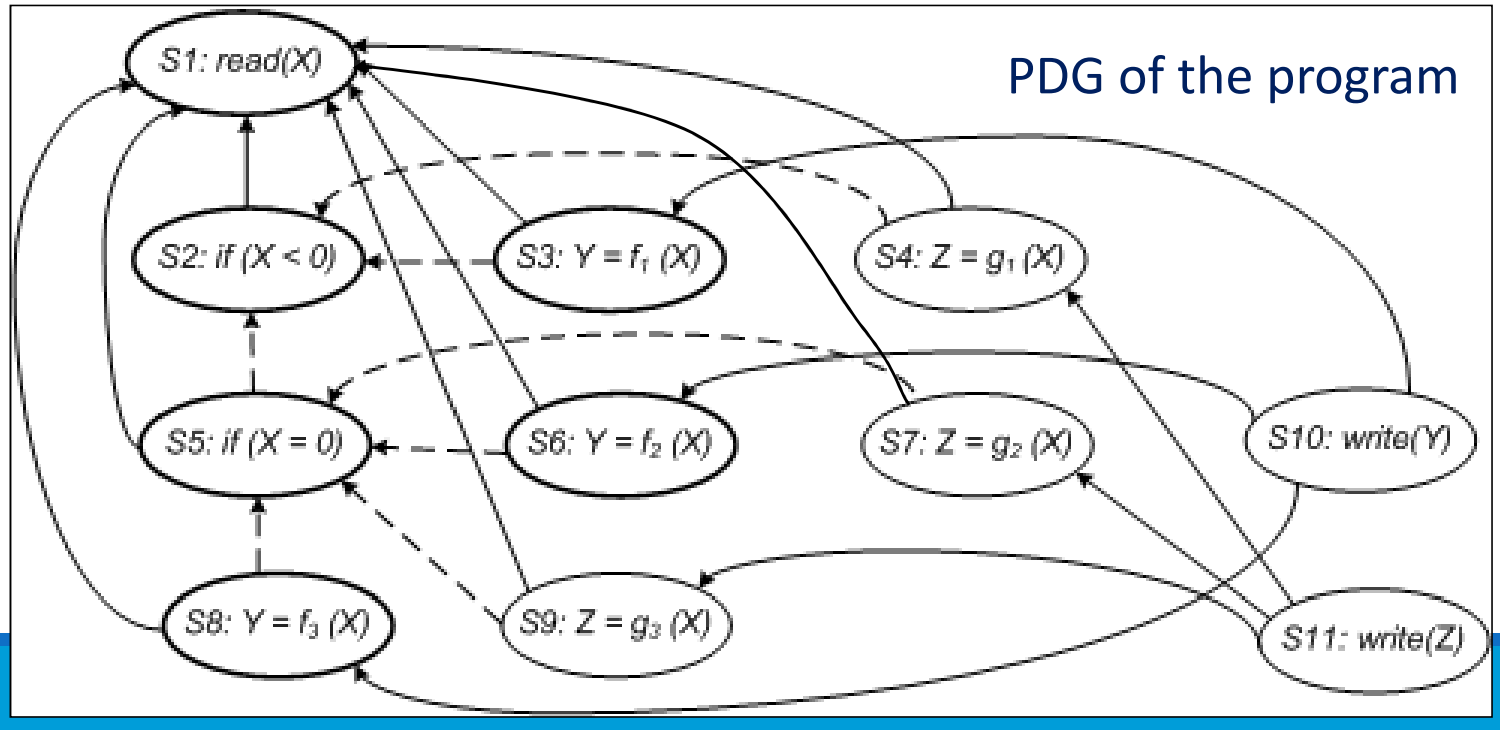


Dependency-based Impact Analysis - Program Dependency Graph

Program Code

```
        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);
            end_if;
        end_if;
S10 :   write(Y);
S11 :   write(Z);
    end
```

- Data dependencies are shown as solid edges, whereas control dependencies are shown as dashed edges.

