Faculty of Computers and Information
BSc. Software Engineering Program

Dr. Lamia Abo Zaid

د. لمياء أبوزيد

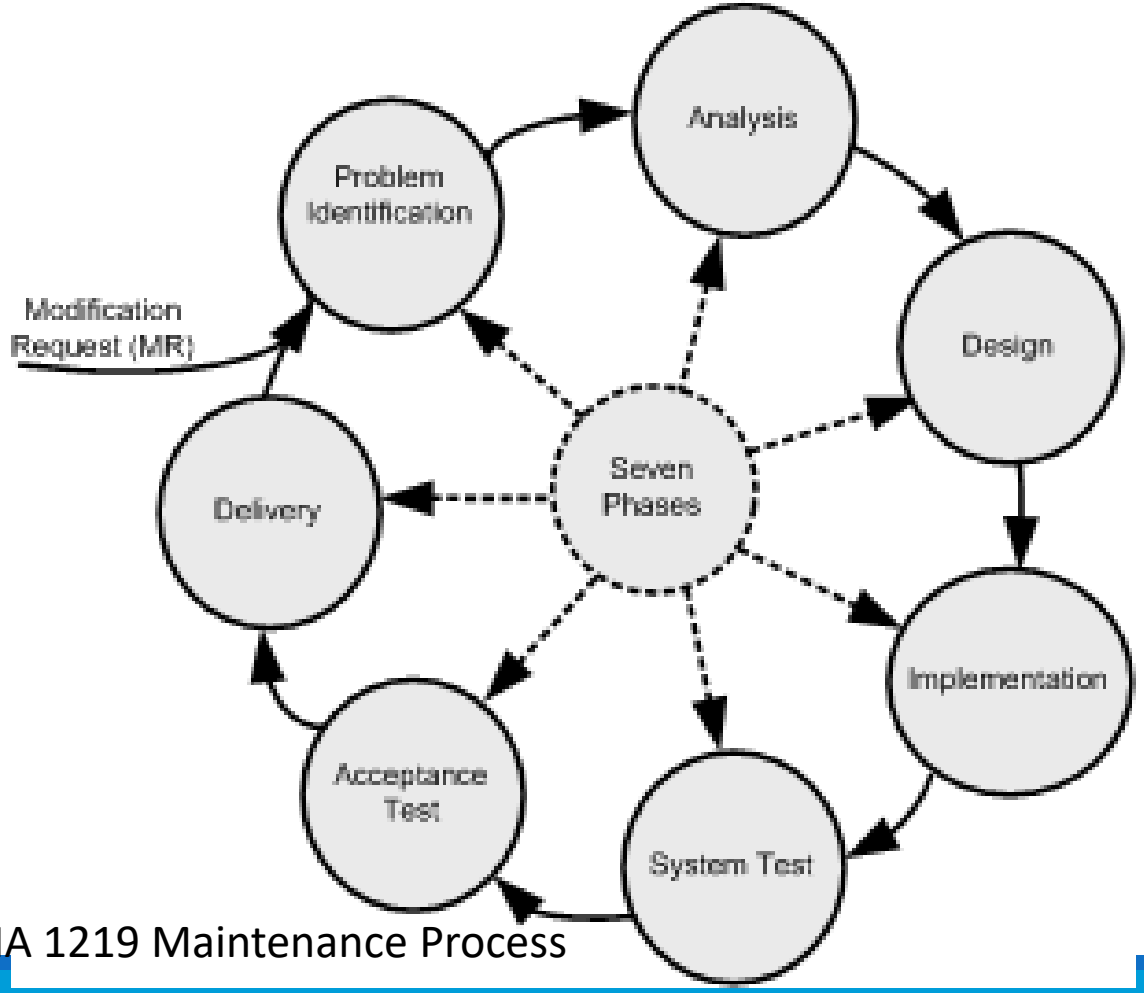
Software Evolution : TOC

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

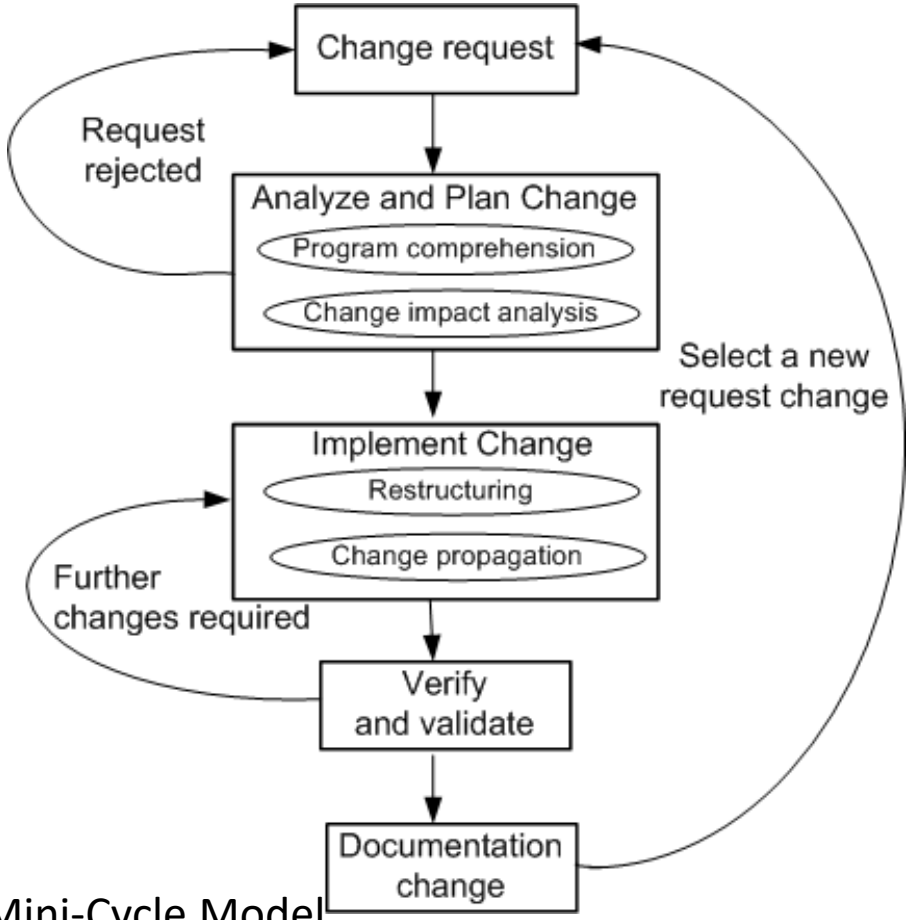
Impact Analysis

- Impact analysis is a tool for controlling change, and thus for avoiding deterioration
- The maintenance process is started by performing impact analysis. Impact analysis enables understanding impact of change via identifying the components that are impacted by the Change Request (CR).
- Impact of the changes are analyzed for the following reasons:
  - to estimate the cost of executing the change request.
  - to determine whether some critical portions of the system are going to be impacted due to the requested change.
  - to record the history of change related information for future evaluation of changes.
  - to understand how items of change are related to the structure of the software.
  - to determine the portions of the software that need to be subjected to regression testing after a change.

Impact Analysis - MR cycle



IEEE/EIA 1219 Maintenance Process



Change Mini-Cycle Model

Traceability

- traceability as the ability to trace between software artifacts generated and modified during the software product life cycle
- Traceability of artifacts between different models is known as external traceability,
- Tracing dependent artifacts within the same model is known as internal traceability.
  - Internal traceability primarily focuses on source code artifacts.
  - classical impact analysis techniques, based on program dependency
    - call-graph-based analysis,
    - static program slicing,
    - dynamic program slicing.

Ripple Effect Analysis

- A topic related to impact analysis is ripple effect analysis.
- Ripple effect means that a modification to a single variable may require several parts of the software system to be modified.
- Analysis of ripple effect reveals what and where changes are occurring.
- Measurement of ripple effects provides the following information about an evolving software systems:
  - between successive versions of the same system, measurement of ripple effect will tell us how the software's complexity has changed.
  - when a new module is added to the system, measurement of ripple effect on the system will tell us how the software's complexity has changed because of the addition of the new module.

Ripple Effect Analysis

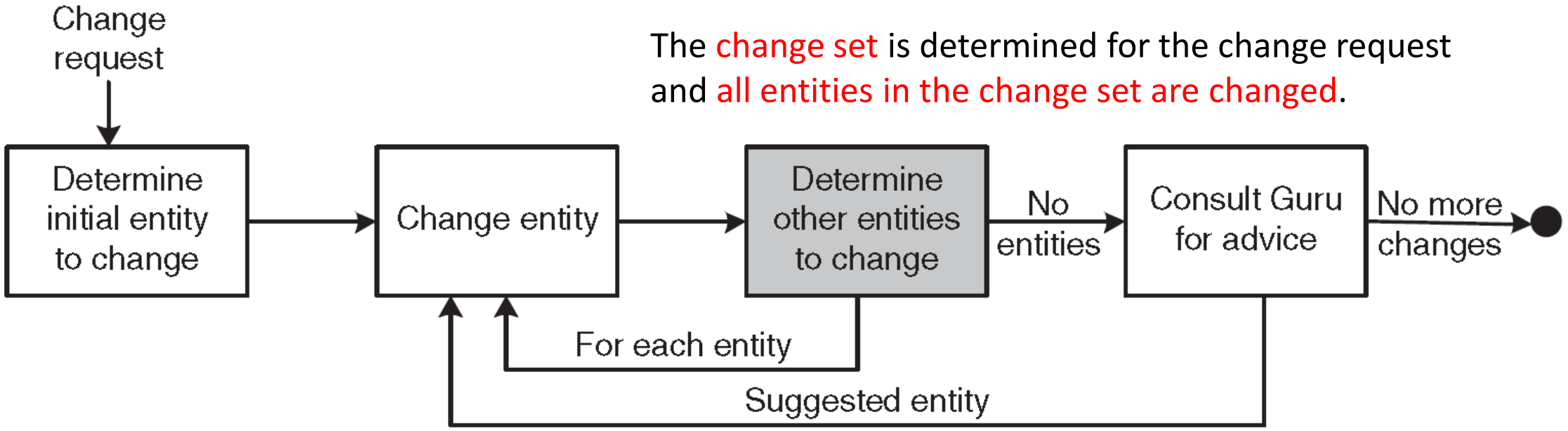
- Ripple effect is computed by means of error flow analysis.
  - In error flow analysis, definitions of program variables involved in a change are considered to be potential sources of errors. Inconsistency can propagate from those sources to other variables in the program. The other sources of errors are successively identified until error propagation is no more possible

Ripple Effect Analysis

- Stability reflects the resistance to the potential ripple effect which a program would have when it is changed.
- Measurement of ripple effects provides the following information about an evolving software systems:
  - between successive versions of the same system, measurement of ripple effect will tell us how the software's complexity has changed.
  - when a new module is added to the system, measurement of ripple effect on the system will tell us how the software's complexity has changed because of the addition of the new module.

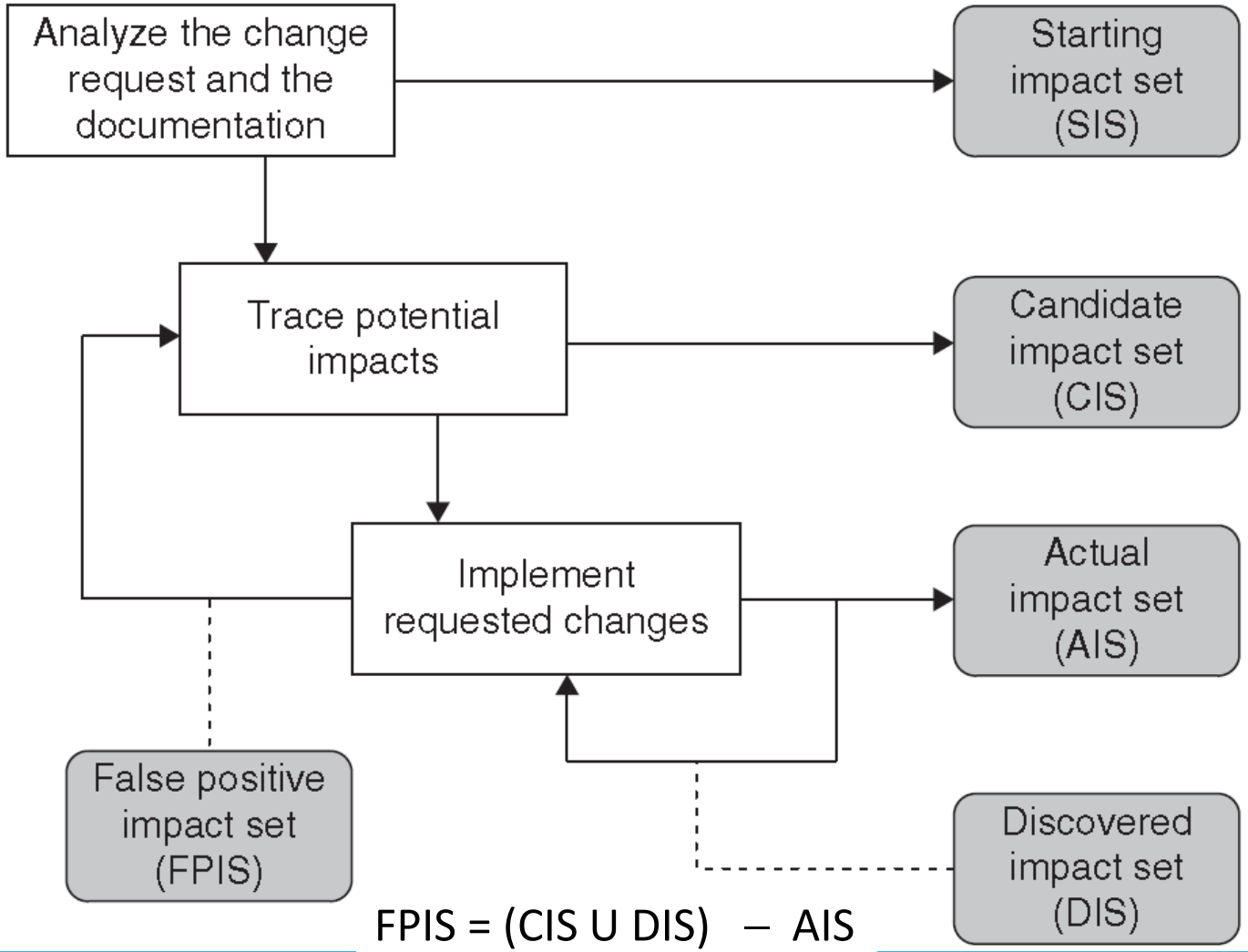
Change Propagation Model

- Change propagation means that if an entity (e.g. a function) is changed, then all related entities in the system are accordingly changed.
- Change propagation model is defined by Hassan and Holt (2006)



The change set is determined for the change request and all entities in the change set are changed.

