6

IMPACT ANALYSIS

An error does not become truth by reason of multiplied propagation, nor does truth become error because nobody sees it.

—Mohandas Karamchand Gandhi

6.1 GENERAL IDEA

A change request (CR) activates an organization's process to modify a software system to carry out maintenance. The maintenance process is started by performing impact analysis. Impact analysis basically means identifying the components that are impacted by the CR [1]. Impact analysis enables understanding and implementing changes in the system. Potential effects of the proposed changes are made visible by performing impact analysis. In addition, it is used in estimating cost and planning a schedule. Before executing a change request, impact of the changes are analyzed for the following reasons [2].

- To estimate the cost of executing the change request. It incurs some cost to execute a change request so estimate the cost before effecting the change. If a change request can potentially impact large disjoint portions of the system, then reexamine the request. A large change is likely to incur higher cost and it may make the system inconsistent. Therefore, if the size of the change is large, reject the request in favor of a safer change.

- To determine whether some critical portions of the system are going to be impacted due to the requested change. If so, more resources are to be allocated to execute the change request.
- To record the history of change-related information for future evaluation of changes. By taking feedback from end-users, the overall quality of changes can be evaluated at a later stage.
- To understand how items of change are related to the structure of the software. This will enable maintenance engineers to better understand the current change request.
- To determine the portions of the software that need to be subjected to regression testing after a change is effected.

There are several ways of looking at impact analysis as follows.

- Richard Turver and Malcolm Munro [3] define impact analysis as the assessment of the impact of a change to the source code of a module on the other modules of the system. It determines the scope of a change and provides a measure of its complexity.
- Queille et al. [4] define impact analysis as the task of assessing the effects of making a set of changes to a software system.
- Bohner and Arnold [5] define impact analysis as identifying potential consequences of a modification or discovering the entities to be modified to accomplish a change.

All the three definitions emphasize the estimation of the potential impacts which is crucial in maintenance tasks, because what is actually changed will not be fully known until after the software change is complete.

Implementation of change requests impact all kinds of artifacts, including the source code, requirements, design documentation, and test scenarios. Therefore, impact analysis traceability information can be used in performing impact analysis [6]. Gotel and Finkkelstein [7] define traceability as the ability to describe and follow the life of an artifact in both the forward and backward directions. Bohner and Arnold [5] define traceability as the ability to trace between software artifacts generated and modified during the software product life cycle. Thus, traceability helps software developers understand the relationships among all the software artifacts in a project. After identifying the high level documents about the feature to be modified, by using the concept of traceability, the maintainer locates the entities that need to be changed. Examples of such entities are design and source code.

There are two broad kinds of traceability: (i) horizontal (external) traceability; and (ii) vertical (internal) traceability. Traceability of artifacts between different models is known as external traceability, whereas internal traceability refers to tracing dependent artifacts within the same model. Internal traceability primarily focuses on source code artifacts. In this context, the three classical impact analysis techniques, based on program dependency, are: call-graph-based analysis, static program slicing, and dynamic program slicing. For example, inserting changes in the code and performing forward slicing from the changed points yield a set of statements that may be affected by the changes.

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. The concept of ripple effect has relevance in software evolution because it concerns changes and their effects [8]. Analysis of ripple effect reveals what and where changes are occurring. Measurement of ripple effects can provide the following information about an evolving software system: (i) between successive versions of the same system, measurement of ripple effect will tell us how the software's complexity has changed; and (ii) 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.

An implementation of a change request may consist of several steps; in each step, a specific module, identified during impact analysis, is visited and refactored. If a visited component is changed, it may no longer properly interact with other components. Therefore, additional changes are to be made in the neighboring components, which may trigger further changes throughout the software system. This process is called change propagation activity [9, 10]. The change propagation activity ensures that a change made in one component is propagated properly throughout the entire system. Maintenance engineers fail to correctly and completely propagate changes in a software system for a number of reasons: lack of experience, unexpected dependencies among code blocks, and misunderstanding of the design details. As a result, projects run the risk of generating new interface defects, if changes are incorrectly propagated [11].

6.2 IMPACT ANALYSIS PROCESS

Figure 6.1 depicts a process of impact analysis [5, 12]. The process begins by analyzing the CR, the source code, and the associated documentation to identify an initial set, called *starting impact set* (SIS), of software objects that are likely to be affected by the required change. To discover additional elements to be affected by the CR, the SIS is analyzed. The union of SIS and the new set generated by analyzing SIS is the *candidate impact set* (CIS) (a.k.a. estimated impacted set). An *actual impact set* (AIS) is obtained after the change is actually implemented. Given that one can implement a CR in many ways, the AIS set is not unique [13].

Impact analysis, which is an iterative process, has been illustrated in Figure 6.1. Newly impacted elements, not present in the CIS, may be identified while implementing a CR. A *discovered impact set* (DIS) represents the collection of all those newly discovered elements, and it indicates an underestimation of impacts of the change. Simultaneously, some members of CIS may not be actually impacted by the CR, and the group of those entities is known as *false positive impact set* (FPIS). FPIS indicates an overestimation of impacts. Ideally, AIS should be equal to $SIS \cup DIS \setminus FPIS$, where $\cup$ denotes set union and $\setminus$ denotes set difference. The error in impact

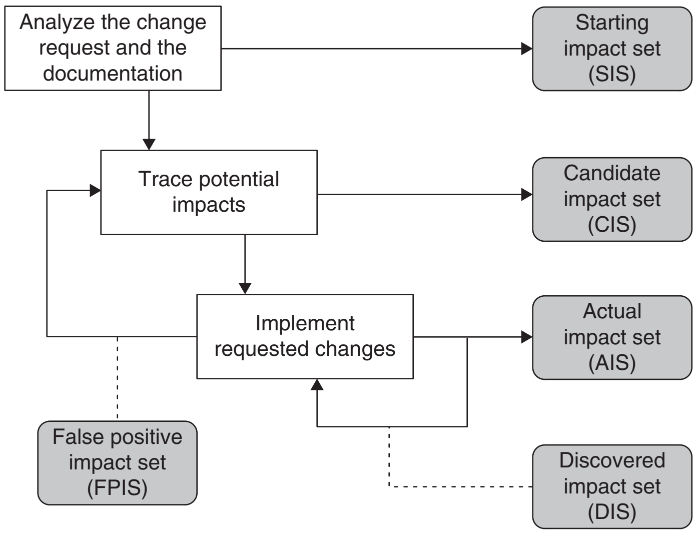
Analyze the change request and the documentation
Starting impact set (SIS)
Trace potential impacts
Candidate impact set (CIS)
Implement requested changes
Actual impact set (AIS)
False positive impact set (FPIS)
Discovered impact set (DIS)

FIGURE 6.1 Impact analysis process. From Reference 6. © 2008 IEEE

estimation can be computed as $(|DIS| + |FPIS|)/|CIS|$. The various sets of components are formally defined as follows.

- *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.
- *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, $FPIS = (CIS \cup DIS) \setminus AIS$.

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. Several metrics are defined in the literature to evaluate the impact analysis process [6, 13, 14]. Here, we discuss two traditional information retrieval metrics: *recall* and *precision*.

- *Recall:* It represents the fraction of actual impacts contained in CIS, and it is computed as the ratio of $|\text{CIS} \cap \text{AIS}|$ to $|\text{AIS}|$. 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. However, this does not happen too often. Therefore, recall might be traded off in favor of precision and vice versa. For a larger CIS, the probability of identifying all actual impacts is higher; the down side is that many false positives are encountered. *Adequacy* and *effectiveness* are two key aspects of any impact analysis approach [14].