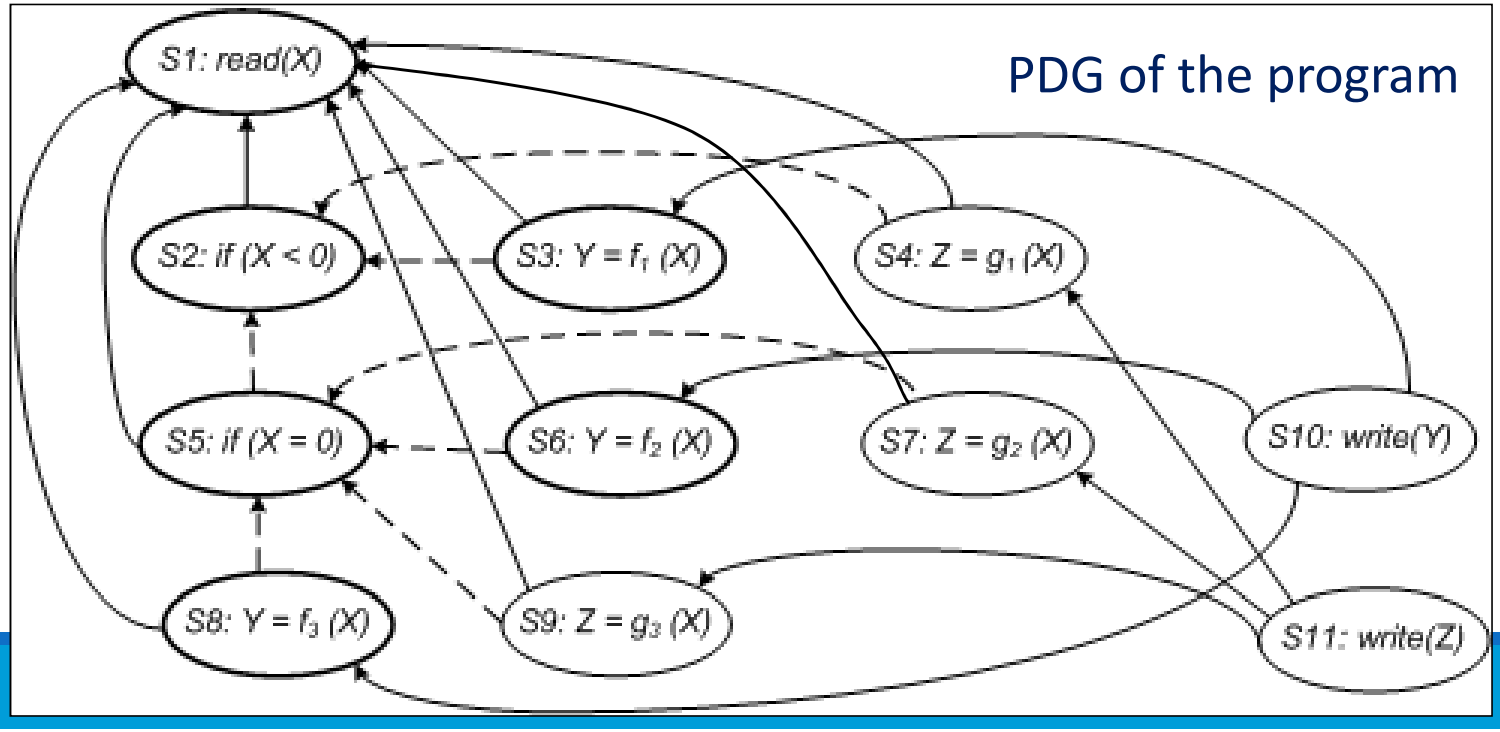


Dependency-based Impact Analysis - Program Dependency Graph

Program Code

```
        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);
            end_if;
        end_if;
S10 :   write(Y);
S11 :   write(Z);
    end
```

- Data dependencies are shown as solid edges, whereas control dependencies are shown as dashed edges.

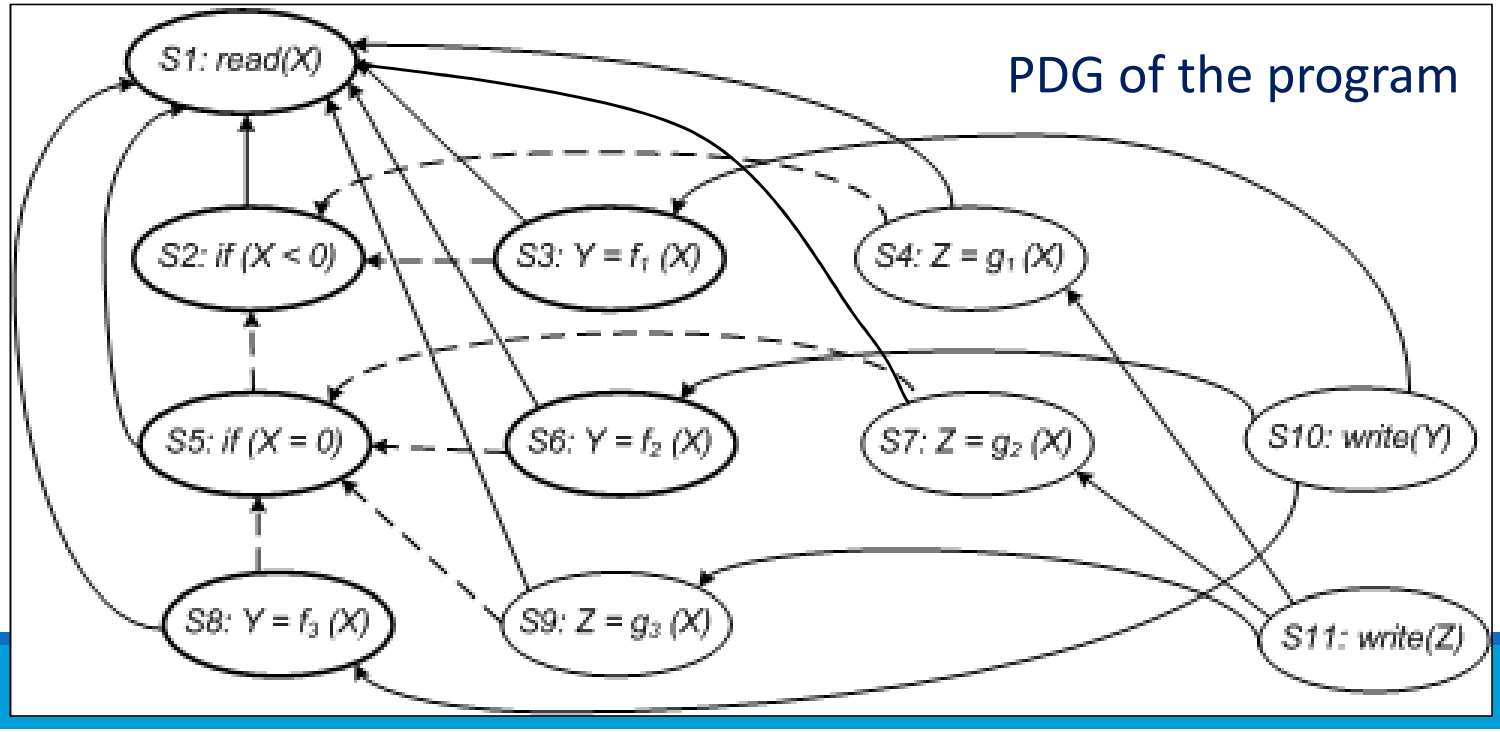


Dependency-based Impact Analysis - Program Dependency Graph

Program Code

```
        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);
            end_if;
        end_if;
S10 :   write(Y);
S11 :   write(Z);
    end
```