Impact Analysis Process



The initial set of objects (or components) presumed to be impacted by a software CR

The set of objects (or components) estimated to be impacted according to a certain impact analysis approach

The set of objects (or components) actually changed as a result of performing a CR

the set of new objects (or components), not contained in CIS, discovered to be impacted while implementing a CR.

FPIS = (CIS U DIS) – AIS

Impact Analysis Process

- In the process of impact analysis it is important to minimize the differences between AIS and CIS, by eliminating false positives and identifying true impacts.
- to evaluate the impact analysis process two traditional information retrieval metrics: recall and precision are used:
  - **Recall:** represents the fraction of actual impacts contained in CIS, and it is computed as the ratio of $|CIS \cap AIS|/|AIS|$.
    - The value of recall is 1 when DIS is empty.
  - **Precision:** represents the fraction of candidate impacts that are actually impacted, and it is computed as the ratio of $|CIS \cap AIS| / |CIS|$.
    - For an empty FPIS set, the value of precision is 1

Impact Analysis Process - Adequacy

- Adequacy of an impact analysis approach is the ability of the approach to identify all the affected elements to be modified. Ideally, AIS $\subseteq$ CIS.
- Adequacy is repressed in terms of a performance metric called inclusiveness, as follows:

$$Inclusiveness = \begin{cases} 1 & \text{if AIS} \subseteq \text{CIS} \\ 0 & \text{otherwise} \end{cases}$$

- An inadequate approach is in fact useless, as it provides the maintenance engineer with incorrect information.

Impact Analysis Process - Effectiveness

- Effectiveness is the ability of an impact analysis technique to generate results that actually benefit the maintenance tasks
- Effectiveness is expressed in terms of three fine grained characteristics:

1. Ripple-sensitivity
2. Sharpness
3. Adherence

Impact Analysis Process - Effectiveness

1. Ripple-sensitivity

- implies producing results that are influenced by ripple effect.
- DISO (directly impacted set of objects) is the set of objects that are directly affected by the change
- IISO (indirectly impacted set of objects) is the set of objects that are indirectly impacted by the change is denoted by The cardinality of IISO is an indicator of ripple effect.
- Amplification is used as a measure of Ripple-sensitivity.

$$Amplification = \frac{|\ IISO\ |}{|\ DISO\ |} \longrightarrow 1$$

where | . | denotes the cardinality operator.

Impact Analysis Process - Effectiveness

2. Sharpness

- Sharpness is the ability of an impact analysis approach to avoid having to include objects in the CIS that are not needed to be changed.
- Sharpness is expressed by means of Change Rate as defined below:

  ChangeRate = | CIS | / | System |

- ChangeRate falls in the range from 0 to 1.
- For Sharpness to be high ➔ Change Rate $\ll$ 1.

Impact Analysis Process - Effectiveness

- **3. Adherence**
- **Adherence** is the ability of the approach to produce a CIS which is as close to AIS as possible.
- A small difference between CIS and AIS means that a small number of candidate objects fail to be included in the actual modification set.
- Adherence is expressed by S-Ratio as follows:

  S-Ratio = | AIS | / | CIS |

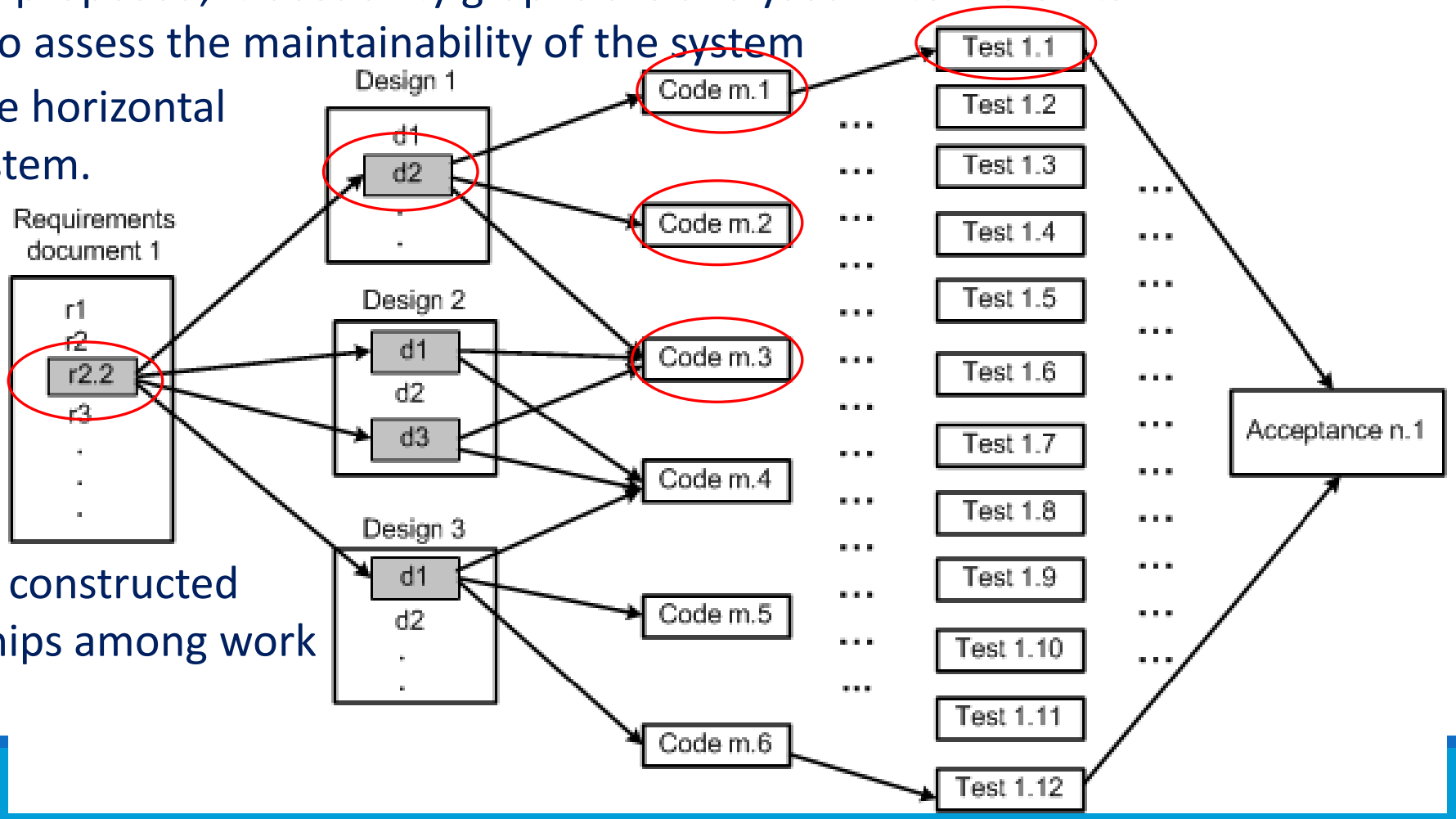
- S-Ratio takes on values in the range from 0 to 1 , Ideally *S-Ratio*= 1.

Identifying the Starting Impact Set (SIS)

- Impact analysis begins with identifying the SIS (Starting Impact Set).
- The CR specification, documentation, and source code are analyzed to find the SIS.
- It takes more efforts to map a new CR's "concepts" onto source code components (or objects).
- In the "concept assignment problem," one discovers human oriented concepts and assigns them to their realization.
- It is difficult to fully automate the concept assignment problem because programs and concepts do not occur at identical levels of abstractions, thereby necessitating human interactions

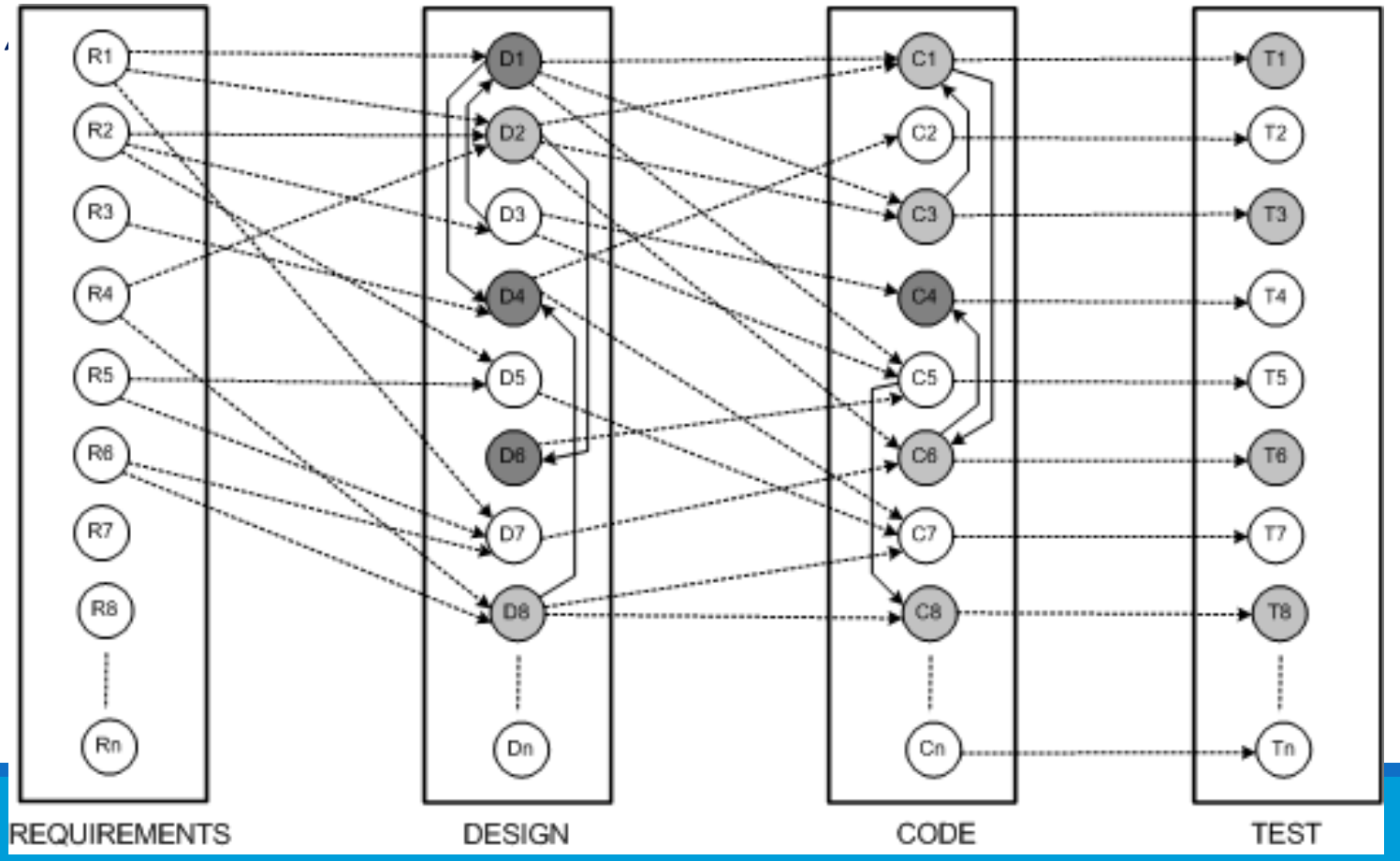
Analysis of Traceability Graph

- whenever change is proposed, traceability graphs are analysed in terms of its complexity and size to assess the maintainability of the system
- The graph shows the horizontal traceability of the system.
- The graph that is so constructed reveals the relationships among work products.