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

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

  The concept of adequacy is essential to assessing the quality of an impact analysis approach. An inadequate approach is in fact useless, as it provides the maintenance engineer with incorrect information. For example, if *inconclusive* is 0, the approach cannot be used because it does not provide the maintenance engineer with all the components to be analyzed. In this case, the actual modification will be certainly affected by errors.
- *Effectiveness:* The ability of an impact analysis technique to generate results, that actually benefit the maintenance tasks, is known as its effectiveness. *Effectiveness* is expressed in terms of three fine-grained characteristics as follows.
  - *Ripple-sensitivity*
  - *Sharpness*
  - *Adherence*

  Ripple-sensitivity implies producing results that are influenced by *ripple effect*, which is discussed in Section 6.4. The set of objects that are directly affected by the change is denoted by DISO (directly impacted set of objects), and it is also known as primary impacted set. Similarly, the set of objects that are indirectly impacted by the change is denoted by IISO (indirectly impacted set of objects), and it is also known as the secondary impacted set. The cardinality of IISO is an indicator of ripple effect. The software maintenance personnel expect that the cardinality of IISO is not far from the cardinality of DISO. Therefore, the concept of *Amplification*, as defined below, is used as a measure of *Ripple-sensitivity*.

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

  where | . | denotes the cardinality operator.

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

$$ChangeRate = \frac{|\,CIS\,|}{|\,System\,|}.$$

It may be noted that CIS is included in "System", and *Change Rate* falls in the range from 0 to 1. For *Sharpness* to be high, we must have *Change Rate* $\ll 1$.

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:

$$S\text{-}Ratio = \frac{|\,AIS\,|}{|\,CIS\,|}.$$

If the impact analysis approach is adequate, AIS is included in CIS, and *S-Ratio* takes on values in the range from 0 to 1. Ideally, the *S-Ratio* is equal to 1.

6.2.1 Identifying the SIS

Impact analysis begins with identifying the SIS. The CR specification, documentation, and source code are analyzed to find the SIS. Larger software systems require more effort to identify the SIS. It takes more effort to map a new CR's "concepts" onto source code components (or objects) as discussed in Section 3.5. In the "concept assignment problem," one discovers human-oriented concepts and assigns them to their realization [15]. 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.

There are several methods to identify concepts, or features, in source code [16]. The "grep" pattern matching utility available on most Unix systems and similar search tools are commonly used by programmers [17]. One can search for variable names and comments in the code by using the "grep" tool. Each block of source code found in the search process needs to be studied to generate more search queries to find more variables, functions, and comments. However, the tool has some deficiencies: it is based on the correspondence between the name for the concept assigned by the programmer and an identifier in the code. The technique often fails when the concepts are hidden in the source code, or when the programmer fails to guess the program identifiers.

The software reconnaissance methodology proposed by Wilde and Scully [18] is based on the idea that some programming concepts are selectable, because their execution depends on a specific input sequence [19]. 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; and (iii) the marked code are likely to be in or close to the code implementing the feature.

Chen and Rajlich [20] 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. The C functions are successively explored to find and understand all the components related to the given feature.

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. Moreover, the complexity and size of the work products are expected to increase. Therefore, 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 [21].

By means of an example, we explain the traceability links and graphical relationships among related work products (see Figure 6.2). The graph is constructed by examining each requirement, and then linking the requirements to the design component that implements it. Next, design components are linked with the corresponding modules of source code. In the final step, source code modules are linked with sets of test cases. The graph that is so constructed reveals the relationships among work products. Specifically, the graph shows the *horizontal traceability* of the system. The

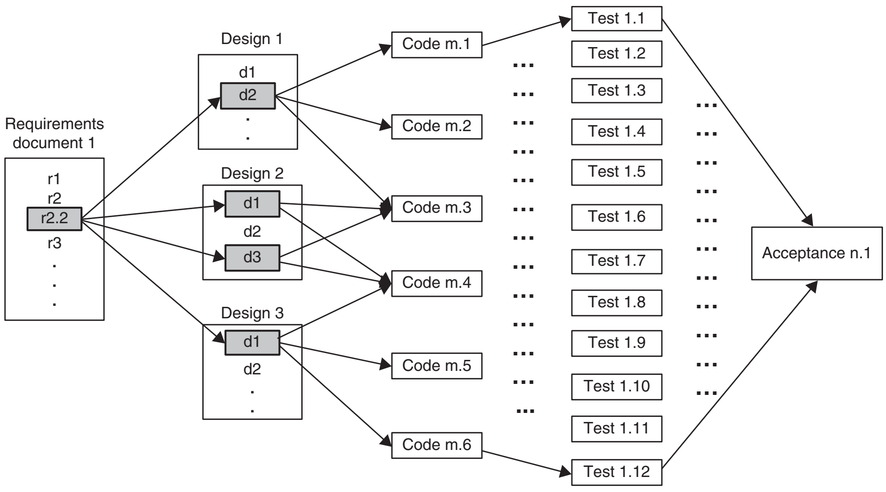

FIGURE 6.2 Traceability in software work products. From Reference 22. © 1991 IEEE

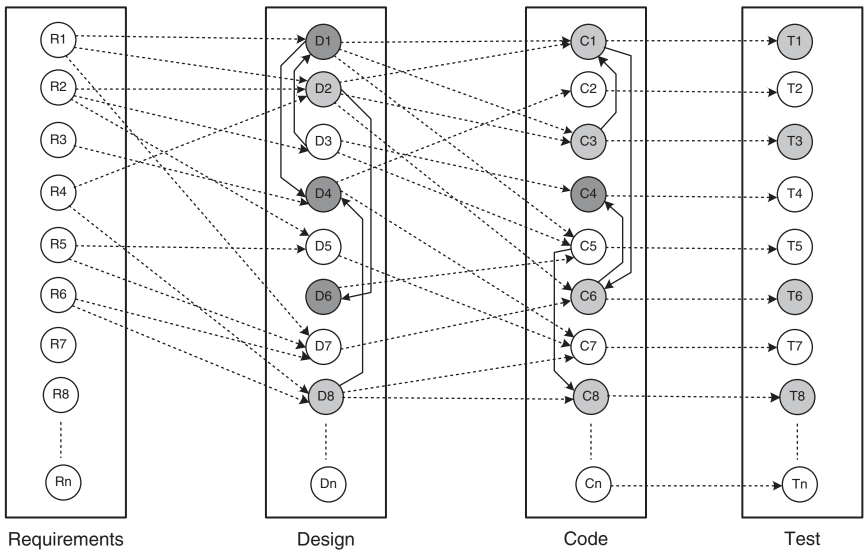

FIGURE 6.3 Underlying graph for maintenance. From Reference 22. © 1991 IEEE

graph has four categories of nodes: requirements, design, code, and test. Figure 6.3 is an example of the general appearance of such a graph. Each category of nodes is represented by a silo, and additional edges can be found within a silo. The edges within a silo represent *vertical traceability* for the kind of work product represented by the silo. Vertical traceability has been represented by solid lines, whereas horizontal traceability by dashed lines. 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.

If some changes are made to requirement object "R4," the results of horizontal traceability and vertical traceability from Figure 6.3 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.

As work products change, both the vertical traceability and horizontal traceability are likely to change. The change to vertical traceability is assessed by considering the complexity and size of the vertical traceability graph within each silo. A common measure of complexity of a graph is the well-known Cyclomatic complexity. Also, node count is a measure of size. It may be noted that vertical traceability metrics are *product metrics*—and those metrics reflect the effect of change on each product. 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.