- Data dependencies are shown as solid edges, whereas control dependencies are shown as dashed edges.

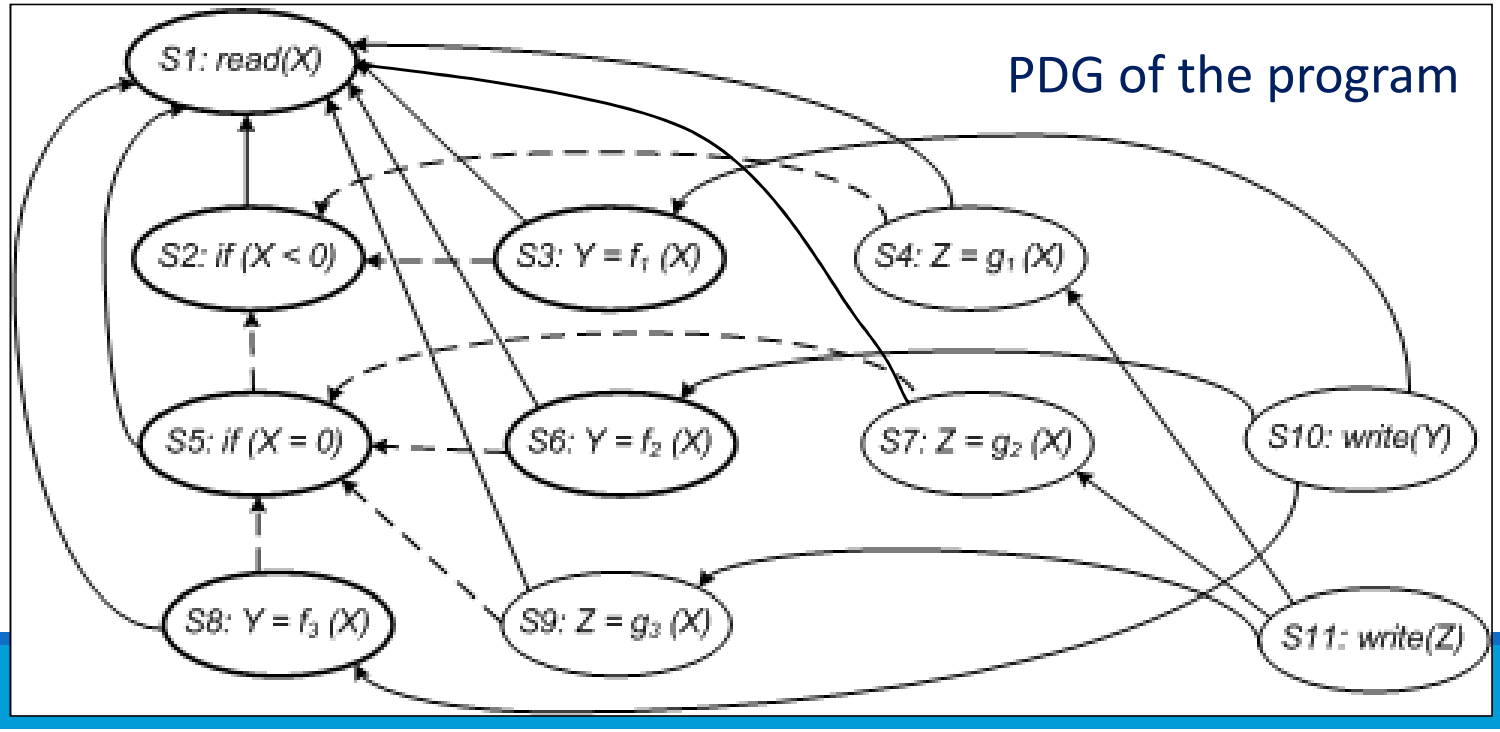


Dependency-based Impact Analysis - Program Dependency Graph

Program Code

```
        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);
            end_if;
        end_if;
S10 :   write(Y);
S11 :   write(Z);
    end
```

- Data dependencies are shown as solid edges, whereas control dependencies are shown as dashed edges.



Dependency-based Impact Analysis - Program Dependency Graph

Program Code

```
        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);
            end_if;
        end_if;
S10 :   write(Y);
S11 :   write(Z);
    end
```

- Data dependencies are shown as solid edges, whereas control dependencies are shown as dashed edges.

