Analyse de la propagation basée sur les graphes logiciels et les données synthétiques

Vincenzo Musco

Université de Lille

EDSPI Ecole Doctorale des Sciences pour l'Ingénieur - Lille

Ecole Doctorale des Sciences pour l'Ingénieur – Lille

THÈSE

pour obtenir le grade de

Docteur de l'Université Lille III

Spécialité "Informatique"

présentée et soutenue publiquement par

Vincenzo Musco

le 3 Novembre 2016

Propagation Analysis based on Software Graphs and Synthetic Data

Directeur de thèse : **Philippe Preux**
Co-encadrant de thèse : **Martin Monperrus**

Jury

Dr. Jean-Rémy Falleri,	Université de Bordeaux	Rapporteur
Pr. Pascal Poizat,	Université Paris Ouest Nanterre la Défense	Rapporteur
Pr. Pascale Kuntz-Cosperec,	Université de Nantes	Examinatrice
Pr. Philippe Preux,	Université de Lille	Examinateur
Dr. Martin Monperrus,	Université de Lille	Examinateur

Centre de Recherche en Informatique, Signal et Automatique de Lille (CRIStAL)
UMR 9189

CRIStAL Centre de Recherche en Informatique, Signal et Automatique de Lille

cnrs

Inria informatics mathematics

Abstract

Programs are everywhere in our daily life: computers and phones but also fridges, planes and so on. The main actor in the process of creating these programs is human beings. As thorough as they can be, humans are known to make involuntary errors without their awareness. Thus, once finished an already hard phase of writing a program, they have to face the maintenance phase on which they have to deal with errors they had previously made. All long their development task, developers have to continuously face their (or their colleagues) errors. This key observation arises the need of aiding developers in their development/maintenance tasks.

Thus, for assisting developers, a large number of tools exist, some still in development, others are integrated in the IDE (debugging, test suites, refactoring, etc.). Some of these tools are manual, while others propose automatic assistance. To be effective, automated tools should capture in the best way as possible how the program is structured and works. For this purpose, one particularly well-suited data structure is graphs, materializing how concepts relate to each other. Using such data structures for proposing assistance tools to developers is a quite promising way to proceed.

In this thesis, we concentrate on tools based on a graph representation of the program. Two big challenges on which we concentrate in this thesis are change impact analysis (CIA) and fault localization (FL). The former concentrates on the determination of the impacts of a potential change which may be issued by the developer while the latter identifies a fault based on what happened during program execution. Both concepts are complementary: the former concentrates on antemortem sight of the problem on which one wants to identify a fault before the failure occurs while the latter concentrates on postmortem sight on which real failures are analyzed to define a way to go back to their sources.

In this thesis, we face two main problems: (i) the lack of a systematic evaluation methodology or framework to assess the performance of change impact analysis techniques and (ii) most current fault localization techniques focus on a specific set of elements reported by their approach without thinking about how they depend on each other across the program as a whole.

In this thesis, we aim at finding solutions to these problems. We present four contributions to address the two presented problems. The two first contributions concentrate mainly on the change impact analysis side of this thesis while the third works on the fault localization side. The last contribution is a possible application for future works. In a few words, this thesis explores the causes and consequences of failures on computer programs by proposing tools based on graphs.

Contents

List of Figures

List of Tables

List of Algorithms

1

Introduction

"Anyone who has never made a mistake has never tried anything new."

— Albert Einstein

1.1 Context

Our today's life is surrounded by machines. A large range of our daily tasks are ensured by computer systems. In order to be able to accomplish anything, computer systems require that each task be clearly defined. This is achieved by the means of computer programs. Developers are the key people who are responsible for creating these programs by writing source code in a way the computer is able to understand.

Unfortunately, developers are humans, and humans are well-known to make mistakes. There is no exception for computer programs: as they are written by humans, they are error-prone. These errors, embedded in the software source code, are a part of the running logic and lead to various undesired run-time behaviors: unexpected program terminations, unresponsive UI components (such as buttons, menu item, etc.), or even system crashes.

Debugging tasks can quickly become a brain-teaser for the developer as source code can be made of thousands of lines of code, split in hundreds of files. As a consequence, the number of possible source code locations for a fault can be very high. Moreover, some errors may not be detected by the compiler and remain silent until the program is executed: an example of such an error is a null pointer exception which is caused by calling a method on an object variable which in fact contains a null value.

Furthermore, when a code change is introduced in a program, it can have side effects. Indeed, a developer can introduce a change C_1 in his source code which make another piece of source code somewhere else in the program defective. These side effects can appear in a totally different point than where the original changes have been made. At the same time, when a developer or a software tester identifies an undesired behavior in the program, the obvious task is to correct the source code in order to make it work correctly. However, the precondition is to locate the fault in the source code: this step may be harder to achieve than fixing the problem itself. For instance, a developer working on a calculator application may change arithmetic-related code, e.g., division function. Then, when running the program, the user interface buttons of the calculator may not respond anymore. At first sight, the developer may think the button source code is defective while the root cause may be the arithmetic-related change.

Towards the omnipresence of these faults and the difficulty to locate them, a large range of approaches have been proposed to assist in debugging and fixing tasks. It begins early in the development stage with the software documentation: the developer adds comments his source code. They are neither compiled nor executed lines of code which help the developer to remember what he has done. Moreover, using an efficient logging system can help the developer as well as anyone else working on the code to identify where he should search and fix the source code.

Software testing is a popular approach in which the developer writes simple use cases of his code which will be run again and again during the development phase to ensure that any fault has been introduced in previously developed code.

Bug report systems and *issue tracking systems* propose to other people to report and discuss failures they face with a program. Thanks to *Version Control Systems* (a.k.a. VCS), such as Git or Subversion, developers can easily collaborate on projects and benefit from useful features such as source code versioning, i.e., tracing the history of program changes back.

A large number of techniques have been proposed by the software engineering research community for assisting developers when facing faults. These techniques are intended to help developers in detecting, locating and even fixing faults. Some tools are not only related to faults, but are also intended for assisting the developers with numerous tasks which may be tedious to handle manually and which may even lead to new faults. As an example, refactoring proposes to automatically handle moving pieces of code, renaming methods and so on, while ensuring the program coherence.

These approaches generally use any software artifacts to achieve their purpose. These artifacts include the source code itself, but also everything previously cited: documentation, bug reports, software testing, version control systems information, etc. For instance, in this thesis, we use the program source code to produce graphs (presented in Section 2.1.3) which expose the links between different program parts, and the program tests as a way to observe the program failures. These pieces of information are used for change impact analysis (presented in Section 2.2) on which we want to estimate the impacted tests based on a specific change and fault localization (presented in Section 2.3) on which we determine a fault in the code based on a set of failing tests.

In conclusion, detecting, locating and fixing program faults is time-demanding, tedious and difficult. Any change can break other program parts. For this reason, many tools are proposed to assist developers in their daily tasks. As time goes by, systems are becoming more and more complex, this assistance will thus be more and more appreciated.

1.2 Problems

In this thesis, we concentrate on problems related to propagation analysis. Propagation analysis consists in analyzing how a change inserted somewhere in the source code spreads through the software method callers, dependent variables and fields, etc. in such a way that it will have influence in other parts of the program. An obvious example is a failure due to the propagation of a fault inserted somewhere in the program.

In this failure perspective, the propagation problem can be seen from two points of view: antemortem and postmortem propagation. Antemortem propagation consists in reporting possible impacts of a change (i.e., a fault) to the developer while he is typing it down on his source code editor, without requiring to run the code to notice the impacts (i.e., a failure). Analyzing the propagation in this way is known as *change impact analysis (CIA)*.

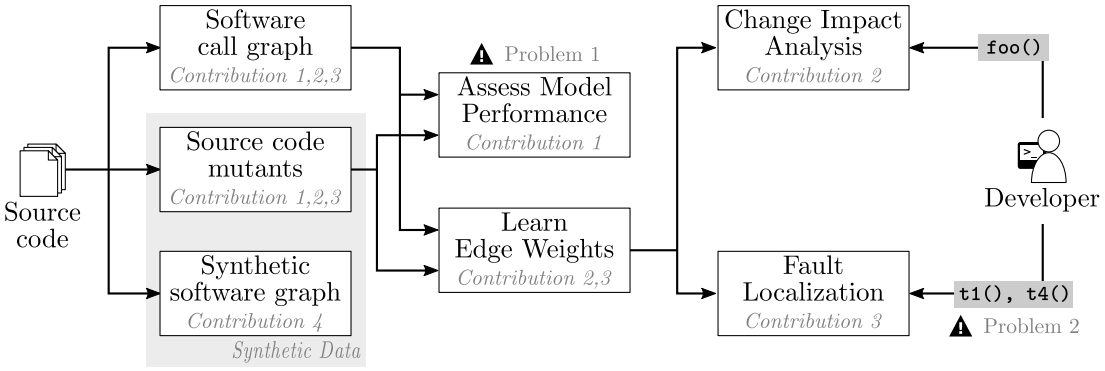


Figure 1.1: Problems and contributions of this thesis.

Problem 1

For existing change impact analysis techniques, there is no systematic evaluation methodology or framework to assess their performances. The performance is defined by the ability to locate elements really impacted by a change.

Postmortem propagation consists in reporting a set of elements (i.e., potential faults) which are responsible of a specific impact (i.e., program or system failure). In this approach, the fault detection is done once the failure occurs, that is, once the developer runs his code. Analyzing the propagation in this way is known as *fault localization*. In essence, the fault localization process tries to capture causality relationships between code elements.

Problem 2

Most current fault localization techniques do not consider the whole program, they focus on a specific set of elements reported by their approach without thinking to how they depend on each other across the program as a whole.

In this thesis, we tackle these problems based on two intuitions. The first intuition is that *software graphs are data structures materializing how code elements are interconnected.* They offer a global vision of the interactions between the different concepts they materialize. Thus, they are a good candidate for exploring propagation and its inherent causality. The second intuition is that *synthetic data are good candidates for simulating hard-to-obtain software information* such as atomic software changes. In change impact analysis and fault localization, software mutants can be used as synthetic faults.

1.3 Contributions

The contributions of this thesis are answers to problems presented in Section 1.2. Figure 1.1 is a simplified representation of how these problems and the contributions proposed in this thesis are articulated. As we can see, the source code can be used for three different purposes:

1. producing a *call graph*;

2. generating *software mutants*;

3. deriving some properties to feed a graph generator.

Based on these graphs and mutants, we propose four contributions that are presented in the remaining of this section. On the first three contributions, we extensively rely on program test suites: we consider them as an accurate way to observe impacts on a program. Thus, these contributions work at the method level.

Two contributions are related to change impact analysis, i.e., determinate potential impacts (failures) related to a change (faults).

Contribution 1

An evaluation framework for assessing the performance of a change impact analysis technique inspired from mutation testing.

The first contribution addresses the first problem. This framework is based on synthetic seeded faults obtained using mutation testing. The performances are assessed based on the program test execution result, i.e., the ability of the change impact analysis technique to report the tests which actually fail on run-time. Any change impact analysis technique would be assessable using this framework. We evaluate impact prediction of four types of call graphs. This evaluation enables us to study how the error propagates and is based on 16,922 mutants created from 10 open-source Java projects using 5 classic mutation operators.

Contribution 2

A novel change impact analysis technique based on information learned from past impacts and call graph.

The second contribution is a novel call graph-based change impact analysis approach. We use software mutants and their execution profile to learn causes of failures on the call graph resulting in a new type of graph, the causal graph. We evaluate our system using our evaluation framework (Contribution 1) and considering 9 open-source Java projects totaling 450,000+ lines of code. We simulate 16,682 changes and their actual impact through code mutations, as done in mutation testing.

Contribution 3

A new fault localization algorithm, built on an approximation of causality based on call graphs.

The third contribution makes use of graphs and mutants to evaluate their potential for fault localization, i.e., determinate causes (faults) of specific impacts (failures). We propose to use similar causal graphs as these used in Contribution 2. They are used to assist popular fault localization techniques in order to improve their performance. We evaluate our approach on the fault localization benchmark of Steimann et al. totaling 5,836 faults. This third contribution addresses the second problem.

Contribution 4

A generative model to create synthetic software graphs.

As we extensively work with synthetic faults, i.e., software mutants, we want to push further the experiments in generating software graphs for propagation analysis. Our last contribution is the first step toward this goal: we propose a new generative model of synthetic software dependency graphs. This model is used to generate synthetic graphs aiming at being similar to real ones. We extract the dependency graph of 50 open-source Java projects, totaling 23,178 nodes and 108,404 edges that we compare to generated graphs for assessing their fitness.

To sum up, we propose in this thesis four contributions for software engineering assistance. Two are used in the improvement of change impact analysis, one for fault localization and the last is a first step for future research.

1.4 Outline

The remaining of this thesis is structured as follows. In Chapter 2, we present the concepts as well as the works related to this thesis. In Chapter 3, we present our evaluation framework for change impact analysis and assess it with four flavors of call graphs (Contribution 1). In Chapter 4, we propose to use a learning technique to improve call graph-based change impact analysis (Contribution 2). In Chapter 5, we present a call graph-based fault localization technique used to improve the state-of-the-art ones (Contribution 3). In Chapter 6, we present our generative model for software dependency graphs (Contribution 4). In Chapter 7, we conclude this thesis and present perspectives.

1.5 Publications

In this section, we present the publications related to contributions presented in Section 1.3.

1.5.1 Published

[1] Vincenzo Musco, Antonin Carette, Martin Monperrus, and Philippe Preux. A Learning Algorithm for Change Impact Prediction. In *Proceedings of the 5th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering co-located with ICSE*, RAISE '16, pages 8–14, 2016.

[2] Vincenzo Musco, Martin Monperrus, and Philippe Preux. An Experimental Protocol for Analyzing the Accuracy of Software Error Impact Analysis. In *Proceedings of the 10th International Workshop on Automation of Software Test co-located with ICSE*, AST '15, pages 60–64, 2015.

[3] Vincenzo Musco, Martin Monperrus, and Philippe Preux. A Large-scale Study of Call Graph-based Impact Prediction using Mutation Testing. *Software Quality Journal*, 2016. To appear.

[4] Vincenzo Musco, Martin Monperrus, and Philippe Preux. Mutation-based graph inference for fault localization. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation*, October 2016.

Publications [2, 3] covers contribution 1, publication [1] covers contribution 2 and publication [4] covers contribution 3.

1.5.2 Under Submission

[1] Vincenzo Musco, Martin Monperrus, and Philippe Preux. A Generative Model of Software Dependency Graphs to Better Understand Software Evolution. *Journal of Software: Evolution and Process*, 2016. Minor Revision.

Publication [1] covers contribution 4.

1.5.3 To be Submitted

[1] Vincenzo Musco, Martin Monperrus, and Philippe Preux. Strogoff: A Recommendation System for Finding Sensitive Method Callers with Weighted Call Graphs.

Publication [1] covers contribution 2.

1.6 Reproducible Research

Many existing publications lack public implementations and, as a consequence, are hardly reproducible. As we opted for open-science, all our source code and dataset used to run our experiments are freely available online. For more information, please refer to Appendix A.

2

State of the Art

> "Smart people learn from their mistakes. But the real sharp ones learn from the mistakes of others"
>
> — Brandon Mull , *Fablehaven*

In this chapter, works related to this thesis are presented. This chapter is split in four parts. Section 2.1 presents essential definitions and concepts used in this thesis. Then, in the three following sections, we present related works. Section 2.2 covers change impact analysis related to contributions 1 and 2. Section 2.3 covers fault localization related to contributions 3 and Section 2.4 covers generative graph models for software engineering related to contributions 4.

2.1 Essential Definitions

In this section, we present fundamental yet essential concepts of software engineering used in this manuscript.

2.1.1 Errors

This manuscript deals with the detection of errors in programs. We present here a terminology used in software engineering based on the error stage.

Many terms are used to refer to software errors and even if they are frequently used interchangeably, they have not exactly the same meaning. To avoid confusion, in this manuscript, we use exclusively the terms *fault*, *error* and *failure* presented by Avizienis et al. [14] defined as:

Fault the physical presence in source code of elements which do not comply with the software specification and can lead to future program malfunctions;

Error the state of a program once a fault has been activated, i.e., the fault code has been run by the system and has started altering the correct state of the system. At this step, the system may continue to run without concrete observable malfunctions;

Failure a program which has deviated from the expected behavior. When a failure occurs, the expected behavior is not anymore respected, resulting in noticeable consequences such as unexpected program terminations due to a null pointer exception.

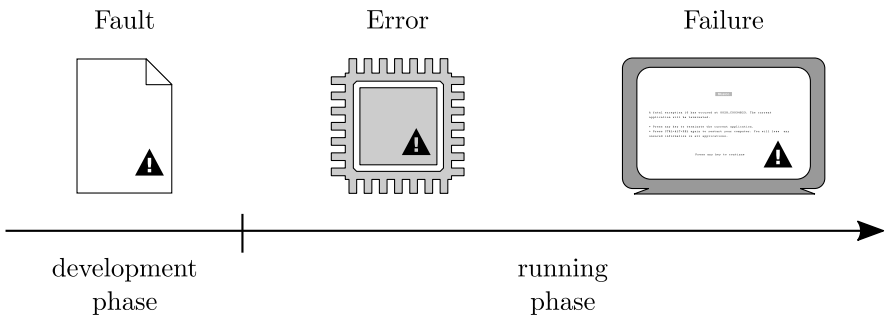


Figure 2.1: In the development phase, the fault occurs in the source code. In the running phase, the error is present in the system memory and the failure is the manifestation of a fault

> A failure can go beyond the scope of the program and reach the whole system, e.g., resulting in kernel crashes observable with the infamous BSOD[1] in Microsoft operating systems.

Figure 2.1 illustrates these concepts. As we can see, faults occur in the development phase, while errors and failures occur during the running phase.

2.1.2 Software Testing and Mutation Testing

Contribution 1, 2 and 3 presented in Section 1.3 mainly rely on software testing and mutation testing concepts. In this Section, we briefly present these concepts. Mutation testing is based on software testing as shown in Figure 2.2 which illustrates them.

2.1.2.1 Software Testing

When a developer decides to change some parts of his software code, he cannot directly estimate the impacts of his change. Generally, a developer has a rough estimation of these impacts, however, effects beyond his estimations may be possible.

The main goal of software testing [115] is to develop special content along with the source code which will be used to ensure the software execution does not deviate from its expected behavior.

Software testing is defined using *test suites*, which are classes grouping test cases related to similar software functionalities. *Test cases* are used to describe the expected behavior based on three components:

1. a set of *input data* used for testing. These can be of different nature: parameters, fields, global variables, etc.;
2. a *test scenario* describing the computation to perform on the input data and the values to return;
3. a *test oracle* determining if the returned values are acceptable or not. In other terms, it determines if the test should pass (i.e., succeed) or fail.

In this way, the developer can ensure his changes have not broken the initial logic by running the test suite of the program. If all tests pass, this indicates that the execution

[1] Blue screen of death

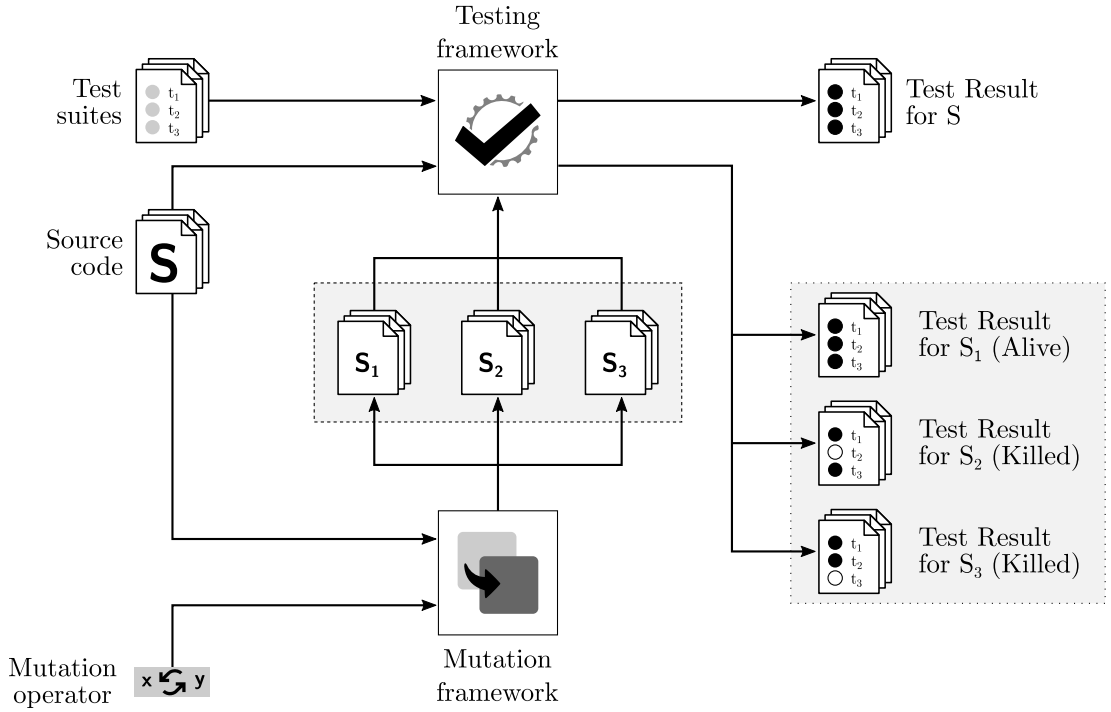


Figure 2.2: Illustration of software testing and mutation testing. Produced (three) mutants are in the gray dashed box. Test execution for mutants is in the gray dotted box. Passing tests are black, failing are white ones.

logic remains unchanged for other software parts. However, if at least one test fails, it means the change has an impact somewhere in the program. As a consequence, the developer will have to check his code to fix the problem.

In order to be useful, software testing requires that the test suite be well-designed. A good testing design requires that each piece of code have a test which is related to it in a way to ensure a good code coverage. The *code coverage* describes the proportion of statements which is covered (i.e., executed) by the tests. A high coverage means that tests run a large number of statements (e.g., each branch of a if-statement, each methods). A poor code coverage implies that test suite does not run a large number of code statements.

Figure 2.2 illustrates the testing process (as well as mutation process presented in the next section). As we can see, the testing framework takes as input test suites and the source code (S) and produces test results (for S, the first result has to be considered). In this example, we can see a test suite made of three test cases. All three test pass (black circle).

2.1.2.2 Mutation Testing

Mutation testing [32, 3, 33] is an approach intended to improve the code coverage of test suites. This approach is based on creating several copies of a program, called *mutants*, in which one (or several) small change(s) is (are) introduced.

By running test suites on each mutant, these are used to exhibit source code parts which require more testing. Indeed, each time we change a part of the code (i.e., we mutate the code), we introduce a little fault which should be detected by our test suites. If the change is not detected, it indicates that the change is not properly covered by test suites.

This is visible by observing the whole test suite results. If all test pass, the mutant is

said to be *alive* and implies the test coverage could be improved. In the opposite case, if at least one test fails, the mutant is said to be *killed* and implies that the test covers the part which has been mutated.

The type and the way elements are changed is defined by a mutation operator. Thus, a *mutation operator* defines:

1. the domain in which code elements should be included in order to be a candidate for mutation;

2. the logic used to concretely achieve the mutation.

As an example, let us consider an arithmetic mutation operator. Its domain includes binary operations which involve an arithmetic operator such as `+`, `-`, `*` and `/` between the left and right operand. The mutation logic is to replace the arithmetic operator by any other one. Thus, the `1 == 2` expression is not included in the domain and cannot be mutated using this mutation operator. On the other side, the expression `1 + 2` belongs to the domain, its mutation will result in expressions such as `1 - 2`, `1 * 2` and `1 / 2`.

Figure 2.2 illustrates the mutation testing process. As we can see, the mutation framework takes as input a mutation operator and a source code (S). It produces some mutants (in this example, three mutants are produced: S_1, S_2 and S_3 in the gray dashed box). Then, we feed these mutants to the testing framework (one at a time) as well as the test suites to obtain test results. In this example, results related to our mutants are ones in the gray dotted box. We can see that, for S_1, all test pass (black circles), which means that S_1 is an alive mutant. For S_2 and S_3, we see that there is one failing test in each (white circle) which means that these mutants are killed.

More information about research in mutation testing is presented in the survey proposed by Jia and Harman [72].

2.1.3 Graphs for Software Engineering

As presented in Section 1, our first intuition is that graphs are good candidates to explore propagation. In this section, we present graphs and all concepts related to graphs which are of interest in the scope of this manuscript. Curious readers interested in graph theory can read good references in the domain among [57, 28, 27, 157, 53, 120].

2.1.3.1 Graph Definition

A *graph*, also known as *network*, is a mathematical tool used to model a large number of problems which involve concepts and connections between these. A graph G is made of two sets: a set of *nodes* N (also called *vertices*) defining concepts and a set of *edges* E describing the connections between concepts. An example of such a graph is the connection of peripherals in a network where nodes are network devices (e.g., computer, router) and edges are added between two peripherals every time they are physically or logically connected together. A formal definition of a graph is given by Equation (2.1).

$$G = \{N, E\} \tag{2.1}$$

Where the edges E can be defined using a pair given in Equation (2.2).

$$E = (n_1, n_2) \tag{2.2}$$

with $n_1, n_2 \in N$. If $(n_1, n_2) \in E$, then n_1 and n_2 are said to be *adjacent* or *neighbors*.

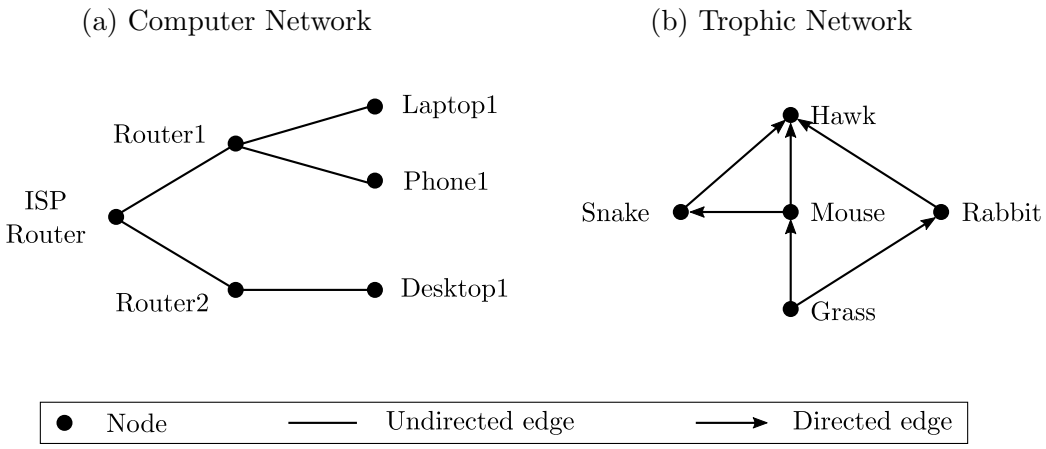


Figure 2.3: (a) illustrates a computer network graph made of 6 nodes and 5 edges. As the communication is bidirectional they can be undirected. (b) shows a trophic network made of 5 nodes and 6 edges. As the concept of "who eats who" is important, they must be directed.

A graph can be either *undirected* or *directed*: in the former, edges direction is not taken into consideration while it is in the latter. Directed graphs are also named *digraphs*. The usage of direction is defined by the modeled concept.

Figure 2.3 gives an example of each type of graph. Figure 2.3a illustrates a network example: an undirected graph is used as peripherals communicate in both directions (i.e., if two peripherals are physically or logically connected together, they can both communicate with each other). Figure 2.3b shows an example of directed graph: the predator-prey graph models how an organism eats another. Indeed, there is a relation between the hawk and the rabbit as the former eats the latter. But this observation is valid in one direction only: the rabbit does not eat the hawk.

Equation (2.2) is used to express both directed and undirected edge. In the former, this pair is ordered as the order expresses the direction of the relationship. Thus, Equation (2.2) expresses a directed edge going from n_1 to n_2 such as $n_1 \longrightarrow n_2$. In the latter, the pair is not ordered as the direction is not important.

A large amount of information can be embedded in nodes and edges. This can be *labels* (e.g., the IP address of a peripheral, an organism class), *weights* or any other type of data.

A *path* is an ordered list of nodes for which each pair of nodes in the list is connected by an edge in the graph. The *path length* is the number of pairs in this list. and the *shortest path* between two pairs of node is the path with the smallest length among all possible paths between these two nodes.

In this thesis, we distinguish two types of graphs: these resulting from an analysis of software systems and these created by a generative model. The former are qualified as "empirical" or "true", the latter being qualified as "synthetic" or "artificial".

2.1.3.2 Graph Degrees

The *in-degree* and the *out-degree* of a node n_1 are respectively the number of edges going to n_1 (i.e., the number of edges $(\cdot, n_1)$) and the number of edges leaving n_1 (i.e., the number of edges $(n_1, \cdot)$). We use the term *degree* to refer to both of these concepts and terms *in-degree* and *out-degree* when the distinction is necessary. In- and out-degree concepts

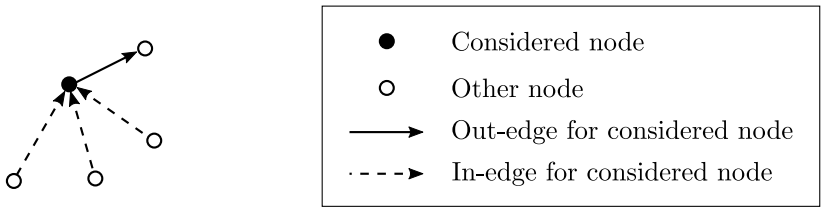


Figure 2.4: In-edges and out-edges for a simple digraph example.

do not exist for undirected graphs. Figure 2.4 illustrates a graph made of 4 nodes. Let us consider the black node, it contains three entering edges (dashed ones) and one leaving edge (plain one). Thus, as a consequence, the in-degree of this node is 3, its out-degree is 1 and its general degree is $3 + 1 = 4$.

The *degree distribution* of a graph is the proportion of each degree in this graph. The total of these proportions sums to 1. In this thesis, we consider *cumulative distribution* functions (CDF) of degrees which express the proportion of nodes whose degree is smaller or equal to a given value. The main reason is that noncumulative distributions are to be avoided as they are sources of mistakes [91]. Cumulative distributions are more appropriate to analyze noisy and right-skewed distributions [119]. The *inverse cumulative distribution* functions of degrees (ICDF) is the inverse of the cumulative distribution, i.e., the proportion of nodes whose degree is greater or equal to a given value k as expressed by Equation (2.3) where n is the number of nodes in the graph.

$$ICDF(k) = n - CDF(k-1) \tag{2.3}$$

Figure 2.5 illustrates these concepts for a simple digraph made of 14 nodes and 16 edges. Upper histograms illustrate the out-degree functions and the lower ones illustrate the in-degree functions. Leftmost histograms illustrate degree function (DF), central ones illustrate the cumulative degree function (CDF) and the rightmost illustrates the inverse cumulative degree function (ICDF). The small gray number of nodes on top of each bar reports the number for considered degree. Thus, considering in-edges, we can observe that there are 2 nodes with no in-edge and 9 nodes with in-degree 1. Now, if we consider the CDF for in-edges, we observe that there are 11 nodes which have an in-degree lower or equal to 1. For the ICDF, there are 12 nodes which have an in-degree higher or equal to 1.

2.1.3.3 Graph Metrics

There are a host of properties that describe various aspects of a graph, the relation between these properties being more or less understood. Let us mention the properties we consider in this manuscript:

Density the graph density expresses the proportion of pairs of nodes being connected in the graph. The graph density is computed using formula (2.4);

$$\delta = \frac{|E|}{|N| \times (|N| - 1)} \tag{2.4}$$

Diameter the length of the longest shortest path between any pair of nodes;

Average shortest path length the average length of the shortest path between any pair of nodes;

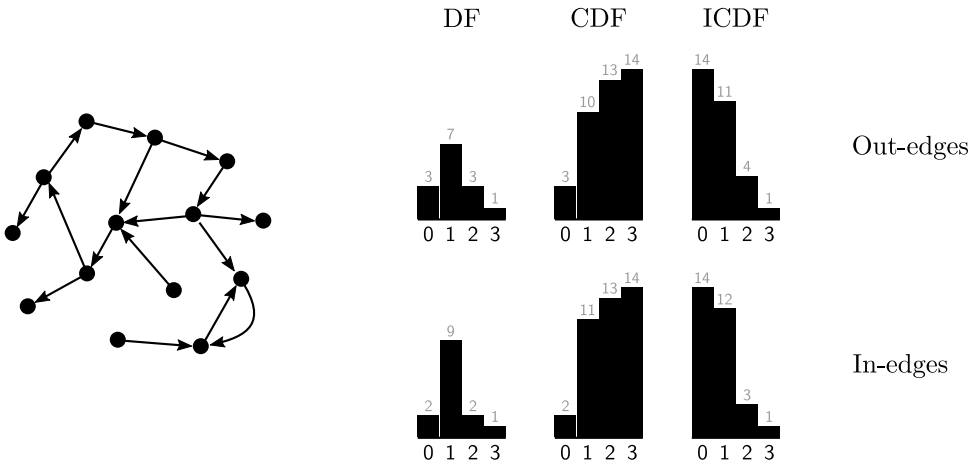


Figure 2.5: An example of directed graph with its corresponding in- and out-degree function (DF), cumulative degree function (CDF) and inverse cumulative degree function (ICDF).

Transitivity (a.k.a. clustering coefficient) indicates how elements are connected to each other. It is defined by the number of triangles divided by the number of paths of length 2 found in the graph. We define a path of length two as two edges between three nodes n_1, n_2 and n_3 such as n_1 is connected to n_2 and n_2 is connected to n_3. A triangle is a closed path of length 2 such as on the previous example, there is a third edge connecting n_3 to n_1. Figure 2.6 illustrates these concepts, Equation (2.5) shows a formal definition of the transitivity C;

$$C = \frac{number\ of\ triangles}{number\ of\ paths\ of\ length\ 2} \tag{2.5}$$

Modularity is a metric used to define how nodes groups are split in *modules*. A module is defined as a subset of nodes which have a dense connection between them (i.e., contain more edges between them) than with the other nodes of the graph.