On the other hand, *process metrics* are useful in examining horizontal traceability. To understand changes in horizontal traceability, it is necessary to understand:

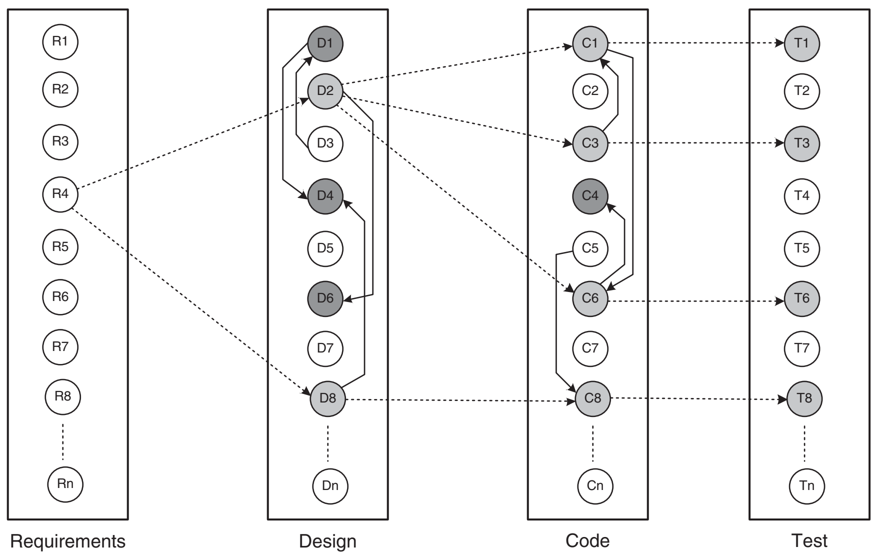

FIGURE 6.4 Determine work product impact. From Reference 22. © 1991 IEEE

(i) the relationships among the work products; and (ii) how work products relate to the process as a whole. In traceability graphs, the relationships among work products are denoted by dashed lines. For a pair of adjacent work products, one can examine the subgraph formed by the nodes of the work products and the dashed lines. Therefore, there exist three graphs: (i) the first one relating requirements to software design; (ii) the second one relating software design to source code; (iii) and the third one relating source code to tests. Next, size and complexity metrics are obtained for those three graphs to know about the work products and the effects of changes. One might conclude that 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.

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 [12]. 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*.

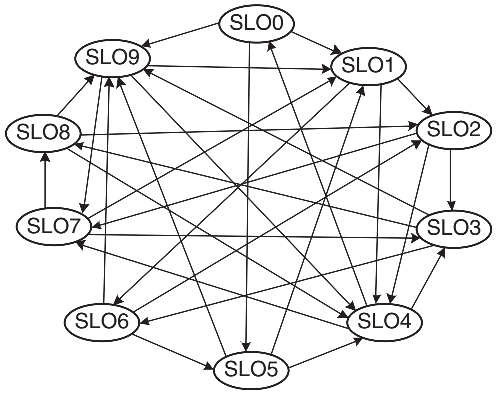

FIGURE 6.5 Simple directed graph of SLOs. From Reference 12. © 2002 IEEE

To understand those concepts, let us consider the directed graph in Figure 6.5 with ten SLOs, SLO0–SLO9. Each SLO represents a software artifact connected to other artifacts. The artifacts can be arbitrary entities, ranging from a requirement of the entire system to the definition of a variable. 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 6.6 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.

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

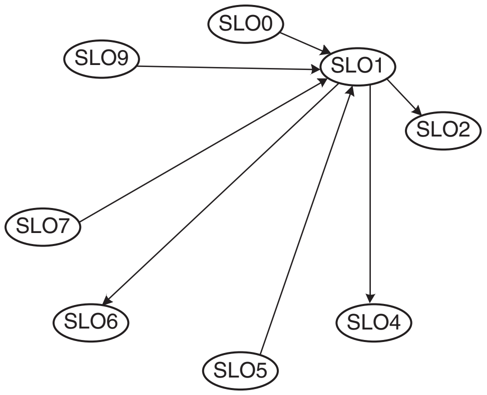

FIGURE 6.6 In-degree and out-degree of SLO1. From Reference 12. © 2002 IEEE

TABLE 6.1 Relationships Represented by a Connectivity Matrix

	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		

Source: From Reference 12. © 2002 IEEE.

impacted by a modification to an SLO, and there is a likelihood of overestimation. 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 the following two approaches.

- Distance-based approach: In this approach, SLOs which are farther than a threshold distance from SLO *i* are considered not to be impacted by changes in SLOW *i*. In Table 6.3, the concept of distance has been introduced in the analysis. One can estimate the scope of the ripple by augmenting Warshall's algorithm [23] with data about the nodes traversed so far.
- Incremental approach: In this approach, the CIS is incrementally constructed [24]. 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. The identification process is recursively executed until all the impacted SLOs are identified.

TABLE 6.2 Relationships Represented by a Reachability Matrix

	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	

Source: From Reference 12. © 2002 IEEE.

TABLE 6.3 Relationship with Distance Indicators

	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	

Source: From Reference 12. © 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 [25] are explained in this section. The first technique is based on call graph, whereas the second one is based on dependency graph [25].

6.3.1 Call Graph

A *call graph* is a directed graph in which a node represents a function, a component, or a method, and 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 [5]. 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*. Under this assumption, all the potential impact relationships in *P* can be calculated by applying the transitive closure relation. However, a procedure can have a variety of calling behavior, as follows: (i) a procedure *p*1 calls a second procedure *p*2, but *p*2 does not

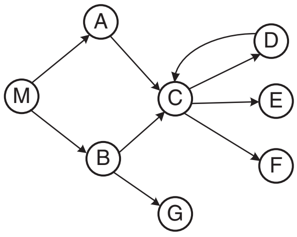

FIGURE 6.7 Example of a call graph. From Reference 26. © 2003 IEEE

M B r A C D r E r r r r x.

FIGURE 6.8 Execution trace

make any further calls; (ii) a procedure $p1$ calls a second procedure $p2$, and $p2$ calls a third procedure $p3$; or (iii) a procedure $p1$ calls a second procedure $p2$, and $p2$ makes calls to many other procedures.

Therefore, the call-graph-based approach to impact analysis suffers from many disadvantages as follows:

- A call graph represents the potential calls by a single procedure, while ignoring the dynamic aspects. Consequently, impact analysis based on call graphs can produce an imprecise impact set. For example, in Figure 6.7, one cannot determine the conditions that cause impacts of changes to propagate from M to other procedures.
- Generally, a call graph captures no information flowing via returns. Therefore, impact propagations due to procedure returns are not captured in the call-graph-based technique. Suppose that in Figure 6.7, 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.

To address the aforementioned issues, researchers have considered more precise ways to assess the impact of changes, using information collected during execution of calls. Law and Rothermel [26] defined a technique called *path-based dynamic impact analysis* that uses *whole path profiling* [27] to estimate the effects of changes. In this approach, if a procedure p is changed, then one considers the impact that is likely to propagate along those executable paths that are seen to be passing through p. As a result, any procedure, that is invoked after p but still appears on the call stack after p terminates, is assumed to be potentially impacted. For the approach to work, the software system needs to be instrumented to collect information. However, it is not necessary to have access to the source code, because program binaries can be instrumented.

Let us consider an execution trace as shown in Figure 6.8. The trace corresponds to a program whose call graph is shown in Figure 6.7. In the figure, r and x represent function returns and program exits, respectively. Let procedure E be modified. The impact of the modification with respect to the given trace is computed by *forward* searching in the trace to find: (i) procedures that are indirectly or directly invoked by E; and (ii) 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 $\{M, A, C, E\}$.

6.3.2 Program Dependency Graph

In the program dependency graph (PDG) of a program [28–30]: (i) each simple statement is represented by a node, also called a vertex; and (ii) each predicate expression is represented by a node. There are two types of edges in a PDG: data dependency edges

```
      begin
S1:        read(X)
S2:        if (X < 0)
           then
S3:            Y = f_1 (X);
S4:            Z = g_1 (X);
           else
S5:            if (X = 0)
               then
S6:                Y = f_2 (X);
S7:                Z = g_2 (X);
               else
S8:                Y = f_3 (X);
S9:                Z = g_3 (X);
               end_if;
           end_if;
S10:       write(Y);
S11:       write(Z);
      end.
```

FIGURE 6.9 Example program. From Reference 31. © 1990 ACM

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. Let us consider the program shown in Figure 6.9 [31]. Functions f_i and g_i, where $i = 1–3$, are assumed to have no side effects. Figure 6.10 shows the PDG of the program shown in Figure 6.9. Data dependencies are shown as solid edges, whereas control dependencies are shown as dashed edges.

