Impact Analysis

© Tripathy & Naik

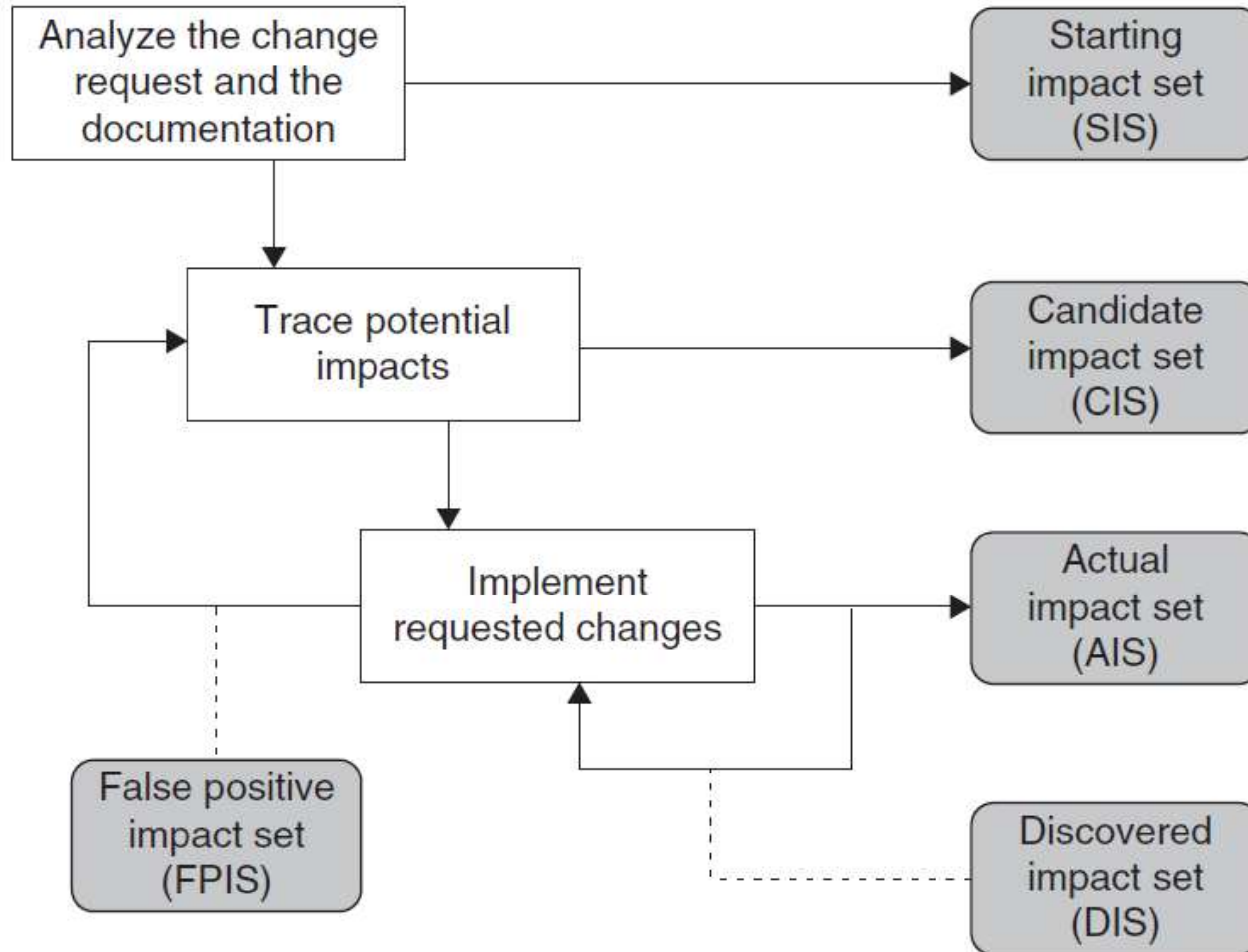
6.1 General Idea

- The maintenance process is started by performing impact analysis.
- Impact analysis basically means 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 determine the portions of the software that need to be subjected to regression testing after a change is effected.

© Tripathy & Naik

6.2 Impact Analysis Process

Figure 6.1 Impact analysis process ©IEEE, 2008



© Tripathy & Naik

6.2 Impact Analysis Process

- **Starting Impact Set (SIS):** The initial set of objects (or components) presumed to be impacted by a software CR is called SIS.
- **Candidate Impact Set (CIS):** The set of objects (or components) estimated to be impacted according to a certain impact analysis approach is called CIS.
- **Discovered Impact Set (DIS):** DIS is defined as the set of new objects (or components), not contained in CIS, discovered to be impacted while implementing a CR. DIS is also called **False Negative Impact Set (FNIS)**
- **Actual Impact Set (AIS):** The set of objects (or components) actually changed as a result of performing a CR is denoted by AIS.
- **False Positive Impact Set (FPIS):** FPIS is defined as the set of objects (or components) estimated to be impacted by an implementation of a CR but not actually impacted by the CR. Precisely, FPIS = (CIS U DIS) \ AIS.

  where U denotes set union and \ denotes set difference.
- 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.