The transitivity and the modularity are two different measures of whether there exist some subsets of nodes which are more connected to each other than the average. Though the exact relation between these two metrics is not yet clear, there are some similarities [123].

2.1.3.4 Software Graph Granularities

Granularity is defined by the Oxford dictionary as "the scale or level of detail present in a set of data or other phenomenon". When working with software data, this granularity concept is central and should be taken into consideration depending on what phenomenon we want to study.

At coarser granularities, we consider more global concepts while finer ones allow to deal with more detailed concepts. As an example, in Java programming language, *packages* are more global than *classes*. Classes are grouped in packages, thus working at the package granularity would imply that many classes are "hidden" behind a specific package.

Coarser granularities result in a smaller amount of data, offering the advantage of being easy to manipulate, but the drawback is offering a poor precision because finer concepts are hidden behind coarser ones. At the opposite, a finer granularity proposes to

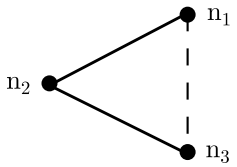


Figure 2.6: Example of a path of lenth 2 (solid lines) and a triangle (solid and dashed lines together).

obtain a larger amount of data which offers a better precision, but requires more resources and time to process as the amount of data can be very important.

Considering the graph perspective, as graphs can illustrate the interconnection of software concepts, they potentially can materialize concepts of any granularities (some authors also mix concepts from several granularities [49, 128]).

Figure 2.7 illustrates some granularities existing in software engineering and more specifically in Java object-oriented programming language. We observe the following granularities (from coarser to finer):

1. system: interactions between machines;
2. software: interaction of programs and libraries in the system itself;
3. package: group of classes generally intended to achieve same type functionalities;
4. class: concepts containing methods and fields which are intended to be initialized as an object by the mean of a constructor;
5. feature: one service proposed by a class and its internal state, i.e., methods and fields;
6. (basic) block: a set of consecutive program instructions which contains one entry point;
7. instruction: one command in a source code;
8. token: one item which assembled with other tokens forms instructions.

These are *vertical granularities* as they can be seen as a Russian nesting doll where each upper level is a generalization of the lower one. Note that as presented previously, coarser granularities indicate a poor precision than finer ones, but in some cases, a coarser granularity is required to analyze some concepts. As an example, in order to study interconnections of systems in a network, considering finer granularities is meaningless.

Horizontal granularity defines for each vertical granularity, how many different types of information can be embedded. Examples of horizontal granularities are:

- at the system vertical granularity: standalone applications, system libraries, user libraries, etc.;
- at the feature vertical granularity: methods, fields, inheritance information, etc.

When working at the software vertical granularity, data can be split in two types: *application* and *library* data. *Application data* belongs to core software itself. *Library data* belongs to an external library or programs called by the core program.

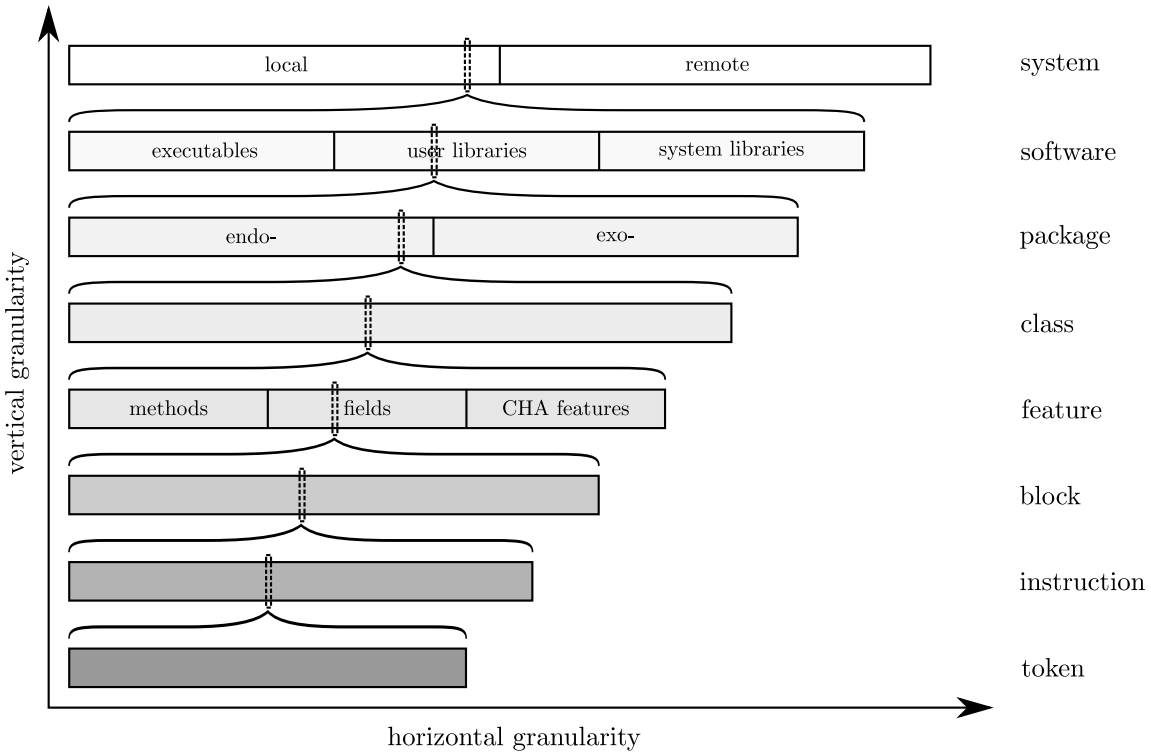


Figure 2.7: Software granularities for Java programming language (not exhaustive).

For instance, in a software graph, application nodes are related to core program concepts while library nodes are related to external library ones. Thus, if the graph is based on a Java program, a class of the software package "Eclipse" may use a class in `java.util` library. The former is an application node, the latter is a library node.

As a consequence, two types of edges exist:

- *app-app edges* application nodes to application nodes, we call them *endo-dependencies*, they express that a core element depends on another core element;
- *app-lib edges* application nodes to library nodes, we call them *exo-dependencies*, they express that a core element depends on a library element.

For example, if a software method calls the `System.out.println` method, it is an exo-dependency as this method is member of the Java standard library. Exo-dependencies can occur at each granularity level, even if in lower ones, it is less meaningful to take this concept into consideration.

Figure 2.8a emphasizes endo-dependencies and Figure 2.8b shows exo-dependencies with dashed arrows crossing the system boundary.

2.1.3.5 Software Graph Types

A large number of graphs can be obtained from a program, each one focusing on particular characteristics. Hence, nodes and edges can have various meanings. A common observation is that software graphs are generally directed.

In the remaining of this section, we present some of the most common software graphs. However, this list is far from being exhaustive. We present here only graphs which are

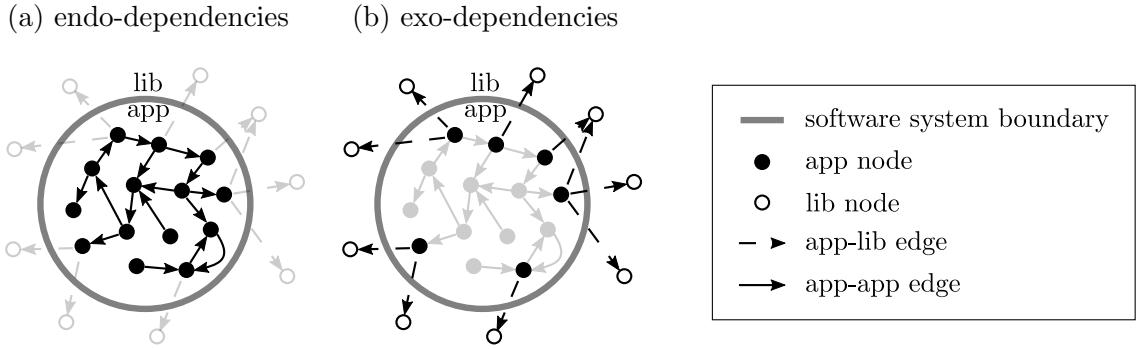


Figure 2.8: Illustration of the set of (a) endo-dependencies and (b) exo-dependencies in a dependency graph.

produced from the program source code. Other types of graphs exist (such as the collaboration graph materializing relations between developers and the source code) but are out of the scope of this thesis.

Dependency Graph a high-level graph materializing the dependence relationship existing between modules of same nature (e.g., packages, classes). A concept `A` depends on a concept `B` if `A` makes any type of usage of `B`. Thus, in this graph, nodes are modules (e.g., packages, classes) and edges reflect that an element uses another one (e.g., function call, inheritance, field access). As an example, the *package dependency graph* reports which package depends on another; nodes are packages and an edge is added from package `A` to package `B` if any piece of code defined in the package `A` depends on (i.e., uses) any piece of code defined in the package `B`. A *class dependency graph* illustrates Java classes usages: each node is a class and each edge illustrates the fact a class uses another class (no matter how, e.g., method, field, constructor, ...). Dependency graphs for packages, classes and features granularities can be obtained in Java using the DepFinder tool [2].

Call Graph (CG) represents the interactions between subroutines (e.g., functions, methods) of a program. Grove et al. [56] define "*the program call graph [as] a directed graph that represents the calling relationships between the program's procedures (...) each procedure is represented by a single node in the graph*". Thus, in a call graph, each node is a subroutine and each edge is the invocation of a subroutine by another one. The source node of an edge is named *caller* and the destination node is named *callee*, the edge connecting a caller to a callee represents a *call site*. The call graph is a dependency graph at the feature vertical granularity level including only method calls. However, there is no unique definition and many types of call graphs can be obtained from a same program. Ryder has been one of the first to publish a paper about creating a call graph [138]. It is possible to include or not some language specific concepts. As an example, the *Class Hierarchy Analysis (CHA) call graph* [41] is a well-known type of call graph which includes inheritance concepts. Call graph are largely studied in related works for years [112, 151, 159].

Control Flow Graph (CFG) illustrates all paths that can be taken by a procedure execution. On such a graph, each node is a block (defined in Section 2.1.3.4) and an edge is added every time there is a flow from a block to another. It was first introduced by Allen [8] in 1970.

[2] `http://depfind.sourceforge.net/`

Data Dependence Graph (a.k.a. as Def/Use Graph) exposes the dependence of data in a basic block [125, 66]. In this graph, each node is a statement and an edge is added when there is a data dependence between two statements. Let `S` and `T` be two statements, `T` follows `S`. There is a *data dependence* from `T` to `S` if there is a common variable between both statements which is assigned by `S` and read by `T`. The opposite situation is called an *anti-dependence*. If both statements assign a common variable, then they are *output dependent*.

Interprocedural Control Flow Graph (ICFG) [65, 84] is an adaptation of the Control Flow Graph for interprocedural calls (i.e., functions and methods). An Interprocedural Control Flow Graph is a graph which contains the union of all control flow graphs we can obtain for each procedure of the program.

Program Dependence Graph (PDG) is made of two subgraphs: the control flow graph and the data dependence graph. Similarly as the control flow graph, it is intended to represent only one single-procedure programs. This graph has been introduced in 1987 by Ferrante et al. [49, 67]. A generalization is the *System Dependence Graph (SDG)*[68, 67] which is a collection of PDG, one for each procedure of the program.

Abstract Syntax Tree (AST) [5] is used to represent syntactic code elements (i.e., tokens such as `if`, `return`, etc.) in a hierarchical view. This is a special case as the abstract syntax tree is a tree. Each syntactic element is presented as a node and each descending edge (children nodes) describes in more details the content of the upper node.

Many other graphs have been proposed in related works. As an example, Callahan presents the Program Summary Graph [35] which includes parameters and global variable flows. Harrold and Mallory [59] proposes the *Unified Interprocedural Graph*, a merge of four types of graphs including the call graph, the program summary graph, the interprocedural control flow graph and the system dependence graph. The *Annotated Dependency Graph* (ADG) proposed by Hassan and Holt [62] which is a dependency graph annotated with information obtained from the source code repository.

2.1.3.6 Static vs. Dynamic Software Graph Extraction

Software graphs are extracted from a program by analyzing it. This analysis can be made in a static or a dynamic manner.

Static analysis consists in exploring the files content to obtain the required information used to extract the graph. These files include source files, bytecode, binaries, etc. The graph is thus obtained by examining declared code in these files and enumerating all possible connections based on the specification of the program.

Dynamic analysis works on different artifacts as we do not examine the software files as in static analysis, but extract the graph from the behavior of the program when it is executed. Elements such as logs, test results, stack traces, etc. are used to gather software information and build a graph. Depending on the type of data to extract, and especially for building a graph, dynamic analysis can require a prior instrumentation phase to force the program to report required information.

The major difference between the two approaches is that a static analysis is exhaustive as all possibilities are taken into consideration. The dynamic approach is more context-

dependent as obtained information is dependent on the execution context such as input values, system state, etc.

As an example, if a class `A` defines a `foo` method and contains subclasses which override a method `foo`, a static approach will consider that invoking the `foo` method on an object `A` may in fact be an invocation in any overridden `foo` method. In a dynamic approach, only one of these methods would be invoked, depending on the context of the execution.

Both can be seen as a strength and as a weakness. Static analysis is exhaustive, which means that too much information may be extracted resulting in imprecise or non-realistic results (i.e., never-occurring scenario). Moreover, the amount of obtained information can be quite important, but the main advantage is that as no execution is required, the needed time to compute a graph this way may be shorter than with a dynamic approach.

Dynamic analysis depends on the execution scenario, which means it may miss some cases which would occur in other execution scenarios. As a consequence, its results may be appropriate only for a specific execution, the scenario may miss a result `A` which would occur in a large number of other scenarios. As a consequence, it results in a smaller amount of information but the execution time is generally more important as the program has to be executed (and even instrumented before).

2.2 Change Impact Analysis

In this section, we present works related to change impact analysis which are of interest for this thesis. These are related to contributions 1 and 2 presented in Section 1.3. In contribution 1, we propose a framework for evaluating change impact analysis techniques; in contribution 2, we present a novel technique for change impact analysis based on past impacts and call graphs.

In Section 2.2.1 and 2.2.2, we introduce respectively the taxonomy and terminology for change impact analysis. In Section 2.2.3, we illustrate change impact analysis based on a modern example. Then, in Section 2.2.4, we present techniques based on call graphs; in Section 2.2.5, ones based on other types of graphs and in Section 2.2.6, other techniques of interest. In Section 2.2.7, we discuss these related works.

A lot of literature has been proposed for years. Curious readers which want a broader view of change impact analysis can refer to surveys such as the reference book of Bohner and Arnold [26], the general survey including the graph-based approach by Li et al. [88] and Lehnert [86], the survey by Tip [150] and another by De Lucia [40] which extensively studies change impact analysis using slicing techniques.

Bohner has defined Change Impact Analysis (a.k.a. CIA) as "the determination of potential effects to a subject system resulting from a proposed software change" [25]. Indeed, software is made of interconnected pieces of code; for instance, at the method level, methods are connected when a method calls another one. Through these connections, the effects of a change in a given part of the code can propagate to many other parts of the program. Acting like a ripple, these other parts may potentially be anywhere.

2.2.1 Taxonomy of Change Impact Analysis

Many categorizations of change impact analysis exist in related works. Bohner and Arnold propose two types of analyses: *dependency analysis* and *traceability analysis* [26]. The former analyzes the source code of the program at a relatively fine granularity (e.g., methods call, data usage, control statements, ...) while the latter compares elements at a coarser granularity such as documentation and specifications (e.g., UML).

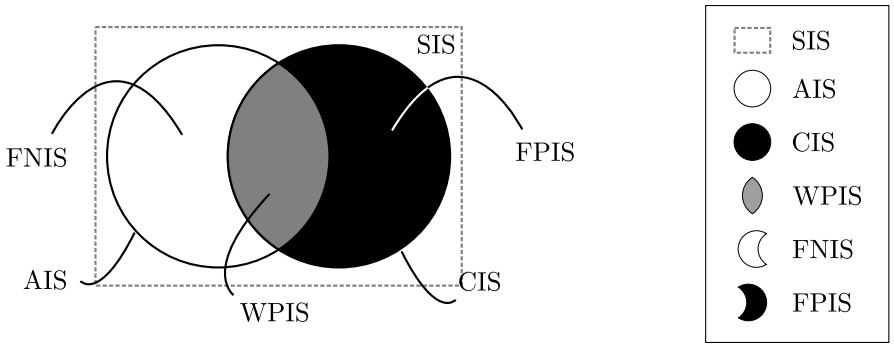


Figure 2.9: Illustration of Bohner's sets

Kilpinen [76] has proposed a taxonomy in 2008. According to this, a change impact analysis technique belongs to one of the three groups:

1. *Dependency* impact analysis for techniques which study internal elements of the source code at a relatively fine granularity (e.g., methods, classes);

2. *Traceability* impact analysis technique if the technique cross data of different abstraction (at coarser granularity) levels such as documentation, specifications (e.g., UML) and source elements;

3. *Experimental* impact analysis technique made of an approach which requires manual inspection of software artifacts.

Lehnert [87, 86] propose a more complex and structured taxonomy to compare change impact analysis techniques. It categorizes proposed techniques according to several criteria. Main criteria include:

1. the scope of the analysis (static/dynamic/online source code, architecture/requirements model or miscellaneous artifacts such as documentation or configuration);

2. the granularity;

3. the used techniques are divided in ten different categories including call graphs and dependency graphs;

4. the experimental result (based on the size of the system, the precision, the recall and the time).

Lehnert's taxonomy is quite complex, we do not report here the entire taxonomy.

2.2.2 Terminology of Change Impact Analysis

In this thesis, we use the terminology of Bohner [25] intended for expressing impact prediction problems. In this terminology, the SIS set (i.e., Starting Impact Set) contains all software parts that can potentially be impacted by a change.

When an element in the program is changed, Bohner terminology proposes these sets to deal with the estimated impacts of a technique:

Candidate Impact Set (CIS) made of software parts predicted as impacted by a change impact analysis technique;

Actual Impact Set (AIS) made of software parts which are actually impacted by the change. This set is also called the *"Estimated Impact Set" (EIS)* [13].

False Negative Impact Set ($FNIS$) made of missed impacts by the change impact analysis technique. Bohner names this set the "Discovered Impact Set" (DIS), but this naming can be confusing in our context, reason why we rename it;

False Positive Impact Set ($FPIS$) made of overestimated impacts returned by the change impact technique.

Moreover, to complete this list of sets, we propose the "*Well-Predicted Impact Set*" ($WPIS$) made of software parts predicted as impacted by a change impact analysis technique and which are actually impacted by the change. All these sets are subsets of the SIS set. Equations (2.6), (2.7) and (2.8) formally define these sets using the AIS and CIS sets. Figure 2.9 proposes an illustration of these sets.

$$WPIS = AIS \cap CIS; \tag{2.6}$$

$$FNIS = AIS - CIS; \tag{2.7}$$

$$FPIS = CIS - AIS; \tag{2.8}$$

2.2.3 Modern Implementation of Change Impact Analysis

Integrated Development Environments (a.k.a. IDE) are the primary tools used by developers to write and maintain software. An IDE provides a large range of features to ease the whole pipeline of creating programs. Its assistance ranges from modeling to testing and running, with advanced features such as code generation and refactoring.

A well-known and useful IDE feature consists in finding method callers. In this thesis, we refer to such a feature as *FindCallers*. Given a method `foo()`, the FindCallers feature returns to the developer the set of methods calling `foo()`. This feature can be called recursively on a found method to explore deep chained calls through the program code. This feature is of great importance: when a developer changes a piece of code, he can ask which methods depend on the changed one. The developer is able to identify the methods depending on it and thus prevents errors due to the propagation of the impact from this change. Thus *the FindCallers feature can be seen as an impact analysis tool.*

Two well-known IDEs are Eclipse [3] and IntelliJ IDEA [4]. Both IDEs include the FindCallers feature. Figure 2.10 shows the corresponding UI for IntelliJ. As we can see, the IDE shows recursively the methods calling `set(int,int,int)` of the `MutableObjectId` object. By simply clicking on the gray triangle next to the method label, the IDE computes the next set of methods calling the selected method. This process can be repeated recursively to explore the call stack as deep as we want. Note the figure also shows the recursive calls, which means a method calling one already called can be expanded indefinitely. In the Figure, we observe that the `NoteMapMerger.mergeFanoutBucket(...)` method calls the `NoteMapMerger.merge(...)` method which calls back the `NoteMapMerger.mergeFanoutBucket(...)` and so forth indefinitely.

In practice, many terms are used to refer to the *call site* concept, presented in Section 2.1.3.5: a *reference* (in Eclipse), a *call* or *usage* (in IntelliJ, as shown in Figure 2.10). In this thesis, we consider these terms are equivalent and we use the term *call site* to the maximum possible extent.

[3] `http://www.eclipse.org`
[4] `http://www.jetbrains.com`

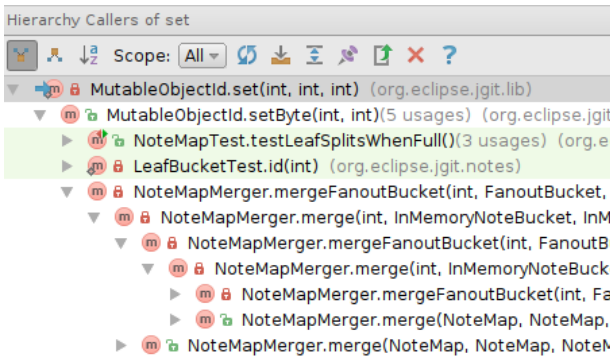


Figure 2.10: An example of the FindCallers feature in IntelliJ IDEA 2016.1. The IDE shows recursively the methods calling the method `MutableObjectId.set(...)` in the Jgit project.

2.2.4 Call Graph-based Approaches

Ryder and Tip [137] have proposed an approach for change impact analysis based on static call graphs. Their approach is divided in two phases: first, they extract atomic all changes from changes observed between two versions of a program. Secondly, they use static software call graph to estimate the impacted tests of each change and changes that may affect a test.

Ren et al. [134, 133] proposes a concrete implementation as an Eclipse Plugin, of the approach presented by Ryder and Tip entitled *Chianti*. They use atomic changes obtained from a version control system and call graphs in order to find the number of affected tests. They also compare, for each test, the average percentage of changes affecting it. They perform an evaluation on 20 commits of a program. In [134], they base their study on static call graphs while in [133], they rely on dynamic ones.

Challet and Lombardoni [38] propose a theoretical reflection about impact analysis using function call graphs; they refer to these as "bug basins". However, no implementation or evaluation is proposed in the paper.

Zhang et al. [163] propose a new approach entitled "FaultTracer" which is based on Chianti [133]. They propose an "extended call graph" to improve Chianti results for affected tests.

Law and Rothermel [85] propose an approach for impact analysis; their technique is based on a code instrumentation to analyze execution stack traces. They compare their technique against simple call-graphs on 38 real fault of the "Space" project.

Badri et al. [17] perform impact prediction based on control call graphs: a flow graph in which each non-method call instructions are removed. They empirically evaluate their method on two projects and take as a baseline the call graph. To obtain an actual impact set, they use a version control system to find changes. They report that the control call graph is more precise and is a perfect trade-off between the call graph and the analysis made by Rothermel by simply observing set sizes and not using metrics.

German et al. [51] propose to use a time aware approach to predict impacts. They propose the "Change Impact Graph", a call graph containing time information about the last version of methods. They mark nodes which have changed and nodes which are affected by these changes. Based on these, they prune the graph to use a smaller graph. They assess their finding on only one C program and study only 5 bug fixing cases.

2.2.5 Other Graph-based Approaches

Many authors propose reflections and formal models to think about the concept of propagation. In 1990, Luqi [99] proposed a formal graph model for software evolution. Podgurski and Clark [129] proposed a discussion around semantic and syntactic dependence based on control and data flow graphs. Three years later, Loyall and Mathisen [96] extended this reflection to an interprocedural extent and proposed a prototype system for software maintenance. This prototype has been tested on a program in ADA [97, 98].

Li and Offutt [90] propose an approach for estimating change impact of object-oriented software based on control flow graph (CFG) and data flow graph (Def-Use). The author discuss the nature of changes (e.g., inheritance) and their impacts. No concrete evaluation is proposed.

Robillard and Murphy [136] introduce the "concern graph" for reasoning on the implementation of features (e.g., methods calls, fields accesses, object creation). They propose the "Feature Exploration and Analysis Tool (FEAT)" used by developers to explore the generated graph. They propose two case studies involving developers which have to use the FEAT tool in their development tasks. However, no empirical evaluation is performed.

Hattori et al. [64, 63] use an approach named *Impala*, based on a graph made of entities (i.e., class, methods and fields) to study the propagation. This graph contains edges marked by the type of dependency. Their approach consists in a set of algorithms for obtaining the impacted nodes based on a depth criterion to stop searching propagation after n calls. They retrieve concrete commits in a version control system, isolate a change and use Impala with various depth values to obtain impacts. Then, they compare with real change. Their evaluation is made on three projects. Their main goals is to (i) show that precision and recall are good tools as evaluation of the performance for impact analysis techniques; (ii) illustrate a correlation between precision and depth criterion.

Walker et al. [155] propose an impact analysis tool named TRE. Their approach uses conditional probability dependency graphs where nodes are classes and edges are added when a class A depends on a class B. The conditional probabilities are extracted from a version control system and defined by the number of times two classes are changed on the same commit. Then, the technical risk for a change is determined based on the conditional probability dependency graph. They work on the class granularity and give no concrete information about the evaluation.

Zimmermann and Nagappan [167] propose to use dependency graphs to estimate the most critical parts of a piece of software. Their approach uses network measures and complexity metrics to make the predictions. They assess their findings using some popular though proprietary software, where they are able to determine software parts that can cause issues.

Petrenko and Rajlich [128] propose to use a graph made of entities coming from three dependency levels: class, method and field in a graph entitled *Class and Member Dependency Graph*. This graph should lead to a more precise impact analysis. This is built according to the Java AST, it is thus a static analysis. They assess their result on two programs, with different versions from a version control system. They report a 100% recall and precision values; however, no baseline is considered to compare with.

Breech et al. [30] propose to use value propagation information to improve two existing approaches: the icfg propagation [96] and the PathImpact [85]. They use propagation information presented as *influences*, resulting in the concept of "influence graph". Authors assess their finding on eight C programs, by computing precision and recall metrics. However, they just compare the size of impact sets without ground truth.

Huang and Song [69, 70] propose a dynamic impact prediction technique specially designed for object-oriented systems. In their first version [69], after instrumenting the code and applying *Execute after-set* to get traces for knowing who calls who for a change, they take method calls and fields access into consideration for related items by analyzing the dependency graph (estimated by object type access). In their second version [70], they also take the CHA aspect into consideration. Their evaluation is made on Java software; their baseline is the CollectEA approach [12]. They show on small cases that they have no imprecision on their set compared to CollectEA. They state an improvement in precision because of the impacted set size reduction. Their baseline is obtained by comparing different versions from a version control system.

Maia et al. [103] propose a hybrid approach mixing static and dynamic approaches to improve predictions. The static approach is rather similar to the one presented by Hattori [63]. The dynamic one is a trace printing exclusively for method entering/leaving and field reading/writing. Moreover, they rank entities from the static analysis with the dynamic one. They assess their finding with five programs and compare false positives and negatives, precision and recall against Impala [63] and CollectEA [12]. It is better for the recall but not for the precision. Data is taken from a version control system.

2.2.6 Other Approaches

Sun et al. [148] propose an evaluation framework for static change impact analysis. It operates at the class granularity. The actual changes are obtained from code and bug repositories. They evaluate three change impact analysis techniques (Columbus, ROSE and IRC^2M) based on five Java programs.

Do and Rothermel [44] describe a protocol to study test case prioritization techniques based on mutation. They use mutants to determine which test is impacted by the change.

Some authors take a machine learning approach for the change impact analysis problem. Jeong and Kim [71] use machine learning on bug tossing graphs (graphs exposing the reassignment of a bug) for bug triage. Kim et al. [77] use machine learning on crash graphs which is an aggregated view of multiple crashes. However, both graphs are representations of the relationship between bugs and not software parts. Elish and Elish [47] use support vector machines to predict defect-prone modules in four projects. Tian and Noore [149] predict software cumulative failure time using neural networks. Anvik et al. [11] use classification to do bug triage (affecting a bug to a developper).

In 1990, a classic paper by Moriconi and Winkler [108] studied error propagation, but they did it with the goal of having a perfect approximation. Their approach is based on inference rules. Their work is more theoretical in essence, assessed only on small toy examples.

Michael and Jones [105] alter variables during the program execution in order to study how this affects ("perturbates" in their phrasing) the program. They focus on data-state perturbation.

Apiwattanapong et al. [12] propose a dynamic impact technique based on code instrumentation and their idea comes from the `CoverageImpact` [124] and `PathImpact` [85]. They assess their approach on two programs. Test cases are entry points for a dynamic analysis. The impact to predict is determined by comparing different versions obtained from a version control system.

Ahsan and Wotawa [6] use labeling and classification on bug reports data to improve prediction of defects. They compute the precision and the recall, but based on expected sets from bug reports.

Antoniol et al. [10] also address impact analysis by taking as input software documentation artifacts (i.e., bug reports or modification requests). Gethers and colleagues [52] adopt a similar methodology.

Cai et al. [34] propose a novel technique for impact prediction. Their technique is dynamic and requires a code instrumentation phase. It is based on assigning sensibility measurement to statements of a static slice. Moreover, their implementation is not publicly available for a quantitative comparison.

Binkley et al. [24, 23] propose observation-based slicing (a.k.a. ORBS). They propose to slice a piece of software in a "delete–execute–observe" paradigm. In this paradigm, effects of a change are observed after (e.g., by running test cases). Their technique focuses on a quite low granularity (i.e., statements).

2.2.7 Discussion

The evaluation of a change impact analysis requires two sets to be compared together: the actual and the candidate impact set, that is respectively the real and the estimated impacts. Most works in the change impact analysis literature try to obtain the actual impacts from real changes. This process is based on extracted information from different versions of a software repository by doing source code difference. Obtaining the data this way cannot guarantee that we are looking at an isolate change, which can result in bias in result interpretation. Moreover, it forces authors to define and isolate atomic changes which is a time-demanding process. Thus, a large part of contributions focus on the meaning of a change in order to extract atomic changes from these. As a consequence, a large number of publications in change impact analysis suffer from a lack of large empirical evaluations. To our knowledge, no author has proposed a technique to obtain automatically a large set of atomic changes from a specific program.

2.3 Fault Localization

Fault Localization is the process of finding the position of a fault in a program source code. Fault localization can be manually handled by a developer or can be achieved automatically using advanced techniques.

Automatic fault localization is a field of software engineering which aims at automatically identifying a faulty element in a source code. A faulty element can be of any granularity (e.g., statement, method, class). An automatic fault localization is based on an algorithm intended to identify suspicious elements in the source code.

In essence, the fault localization process tries to capture causality relationships between code elements. Indeed, early works in fault localization were based on program slices [4], which are refined versions of the most obvious causal relationship: the bug must lie somewhere in the code that has been executed.

In this thesis, we mainly consider three families of fault localization techniques: spectrum-based, graph-based and mutation-based ones. However, curious readers are invited to read surveys on the topic such as ones proposed by Steinder and Sethi [146] or by Wong et al. [158].

2.3.1 Spectrum-based Fault Localization

Spectrum-based fault localization is a popular fault localization technique. In these techniques, "*spectrum*" refers to the behavioral traces of the program when executing the test

cases [60].

A spectrum-based fault localization algorithm generally attributes a *suspiciousness score* to code elements. The *suspiciousness score* is the likelihood for a code element to be faulty. The higher the score is, the higher are the chance the element is the faulty one. Once all code elements contain a suspiciousness score, the developer can examine code elements in descending order of their suspiciousness score. It is computed based on test execution results (i.e., passing/failing tests).

The performance of a spectrum-based fault localization algorithm can be determined by computing its *wasted effort*. The wasted effort is the number of elements the developer will examine before examining the faulty one. The lower the wasted effort is, the faster the developer will find and fix a fault.

Spectrum-based fault localization techniques are also causal to a certain extent, but with a really strong approximation: the causal relations are only captured by the fact that an element is covered by passing or failing test cases. As most fault localization techniques are *spectrum-based*, we concentrate on such approaches in this thesis. Moreover, we refer to automatic spectrum-based fault localization as "fault localization" for short.

The state-of-the-art in spectrum-based fault localization necessarily includes Tarantula by Jones et al. [73]. Another well-cited metric is is the Jaccard and Ochiai ones [2]. Xie et al. [160] propose a theoretical analysis on multiple ranking metrics of fault localization and divide these metrics into categories according to their effectiveness.

Santelices et al. [139] combine multiple types of code coverage to find out the faulty statements in a program. Baah et al. [16] employ an outcome model to find out the dynamic program dependencies for fault localization. Xu et al. [161] develop a noise-reduction framework for localizing faults. DiGiuseppe and Jones [42] has recently proposed a semantic fault diagnosis approach, which employs natural language processing to detect the fault locations. Xuan and Monperrus [162] develop a learning-based approach to combine multiple ranking metrics for fault localizing.

To our knowledge, only Baah et al. [15] and Shu et al. [143] have set notable milestones using causal inference for better approximating causal effects in fault localization.

2.3.2 Graph-based Approaches

Stoerzer et al. proposed "JUnit/CIA", a variant of Chianti which classifies each change as green, yellow or red [147]. The classifier is defined according to the nature of the change: improved and/or worsened test execution. This reduces the set of methods to inspect. They propose an implementation as an Eclipse plugin (extension of the Chianti Eclipse plugin).

