# Lecture 4. Impact Analysis (Ch. 6)

Impact analysis, impact analysis process, dependency-based impact analysis, ripple effect, change propagation model [1], p. 223-247

Contents

[Lecture 4. Impact Analysis (Ch. 6) 1](#_Toc180948380)

[1. GENERAL IDEA 1](#_Toc180948381)

[2. IMPACT ANALYSIS PROCESS 1](#_Toc180948382)

[2.1. Identifying the SIS 4](#_Toc180948383)

[2.2. Analysis of Traceability Graph 4](#_Toc180948384)

[2.3. Identifying the Candidate Impact Set 6](#_Toc180948385)

[3. DEPENDENCY-BASED IMPACT ANALYSIS 8](#_Toc180948386)

[3.1. Call Graph 9](#_Toc180948387)

[3.2. Program Dependency Graph 10](#_Toc180948388)

[4. RIPPLE EFFECT 11](#_Toc180948389)

[5. CHANGE PROPAGATION MODEL 15](#_Toc180948390)

[5.1 Recall and Precision of Change Propagation Heuristics 15](#_Toc180948391)

[5.2 Heuristics for Change Propagation 16](#_Toc180948392)

#  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. 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.

# IMPACT ANALYSIS PROCESS

Figure 6.1 depicts a process of impact analysis. The process begins by analysing 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. 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* ∪ *DIS* ⧵ *FPIS*,

where ∪ denotes set union and ⧵ denotes set difference. The error in impact 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* ∪ *DIS*) ⧵ *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. 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 ∣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. 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.

* *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 expressed in terms of a performance metric called *inclusiveness*, as follows.

*Inclusiveness* =1 if AIS *⊆* CIS and 0 otherwise *.*

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* = ∣ *IISO* ∣ /∣ *DISO* ∣ ⟶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* = ∣ *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 ≪* 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-Ratio* = ∣ *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.

## Identifying the SIS

Impact analysis begins with identifying the SIS. The CR specification, documentation, and source code are analysed 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). In the “concept assignment problem,” one discovers human-oriented concepts and assigns them to their realization. It is difficult to fully automate the concept assignment problem because programs and concepts do not occur at identical levels of abstractions, thereby necessitating human interactions. 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. 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 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; and (iii) the marked code are likely to be in or close to the code implementing the feature. A dependency-graph-based feature location method for C programs starts search 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.

## 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. 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 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.



## 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*.

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



SLO0=>SLO1=>SLO4=>SLO0

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 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 SLO *i*. In Table 6.3, the concept of distance has been introduced in the analysis.



SLO8=>SLO4=>SLO0, hence, distance(SLO8,SLO0)=2

* Incremental approach: In this approach, the CIS is incrementally constructed. For every SLO in the SIS, one considers all the SLOs interacting with it, and only SLOs that are actually impacted by the change request are put in the CIS. The identification process is recursively executed until all the impacted SLOs are identified.

# 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 in this section. The first technique is based on call graph, whereas the second one is based on dependency graph.

## 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. 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 make any further calls; (ii) a procedure *p*1 calls a second procedure *p*2, and *p*2 calls a third procedure *p*3; or (iii) a procedure *p*1 calls a second procedure *p*2, and *p*2 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 impact 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. A technique called *path-based dynamic impact analysis* uses *whole path profiling*  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}*.

## 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; and (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 *vi* and *vj* be two nodes in a PDG. If there is a data dependency edge from node *vi* to node *vj*, then the computations performed at node *vi* are directly dependent upon the results of computations performed at node *vj*. A control dependency edge from node *vi* to node *vj* indicates that node *vi* may execute based on the result of evaluation of a condition at *vj*. Let us consider the program shown in Figure 6.9. Functions *fi* and *gi*, 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 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 *S*10. First, find all the reaching definitions of *Y* at node *S*10—and the answer is the set of nodes *{S*3, *S*6, and *S*8*}*. Next, find the set of all nodes which are reachable from *{S*3, *S*6, and *S*8*}*—and the answer is the set *{S*1, *S*2, *S*3, *S*5, *S*6, *S*8*}*. 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, *S*3, *S*6, or *S*8, 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 *S*3 is executed. Therefore, with respect to variable *Y* at *S*10, the dynamic slice will contain only *{S*1, *S*2, and *S*3*}*. For −1 as the value of *X*, if the value of *Y* is incorrect at *S*10, one can infer that either *fi* is erroneous at *S*3 or the “if” condition at *S*2 is incorrect. Thus, a dynamic slice is more useful in localizing the defect than the static slice.

# 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).



d, not b

m3, not m2

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. A matrix *Vm* 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 *Vm*, variable definitions are uniquely identified. In case a variable is defined twice, then separate entries for each definition are included in *Vm*. 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 *xdi* means that the variable *x* has been defined at line (*i*). Similarly, the notation *xui* 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 *Vm*1 for the lines of code in *m*1 is expressed as

*.*

A zero–one (0–1) matrix *Zm* indicates values of what variables propagate to other variables in the same module. Individual occurrences of variables are denoted by rows and columns of *Zm*. 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:



It is easy to observe that *Zm*1 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~~ m3, *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 *Xm*1 as follows:



The intermodule change propagation for variables occurring in *m*1 is obtained by means of the Boolean product of the two matrices *Zm*1 and *Xm*1, as follows:



In the Boolean product *Zm*1*Xm*1, the “1” in row 2, column 3 indicates change in propagation from d*u*1

to *m*3; similarly, the “0” in row 3, column 2 indicates no change in propagation from *dd*2 to *m*2. The Boolean product of *Vm*1 and *Zm*1*Xm*1 indicates the variable definitions that propagate from *m*1 to other modules:



1

1

1

1

Now, *Vm*1*Zm*1*Xm*1 indicates that there are no change propagations to *m*1, one change propagation to *m*2 (via a1d), and three change propagations to *m*3 (via a1d,d1d, and a2u). 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 can be integrated with the ongoing computation of change propagation. A *C* matrix of dimension 1 × *n* is chosen to represent McCabe’s cyclomatic complexity, where *n* is the number of modules:



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 *Vm*1*Zm*1*Xm*1 and *C* is



This number represents the complexity-weighted total variable definition propagation for module *m*1. Dividing by the number of variable definitions in module *m*1, |**V***m*1 | we get the mean complexity weighted variable definition propagation per variable definition in module *m*1. In our example |**V***m*1| =3, and the logical ripple effect for module *m*1 is defined to be 4/3= 1*.*33. The logical stability measure for module *m*1 is defined to be its reciprocal, i.e. 3/4 = 0*.*75. The formula for computing Logical Ripple Effect for a Program (*LREP*) is



where *m* = module number, and *n* = number of modules.

# CHANGE PROPAGATION MODEL



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.

## 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 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. 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 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:



## 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.

*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.

***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 <https://en.wikipedia.org/wiki/Zipf%27s_law> , 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.