© Tripathy & Naik

6.2 Impact Analysis Process

Two traditional information retrieval metrics:

- **Recall:** It measures the degree the CIS cover the real changes and it is computed as the ratio of |CIS ∩ AIS| to |AIS|.
  - The value of recall is 1 when DIS is empty.
- **Precision:** It represents the fraction of candidate impacts that are actually impacted, and it is computed as the ratio of |CIS ∩ AIS| to |CIS|.
  - For an empty FPIS set, the value of precision is 1.
- Note that if AIS is equal to CIS, both **recall** and **precision** are computed to be equal to 1.

© Tripathy & Naik

Knowledge

University of Waterloo

6.2.1 Identifying the Starting Impact Set (SIS)

- Impact analysis begins with identifying the SIS.
- The CR specification, documentation, and source code are analyzed to find the SIS.
- This step is also called concept location or feature location, which is the activity of identifying initial location in the source code that implements functionality in a software system.
- Programmers use feature location to find where in the source code the initial change needs to be made.

© Tripathy & Naik

6.2.1 Identifying the Starting Impact Set (SIS)

- There are several methods to identify concepts, or features, in source code.
- The "grep" pattern matching utility available on most Unix systems and similar search tools are commonly used by programmers.
- The technique often fails when the concepts are hidden in the source code, or when the programmer fails to guess the program identifiers.

© Tripathy & Naik

6.2.1 Identifying the Starting Impact Set (SIS)

- Another approach 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.

© Tripathy & Naik

6.2.1 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 C functions are successively explored to find and understand all the components related to the given feature.
- 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.

© Tripathy & Naik

6.2.2 Analysis of Traceability Graph

- Software maintenance personnel may choose to execute the CR differently, or they may not execute it at all, if the complexity and/or size of the traceability graph increases as a result of making the proposed change.

- Whenever change is proposed, it is necessary to analyze the traceability graphs in terms of its complexity and size to assess the maintainability of the system.

© Tripathy & Naik

6.2.2 Analysis of Traceability Graph

- The graph shows the horizontal traceability of the system.
- The graph that is so constructed reveals the relationships among work products.