Three years later, Ren et al. proposed a heuristic for ranking the method changes which cause a test to fail [132]. This heuristic is based on graph metrics: the number of ancestors and descendants in the call graph for the changed method is linked to the probability to be responsible of the failure. This probability increases even more if caller and callee are changed together. They assess their approach on three pairs of versions of the Eclipse JDT project.

Zhang et al. [163] propose an approach entitled "FaultTracer" also based on Chianti [133]. Their approach (i) extends the dynamic call graph used by Chianti with field access information; (ii) uses fault localization approaches (i.e., suspiciousness score) in order to prioritize the candidate changes. Their evaluation is compared to a personal implementation of Chianti (as the code is not available) for 22 failures on four projects.

They consider four fault localization suspiciousness score metrics: Tarantula [73], SBI [92], Jaccard and Ochiai [2].

Ko and Myers [80] propose an original approach entitled "Whyline". Their approach is not directly intended for fault localization in the conventional way. Instead, they propose a debugging tool on which the developer can ask questions based on the execution trace. These questions are related to a behavior of the program during the execution and can be, per se, seen as a fault. "Whyline" uses different approaches such as static/dynamic slicing, static call graphs and other algorithms to explain what is the cause of such a consequence. They assess the performance of their tool on five Java programs.

Zhao et al. [165, 166] propose the "FP" technique. It uses coverage information obtained from a program control flow graph to establish suspiciousness of code blocks. They assess their approach with three C programs from the SIR benchmark suite [43]. They compare their result with the classic Tarantula approach. Moreover, they also improve their result by creating an hybrid approach: Tarantula approach is used and, after inspecting 10% of the code, it switches to FP to speed up the process.

Renieris and Reiss [135] propose an approach based on software traces. They consider two types of traces: traces of successful runs and of faulty ones. Their approach tries to find the successful run which is closer to the faulty one. Reports are produced and scored using the program dependence graph (PDG).

Liu et al. [94] propose a machine learning approach. They use graph mining and support vector machine (SVM) on "software behavior graphs" in order to classify program execution traces.

Some authors propose to use regression models as a fault localization algorithm. Baah et al. [15] build a regression model using spectrum information and data extracted from a program dependence graph (i.e., at the statement granularity). They assess their approach on faults from the Siemens, Sed and Space datasets. Shu et al. [143] use a similar approach but based on dynamic call graphs (i.e., at the method granularity) mixed with dynamic data dependencies. They assess their approach on faults randomly selected from a bug database for four programs.

2.3.3 Mutation-based Approaches

Mutation-based fault localization has been recently proposed. The central idea of mutation-based fault localization is to localize faults by injecting faults.

Baudry et al. [20] leverage the concept of dynamic basic blocks to maximize the ability of diagnosing faults with a test suite. They base their study on biology concepts and propose to use software mutation to simulate faults. They use the JMutator framework to mutate code according to seven mutation operators on two Java object-oriented programs. The approach proposed by Baudy et al. to use mutants as simulated faults has been investigated by Ali et al. [7]. They conclude that there is "no reason to believe that mutants are unsuitable as candidates for faulty versions for the purpose of studying FL algorithms".

Zhang et al. [164] propose "FIFL", a fault injecting approach to localize faulty changes in evolving Java programs. This approach is based on the "FaultTracer" approach [163], itself based on Chianti [133]. In this variant, mutation is used to inject fault in order to improve the fault localization results. Here, candidate changes are ranked based on the suspiciousness of mutants, resulting in an improvement of FaultTracer efficiency. They assess their model on 26 pairs of versions taken from nine projects. They consider four

fault localization suspiciousness score metrics: Tarantula [73], SBI [92], Jaccard and Ochiai [2]. The mutation part of the contribution is done using the Javalanche framework [140].

Mun et al. [111] propose the "MUtation-baSEd fault localization technique (MUSE)": an approach based on both test suite and mutation of faulty and correct statements in the program. Suspiciousness in this approach is computed based on the change of test execution after mutation, i.e., tests that fail (or pass resp.) before mutation and do pass (or fail resp.) after. Moreover, they propose the "Locality Information Loss (LIL)" metric for determining how a fault localization technique is suited for automatic fault repair/human debuggers. They assess their approach on fourteen faulty versions from five C projects from the SIR benchmark suite [43]. They use the "Proteum mutation" framework [104] for their approach.

Papadakis and Le Traon [126] develop "Metallaxis-FL", a mutation-based technique for fault localization on C programs. Their work shows that test cases that are able to kill mutants can enable accurate fault localization based on the point where the mutation occurred. Suspiciousness score is computed using the well-known Ochiai formula [2]. They evaluate their approach on two sets of C programs: the Siemens suite composed of 132 faults in seven programs (enriched of 100 mutant-based faults) and 40 faults from four projects from the SIR benchmark suite [43]. Similarly as Mun, they use the "Proteum mutation" framework [104].

2.3.4 Discussion

A large number of contributions in fault localization are based on the supposition that the faulty method must lie in between a test and the executed statements. Most spectrum-based fault localization contributions obtain these statement by software slicing. The limitation with this approach is that the technique is based on a limited view of the program. Indeed, as slicing reports lines of code, they do not consider how these lines are in relation to one another. That is, there is no information of how these statements are related (e.g., in which method, which other method calls them).

On the other side, some approaches are based on a more global view of the problem. An example of such approaches is ones which use software graphs for fault localization. Indeed, software graphs offer the advantage to materialize all interconnection information between software elements (which may be of different nature, depending on the chosen granularity). However, to our knowledge, no works propose to conciliate both aspects, that is, the one reported by spectrum-based fault localization and the other with a more general insight of software such as one could obtain by using software graphs.

2.4 Generative Models of Software Data

As presented in Section 1.2, our second intuition is that synthetic data is a good candidate for obtaining missing or hard-to-obtain data. In the first to third contribution, we use mutants as synthetic data for simulating software faults.

In the last part of this thesis, we want to explore other types of synthetic data such as synthetic software graphs. In this section, we explore existing graph models (related or not to software engineering).

Oxford dictionary defines a model such as "a simplified description, especially a mathematical one, of a system or process, to assist calculations and predictions". Thus, a generative model is an algorithm which produces random observable data. A generative model takes a set of parameters as input (such as the number of nodes or a threshold).

A generative model may be *deterministic* or *stochastic*: for a given set of parameters, a deterministic one always generates the exact same graph whereas a stochastic one generates a new graph each time it is run. A generative model for graphs is an algorithm that generates graphs.

2.4.1 Generic Generative Models

The simplest and earliest ones are the models independently introduced by Gilbert, Erdös and Rényi in 1959, both known as *Erdos-Rényi models*[48]: ER(n, m) generates a random graph with n nodes and m edges chosen uniformly at random and ER(n, p) generates a random graph with n nodes and any pair of nodes being connected with a probability p. These are very nice theoretical models, but no graph observed in practice is ER: they have properties that real graphs do not have and it is not possible to generate ER graphs that share the properties of real graphs.

Watts and Stogatz proposed in 1998 a model for undirected graphs entitled the *Small-world graphs* [156]. It starts from a regular graph of n nodes, each connected to k neighbors. Then, with a probability p (with $p \in [0 : 1]$), it reconnects an edge randomly to another. Then, with p raising to 1, we obtain a random graph such as the Erdos-Rényi one and with intermediate values, we obtain small-world graphs.

Barabási and Albert proposed in 1999 a well-known model entitled *Scale-free graph* [18]. This model uses a *preferential attachment*, which means the more a node is attached (i.e., the number of edges connecting to it), the more it is susceptible to receiving new attachments. In other terms, the graph is generated incrementally, adding one node at each iteration and connecting this node to already existing nodes. The idea is that the more connections a node already has, the more likely it is to be connected to the new node ("rich get richer" principle). Though it is a relatively old idea dating back at least to 1925, this sort of graphs has gained considerable attention these last years. Indeed, it was observed that many real graphs may be described by such a preferential attachment process. Such graphs have the degree distribution (*cf.* Section 2.1.3.2) that usually follows a power law or a log-normal law; there is some controversy on this point [118, 107, 46], also in the field of software engineering [109, 110].

2.4.2 Software Generative Models

Some other authors have proposed models for generating software graphs.

Valverde and Solé use a model of duplication and rewiring to generate software graphs with similar motifs [153]. Moreover, they observe the tendency of software graphs to follow a growth mechanism similar to the one presented by Krapivsky and Redner [82] as well as an asymmetry between the in-degree and out-degree distributions [152]. This asymmetry has been reported by other authors such as Myers [114], Challet and Lombardoni [38] and Baxter and Frean [21].

Myers [114] propose a generative model based on binary strings to materialize the software evolution rules. Baxter and Frean propose a generative model of software graphs [21], based on a preferential attachment which depends on the node degree distributions. Their model requires as input a fixed edge number. Both works [114, 21] study a large amount of graph and software metrics.

Maddison and Tarlow [102] propose a generative model of source code at the abstract syntax tree level. Li et al. [89] also propose a generative model based on the "modular attachment mechanism".

Chaikalis and Chatzigeorgiou [36] propose a graph-based predictive model at the class level. Their goal is to predict the growth and coupling of future versions.

Lin and Whitehead [93] have studied power laws on software change size and they propose a generative model for such distributions. They work a the AST granularity and only focus on change sizes.

Some authors have studied other structure characteristics on different kinds of graphs without proposing any concrete generative model. Louridas et al. [95] study the "pervasive" presence of power-law distributions in software dependency graphs at the class and feature level for a large range of programs written in various languages. Harman et al. [58] focus on dependency clusters to demonstrate the widespread existence of clusters in software source code. Mitchell and Mancoridis [106] use clustering techniques to infer an aggregated view of a software system; their goal is to improve debugging and refactoring. However, they do not focus on generalities about software.

2.4.3 Discussion

In related works, a large number of graph models are proposed in various domains, and only a few are intended for software graphs. As this topic tends to be quite theoretical, many contributions are only at the theory stage, sometimes demonstrated with mathematical tools, but less commonly assessed empirically. Almost no implementation can be found online and only a small number of authors propose a pseudo-code or algorithmic description of their approach. The lack of available implementations makes empirical comparisons with existing models difficult: the definition of these models may be misunderstood, leading to wrong implementations. To the best of our knowledge, no author has proposed a generative model for software dependency graphs which is extensively assessed in an empirical manner, based on a large corpus of real software graphs.

3

An Evaluation Framework for Change Impact Analysis

"I have not failed. I've just found 10,000 ways that won't work."

— Thomas Alva Edison

This chapter at a glance...

In this chapter, the following contributions are presented:

- an algorithm to numerically analyze the accuracy of an impact analysis technique based on mutation testing;
- the definition of four kinds of call graphs for impact prediction.
- a large-scale impact prediction experiment on 10 open-source Java projects and 16,922 mutants comparing these four kinds of call graphs.

The following publications are related to this chapter:

[1] Vincenzo Musco, Martin Monperrus, and Philippe Preux. An Experimental Protocol for Analyzing the Accuracy of Software Error Impact Analysis. In *Proceedings of the 10th International Workshop on Automation of Software Test co-located with ICSE*, AST '15, pages 60–64, 2015.

[2] Vincenzo Musco, Martin Monperrus, and Philippe Preux. A Large-scale Study of Call Graph-based Impact Prediction using Mutation Testing. *Software Quality Journal*, 2016. To appear.

Change impact analysis consists in analyzing and reporting potential impacts for a specific change in a software package. One challenge in this research area is to propose an approach for automatically assessing the performance of an impact prediction technique over a large dataset.

Let us explain this challenge: the performance of a large number of prediction techniques proposed in related works are poorly or not assessed in an empirical manner (i.e., with real data). When an assessment is presented, it is generally based on a small amount of changes. This is due to the commonly used approach: real changes are obtained from different versions of a program using the version control system.

The problem with this approach is that a commit does not necessary exhibit one single change. Thus, it implies one more precondition: changes must be split down in atomic changes. Moreover, even once these changes have been reverted, observing the impact to which they relate is not straightforward. In our work, we propose a solution allowing to assess the performance of a technique based on a large number of changes without dealing with such constraints.

In this chapter, we present a novel evaluation framework for change impact analysis. Our key idea for producing a large number of changes in a program is to use mutation testing. Indeed, mutation testing will create a large number of copies of the program source code, on which one small change is introduced in each. This is based on the observation made by Ali et al. [7] that there is no reason to think that a software mutant cannot amount to a software fault.

To reason on the impact of a change, we use the test suite execution. We use test failures as a way to observe the actual impact of a change in the program. Indeed, if the test passes before the change and fails after, there are great chances that this is due to the change itself. To assess the performance of a change impact analysis technique, we use precision and recall metrics to express how accurate the technique is.

To do so, we propose to run our evaluation framework on call graphs used for change impact analysis. However, as there is no unique call graph definition *per se*, we investigate the performance for impact prediction of four types of call graphs, each one having its particularity. The first taken from an external implementation, and the three others are proposed by us: with and without call hierarchy analysis and taking into consideration data dependencies.

We run our protocol on 10 mainstream open-source Java software packages. For each of them, we create a total of 16,922 mutants using 5 different mutation operators. Then we compare the precision and recall of the prediction depending on the call graph that is used. Our results show that the sophistication indeed increases the completeness of impact prediction (higher recall). However, and surprisingly to us, the simplest call graph gives the highest trade-off between precision and recall for impact prediction (as computed by the F-Score).

This chapter is structured as follows. In Section 3.1, an overview of the approach is presented. In Section 3.2, four types of call graphs for impact prediction are presented. Section 3.3 lists the research questions on which we focus in this chapter. In Section 3.4, our experimental evaluation is explained. It is based on four metrics: the precision, the recall, the F-score and the completeness. Moreover, the dataset, the configuration and results of this work are presented. Section 3.5 concludes this chapter.

3.1 Main Algorithm

We propose a novel approach for evaluating a change impact analysis technique I. The evaluation is based on the concept of actual impact set and candidate impact set presented in Section 2.2.2. The closer the actual impact set and the candidate impact set determined by I are, the more accurate is the technique.

Our evaluation consists in assessing repetitively the impact prediction technique I with a changed version of a program to determine how accurate the prediction is. These changed versions of the program are artificially obtained using mutation testing which are then used for determining the accuracy of a change impact analysis technique.

Software mutants are used here as a way to simulate artificial faults. Indeed, a mutation consists in a random change in the source code. This is likely to result in an unexpected (i.e., faulty) behavior, and thus in failing test cases. Running the test cases on mutants produces the actual impact set of failing tests. These failing test cases are the actual impact set.

Then, using a change impact technique I, the candidate impact set is obtained.

Algorithm 3.1 illustrates the global process of generating changes, obtaining the actual impacts (i.e., the AIS – Actual Impact Set) and the estimated impact (i.e., the CIS – Candidate Impact Set) using an impact prediction technique I.

Algorithm 3.1: Computes the candidate and actual impacted sets using mutation injection, test execution and call graph.

Input: Σ the software package. I an impact prediction technique. m_{op} a mutation operator.

Output: a map containing for each mutant (key) the CIS and AIS sets.

```
1  begin
2      IP ← empty_map()
3      T ← testCases(Σ)
4      for each e in filterElements(Σ, m_op) do
5          for each m in mutants(Σ, e, m_op) do
6              if m compiles and is killed then
7                  CIS_m ← impactedTests(m, I)
8                  AIS_m ← failingTests(m, T)
9                  IP_m ← {AIS_m, CIS_m}
10     return IP
```

This algorithm takes as input:

1. the software package under study;
2. an impact prediction technique;
3. a mutation operator that is responsible for mutation injection.

The output of the algorithm is a map which contains for each mutant, the actual impact set (AIS) and the candidate impact set (CIS). In line 3, we get the set of test cases (`testCases`) from the input software Σ. In lines 4–6, we select, mutate and test the appropriate elements in the software package. The selection operation is made with the `filterElements` function and the mutation operation is done with the `mutants` function.

Table 3.1: The four types of call graph we define for error impact prediction.

Name	Hierarchy	Fields	Description
$\mathscr{C}_S$	No	No	Call graph extracted using JavaPDG [142].
$\mathscr{C}_B$	No	No	Call graph considering only method calls. Calls to inherited methods are resolved.
$\mathscr{C}_H$	Yes	No	$\mathscr{C}_B$ with Class Hierarchy Analysis (CHA), a standard call graph in object-oriented static analysis.
$\mathscr{C}_F$	Yes	Yes	$\mathscr{C}_H$ with field analysis: each read/write access to a field may propagate an error.

Appropriate elements are syntactic entities to which the specific change can be applied. In line 7, we determine the test cases impacted by the mutation (`impactedTests`) according to the impact prediction technique I (i.e., the candidate impact set). In line 8, the function `failingTests` returns the set of test cases that fail when running the mutated version of the program (i.e., the actual impact set).

Some mutants are said to be unbounded. An *unbounded mutant* is a mutant for which an impact prediction technique is not able to predict anything because of a lack of information (which is different from predicting no impact). The reason for which this happens is related to the prediction technique under consideration. For the one considered in this thesis, the unbounded mutants are discussed in Section 3.2.

3.2 Application to Call Graph Based Impact Prediction

In this chapter, call graphs are used to obtain the candidate impact set. Thus, our evaluation framework enable us to evaluate the impact prediction potential of a call graph.

Call graphs model how software methods are called. If an error is present in a software method, methods calling it may themselves be impacted by the error. Exploring the call graph is a way for estimating the impact of a change. As an example, a `drawSquare` method calls a `drawLine` one. In the resulting call graph, there is an edge such as `drawSquare` $\longrightarrow$ `drawLine`. If the `drawLine` method has been changed, this is likely that the `drawSquare` method which calls it (i.e., depends on it) will be also impacted by the change.

Figure 3.1 illustrates an example of error-introducing change. This figure is based on real data obtained in our experiments. This illustration includes both concepts: the call graph *per se* and the Bohner sets presented in Section 2.2.2. Each node represents a method and each edge represents a call to a method. The blue cross is the node where the change (i.e., the mutation) occurs. Purple stars are missed test cases (i.e., detected only by the test suite execution), red diamonds are incorrectly predicted test cases (i.e., predicted by the call graph but not by the test suite execution), green boxes are correctly predicted test cases (i.e., found by both techniques) and black circles are application nodes. As an example, the graph illustrated on Figure 3.1 is composed of 78 nodes with 13 correctly predicted, 7 missed test cases, 36 incorrectly predicted nodes and 21 application nodes. In the example, we notice the multiple propagation paths that exist from the node in which the error has been introduced to the impacted test nodes.

Call graph is defined in such a way that it allows many variations. Thus, in this chapter, we consider a family of four different types of call graphs we use for impact pre-

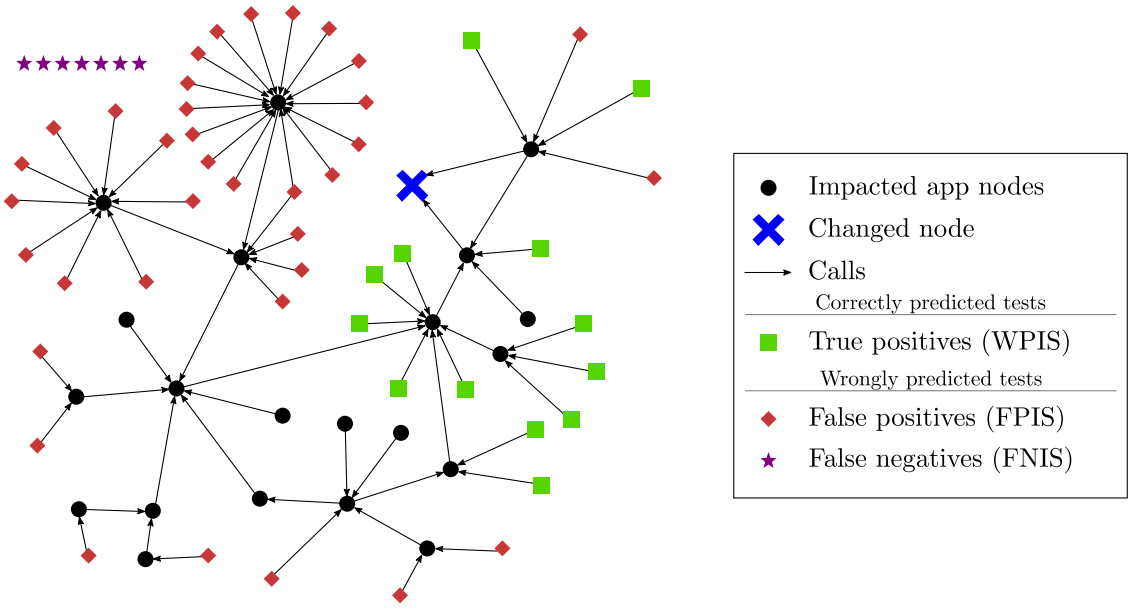


Figure 3.1: Visualization of the effect of a particular mutation. Black circles are the nodes that propagate the mutation injected in the node denoted with a blue cross. Nodes illustrated by green boxes, red diamonds and purple stars are test cases related to the injected mutation. Green boxes nodes are test cases that are correctly predicted as impacted by the injected mutation; these are true positives. Red diamonds nodes are test cases that are predicted as impacted, but are not; these are false positives. Purple stars nodes are test cases that should have been predicted as being impacted but have not been; these are false negatives.

diction. These are listed in Table 3.1 and their key differences are illustrated in Figure 3.2. Each member of this family abstracts a particular way error may propagate in a piece of software. A discussion about the reason we choose to use call graphs as a change impact prediction technique is presented in Section 3.4.5.1. To the best of our knowledge, no author has proposed to use such variants of call graphs from the point of view of change impact prediction. Consequently, no accuracy comparison study of these call graphs has ever been made before.

The first one is the call graph obtained using the JavaPDG tool by Shu et al. [142]. We refer to such a call graph as $\mathscr{C}_S$, where "S" refers to the first author of the paper. In such a call graph, overriding methods are not resolved. Thus, if the method `A.foo()` overrides the method `B.foo()`, and the method `C.bar()` calls `A.foo()`, the call graph will contain a call from `C.bar()` to `A.foo()`. Figure 3.2 gives another example of this point: methods `biz1()` and `biz2()` both call the same method, but the former calls it on an object `B` and the latter on an object `A`. However, in the call graph, both are resolved with the same node.

$\mathscr{C}_B$ is a similar basic call graph which uses the signature of the class according to the static type of the receiver, as illustrated in Figure 3.2. We see that in this call graph, the method `biz1()` and `biz2()` both call a `foo()` method, but the former on a `B` object and the latter on a `A` object. Formally, for $\mathscr{C}_S$ and $\mathscr{C}_B$, if method `m` calls method `n`, there is an edge $node_m \longrightarrow node_n$. However, errors may propagate through edges that are neither in $\mathscr{C}_S$ nor in $\mathscr{C}_B$. Thus, we propose two other flavors enriching $\mathscr{C}_B$ by handling some object-oriented programming concepts.

$\mathscr{C}_H$ takes class hierarchy analysis into consideration (a.k.a. CHA)[41] to take inheritance and interface implementation into consideration. To do so, for each method, we

(a) Source code

```
public abstract class A{
   abstract void foo();
}

public class B extends A{
   public void foo(){
      // B.foo() body
   }
}

public class C{
   int bar = 0;

   public void biz1(){
      B aB = new B();

      bar = 1;
      aB.foo();
   }

   public void biz2(){
      A aA = new B();

      if(bar > 0){
         aA.foo();
      }
   }
}
```

(b) Extracted call graphs

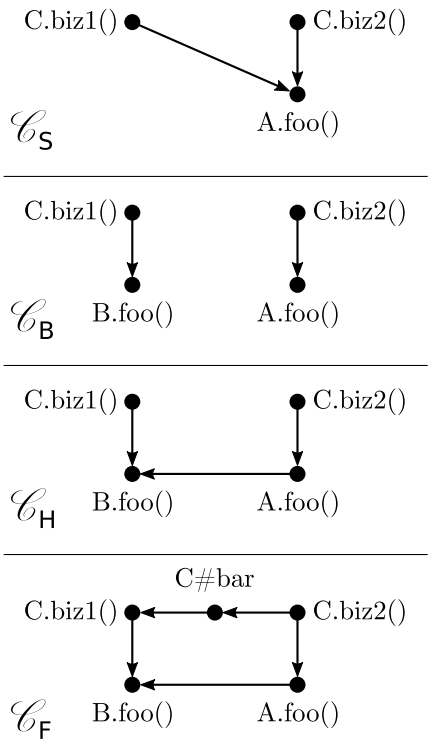


Figure 3.2: (a) a simple Java source code and (b) the four types of graphs obtained from it: $\mathscr{C}_S$, $\mathscr{C}_B$, $\mathscr{C}_H$ and $\mathscr{C}_F$.

explore the classes extended and the interfaces implemented by the class in which the method is defined. We add edges from the parent definition method to the overridden method in the hierarchy. Formally, if a method `m` implements an abstract class or an interface method `n`, there is an edge $node_n \longrightarrow node_m$. This is illustrated on Figure 3.2 where we observe that an edge has been added from `A.foo()` to `B.foo()`.

$\mathscr{C}_F$ takes class hierarchy analysis but also reads and writes to fields into consideration. Indeed, when a method writes to a field, it modifies its content and thus, potentially inserts an error in it. In the opposite situation, a method which reads a variable in which an error has been inserted may be impacted by this error. Thus, when writing to a variable, the propagation goes from the method to the variable, but on the opposite way, when reading, the propagation goes from the variable to the method. Formally, if method `m` reads the field `f`, there is an edge $node_m \longrightarrow node_f$. If `m` writes the field `f`, there is an edge $node_m \longleftarrow node_f$. This is illustrated on Figure 3.2: a node has been added for the `bar` field and two edges have been added: one from `C.biz2()` to `C#bar` for the read operation and one from `C#bar` to `C.biz1()` for the write operation. This feature is similar to the method-level data dependency edge presented by Shu et al. [143] with the difference that we add a node and two edges between the calls where they directly add an edge. However, from a propagation point of view, both approaches are totally equivalent.

Figure 3.3 illustrates a simple application of the Algorithm 3.1 using a call graph as an impact prediction technique. Three types of nodes are presented: application nodes (plain circle), test nodes (circle with a T) and the changed node which is itself an application node (double circle). The `mul` method is a multiplication method. As we can see, both the power method (`pow`) and the factorial method (`fac`) use the multiplication one. Moreover,

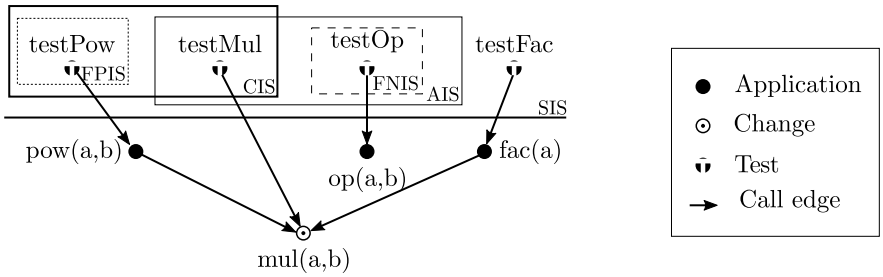


Figure 3.3: Example of a call graph in which a change has been introduced. It includes application nodes, test nodes and call edges. The rectangles illustrate Bohner's sets.

another operand method (`op`) is also defined but not called explicitly in the call graph. This method uses reflection (which is not resolved statically) to call the `mul` method, resulting in the absence of edge between `op` and `mul`. Moreover, each method has its associated test method prefixed by `test`.

All test nodes belong to the *System Impact Set (SIS)* and are all potentially impacted. We use mutation injection to produce a change to the `mul` method. Running test cases on the changed version of the code gives a list of failing and passing test cases. As these results are obtained by the execution of the program, the failing test cases make the actual impact set (AIS). In this example, we suppose that there are two actually impacted test cases: `testMul` and `testOp` illustrated by the thin box.

We use call graphs as an impact prediction technique. This is achieved by computing transitive closures. A *transitive closure* $T(n)$ for a node n returns the list of all nodes which can be reached by any path from n. As we can see in our example, to determine which nodes are connected to the impacted one (the `mul` method node), we need to compute the transitive closure by *recursively exploring the edges in the reverse direction.* Therefore, we reach two test nodes: `testPow` and `testMul`. These form the candidate impact set (CIS), illustrated by the thick box.

We observe two other sets. The `testPow` test method is a false positive as it is reported as impacted by our impact prediction technique but does not actually fail when running the test cases. This test case belongs to the *False Positives Impact Set (FPIS)* illustrated by a dotted box. On the other hand, the `testOp` test method is a false negative: running the test cases reports this test method as impacted, but there is no path from the impacted method (`mul`) to the `testOp` test method. This test case belongs to the *False Negatives Impact Set (FNIS)* illustrated by a dashed box.

Let us now discuss the case of unbounded mutants presented in Section 3.1. $\mathcal{N}$ contains all mutants for which the prediction is not possible because of the call graph. This happens for different reasons: certain call graphs such as the $\mathscr{C}_S$ may contain only nodes corresponding to the first definition of a method and do not resolve the inherited ones. Thus if the change occurs in an overridden method, it would not be found in the call graph. Another scenario is when the mutation occurs in a method which is defined but not actually called in the code (e.g., as `equals`).

3.3 Research Questions

In this section, we present the research question we want to answer in this chapter.

Research Question 3.1 What is the difference between the different types of call graphs in terms of impact prediction accuracy? We determine the prediction capabilities offered

by each call graph and whether field analysis and inheritance analysis improve or decrease the prediction of error propagation.

Research Question 3.2 Is impact prediction project-dependent or mutation-dependent? It may happen that one call graph is good for predicting the error propagation given a specific mutation operator. This is what we call mutation-dependent error impact prediction. The same question may be raised regarding projects. Answering this question allows us to determine the level of genericity of our approach.

Research Question 3.3 What are the reasons of the bad accuracy of impact prediction using call graphs? To answer this question, we manually investigate some cases where the prediction is poor to better understand the reasons leading to a discrepancy between predictions and the actual execution of code.

Research Question 3.4 What is the trade-off between the accuracy and the time needed to compute the impact prediction? Running the test suite is a good and precise way to know the actual impact of a change, but this requires important execution time. On the other hand, a method based on call graphs is cheap in time but less precise in its prediction. As explained above, it may over-estimate or under-estimate the actual propagation of a change. We want to better characterize the trade-off between accuracy and time needed for impact analysis.

Research Question 3.5 How rich are the test executions? In our context, the richness of a test relates to the number of program elements (i.e., methods) involved in testing. We would like to know whether many nodes are involved in error propagation.

3.4 Experimental Evaluation

In this section, we present our evaluation protocol as well as the dataset and mutation operators considered. Then, based on these, we present our results.

3.4.1 Evaluation Protocol

We present here the metrics used for assessing the performance of a change impact analysis technique based on the approach presented in Section 3.1. Section 3.4.1.1 presents metrics intended for computing the performance of one fault (i.e., mutant). Then, Section 3.4.1.2 presents metrics used to compute the global accuracy over all considered faults.

3.4.1.1 One-Impact Mutant-Level Accuracy Metrics

In this section, we define 3 metrics used to analyze the output of Algorithm 3.1 for each mutant (i.e., each prediction). These 3 metrics quantify and characterize the accuracy of an error impact analysis.

The *precision* P is the proportion of test cases predicted by the impact prediction technique which are actually impacted. It is computed using Equation (3.1). The *recall* R is the proportion of test cases predicted by the call graph with regards to all test cases that are actually impacted. It is computed using Equation (3.2). The *F-score* F combines both metrics by computing their harmonic mean as in Equation (3.3). The precision, recall, and F-score are computed for a given mutant m. We have:

$$P_m = \frac{|AIS_m \cap CIS_m|}{|CIS_m|} \tag{3.1}$$

$$R_m = \frac{|AIS_m \cap CIS_m|}{|AIS_m|} \tag{3.2}$$

$$F_m = 2 \times \frac{P_m \times R_m}{P_m + R_m} \tag{3.3}$$

where vertical bars such as $|E|$ denote the cardinality of the set E.

3.4.1.2 Global Accuracy Metrics

In this section, we present the metrics used to determine the accuracy of a change impact analysis technique I as a whole. This is a global accuracy over observations made on results over all impacts presented in Section 3.4.1.1.

Let $\mathcal{K}$ be the set of all killed mutants considered in a given experiment. A mutant is considered as killed as soon as at least one test case fails after running the mutant, while it did not fail on the un-mutated version of the program. The accuracy of a change impact prediction technique is characterized by the average of the precision (P), the recall (R) and the F-scores (F) over all elements of $\mathcal{K}$.

Moreover, inspired by Arnold et al. [13], we define four sets to categorize the four types of possible predictions: $\mathcal{S}$ (*same*), $\mathcal{O}$ (*overestimate*), $\mathcal{U}$ (*underestimate*) and $\mathcal{D}$ (*different*). These are based on Bohner's sets presented in Section 2.2.2. Each mutant belongs to either one of these 4 sets. For a given mutant, we compute $FPIS$ and $FNIS$. Then, there are four cases:

- if $FPIS = FNIS = \emptyset$, the mutant belongs to the $\mathcal{S}$ set. It implies that the CIS and the AIS are strictly equal ($AIS \cap CIS = AIS = CIS$)). $\frac{|\mathcal{S}|}{|\mathcal{K}|}$ is the proportion of cases for which our method finds all and only actual impacts, which implies we cannot do better predictions for these cases;

- if $FPIS \neq \emptyset$ and $FNIS = \emptyset$, the mutant belongs to the $\mathcal{O}$ set. In this case, we have $AIS \subset CIS$. The change impact analysis technique is able to determine all impacts but it over-estimates them as it returns more impacts than actually happens. These scenarios are not perfect but are considered as safe [13] as they return at least all the impacted elements;

- if $FPIS = \emptyset$ and $FNIS \neq \emptyset$, then the mutant belongs to the $\mathcal{U}$ set. In this case, we have $CIS \subset AIS$. The change impact analysis technique under-estimates the impact set as it returns less elements than the number of elements actually impacted;

- if $FPIS \neq \emptyset$ and $FNIS \neq \emptyset$, then the mutant belongs to the $\mathcal{D}$ set. The change impact analysis technique returns different impacts than the actual ones (even if some impacts may be estimated correctly).