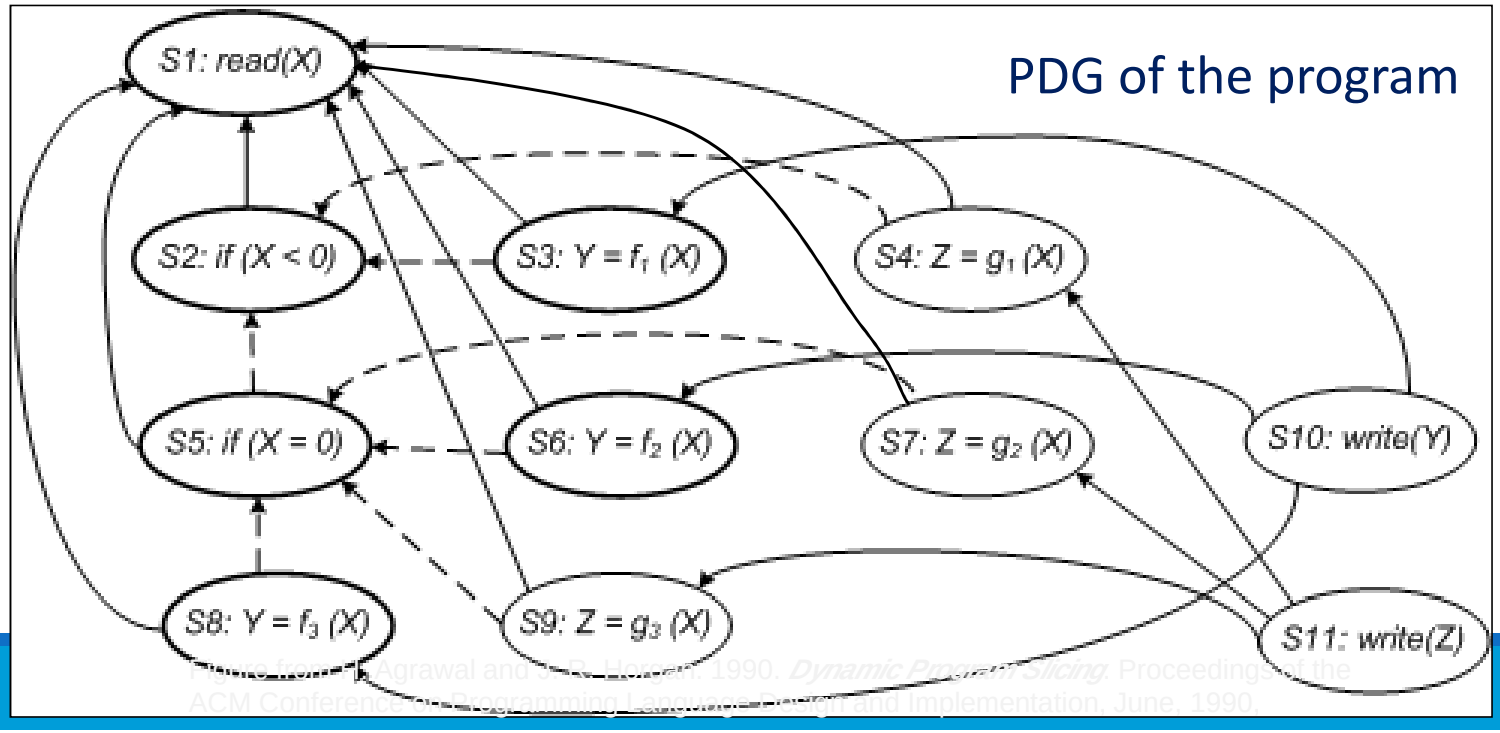


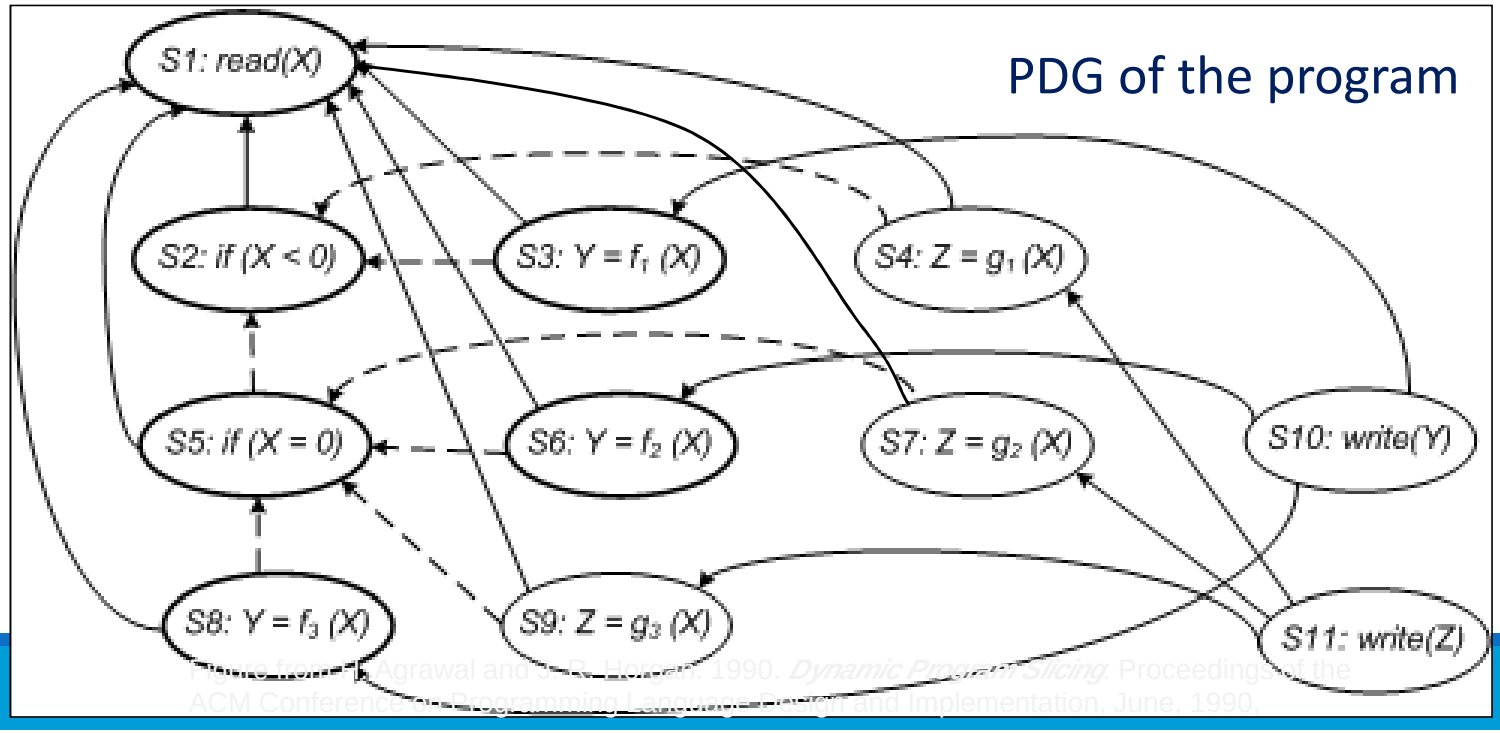
Figure from H. Agrawal and J. R. Horgan. 1990. *Dynamic Program Slicing.* Proceedings of the ACM Conference on Programming Language Design and Implementation, June, 1990.