A static program slice is identified from a PDG as follows: (i) for a variable *var* at node *n*, identify all reaching definitions of *var*; and (ii) find all nodes in the PDG

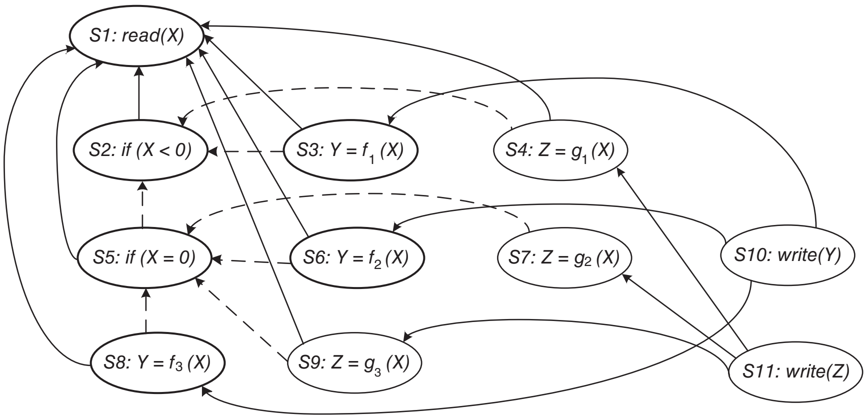

FIGURE 6.10 Program dependency graph of the program in Figure 6.9

which are reachable from those nodes. The visited nodes in the traversal process constitute the desired slice. Now we give an example of finding a static program slice. Consider the program in Figure 6.9 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 Figure 6.10, the nodes belonging in the slice have been identified in bold.

Referring to the static slice example discussed above, 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 $S3\}$. For -1 as the value 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. Thus, a dynamic slice is more useful in localizing the defect than the static slice.

Agrawal and Horgan [31] proposed some approaches to computing dynamic slices of programs. 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 $< v_1, v_2, \dots, v_n >$. 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. Note that unlike slicing where a slice is defined with respect to a given location in the program, dynamic slicing is defined with respect to the end of an execution history.

Now, in our example discussed before, the static program slice with respect to variable Y at $S10$ for the code shown in Figure 6.9 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; and (ii) traverse the marked nodes in the graph.

Figure 6.11 illustrates how a dynamic slice is obtained from the program in Figure 6.9 with respect to variable Y at the end of execution. For the case $X = -1$,

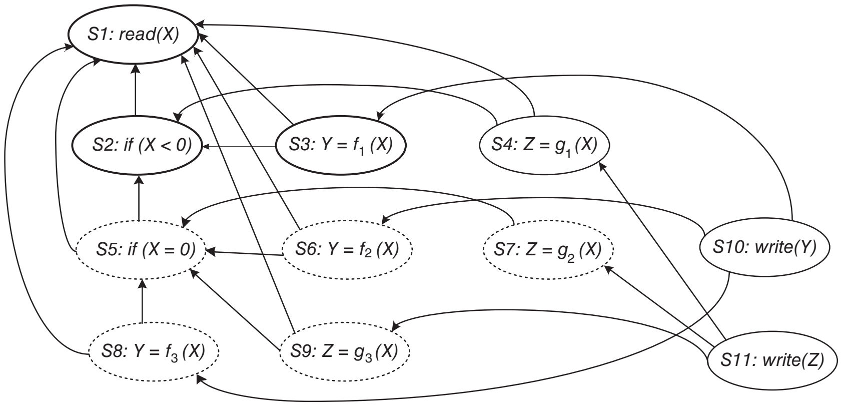

FIGURE 6.11 Dynamic program slice for the code in Figure 6.9, text case $X = -1$, with respect to variable Y

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 $\{S1, S2, S3\}$.

6.4 RIPPLE EFFECT

Haney [32] introduced the concept of ripple effect in the early 1970s, and the concept of *module connection analysis* was described by means of matrix algebra. Matrix algebra was applied in the estimation of the total number of modifications required to stabilize a changing system. The probability that a change in one module would require a change in any other modules is recorded in a matrix. From the functional and performance perspectives, Yau, Collofello, and McGregor [33] defined the concept of ripple effect in 1978. They viewed ripple effect as a complexity measure to compare changes to source code. In the scheme of Yau et al., 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, and 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 later extended to include stability measure. Stability reflects the resistance to the potential ripple effect which a program would have when it is changed [34]. 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 was studied by Yau and Collofello [35] by means of an algorithm, which computes stability based on design documentation. Specifically, one counts the number of assumptions made about shared global data structures and module interfaces. The key difference between design level stability and code level stability is as follows: design level stability does not consider change propagations within modules. In the next subsection we explain how to compute ripple effect, based on the works of Sue Black [36] and Yau et al. [33]. The basic idea is to identify the impact of a change to one variable on the program.

6.4.1 Computing Ripple Effect

Contained in module m_1 of Figure 6.12, consider the three lines of code referring to variables a, b, and d. A change in the value of b will impact a in line (1), and it will propagate to a in line (2). Variable a affects variable d in line (2) and this will propagate to variable d in line (3). Based on the above example, *intramodule change propagation* is defined as the propagation of changes from one source code line in a module to another source code line within the same module.

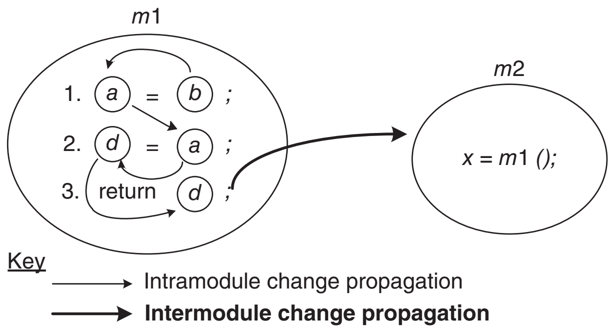
m1
1. a = b ;
2. d = a ;
3. return d ;
m2
x = m1 ();
Key
Intramodule change propagation
Intermodule change propagation

FIGURE 6.12 Intramodule and intermodule change propagation. From Reference 36. © 2001 John Wiley & Sons

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:

1. x is *defined* in an assignment statement;
2. x is *assigned* a value in a read input statement;
3. x is an *input* to an invoked module;
4. x is an *output* from an invoked module;
5. x is a *global* variable.

In V_m, variable definitions are uniquely identified. In case a variable is defined twice, then separate entries for each definition are included in V_m. Variable occurrences satisfying any of the five conditions—defined, assigned, input, output, and global— are denoted by "1"; otherwise, an occurrence is denoted by "0." In addition, the notation x_i^d means that the variable x has been defined at line (i). Similarly, the notation x_i^u means that the variable x has been used at line (i). In module m_1, variable a is global and it is considered to be defined. Matrix V_{m1} for the lines of code in m_1 is expressed as

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

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. It is to be noted that the value of a variable propagates from an occurrence in row i to an occurrence in column j. As an example, the propagation of the value of a occurring in line (2) to variable d occurring in the same line is recorded in the cell at row 4 and column 3—and not in the cell at row 3 column 4. The source code of module m_1 results in the following matrix:

$$Z_{m1} = \begin{array}{c} \\ a_1^d \\ d_1^u \\ d_2^d \\ a_2^u \\ d_3^u \end{array} \begin{array}{c} \begin{array}{ccccc} a_1^d & d_1^u & d_2^d & a_2^u & d_3^u \end{array} \\ \left(\begin{array}{ccccc} 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{array}\right) \end{array}.$$

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.