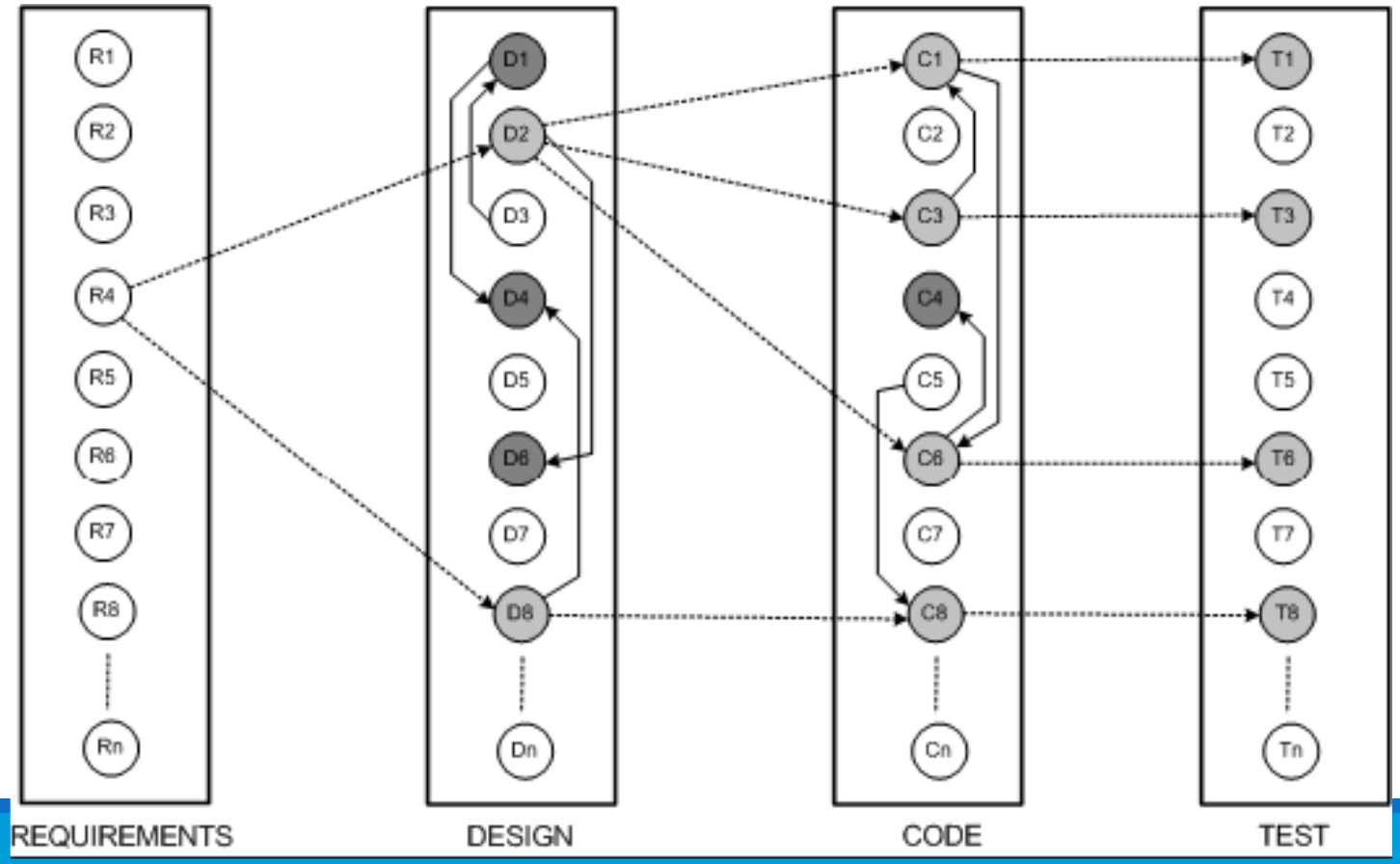
Analysis of Traceability Graph

- The graph has four categories of nodes: requirements, design, code, and test.
- The edges within a silo represent vertical traceability for the kind of work product represented by the silo.



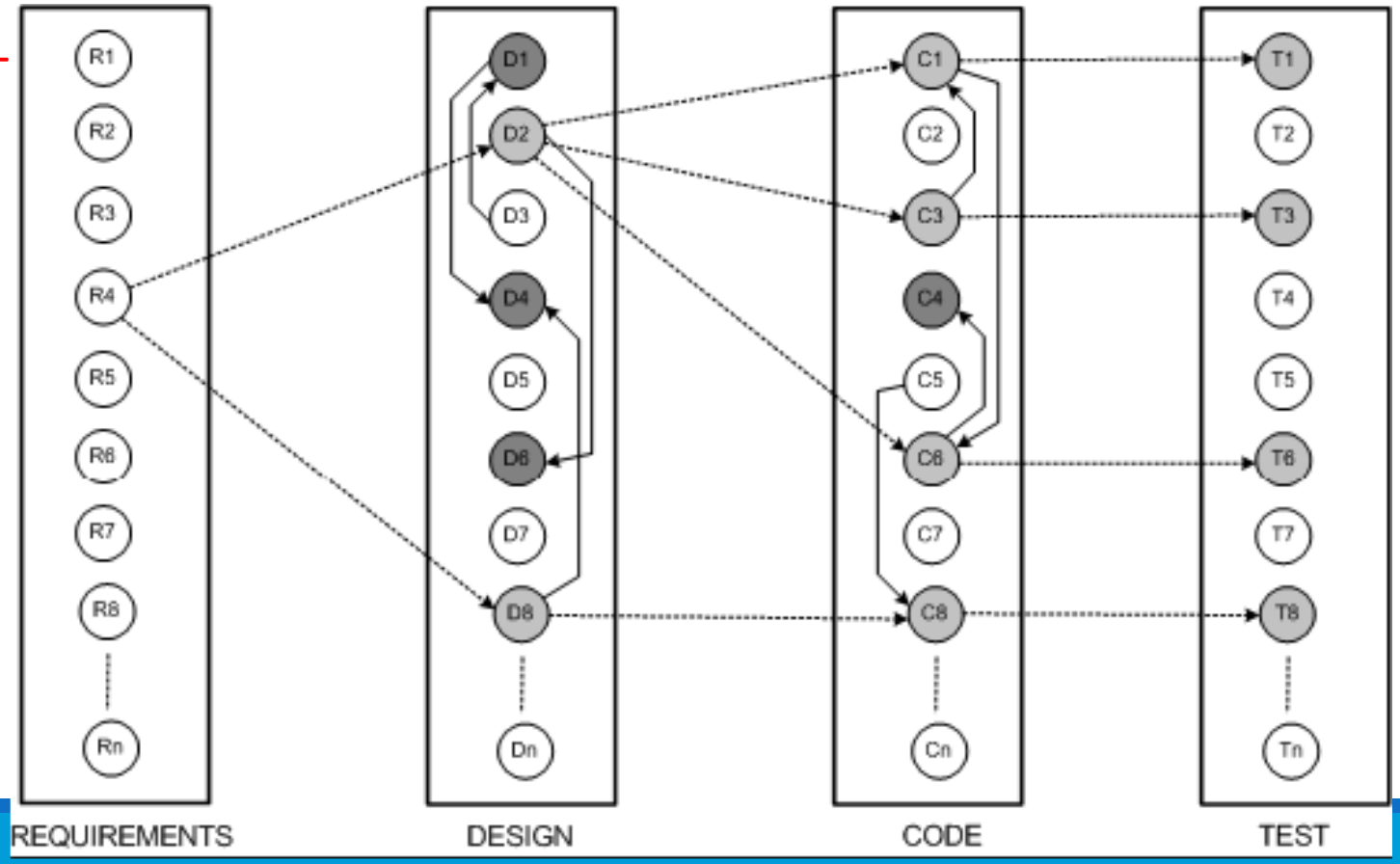
Analysis of Traceability Graph

- For a node i its in-degree **in(i)** counts the number of edges for which i is the destination node, **in(i)** denotes the number of nodes having a direct impact on i.
- The out-degree **out(i)** of node i is the number of edges for which i is the source. out(i) is a measure of the number of nodes which are likely to be modified if node i was changed.



Analysis of Traceability Graph

- node count is a measure of size
- To minimize the impact of a change, out-degrees of nodes need to be made small.
- For nodes with large out-degrees, one may partition the nodes to uniformly allocate dependencies across multiple nodes
- Low in-degrees of nodes are an indication of a good design



Analysis of Traceability Graph

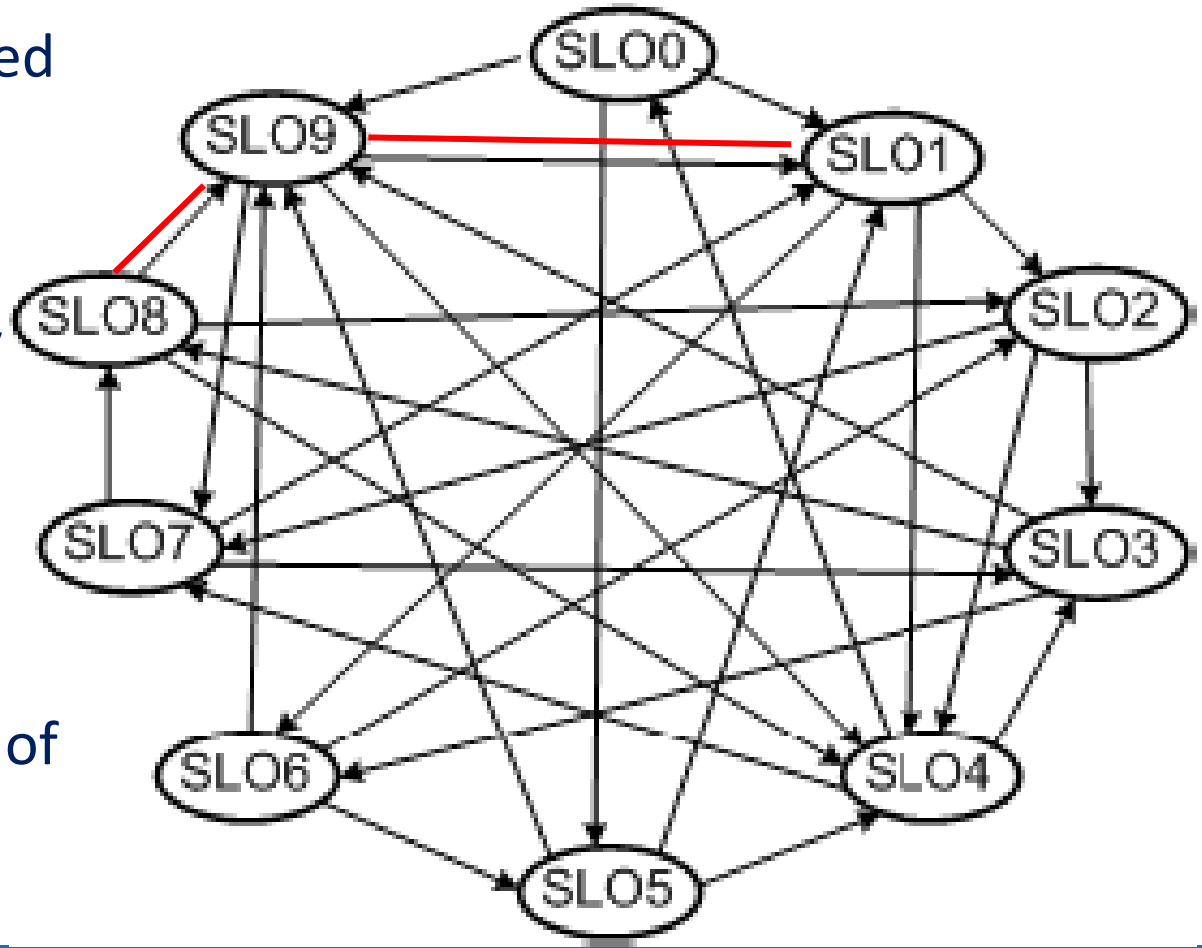
- To understand changes in horizontal traceability, it is necessary to understand:
  - the relationships among the work products.
  - how work products relate to the process as a whole.
- There exist three graphs:
  1. Relating requirements to software design.
  2. Relating software design to source code.
  3. Relating source code to tests.
- If a proposed change results in increased size or complexity of the relationship between a pair of work products, the resulting system will be more difficult to maintain