Dependency-based Impact Analysis - Program Dependency Graph

Program Code

```
        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);
            end_if;
        end_if;
S10 :   write(Y);
S11 :   write(Z);
    end
```

- Data dependencies are shown as solid edges, whereas control dependencies are shown as dashed edges.



PDG of the program

Figure from R. Agrawal and H. Pohlmann. 1990. *Dynamic Program Slicing.* Proceedings of the ACM Conference on Programming Language Design and Implementation, June, 1990.

Dependency-based Impact Analysis - Program Dependency Graph

Program Code

```
        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);
            end_if;
        end_if;
S10 :   write(Y);
S11 :   write(Z);
    end
```

- Data dependencies are shown as solid edges, whereas control dependencies are shown as dashed edges.

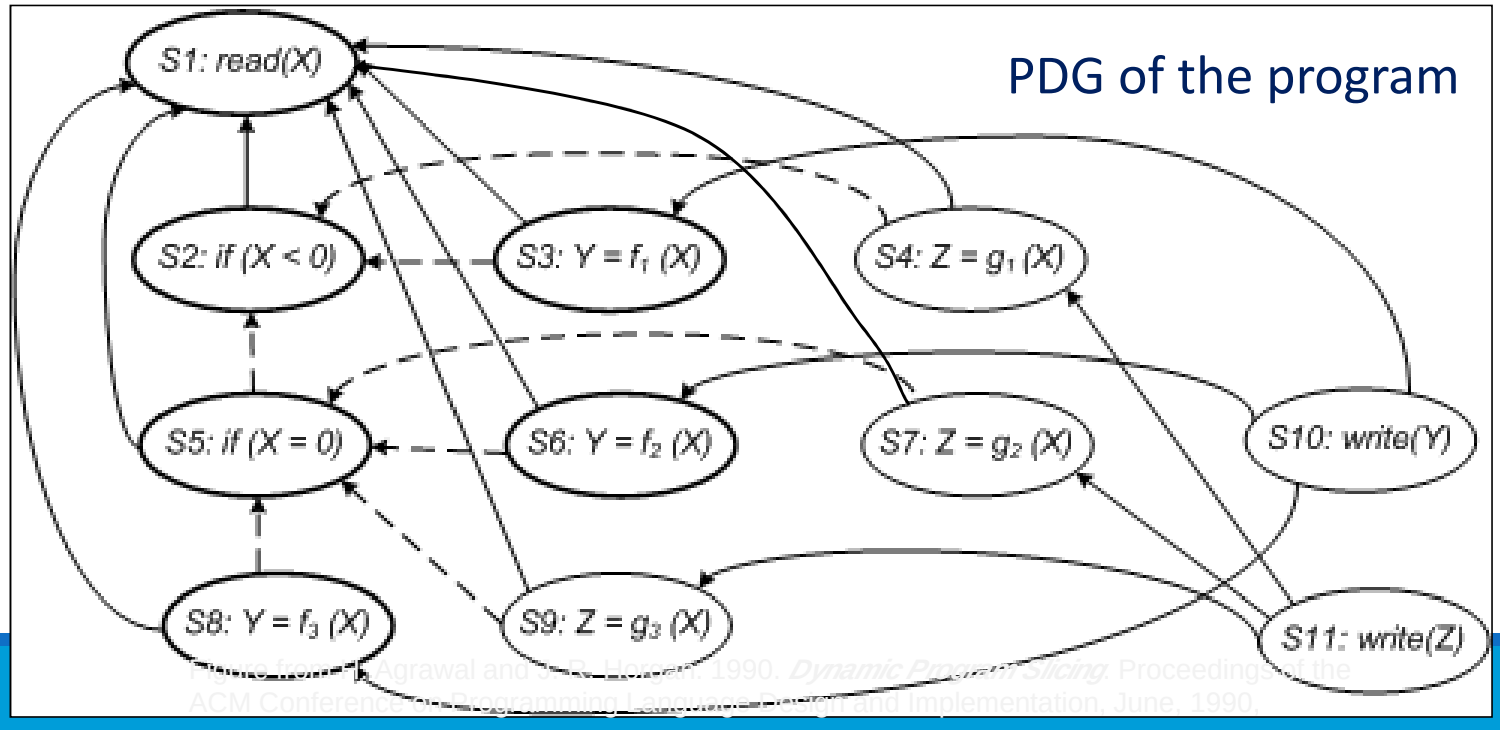


Figure from H. Agrawal and J. R. Horgan. 1990. *Dynamic Program Slicing.* Proceedings of the ACM Conference on Programming Language Design and Implementation, June, 1990.