Propagation of values of variables in one module to variables in a different module is referred to as *intermodule change propagation*. Intermodule change propagation of values of a variable w occurs in the following ways:

1. If w is a global variable, then a change made to w by one module is seen by another module accessing w.
2. 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.
3. If w is an output parameter, then its value propagates from the module that makes an output to the module that accepts the output.

Now we examine the code segment in Figure 6.12. Variable d propagates to any module that calls m_1, 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. 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 m_1, propagation of their values to other modules is captured by an X matrix, denoted by X_{m1} as follows:

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

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 in propagation from b_1^u to m_3; similarly, the “0” in row 3, column 2 indicates no change in propagation from d_2^d to m_2.

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 the 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 [37] can be integrated with the ongoing computation of change propagation. A C matrix of dimension $1 \times n$ 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

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

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

TABLE 6.4 Laws of Software Evolution

Laws of Lehman	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
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 to 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

Source: Adapted from Reference 8. © 2006 John Wiley & Sons

the total number of variable definitions in m_1, represents the mean complexity-weighted propagation of variable definition 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 [36]:

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

where m = module and n = number of modules.

A tool called Ripple Effect and Stability Tool (REST) by Sue Black [38] automates the calculation of ripple effect by: (i) approximating intramodule change propagation; and (ii) excluding control flow within source code.

Sue Black examined some links between ripple effect measurement, as summarized in Table 6.4 [8], and Lehman's laws of software evolution, explained in Chapter 2.

6.5 CHANGE PROPAGATION MODEL

Change propagation means that if an entity, say, a function, is changed, then all related entities in the system are changed accordingly. Based on the work of Hassan

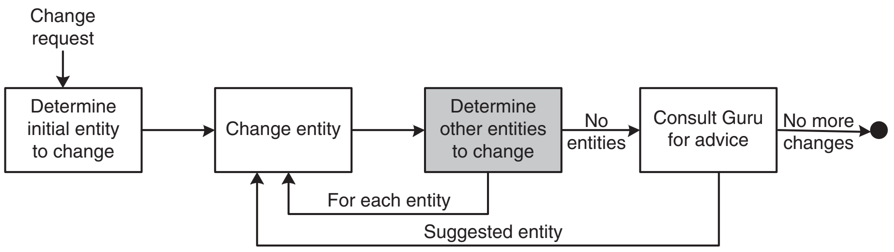

FIGURE 6.13 Change propagation model. From Reference 10. © 2004 IEEE

and Holt [10], a change propagation model has been illustrated in Figure 6.13. After receiving a change request, one identifies the initial entity in the system that needs to be changed. After changing the function, the maintainer must analyze the code to find out other, related entities to change. Then, those entities are actually modified to propagate the change. Similarly, the propagation process is repeated for each changed entity. A *Guru* is consulted when the maintenance engineer cannot identify more entities to modify. A *Guru* can be a senior developer or even a comprehensive test suite. If the *Guru* suggests that an entity be considered, then the entity is taken up for modification and the process for change propagation is applied to that entity. This iterative process is continued until all the desired entities have been changed. Eventually, the *change set* is determined for the change request and all entities in the *change set* are changed.

6.5.1 Recall and Precision of Change Propagation Heuristics

Gurus rarely exist and comprehensive test suites are generally incomplete in large maintenance projects. Therefore, software maintenance engineers need good change propagation heuristics, that is, good software tools that can guide them in identifying entities to propagate a change. The heuristic should possess a high precision attribute to be accurate and a high recall attribute to be complete. A heuristic with low precision will waste software maintainers' time, and maintainers will stop using them. On the other hand, if a heuristic has low recall, then software maintainers will miss to change some entities , thereby introducing defects.

We explained the concepts of *recall* and *precision* in Section 6.2. In this section, we explain the use of those two metrics to measure the change propagation heuristic by means of an example. Let us assume that Rohan wants to enhance an existing feature of a legacy information system. He first identifies that entity A needs to be changed. After changing A, a heuristic tool is queried for suggestions, and entities B and X are suggested by the tool. Next, B is changed and he determines that X should not be changed. Now the tool is given the information that B was changed, and the tool suggests that Y and W need to be changed. However, neither Y nor W need to be changed so no changes are performed on Y and W. After having used the tool, now Rohan consults a Guru, Krushna. Krushna indicates that C should be changed. Now, Rohan modifies C and queries the heuristic for additional entities to change.

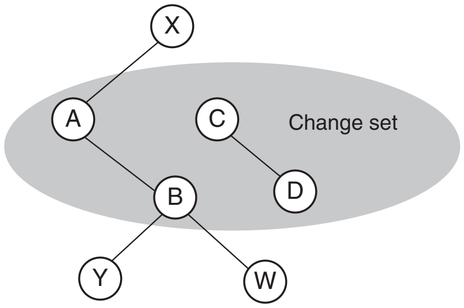

FIGURE 6.14 Change propagation flow for a simple example. From Reference 10. © 2004 IEEE

In response, D is suggested by the tool. Next, D is changed and Krushna is further queried. However, this time Krushna does not suggest any more entities for change. Now, Rohan stops changing the legacy system.

All the changed entities in the example represent the *change set* for this enhancement. The entities and their interrelationships have been shown in Figure 6.14. According to the heuristic, one edge is added from A to B and another edge from A to X. Since the change set contains A, the set contains both B and X. Similarly, edges are added from C to D, from B to Y, and from B to W. Now let us calculate the *recall* and *precision* for this example. The set of entities that are changed will be called *change set*; *change* = {A, B, C, D}. The set of entities suggested by the tool is called a *predicted* set. In the Rohan example, *predicted* = {B, X, Y, W, D}. The entities that were required to be predicted, but were found from Guru, are put in a set called the *occurred* set. In the Rohan example, *occurred* = {B, C, D}. The *occurred* set does not include A, which was initially selected by Rohan, because there is no need to predict it. That is, *occurred* = *change* − {*initial entity*}. Now, *recall* and *precision* for this example are computed as follows.

$$recall = \frac{|\,predicted \cap occurred\,|}{|\,occurred\,|} = \frac{2}{3} = 66\%$$

$$precision = \frac{|\,predicted \cap occurred\,|}{|\,predicted\,|} = \frac{2}{5} = 40\%$$

In the analysis of the above example to measure *recall* and *precision*, the authors, Hassan and Holt [10], made three assumptions.

- *Symmetric suggestions:* This assumption means that if the tool suggests entity F to be modified when it is told that entity E was changed, the tool will suggest entity E to be modified when it is told that entity F was changed. This assumption has been depicted in Figure 6.14 by means of undirected edges.

- *Single entity suggestions:* This assumption means that each prediction by a heuristic tool is performed by considering a *single entity* known to be in the *change set*, rather than multiple entities in the change set.
- *Query the tool first:* This assumption means that the maintainer (e.g., Rohan) will query the heuristic before doing so with the Guru (e.g., Krushna).

All the three assumptions together indicate that different orderings of selection and queries to a heuristic do not affect *precision* and *recall*.

6.5.2 Heuristics for Change Propagation

The “Determine Other Entities to Change” step in Figure 6.13 is executed by means of several heuristics. The set of entities that need to be changed as a result of a changed entity is computed in the aforementioned step. Modification records are central to the design of the heuristics. In general, source control repositories are used to keep track of all the changes made to files in the system. For each modification, a modification record stores the date and time of change, the name of the person making the change, the reason for the change, the code blocks that were changed, and names of the modified files. The changes can be recorded at the level of source code entities, namely, data type definitions, variables, and functions, to be able to track the following details.