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

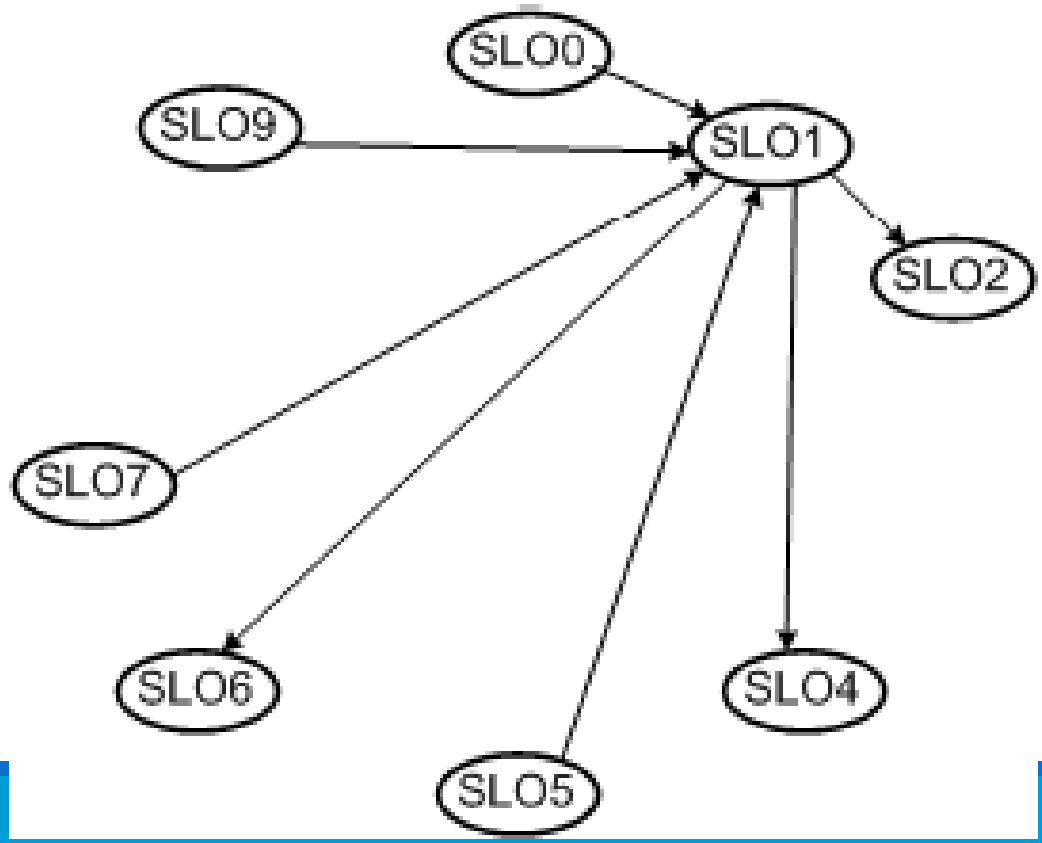
Identifying the Candidate Impact Set

- 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.

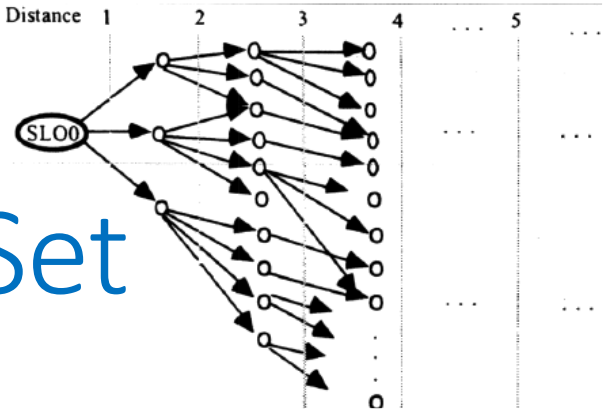
	SLO0	SLO1	SLO2	SLO3	SLO4	SLO5	SLO6	SLO7	SLO8	SLO9
SLO0		x				x				x
SLO1			x		x		x			
SLO2				x	x			x		
SLO3							x		x	x
SLO4	x			x				x		
SLO5		x			x					x
SLO6			x			x				x
SLO7		x		x					x	
SLO8			x		x					x
SLO9		x			x			x		

Identifying the Candidate Impact Set

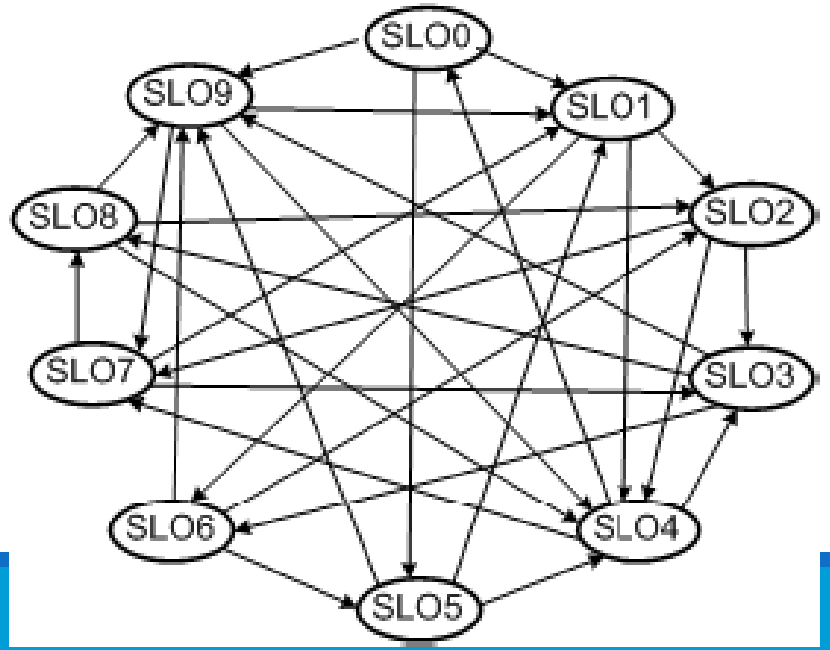
- The dense reachability matrix has the risk of over-estimating the CIS.
- To minimize the occurrences of false positives, one might consider the following two approaches:
  - Distance based approach.
  - Incremental approach.

	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



- 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.



	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	

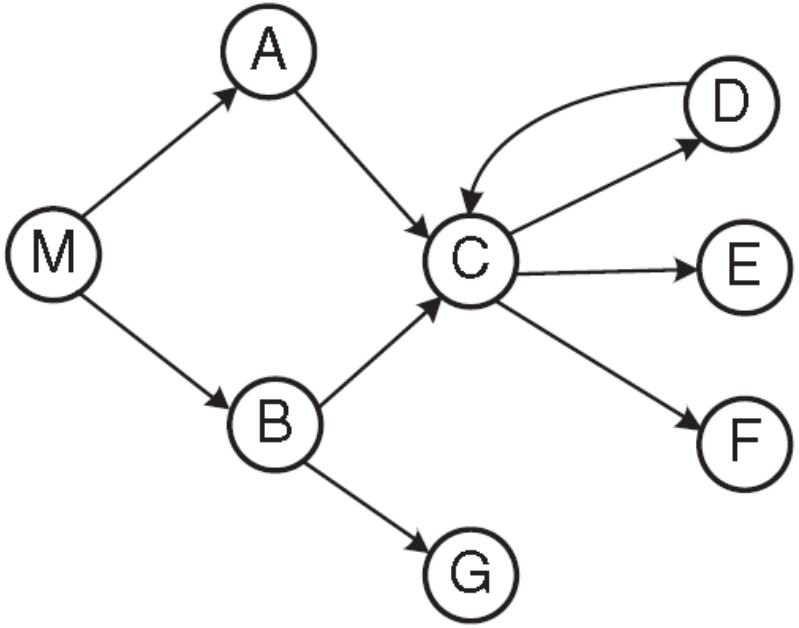
Dependency-based Impact Analysis

- 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.

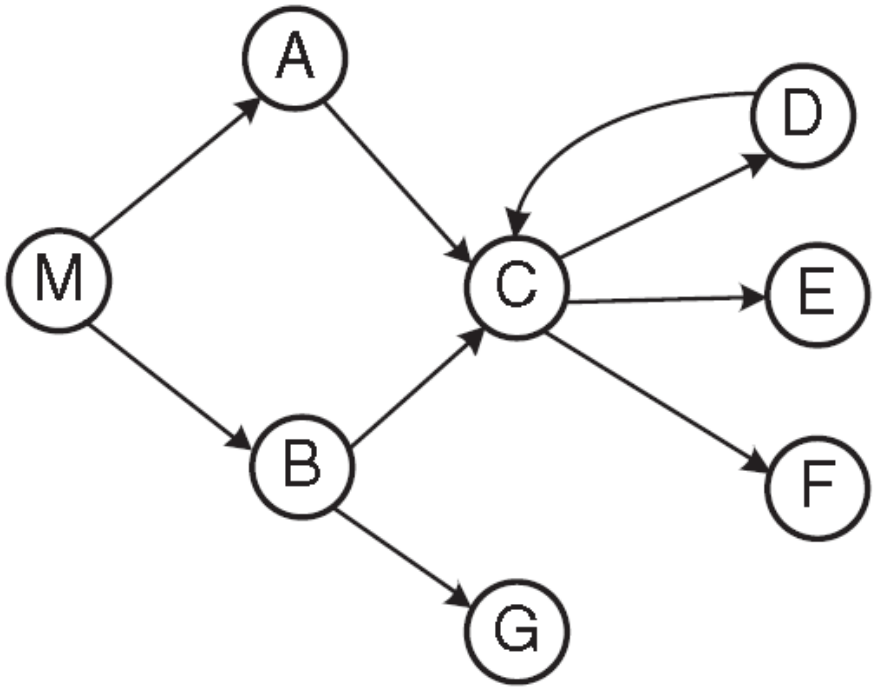
Dependency-based Impact Analysis - Call Graph

- 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.

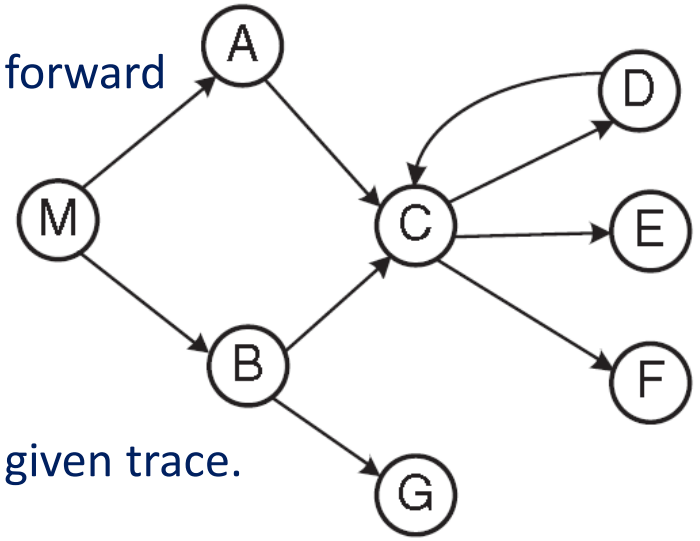


Dependency-based Impact Analysis - Call Graph

- 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 .

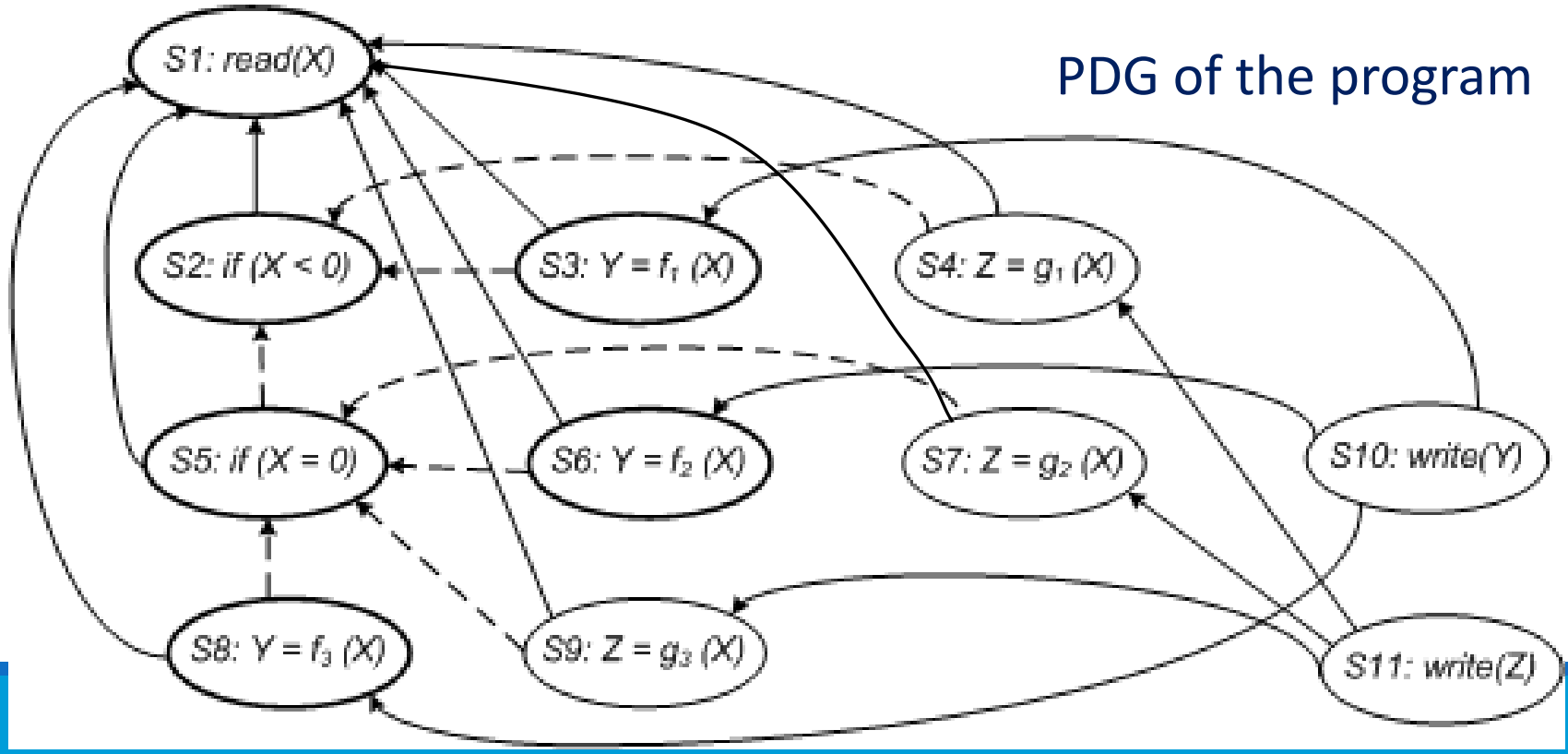
Dependency-based Impact Analysis - Program Dependency Graph