In the two last cases, the change impact analysis technique under study misses impact candidates. Equation (3.4) shows that these 4 sets are disjoint, and each killed mutant belongs to either one of these 4 sets. $\{\mathcal{S}, \mathcal{O}, \mathcal{U}, \mathcal{D}\}$ is a partition of the set of killed mutants $\mathcal{K}$.

$$|\mathcal{S}| + |\mathcal{O}| + |\mathcal{U}| + |\mathcal{D}| = |\mathcal{K}| \tag{3.4}$$

Algorithm 3.2 describes how each mutant is assigned to a set. As an example, the previously presented example Figure 3.1 belongs to the $\mathcal{D}$ set as there are missed and incorrectly predicted test cases.

Algorithm 3.2: Computes the sets $\mathcal{S}$, $\mathcal{O}$, $\mathcal{U}$ and $\mathcal{D}$ for a set of mutants and their actual and candidate impact sets.

Input: IP the map containing each mutant and its actual and candidate impact sets (obtained using Algorithm 3.1)
Output: $\mathcal{S}$, $\mathcal{O}$, $\mathcal{U}$ and $\mathcal{D}$: sets of mutants as defined in the text.

```
1  begin
2      S ← O ← U ← D ← ∅
3      for each m in IP do
4          AIS, CIS ← IP_m
5          FPIS ← CIS − (AIS ∩ CIS)
6          FNIS ← AIS − (AIS ∩ CIS)
7          if FPIS = ∅ and FNIS = ∅ then
8              S ← S ∪ {m}
9          else if FPIS ≠ ∅ and FNIS = ∅ then
10             O ← O ∪ {m}
11         else if FPIS = ∅ and FNIS ≠ ∅ then
12             U ← U ∪ {m}
13         else
14             D ← D ∪ {m}
15     return S, O, U, D
```

We also define the set $\mathcal{C}$ (*complete*) as being the set of mutants for which the candidate impact set contains all actually impacted methods, maybe more. In other words, for these mutants, the change impact analysis method does not miss any impact. Formally, we define the set $\mathcal{C}$ by Equation (3.5).

$$\mathcal{C} = \mathcal{S} \cup \mathcal{O} \tag{3.5}$$

Table 3.2: Statistics about the projects considered in this chapter.

Project Name				
Short	**Full**	**Version**	**Commit**	**LOC**
Codec	Apache Commons Codec	1.11	r1676715	17,531
Coll.	Apache Commons Collections	4.1	r1610049	55,081
Gson	Google Gson	2.3.2	#fcfd397	20,072
Io	Apache Commons Io	2.5	r1684201	26,528
Jgit	JGit	4.1.0	#3c33d09	133,865
Joda.	Joda-time	2.8.1	#6da4053	85,000
Lang	Apache Commons Lang	3.5	#6965455	67,509
Shindig	Shindig	2.5.3	r1687149	15,710
Sonar.	Sonarqube	5.2	#1385dd3	29,342
Spojo	Spojo	1.0.7	#8fb2194	3,371
Total				**454,009**

We define the completeness as $p_{\mathcal{C}} = \frac{|\mathcal{C}|}{|\mathcal{K}|}$. It quantifies the extent to which a given call graph approximates the impact of a given mutation. $p_{\mathcal{S}} = \frac{|\mathcal{S}|}{|\mathcal{K}|}$ quantifies the extent to which a given call graph perfectly determines the impact of a given mutation.

Unbounded mutants $\mathcal{N}$ presented in Section 3.1 belong to the $\mathcal{U}$ set ($\mathcal{N} \subset \mathcal{U}$). The precision and recall for these unbound mutants are both equal to 0. Thus, mutants in $\mathcal{N}$ set are removed from $\mathcal{K}$ (and consequently from $\mathcal{U}$). Clearly, the set of mutants belonging to $\mathcal{N}$ depends on the call graph being used. This point is visible in the experimental section, where we give the cardinality of $\mathcal{N}$ for each type of call graph we work with.

3.4.2 Dataset

We consider a dataset composed of 10 *Java* software packages. It is composed of the following projects: *Apache Commons Lang*, *Apache Commons Collections*, *Apache Commons Codec*, *Apache Commons Io*, *Google Gson*, *Jgit*, *Jodatime*, *Apache Shindig*, *Spojo* and *Sonarqube*. When the project is made of several subprojects, we consider only the main one. Tables 3.2 and 3.3 report the key descriptive statistics about these projects. Table 3.2 gives the name, the version, the git commit-id (starting with #) or the svn revision number (starting with 'r') and the number of lines of code (computed using *cloc*[1]) of the program being analyzed. Table 3.3 describes the different call graphs under investigation. This table is made of four pairs of columns which give the number of nodes and edges composing each call graph, namely $\mathscr{C}_S$ (call graph obtained using JavaPDG tool), $\mathscr{C}_B$ (our basic call graph), $\mathscr{C}_H$ (our call graph with CHA) and $\mathscr{C}_F$ (our call graph with CHA and fields).

We observe that the $\mathscr{C}_S$ contains less nodes and edges than $\mathscr{C}_B$, $\mathscr{C}_H$ and $\mathscr{C}_F$ (excepted for *Gson*, where the $\mathscr{C}_S$ has more nodes than $\mathscr{C}_B$). This is due to the fact that $\mathscr{C}_S$ does not resolve the inherited method name, which means that if a method `A.foo` calls a method `B.bar` which extends `C.bar`, the graph only contains calls to the super method `C.bar`.

Since we have the same number of nodes for $\mathscr{C}_B$ and $\mathscr{C}_H$, this validates our implementation because we just added calls between some classes belonging to the same hierarchy. These methods are already present in $\mathscr{C}_B$, they are just called by the callee. In $\mathscr{C}_H$, we add edges between methods belonging to the same hierarchy, (i.e., overridden methods). The number of nodes and edges increases in $\mathscr{C}_F$ because we introduce nodes and edges to reflect fields and their use (reads and writes).

3.4.3 Mutation Operators

Our technique requires mutation operators. We consider the five mutation operators presented by King and Offutt [79]. As shown by Offutt et al., these five operators are sufficient to effectively implement mutation testing [122]. These operators are listed in Table 3.4: the leftmost column is the three letter acronym used by King and Offutt, the central column is the full name, and the rightmost column lists the set of operators implied in the mutation. A mutation operator changes a single atomic element. Any software source code elements may be considered in a mutation.

As these operators are originally intended for the Fortran programming language, we adapted them in order to make them compatible with Java programming language (see the Java operators implied in the rightmost column of Table 3.4). Our Fortran to Java adaptations of the operators are:

[1] `http://cloc.sourceforge.net/`

Table 3.3: Statistics about the call graphs for considered projects.

	$\mathscr{C}_S$		$\mathscr{C}_B$		$\mathscr{C}_H$		$\mathscr{C}_F$	
Project	**#N**	**#E**	**#N**	**#E**	**#N**	**#E**	**#N**	**#E**
Codec	1,338	1,959	1,490	2,218	1,490	2,336	1,884	3,588
Coll.	6,008	7,747	6,678	9,252	6,678	12,047	7,637	17,178
Gson	2,630	5,492	2,480	5,381	2,480	5,674	3,317	9,101
Io	2,382	3,634	2,662	3,974	2,662	4,198	3,305	7,004
Jgit	11,571	31,647	12,560	35,953	12,560	37,679	17,350	60,458
Joda.	8,531	23,283	9,809	31,329	9,809	33,991	11,879	44,956
Lang	6,033	8,892	6,220	9,004	6,220	9,345	7,577	16,094
Shindig	1,410	2,020	1,933	2,373	1,933	2,621	2,723	5,096
Sonar.	3,126	5,025	4,322	5,737	4,322	5,852	5,960	10,706
Spojo	306	630	417	884	417	917	521	1,331
Total	**43,335**	**90,329**	**48,571**	**106,105**	**48,571**	**114,660**	**62,153**	**175,512**

- *Absolute value insertion (ABS)* in which each numerical expression (variable or method call) or literal is replaced by its absolute value;

- *Arithmetic operator replacement (AOR)* in which each arithmetic expression using Java arithmetic operators `+`, `-`, `*`, `/`, `%` is replaced by a new arithmetic expression with the same operands but where the operator is changed into another one of the same family, chosen uniformly at random. Two other mutation candidates are also the left and the right operand alone, after removing the operator and one of the two operands;

- *Logical connector replacement (LCR)* in which each logical expression using Java logical operators `&&` and `||` is replaced by a new logical expression with the same operands but where the logical operator is changed by another one. Moreover, each logical expression may also be mutated by the constants `true` and `false`. Two other mutation candidates are also the left and the right operand alone, after removing the operator and one of the two operands;

- *Relational operator replacement (ROR)* in which each relational expression using Java relational operators `<`, `<=`, `>`, `>=`, `==` and `!=` is mutated to a relational expression with the same operands but where the relational operator is changed with another one. Moreover, each relational expression may be replaced by the constants `true` and `false`;

- *Unary operator inversion (UOI)* in which each arithmetic and logical expression is mutated. Arithmetic expressions are mutated to their opposite value (i.e., multiplied by `-1`), their incremented value (i.e., add `1`) and their decremented value (i.e., subtract `1`). Logical expressions are complemented (i.e., apply the `not` (`!`) Java operator).

3.4.4 Empirical Results

We now address the research questions introduced in Section 3.3. In particular, we present the accuracy for error impact analysis obtained with the different types of call graphs.

Table 3.4: List of mutation operators considered. Java operators T and F stand respectively for *true* and $false$ boolean types. With binary operators, L and R stand respectively for *left operand* and *right operand*.

ID	Name	Java operators
ABS	Absolute value insertion	$java.lang.Math.abs()$
AOR	Arithmetic operator replacement	$+$, $-$, $*$, $/$, $\%$, L, R
LCR	Logical connector replacement	&&, ||, T, F, L, R
ROR	Relational operator replacement	$<$, $<=$, $>$, $>=$, $==$, $!=$, T, F
UOI	Unary operator inversion	!, $++$, $--$

Research Question 3.1 What is the difference between the different types of call graphs in terms of impact prediction accuracy?

To answer this question, we compute the metrics presented in Section 3.4.1.2. Their values are given in Tables 3.5, 3.6 and 3.7. In the three tables, the first and second columns give the project name and the mutation operator and the end of the table is split into four parts, one for each type of call graphs ($\mathscr{C}_S$, $\mathscr{C}_B$, $\mathscr{C}_H$ and $\mathscr{C}_F$).

In Table 3.5, the third column gives the number of killed mutants for the project. The four remaining columns shows the number of mutants for which there is no node in the graph which corresponds to the method being mutated, or for which the corresponding node has no neighbor, i.e., contains no in/out edges. Each column is related to a different type of call graph.

In Table 3.6, the $p_{\mathcal{S}}$ column is the proportion of mutants for which the impact prediction is perfect (the failing test cases obtained from the call graph are exactly the ones obtained by test suite execution). The $p_{\mathcal{C}}$ column is the proportion of mutants for which the impact prediction is complete, i.e., include all failing test cases.

In Table 3.7, the P, R and F columns report respectively the precision, recall and F-score averaged over all considered mutants. For each line, the value in bold font is the best F-score obtained among the four types of call graph.

The first observation is that $\mathscr{C}_S$ has an important number of unbound mutants, more than 50% in some cases such as Codec with ABS mutation operator. The three other call graphs have less unbound mutants. Further investigations show that the main reason of unbound mutants for $\mathscr{C}_S$ is that the mutation occurs in an inherited node method which is not resolved by $\mathscr{C}_S$ (as presented in Section 3.2). For $\mathscr{C}_B$, $\mathscr{C}_H$ and $\mathscr{C}_F$, unbound nodes are always nodes which are isolated (i.e., no connected edges). We noticed that in $\mathscr{C}_B$, $\mathscr{C}_H$ and $\mathscr{C}_F$, these nodes are not always called in the program. Examples of such methods are `equals`, `compare` or some state testing method such as `isInAlphabet` for Apache Commons Codec project.

This explains low scores for $\mathscr{C}_S$ as every unbound mutant has a precision and a recall equal to 0. This strongly highlights the fact this graph is not complete enough to perform good impact prediction analysis.

Considering the F-score values (F), we see that $\mathscr{C}_B$ is the one which gives the best F-scores (in 18 cases out of 50, that is in more than 30% of cases) which indicates it is the best suited call graph for impact prediction. However, $\mathscr{C}_H$ F-scores are close to the $\mathscr{C}_B$ ones: in 10 cases out of 50, $\mathscr{C}_H$ has similar values and in 16 cases out of 50 it has better ones, which means that $\mathscr{C}_H$ is also a good candidate for impact prediction.

Table 3.5: Mutation statistics based on four different call graphs. $|\mathcal{K}|$ is the number of killed mutants. $\mathscr{C}_S$, $\mathscr{C}_B$, $\mathscr{C}_H$ and $\mathscr{C}_F$ are the number of unbound mutants ($|\mathcal{N}|$) for each project and mutation operator based on each call graph type.

Project	Op.	$\lvert\mathcal{K}\rvert$	$\mathscr{C}_S$	$\mathscr{C}_B$	$\mathscr{C}_H$	$\mathscr{C}_F$
Codec	ABS	302	170	5	3	0
	AOR	458	157	6	5	0
	LCR	497	128	9	0	0
	ROR	484	173	14	5	0
	UOI	470	139	6	3	0
Coll.	ABS	220	71	13	9	0
	AOR	380	194	59	15	0
	LCR	332	69	23	16	3
	ROR	381	83	37	16	0
	UOI	386	150	44	17	1
Io	ABS	251	102	6	6	2
	AOR	387	242	4	4	4
	LCR	446	102	4	0	0
	ROR	454	154	10	6	4
	UOI	351	175	4	3	3
Lang	ABS	278	145	2	0	0
	AOR	480	307	14	1	0
	LCR	447	162	8	1	0
	ROR	466	217	7	1	0
	UOI	452	242	8	0	0
Gson	ABS	225	19	5	5	0
	AOR	310	1	0	0	0
	LCR	431	43	32	32	21
	ROR	703	82	28	27	4
	UOI	418	12	12	12	9

Project	Op.	$\lvert\mathcal{K}\rvert$	$\mathscr{C}_S$	$\mathscr{C}_B$	$\mathscr{C}_H$	$\mathscr{C}_F$
Jgit	ABS	199	90	1	0	0
	AOR	386	189	5	0	0
	LCR	334	87	3	2	0
	ROR	356	129	4	1	0
	UOI	405	187	6	2	0
Joda.	ABS	294	109	0	0	0
	AOR	541	205	16	0	0
	LCR	438	143	0	0	0
	ROR	426	116	4	1	0
	UOI	499	168	7	1	1
Shindig	ABS	247	51	19	19	0
	AOR	314	61	26	22	0
	LCR	300	25	25	12	4
	ROR	338	36	23	18	2
	UOI	251	32	21	18	1
Sonar.	ABS	288	68	39	39	3
	AOR	253	39	0	0	0
	LCR	177	17	22	22	5
	ROR	462	87	28	28	7
	UOI	172	20	9	9	1
Spojo	ABS	8	0	0	0	0
	AOR	20	0	0	0	0
	LCR	48	0	0	0	0
	ROR	142	0	0	0	0
	UOI	15	0	0	0	0

If a call graph shows better F-scores, the observation is valid for all mutation operators of the project, which indicates the impact prediction technique is project-dependent (see Research Question 3.2).

Let us now consider the $p_{\mathcal{C}}$ metric, which indicates whether the prediction is sound (in which proportion of mutations impact that actually happens is not missed). From $\mathscr{C}_S$ to $\mathscr{C}_B$, we have an average increase of $p_{\mathcal{C}}$ around 20%. Then, considering fields and hierarchy indeed better captures the error propagation: the $p_{\mathcal{C}}$ metric increases in average of 15% when taking into consideration class hierarchy analysis (from $\mathscr{C}_B$ to $\mathscr{C}_H$) and of 5% when also considering fields ($\mathscr{C}_H$ to $\mathscr{C}_F$). However, if we look at the increase project by project, we see important differences. Considering the inclusion of hierarchy (from $\mathscr{C}_B$ to $\mathscr{C}_H$), Gson and Jodatime have high $p_{\mathcal{C}}$ average increases of respectively 49% and 41%, which implies a high usage of hierarchy in these projects. At the opposite, Collections and Io have lower $p_{\mathcal{C}}$ values with both an average increase of 1.8%. The $p_{\mathcal{C}}$ value can reach values as high as 100% for Spojo. The best increases are for Lang (or Io resp.) with AOR mutation operator where $p_{\mathcal{C}}$ raises from $\mathscr{C}_S$ to $\mathscr{C}_B$ from 31% to 90% (or from 27% to 86% resp.) and for Gson with ROR mutation operator where $p_{\mathcal{C}}$ raises from $\mathscr{C}_B$ to $\mathscr{C}_H$ from 36% to 93%.

A high recall value indicates that the prediction includes the actual impacted test cases. Thus, the complete set value is strongly linked with the recall. Indeed, we observe that the complete set value ($p_{\mathcal{C}}$) is high when the recall value (R) is high. This is also a piece of evidence of the correctness of our experimental evaluation technique.

$\mathscr{C}_B$ gives the best precision values of all other call graphs. Precision decreases when taking into account the hierarchy or the access to fields ($\mathscr{C}_H$ and $\mathscr{C}_F$). This makes us think that more nodes/edges are added, more impacted test cases can be found, which

Table 3.6: Proportion of predictions in $p_{\mathcal{S}}$ and $p_{\mathcal{C}}$ sets for each call graphs. $p_{\mathcal{S}} = \frac{|\mathcal{S}|}{|\mathcal{K}|}$ is the proportion of mutants for which the predicted impact exactly matches the actual one. $p_{\mathcal{C}} = \frac{|\mathcal{C}|}{|\mathcal{K}|}$ is the proportion of mutants for which the prediction is complete (i.e., does not miss any impacted test case).

		$\mathscr{C}_S$		$\mathscr{C}_B$		$\mathscr{C}_H$		$\mathscr{C}_F$	
Project	**Op.**	$p_{\mathcal{S}}$	$p_{\mathcal{C}}$	$p_{\mathcal{S}}$	$p_{\mathcal{C}}$	$p_{\mathcal{S}}$	$p_{\mathcal{C}}$	$p_{\mathcal{S}}$	$p_{\mathcal{C}}$
Codec	ABS	2%	26%	3%	62%	2%	79%	1%	88%
	AOR	3%	40%	3%	70%	0%	81%	0%	86%
	LCR	0%	43%	2%	62%	1%	83%	1%	89%
	ROR	4%	32%	5%	55%	2%	78%	2%	82%
	UOI	3%	42%	4%	66%	1%	79%	1%	85%
Coll.	ABS	12%	20%	25%	35%	25%	37%	18%	41%
	AOR	10%	12%	35%	43%	34%	44%	30%	49%
	LCR	20%	39%	28%	51%	28%	52%	22%	55%
	ROR	12%	21%	18%	32%	16%	36%	13%	39%
	UOI	17%	21%	36%	44%	34%	45%	27%	51%
Io	ABS	13%	38%	23%	71%	23%	72%	21%	79%
	AOR	14%	27%	27%	86%	27%	86%	27%	89%
	LCR	6%	60%	10%	82%	10%	85%	9%	88%
	ROR	14%	47%	24%	76%	24%	79%	20%	86%
	UOI	14%	38%	26%	82%	26%	84%	25%	88%
Lang	ABS	19%	43%	42%	92%	42%	95%	40%	96%
	AOR	9%	31%	40%	90%	40%	95%	36%	95%
	LCR	16%	59%	27%	89%	27%	95%	26%	95%
	ROR	22%	49%	43%	90%	43%	94%	38%	96%
	UOI	15%	40%	41%	88%	40%	93%	35%	94%
Gson	ABS	1%	38%	1%	42%	1%	96%	1%	97%
	AOR	1%	57%	1%	60%	1%	99%	0%	99%
	LCR	2%	37%	2%	38%	1%	81%	0%	85%
	ROR	2%	34%	2%	36%	2%	93%	2%	94%
	UOI	2%	41%	2%	42%	2%	92%	0%	94%
Jgit	ABS	2%	16%	2%	35%	2%	56%	2%	70%
	AOR	1%	18%	3%	44%	2%	62%	2%	69%
	LCR	0%	25%	1%	48%	1%	66%	1%	76%
	ROR	2%	21%	3%	44%	3%	62%	1%	73%
	UOI	1%	19%	3%	47%	2%	65%	2%	73%
Joda.	ABS	14%	33%	19%	50%	19%	86%	19%	89%
	AOR	11%	25%	13%	33%	11%	87%	11%	89%
	LCR	8%	27%	9%	49%	9%	83%	9%	84%
	ROR	16%	35%	19%	47%	19%	81%	18%	85%
	UOI	13%	27%	15%	37%	13%	85%	13%	86%
Shindig	ABS	25%	54%	28%	67%	28%	81%	26%	87%
	AOR	31%	57%	39%	75%	40%	87%	36%	91%
	LCR	14%	54%	15%	60%	15%	71%	15%	78%
	ROR	18%	48%	20%	54%	20%	68%	18%	75%
	UOI	27%	56%	32%	68%	32%	80%	30%	84%
Sonar.	ABS	33%	51%	43%	74%	43%	77%	40%	81%
	AOR	26%	79%	28%	91%	28%	91%	26%	91%
	LCR	25%	57%	27%	68%	24%	76%	21%	78%
	ROR	30%	61%	34%	78%	34%	82%	33%	84%
	UOI	28%	72%	31%	81%	30%	84%	28%	85%
Spojo	ABS	0%	62%	0%	62%	0%	62%	0%	62%
	AOR	0%	0%	0%	10%	0%	10%	0%	10%
	LCR	0%	92%	0%	92%	0%	100%	0%	100%
	ROR	2%	67%	5%	72%	4%	86%	4%	90%
	UOI	0%	73%	0%	73%	0%	80%	0%	80%

Table 3.7: The main metrics of impact prediction based on four different call graphs. P is the precision averaged over all killed mutants, R is the recall averaged over all killed mutants, and accordingly, F is the F-score averaged over all killed mutants.

		$\mathscr{C}_S$			$\mathscr{C}_B$			$\mathscr{C}_H$			$\mathscr{C}_F$		
Project	**Op.**	**P**	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
Codec	ABS	0.24	0.30	0.15	0.43	0.68	**0.31**	0.25	0.87	0.29	0.18	0.93	0.21
	AOR	0.36	0.43	0.23	0.42	0.75	**0.34**	0.25	0.89	0.30	0.19	0.92	0.23
	LCR	0.25	0.49	0.21	0.36	0.68	**0.28**	0.17	0.91	0.24	0.15	0.95	0.21
	ROR	0.37	0.39	0.26	0.51	0.65	**0.39**	0.29	0.88	0.33	0.24	0.90	0.26
	UOI	0.39	0.47	0.26	0.48	0.73	**0.39**	0.28	0.88	0.32	0.23	0.91	0.26
Coll.	ABS	0.57	0.22	0.19	0.84	0.42	0.38	0.74	0.44	**0.39**	0.51	0.49	0.30
	AOR	0.43	0.13	0.12	0.89	0.47	**0.45**	0.77	0.48	0.44	0.58	0.54	0.39
	LCR	0.65	0.41	0.35	0.81	0.56	**0.48**	0.70	0.57	0.47	0.53	0.60	0.35
	ROR	0.66	0.26	0.23	0.82	0.39	**0.33**	0.62	0.44	0.32	0.44	0.47	0.27
	UOI	0.54	0.23	0.22	0.88	0.49	**0.47**	0.75	0.51	0.46	0.55	0.57	0.38
Io	ABS	0.42	0.43	0.32	0.64	0.78	**0.53**	0.58	0.79	0.50	0.50	0.83	0.43
	AOR	0.27	0.29	0.22	0.52	0.89	**0.51**	0.49	0.90	0.49	0.40	0.92	0.38
	LCR	0.32	0.66	0.30	0.43	0.88	**0.43**	0.39	0.90	0.41	0.30	0.92	0.32
	ROR	0.42	0.52	0.36	0.62	0.84	**0.57**	0.55	0.87	0.54	0.41	0.91	0.41
	UOI	0.32	0.41	0.27	0.55	0.88	**0.53**	0.51	0.90	0.51	0.41	0.92	0.41
Lang	ABS	0.32	0.44	0.32	0.68	0.94	**0.70**	0.64	0.98	**0.70**	0.56	0.99	0.61
	AOR	0.20	0.32	0.19	0.68	0.91	0.66	0.61	0.99	**0.67**	0.50	0.99	0.55
	LCR	0.33	0.60	0.35	0.55	0.90	0.55	0.48	0.97	**0.56**	0.45	0.97	0.52
	ROR	0.34	0.51	0.37	0.67	0.94	**0.69**	0.64	0.97	**0.69**	0.54	0.98	0.59
	UOI	0.30	0.41	0.29	0.71	0.90	0.69	0.64	0.98	**0.70**	0.52	0.98	0.56
Gson	ABS	0.27	0.61	**0.24**	0.31	0.66	0.21	0.13	0.97	0.16	0.11	0.98	0.14
	AOR	0.19	0.78	**0.20**	0.17	0.83	**0.20**	0.09	1.00	0.13	0.07	1.00	0.12
	LCR	0.41	0.53	**0.22**	0.47	0.56	0.21	0.24	0.88	0.19	0.20	0.89	0.15
	ROR	0.36	0.54	**0.22**	0.43	0.58	0.21	0.17	0.94	0.17	0.15	0.95	0.15
	UOI	0.35	0.63	**0.24**	0.34	0.68	**0.24**	0.19	0.95	0.20	0.15	0.96	0.17
Jgit	ABS	0.31	0.25	0.12	0.50	0.54	**0.22**	0.23	0.87	**0.22**	0.11	0.94	0.12
	AOR	0.23	0.31	0.13	0.47	0.65	**0.27**	0.23	0.90	0.24	0.14	0.94	0.15
	LCR	0.34	0.38	0.13	0.40	0.63	**0.20**	0.19	0.88	**0.20**	0.11	0.94	0.11
	ROR	0.34	0.33	0.15	0.49	0.62	**0.26**	0.26	0.88	0.25	0.15	0.94	0.15
	UOI	0.25	0.28	0.11	0.45	0.64	**0.24**	0.22	0.91	**0.24**	0.13	0.95	0.14
Joda.	ABS	0.47	0.35	0.25	0.74	0.56	0.39	0.38	0.97	**0.42**	0.35	0.98	0.38
	AOR	0.50	0.27	0.19	0.82	0.38	0.27	0.28	0.97	**0.33**	0.26	0.98	0.30
	LCR	0.46	0.33	0.18	0.61	0.57	0.30	0.29	0.95	**0.33**	0.24	0.96	0.27
	ROR	0.57	0.39	0.28	0.76	0.54	0.38	0.41	0.93	**0.41**	0.37	0.95	0.37
	UOI	0.53	0.30	0.22	0.80	0.42	0.30	0.31	0.97	**0.36**	0.29	0.98	0.33
Shindig	ABS	0.56	0.59	0.43	0.68	0.75	0.56	0.60	0.86	**0.57**	0.53	0.90	0.52
	AOR	0.54	0.60	0.44	0.65	0.79	0.58	0.63	0.87	**0.60**	0.60	0.91	0.57
	LCR	0.55	0.61	0.37	0.59	0.68	0.40	0.51	0.74	**0.41**	0.49	0.80	0.40
	ROR	0.62	0.55	0.38	0.68	0.64	0.45	0.57	0.76	**0.47**	0.53	0.82	**0.47**
	UOI	0.61	0.63	0.47	0.69	0.74	0.56	0.66	0.81	**0.57**	0.62	0.86	0.55
Sonar.	ABS	0.63	0.56	0.49	0.80	0.78	0.66	0.79	0.81	0.67	0.76	0.84	**0.68**
	AOR	0.50	0.81	0.53	0.57	0.93	**0.59**	0.57	0.93	**0.59**	0.51	0.94	0.55
	LCR	0.70	0.63	0.50	0.75	0.73	**0.57**	0.68	0.78	0.53	0.64	0.78	0.49
	ROR	0.60	0.65	0.51	0.70	0.82	0.62	0.69	0.85	**0.63**	0.67	0.86	0.61
	UOI	0.59	0.75	0.51	0.66	0.85	**0.58**	0.64	0.86	**0.58**	0.59	0.87	0.54
Spojo	ABS	0.42	0.62	**0.08**	0.42	0.62	**0.08**	0.42	0.62	**0.08**	0.42	0.62	**0.08**
	AOR	1.00	0.00	0.00	0.45	0.13	**0.12**	0.45	0.13	**0.12**	0.45	0.13	**0.12**
	LCR	0.31	0.96	**0.41**	0.31	0.96	**0.41**	0.28	1.00	0.40	0.27	1.00	0.38
	ROR	0.55	0.69	0.36	0.46	0.78	**0.45**	0.36	0.90	0.44	0.31	0.92	0.39
	UOI	0.63	0.76	**0.49**	0.63	0.76	**0.49**	0.59	0.80	**0.49**	0.58	0.80	0.48

also means, more false positives.

Moreover, the precision varies greatly depending on the mutation operators and project: if we consider $\mathscr{C}_H$, it goes from lower values such as 0.09 for Gson with AOR mutation operator to higher ones such as 0.79 for Sonarqube with ABS mutation operator. This observation underlines again the project-dependent side of the impact prediction technique (see Research Question 3.2).

The four types of call graph under consideration are not equivalent for impact prediction. According to our protocol, the best one is $\mathscr{C}_B$, which does not consider Class Hierarchy Analysis and field analysis. The main reason is that the sophistication of Class Hierarchy Analysis and field analysis increases the recall of impact prediction but decreases too much the precision.

Research Question 3.2 Is impact prediction project-dependent or mutation-dependent?

Let us again consider Table 3.6. Now, we focus on the difference between projects and mutation operators:

- the values differ strongly from one project to another for a given mutation operator (e.g., considering the ABS mutation operator with $\mathscr{C}_B$, 3% in $p_{\mathcal{S}}$ for Apache Commons Codec, 19% for Jodatime and 43% for Sonarqube);
- the values differ less from a mutation operator to another for a given project (e.g., considering the Apache Commons Codec with $\mathscr{C}_B$, values range from 2% in $p_{\mathcal{S}}$ for LCR mutation operator to 5% for ROR mutation operator).

These observations highlight the fact that the accuracy of the call graph impact prediction technique depends more on the project than on the mutation operator. Though instantiated through software projects, this observation really concerns the architecture of the software project, or the development patterns (e.g., extensive usage of hierarchy, of reflection) employed to realize the project.

Similar observations have already been reported in Research Question 3.1: the fact that a call graph shows better F-scores for all mutations operators of the project and the fact that the precision may have really low or high values depending on the project.

Call graph-based impact prediction is influenced by the structure of call graphs and by the mutation operators used in the experiment. The project-dependence is higher than the mutation operator dependence.

Research Question 3.3 What are the reasons of the bad accuracy of impact prediction using call graphs?

The answer to Research Question 3.1 has reported both low and high accuracy of impact prediction using call graphs. To gain even better knowledge about call graphs, we have performed an investigation on a set of cases for which the prediction error is particularly bad. We now discuss our main findings.

Our technique is based on a static call graph. Hence, the call graph does not handle the use of Java Reflection mechanism. The reflection mechanism is resolved at run-time

while the call graph is built in a static manner. Obviously, this leads to discrepancies between the results of our analysis, and the outcome of the execution. However, we may detect the use of reflection in a project since then, the source code refers to some specific classes/packages in the Java library (package `java.reflect`). In practice, we may raise a warning to the user about the use of reflection mechanism so that he would take special care when interpreting the impact.

We also notice that the test cases are not independent from each other. Since mutation analysis is costly, we execute them in parallel. In the case of Apache Common Io, the parallel execution of tests sometimes results in failing test cases, where the failure is due to parallel execution and not the mutation itself. The reason is that Apache Common Io extensively uses the hard drive. As our parallel test cases run on the same hard drive space (i.e., folder), they try to read/write/create identical folders/files. Consequently, some test cases fail due to this parallel I/O but it is not due to the mutant itself. There are different ways to address this problem: the easiest one is to run one instance of test at a time in a manner that the I/O is not shared. Another way is to duplicate the project for each mutation operator in a way that if they run in parallel, each one benefits of its own drive space.

The bad accuracy is related to a low recall and/or a low precision. The low recall of call graph-based impact prediction is caused by missing edges in the graphs (e.g., because of reflection). The low precision is caused by too many edges in the considered graphs, especially for $\mathscr{C}_H$ and $\mathscr{C}_F$.

Research Question 3.4 What is the trade-off between the accuracy and the time needed to compute the impact prediction?

Now that we have a clearer understanding of the precision and recall of call graph-based impact prediction, we concentrate on the execution time of the prediction.

Table 3.8 gives the computation time for each project (column 1) of our dataset. Each time related to the call graph is given for the four types of call graph ($\mathscr{C}_S$, $\mathscr{C}_B$, $\mathscr{C}_H$ and $\mathscr{C}_F$). These times are:

- t_{test}, the time required to run the test suite (column 2);
- the time required to build the call graph for each call graph type (columns 3, 5, 7, and 9);
- the average time of impact prediction based on the call graph, i.e., computing one impact prediction (column 4, 6, 8, 10).

The average time of impact prediction is expressed in milliseconds, for instance it takes 0.11 millisecond in average to make an impact prediction in Apache Commons Codec with $\mathscr{C}_H$.

First, we observe the time needed to generate call graphs with JavaPDG ($\mathscr{C}_S$). We observe that it takes several hours to generate the call graph with all elements. For the smallest project, Spojo, it takes 16 minutes. For the largest, Jgit, it takes almost 2 days of computation. This aspect is linked to the fact that JavaPDG also builds a finer graph (the Program Dependence Graph) before extracting the call graph. Thus, using JavaPDG has an important cost in time.