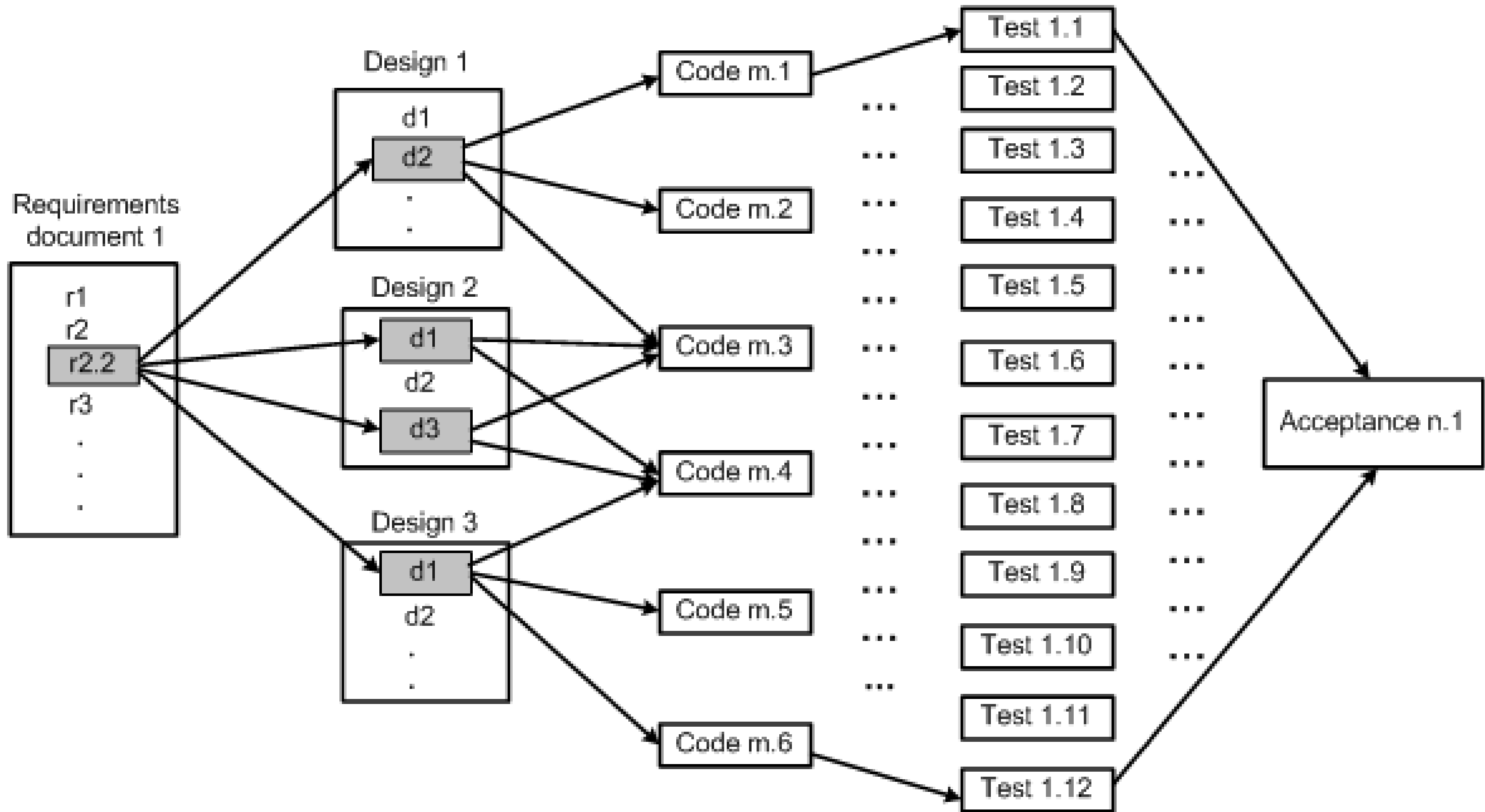


Figure 6.2 Traceability in software work products ©IEEE, 1991

© Tripathy & Naik

6.2.2 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.