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



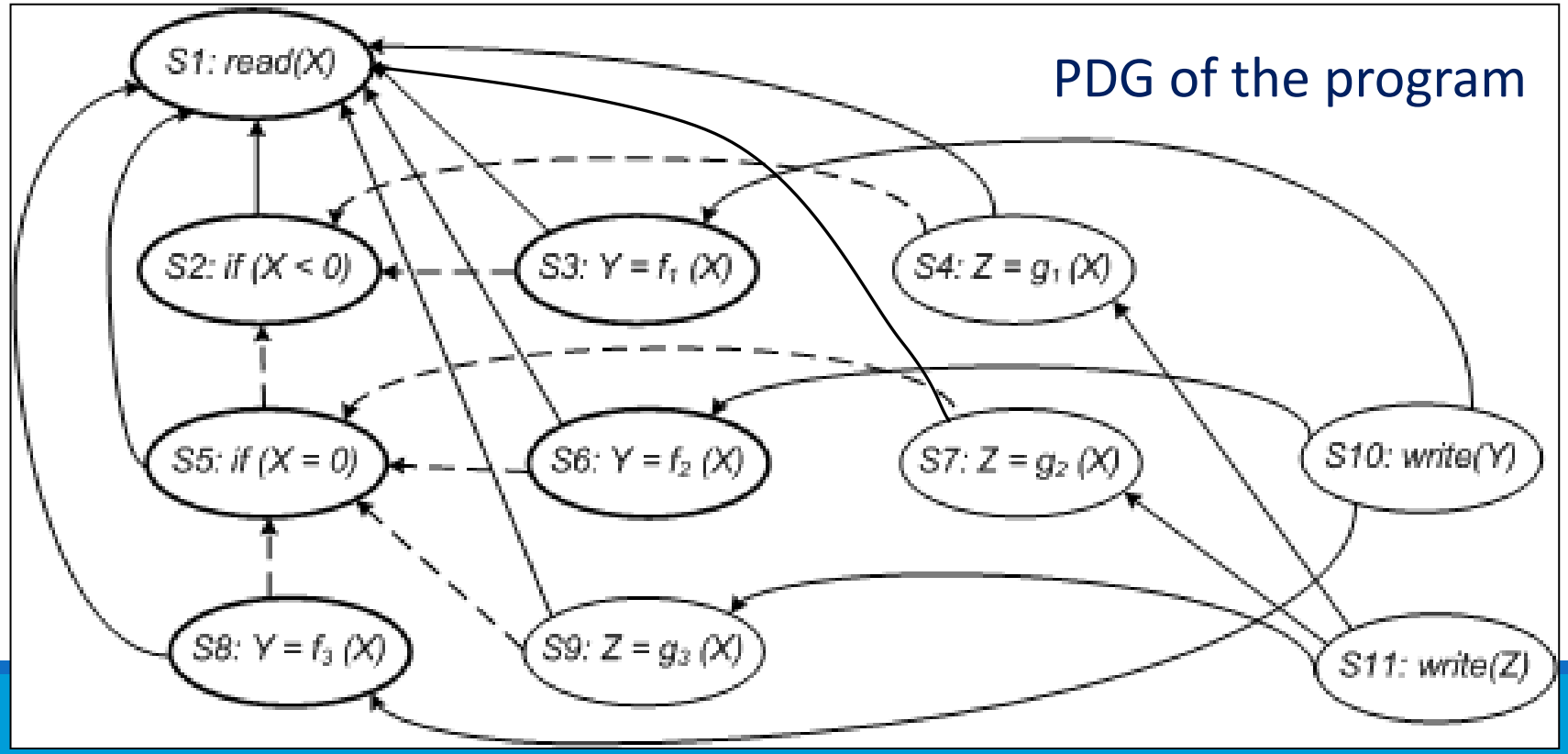
Dependency-based Impact Analysis - Program Dependency Graph

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.

PDG of the program

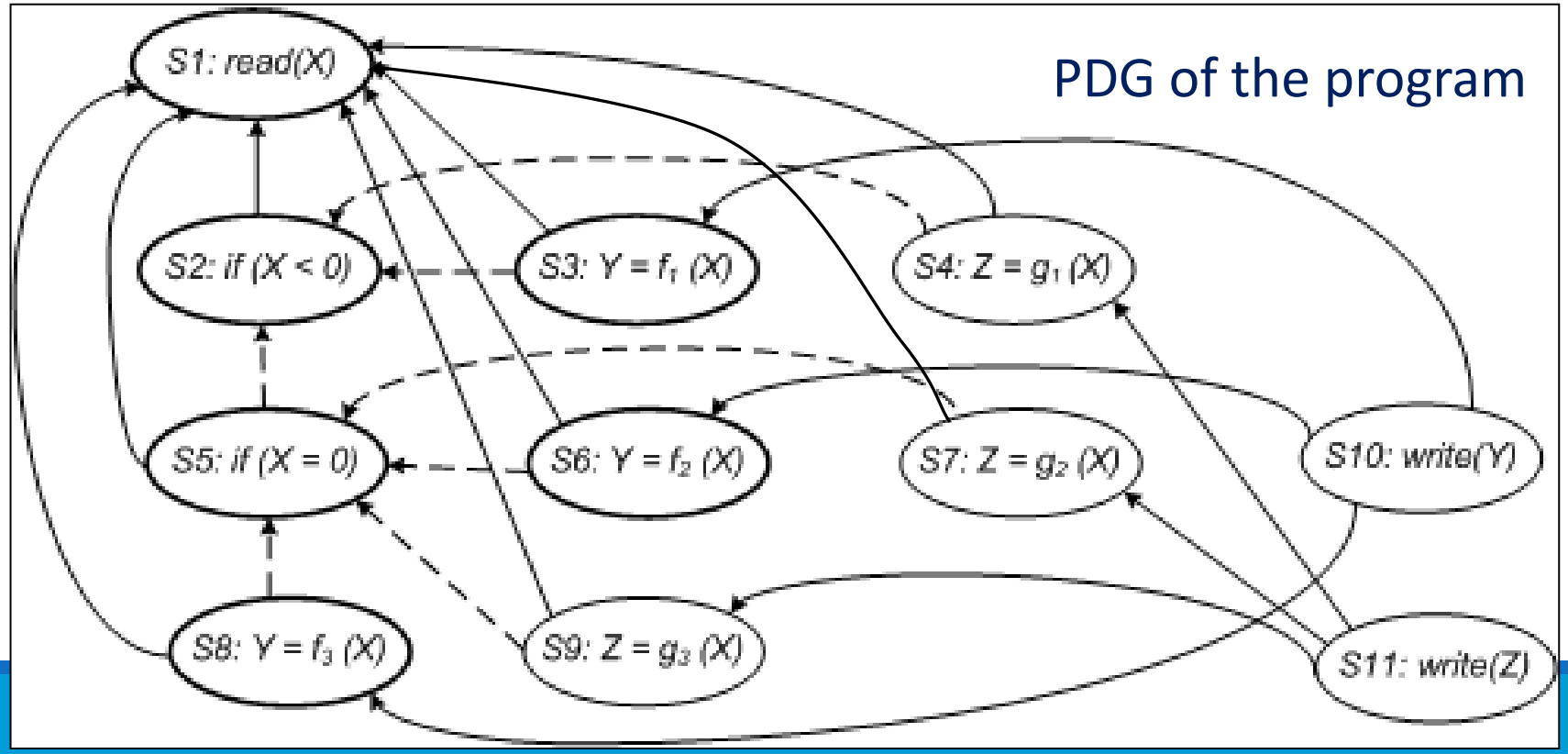


Dependency-based Impact Analysis - Program Dependency Graph

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.



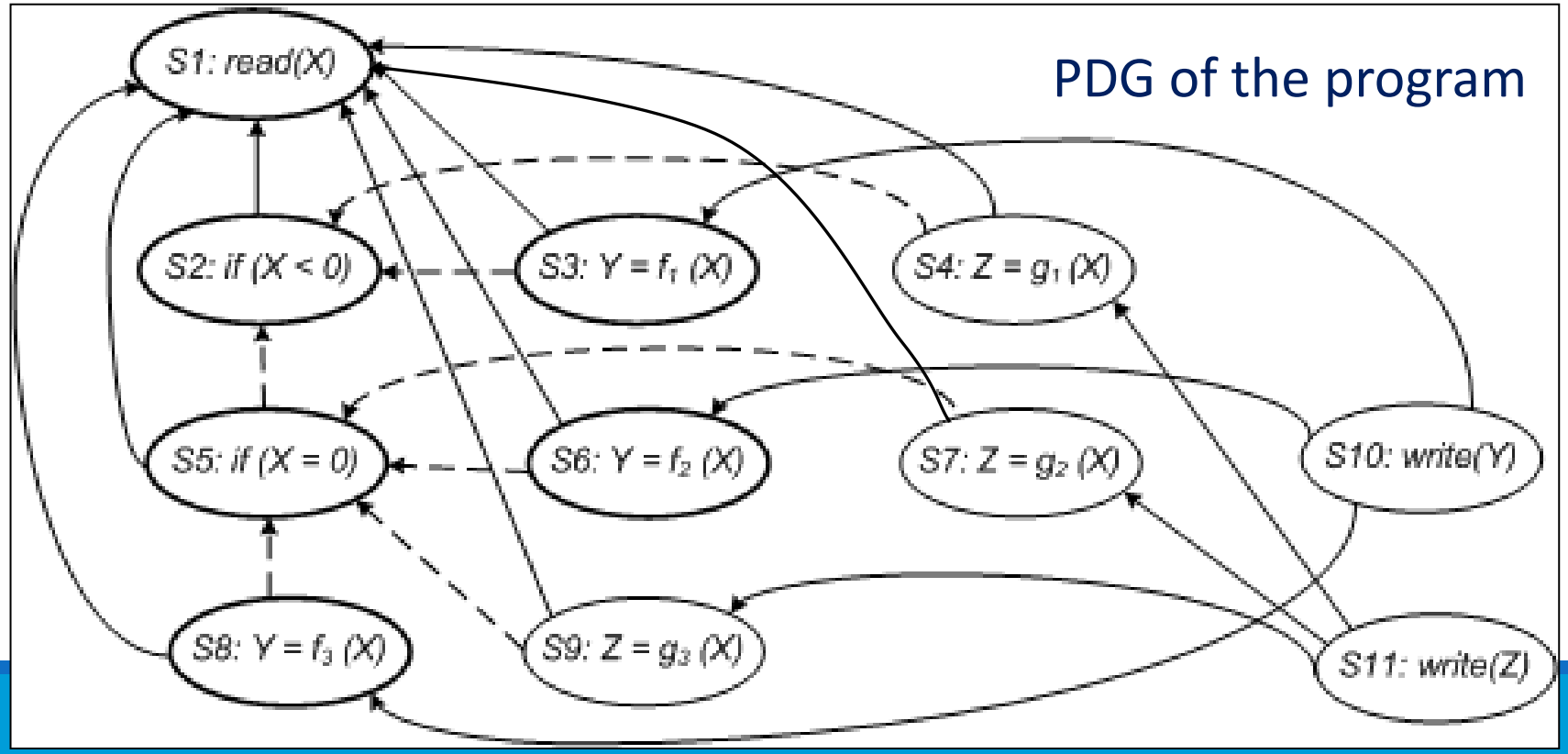
Dependency-based Impact Analysis - Program Dependency Graph

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.