Table 3.8: Main computation time to run each test suite, to build each call graph and to predict one impact using each call graph.

		$\mathscr{C}_S$		$\mathscr{C}_B$		$\mathscr{C}_H$		$\mathscr{C}_F$	
Project	t_{test}	**build**	**pred.**	**build**	**pred.**	**build**	**pred.**	**build**	**pred.**
Codec	32.2s	3h+	0.09ms	0.78s	0.04ms	0.96s	0.11ms	0.90s	0.17ms
Coll.	38.9s	9h+	2.03ms	4.14s	0.03ms	3.78s	0.08ms	3.98s	0.48ms
Gson	13.7s	2h+	0.34ms	1.01s	0.54ms	1.16s	1.66ms	0.90s	3.23ms
Io	90.1s	3h+	0.21ms	1.51s	0.05ms	0.91s	0.05ms	0.86s	0.35ms
Jgit	195.5s	40h+	5.25ms	10.80s	0.99ms	6.52s	3.48ms	6.03s	40.29ms
Joda.	31.2s	25h+	2.55ms	8.12s	0.61ms	4.92s	10.50ms	4.36s	23.14ms
Lang	40.0s	15h+	1.81ms	2.82s	0.05ms	2.75s	0.06ms	2.75s	0.31ms
Shindig	14.1s	1h+	0.17ms	0.65s	0.02ms	0.58s	0.04ms	0.63s	0.08ms
Sonar.	387.3s	3h+	0.76ms	1.74s	0.02ms	1.42s	0.01ms	1.25s	0.10ms
Spojo	2.7s	16m	0.06ms	0.22s	0.04ms	0.24s	0.07ms	0.25s	0.12ms

If we focus on $\mathscr{C}_B$, $\mathscr{C}_H$ and $\mathscr{C}_F$, we observe that building these graphs takes from 1 to 11 seconds (with an average time of 2.6 seconds) for all projects and call graphs being considered. Furthermore, it takes less than 5 seconds for almost all projects (except for Jgit and Jodatime which are the two largest projects, which both require respectively up to 10.8s and 8.1s). The building process seems to last longer with lighter types of the graph (i.e., $\mathscr{C}_B$) than with heavier ones (i.e., $\mathscr{C}_F$). However, our implementation always lists all elements, but just filter out some nodes depending on the type of graph. Thus, these differences in time are more probably explained by the system load at the moment of the generation.

Once the graph is built, determining an impact takes less than 45ms for all projects, and even less than 5ms seconds for all projects except again for Jgit and Jodatime. These observations also apply to graphs generated using JavaPDG: all predictions are made in less than 5ms (except for Jgit: 5.25ms). We observe that prediction times using JavaPDG are generally larger. These differences are likely to be related to the fact that the graph is obtained by third-party software, the data returned is not exactly the same as ours. Thus, some additional on-the-fly data transformations are required to find good nodes in the graph. Overall, these prediction times are equivalent.

If we compare the prediction for $\mathscr{C}_B$, $\mathscr{C}_H$ and $\mathscr{C}_F$, we can see that the prediction time increases with the size of the program and the graph. Thus, $\mathscr{C}_F$ which is the largest graph (as it contains more nodes and edges) increases the required time for prediction, but this increase remains reasonable (maximum absolute value of 41ms). This is expected since prediction is based on path enumeration in the graph. Prediction with a lighter version of the call graph ($\mathscr{C}_B$) performs impact analysis in less than 1ms for all cases.

Actual impacts can be determined directly by running the program test cases. Thus, if we look at the time required for the execution of the test cases, we see that the minimal time required is 3 seconds for the smallest project (Spojo) and can reach values as high as 387.3 seconds for Sonarqube. The time required to build the call graph is smaller to the time required to run test cases for software. This shows that using call graphs to predict impacts costs less than running test suites. If we consider Apache Common Codec, the time required to build a call graph is more than 33 times smaller than the time required to run the entire test suite. And by comparison, the time to make a prediction is orders of magnitude smaller than the time to run the test suite. This observation is interesting as it underlines the fact that using an impact prediction technique based on a call graph gives a quick insight of what are the consequences to the tests according to a change. Then, it

is possible to run first the returned test to directly find failing ones, which represents a gain of time for the developer.

Furthermore, the call graph has the advantage that during software evolution, many changes would have no impact on it (e.g., changing an operator, shifting a line in the code). Thus, the same call graph can be used for predicting the impact of several simultaneous changes before requiring graph regeneration. Moreover, when it is required to recompute it, the time to build the call graph is reasonable, within seconds for our dataset with a maximum of 11 seconds for Jgit. This makes it possible to use such an impact prediction on the fly in the development environment (IDE).

This opens interesting research avenues, where one first performs very fast approximation of error propagation before performing more sophisticated static analyses. This can even be used in a pre-processing step for a dynamic analysis.

Now consider that large companies have hundreds of thousands of interrelated test cases, as in the case of Google [141]. It is likely that these scenarios will be more and more common, and that low-level, detailed analysis of the computation will fail to scale. We think that such settings will need very fast approximation of impacts. The preliminary performance results we report here, with un-optimized software, make us confident that this is indeed possible.

Call graph-based impact prediction is orders of magnitude faster than actually running the test suite. The time cost to build the call graph is also much smaller. In a software codebase with a very large number of methods and test cases, the imprecision of call graph-based impact prediction is compensated by the gain in execution time. Passing from $\mathscr{C}_B$ to $\mathscr{C}_F$ makes prediction times slower, but these times remains acceptable for prediction and much faster than running test cases.

Research Question 3.5 How rich are the test executions?

In this question, we want to determine what is the number of impacted nodes returned by our call graph-based impact prediction technique.

Figure 3.4 shows box plots for each project with the number of impacted nodes for each ABS mutation found with $\mathscr{C}_B$ call graph. As we can see, most graphs have a size smaller or equal to 100. The median size is low for all projects, ranging from 1 to 10 excepted for Apache Common Codec, JGit and Gson which have respectively 25, 30 and 47 as median size value.

However, there exist complex test scenarios: Sonarqube, Jgit and Jodatime have outliers which go up to respectively 509, 2647 and 4564 impacted nodes. We removed these outliers from Figure 3.4 for readability. Moreover, if we look at Gson, we can see that more than having a higher median size of 47, the size of graphs ranges from 1 to 757. We observe a similar situation for JGit for which the size of graphs ranges from 1 to 367.

If we look at the example on Figure 3.1, the size of the graph is 78 nodes, and the impact propagation is not straightforward.

The richness of a test execution is generally small (lower than 50); however, some cases can be quite complex with impacted nodes set size reaching up to 4,000 nodes.

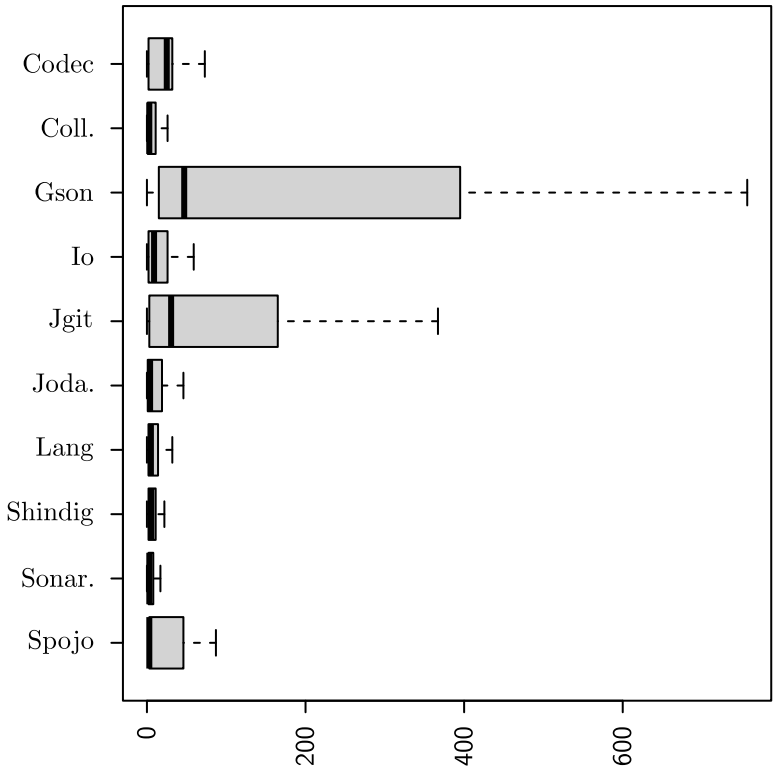


Figure 3.4: Number of impacted nodes for ABS mutants for each project.

3.4.5 Discussion

We discuss here choices regarding the type of graph we considered and limitations, that is, the reasons for which we cannot compare easily with other techniques and the threats to validity.

3.4.5.1 Other Software Graphs

We present here what motivated our choice of the call graph as a change impact analysis technique. Other software graphs can be used for impact prediction Two examples discussed here are:

- the program dependence graph (a.k.a. PDG) which contains more nodes/edges than the call graph as it contains more low-level elements (i.e., code instructions) from the source code;
- the class or the package dependency graph which contains less nodes/edges than the call graph as it contains only dependencies between classes or packages. That is, an edge is added between a class or a package `A` and a class or a package `B` every time a method of `A` access to any element (e.g., class, method, field constant) of the class or package `B`.

If we consider a finer granularity graph (such as the program dependence graph, a.k.a. PDG), it will hardly scale with large programs. This intuition is validated with our first experiments: the time required for building PDG with the well-known program JavaPDG are important (*cf.* Table 3.8). Building a graph for all programs of our experimental dataset took more than 4 days (with an average time per project of more than 10 hours).

To the opposite, coarser granularity (such as a class or package dependence graph) contains less information. An impact prediction for a change in a method `Pkg.Foo.bar()`

is interpreted as a change introduced in the `Pkg.Foo` class or in the package `Pkg`. As a consequence, the resulting impacts are inevitably of the same granularity (i.e., classes or packages). This results in considering all tests of a test class (or a package) as also failing. That is, considering the amount of methods or fields a class can contain (and the number of class a package can contain), the resulting prediction will be inevitably bad, and it would be difficult to precisely locate where the impacts are. To better understand this point, we have computed the class and package dependency graphs for the projects in our dataset. We observe that we have more or less 10 times less nodes from call graph to class dependency graph and less than 30 nodes for the package dependency graph. Similar observation can be made regarding the edges. Now if we consider our smaller project (Spojo) which contains 330 methods (nodes) and 890 calls (edges), we observe that the class dependency graph contains only 37 classes (nodes) and 69 dependencies (edges). These results become even worse with the package dependency graph which contains only 7 packages (nodes) and 13 dependencies (edges).

To sum up, we use call graphs for impact prediction because it exhibits a good trade-off between performance and cost. Moreover as a test is a method, it is also a natural unit of decomposition.

3.4.5.2 Comparison against Impact Prediction Techniques

In this chapter, we focus on characterizing the efficiency of different call graphs for impact prediction (depending on which features we include in the call graph computation – inheritance and fields). Comparing the accuracy of this technique to existing ones is another research question. We wanted to answer to such a research question but this is impossible so far. We identify two reasons that make such a comparative study a complex challenge.

The first reason is that the proposed tools do not necessarily work at the same granularity and/or language. As an example, some may observe code statements or C language [131]. The second reason is that the techniques which can be compared to ours [88] do not provide a publicly available implementation (even by contacting directly the authors). The latter reason is why we make all our implementation publicly available. To sum up, due to lack of open tools, a comparative evaluation of impact prediction on Java software at the level of method is not possible.

3.4.5.3 Threats to Validity

At a conceptual level, the main threat to the validity of our experimental results is that we consider the test suite execution as ground truth. However, it may be the case that the test cases miss the assertions that would detect the actually propagated error and thus fail. This threat is mitigated by our manual analysis.

Our large scale experiment confirms known and yet essential facts to be taken into account when doing mutation analysis. One of such consideration is the fact a single mutant sometimes makes an entire test suite broken. As an example, if we use a static field in a test class which is initialized by default with a mutated constructor then, if the mutation has made the constructor ineffective, it results in an unexpected behavior and an entire test class cannot be initialized. In such a situation, the test suite is reported as failing, and consequently, all test cases belonging to the test suite are reported similarly.

Another example is the fact a test may hang. Indeed, let us imagine the mutation changes a loop condition which results in an infinite looping. To circumvent this problem, we add a timeout for each test. This way, we can determine if some hangs or not. It is

equally important to use a reasonable timeout value for the project to avoid considering a test as hanging when it is not.

3.5 Conclusion

In this chapter, we proposed an evaluation framework for change impact analysis. This framework is based on actual impacts obtained from test suite execution and based on synthetic faults produced with mutation testing techniques. This novel technique is fully automated and enables us to compute standard precision and recall measures.

Specifically, we have executed our protocol on 10 mainstream open-source Java software packages. The analysis of the predicted impact of 16,922 mutants shows that one of the call graphs provides a good trade-off between precision and recall. Moreover, this call graph offers good execution times; this let us use it in real execution scenarios such as real-time tools for assisting a developer while he is editing his source code; it may also be used as a tool for regression test selection.

4

Causal Graph for Change Impact Analysis

"Doubt is an uncomfortable condition, but certainty is a ridiculous one."
— Voltaire

This chapter at a glance...

In this chapter, the following contributions are presented:

- the definition of the causal graph, a weighted call graphs in the context of change impact analysis;
- an approach for learning the likelihood of impact propagation in software, where mutation testing data is used to learn the call graph weights;
- an experimentation made on 9 popular Java open-source programs, totaling 450,000+ lines of code. The experiment shows that our approach recommends impacted method callers accurately.

The following publications are related to this chapter:

[1] Vincenzo Musco, Antonin Carette, Martin Monperrus, and Philippe Preux. A Learning Algorithm for Change Impact Prediction. In *Proceedings of the 5th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering co-located with ICSE*, RAISE '16, pages 8–14, 2016.

[2] Vincenzo Musco, Martin Monperrus, and Philippe Preux. Strogoff: A Recommendation System for Finding Sensitive Method Callers with Weighted Call Graphs.

In Chapter 3, we presented a framework to assess the performance of a change impact analysis. In this chapter, we propose to define the *causal graph*, a new type of graph obtained using a learning perspective on top of class hierarchy analysis call graphs. The causal graph intends to better explain the cause-effect relationship between code elements and failing tests. The causal graph is then used to better filter the returned elements by an IDE.

An *Integrated Development Environment* (IDE) can be considered today as the exoskeleton of professional developers. It enables developers to write code faster, of better quality, with more ease. A typical IDE contains a number of sophisticated features: refactoring, navigation, binding to collaborative development systems such as issue trackers, etc. Research has proposed over time a number of improvements for IDEs, such as breakthrough user-interfaces [29]. Among these proposals, some are recommendation systems. The key intuition is that when a developer manipulates a set of items (either code elements, issues or others), there must be an order and/or filtering that is better than a random or lexicographical one.

One of such recommendation systems is the "FindCallers" feature proposed by an IDE to the developer. Finding callers, a.k.a. "Method Reference Finder", is key for two development tasks in particular: understanding how a method is used and the possible impacts of a change. In this chapter, we propose to take advantage of the change impact analysis potential of call graphs to filter "FindCallers" elements proposed by an IDE to the developer.

We propose to learn how the impact has probably propagated according to a set of changes and their actual impacts. This approach is entitled Strogoff and works as follow. A call graph is built and its edges are decorated with weights ranging from 0 to 1 and representing the likelihood of being subject to error propagation. These weight are learned using the results of mutation testing in an unconventional way. We use a shortest path algorithm to determine the shortest path between a change and a test because the edges belonging to this path are more likely to propagate the impact. The weights are updated using two different approaches: a simpler one which consists in directly setting the highest weight, and a second one which gradually increases it each time the edge is member of a shortest path. Finally, the method call sites are filtered according to the learned weights, i.e., only keeps the sensitive call sites (defined in Section 2.1.3.5).

The performance of Strogoff to filter the "FindCallers" elements is assessed using the evaluation framework (as well as the dataset) presented in Chapter 3. We consider 9 Java open-source application totaling 450,545 lines of code. The corresponding call graphs contain 46,244 nodes and 114,390 edges (call sites). We build a set of 16,682 changes and their actual impact through code mutations. We learn the call graph weights based on these changes with two different algorithms. Then, we simulate scenarios of searching for call sites pointing to a method under consideration and find that our approach can predict the sensitive methods with a precision of 65%, a recall of 76%, corresponding to a F-score of 58%.

This chapter is structured as follows. In Section 4.1, we present an overview of our approach. In Section 4.2 and 4.3, we present respectively the learning phase and the prediction phase: the two main phases composing the whole approach. In Section 4.4, we pose our research question. In Section 4.5, we present our experimental evaluation and our results which allows us to answer to our research questions. In Section 4.6, we discuss the threat to validity of work presented in this chapter. In Section 4.7, we conclude this chapter.

4.1 Approach Overview

A call graph contains information about the methods and the way they call each other. Our intuition is that call graphs are an approximation of causality. Causality is defined by the Oxford dictionary as "the relationship between causes and effects". Indeed, if `A` calls `B`, a bug in `B` might result in a buggy output for `A`. However, this assumption does not always hold in practice because the ideal perfect causal graph is not known. A call graph is indeed only an approximation of cause-effect chains.

In this chapter, we define the *causal graph*, a graph obtained from a class hierarchy analysis call graph (*cf.* Section 3.2) which tries to better approximate the causality. In this chapter, a causal graph is a call graph with class hierarchy analysis made stochastic by adding weights on the call graph edges. These weights range from 0 to 1 where 0 means that the edge never propagates the impact, while 1 means that the edge always propagates it. A value in between means the impact is more or less likely to be propagated, so that sometimes the impact is propagated, sometimes it is not. The initial weights are set to 0, meaning that we start by assuming that no impact is propagated at all. In this approach, we note that nodes are never causal per se, only edges can be deemed as causal.

Mutation testing is used for gathering a learning corpus from which we learn real impacts of a specific change (i.e., the mutation). The causal graph weights are learned from past executions, used to better capture the causality between application and test methods, i.e., the causes (faults in application methods) leading to a specific effect (the failing test).

We proposes Strogoff: a new recommendation system for finding more relevant call sites in the FindCallers feature presented in Section 2.2.3. Our idea is that some method calls are more relevant to the others and should be recommended first. The relevance criterion is the likelihood to propagate errors. Strogoff, learns the relevance and recommends call sites accordingly. Strogoff's recommended call sites are called "sensitive call sites".

In this chapter, a causal graph edge with a high weight close to one is called a *sensitive method call site*, because it is likely to propagate an error. In an IDE, the suggested method callers correspond to the origin of call graph edges pointing a given method. *Thus, we propose that the IDE emphasizes sensitive method callers based on call graph edge weights.* As an example, in Figure 2.10, it will result in the introduction of a filter so that the developer sees first the methods most likely to be impacted by a change in `MutableObjectId.set`.

Our approach is composed of two distinct phases. The *learning phase* consists of learning the weights based on a set M of changes and their actual impacts. In the context of call graphs, a change is modeled as a modified call graph node and an impact is a set of nodes whose behavior is impacted by the change. The *prediction phase* is when a developer is about to change a method (i.e., a node of the call graph), Strogoff computes the set of sensitive method call sites. The prediction represents all methods that may be broken by the change to come.

Figure 4.1 illustrates the causal graph extracted from the call graph presented in Figure 3.1 using the Strogoff's technique. The causal graph obtained is the previous call graph decorated with weights. Assume that a change has occurred at the method denoted by a blue cross. The edges represent method call sites and the thickness of the edges represents the weight of the edge after learning. A thicker edge means a weight close to 1 and a thinner means a weight close to 0.2. The dashed edges are these which have a weight smaller than 0.2 and which are not considered for propagation. The black

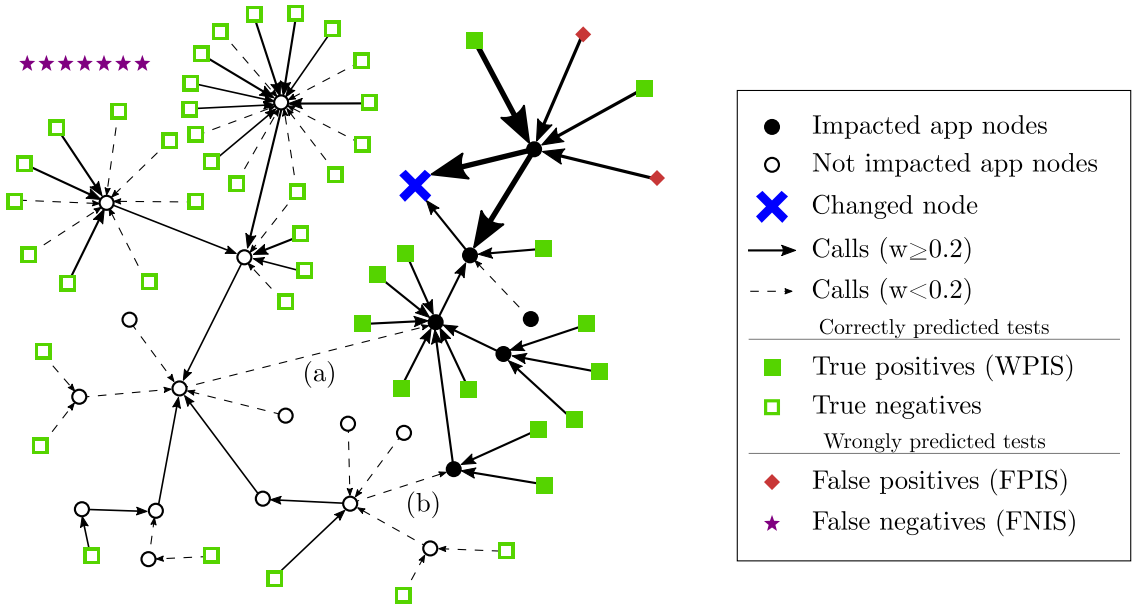


Figure 4.1: An illustration of Strogoff's technique based on weighted call graphs. Edges with a low weight (< 0.2) are considered as non-propagating the impact of the change.

circles, green squares and red diamonds represent nodes predicted as impacted by our approach. The prediction is of high accuracy because the weights of the two edges (a) and (b) is low after learning. Consequently, the impact of the change is stopped and does not ripple to the left-hand side of the graph. On the contrary, a basic impact prediction based on a transitive closure on the call graph would predict far too many tests (all the true negatives would become false positives).

4.2 Learning Phase

Strogoff's learning is made of two components, obtaining learning data and computing the call graph weights.

4.2.1 Input Learning Data

Strogoff takes as input the actual impact sets for a list M of changed methods m using an impact oracle $I(m)$. In this chapter, I is based on mutation testing similarly as it is in Chapter 3: the changed methods m and their impact can here be seen as a set of "learning examples". The core algorithm of Strogoff estimates the edge weights based on these examples. For instance, in the experiments presented in this chapter, we estimate weights based on 16,682 (mutant, impact) pairs.

4.2.2 Computing Weights

The core call graph weight learning algorithm is shown in Algorithm 4.1.

For all nodes m that must be changed in the program (line 3) and each actually impacted node t determined using I (line 4), we update the weight of edges belonging to the shortest path (*cf.* Section 2.1.3) from m to t following an update algorithm (line 5). The rationale of using the shortest paths is twofold. First, it is required so that the approach scales to large software (up to thousands of nodes and edges as shown later in

Algorithm 4.1: The call graph weight learning algorithm using call graph. *update_weight* is a sub-algorithm which updates the weights.

Input: G the call graph, M the list of methods m that must be changed, I the impact oracle.
Output: a weighted graph

```
1 begin
2     L ← G with weights = 0 for all edges
3     for each m ∈ M do
4         for each t ∈ I(m) do
5             update_weight(m, t, L)
6     return L;
```

the evaluation). Second, it reflects the idea that at run-time, shortest paths are more likely to be executed and propagate the error than longer ones. Moreover, it may be theoretically possible to consider all the paths, nevertheless it is impossible in practice.

In this chapter, we propose two algorithms for updating the weights, Binary and Dichotomic that are presented next.

4.2.2.1 Binary Update Algorithm

This algorithm assumes a binary impact propagation: given a mutant and a test, either the impact is always propagated when the test is run, or it is never impacted. Thus, the model consists in assigning 0 or 1 to the edge weights as follows. If at least one impact has been observed between a graph node and the changed node, then all edges belonging to the shortest path going from the former to the latter are labeled as 1; otherwise, these edges are labeled with a 0. The intuition is that the shortest path is more likely to have propagated the impact, or at least, that the actual impact path has followed edges of the shortest path. The good effectiveness presented in our experimental section validates this intuition.

Thus, this approach considers that if an edge has once propagated an impact, it will always do so. Algorithm 4.2 formalizes this idea. This algorithm is deterministic. Thus, running the same algorithm twice on the same data produces exactly the same result.

Algorithm 4.2: Algorithm Binary for updating the edge weights in paths between a node and the changed point.

Input: m the node which has changed, t the considered impacted node and L the weighted call graph
Output: the weights of L are updated

```
1 begin
2     for each edge in the shortest path from m to t in L do
3         w_edge ← 1
```

4.2.2.2 Dichotomic Update Algorithm

We now explore a more realistic model where impact propagation is not straightforward as it may be conditioned by the current execution state. This means that some edges

propagate impacts but only sometimes, in particular cases (e.g., the propagation occurs if coming from the if-branch of a condition and not if coming from the else branch). This is represented by a weight that is neither 0 nor 1 but in between.

The Dichotomic algorithm updates the weights according to an estimation of the probability that a node would be broken by a change. This estimation is based on training data using Equation (4.1)

$$p_{t,m} = \frac{\alpha_t}{\beta_m}; \tag{4.1}$$

where α_t is the number of times the node t is impacted over all changes occurring on the same method m and β_m is the number of times the method has been changed.

The idea of Algorithm Dichotomic (Algorithm 4.3) is to slowly converge to $p_{t,m}$, example after example. For each training example, the weight w of each edge which belongs to a path between a changed node m and an impacted node t is computed in a dichotomic way: the new weight is the mean value between the current weight and the empirical probability.

Algorithm 4.3: Algorithm Dichotomic for updating the edge weights between a node and the changed point.

Input: m the node which has changed, t the considered impacted node, $p_{t,m}$ the empirical probability for (m, t) and L the weighted call graph
Output: the weights of L are updated

```
1 begin
2     for each edge in the shortest path from m to t in L do
3         w_edge ← (w_edge + p_{t,m})/2
```

4.3 Prediction Phase

Algorithm 4.4: The impact prediction algorithm that uses the learned weights of the call graph

Input: L the weighted call graph, n the changed node, th the threshold
Output: the set of nodes which are predicted as impacted

```
1 begin
2     CIS ← {}
3     for each node i connected to n in L do
4         if i is not visited then
5             mark node i as visited
6             if w_i >= th then
7                 CIS ← CIS ∪ {i}
8                 CIS ← CIS ∪ visit(i)
9     return CIS
```

At prediction time, Strogoff is based on Algorithm 4.4. This algorithm takes as input the node n corresponding to a method in the code for which a developer wants to know

the sensitive calls sites. It also takes as input a value th lying in the range $[0, 1]$, that will act as threshold on call graph edges. It returns a candidate impacted set CIS composed of all nodes predicted as impacted (i.e., nodes corresponding to sensitive methods). To do so, starting at the node being changed, the graph edges are followed to determine which nodes can be reached. The weights are used to prune some edges which are unlikely to propagate the change, according to the threshold value th (line 6). If the weight is lower than the threshold value, the edge is considered not to propagate the impact.

4.4 Research Questions

We are especially interested in answering the following research questions.

Research Question 4.1 Does Strogoff's algorithm improve the accuracy of finding sensitive method call sites? In this question, we want to determine whether our prediction algorithm based on learning can improve the prediction scores compared to a standard IDE approach such as the one described in 4.5.1.2.

Research Question 4.2 For the Dichotomic approach, what are the best threshold values to be used on call graph weights?

The Dichotomic Algorithm requires a threshold value to decide whether a weighted call graph edge propagates an impact or not. This research question determines the importance of this value for the Dichotomic and enables us to set the best threshold value for each project.

Research Question 4.3 What is the execution time of Strogoff?

In this question, we focus on the time required to run each part of Strogoff's pipeline. Answering this question helps us to assess whether Strogoff can be used for real in an IDE.

4.5 Experimental Evaluation

We explain how we evaluate our approach, the dataset and the configuration parameters we use in our experimental evaluation.

4.5.1 Evaluation Protocol

Our evaluation framework presented in Chapter 3 is used to evaluate the ability of causal graph to predict sensitive call-sites from a specific changed point.

The evaluation follows several steps:

1. similarly as in Section 3.1, we create mutants for a software application and compute their impacts, they simulate changes;
2. similarly as in Section 3.2, we extract the corresponding call graphs with class hierarchy analysis;
3. we split the dataset of mutants in a training set and a testing set;
4. we run our learning algorithm based on the mutants of the training set. This results in a weighted call graph.

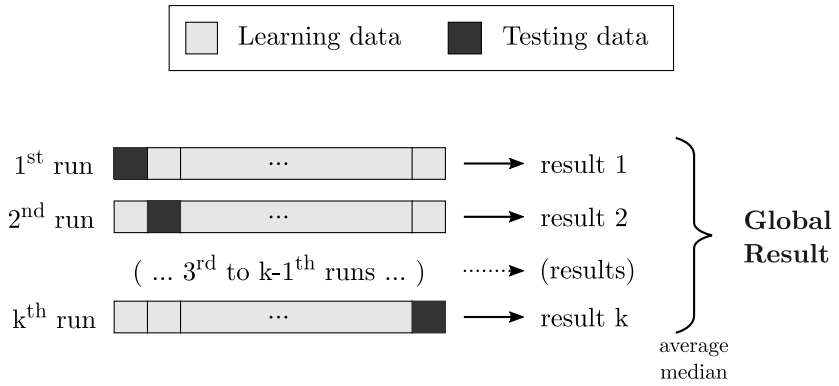


Figure 4.2: Illustration of the k-fold cross validation process.

5. we use the learned weights to compute the sensitive callers in the testing set. That is, the mutated method of the testing set simulate the method for which a developer asks for sensitive callers.

For each mutant of the testing set:

- we predict the impact set for each mutant with our technique, using the weights learned in the previous step;

- similarly as in Section 3.4.1, we compute performance metrics by comparing the predicted impact set and the actual impact set. Since the impact is only computed on test nodes, we remove application nodes from the impacted nodes in the predicted set of sensitive callers.

This measures the effectiveness of our approach.

In addition, we use 10-fold cross-validation [81]: for each program, we partition the mutants into 10 subsets of equal size. We take 9 subsets to train the model (with Algorithm 4.1) and the one remaining is used to assess the model (with Algorithm 4.4). This process is run 10 times. We compute the mean value of the evaluation metrics considered over these 10 runs. Figure 4.2 illustrates the k-fold process.

For Dichotomic, we use a project-dependent threshold value for prediction. Research Question 4.2 focuses on the determination of the best threshold per project. This value is the best one according to a systematic grid search of all thresholds ranging from 0 to 1 with an increment of 0.1.

4.5.1.1 Evaluation and Dataset

The evaluation of the performances of our impact prediction techniques is similar to one presented in Section 3.4.1.1. *Our key goal is to improve the F-score of impact prediction as it takes into consideration both precision and recall.*

Dataset considered in this chapter is made of 9 out of 10 projects from ones presented in Section 3.4.2. We removed Spojo from the dataset as it is too small to efficiently learn anything with it. Thus, we consider 16,682 mutants from the 16,922 considered in Chapter 3. The key descriptive statistics about these projects can be found on Table 3.2 and 3.3. Mutation operators are the same as ones considered in Section 3.4.3.

4.5.1.2 Comparison

We compare Strogoff against the current IDE technique to find callers, the one that we generically call FindCallers in this thesis. However, it is impossible to automatically run the actual code that does this, in isolation of the rest of the IDE, incl. the graphical interface thread. For this reason, we implement FindCallers as follows. FindCallers (referred to IDE in the following tables) returns the callers according to the class hierarchy analysis call graph we use. In our experiment, beyond computing the direct caller, we also consider the transitive closure of all sensitive method callers potentially impacted by the change. This reimplementation has another advantage, it means that both Strogoff and FindCallers are based on the same graph, which removes an uncontrolled variable.

4.5.2 Empirical Results

In this section, we answer our research questions.

Research Question 4.1 Does Strogoff's algorithm improve the accuracy of finding sensitive method call sites?

Table 4.1 gives the value of the evaluation metrics presented in Section 4.5.1.1. The first and second column are respectively the name of the package and the mutation operator. Then we have three multi-columns, one for each metric (precision, recall and F-score). Each multi-column is made of three columns which are the value obtained using a standard IDE approach (IDE), the value obtained with the Binary algorithm and the value obtained with the Dichotomic algorithm. These values are the average over ten-fold cross validation. For each multi-column, the best value is shown in bold face. We consider in this experiment the same number of mutants as the ones in Chapter 3, presented in Table 3.5, column $|\mathcal{K}|$.

First, if we compare the performance of the IDE approach (IDE) and the two learning algorithms, we observe that in almost all cases, both the precision and the F-score are improved using our learning recommendation algorithms. For precision, in all cases excepted Jgit with ABS operator, Dichotomic reports better values. For F-score, Dichotomic reports the best values over IDE except for Jgit with ABS operator and Jodatime AOR which reports same F-scores. For the recall, in 23 cases out of 45 our learning algorithms give better recall scores and in 3 cases out of 45 they are equivalent.

Figure 4.3 shows a scatter plot of the average F-scores obtained for all projects and mutation operators for FindCallers IDE (x-axis) and Dichotomic (y-axis). The line represents $y = x$. Since all points are in the upper-left part, above $y = x$, this figure graphically highlights that Dichotomic much improves the prediction. In addition, we also graphically note that some operators are really far from $y = x$, which means a strong improvement.