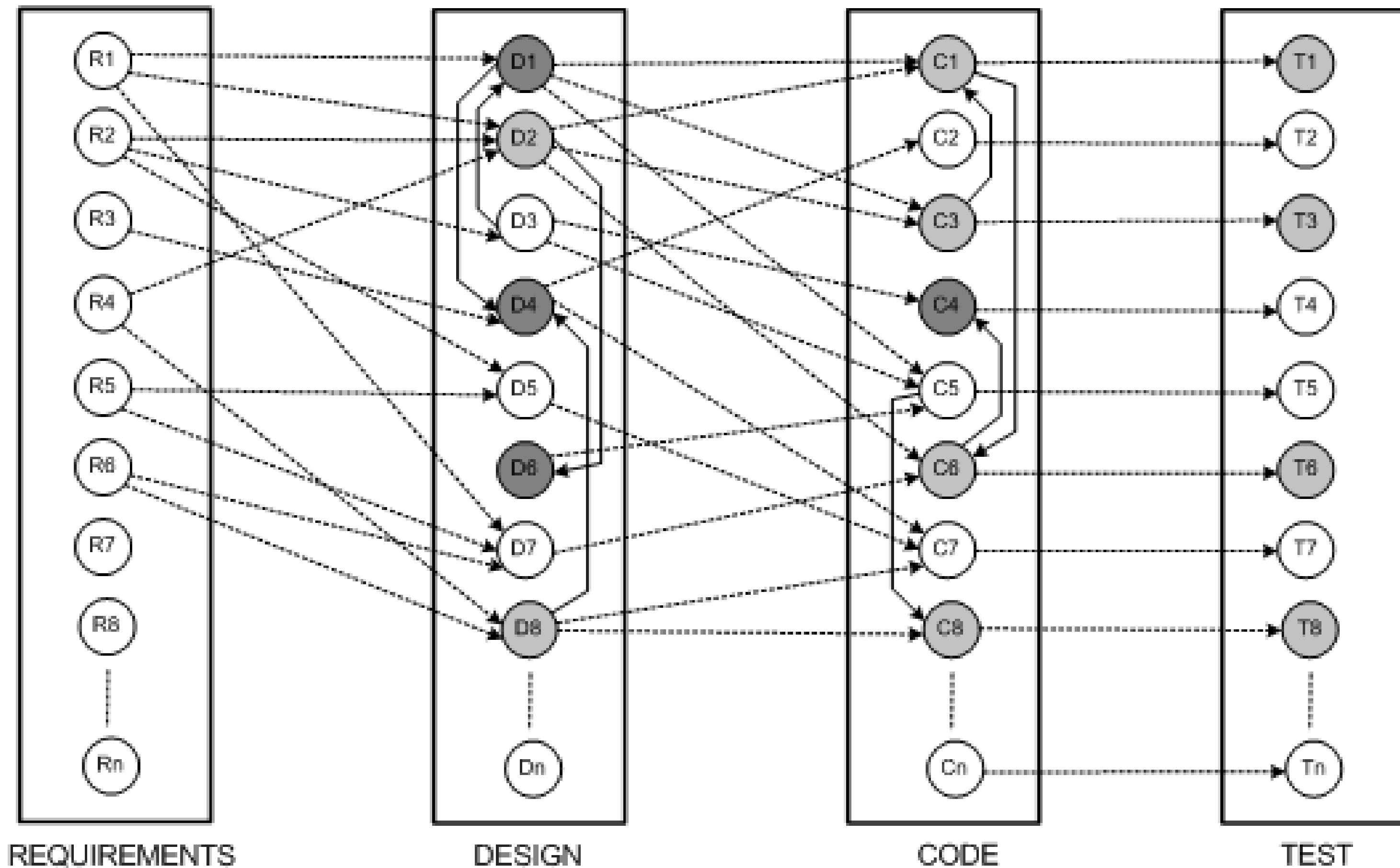


Figure 6.3 Underlying graph for maintenance ©IEEE, 1991

© Tripathy & Naik

6.2.2 Analysis of Traceability Graph

- If some changes are made to requirement object "R4," the results of horizontal traceability and vertical traceability are shown in Figure 6.4.
- The horizontally traced objects have been shown as lightly shaded circles, whereas the vertically traced objects have darkly shaded circles.

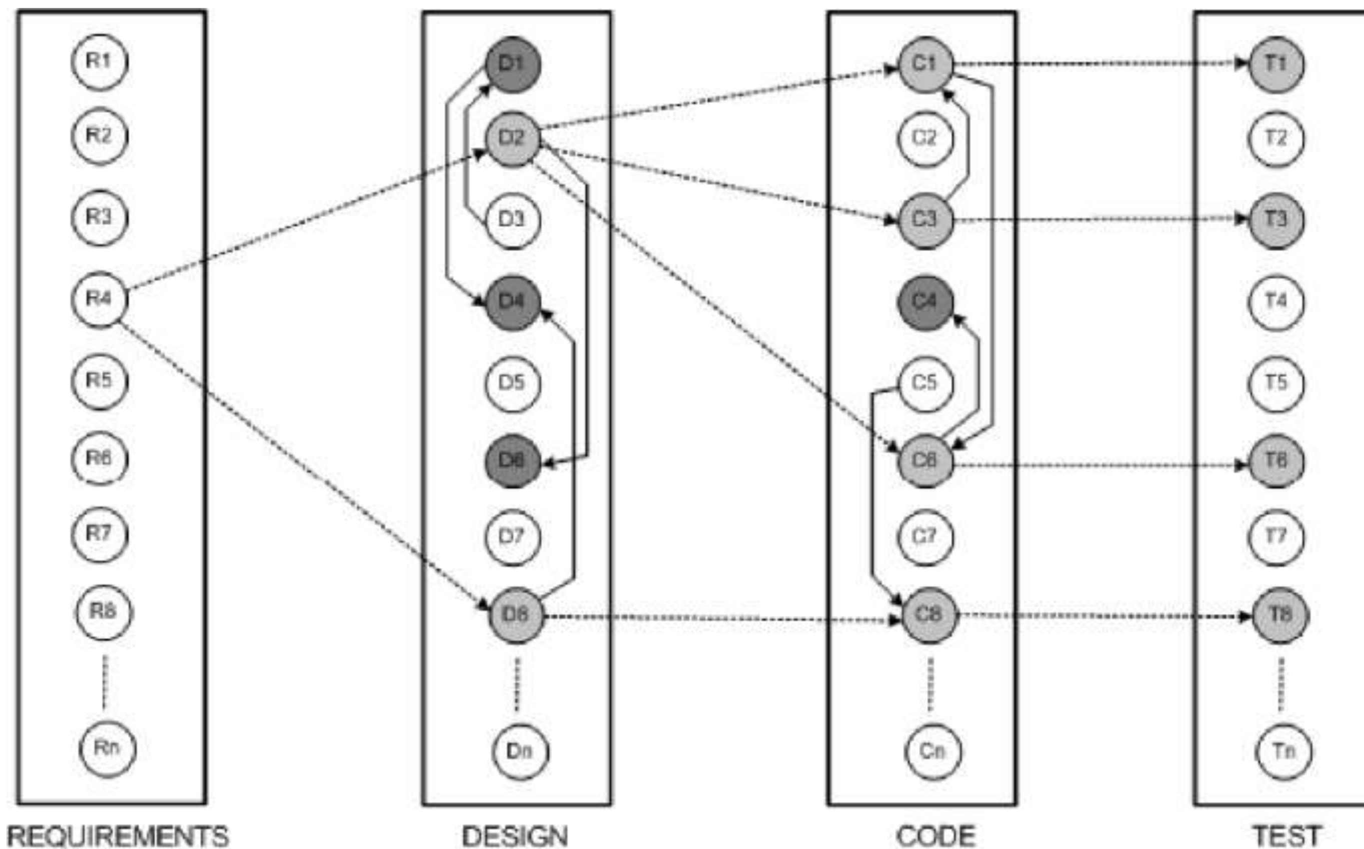


Figure 6.4 Determine work product impact ©IEEE, 1991

© Tripathy & Naik

6.2.2 Analysis of Traceability Graph

- For a node i in a graph, its in-degree in(i) counts the number of edges for which i is the destination node, and in(i) denotes the number of nodes having a direct impact on i.