Dependency-based Impact Analysis - Program Dependency Graph

Program Code

```
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);
            end_if;
        end_if;
S10 :   write(Y);
S11 :   write(Z);
end
```

- Data dependencies are shown as solid edges, whereas control dependencies are shown as dashed edges.

PDG of the program

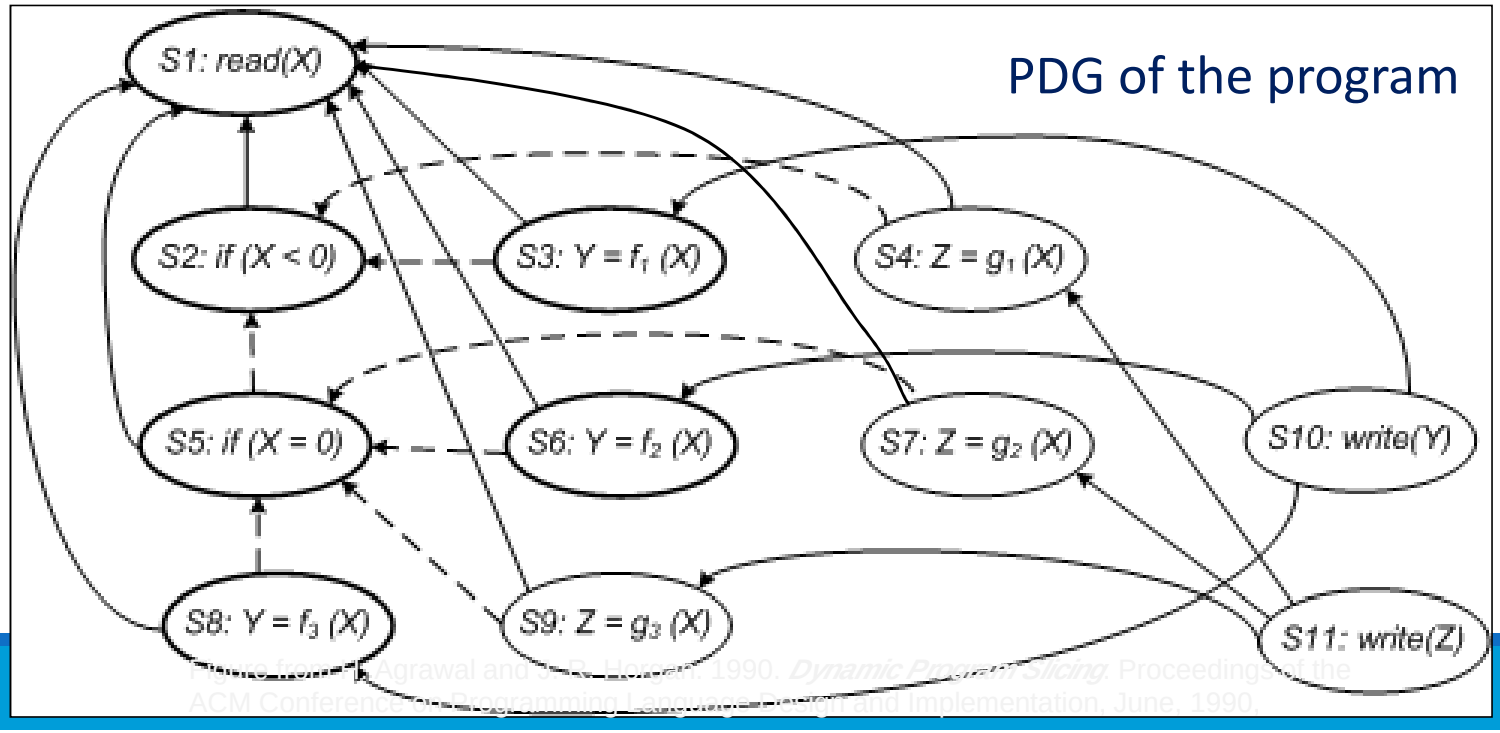


Figure from H. Agrawal and J. R. Horgan. 1990. *Dynamic Program Slicing.* Proceedings of the ACM Conference on Programming Language Design and Implementation, June, 1990.

Dependency-based Impact Analysis - Program Dependency Graph

Program Code

```
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);
            end_if;
        end_if;
S10 :   write(Y);
S11 :   write(Z);
end
```

- Data dependencies are shown as solid edges, whereas control dependencies are shown as dashed edges.