For the precision, the best improvement using Binary is for the Shindig project with the ABS operator where the precision raises from 0.57 for IDE to 0.79 (+0.22). The best improvement using Dichotomic is for Io with LCR operator where values raise from 0.39 for IDE to 0.86 (+0.47) for Dichotomic. Overall, Binary and Dichotomic algorithms provide respectively an average precision improvement of 0.07 and 0.21 when averaged over all projects and all mutation operators.

For the recall, the values are quite stable, the best improvement for Binary and Dichotomic algorithms are both with Collections with ABS operator. For Binary, the recall is improved from 0.46 to 0.71 (+0.025). For Dichotomic, the recall is improved from 0.46 to 0.75 (+0.29). Globally, there is no large improvement on the recall compared to the IDE standard approach. there is even a small decrease of −0.11 in recall for Dichotomic.

Table 4.1: Comparative effectiveness of Strogoff and the FindCallers IDE technique for predicting the sensitive call sites (recursively). The bold faced values indicate the best results for a single metric (up to rounding precision). The higher, the better.

		Precision			**Recall**			***F*-score**		
Package	**Op.**	**IDE**	**Bin**	**Dic**	**IDE**	**Bin**	**Dic**	**IDE**	**Bin**	**Dic**
Codec	ABS	0.24	0.34	**0.44**	0.88	**0.89**	0.84	0.30	0.41	**0.43**
	AOR	0.24	0.40	**0.47**	0.90	**0.92**	0.84	0.30	0.48	**0.54**
	ROR	0.28	0.37	**0.40**	0.89	**0.90**	0.84	0.33	0.44	**0.46**
	LCR	0.17	0.27	**0.40**	0.91	**0.92**	0.74	0.24	0.35	**0.43**
	UOI	0.28	0.40	**0.47**	0.89	**0.91**	0.87	0.33	0.47	**0.53**
	All	0.24	0.36	**0.44**	0.89	**0.91**	0.83	0.30	0.43	**0.48**
Coll.	ABS	0.73	0.80	**0.90**	0.46	0.71	**0.75**	0.40	0.70	**0.75**
	AOR	0.76	0.83	**0.92**	0.50	0.70	**0.76**	0.46	0.69	**0.76**
	ROR	0.61	0.67	**0.85**	0.45	0.66	**0.73**	0.34	0.59	**0.70**
	LCR	0.69	0.77	**0.86**	0.60	**0.81**	0.80	0.50	0.72	**0.77**
	UOI	0.74	0.81	**0.89**	0.53	0.75	**0.80**	0.48	0.71	**0.77**
	All	0.70	0.78	**0.88**	0.51	0.72	**0.76**	0.43	0.68	**0.75**
Gson	ABS	0.11	0.19	**0.46**	**1.00**	0.96	0.49	0.16	0.23	**0.26**
	AOR	0.09	0.14	**0.48**	**1.00**	0.99	0.55	0.13	0.21	**0.28**
	ROR	0.14	0.15	**0.31**	**0.98**	**0.98**	0.54	0.18	0.19	**0.23**
	LCR	0.18	0.21	**0.41**	**0.95**	0.94	0.47	0.20	0.23	**0.30**
	UOI	0.17	0.19	**0.33**	**0.98**	0.97	0.70	0.21	0.23	**0.29**
	All	0.14	0.17	**0.40**	**0.98**	0.97	0.55	0.18	0.22	**0.27**
Io	ABS	0.57	0.62	**0.80**	0.81	**0.89**	0.68	0.51	0.65	**0.66**
	AOR	0.48	0.56	**0.93**	0.91	**0.96**	0.82	0.49	0.62	**0.83**
	ROR	0.54	0.57	**0.73**	0.88	**0.91**	0.74	0.55	0.62	**0.67**
	LCR	0.39	0.44	**0.86**	0.90	**0.94**	0.58	0.41	0.52	**0.61**
	UOI	0.50	0.54	**0.88**	0.90	**0.95**	0.67	0.52	0.61	**0.69**
	All	0.50	0.55	**0.84**	0.88	**0.93**	0.70	0.50	0.60	**0.69**
Jgit	ABS	**0.23**	0.21	0.21	**0.87**	0.83	0.80	**0.22**	**0.22**	**0.22**
	AOR	0.23	0.26	**0.34**	**0.90**	0.89	0.83	0.24	0.29	**0.34**
	ROR	0.25	0.30	**0.36**	**0.89**	0.86	0.77	0.25	0.32	**0.34**
	LCR	0.19	0.23	**0.34**	**0.89**	0.85	0.65	0.20	0.25	**0.28**
	UOI	0.22	0.29	**0.37**	**0.91**	**0.91**	0.80	0.24	0.33	**0.34**
	All	0.22	0.26	**0.33**	**0.89**	0.87	0.77	0.23	0.28	**0.30**
Joda.	ABS	0.38	0.44	**0.48**	**0.97**	0.85	0.81	0.42	0.44	**0.46**
	AOR	0.28	0.28	**0.49**	**0.97**	0.95	0.64	**0.33**	0.32	**0.33**
	ROR	0.41	0.42	**0.49**	**0.94**	0.87	0.79	0.41	0.42	**0.43**
	LCR	0.29	0.30	**0.53**	**0.95**	0.92	0.70	0.33	0.34	**0.39**
	UOI	0.31	0.32	**0.41**	**0.97**	0.95	0.85	0.36	0.36	**0.38**
	All	0.33	0.35	**0.48**	**0.96**	0.91	0.76	0.37	0.37	**0.40**
Lang	ABS	0.64	0.73	**0.83**	**0.98**	0.87	0.83	0.70	0.73	**0.78**
	AOR	0.61	0.78	**0.82**	**0.99**	0.95	0.92	0.68	0.81	**0.83**
	ROR	0.64	0.67	**0.73**	**0.98**	0.88	0.84	0.69	0.68	**0.70**
	LCR	0.48	0.54	**0.68**	**0.97**	0.86	0.80	0.56	0.57	**0.66**
	UOI	0.64	0.78	**0.82**	**0.98**	0.94	0.93	0.70	0.81	**0.83**
	All	0.60	0.70	**0.78**	**0.98**	0.90	0.86	0.66	0.72	**0.76**
Shindig	ABS	0.57	0.79	**0.90**	0.93	**0.96**	0.83	0.62	**0.82**	0.81
	AOR	0.61	0.78	**0.91**	0.93	**0.96**	0.91	0.64	0.82	**0.89**
	ROR	0.55	0.60	**0.78**	0.80	**0.88**	0.71	0.50	0.63	**0.64**
	LCR	0.49	0.52	**0.84**	0.77	**0.88**	0.57	0.43	0.57	**0.60**
	UOI	0.63	0.70	**0.93**	0.87	**0.93**	0.74	0.61	0.74	**0.77**
	All	0.57	0.68	**0.87**	0.86	**0.92**	0.75	0.56	0.71	**0.74**
Sonar.	ABS	0.76	0.78	**0.91**	**0.93**	0.89	0.83	0.78	0.79	**0.81**
	AOR	0.57	0.75	**0.84**	0.93	**0.96**	0.95	0.59	0.79	**0.85**
	ROR	0.67	0.68	**0.79**	**0.91**	**0.91**	0.79	0.67	0.72	**0.75**
	LCR	0.64	0.66	**0.85**	0.89	**0.90**	0.80	0.61	0.73	**0.80**
	UOI	0.62	0.77	**0.79**	0.91	**0.94**	0.89	0.61	**0.79**	**0.79**
	All	0.65	0.73	**0.84**	0.91	**0.92**	0.85	0.65	0.76	**0.80**
Total		0.44	0.51	**0.65**	0.87	**0.89**	0.76	0.43	0.53	**0.58**

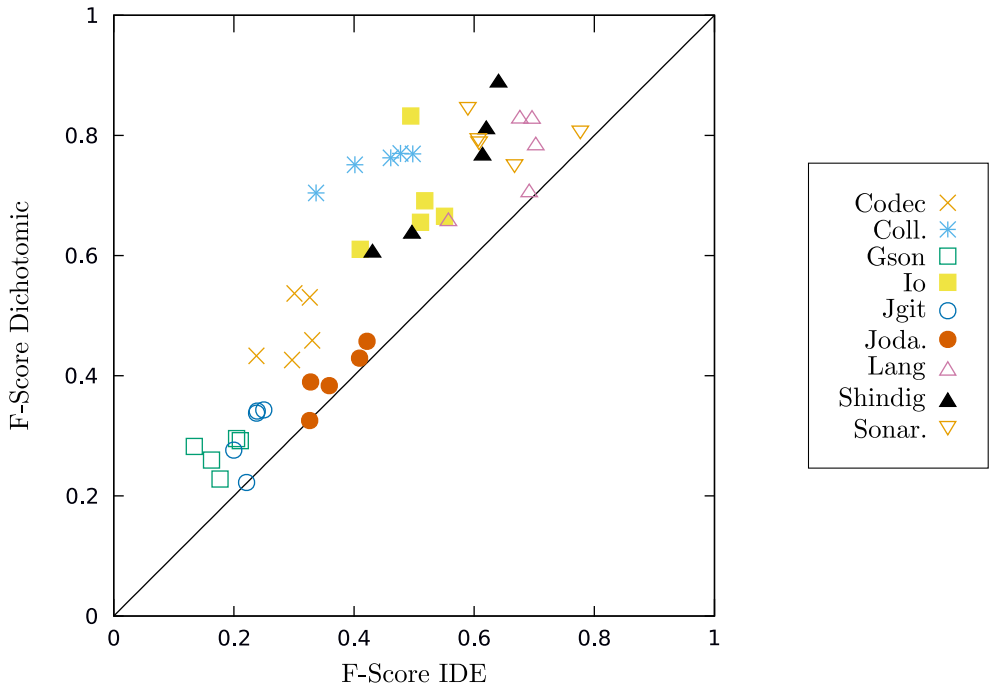


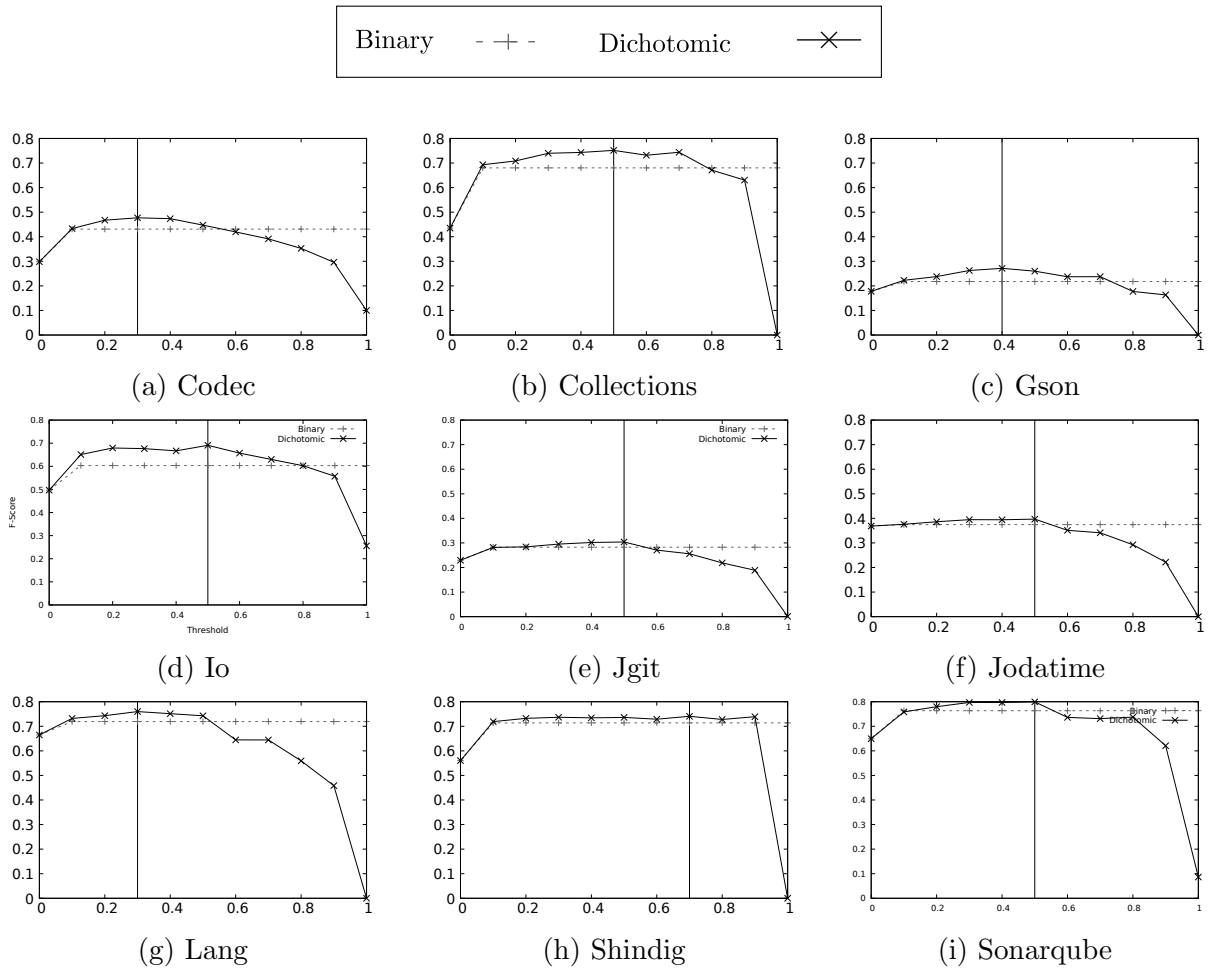
Figure 4.3: The performance improvement of the Dichotomic algorithm impact prediction over the FindCallers IDE technique. One point represents a mutation operator for a given project and is located at coordinates ($x = F_{IDE}, y = F_{Dichotomic}$).

For the F-score, the best improvement is obtained on the Collections project. For Binary the best improvement is for ABS operator on which the F-score raises from 0.40 to 0.70 (+0.30). For Dichotomic, the best improvement is for ROR operator where the values raise from 0.34 to 0.70 (+0.36). Globally, Binary and Dichotomic algorithms lead respectively an average F-score improvement of 0.10 and 0.15.

If we compare the two approaches, we can see that the Dichotomic algorithm has better precision values than the Binary for 44 cases out of 45. But, Dichotomic only has better recall in 4 cases out of 45. F-scores are also better for the Dichotomic algorithm in 42 cases out of 45 than the Binary one (in 2 cases, they have the same score and in 1 case, Binary reports better value). All these observations suggest that Dichotomic algorithm is better than Binary.

To ensure that the performance difference between Dichotomic and Binary is significantly different, we run a Mann-Whitney-Wilcoxon statistical test on precisions, recalls and F-scores. The null hypothesis (H_0) is: "the metrics (either precision, or recall, or F-score) obtained for the Binary and the Dichotomic algorithms are drawn from the same distribution". The obtained p-values for precision, recall and F-score is of 2.2×10^{-16}. For each metric, the conclusion is then that the null hypothesis is rejected, which means that Dichotomic is better in a statistically valid sense.

Moreover, we compute the Cohen's d effect size [39, 75] to determine the magnitude between the measurements. It is computed by the difference between two sample means over the pooled standard deviation [39]. The Cohen's d effect size is of 0.37 for the precision, 0.52 for the recall and 0.13 for the F-score. The reported Cohen's d effect size for the F-score is low which implies that both techniques are almost equivalent from a practical point of view.



(a) Codec (b) Collections (c) Gson

(d) Io (e) Jgit (f) Jodatime

(g) Lang (h) Shindig (i) Sonarqube

Figure 4.4: The impact of the prediction threshold (x-axis) on the F-score (y-axis). Low (< 0.1) and high (> 0.5) thresholds yield bad performances, the rest forms a plateau – Binary is a dashed line and Dichotomic is a plain one.

To sum up, using weighted call graphs is effective for identifying sensitive call sites. It is more effective than the default IDE approach that makes no difference between call sites. Among the two weight-learning algorithms we propose in this chapter, the best one is the Dichotomic approach, however only to a small extent.

Research Question 4.2 For the Dichotomic approach, what are the best threshold values to be used on call graph weights?

We now focus on Dichotomic because it is the best algorithm for learning the weights, as seen in the previous RQ. We want to study the impact of the threshold used for sensitivity. Figure 4.4 shows the F-scores obtained for each project using different threshold values. Recall that a threshold t is used at prediction time: if the weight of an edge is higher than t, then we consider that the impact propagates and the caller is sensitive.

In general, we note an inverse U shape for most curves. This means that a very low threshold yields a bad performance, as well as a high threshold. This can be explained as follows. A low threshold means that most edges are considered as propagating the impact. In this case, the precision drops too much (indeed, the F-score for a threshold of zero is exactly the one of the IDE case). On the other hand, a high threshold means that very

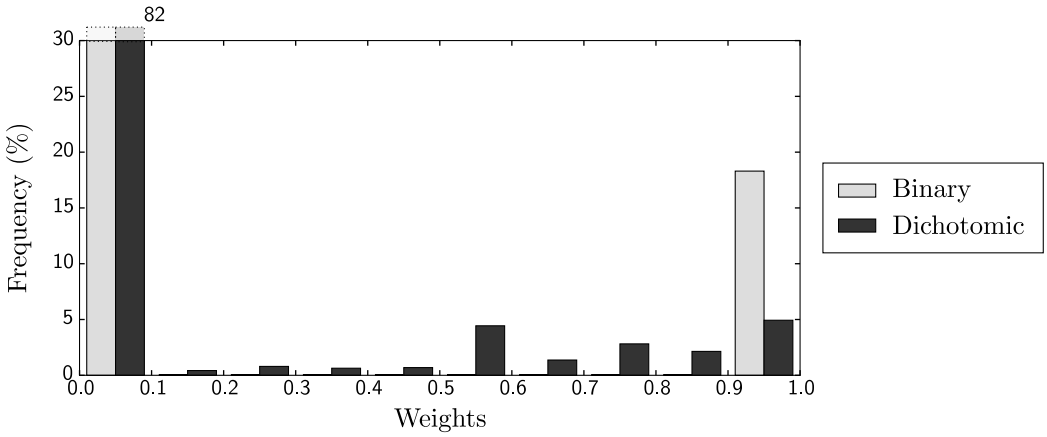


Figure 4.5: Histogram of the weights learned using the Binary (light gray) and the Dichotomic (dark gray) algorithms aggregated over all projects and all mutation operators. Each bin represents the amount of weights whose value lies in the range $[x, x+0.1[$, where x is the value indicated under the bin. Dichotomic and Binary have the same number of disabled edges (weight equals to zero), but Dichotomic is much finer for the other sensitive edges.

few edges are considered as propagating. Since the Dichotomic convergence slowly raises from 0, the bad performance of high threshold values is due to the fact the threshold is too high with respect to the convergence to the empirical probability used in Dichotomic.

For all projects except Shindig, the best threshold is lower or equal to 0.5. For 5 projects out of 9 (Collections, Io, Jgit, Jodatime and Sonarqube) the best threshold is around 0.5. For Codec and Lang, the best threshold is around 0.3 and for Gson, the best one is 0.4. Regarding Shindig, the best threshold is around 0.7.

Figure 4.5 shows an histogram of the weights obtained with both algorithms over the 9 software projects. As expected, for Binary, we observe that all values are either 0 or 1. For Dichotomic, we observe that most edges have a weight between 0 and 0.1. Then, the distribution is rather flat with learned weights spread over the remaining values. This figure confirms that learning indeed happens. Edges that have never been visited are still at zero. Edges that are involved only in few propagation have a small weight. Consequently, a threshold around 0.2 indeed drops all edges with a small weight, these that are unlikely to propagate an impact.

To predict that an edge propagates an impact or not, very low and very high thresholds are both bad according to our experiment. A good threshold value lies between 0.3 and 0.5 depending on the project. This is the range that would be used in development environments.

Research Question 4.3 What is the execution time of Strogoff?

We now study the execution performance of Strogoff. Table 4.2 reports all execution times involved in Strogoff. The first column is the package name. The second column is the time required to generate the code of one mutant. The third column is the time required to run the full test suite of the program once. This time is the time required for running the tests without mutation. Indeed, in some scenarios this time can be larger as some tests hang (e.g., due to an infinite loop). The fourth column is the time required to

Table 4.2: Execution time of Strogoff. The values are averaged over all mutation operators.

Project	Mutate	Test	Graph	Learn	Predict
Codec	48.0ms	32.2s	2.7s	0.291s	0.16ms
Collections	175.8ms	38.9s	7.8s	0.368s	0.04ms
Gson	136.1ms	13.7s	3.7s	4.414s	1.41ms
Io	39.1ms	90.1s	3.4s	0.108s	0.05ms
Jgit	154.3ms	195.5s	10.8s	114.026s	4.11ms
Jodatime	135.1ms	31.1s	8.7s	73.669s	10.89ms
Lang	136.1ms	40.0s	7.2s	0.478s	0.06ms
Shindig	267.6ms	14.1s	2.6s	0.094s	0.03ms
Sonarqube	167.7ms	387.3s	4.4s	0.181s	0.02ms

generate the call graph from source code. The fifth column is the time required to learn over mutants and the last column is the time required to perform one prediction using the previously learned information. The learning and prediction times are the mean value over all executions (over all mutation operators).

As an example, if we consider the Apache Commons Codec package, it requires 48ms to generate one mutant and 32.2 seconds to compute its impacts (a total of 32.248s). This means that for generating 100 mutants, one needs around one hour (32.248×100). For the same project, we obtain a call graph is 2.7 seconds and we can learn over with the mutants in 291ms. Finally, an impact can estimated in less than a millisecond ($0.16ms$).

As we can observe, the slowest part of the work is to generate the mutants and running the tests on them. If generating one mutant always takes less than one second on average, running the tests on the mutants takes a minimum of 13.7 seconds for Gson and up to 6 minutes for Sonarqube. As one requires a large number of mutants to correctly learn impact information, this process is quite heavy and it is the dominating time factor of Strogoff. However, this process does not take place in the IDE: it can be performed during the night on continuous integration or regression servers.

The time required for generating the graph takes from 2.6 seconds for Shindig to 10.8 seconds for Jgit. This is short enough so that the call graph can be regenerated for each code change. Regarding the learning time, we observe that it generally takes less than one second except for Jodatime and Jgit which take less than 2 minutes. This is acceptable.

The prediction time is the most important one as it is the time required for a developer to obtain its suspicious methods in the IDE directly. We observe that this time is generally small: less than one millisecond except for Jodatime, JGit and Gson which require respectively 10.89ms, 4.11ms and 1.41ms. In all cases the required time is less than one second which is acceptable in a user interface. This shows that developers can indeed obtain more information about sensitive call sites while coding.

To sum up, Strogoff can be used in an IDE. However, mutation testing must be done offline. The rest of the approach – call graph extraction, weights learning and sensitive caller prediction – is fast enough to be performed online, in the IDE.

4.6 Threats to Validity

In this section, we present now the major threats to validity of the works presented in this chapter.

4.6.1 Internal Validity

Our results are of computational nature. A major bug in our software can invalidate our findings. We have published all our code on Github so as to facilitate reproduction and falsification of our results, if necessary.

4.6.2 Construct Validity

In our evaluation, we use synthetic changes for exploring the performance of our technique. Our motivation for using synthetic changes is to have a large amount of data, which is necessary for learning. Another option would be to use actual usages of the FindCallers features in IDEs. However, these are difficult to obtain, unless one ships instrumented IDEs to real developers.

In this chapter, we use 600 mutants per mutation operator and per project. After removing equivalent mutants (i.e., mutants which do not break any test) we have an average number of 371 mutants per project and per mutation operator. Using ten-fold cross validation, this results in testing sets of 37 items on average. Since the aggregated performance measures (precision, recall and F-score) are rather stable over folds, we have confidence that this is enough to back up our conclusions. However future work with more mutants is required.

4.6.3 External Validity

The impact prediction depends very much on the structure of the call graphs [113]. For instance, the presence of large utility methods, with many incoming and outgoing edges has a direct impact on the prediction performance. Our results may only be valid for Java software, or even worse only valid for the projects under study. Future work in this field will strengthen the external validity of our findings.

4.7 Conclusion

In this chapter, we have presented Strogoff, an approach used to reason about the likelihood of impact propagation based on a weighted call graph also named the causal graph. This weighted call graph is used to predict sensitive call sites in a similar fashion as standard Integrated Development Environments (IDE) used by a large number of developers today. The weights are learned based on mutation testing data. Our experiments based on 16,682 killed mutants in 9 subject program totaling 46,244 call graph nodes, show that our technique is more effective than using a call graph alone, as done in today's development environments.

Our experiments showed really good performances. The prediction, once the learning phase is done, can be achieved in less than 15ms. However, the learning phase, based on mutation testing, required more time to complete. In a concrete usage, the learning phase may be done offline from time to time, offering the opportunity to developers working on a same program to run as many predictions as they want with the learned data. The

average improve of the F-Score is of 0.15, going from 0.43 with a simple call graph to 0.58 with the causal graph. Considered the simplicity of the learning logic, these results are encouraging to further investigate the field, using more complex learning approaches and proposing other types of recommendation systems.

5

Causal Graph for Fault Localization

"The best way to predict the future is to invent it."

— Alan Kay

This chapter at a glance...

In this chapter, the following contributions are presented:

- a new fault localization algorithm, called *Vautrin*, that uses a graph-based approximation of causality for fault localization;
- an empirical evaluation of Vautrin on 5,386 faults from the Steimann's dataset, showing that Vautrin outperforms the state of the art, both in terms of wasted effort and perfect fault localization prediction.

The following publication is related to this chapter:

[1] Vincenzo Musco, Martin Monperrus, and Philippe Preux. Mutation-based graph inference for fault localization. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation*, October 2016.

In Chapter 3 and 4 we used call graphs and mutation testing to perform change impact analysis on Java programs. Moreover, in Chapter 4, we presented the causal graph, a special type of call graph intended to better approximate the cause-effect between software methods and tests. In this chapter, we use information learned in *causal graphs* to perform fault localization.

Fault localization is the process of identifying the elements of software that are faulty, i.e., elements that are responsible of a bug. A classic way of stating the fault localization problem is that the bug is reproduced and asserted by a failing test case, and the goal is to predict the function that contains the buggy code. In essence, the fault localization process tries to capture causality relationships between code elements [15, 143].

Indeed, early works in fault localization were based on program slices [4], which are refined versions of the most obvious causal relationship: the bug must lie somewhere in the code that has been executed. Spectrum-based fault localization is also causal to a certain extent, but with a really strong approximation: the causal relations are only captured by the fact that an element is covered by passing or failing test cases. That is, fault localization is only an approximation of the true cause-effect chain of error propagation that happens at run-time. To our knowledge, only Baah et al. [15] and Shu et al. [143] have set notable milestones using causal inference for better approximating causal effects in fault localization.

We propose Vautrin, a fault localization approach which approximates causality by analyzing what happens between a failing test and a method using causal graphs as defined in Section 4.1. Similarly as Strogoff, Vautrin is based on the assumption that mutants and their execution profiles do contain valuable information that can be used to approximate the causality. Based on information contained in the causal graph and a set of failing tests, Vautrin determines potential graph-suspicious elements. As Vautrin does not allow to order elements but just filter out, when multiple methods are graph-suspicious, they are ranked using standard spectrum-based suspiciousness scores. In this chapter, we considered the five following: Tarantula, Ochiai, Zoltar, Naish and Steimann.

To evaluate our algorithm, we consider the fault localization benchmark from Steimann et al. [145] published at ISSTA'13 composed of ten Java programs and 6 different mutation operators. The performance assessment is made using the "wasted effort" metric, as well as the less-used one: the "perfect prediction" which reports the number of cases the fault localization algorithm reports directly the real fault.

We show that our method-level fault localization algorithm outperforms the most recent algorithms, including Ochiai [1] and Naish [116]. The improvement ranges from a minimum of 3% to 55% less methods to consider after fault localization. Using our new technique, over the whole Steimann's dataset, the number of perfect predictions is 2,310 out of 5,836 which means a percentage of 40%, representing an improvement of 14% over the related works.

The chapter is structured as follows. In Section 5.1, we present our technical contribution by presenting our general intuition before introducing Vautrin, our fault localization algorithm. In Section 5.2, we introduce the research questions under investigation. In Section 5.3, we present our empirical protocol, report our evaluation results and answer to our research questions. In Section 5.4, we discuss the threats to validity of our work. Finally, we conclude in Section 5.5.

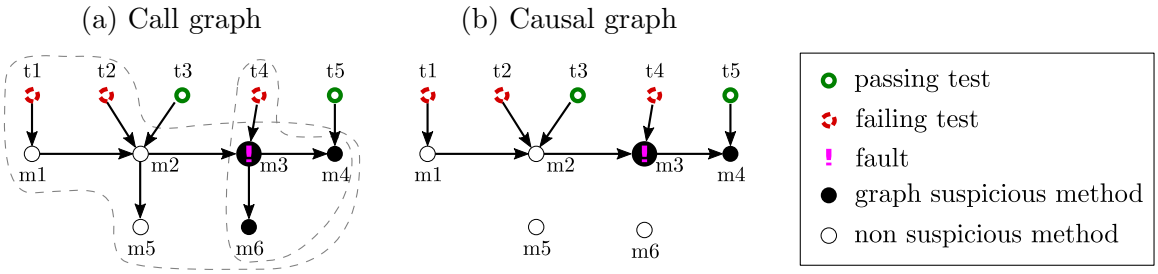


Figure 5.1: Example of (a) a call graph, and (b) the causal graph extracted from (a) using our technique. The causal graph only contains edges for which causal evidence exists.

5.1 Technical Contribution

In this section, we present how we can use graphs for reasoning in terms of fault localization (Section 5.1.1). Then, in Section 5.1.2, we present the algorithm used to practically perform causal graph-based fault localization.

5.1.1 Intuition

Fault localization is a field of software engineering which aims at automatically identifying a faulty element in a source code. A faulty element can be of any granularity (e.g., statement, method, class). Most fault localization techniques are *spectrum-based* techniques where "spectrum" refers to the behavioral traces of the program when executing the tests [60].

We introduce Vautrin, a novel fault localization approach working at the method level and using another source of information: the causal graph presented in Section 4.1. Vautrin is built on the assumption that if two tests fail, the bug might lie at the intersection of the nodes reachable from both tests.

The figure 5.1 illustrates a call graph for a simple program composed of 6 methods and 5 tests. Passing tests are nodes with a solid green border, and failing tests are nodes with a dashed red border. The faulty method is marked with an exclamation point. We compute the transitive closure (as presented in Section 3.2) of each failing test `t1`, `t2` and `t4`, that is the set of nodes reachable from each failing test. This transitive closure is represented with dashed lines on the figure. The intersection of the transitive closures gives a set of three nodes `m3`, `m4`, and `m6` (marked as plain black nodes), which contains the faulty method. In this chapter, the methods in the intersection are called the *graph-suspicious* methods (as opposed to spectrum-suspicious ones which are nodes determined using a spectrum-based fault localization algorithm). Conversely, other application nodes are marked with white nodes and are not suspicious.

In our example, method `m2` calls method `m5`. Let us imagine that method `m5` is a logging method. In such a case, a logging function is unlikely to propagate a fault as it only prints strings to a file. Thus, there is a difference between the causality flow and the method call flow. Some method calls may not alter the state of the system and thus should not be considered as causal. Moreover, a test may not call a method at run-time. This occurs when the method invocation is optional, e.g., when located in a conditional block. In such cases, the causality of these edges should also be reconsidered. Indeed, the call graph contains all possible connections that are found by statically analyzing the

source code[1]. Our idea is to only keep the call graph edges for which there exist pieces of evidence that they are causal.

For this purpose, we use the *causal graph* and the learning approach presented in Chapter 4. However, in this chapter, we consider a narrower vision of the *causal graph* as we only consider the binary learning algorithm presented in Section 4.2.2.1. As a consequence, our causal graph only have edges with weights of 0 or 1. To ease the reading of this chapter, we consider that if the weight is equal to 0, the edge is removed from the causal graph.

Figure 5.1b illustrates such a causal graph extracted from the call graph presented in Figure 5.1a. This causal graph contains the edges of the call graph except the ones between `m2` and `m5` and between `m3` and `m6` for which causal evidence does not exist.

To explain the extraction process, let us assume that a first mutant in method `m3` results in 4 failing test cases: `t1`, `t2`, `t3` and `t4`. The algorithm will result in 4 paths: one from `t1` to `m3`, a second from `t2` to `m3`, a third from `t3` to `m3` and a last one from `t4` to `m3`. After processing this mutant, the causal graph contains 6 causal edges. Let us now imagine that we generate a second mutant occurring on `m4` resulting in two failing tests: `t4` and `t5`. The graph analysis of this second mutant results in two paths: the first from `t4` to `m4` and the second from `t5` to `m4`, this results in adding two more causal edges. With these two mutations, we obtain the causal graph shown in Figure 5.1b.

After the extraction, we observe that the set of faulty nodes based on graph analysis, contains one less node and still contains the faulty one.

Then, when a fault has to be localized, Vautrin computes the intersection of the transitive closure of each failing test according to the causal graph. The nodes belonging to this intersection form the set of "graph-suspicious nodes". If the intersection is made of several nodes, Vautrin uses an external spectrum-based fault localization algorithm to rank them.

5.1.2 Algorithm

Once a causal graph has been obtained, a prediction algorithm is used at a production stage, i.e., a developer wants to debug a new fault. This algorithm is executed at run-time within a debugging session once a developer needs fault localization assistance. During this phase, Vautrin takes as input a set of failing tests and produces an ordered list of suspicious elements. The first element of this list is the most suspicious method, and the last element is the less suspicious method.

The prediction algorithm takes as input a set T_f of failing tests and returns the list of methods ranked according to their suspiciousness. It takes as input the causal graph, first computes graph-suspiciousness, then computes spectrum suspiciousness and combines both into a localization diagnosis. This is given in Algorithm 5.1.