- Similarly, the out-degree of node i, denoted by out(i), is the number of edges for which i is the source.

- Node i being changed, out(i) is a measure of the number of nodes which are likely to be modified.

© Tripathy & Naik

6.2.3 Identifying the Candidate Impact Set

- A CIS is identified in the next step of the impact analysis process.
- The SIS is augmented with software lifecycle objects (SLOs) that are likely to change because of changes in the elements of the SIS.
- Changes in one part of the software system may have direct impacts or indirect impacts on other parts.
- Both direct impact and indirect impact are explained in the following.
  - **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.

© Tripathy & Naik

6.2.3 Identifying the Candidate Impact Set

- let us consider the directed graph in Figure 6.5 with ten SLOs.
- Each SLO represents a software artifact connected to other artifacts.
- Dependencies among SLOs are represented by arrows.
- In the figure, SLO1 has an indirect impact from SLO8 and a direct impact from SLO9.
- The in-degree of a node i reflects the number of known nodes that depend on i.

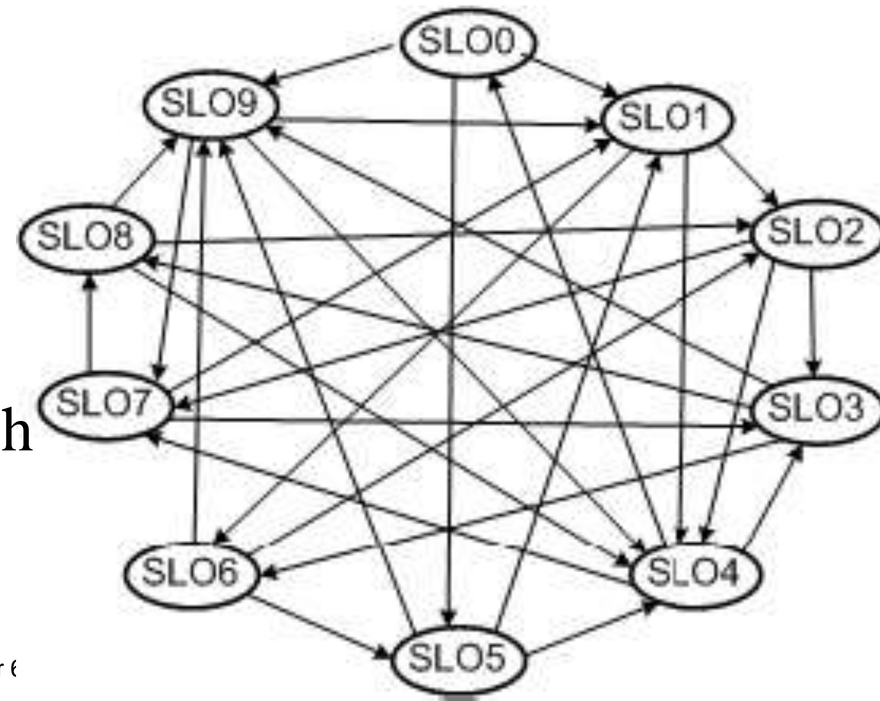


Figure 6.5 Simple directed graph of SLOs ©IEEE, 2002

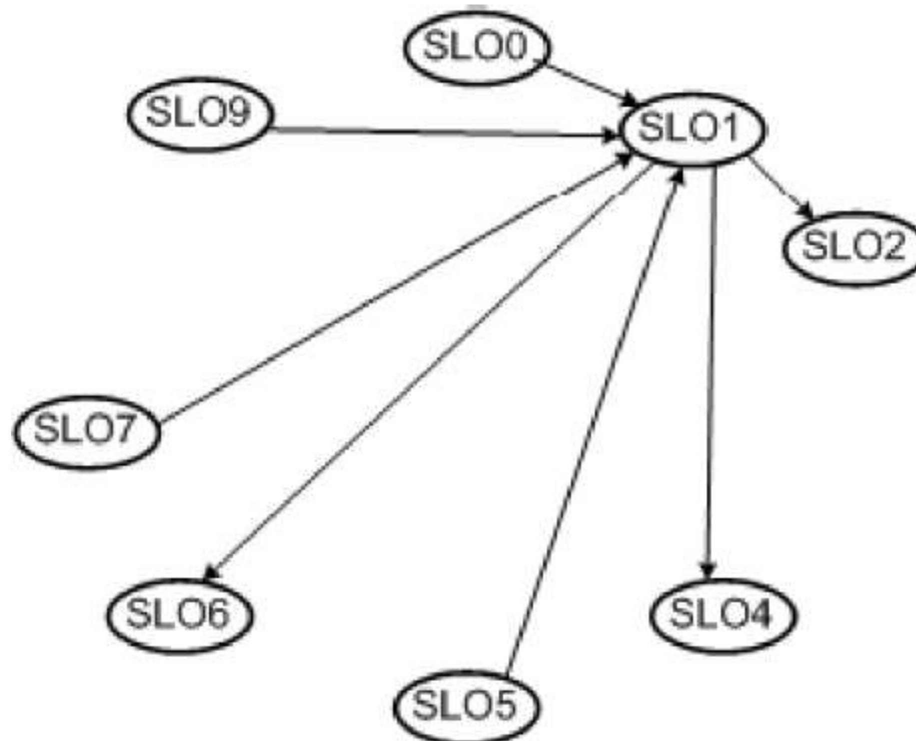
Knowledge

6.2.3 Identifying the Candidate Impact Set