- Modification, deletion, and addition of a source code entity.
- Alterations to dependencies between the changed entities and other entities in source code. For instance, it may be determined that a variable is no longer needed by a function.
- For each modification to the code, the corresponding modifications made to other files.

Each heuristic discussed in this section is characterized by: (i) data source; and (ii) pruning technique.

Heuristic Information Sources A heuristic can use one of many information sources to predict the entities that need to be modified. The objectives of the heuristics are to: (i) ensure that the entities that need to be modified are predicted; and (ii) minimize the number of predicted entities that are not going to be modified. Some potential information sources are as follows.

Entity information: In a heuristic based on entity information, a change propagates to other entities as follows.

- If two entities changed together, then the two are called a *historical co-change* (HIS).
- Static dependencies between two entities may occur via what is called CUD relations: *call*, *use* and *define*. A *call* relation means one function calls another

function; a *use* relation means a variable is used by a function; and a *define* relation means a variable is defined in a function or it appears as a parameter in the function.

- The locations of entities with respect to subsystems, files, and classes in the source code are represented by means of a *code layout* (FIL) relation. Subsystems, files, and classes indicate relations between entities— generally, related entities simultaneously.

Developer information (DEV): In a heuristic based on developer information, a change propagates to other entities changed by the same developer. In general, programmers develop skills in specific subject matters of the system and it is more likely that they modify entities within their field of expertise.

Process information: In a heuristic based on process information, change propagation depends on the development process followed. A modification to a specific entity generally causes modifications to other recently or frequently changed entities. For example, a recently changed entity may be the reason for some system-wide modifications.

Textual information: In a heuristic based on name similarity, changes are propagated to entities with similar names. Naming similarities indicate that there are similarities in the role of the entities [39].

Pruning Techniques A heuristic may suggest a large number of entities to be changed. Several techniques can be applied to reduce the size of the suggested set, and those are called pruning techniques, as explained in the following.

- *Frequency* techniques identify the frequently changing, related components. The number of entities returned by these techniques are constrained by a threshold. In a Zipf distribution, a small number of entities tend to change frequently and the remaining entities change infrequently.
- *Recency* techniques identify entities that were recently changed, thereby supporting the intuition that modifications generally focus on related code and functionality in a particular time frame.
- *Random* techniques randomly choose a set of entities, up to a threshold. In the absence of no frequency or recency data, one may use this technique.

6.5.3 Empirical Studies

Hassan and Holt [40] studied the performance of four heuristics—DEV, HIS, CUD, and FIL—using the development replay (DR) framework. To evaluate heuristics for change propagation, real change sets from the project's history are used in the DR framework. To evaluate a heuristic: (i) historical changes are played; and (ii) the *recall* and *precision* attributes are measured for every change set. They used five open software systems—NetBSD, FreeBSD, OpenBSD, Postgres and GCC—and the performance results are given in Table 6.5. The results show that historical co-change information is a practical indicator of change propagation. The authors

TABLE 6.5 Performance of Change Propagation Heuristics for the Five Software Systems

Application Name	Application Type	DEV		HIS		CUD		FIL	
		Recall	*Precision*	*Recall*	*Precision*	*Recall*	*Precision*	*Recall*	*Precision*
NetBSD	OS	0.74	0.01	0.87	0.06	0.37	0.02	0.79	0.16
FreeBSD	OS	0.68	0.02	0.87	0.06	0.40	0.02	0.82	0.11
OpenBSD	OS	0.71	0.02	0.82	0.08	0.38	0.01	0.80	0.14
Postgres	DBMS	0.78	0.01	0.86	0.05	0.47	0.02	0.77	0.12
GCC	C/C++ Compiler	0.79	0.01	0.94	0.03	0.46	0.02	0.96	0.06
	Average	0.74	0.01	0.87	0.06	0.42	0.02	0.83	0.12

Source: From Reference 10. © 2004 IEEE.

applied the pruning techniques discussed in Section 6.5.2 to improve *precision*, while maintaining high *recall*.

6.6 SUMMARY

We introduced three related concepts in this chapter: analysis of impacts of changes to the system, ripple effect, and change propagation. Impact analysis is defined as the task of discovering the effects of making a change to a software system. It reveals likely effects of the proposed change before the changes are actually implemented. The main motivation for performing impact analysis is to minimize unexpected side effects from an intended change to the system. On the other hand, ripple effect analysis emphasizes on the tracing of repercussions in source code when code is changed. The ripple effect measure can provide insights into the system through its evolution: (i) know the change in the system's complexity compared to the previous version; (ii) know the complexity of individual modules; (iii) know the change in complexity of a system after a new module is added. The change propagation activity ensures that a change made in one component is propagated properly throughout the entire system.

Next, we described the process of impact analysis process. In this framework, we explained four different types of software object (or, components) sets, namely, SIS, CIS, AIS, and FPIS. The AIS set is not unique because a modification can be effected in many ways.

In general, tools for impact analysis estimate SIS, CIS, and AIS, and we discussed guidelines to estimate those sets. In addition, we introduced several metrics to evaluate the impact analysis process. The two commonly used information retrieval metrics are recall and precision.

Precision is a measure of the performance of a retrieval technique to not return non-relevant items. The precision of a technique is equal to 1 if every entity returned by the technique is relevant to the query. However, a technique with precision equal to 1 runs the risk of missing relevant entities. Therefore, precision and recall should be considered.

The performance metric *recall* is the proportion of relevant entities, that is data or documents, retrieved by a technique from all available entities. The recall of a technique is equal to 1 if every relevant entity is retrieved. Theoretically, it is possible to achieve a recall value of 1 by returning every entity in the collection. In that sense, recall by itself is not a good measure. Therefore, recall needs to be considered with precision.

Next, we discussed traceability graph techniques to support impact analysis. Dependencies and relationships among software artifacts are understood by means of traceability links. After identifying the high level documents concerning the feature to be modified, traceability is used to locate the portions of software design, source code, and tests that are needed to be updated. Traceability is classified into two types: horizontal (external) and vertical (internal). In general, source code objects are used to provide vertical traceability, and dependency-based analysis techniques are used to identify dependencies among the source code objects. Dependency-based impact analysis techniques asses the effects of change by identifying syntactic dependencies that may indicate semantic dependencies, and we examined two such techniques which operate on call graphs and PDGs.

We provided a brief introduction to ripple effect and differentiated it from stability analysis. Then we described the ripple effect algorithm with a simple example. Next, we examined the link between Lehman's laws of software evolution and measurement of ripple effect.

Finally, we introduced the concept of change propagation and provided a model for it, and we discussed several heuristics. The heuristics were characterized by data sources and pruning techniques. Empirical results of these heuristics were presented.

LITERATURE REVIEW

The techniques for performing impact analysis are partitioned into two major categories: one group is based on dependency analysis and the other group is based on traceability analysis [5]. Dependency-based techniques attempt to assess the effects of a change on semantic dependencies between program entities. Semantic dependencies are identified by means of syntactic dependencies. The techniques used to identify syntactic dependencies include slicing techniques and transitive closure on call graphs. One can follow requirement traceability links between source code and documentation to locate features described in the requirements document. To identify an impact set for a change request, Turver and Munro [3] have proposed the idea of a ripple propagation graph for modeling documentation entities and their links to source code. Ibrahim et al. [41] developed a traceability tool, called Catia, to trace ripple effects. Bianchi et al. [42] experimented with many examples of traceability link using the ANALYST tool; their aim was to assess: (i) the effectiveness of these traceability links to support impact analysis in object-oriented environment; and (ii) the accuracy of the maintenance process.