On line 2, all nodes in L are put into the set I, which means that by default all nodes are considered as graph-suspicious. Then, each failing test of the fault under debug is explored (line 3). For each failing test, we compute the nodes belonging to the transitive closure from the failing test (line 4). The resulting set of nodes is intersected with I (line 5). In this manner, by progressively exploring all failing tests, nodes are removed from the set I, because they are not considered as graph-suspicious.

Finally, a standard spectrum-based fault localization algorithm is used to determine the score of each element in I (line 6). This fault localization algorithm can be any

[1]Some calls may be missed which are not statically observable, such as these resulting from the use of reflection.

Algorithm 5.1: Vautrin's prediction algorithm using a causal graph L and failing tests T_f to estimate the suspiciousness of a method.

```
   Input: L a causal graph. T_f the list of failing tests.
   Output: I: ranked suspicious nodes
1  begin
2      I ← all nodes of L
       /* Compute graph suspiciousness                    */
3      for each t ∈ T_f do
4          F ← transitive closure from t in L
5          I ← I ∩ {F}
6      S ← compute spectrum suspiciousness
7      sort elements in I according to S
8      return I
```

spectrum-based fault localization algorithm. According to our investigations, the best fault localization algorithm to use during prediction phase is the Steimann's one [145] (*cf.* Research Question 5.6); thus, this is the one that is used by default in Vautrin and that we consider in the evaluation. On line 7, elements are ordered in descending order according to the spectrum suspiciousness. The first element is the most suspicious. For instance, considering again Figure 5.1(b), we can assume that a good spectrum-based suspiciousness score ranks m_3 as more suspicious than m_4.

Note that I may be empty if the causal graph is disconnected or incorrect (that is, the approximation of the causality of some edges is wrong). In such a scenario, Vautrin is not able to predict anything based on graph suspiciousness. Thus, it uses a *fallback mode* in which the scores are computed without using the causal graph, only using spectrum suspiciousness. Consequently, Vautrin necessarily gives results that are at least as good as the one returned by the underlying spectrum-based fault localization algorithm. In other words, Vautrin never degrades the spectrum-based diagnostic if it is already good.

5.2 Research Questions

In this section, we present the research questions we raise and address in this paper.

Research Question 5.1 Does Vautrin localize faults with less wasted effort than the state of the art? In this question, we focus on the improvement we achieve using Vautrin with respect to a classic evaluation metric in fault localization: the wasted effort. Since Vautrin works at the method level, the wasted effort is measured as the number of methods being inspected before the faulty one is reached.

Research Question 5.2 Does Vautrin give better perfect predictions than the state of the art? Beyond the wasted effort, a fault localization technique is said to make a perfect prediction when the faulty element is ranked at the top of the list. In such cases, the developer wastes no time. Up to our knowledge, this evaluation metric has not been extensively studied so far.

Research Question 5.3 What is the execution time cost of fault localization with Vautrin? In this question, we want to analyze the computational cost of preparing the graph for

fault localization, and thus assess whether the approach is usable in practice by a developer.

Research Question 5.4 To what extent does Vautrin fall back to the traditional, graphless, fault localization? As explained in Section 5.1.2, there are cases in which the obtained causal graph is disconnected. When this happens, it falls back into a mode where the Steimann's spectrum-based suspicious score is used alone. With this question, we study whether our fault localization algorithm is able to improve many cases or whether it only works for a small proportion of fault localization tasks.

Research Question 5.5 Do non-causal edges have an impact on fault localization effectiveness? Vautrin builds the causal graph by only considering causal edges of the call graph, this is the essence of the intuition presented in Section 5.1. We want to assess to which extent the causal graph improves the fault localization effectiveness.

Research Question 5.6 What is the best spectrum metric to be used with Vautrin? Vautrin computes graph-suspicious nodes according to the causal graph. When several nodes are graph-suspicious, Vautrin requires a spectrum-based metric to extract the most suspicious ones. In this Research Question, we determine which is the best spectrum-based metric to use with Vautrin.

5.3 Experimental Evaluation

We now present the evaluation of Vautrin based on the presented research questions. This includes the considered metrics and dataset. Then, we present our empirical results.

5.3.1 Evaluation Protocol

To evaluate Vautrin, we measure its ability to localize the source of a fault. To that end, we use a dataset that has been proposed by Steimann et al. at ISSTA'13 [145] and the empirical results published therein. This dataset is based on a set of 10 subject programs. For each program, a set of mutants has been generated. For each mutant, a set of tests is executed; some pass, others fail leading to a "fault". Then, the question is: which method is at the origins of the fault? As we use mutation testing to assess our model, the performance measured of our fault localization method depends on the set of mutants being derived from software and from the set of tests being executed. Measuring this performance on a single set of mutants and test nodes is not meaningful as it is not reflecting the performance of the fault localization method in general, that is for any set of mutants and for any set of tests. We address this issue by performing a cross-validation such as done in Section 4.5.1

5.3.1.1 Evaluation Metrics

In this section, we present the metrics we use to evaluate our approach. We use the *wasted effort* to determine the accuracy of a fault localization algorithm. For each fault, the fault localization algorithm assigns a suspiciousness score to each method. Let us denote m^* the method at the origins of the fault. Then, for this fault, the wasted effort is simply the number of methods that will be investigated before m^*; phrased in other terms, the wasted effort is the number of methods for which the score is higher that the score of m^*.

Equation (5.1) defines the wasted effort in a more formal way. M is the number of methods being considered and $S(m^*)$ is the score for method m^*.

$$W(m^*) = \sum_{i=1}^{M} 1[S(i) > S(m^*)] + \frac{\sum_{i=1}^{M} 1[S(i) = S(m^*)]}{2} \tag{5.1}$$

The wasted effort estimates the effort for a developer to find the origin of a fault if he/she considers all suspicious methods ordered by their suspiciousness score. However, we think this metric, even if valuable, is not representative of the ability of a fault localization algorithm to assist fault localization. When a developer uses a tool to assist him/her to localize a fault, he/she wants to save time: if he/she gets a list of suspicious methods, he/she will probably have a look at the first one, but if this method is not the faulty one, he/she will hardly have a look to the following method(s).

This is the reason why we use a second evaluation metric we call the *perfect prediction*. This metric is the number of faults for which the wasted effort is zero. In other words, this metric is the number of faults for which the fault localization algorithm proposes the faulty method (m^*) at the top of the list of suspicious elements, at position #1. It corresponds to the evaluation metric "is in top-k" (used for instance in [162]) with $k = 1$. Formally, the perfect prediction P is given in Equation (5.2) with N being the number of faults being considered. Another evaluation metric is MAP (mean average precision), but it does not reflect as well as *perfect prediction* the fact that the developer builds trust based on the top result.

$$P = \sum_{f=1}^{N} 1[W_f = 0] \tag{5.2}$$

5.3.1.2 Comparison

We conduct a comparative evaluation. We consider 5 fault localization algorithms of the literature: Tarantula[73], Ochiai[1], Zoltar[54], Naish[116] and Steimann[145]. Tarantula and Ochiai are largely used in the fault localization literature, e.g., [1, 116, 139]. Thus, for historical reasons, we consider these fault localization algorithms, even if they are not among these performing the best. Zoltar, Naish and Steimann are currently the most accurate fault localization algorithm. Shu et al. [143] also do method-level fault localization. We considered them for a quantitative comparison, however, their heavyweight implementation is not available (we have asked for it).

In the remaining of this section, we present these fault localization algorithms. When running the software tests T, we obtain: T_p the set of passing tests and T_f the set of failing tests. If we consider an execution from the point of view of a specific code element e, only a subset of T does actually call the code element e. Let E be the set of tests which actually call e. Then, E_p is a subset of T_p made only of passing tests actually calling e. E_f is a subset of T_f made only of failing tests actually calling e. N is the set of tests which do not call e, that is $N = T - E$, and accordingly $N_p = T_p - E_p$ and $N_f = T_f - E_f$.

According to these sets, a *fault localization algorithm* is used to assign a *suspiciousness score* to each code element e. The higher the score, the higher the chance the code element is faulty. As this suspiciousness is determined using the spectrum-based fault localization, we refer to it in this thesis as the *spectrum suspiciousness*. In this chapter, we consider the following suspiciousness metrics. Tarantula was proposed by Jones et al. in 2002 [73]. To rank and determine the origins of a fault, Tarantula takes into consideration the

proportion of executed passing and failing tests.

$$T = \frac{\frac{|E_f|}{|T_f|}}{\frac{|E_f|}{|T_f|} + \frac{|E_p|}{|T_p|}} \tag{5.3}$$

Ochiai, a biology metric also used as fault localization algorithm by Abreu et al. [1], focuses only on failing tests (T_f) and executed tests (E_p and E_f). Equation (5.4) defines this fault localization algorithm.

$$O = \frac{|E_f|}{\sqrt{(|E_p| + |E_f|) \times |T_f|}} \tag{5.4}$$

Zoltar was proposed by Gonzalez in 2007 [54]. Equation (5.5) defines the Zoltar fault localization algorithm. This fault localization algorithm considers the executed tests (E_p and E_f) and non-executed tests (N_p and N_f).

$$Z = \frac{|E_f|}{|E_f| + |E_p| + |N_f| + \frac{10000 \times |E_p| + |N_f|}{|E_f|}} \tag{5.5}$$

Naish et al. proposed a fault localization algorithm that is perfect under certain assumptions [116]. This fault localization algorithm is based on the non-executed tests (N_p and N_f): if at least one non-executed test fails (N_f), then the code element is not suspicious. Otherwise, its suspiciousness score is simply the number of non-executed passing tests (N_p). Equation (5.6) defines this fault localization algorithm.

$$N = \begin{cases} -1, & \text{if } |N_f| > 0 \\ |N_p|, & \text{otherwise} \end{cases} \tag{5.6}$$

Steimann et al.'s metric $T*$ was proposed in 2013 [145]. This fault localization algorithm is strongly based on the Tarantula one as defined in Equation (5.7) (the T in the equation refers to the Tarantula score). To avoid confusion, we refer to the $T*$ fault localization algorithm of Steimann as S.

$$S = T * max(\frac{|E_p|}{|T_p|}, \frac{|E_f|}{|T_f|}) \tag{5.7}$$

5.3.1.3 Dataset

We consider the dataset proposed by Steimann et al. in ISSTA '13 [145]. Their dataset is made of 10 subject programs (they call them "probands") totaling 5386 one-point mutants. The dataset is composed of execution information for the non-mutated subject program (i.e., the reference) and a collection of mutated version of the subject program. It provides following information for each mutated version of the program:

- the list of executed tests;
- their execution result (passing or failing);
- the list of methods contained in the program;
- the mutation operator if applicable;
- a list of methods executed by each test (the method-level spectrum).

Table 5.1: Descriptive statistics of the fault dataset used in this experiment. # of nodes and edges respectively correspond to the number of methods and the number of method calls.

						Graph	
Program	**Version**	**LOC**	**#Classes**	**#Tests**	**#Mutants**	**#Nodes**	**#Edges**
AC Codec	1.3	2,446	25	188	543	488	825
Daikon	4.6.4	147,153	1,109	157	352	48,689	63,250
Draw2d	3.4.2	22,895	317	89	570	12,298	16,091
Eventbus	1.4	3,572	53	91	577	2,377	2,930
Htmlparser	1.6	21,764	161	600	599	7,905	11,895
Jaxen	1.1.5	12,466	205	695	600	3,171	6,513
Jester	1.37b	1,621	46	64	411	467	645
Jexel	1.0.0b13	1,349	46	335	537	984	1,753
Jparsec	2.0	4,950	122	510	598	5,263	6,723
AC Lang	3.0	18,400	135	1,666	599	6,730	8,906
Total		**236,616**	**2,219**	**4,395**	**5,386**	**88,372**	**119,531**

We produce the subject program call graphs: the dataset totals 88,372 nodes and 119,531 edges. Table 5.1 shows the 10 subject programs under study. The first column is the subject program name, the second is the considered version, the third is the number of lines of code[2], the fourth is the number of classes for the subject program, the #Tests column contains the number of tests in the program, and the #Mutants gives the number of available one-change mutants (i.e., a mutant where only one point in the code has been changed). The last two columns are call graph information: the number of nodes and the number of edges which correspond respectively to the number of methods and the number of method calls. All the programs are daily used ones and consist in a total of 230,000+ lines of code and 4,000+ test cases. They can be considered as realistic.

5.3.1.4 Mutation Operators

The following mutation operators are considered in the Steimann's dataset:

- *Negate Decision (ND):* `if` or `while` statement conditions are negated;
- *Replace Constant (RC):* integer constants C are replaced by 0, 1, -1, $C+1$ or $C-1$;
- *Delete Statement (DS):* deletion of a statement;
- *Replace Operator (RO):* replacement of an arithmetic, relational, logical, bitwise logical, increment/decrement or arithmetic- assignment operator by an operator from the same class;
- *Assign Null (AN):* replacement of the right hand side of assignment by `null`;
- *Return Null (RN):* replacement of `return` expressions by `null`.

These mutation operators are ones presented in [7, 9, 117, 144].

[2]computed using CLOC (`http://cloc.sourceforge.net/`)

Table 5.2: Average wasted effort, in number of inspected methods, for different fault localization algorithms over the benchmark of [145]. The lower, the better. The best scores are bold-faced.

Software	**T** 2002	**O** 2006	**Z** 2007	**N** 2011	**S** 2013	**V** 2016	**Impr.**
AC Codec	6.01	3.08	2.85	2.86	2.66	**1.81**	47%
Daikon	142.07	125.22	124.78	149.30	124.65	**121.07**	3%
Draw2d	35.41	25.26	23.98	34.12	24.01	**15.46**	55%
Eventbus	17.56	6.16	9.69	50.51	5.99	**4.42**	36%
Htmlparser	21.59	6.11	5.13	21.20	4.82	**3.30**	46%
Jaxen	49.62	18.29	9.27	12.00	11.30	**7.59**	49%
Jester	4.60	2.76	2.55	2.34	2.38	**1.57**	52%
Jexel	15.96	9.06	7.06	6.69	6.64	**5.65**	18%
Jparsec	15.62	3.95	**3.00**	21.90	4.39	3.53	24%
AC Lang	4.87	2.76	2.69	18.10	2.68	**2.40**	12%
Average	**31.33**	**20.27**	**19.10**	**31.90**	**18.95**	**16.68**	**14%**

5.3.2 Empirical Results

Research Question 5.1 Does Vautrin localize faults with less wasted effort than the state of the art?

As presented in Section 5.3.1.1, the wasted effort is the number of wrongly predicted methods returned by the fault localization algorithm before finding the good one. The lower this value, the better, because it directly relates to the time spent by a developer to analyze inaccurate results. For instance, if the wasted effort is 5, this means that a fault localization algorithm has reported 5 non-faulty methods before reporting the actually faulty one.

Table 5.2 shows the average wasted effort for each subject program of our dataset[3]. The first column is the program name and the second to the sixth column give the average wasted effort for respectively Tarantula, Ochiai, Zoltar, Naish and Steimann (the unit is an absolute number of methods to inspect, because in this chapter, we perform fault localization at the method level). The two last columns are the average wasted effort with our fault localization algorithm, Vautrin and the relative improvement obtained using Vautrin compared to Steimann.

As an example, if we consider the Jaxen subject program, the faulty method is ranked on average at the 11^{th} position using Steimann's fault localization algorithm. For the same subject program, it is ranked at the 7^{th} position with Vautrin. From a software engineering point-of-view, this means that a developer who uses our fault localization algorithm will have to analyze 3.71 less methods on average if he uses Vautrin's fault localization algorithm instead of the Steimann's one.

First, by comparing existing fault localization algorithm listed in Table 5.2 (column 2-6), on this dataset, the best fault localization algorithm so far is Steimann: it scores better in 6 cases out of 10. Thus, in the rest of this question, we always refer to it for comparison, and consider it at "the state-of-the-art" (we note that this is always qualified with respect to the dataset under consideration, another system may be better on another dataset). In all cases, Vautrin improves the performances of Steimann's fault localization

[3]the lower the better in general, there may be isolated bugs that are better localized with one approach even if the average performance over all bugs is worse

Table 5.3: Number of perfect predictions for different fault localization algorithms (i.e. the faulty method is ranked at top). The higher, the better. Best score are bold-faced.

Software	**#Faults**	**T** 2002	**O** 2006	**Z** 2007	**N** 2011	**S** 2013	**V** 2016	**Impr.**
AC Codec	543	77	221	228	237	237	**251**	6%
Daikon	352	27	38	38	39	39	**42**	8%
Draw2d	570	48	84	85	87	86	**106**	23%
Eventbus	577	26	131	135	103	148	**163**	10%
Htmlparser	599	113	193	197	200	204	**237**	16%
Jaxen	600	121	229	252	257	249	**301**	21%
Jester	411	30	45	45	49	49	**105**	114%
Jexel	537	86	293	301	341	331	**349**	5%
Jparsec	598	69	199	206	216	214	**325**	52%
AC Lang	599	233	384	392	407	408	**431**	6%
Total	**5,386**	**830**	**1,817**	**1,879**	**1,936**	**1,965**	**2,310**	**18%**

algorithm as the wasted effort values are always lower for Vautrin. The lowest relative improvement is for Daikon which have a wasted effort of 124.65 methods using Steimann and 121.07 methods using Vautrin, which represents a relative improvement of only 3%. The highest relative and absolute improvement is for Draw2d. Its relative improvement is 55%, with a wasted effort of 24.01 using Steimann and 15.46 using Vautrin. This represents an absolute improvement of 8.55 methods. In other words, a developer would not waste his time in uselessly inspecting 8 methods. The only case on which Vautrin has worse results than one of the other considered fault localization algorithms is for Jparsec for which the wasted effort goes from 3 using Zoltar to 3.53 using Vautrin.

Recall that our core intuition is that the spectrum-based fault localization algorithm misses causal information about the propagation of a fault in the program. Using our approach based on call graph enriches the fault localization process with causal information. Our empirical observations validate this core intuition. Using a causal graph obtained from the call graph for filtering out suspicious methods gives better results.

> On the considered dataset, Vautrin consistently improves the wasted effort for method-level fault localization, from 3% to 55%, with an average of 14%.

Research Question 5.2 Does Vautrin give better perfect predictions than the state of the art?

As presented in Section 5.3.1.1, a perfect prediction is a prediction where the faulty method is ranked at the top, and is the single method predicted at rank #1 (i.e., has a wasted effort equal to zero). In such a case, the developer does not wait a single minute, and the method that he starts to analyze is the one in which he will write the fix.

Table 5.3 reports the number of perfect predictions for the faults in the dataset (on average of the cross-validation). The first column is the subject program name, the second column is the number of fault considered and the third to the seventh column give the perfect prediction rate for respectively Tarantula, Ochiai, Zoltar, Naish and Steimann. The two last columns are the number of perfect predictions with our fault localization algorithm, Vautrin and the relative improvement obtained using Vautrin compared to Steimann.

As an example, if we consider the 598 faults for JParsec subject program in the dataset,

there are 214/598 of them (36%) for which Steimann makes a perfect prediction. For the same subject program, Vautrin's perfect predictions are 325/598 cases (54%), which represents a relative improvement of 52%.

If we observe only the fault localization algorithm under comparison (and not our technique), we observe that Naish and Steimann are the two fault localization algorithm with the highest number of perfect predictions. In 4 cases out of 10 Naish gives the highest number of perfect predictions, in 3 cases out of 10, Steimann does, and in the 3 remaining cases, both have the same number of perfect predictions. Now, we compare against Steimann as we have done in Research Question 5.1. Thus, the improvement may be slightly overestimated in cases where Naish reports better perfect prediction scores than Steimann.

For 10 programs out of 10, Vautrin obtains a higher number of perfect predictions than using Steimann. The best relative improvement is for Jester for which the number of perfect predictions is 49 for Steimann and 105 for Vautrin, which represents a relative improvement of 114% (more than twice as many perfect perfections). The smallest relative improvement is for Jexel which goes from 331 with Steimann to 349 with Vautrin, that is, an improvement of 5%.

To sum up, Vautrin also achieves the best result according to the amount of perfect predictions. On the benchmark under consideration, there are 18% more faults which are perfectly predicted using our technique.

Research Question 5.3 What is the execution time cost of fault localization with Vautrin?

As presented in Section 5.3.1, our approach is composed of four steps: generating the graph, performing mutation analysis, obtaining the causal graph and predicting faulty elements when facing a fault. Since we use an existing dataset, we do not have two important measures. The first is the time needed for the generation of mutants. The second is the time required for analyzing the subject program spectrum, i.e., the execution of tests for obtaining the propagation paths. These are used to compute spectrum suspiciousness.

We compute the time cost for all steps but the two cited and report them in Table 5.4. The first column is the name of the subject program. The second column is the time for generating the call graph. The third column is the time required for computing causal edges based on mutation results. This time is for one fold in our setup, i.e., for 90% of the available mutants (e.g., 488 mutants for codec). The last column is the average time for predicting the faulty method for one single fault. All times are expressed in seconds. All experiments were made on a HP EliteBook 8570w Mobile Workstation, i7-3740QM quad core, 2.7Ghz, under Arch Linux. As an example, let us consider Jester: each of the three steps lasts less than one second.

If we take a look at the graph generation times, we observe that the generation of 9 out of 10 graphs takes less than 5 seconds. The only exception is Daikon with the slowest time: 27 seconds. The average time is 4 seconds. The time required to obtain a causal graph is generally fast as it takes less than a second in 7 cases out of 10. In the three remaining cases, it takes up to 8 seconds. These two phases are meant to be done offline, for instance every night on a continuous integration server. This experiment suggests that the graph building phase and the causal graph extraction phase do not take too long for this scenario. However, we expect the time for mutation analysis to be much larger.

Table 5.4: Times (in seconds) required for each steps of Vautrin for which we have the measures. The mutation and spectrum analysis time is not reported in the benchmark paper [145].

	Offline		**Online**
	Graph	**Causal Graph**	**Graph-Susp.**
AC Codec	1s	< 1s	< 1s
Daikon	27s	< 1s	< 1s
Draw2d	2s	2s	< 1s
Eventbus	2s	< 1s	< 1s
Htmlparser	2s	6s	< 1s
Jaxen	2s	8s	< 1s
Jester	< 1s	< 1s	< 1s
Jexel	< 1s	< 1s	< 1s
Jparsec	2s	< 1s	< 1s
AC Lang	4s	< 1s	< 1s
Average	**4s**	**2s**	**< 1s**

Regarding the prediction times, it always takes less than 1 second (with an average time of 45ms). This step is meant to be done on-the-fly within the development environment. To this extent, it is acceptable for developers to wait for a couple of milliseconds to get the fault localization diagnosis.

For developer usage, Vautrin does not impose a significant overhead compared to spectrum-based fault localization. In addition to the time required to run the test suite, it adds a step which lasts less than 1 second.

Research Question 5.4 To what extent does Vautrin fall back to the traditional, graph-less, fault localization?

As presented in Section 5.1.2, Vautrin uses a causal graph to filter out suspicious methods. However, it happens that the intersection of reachable nodes is empty. In this case, Vautrin returns fallback ranking from the Steimann's fault localization algorithm. We analyze the number of cases with fallback from the results already discussed in Research Question 5.1 and 5.2. For the sake of space, we do not report the whole data.

The worst case is Daikon, for which Vautrin falls back in 82% of the time which means that for 290 faults over 352, we are not able to improve the score given by Steimann's fault localization algorithm. For Daikon, Vautrin is able to perform graph-based causal reasoning in 62 cases. On the other side, for 5 subject programs out of 10, fallback happens in less than 25% of the considered faults: Jester, Jexel, Codec, Htmlparser and Eventbus which fall back in respectively 8%, 12%, 15%, 22% and 22% of the faults. For Jester, Vautrin does graph-based reasoning in 378 faults over 411. In total, for 3,883 faults over 5,386, Vautrin predicts faulty elements based on the intersection of transitively reachable nodes.

For the majority of faults (72%), Vautrin has enough information to go beyond simple spectrum-based analysis and to perform graph-based reasoning.

Research Question 5.5 Do non-causal edges have an impact on fault localization effectiveness?

To answer this question, we look at the improvement in terms of wasted effort and perfect prediction for the dataset with and without considering causal edges. For sake of space, we discuss the empirical results inline. In 9 programs out of 10, considering causal edges improves the average wasted effort. The improvement of performances using the raw call graph and the causal one ranges from 0.21 method for AC Lang to 7.15 methods for Draw2d. However, in 1 case out of 10 (Daikon), there is a decrease in the average wasted effort when considering causal edges. This again suggests that Daikon is an outlier, for which Vautrin's technique for approximating the causal relationship is not working. We explain the reason behind this result at the end of this section. Considering the perfect predictions, using causal graph with mutants does improve the prediction for all programs compared to performance when using call graphs directly.

To further understand the propagation using causal graph, we have measured the proportion of edges considered as causal in the causal graph. It depends much on the subject program: in all cases, Vautrin considers from 0.4% to 63.3% of edges as causal. In 8 cases out of 10, Vautrin considers more than 15% of the edges as causal. The two remaining cases only consider less than 5% of the edges as causal. Daikon is the smallest causal graph with 0.4% of the causal edges. We remark that this proportion of causal edges is connected to the frequency in which the approach fallback. Indeed, the less edges are considered, the more the graph is sparse. This implies that in many cases, the intersection of reachable nodes is empty because of sparseness. That is the reason why Daikon fallbacks so frequently (for 82% of faults): because its causal graph consider so few edges. To us, given that Daikon has a call-graph with a large number of nodes, this means that the number of mutants is too small to sufficiently approximate the causality, which leads to poor results.

To sum up, our approximation of causal effects based on mutants and call graphs is an important factor behind the effectiveness of Vautrin reported in RQ1 and RQ2. Over the considered dataset, Vautrin finds 9% of edges on average for which causal evidence exists and are kept from the call graph.

Research Question 5.6 What is the best spectrum metric to be used with Vautrin?

As presented in Section 5.1, within an equivalence class of graph-suspicious elements, Vautrin uses a spectrum-based metric to assign scores to rank suspicious elements. In all experiments, we have used Steimann for spectrum suspiciousness, because it is the best according to the experiment reported in Table 5.2. What if we use Vautrin with the other fault localization algorithm presented in Section 5.3.1.2? We now report on the fault localization effectiveness with other spectrum suspiciousness plugged into Vautrin. We note *Vautrin/Y* when we consider Vautrin using the score obtained using the Y fault localization algorithm. Thus, Vautrin/Steimann stands for our approach using the scores obtained using the Steimann's fault localization algorithm. Due to space limitation, we do not report the whole data.

We observe that Vautrin/Steimann is the best combination in 4 cases out of 10 for the average wasted effort and 7 cases out of 10 for the perfect prediction. Vautrin/Zoltar is the

best in 5 cases out of 10 for the average wasted effort and 1 cases out of 10 for the perfect prediction. Vautrin/Naish is the best in 1 case out of 10 for the average wasted effort and in 6 cases out of 10 for the perfect prediction. Regarding wasted effort, Vautrin/Zoltar may be a better alternative (5 versus 4 for Vautrin/Steimann). But, we have to keep in mind that this combination always produces worse results when we consider perfect predictions. Regarding perfect prediction, Vautrin/Naish is an acceptable alternative (6 versus 7 for Vautrin/Steimann), yet not good for wasted effort. This observation suggests that these two evaluation metrics are not necessarily completely correlated: they capture two different aspects of the fault localization process. If we want to maximize effectiveness with respect to both evaluation metrics (wasted effort and perfect prediction), the best candidate seems to be the Vautrin/Steimann fault localization algorithm, which further validates the choice of Steimann's suspiciousness metric as default choice.

In addition, we setup a small experiment which consists in using a random function for computing the suspiciousness score. Naturally, this experiment shows that the wasted efforts with such a random spectrum fault localization algorithm are really bad (ranging from 74 to 1,058). But, we also observe that applying the graph fault localization algorithm on top of random scores, gives a minimum improvement of 19%, an average improvement of 211% and a maximum improvement up to 540%. This shows that the causality approximation by computing causal edges is indeed effective, even using the worst possible suspiciousness metric we can imagine.

To sum up, to discriminate within an equivalence class of graph-suspicious elements, the best spectrum-based metric to be used with Vautrin is Steimann's as it outperforms the other ones as much with respect to the wasted effort as with respect to the number of perfect predictions.

5.4 Threats to Validity

Our results are of computational nature. A major bug in our software can invalidate our findings. We have published all our code on Github so as to facilitate reproduction and falsification of our results, if necessary.

Our approach uses mutants for two different purposes. The first is to obtain a causal graph, the second is for assessing the fault localization effectiveness. There are threats to validity for both aspects.

For obtaining a causal graph, the quantity and the characteristics of the mutants can impact our findings. The characteristics of the mutation include the operator and the candidate element for mutation. It may be possible that the considered operators and elements may not be the best ones for approximating the causality. However, to avoid being biased by the dataset, we considered an external, peer-reviewed one, that is unbiased with respect to our approach.

For assessing the fault localization, the dataset of Steimann et al. composed of mutants, are considered as artificial faults. There is little evidence that a mutation is always equivalent to a real fault [74, 9].

We consider in this chapter the 10 Java subject programs from Steimann et al.'s dataset to conduct our experiments. However, these 10 programs may have specific graph structures due to developer choices and/or to the used programming language. As a

consequence, our results may only be valid for Java subjects, or even worse, only valid for the subject programs under study.

5.5 Conclusion

We have presented Vautrin, a novel approach for fault localization. Vautrin is based on the idea of approximating causal effects at the method level: it consists of extracting an approximate causal graph out of a call graph based on execution information obtained from mutation testing.

We have evaluated our approach on the dataset by Steimann et al. The evaluation setup results in 5,386 fault localization diagnosis. Overall, Vautrin is able to make 2,310 perfect predictions, which means that it predicts the faulty method on the top of ranked ones. Our experiments show that Vautrin outperforms the most recent algorithms with an improvement of wasted effort ranging from 3% to 55% and an improvement of up to 114% for the number of perfect predictions.

6

Generation of Synthetic Software Dependency Graphs

"Men are not disturbed by things, but the view they take of things."
— Epictetus

This chapter at a glance...

In this chapter, the following contributions are presented:

- empirical evidence of the common asymmetric structure of dependency graphs in object-oriented software systems written in Java;
- a generative model of software dependency graphs called GD-GND;
- the validation of the generative model regarding its ability to fit 50 graphs of real software systems totaling 17,209 nodes and 61,824 edges;
- a speculative explanation of the evolution rules of software.

The following publication is related to this chapter:

[1] Vincenzo Musco, Martin Monperrus, and Philippe Preux. A Generative Model of Software Dependency Graphs to Better Understand Software Evolution. *Journal of Software: Evolution and Process*, 2016. Minor Revision.

In this thesis, one of our main goals is to use software graphs and synthetic data to aid developers in their development tasks. In the previous chapters, we presented how call graphs with synthetic faults (generated using mutation testing) can be used to perform change impact analysis and fault localization.

In this last chapter, we propose to change the type of synthetic data from generated faults to generated graphs. Indeed, we want to obtain synthetic graphs which look like real ones and which can be used in the learning and prediction processes presented in previous chapters.

We propose here initial works regarding the generation of such graphs. In this chapter, we consider software dependency graphs of object-oriented software, where each node represents a class and each edge corresponds to a compile-time dependency. Call graph (and causal ones) presented in previous chapters are also dependency graphs at the feature granularity, where only methods are considered. We decide to consider dependency graph at the class granularity in order to get a global view of the program without call graphs side effects. That is, we reduce the chances to have too large graphs as ones obtained using a static extraction process or to have missing edges that would occur in a dynamic extraction process.

In the first part of this chapter, we present a study which puts the light on the common topologies which exist in software dependency graphs. Indeed, finding a common topology in software graphs is a precondition to propose a generative model as we must know the topology of the generated data. To that extent, we extracted graphs from 50 Java programs and run statistical comparisons between each pair of program, regarding the in- and out-degree distribution.

In the second part, we propose a generative model of software dependency graphs. As presented in our intuition in Chapter 1, we want to explore the ability of synthetic data to be used to improve software engineering research. In the rest of this manuscript, we considered software mutants as synthetic data. We propose here to explore another direction where the previous synthetic data (i.e., the mutants) is replaced by another type of synthetic data: generated software graphs.

We propose a generative model based on GNC (Growing Network model with Copying), a generic generation model proposed by Krapivski et al. [82]. We call our model "*Generalized Double GNC*" (GD-GNC) as it uses the GNC attachment primitive. We assess it against real software data used in the first part of this chapter, as well as the one from Baxter and Frean [21], which is, according to our investigation, one of the only other models intended to generate similar software graphs than ours. We use similar comparisons than ones used for the common structure to compare both models. We also explore some other properties such as the diameter, average shortest path length, transitivity and modularity.

Then, based on the GD-GNC primitives, we attempt a speculative explanation of the evolution rules which drive programs. Indeed, if this model produces graphs that fit the empirical data, it would mean that the generative operations are good candidates to describe the core operations which result in software graph structure. In other words, a good generative model may encode the evolution rules that are behind the graph structure of software systems.

This chapter is structured as follows. In Section 6.1, we present our works related to the common structure of dependency graphs. In Section 6.2, we propose GD-GNC, our generative model. In Section 6.3, we present the evaluation of GD-GNC as well as a speculative discussion of evolution rules in programs. In Section 6.4, we discuss the threats to validity and the practical implications. In Section 6.5, we conclude this chapter.

(a) in-degree (b) out-degree

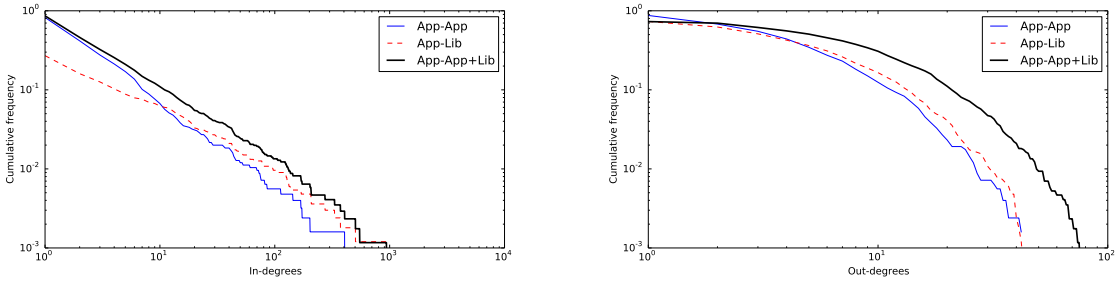


Figure 6.1: Node in- and out-degree distributions for the software package Ant. Three kinds of dependency are considered: app-app, app-lib and both. X-axis are degrees and y-axis is inverse cumulative degree frequencies. Both axes are on logarithmic scale. It is important to differentiate endo- and exo- dependencies because their topology is different.