University of Waterloo

- Figure 6.6 shows the four nodes – SLO0, SLO5, SLO7 and SLO9 – that are dependent on SLO1, and the in-degree of SLO1 is four.
- In addition, the out-degree of SLO1 is three.

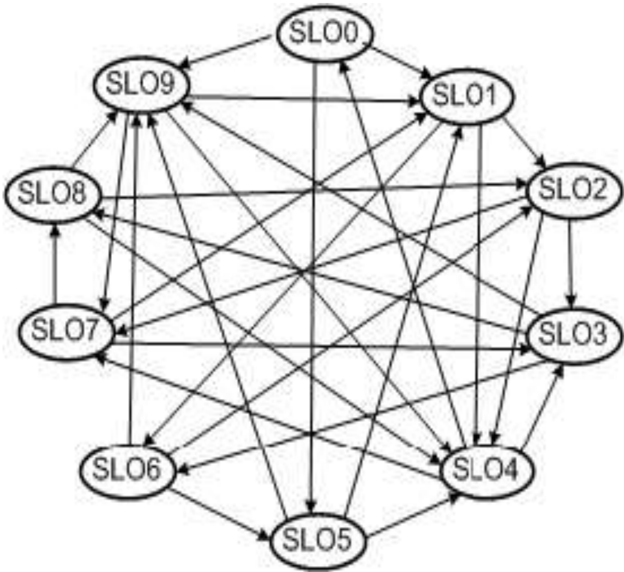
Figure 6.6 In-degree and out-degree of SLO1 ©IEEE, 2002



© Tripathy & Naik

6.2.3 Identifying the Candidate Impact Set

- The connectivity matrix of Table 6.1 is constructed by considering the SLOs and the relationships shown in Figure 6.5.
- 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		

Table 6.1: Relationships represented by a connectivity matrix [13] (©[2002] IEEE).

© Tripathy & Naik

6.2.3 Identifying the Candidate Impact Set

- The dense reachability matrix of Table 6.2 has the risk of over-estimating the CIS.
- To minimize the occurrences of false positives, one might consider Distance based 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	

Table 6.2: Relationships represented by a reachability matrix [13] (©[2002] IEEE)

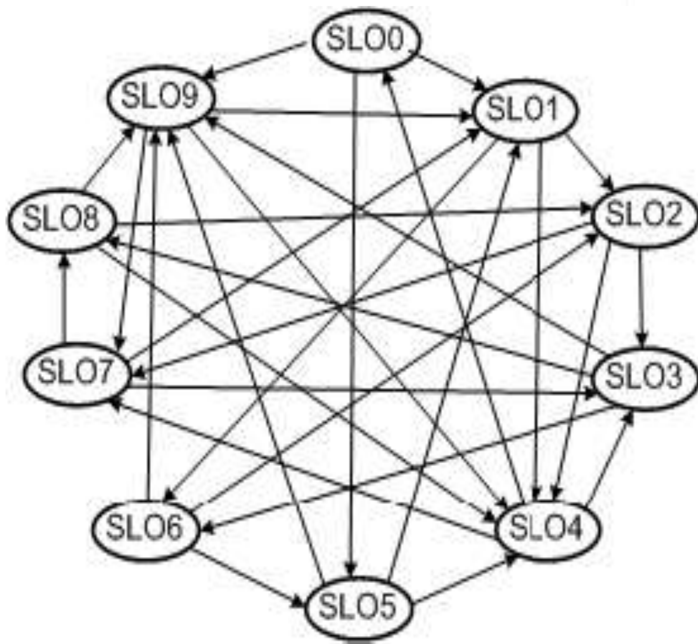
© Tripathy & Naik

Knowledge

6.2.3 Identifying the Candidate Impact Set

University of Waterloo

- **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.



	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	

Table 6.3: Relationship with distance indicators [13] (©[2002] IEEE).