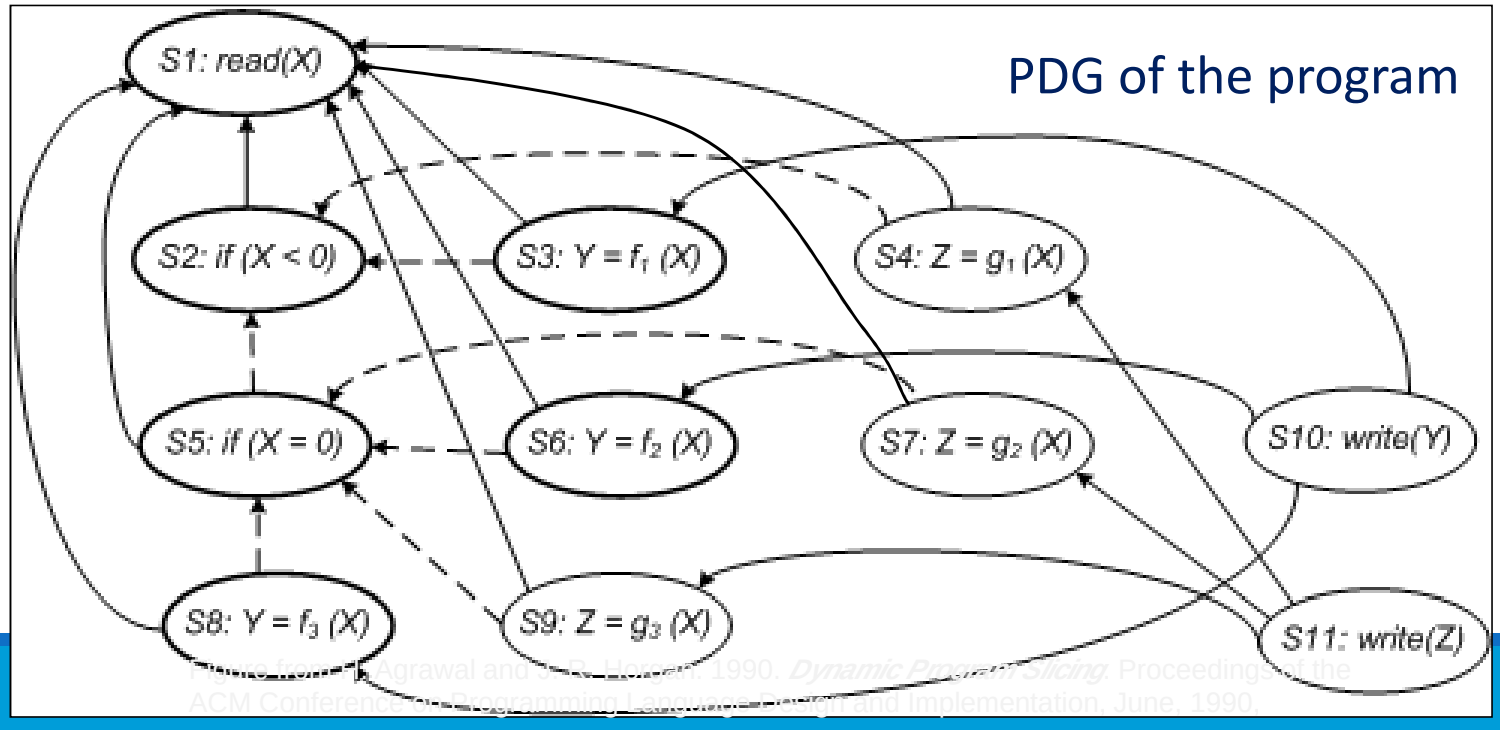
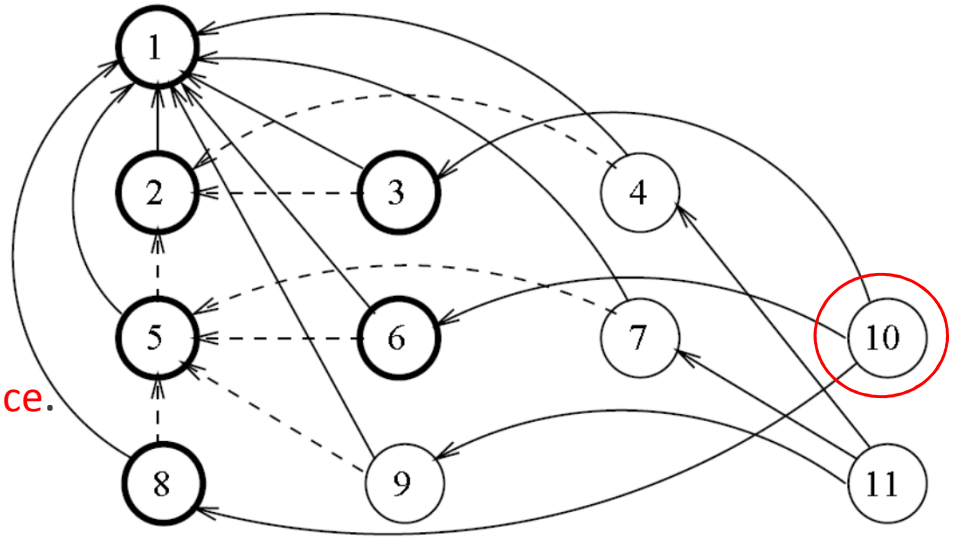


Figure from H. Agrawal and J. R. Horgan. 1990. *Dynamic Program Slicing.* Proceedings of the ACM Conference on Programming Language Design and Implementation, June, 1990.

Dependency-based Impact Analysis -Program Dependency Graph- Static Program Slice



- A static program slice is identified 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.
- Consider variable Y at S10.