PDG of the program



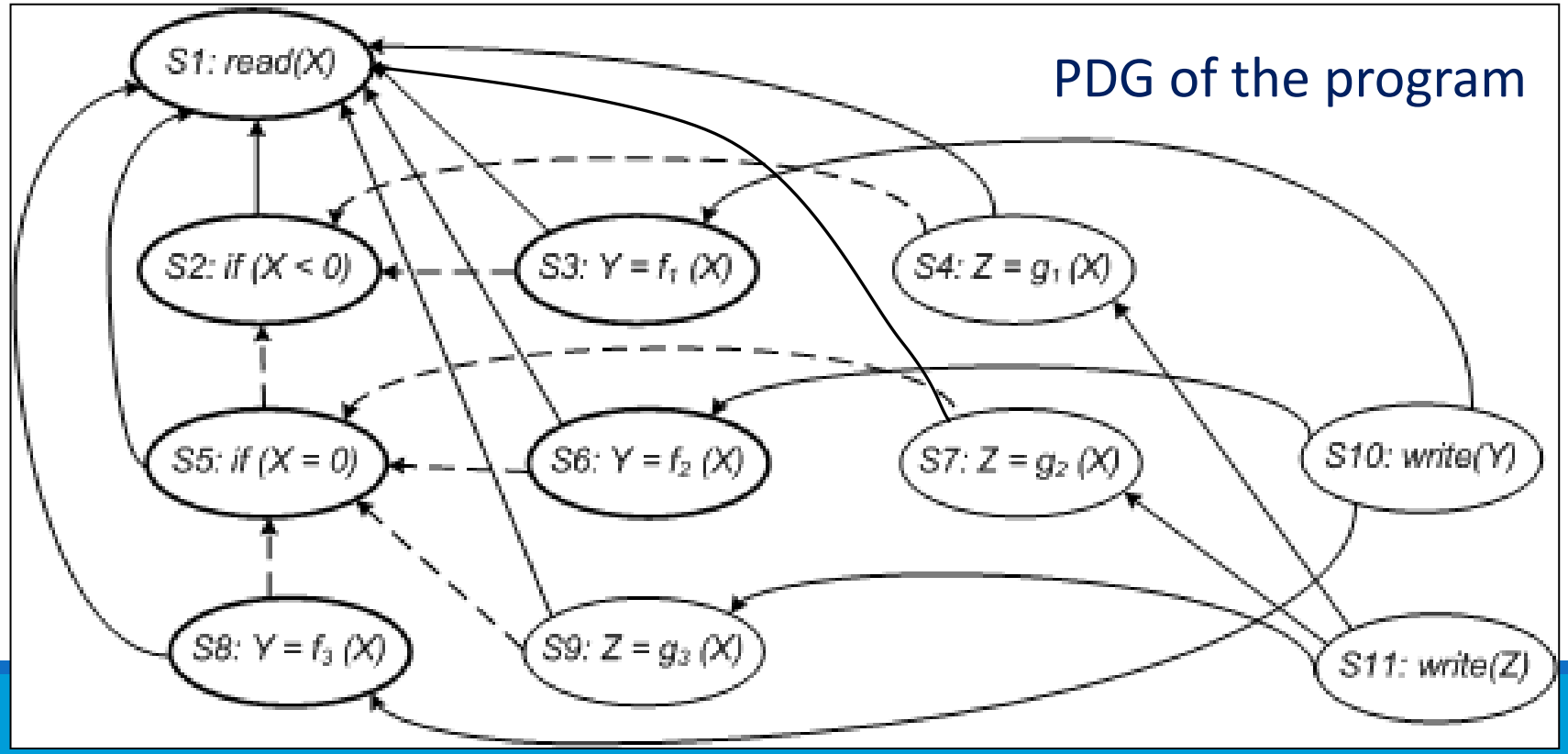
Dependency-based Impact Analysis - Program Dependency Graph

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.

PDG of the program



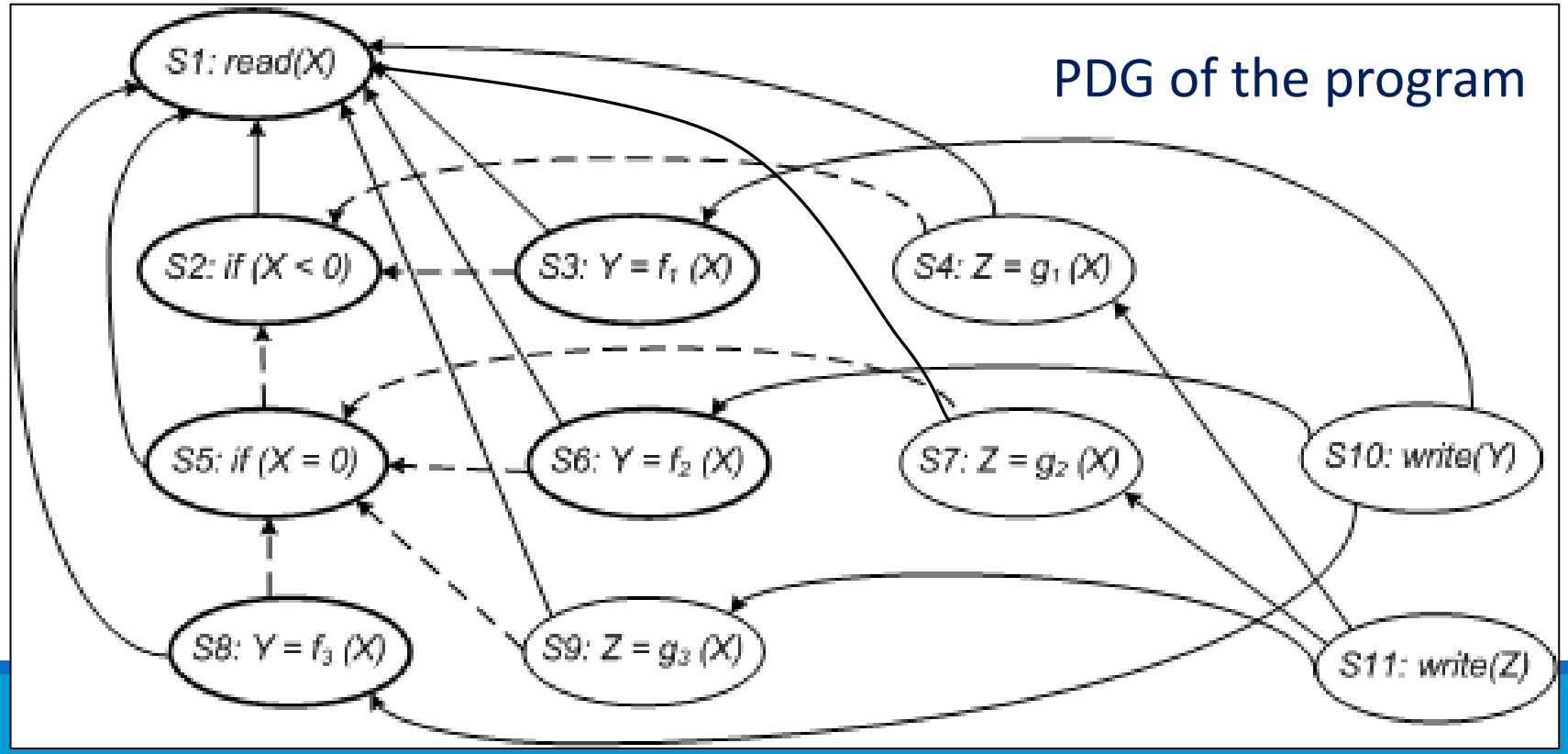
Dependency-based Impact Analysis - Program Dependency Graph

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.

PDG of the program



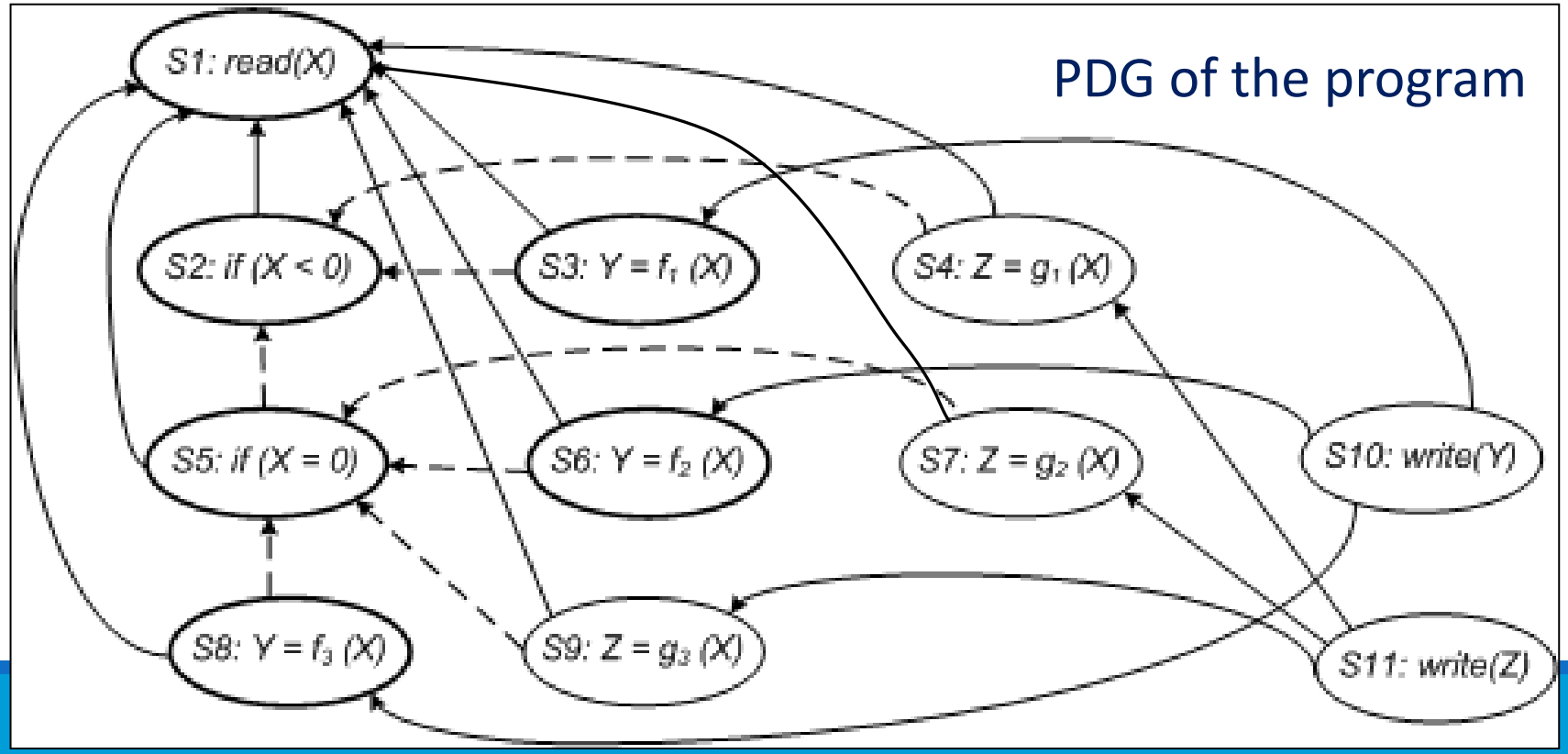
Dependency-based Impact Analysis - Program Dependency Graph

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.

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.