Impact analysis based on transitive closure of call graphs has been discussed in an article by Law and Rothermel [26]. A change process was modeled as a sequence of snapshots by Rajlich [9]. One snapshot represents one moment in the change process. In this technique a program is represented by a directed graph, where nodes represent *entities* (e.g., classes). There are two kinds of directed edges in a graph: one kind of edges represents dependencies and the second kind of edges represents inconsistencies. For example, classes a and b are represented by two nodes with the same name. Inconsistency between nodes a and b is represented by a directed edge $I < a, b >$, where b is to be updated for any change to a. For each edge $I < a, b >$, either a depends upon b or b depends upon a.

Intuitively, a program graph is said to be consistent if there are no I-type of edges. Based on the above graph model, four kinds of change propagation strategies were applied: *top-down*, *random change-and-fix*, *strict change-and-fix*, and *bottom-up*. Luqi [43] uses graphs and sets to represent changes. Ajila [2] explicitly defines elements and relations between elements to be traced with intra-level and inter-level dependencies. Impact analysis based on transitive closure of call graphs is discussed by Law [26]. Similarly, Lindvall et al. [44] show tracing across phases again with intra-level and inter-level dependencies. The authors also discuss an impact analysis technique based on traceability data of an object-oriented system. Impact analysis for software architectures has been studied by Zhao et al. [45]. Based on an architectural slicing and chopping technique, they propose an approach to performing change impact analysis at the architectural level of software systems. Berg et al. [46] proposed a framework for the impact analysis of software artifacts across several phases of software development.

Computing ripple effects for object-oriented systems is a research topic of considerable interest. To compute ripple effect at design level for object-oriented systems, Elish and Rine have given an algorithm in Reference 47. To compute ripple effect at the code level in C++, Chaumun et al. [48] studied the impact of changes in a system. To perform regression testing of modules implemented as objected-oriented code, Li and Offut [49] performed impact analysis to identify those modules. By means of experiments on two software systems, Kabaili et al. [50] studied the relationship between coupling metrics and changeability. They concluded that coupling is a predictor of changeability.

REFERENCES

[1] L. J. Arthur. 1988. *Software Evolution: The Software Maintenance Challenge*. John Wiley & Sons, New York, NY.

[2] S. Ajila. 1995. Software maintenance: An approach to impact analysis of object change. *Software Practice and Experience*, 25(10), 1155–1181.

[3] R. J. Turver and M. Munro. 1994. An early impact analysis technique for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 6(1), 35–52.

[4] J. P. Queille, J. F. Voidrot, N. Wilde, and M. Munro. 1994. *The Impact Analysis Task in Software Maintenance: A Model and a Case of Study*. Proceedings of the International

Conference on Software Maintenance (ICSM), October, 1994, Phoenix, Arizona, IEEE Computer Society Press, Los Alamitos, CA, pp. 234–242.

[5] S. A. Bohner and R. S. Arnold. 1996. An introduction to software change impact analysis. In: *Software Change Impact Analysis* (Eds S. A. Bohner and R. S. Arnold). IEEE Computer Society Press, Los Alamitos, CA.

[6] A De Lucia, F. Fasano, and R. Oliveto. 2008. *Traceability Management for Impact Analysis*. Proceedings of the 2008 Frontiers of Software Maintenance (FoSM), October, 2008, Beijing, China, IEEE Computer Society Press, Los Alamitos, CA, pp. 21–30.

[7] O. C. Z. Gotel and A. C. W. Finkelstein. 1994. *An Analysis of the Requirements Traceability Problem*. Proceedings of First International Conference on Requirements Engineering, 1994, pp. 94–101.

[8] S. Black. 2006. The role of ripple effect in software evolution. In: *Software Evolution and Feedback: Theory and Practice* (Eds N. H. Madhavji and J. F. Ramil and D. E. Perry). John Wiley, West Sussex, England.

[9] V. Rajlich. 1997. *A Model for Change Propagation Based on Graph Rewriting*. Proceedings of the International Conference on Software Maintenance (ICSM), October, 1997, Bari, Italy, IEEE Computer Society Press, Los Alamitos, CA, pp. 84–91.

[10] A. E. Hassan and R. C. Holt. 2004. *Predicting Change Propagation in Software Systems*. Proceedings of the International Conference on Software Maintenance (ICSM), October, 2004, Chicago, USA, IEEE Computer Society Press, Los Alamitos, CA, pp. 284–293.

[11] D. E. Perry and W. M. Evangelist. 1987. An Empirical Study of Software Interface Faults – An Update. Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences, January, 1987, Volume II, pp. 113–126.

[12] S. A. Bohner. 2002. *Software Change Impacts: An Evolving Perspective*. Proceedings of the International Conference on Software Maintenance (ICSM), October, 2002, Phoenix, Arizona, IEEE Computer Society Press, Los Alamitos, CA, pp. 263–271.

[13] R. S. Arnold and S. A. Bohner. 1993. *Impact Analysis – Towards a Framework for Comparison*. Proceedings of the International Conference on Software Maintenance (ICSM), October, 1993, IEEE Computer Society Press, Los Alamitos, CA, pp. 292–301.

[14] A. R. Fasolino and G. Visaggio. 1999. *Improving Software Comprehension Through an Automated Dependency Tracer*. Proceedings of 7th International Workshop on Program Comprehension, 1999, IEEE Computer Society Press, Los Alamitos, CA, pp. 58–65.

[15] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster. 1994. Program understanding and the concept assignment problem. *Communications of the ACM*, May, 72–82.

[16] N. Wilde, M. Buckellew, H. Page, V. Rajlich, and L. Pounds. 2003. A comparison of methods for locating features in legacy software. *The Journal of Systems and Software*, 65, 105–114.

[17] S. E. Sim, C. L. A. Clarke, and R. C. Holt. 1998. *Archetypal Source Code Searches: A Survey of Software Developers and Maintainers*. Proceedings of International Workshop on Program Comprehension, 1998, IEEE Computer Society Press, Los Alamitos, CA, pp. 180–187.

[18] N. Wilde and M. Scully. 1995. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance and Evolution: Research and Practice*, 7, 49–62.

[19] V. T. Rajlich and N. Wilde. 2002. The role of concepts in program comprehension. IWPC, June, 2002, Paris, France, IEEE Computer Society Press, Piscataway, NJ, pp. 271–278.

[20] K. Chen and V. Rajlich. 2000. *Case Study of Feature Location Using Dependence Graph*. Proceedings of International Workshop on Program Comprehension, 2000, IEEE Computer Society Press, Los Alamitos, CA, pp. 241–249.

[21] S. L. Pfleeger and S. A. Bohner. 1990. *A Framework for Software Maintenance Metrics*. Proceedings of International Conference on Software Maintenance, 1990, IEEE Computer Society Press, Los Alamitos, CA, pp. 320–327.

[22] S. A. Bohner. 1991. *Software Change Impact Analysis for Design Evolution*. Proceedings of International Conference on Software Maintenance and Reengineering, 1991, IEEE Computer Society Press, Los Alamitos, CA, pp. 292–301.

[23] S. Warshall. 1962. A theorem on boolean matrices. *Journal of the ACM*, 9(1), 11–12.

[24] V. Rajlich and P. Gosavi. 2004. Incremental change in object-oriented programming. *IEEE Software*, 21(4), 62–69.

[25] M. J. Harrold and B. Malloy. 1993. A unified interprocedural program representation for maintenance environment. *IEEE Transactions of Software Engineering*, 19(6), 584–593.

[26] J. Law and G. Rothermel. 2003. *Whole Program Path-based Dynamic Impact Analysis*. Proceedings of the International Conference on Software Engineering (ICSE), May, 2003, IEEE Computer Society Press, Los Alamitos, CA, pp. 308–318.

[27] J. Larus. 1999. *Whole Program Paths*. Proceedings of the ACM Conference on Programming Language Design and Implementation, May, 1999, ACM Press, New York, pp. 1–11.