1. find all the reaching definitions of Y at node S10 – and the answer is the set of nodes {S3, S6 and S8}.
2. find the set of all nodes which are reachable from {S3, S6 and S8} – and the answer is the set {S1, S2, S3, S5, S6, S8}.

Dependency-based Impact Analysis -Program Dependency Graph - Dynamic Slice

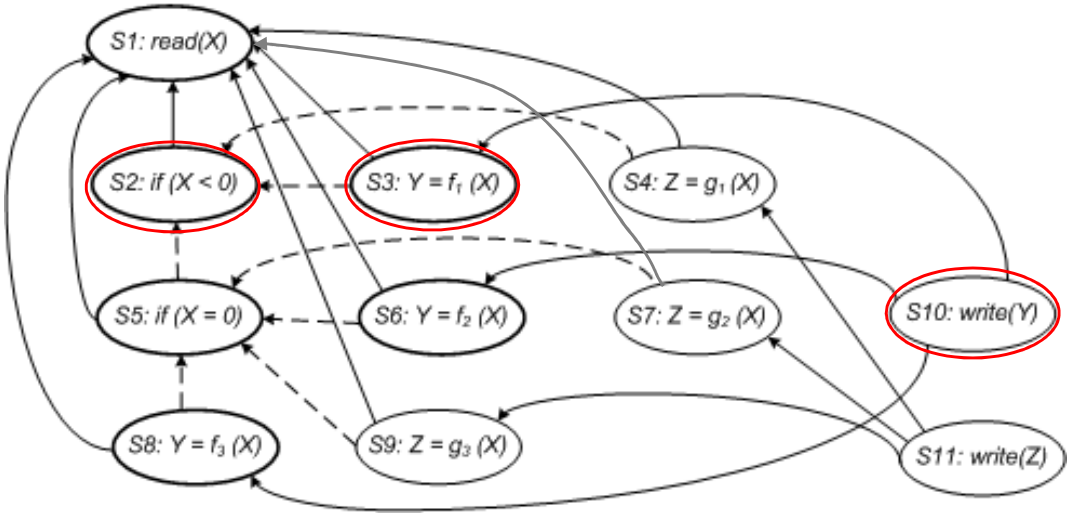
- A dynamic slice is more useful in localizing the defect than the static slice.
- Only one of the three assignment statements, S3, S6, or S8, may be executed for any input value of X.
- Consider the input value −1 for the variable X.
- For −1 as the value of X, only S3 is executed.
- Therefore, with respect to variable Y at S10, the dynamic slice will contain only {S1, S2 , S3}.

Program Code

```
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);
            end_if;
        end_if;
S10 :   write(Y);
S11 :   write(Z);
```

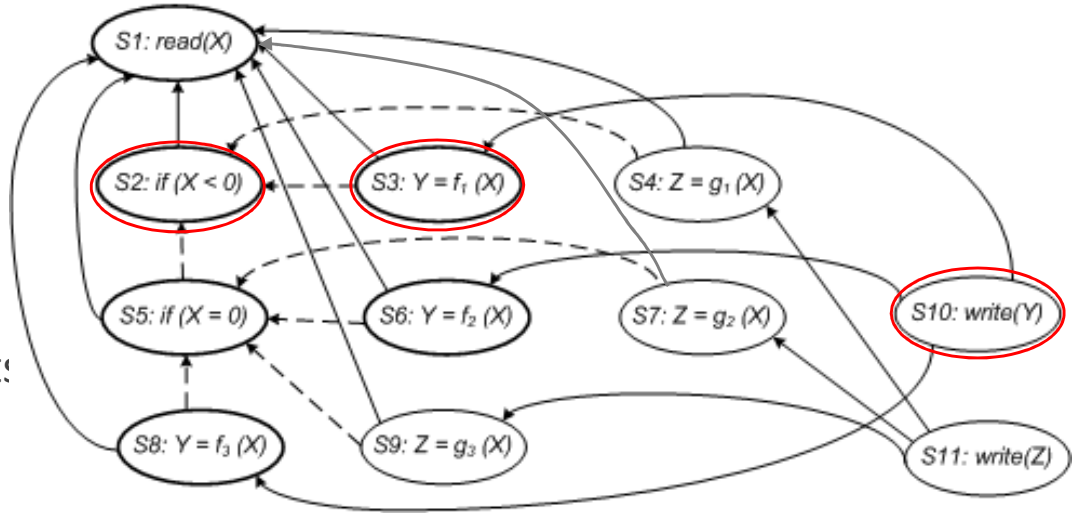
Dependency-based Impact Analysis -Program Dependency Graph - Dynamic Slice

- Now, in our example the static program slice with respect to variable Y at S10 for the code contains all the three statements – S3, S6, and S8.
- However, for a given test, one statement from the set {S3, S6, and S8} is executed.
- A simple way to finding dynamic slices is as follows:
  - for the current test, mark the executed nodes in the PDG.
  - traverse the marked nodes in the graph.



Dependency-based Impact Analysis -Program Dependency Graph - Dynamic Slice

- For −1 as the value of X, if the value of Y is incorrect at S10, one can infer that either f_1 is erroneous at S3 or the "if" condition at S2 is incorrect.
- A simple approach to obtaining dynamic program slices is:
  - Given a test var and a PDG, let us represent the execution history of the program as a sequence of vertices < v_1, v_2, ..., v_n >.
  - The 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.



Questions

?

Reading

- Chapter 6