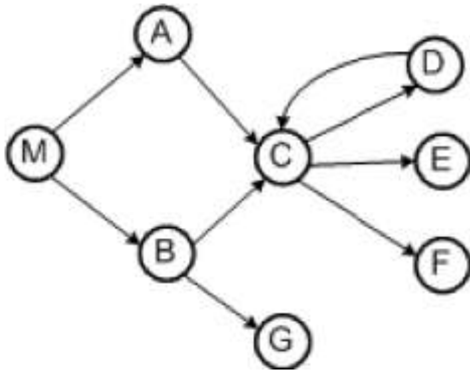
6.3 Dependency-based Impact Analysis

- 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:
  - The first technique is based on call graph.
  - the second one is based on dependency graph.

© Tripathy & Naik

6.3.1 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.
- An example call graph has been shown in Figure 6.7



- 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.

Figure 6.7 Example of a call graph ©IEEE, 2003

© Tripathy & Naik

6.3.1 Call Graph

- The call graph-based approach to impact analysis suffers from the following disadvantage:
  - impact propagations due to procedure returns are not captured in the call graph-based technique. Suppose that, in Figure, *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.

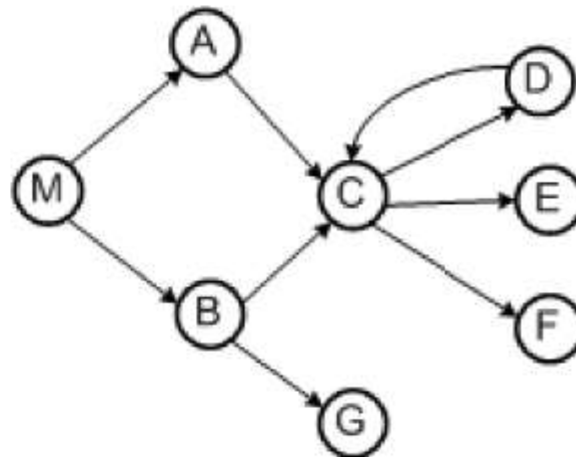


Figure 6.7 Impact analysis process ©IEEE, 2003

© Tripathy & Naik

Knowledge

6.3.1 Call Graph

University of Waterloo

- Let us consider an execution trace as shown in below.

  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.
- One can 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}.

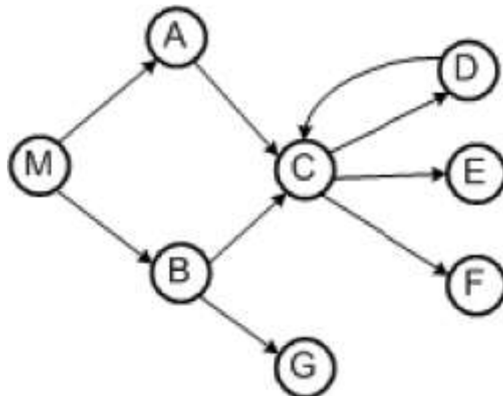


Figure 6.7 Impact analysis process ©IEEE, 2003

© Tripathy & Naik

6.3.2 Program Dependency Graph

- In the program dependency graph (PDG) of a program:
  - (i) each simple statement is represented by a node, also called a vertex;
  - (ii) 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 .

© Tripathy & Naik

6.3.2 Program Dependency Graph

- Figure 6.10 shows the PDG of the program given in Figure 6.9.
- Data dependencies are shown as solid edges, whereas control dependencies are shown as dashed edges.

Figure 6.9 Example program ©ACM, 1990

```
      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
```