[28] J. Ferrante, K. J. Ottenstein, and J. D. Warren. 1987. The program dependence graph and its uses in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3), 319–349.

[29] K. J. Ottenstein and L. M. Ottenstein. 1984. *The Program Dependence Graph in a Software Development Environment*. Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments, May, 1984, Pittaburgh, Pennsylvania, April, 1984, SIGPLAN Notices, ACM Press, New York, Vol. 19, No. 5, pp. 177–184.

[30] S. Horwitz, T. Reps, and D. Binkley. 1990. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1), 26–60.

[31] H. Agrawal and J. R. Horgan. 1990. *Dynamic Program Slicing*. Proceedings of the ACM Conference on Programming Language Design and Implementation, June, 1990, SIGPLAN Notices, ACM Press, New York, Vol. 25, No. 6, pp. 246–256.

[32] F. M. Haney. 1972. *Module Connection Analysis – a Tool for Scheduling of Software Debugging Activities*. Proceedings of the AFIPS Fall Joint Computer Conference, 1972, AFIPS Press, Reston, VA, pp. 173–179.

[33] S. S. Yau, J. S. Collofello, and T. MacGregor. 1978. Ripple effect analysis of software maintenance. COMPSAC, November, 1978, Chicago, Illinois, IEEE Computer Society Press, Piscataway, NJ, pp. 60–65.

[34] S. S. Yau and J. S. Collofello. 1980. Some stability measures for software maintenance. *IEEE Transactions of Software Engineering*6(6), 545–552.

[35] S. S. Yau and J. S. Collofello. 1985. Design stability measures for software maintenance. *IEEE Transactions of Software Engineering*, 11(9), 849–856.

[36] S. Black. 2001. Computing ripple effect for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13, 263–279.

[37] T. J. McCabe. 1976. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4), pp. 308–320.

[38] S. Black. 2008. Deriving an approximation algorithm for automatic computation of ripple effect measures. *Information and Software Technology*, 50, 723–736.

[39] N. Anquetil and T. Lethbridge. 1998. *Extracting Concepts from File Names: A New File Clustering Criterion*. Proceedings of the International Conference on Software Engineering (ICSE), April, 1998, Kyoto, Japan, IEEE Computer Society Press, Los Alamitos, CA, pp. 84–93.

[40] A. E. Hassan and R. C. Holt. 2006. Replaying development history to assess the effectiveness of change propagation tools. *Empirical Software Engineering*, 11(3), 335–367.

[41] S. Ibrahim, N. Idris, M. Munro, and A. Deraman. 2005. Implementing a document-based requirement traceability: A case study. IASTED International Conference on Software Engineering (SE 2006), February, 2005, Innsbruck, Austria, ACTA Press, Calgary, Canada, pp. 124–131.

[42] A. Bianchi, A. Fasolino, and G. Visaggio. 2000. *An Exploratory Case Study of the Maintenance Effectiveness of Traceability Models*. Proceedings of International Workshop on Program Comprehension, 2000, IEEE Computer Society Press, Los Alamitos, CA, pp. 149–158.

[43] Luqi. 1990. A graph model for software evolution. *IEEE Transactions of Software Engineering*, 18(8), 917–927.

[44] M. Lindvall and K. Sandahl. 1998. Traceability aspects of impact analysis in object-oriented systems. *Journal of Software Maintenance and Evolution: Research and Practice*, 10, 37–57.

[45] J. Zhao, H. Yang, L. Xiang, and B. Xu. 2002. Change impact analysis to support architectural evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 14, 317–333.

[46] K. V. Berg, J. M. Conejero, and Juan Hernández. 2007. Analysis of crosscutting in early development phases base on traceability. In: *transactions on aspect-oriented software development* (Eds A. Rashid and M. Aksit), LNCS, Vol. 4620, November, pp. 73–104. Springer, Berlin.

[47] M. O. Elish and D. Rine. 2003. *Investigation of Metrics for Object-oriented Design Logical Stability*. Proceedings of Seventh European Conference on Software Maintenance and Reengineering (CSMR), 2003, IEEE Computer Society Press, Los Alamitos, CA, pp. 193–200.

[48] M. A. Chauman, H. Kabaili, R. K. Keller, and F. Lustman. 1999. *A Change Impact Model for Changeability Assessment in Object-oriented Software Systems*. Proceedings of Third European Conference on Software Maintenance and Reengineering (CSMR), 1999, IEEE Computer Society Press, Los Alamitos, CA, pp. 130–138.

[49] L. Li and A. J. Offut. 1996. *Algorithm Analysis of the Impacts of Changes to Object-oriented Software*. Proceedings of the International Conference on Software Maintenance (ICSM), Phoenix, Arizona, IEEE Computer Society Press, Los Alamitos, CA, pp. 171–184.

[50] H. Kabaili, R. K. Keller, and F. Lustman. 2005. Assessing object oriented software changeability with design metrics. IASTED International Conference on Software Engineering (SE 2006), February, 2005, Innsbruck, Austria, ACTA Press, Calgary, Canada, pp. 61–66.

EXERCISES

1. Which of the following relationships between SLOs indicate a potential dependency between the objects such that a change to the second may require a change to the first?
   - **(a)** <Test case T> is derived from <requirement R>.
   - **(b)** <Object code O> is compiled from <source code C>.
   - **(c)** <Requirement spec S> was produced according to <plan P>.
   - **(d)** <Module M1> was written by the same person as <module M2>.
   - **(e)** <Requirement R> is tested with <test case T>.
   - **(f)** <Program P1> is an earlier version of <program P2>.
2. Explain the adequacy and effectiveness aspects of impact analysis.
3. Determine the set of potentially impacted procedures due to the change in procedure **A** for the execution traces **M B G r G r r A C E r r r x** and **M A C E r D r r r x** using the call graph given in Figure 6.7.
4. Consider the execution trace **M B r A C D r E r r r r x M B G r r r x M B C F r r r r x** and the call graph given in Figure 6.7. For the above trace, determine the set of potentially impacted procedures due to changes in procedures: (a) **G**; and (b) **C**.
5. Discuss the differences between static and dynamic program slicing.
6. Consider the program given in Figure 6.15. Obtain the dynamic slice using the simple approach discussed in the book for the input value of 1 for N and variable Z at the end of execution. Explain your observation. Suggest a new approach that avoids the problem you observed.

```
    begin
S1:     read(N)
S2:     Z = 0;
S3:     Y = 0;
S4:     I = 1;
S5:     while(I <= N);
        do
S6:       Z = f_1(Z,Y);
S7:       Y = f_2(Y);
S8:       I = I + 1;
        end_white;
S9:     while(Z);
    end.
```

FIGURE 6.15 Program

7. In a data dependency graph, what can you tell from the number of edges that enter a node corresponding to a statement S?
   - **(a)** This is the number of variables appearing in the statement S.

(b) This is the number of statements which use the values of variables set by the statement S.

(c) This is the number of statements which assign values to variables, where those values may be used in the execution of statement S.

(d) This is the number of variables whose values are used in the execution of statement S.

8. What is the major difference between stability analysis and impact analysis?

9. Derive the average *recall* and average *precision* over time for *M* multiple change sets.

10. Given *predicted* = {B, X, Y, Z, W, D } and *occurred* = {B, C, D, E}, calculate the value of *recall* and *precision*.

11. Calculate the value of *recall* for the change set {{A,B}, {H, G, L}, {P, Q, S, T }}, where {A, B}, {H, G, L}, and {P, Q, S, T} are three connected components.

12. Which of the following characteristics of data-intensive systems make it difficult to use traditional impact analysis techniques?

(a) There are a large number of different users of the data.

(b) There is a large volume of data.

(c) The applications which access the data are mission critical.

(d) There is a lack of knowledge of what data is stored, and how and why it is used.

(e) The data is often of very poor quality.