PDG of the program

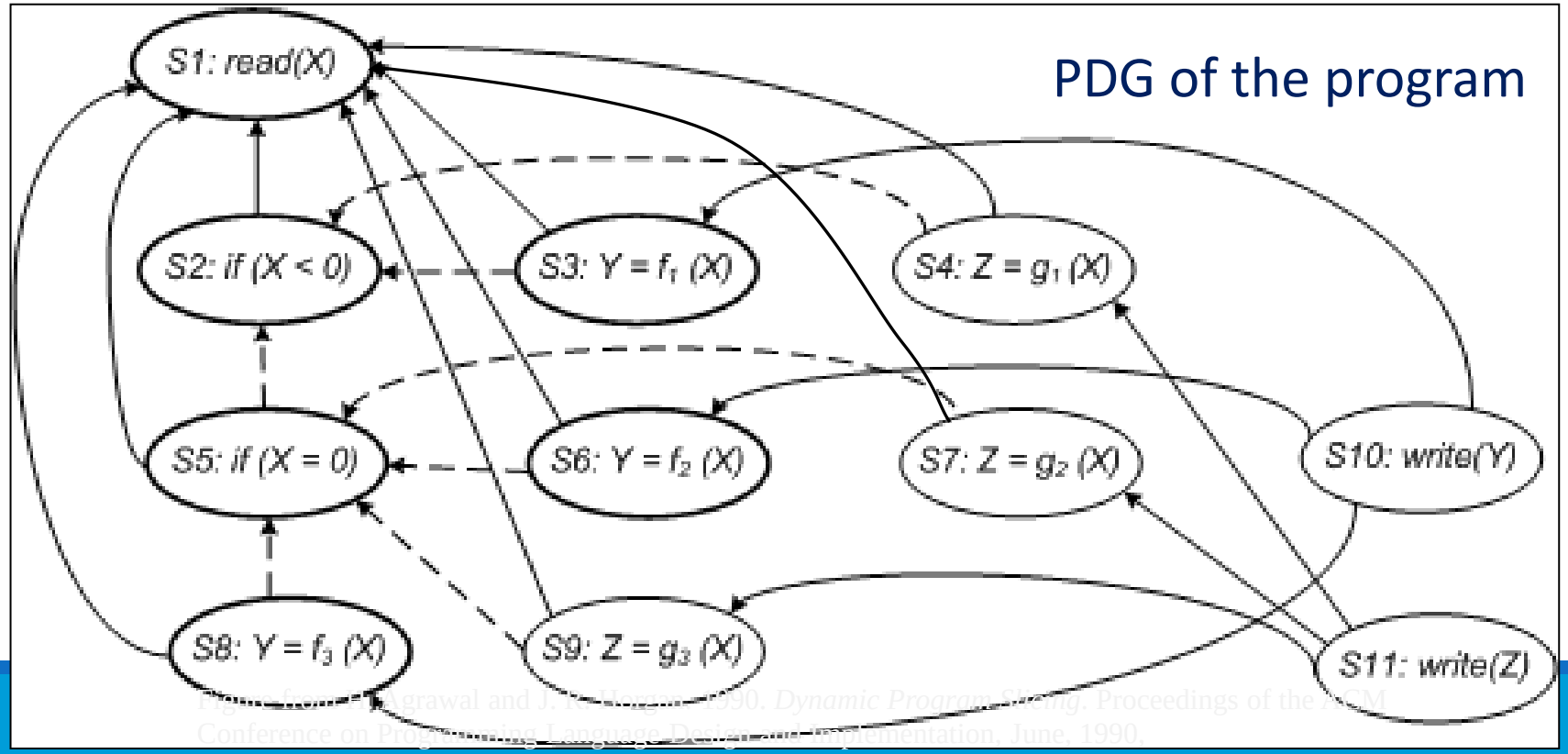


Figure from: R. Agrawal and J. R. Horgan. 1990. *Dynamic Program Slicing.* Proceedings of the ACM Conference on Programming Language Design and Implementation, June, 1990.

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

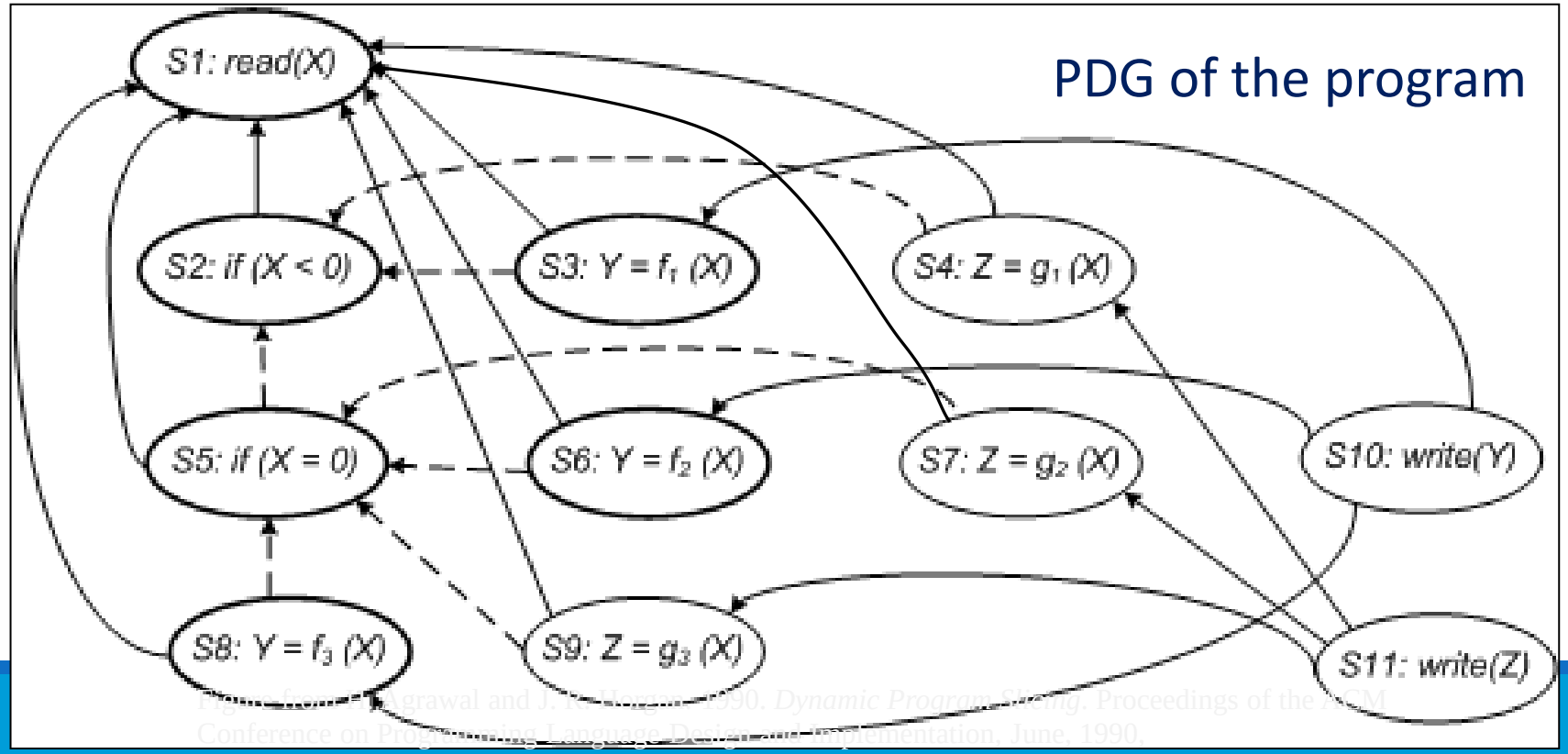


Figure from: R. Agrawal and J. R. Horgan, 1990. *Dynamic Program Slicing.* Proceedings of the ACM Conference on Programming Language Design and Implementation, June, 1990.

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

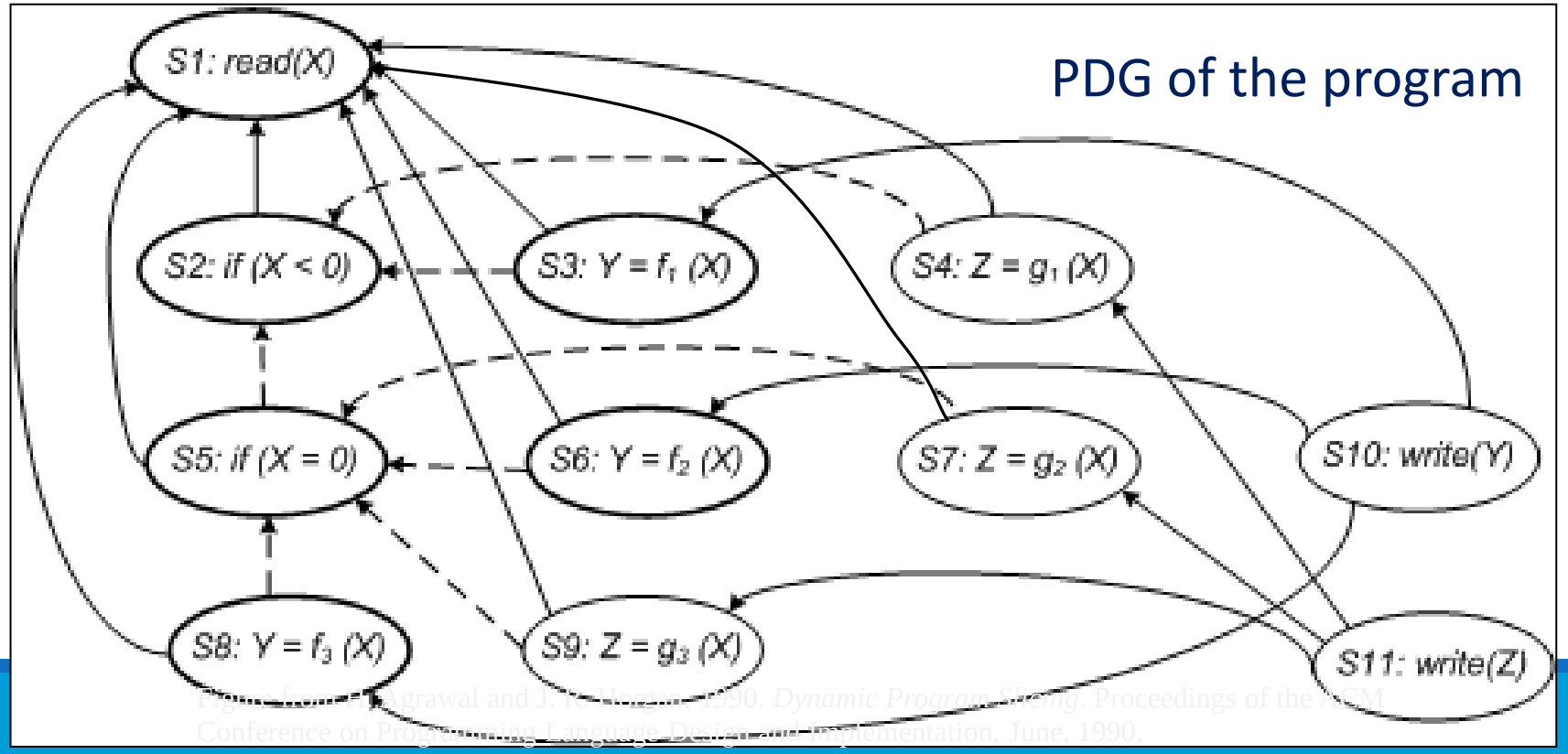


Figure from: R. Agrawal and J. R. Horgan. 1990. *Dynamic Program Slicing.* Proceedings of the ACM Conference on Programming Language Design and Implementation, June, 1990.

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

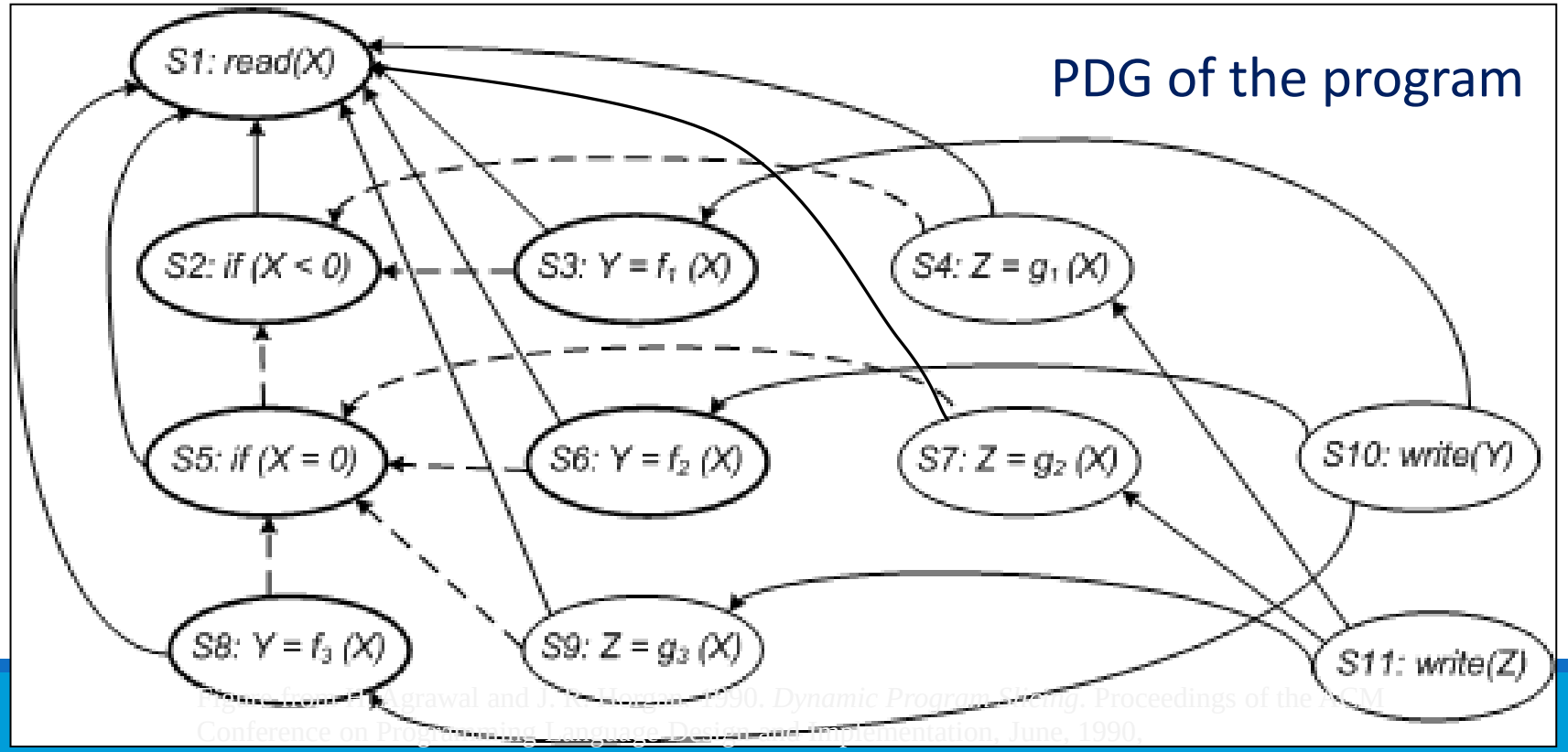


Figure from: R. Agrawal and J. R. Horgan. 1990. *Dynamic Program Slicing.* Proceedings of the ACM Conference on Programming Language Design and Implementation, June, 1990.