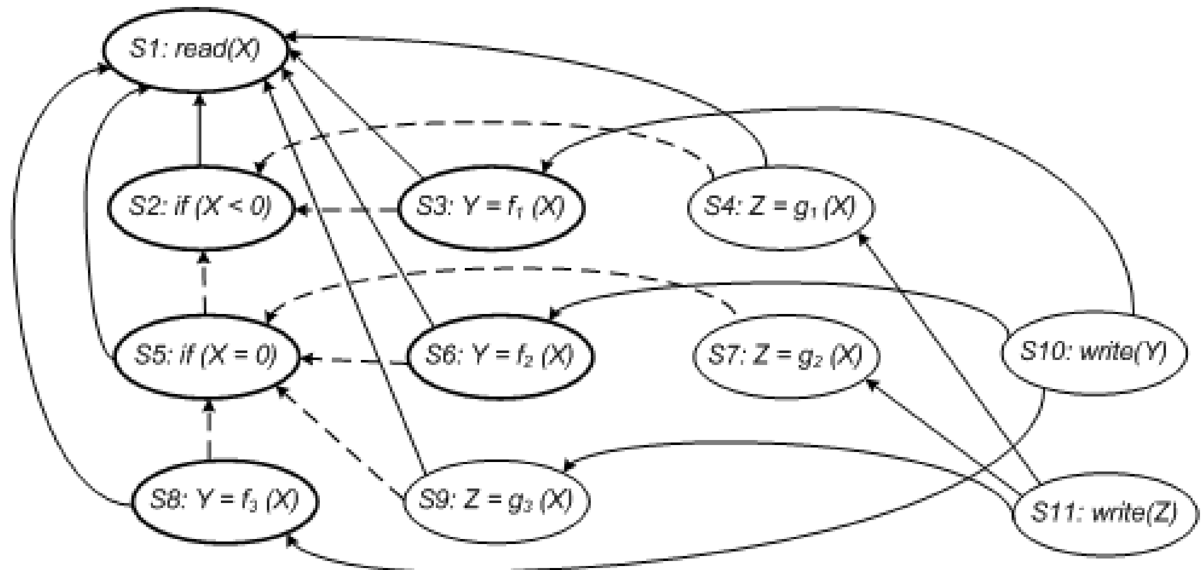


Figure 6.10 Program dependency graph of the program in Figure 6.9

© Tripathy & Naik

6.3.2 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 the program in the previous slide and variable Y at S10.
- First, find all the reaching definitions of Y at node S10 – and the answer is the set of nodes {S3, S6 and S8}.
- Next, 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}.
- In the program dependency graph the nodes belonging in the slice have been identified in bold.

© Tripathy & Naik

6.3.2 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 and S3}.

© Tripathy & Naik

Dynamic Slice

- 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.
- A simple approach to obtaining dynamic program slices is explained here.
- Given a test 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.

© Tripathy & Naik

Dynamic Slice

- Now, in our example discussed before, 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:

  (i) for the current test, mark the executed nodes in the PDG.

  (ii) traverse the marked nodes in the graph.

© Tripathy & Naik

6.3.2 Program Dependency Graph

Dynamic Slice

- Figure 6.11 illustrates how a dynamic slice is obtained from the program with respect to variable Y at the end of execution.
- For the case X = −1, the executed nodes are: < S1, S2, S3, S4, S10, S11 >.
- Initially, all nodes are drawn with dotted lines.
- If a statement is executed, the corresponding node is made solid.
- Next, beginning at node S3, the graph is traversed only for solid nodes.
- Node S3 is selected because the variable Y is defined at node S3.

- All nodes encountered while traversing the graph are represented in bold.
- The desired dynamic program slice is represented by the set of bold nodes {S1, S2, S3}.

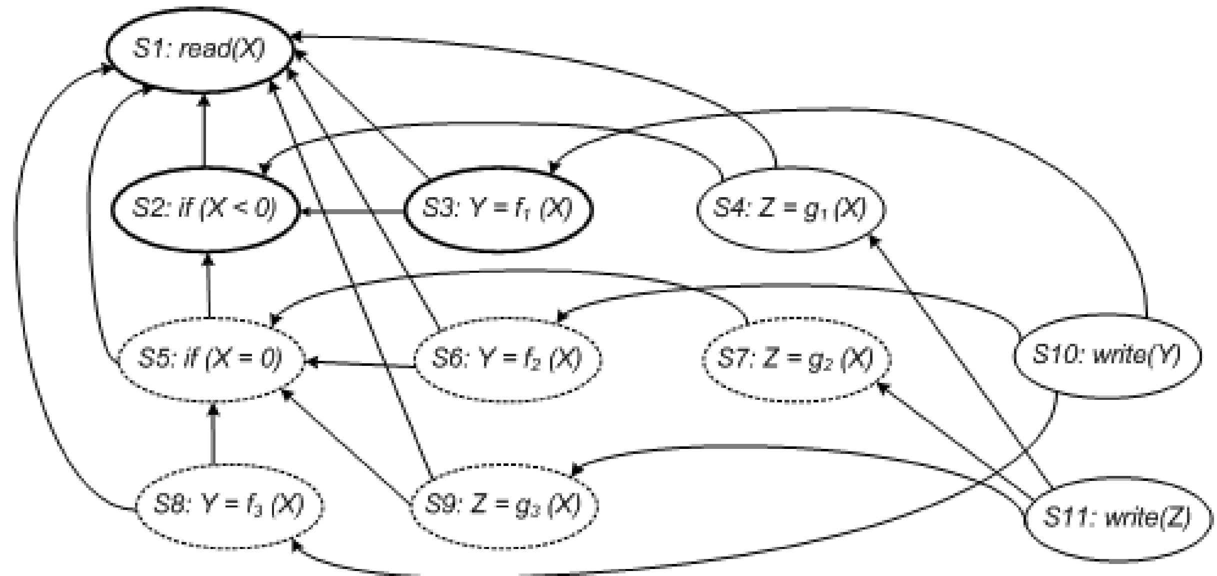


Figure 6.11 Dynamic program slice for the code in the figure 6.9, test case X= -1 with respect to a variable Y

© Tripathy & Naik