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);
```

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

PDG of the program

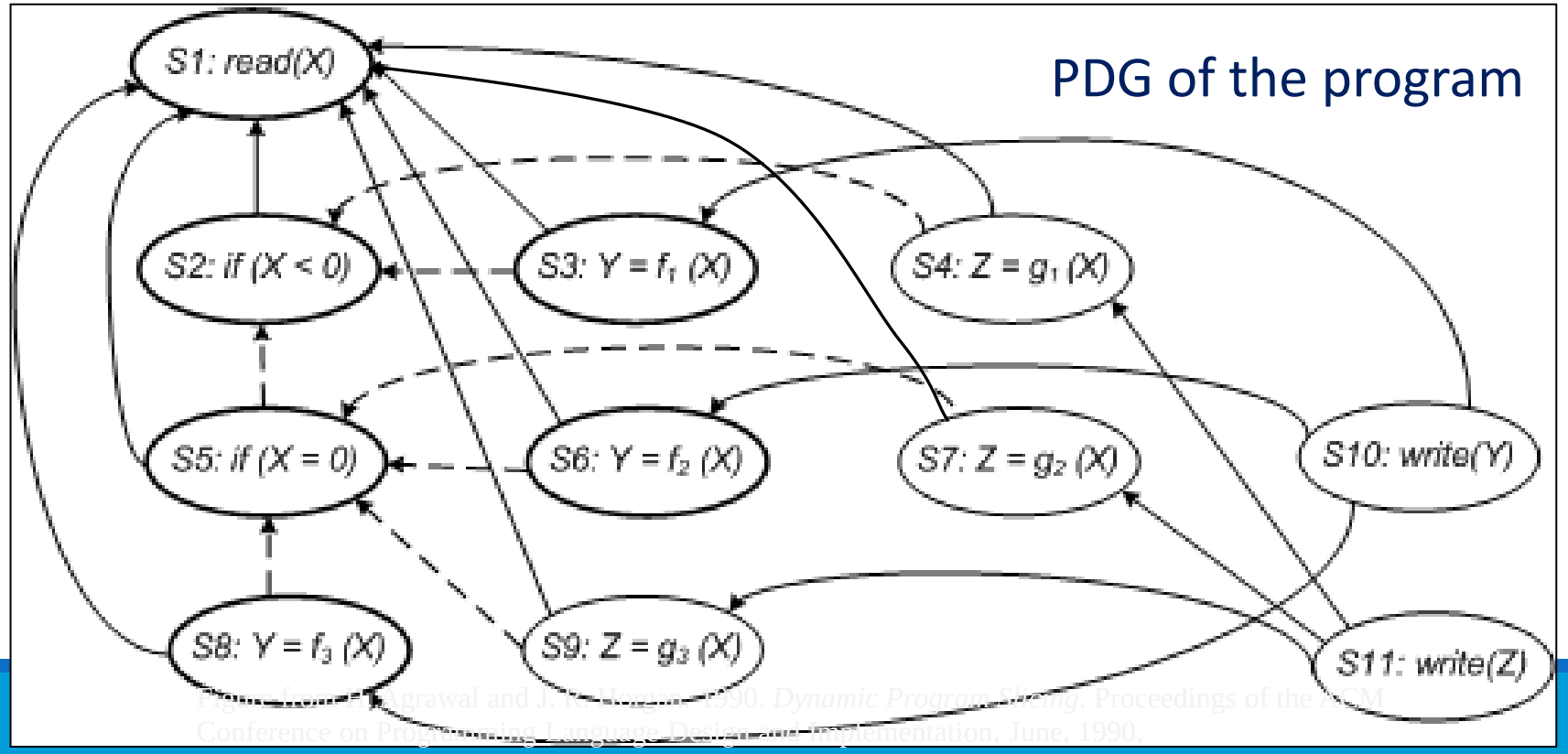
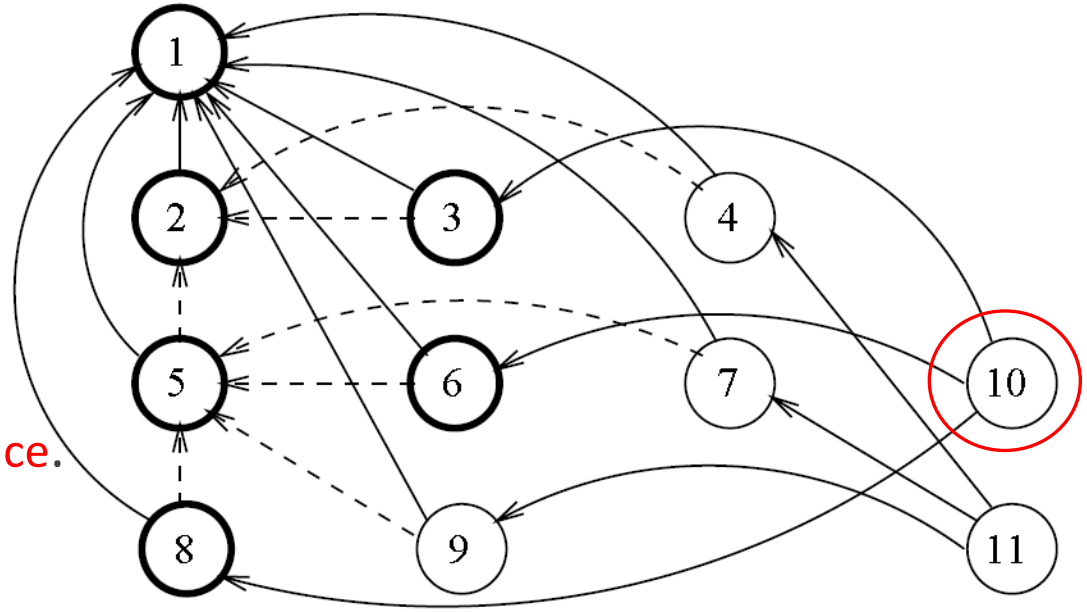


Figure from: R. Agrawal and J. R. Horgan, 1990. *Dynamic Program Slicing.* Proceedings of the ACM Conference on Programming Language Design and Implementation, June, 1990.

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

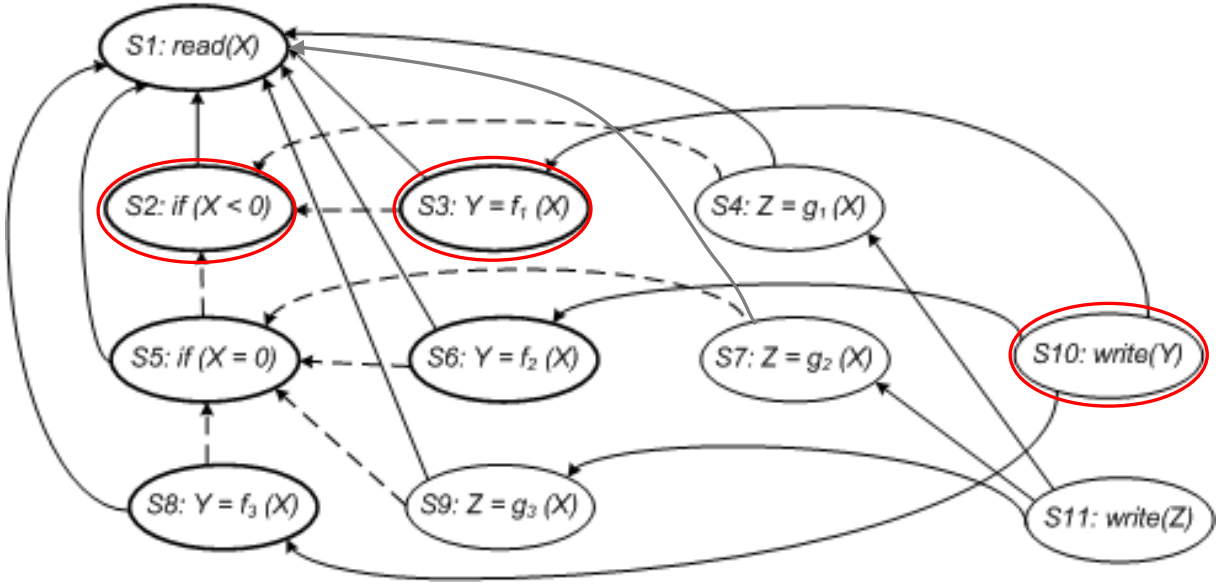
- A dynamic slice is more useful in localizing the defect than the static slice.
- Only one of the three assignment statements, S3, S6, or S8, may be executed for any input value of X.
- Consider the input value −1 for the variable X.
- For −1 as the value of X, only S3 is executed.
- Therefore, with 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);
```

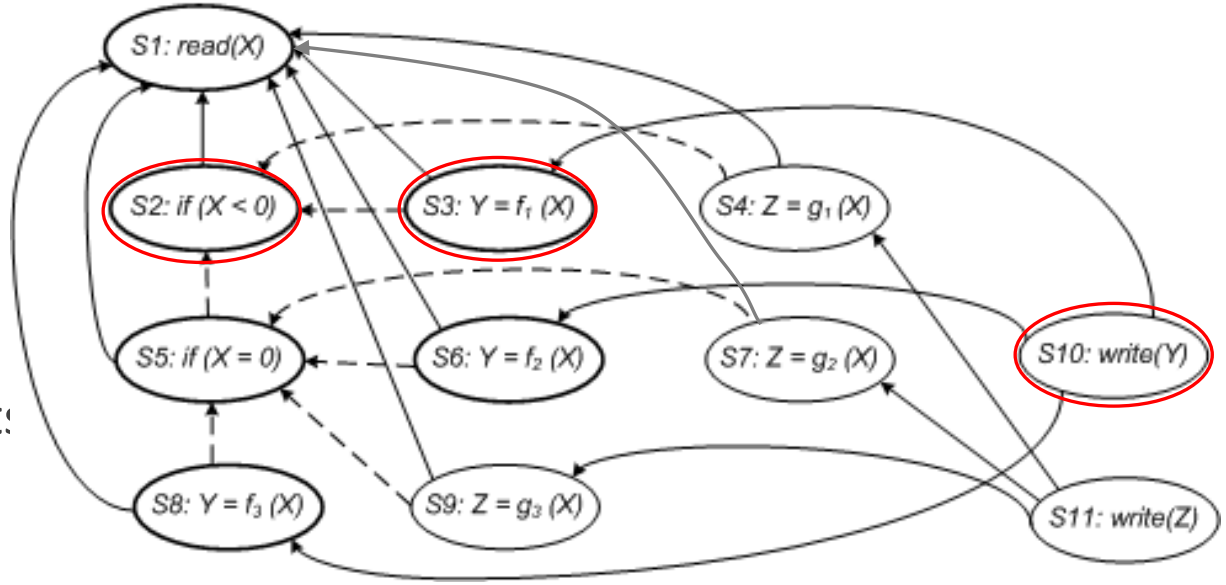
Dependency-based Impact Analysis -Program Dependency Graph - Dynamic Slice

- 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

Identifying the Starting Impact Set (SIS)

- The software reconnaissance methodology proposed by Wilde and Scully is based on the idea that some programming concepts are selectable, because their execution depends on a specific input sequence.
- Selectable program concepts are known as features. By executing a program twice, one can often find the source code implementing the features:

(i) execute the program once with a feature and once without the feature.

(ii) mark portions of the source code that were executed the first time but not the second time.

(iii) the marked code are likely to be in or close to the code implementing the feature.

Identifying the Starting Impact Set (SIS)

- Chen and Rajlich proposed a dependency graph based feature location method for C programs.
- The component dependency graph is searched, generally beginning at the main().
- Functions are chosen one at a time for a visit.
- The maintenance personnel reads the documentation, code, and dependency graph to comprehend the component before deciding if the component is related to the feature under consideration.