6.1 Common Structure of Dependency Graphs

We want to determine whether there exist structures shared by different software applications. As the production of these software packages is influenced by different factors (management and development teams, development techniques, etc.), it is *a priori* expected that there is no common structure. To the opposite, finding common structures would be an interesting fact as it would mean that there exist common evolution mechanisms shared across application domains and development styles. Hence, our first research question is:

Research Question 6.1 Are there common structures of node degree distributions in software dependency graphs?

6.1.1 Protocol

Our protocol consists in a technique to extract graphs from software code, applied to a given dataset and a statistically-sound analysis method.

6.1.1.1 Dependency Graph Extraction

We now present our method to extract dependency graphs. We consider object-oriented software written in Java. As presented in Section 2.1.3.4, different horizontal and vertical granularities may be considered to generate software graphs. In this chapter, we focus on the *class* vertical granularity (i.e., one node represents one class), as this is the most important modularity unit in object-oriented software. Among horizontal granularities, two important types are endo- and exo-dependencies also discussed in 2.1.3.4. In this chapter, we only consider endo-dependencies, that is, edges connecting internal nodes of the project to each other and not these connecting to external libraries because we aim at understanding the inner structure of software graphs, regardless of the number of libraries that are included and the amount of calls to library functions.

Figure 6.1 illustrates the difference between both dependencies as a line chart plotting the inverse cumulative distributions of degrees for endo- (app-app), exo- (app-lib) and both dependencies for Apache Commons Ant 1.9.2. Two distributions have to be considered as these graphs are directed: one for in-degree (6.1a) and one for out-degree (6.1b).

Plots are on a logarithmic scale. We see that endo- and exo-dependencies have close yet different distributions. For in-degree, the slope is different. If we consider only app-lib dependencies, i.e., we exclude app-app links (straight thin line), the number of zero in-degree nodes strongly increases because lib nodes never connect to an app node.

The dependency graph is extracted using Dependency Finder[1]. This mature and open-source tool takes as input Java bytecode and outputs all dependencies being found. In this section, graph metrics are computed using the NetworkX[2] library. The dependencies are computed on the main source; test cases are not considered. We use Dependency Finder in class mode (hence discarding package-level dependencies).

The discrimination between endo-dependencies is done using a filter on the package of the source and the destination of the dependency edge. For each subject, we define a package prefix (e.g. "org.myapp") and if both the source and destination fully-qualified classes start with this prefix, it is considered endo- otherwise it is considered as exo-dependency.

6.1.1.2 Dataset

To determine an acceptable dataset size, we examined the dataset size in related publications: they range from 1 to 12 [21, 22, 95, 152, 130, 114, 82]. We aim at building a dataset that is at least as big as these used in previous works. Also, we aim at reusing a peer-reviewed dataset of Java applications in order to mitigate the risk of cherry-picking.

SF100 [50] is a dataset that meets our requirements[3]. It contains 100 Java applications, given as Jar files containing the classes in Java bytecode. In a pilot experiment, we realized that SF100 contains many Java projects that are too small (a few classes only). However, to observe the structural properties we are interested in, a certain size of graphs is necessary. Consequently, in our experiments, we consider the 50 largest projects of SF100, that is all projects containing at least 56 classes.

The resulting dataset is presented in Table 6.1. For each program, this table contains its name, the number of nodes $|N|$ and the number of edges $|E|$ contained in the graphs we extract and the graph density δ (*cf.* Section 2.1.3.3).

The graph size ranges from 56 to 6818 nodes, from 116 to 24096 edges and has a graph density γ value ranging from 0.001 to 0.070.

6.1.1.3 Comparison of Degree Distributions

We want to compare the degree distributions of a set of graphs, i.e., software.

To compare two degree distributions, we use the Kolmogorov-Smirnov statistic K given by Equation (6.1) in which *sup* is the supremum of a set, F_1 and F_2 are the degree distributions to compare and x ranges over degree values.

$$K_{F_1,F_2} = \sup_x |F_1(x) - F_2(x)| \tag{6.1}$$

K is a numerical value that indicates how close two distributions are: the lower K, the closer the distributions. K does not depend on any hypothesis made on the distribution: the Kolmogorov-Smirnov test is non parametric.

[1] `http://depfind.sourceforge.net/`

[2] `http://networkx.lanl.gov/`

[3] The dataset can be downloaded at `http://www.evosuite.org/files/SF100-20120316.tar.gz`

Table 6.1: The 50 Java programs considered in our experiments: we provide their names and basic statistics of their dependency graphs: the number of nodes $|N|$, the number of edges $|E|$, and the density γ. all projects total 17,209 nodes and 61,824 edges.

Project	**|N|**	**|E|**	δ
a4j	84	127	0.018
apbsmem	64	123	0.031
at-robots2-j	302	1,128	0.012
beanbin	88	281	0.037
caloriecount	968	2,595	0.003
corina	1016	3,774	0.004
db-everywhere	93	352	0.041
dom4j	212	1036	0.023
echodep	143	428	0.021
feudalismgame	59	178	0.052
fim1	127	323	0.020
follow	97	290	0.031
geo-google	120	279	0.020
gfarcegestionfa	65	185	0.044
glengineer	60	251	0.071
heal	97	386	0.041
hft-bomberman	138	555	0.029
httpanalyzer	78	171	0.028
ifx-framework	6,818	24,096	0.001
javathena	56	116	0.038
jaw-br	124	305	0.020
jcvi-javacommon	1,024	4,152	0.004
jdbacl	184	490	0.015
jhandballmoves	111	421	0.034
jigen	90	262	0.033
jiggler	187	825	0.024
jiprof	151	367	0.016
jmca	56	146	0.047
jnfe	212	526	0.012
jsecurity	136	336	0.018
jtailgui	58	168	0.051
jwbf	102	472	0.046
lagoon	90	270	0.034
lhamacaw	173	1,344	0.045
lilith	850	3,075	0.004
lotus	59	178	0.052
nutzenportfolio	83	271	0.040
objectexplorer	84	439	0.063
openhre	112	395	0.032
openjms	811	3,081	0.005
petsoar	79	181	0.029
quickserver	107	357	0.031
sbmlreader2	83	311	0.046
schemaspy	120	418	0.029
summa	759	3,791	0.007
twfbplayer	152	451	0.020
water-simulator	120	386	0.027
wheelwebtool	131	520	0.031
xbus	207	968	0.023
xisemele	69	244	0.052

6.1.2 Results

Figure 6.2 shows the plot of the in-degree (6.2a) and out-degree distributions (6.2b) for our dataset. The scale is bi-logarithmic. There is a different line and a different color for each software of the dataset. We make two observations.

First, the in-degree distribution looks different from the out-degree distributions. The out-degree inverse cumulative distributions is more bended, while the in-degree inverse cumulative distributions is straighter. This observation has already been made in previous works [152, 114, 38]: in-degree distributions and out-degree distributions are different. As mentioned above, some have seen power laws in these distributions. This observation has been disputed, indeed others may see log-normal distributions. This controversy has already been studied [95, 130] and thus remains out of the scope of this thesis.

Second, the position and the shape of distributions are graphically similar, this indicates there are common structures across software applications. In order to assess this observation in a statistical manner, we now express our null hypothesis and the alternate hypothesis:

(a) in-degree (b) out-degree

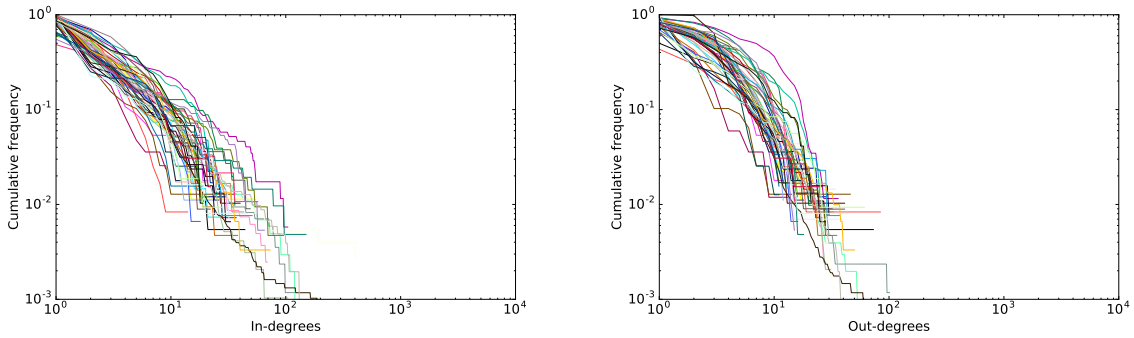


Figure 6.2: Inverse cumulative in- and out-degree distribution for the 50 software applications of our dataset (axes on a logarithmic scale).

Null Hypothesis (H_0): Samples from the software in-degree distributions (or out-degree distributions resp.) are drawn from the same distribution.

Alternative Hypothesis (H_1): Samples from the software in-degree distributions (or out-degree distributions resp.) are not drawn from the same distribution.

Using the *two-sample Kolmogorov-Smirnov test* on each pair of degree distributions of the dataset, we can statistically determine the presence of a similar structure across software applications in our dataset. Based on the statistic K, we can decide on rejecting or not H_0 according to a given confidence level. If H_0 is not rejected, we gain confidence about the common structure for these two programs. On the other hand, if H_0 is rejected, the test outcome cannot be used to conclude about the common structure (which does not necessarily mean that the two software graphs are not drawn from the same distribution).

Table 6.2 gives the results of running 2,450 two-sample Kolmogorov-Smirnov tests with a confidence level α of 0.01 (we need to test each pair of software, hence $C_{50}^2 = 1,225$ tests, which is doubled ($2 \times 1,225 = 2,450$) since we test in-degrees and out-degrees). In this table, the rows provide the results for in-degree, out-degree, and both distributions. The second and third columns provide the number and the ratio of tests for which the two-sample Kolmogorov-Smirnov test has rejected H_0. The fourth and fifth columns provide the information when H_0 cannot be rejected.

As we can see, for 79% of the tested pairs, the common distribution hypothesis cannot be rejected. However, this affirmation does not necessarily involve that there is a unique distribution shared by all these programs. On the other hand, for the remaining 21% of tested pairs for which H_0 is rejected at this confidence level, no conclusion can be drawn.

6.1.3 Summary

For the majority of subjects, there is not enough evidence to reject the hypothesis of a common structure. What is the reason behind this common structure? It is not due to the fact that all applications were developed by the same team. They were indeed developed by different people with different background from all over the world. In this paper, we explore a specific assumption: the way people evolve software is similar across projects, and as a result, software applications share a common structure eventually.

Table 6.2: Number of times the H_0 hypothesis is rejected or accepted for in-degree, out-degree and both cumulative degree distributions according to the two-sample Kolmogorov-Smirnov test with a confidence level α of 0.01.

	H_0 **Rejected**		H_0 **Not rejected**	
	Count	Ratio	Count	Ratio
In	208/1,225	17%	1,017/1,225	83%
Out	300/1,225	24%	925/1,225	76%
Total	**508/2,450**	**21%**	**1,942/2,450**	**79%**

6.2 A Generative Model for Dependency Graphs

In this section, we present a new generative model of software dependency graphs. This model generates an arbitrary number of artificial dependencies. It is parametrized by three values: the expected number of nodes and two probabilities.

6.2.1 Assumptions

Our model is built on three assumptions on how software evolution works.

Assumption #1 (increment) When creating new features (new classes), they are built on top of existing classes. Hence when a new node is added to the generated graph, it is directly connected to existing nodes.

Assumption #2 (remix) New features (new classes) depend on classes that are designed to be used together or at least that are compatible with each other.

Assumption #3 (refactoring) Developers sometimes identify reusable units. In that case, they create a new class. This new unit of reusable functionalities is used later on.

6.2.2 The Generalized Double GNC Algorithm (GD-GNC)

We name our generative model of software dependency graphs GD-GNC. It generalizes the GNC model:

GNC [82] this model is an iterative algorithm where, at each iteration, a new node n_i is added to the graph and connected at random to a set of already existing nodes. In GNC, an existing node n_j is selected according to a uniform distribution and directed edges are created from the new node n_i to this node n_j along with all its successors. This "GNC-Attach" primitive is illustrated in Figure 6.3. Algorithm 6.1 shows the core primitive for attaching nodes using GNC. GNC requires one parameter: the number of nodes of the resulting graph. It executes n times the core function to create a graph with n nodes.

GD-GNC implements the three assumptions presented in Section 6.2.1 in an algorithmic way. Its pseudo-code is shown in Algorithm 6.2.

It consists in a main loop which at each iteration:

1. adds a new node n_i to the existing graph (**Assumption #1 – increment**);

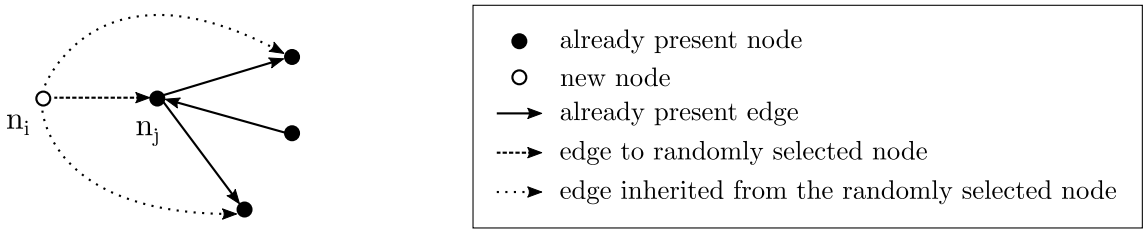


Figure 6.3: Illustration of GNC-Attach, the GNC primitive operation. The gray node n_i is a new node added to the graph using the GNC primitive. The central node n_j is selected uniformly at random and a directed edge is added from the new node to it (dashed edge). Then, a directed edge is also added from the added node to all successors of n_j (dotted edges).

Algorithm 6.1: GNC-Attach Algorithm.

Input: $G_{\mathcal{N},\mathcal{E}}$ the digraph to which a new node n_i is going to be added. $G_{\mathcal{N},\mathcal{E}}$ is composed of two elements: the set of existing nodes ($\mathcal{N}$) and the set of existing directed edges ($\mathcal{E}$).

Output: $G_{\mathcal{N},\mathcal{E}}$ has been updated with a new node, and a set of new directed edges.

```
1 Function GNC_Attach(G_{N,E}, n_i) is
2     Select uniformly at random a node n_j in G_{N,E}
3     Add an edge from n_i to n_j
4     for all edges (n_j, n_d) leaving n_j do
5         Add an edge from n_i to n_d
```

2. selects an existing node n_j uniformly at random;

3. adds edges leaving n_i:

   - with probability p, n_i is connected to n_j in the same way as in the GNC-Attach algorithm (i.e., a directed edge is created from n_i to n_j and from n_i to each successor of the node n_j); With probability q, we repeat this GNC-attachment once: an existing node is selected uniformly at random; if this selects the same node as previously, this second operation aborts (**Assumption #2 – remix**).
   - with probability $1-p$, n_j is connected to n_i (**Assumption #3 – refactoring**).

6.2.2.1 Relation to Assumption #1 and #2

The first operation of the model is a node creation followed by an attachment to existing nodes using a GNC-Attach. This represents the creation of a new class implementing a new feature. This new feature depends upon existing classes. The point of being attached to all dependent classes of a class means that these classes are already used, depending upon each other. If class X depends on classes A, B and C, it means that A, B and C interact together in a way that is defined by X. When a new node n_i is connected to X by way of a GNC-Attach, it is also connected to A, B and C. In other words, *the new class n_i creates a novel interaction between A, B, and C.* Executing GNC-Attach twice reflects the fact that the new class mixes two existing groups of classes. In the model, there is never more than two groups of already interacting classes being linked from a new node

Algorithm 6.2: Iterative algorithm for the "GD-GNC" generative model.

Input: N the number of iterations to execute, p the probability to perform a GNC-Attach, and q the probability to do a double GNC-Attach.

Output: a digraph $G_{\mathcal{N},\mathcal{E}}$ is composed of two sets: nodes ($\mathcal{N}$) and directed edges ($\mathcal{E}$)

```
1  begin
2      N ← ∅
3      E ← ∅
4      for i ∈ {1, ..., N} do
5          Create a node n_i and add it to N
6          if rand() ≤ p then
7              GNC_Attach(G_{N,E}, n_i)
8              if rand() ≤ q then
9                  GNC_Attach(G_{N,E}, n_i)
10         else
11             Select uniformly at random a node n_j in G_{N,E}
12             Add an edge from n_j to n_i
```

(a new feature). We did experiments allowing more than 2 successive GNC-Attach in the GD-GNC main loop: this has never significantly increased the fit to real data. Figure 6.4 illustrates the in- (6.4a) and out- (6.4b) degree distribution for a graph generated with 1000 nodes using the GNC-Attach primitive run one, two and three times. As we can see, running a third time the GNC-Attach primitive does not improve the resulting plot. Moreover, it makes the plot running out from the ranges observed on empirical software. Thus, the best fit is between one to two times the GNC-Attach primitive. These groups were already interacting together separately (as witnessed by the fact that another class depends on the classes of each group). The new call brings the two groups together to provide one class with new and useful functionalities.

6.2.2.2 Relation to Assumption #3

The second core operation of GD-GNC (the top-level else condition) is a reverse attachment from an existing node to the added node. It represents a refactoring operation, where a piece of code is extracted from an existing class, in order to ease reuse and to simplify the code. Once the refactoring is performed, the newly created class is ready for being reused. This can happen in subsequent iterations of the algorithm with the GNC-Attach.

6.2.2.3 Analysis

We note that this model never modifies existing edges: at each iteration, GD-GNC adds a single node and a set of edges. We emphasize the fact that no explicit preferential attachment is explicitly coded in the algorithm. However, an implicit preferential attachment is still present. GNC-Attach connects a new node to an existing one, but also to all successors of this existing node. As a consequence, if a node has a high in-degree, it has a higher probability of receiving a new edge.

(a) in-degree (b) out-degree

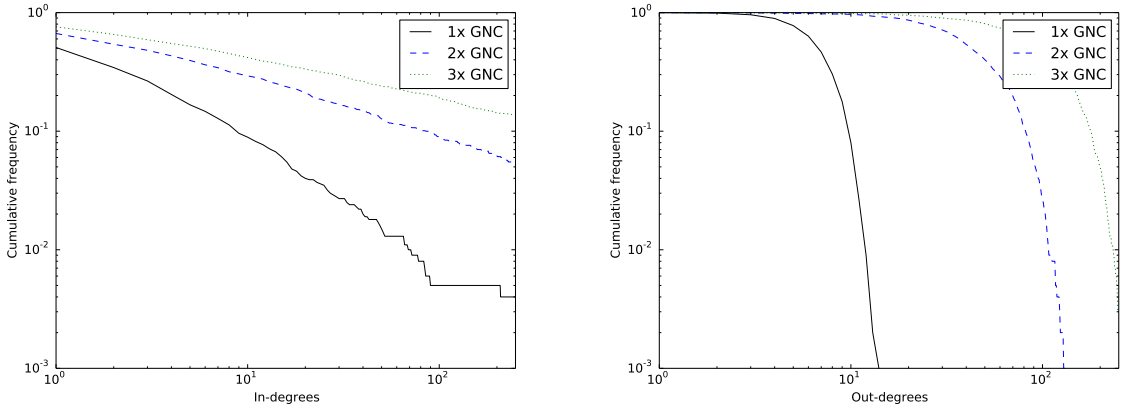


Figure 6.4: Inverse cumulative degree distributions for in- and out-degree for a 1000 nodes graph generated with the GNC-Attach primitive run one (plain), two (dashed) and three times (dotted).

6.2.2.4 Parameters

Two parameters are required by GD-GNC and influence the growth of the graph. p determines whether the new node n_i must be added following GNC-Attach or it is connected to by an existing node. Hence, there is a proportion of p nodes that have no outgoing edges (sink nodes). q is the probability of a second attach conditioned on p. Increasing the number of GNC-Attach impacts the in-degree distribution, the distribution decreases more slowly when q increases. Regarding the out-degree, the convexity of the distribution increases as q increases. GD-GNC is a generalization of GNC-Attach: GNC-Attach is a special case where $p = 1$ and $q = 0$. Figure 6.5 illustrates the influence of the p parameter on the generated graph using the GD-GNC algorithm where $q = 0.5$ for a generated graph containing 1000 nodes. The figure shows the proportion of nodes (x-axis) which have a certain in-/out- degree (y-axis). Figure 6.5a is for in-degree and Figure 6.5b for out-degree. As we can see, as p increases, the number of nodes with a null in-degree increases and out-degree decreases. The interesting point with this figure is to notice how we can approximately fix the number of null in-/out-degree nodes thanks to the p parameter.

6.3 Experimental Evaluation

In this section, we want to determine whether GD-GNC can generate graphs that are similar to real software graphs.

6.3.1 Comparison

We have explored several generative models described in the related work. We emphasize the fact that we are interested in models that may be interpreted in terms of rules of software development. However, according to our experiments, Baxter and Frean's model [21] is the only one which gives reasonable in- and out-degree distributions. The other ones are discarded because the resulting degree distributions are really different from the ones we observe in our dataset. Hence, we consider Baxter and Frean's model as a baseline.

(a) in-degree (b) out-degree

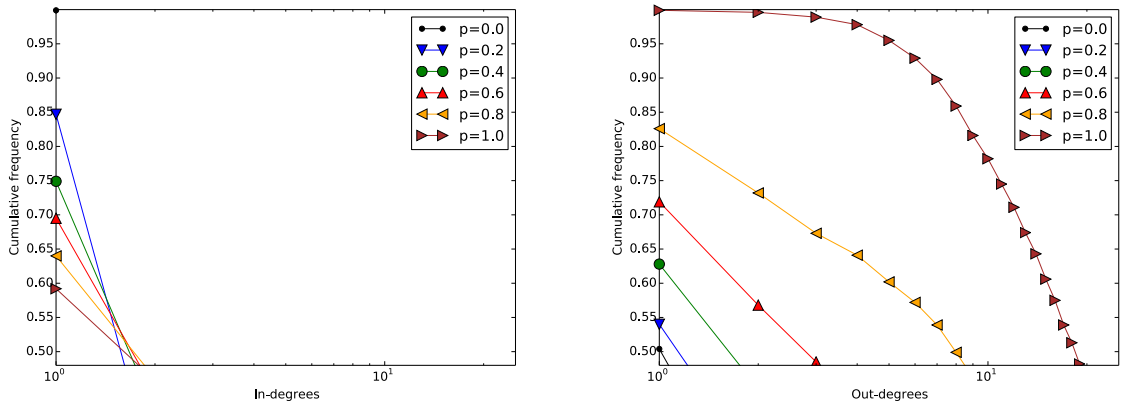


Figure 6.5: Influence of the p parameter on the generated graph using the GD-GNC algorithm where $q = 0.5$ for a generated graph containing 1000 nodes.

Baxter and Frean's model [21] encodes preferential attachment based on the out-degree of nodes. Its logic is based on:

1. edge creation;
2. edge transfer between nodes of the graph: a transfer means that either the source or the destination of an edge is modified and points to another node of the graph.

Hence, we formulate our research question as:

Research Question 6.2 How does GD-GNC compare to other generative models? We consider Baxter & Frean's model which is the best model up to date.

6.3.2 Protocol

To address this question, we first run a parameter optimization (Section 6.3.2.2) for each model (GD-GNC and Baxter & Frean) on all programs of our dataset. Table 6.3 reports the best parameter determined for each project. Then, we generate 30 synthetic graphs with each model, using the best parameters found for fitting each program. Finally, we compute the inverse cumulative degree distribution of each graph and we compute the fitness value according to its Δ value defined by Equation (6.2).

6.3.2.1 Error metric

To statistically determine which model generates the closest graph to the true one, we compare the Kolmogorov-Smirnov statistic K (as presented on section 6.1.1.3) for in-degree and out-degree cumulative degree distribution of the generated graph $\mathcal{G}$ and the real graph $\mathcal{R}$. For this purpose, we define the Δ function, as shown in Equation (6.2), which is the maximum between the two Kolmogorov-Smirnov distances: the distance $K_{\mathcal{R}_{in},\mathcal{G}_{in}}$ between the in-degree cumulative distribution of the artificial graph $\mathcal{G}_{in}$ and the real one $\mathcal{R}_{in}$, and likewise for the out-degree distribution ($K_{\mathcal{R}_{out},\mathcal{G}_{out}}$, $\mathcal{G}_{out}$, $\mathcal{R}_{out}$ resp.).

$$\Delta_{\mathcal{R},\mathcal{G}} = \max(K_{\mathcal{R}_{in},\mathcal{G}_{in}},\ K_{\mathcal{R}_{out},\mathcal{G}_{out}}) \tag{6.2}$$

Table 6.3: Result of the grid-search for determining the best parameters for Baxter & Frean and GD-GNC generative models according each programs of the dataset

	Baxter	GD-GNC			Baxter	GD-GNC	
Project	γ	**p**	**q**	**Project**	γ	**p**	**q**
a4j	0.9	0.3	0.7	jiggler	0.4	0.9	0.2
apbsmem	0.1	0.7	0.8	jiprof	0.2	0.4	0.5
at-robots2-j	0.1	0.4	1.0	jmca	0.2	1.0	0.2
beanbin	0.2	1.0	0.6	jnfe	0.3	0.4	0.9
caloriecount	0.2	0.4	1.0	jsecurity	0.3	0.5	0.5
corina	0.1	0.6	0.7	jtailgui	0.2	0.6	0.9
db-everywhere	0.3	0.8	0.7	jwbf	0.1	0.6	1.0
dom4j	0.2	1.0	1.0	lagoon	0.2	0.7	1.0
echodep	0.1	0.4	1.0	lhamacaw	0.2	0.9	0.9
feudalismgame	0.2	0.5	0.6	lilith	0.1	0.5	0.8
fim1	0.1	0.5	0.9	lotus	0.1	0.5	0.9
follow	1.0	0.3	0.9	nutzenportfolio	0.1	0.4	1.0
geo-google	1.0	0.6	0.8	objectexplorer	0.1	0.4	1.0
gfarcegestionfa	0.1	0.4	0.9	openhre	0.1	0.5	1.0
glengineer	0.1	0.4	0.5	openjms	0.3	0.4	0.9
heal	0.1	0.3	0.8	petsoar	0.3	0.5	0.3
hft-bomberman	0.1	0.5	0.8	quickserver	1.0	0.3	1.0
httpanalyzer	1.0	0.4	0.9	sbmlreader2	0.1	0.6	1.0
ifx-framework	1.0	0.5	0.9	schemaspy	0.1	1.0	1.0
javathena	0.1	0.5	0.8	summa	0.2	0.6	0.7
jaw-br	1.0	0.7	0.9	twfbplayer	0.1	0.6	1.0
jcvi-javacommon	0.1	0.5	0.9	water-simulator	0.1	0.5	0.9
jdbacl	1.0	0.5	0.5	wheelwebtool	0.2	0.5	0.7
jhandballmoves	0.1	0.4	0.9	xbus	0.4	0.8	0.2
jigen	0.2	0.6	0.6	xisemele	0.1	0.6	0.8

Combining in-degree and out-degree distributions is necessary because both distribution are intimately related to each other: considering only in-degree or out-degree distribution would be meaningless as a good in-distribution does not necessarily involve a good out-distribution and vice-versa. Δ is a measure of error and we aim to minimize it (the lower the better).

6.3.2.2 Parameter Optimization

To determine the best values of p and q to generate graphs as close as possible to real ones, we perform a grid-search of the space of parameters, trying each value for p and q between 0 and 1 with a step of 0.1. For each parameter value, we generate 30 graphs. Then, we use the K statistic to assess the fitness between the true graph and the generated one. The graph with the smallest K value is the one that is mostly similar to the real graph. Table 6.3 reports the best parameters found for each program of the dataset.

6.3.3 Results

For each program in our dataset, Table 6.4 gives the average fit error Δ of Baxter & Frean's model (column labeled **B**) and GD-GNC (column labeled **G**). The column labeled **p** gives the p-value determined using the Mann-Whitney test; this p-value assesses whether one

Table 6.4: Average Δ error of GD-GNC (G) and Baxter & Frean's models (B) expressed in 10^{-3}. The third column gives the p-value determined using the Mann-Whitney test: it assesses whether GD-GNC is significantly different from Baxter & Frean's model. The last column gives the effect size according to Cohen's formula (d).

Project	**B**	**G**	**p**	**d**
a4j	1.56	1.67	0.43	-0.12
apbsmem	1.50	1.71	0.00	0.70
at-robots2-j	1.77	2.33	0.23	-0.39
beanbin	2.26	1.53	0.00	1.92
caloriecount	1.47	1.08	0.00	1.33
corina	1.46	1.79	0.12	-0.40
db-everywhere	2.16	1.08	0.00	1.97
dom4j	1.65	1.09	0.00	1.87
echodep	1.98	2.17	0.15	-0.37
feudalismgame	2.13	1.68	0.12	0.35
fim1	1.74	2.21	0.01	-0.72
follow	1.67	2.93	0.00	-1.42
geo-google	1.67	2.80	0.00	-3.64
gfarcegestionfa	1.96	2.19	0.00	-1.24
glengineer	1.93	1.85	0.01	-0.79
heal	1.24	1.71	0.06	-0.56
hft-bomberman	1.33	1.56	0.88	-0.16
httpanalyzer	1.67	2.81	0.00	-3.00
ifx-framework	1.67	2.11	0.64	-0.36
javathena	1.52	1.94	0.00	-1.04
jaw-br	1.67	2.45	0.00	-0.76
jcvi-javacommon	1.46	1.81	0.08	-0.43
jdbacl	1.67	2.67	0.00	-2.05
jhandballmoves	1.91	2.09	0.76	0.10
jigen	1.32	1.20	0.03	0.61

Projet	**B**	**G**	**p**	**d**
jiggler	1.89	1.17	0.00	2.06
jiprof	1.20	1.40	0.01	-0.76
jmca	2.16	1.64	0.00	0.87
jnfe	1.94	2.24	0.52	-0.27
jsecurity	1.84	1.70	0.02	0.63
jtailgui	1.82	1.67	0.22	-0.21
jwbf	1.66	1.15	0.13	0.53
lagoon	1.40	1.45	0.91	0.04
lhamacaw	1.66	1.14	0.00	1.44
lilith	2.17	2.30	0.00	-0.86
lotus	1.27	1.30	0.11	0.30
nutzenportfolio	1.15	1.49	0.34	-0.34
objectexplorer	1.72	1.79	0.44	-0.26
openhre	1.80	2.03	0.00	-0.84
openjms	1.18	1.28	0.51	0.13
petsoar	1.24	1.39	0.11	-0.39
quickserver	1.67	2.68	0.00	-1.40
sbmlreader2	1.60	1.79	0.22	-0.33
schemaspy	1.86	1.97	0.02	-0.63
summa	1.96	1.39	0.00	2.78
twfbplayer	1.37	1.49	0.63	-0.16
water-simulator	1.59	1.76	0.08	-0.43
wheelwebtool	2.08	2.03	0.48	-0.10
xbus	2.57	1.05	0.00	2.60
xisemele	1.90	2.58	0.02	-0.76

model is significantly better than the other one. The last column labeled **d** gives the effect size according to Cohen's d effect size (*cf.* Section 4.5.2).

These number are interpreted as follows: p is used to ensure B and G do not belong to the same population with a confidence level of 0.01 ($p < 0.01$). The Cohen's d effect size is used to estimate if the mean of Δ increases or decreases from B to G. In other words, if $d > 0$, it implies that Δ decreases between B and G and thus implies that GD-GNC is statistically better.

We observe that GD-GNC in-degree and out-degree distributions are better than Baxter & Frean's ones in 18/50 cases ($d > 0$). In these cases, the GD-GNC algorithm tends to produce synthetic software graphs whose in- and out-degree distributions better fit these of real software dependency graphs.

We now move to a statistical assessment of the fit.

Null Hypothesis (H_0): the Δ error values obtained for GD-GNC and the ones obtained from Baxter & Frean model belong to an identical population.

Alternative Hypothesis (H_1): the Δ error values obtained for GD-GNC and the ones obtained from Baxter & Frean model belong to a different population.

Now considering the p-value, there are 10 subjects that are more similar to the real graphs using GD-GNC (i.e., $d > 0$ and $p < 0.01$) and in the other way, there are 12 subjects for which Baxter & Frean's model is significantly better (i.e., $d < 0$ and $p < 0.01$). For 28 subjects, there is not enough evidence to say that one model or the other is significantly better. To sum up, according to our experiments on the degree distributions, there are 10 subjects for which our GD-GNC model better models the software dependency structure.

6.3.4 Scalar Properties

Though focusing on the presentation of the degree distribution, we also studied many other graph properties. We report the key observations in this section.

Regarding the size of the generated graph:

- Baxter and Frean's model generates graphs that have the correct number of edges, but they do not have the correct number of nodes. The number of nodes of these graphs ranges between half of the real graph and 2.5 times larger. For a given real graph, the graphs generated with Baxter and Frean's model have their number of nodes that varies a lot. The coefficient of linear correlation between the number of nodes of the synthetic graphs and the real graphs is 0.82.
- GD-GNC generates graphs with the correct number of nodes, and generally with a number of edges which is varying between half and 3 times the number of edges of the empirical graph. The coefficient of linear correlation between the number of edges of the GD-GNC graphs and the real graphs is 0.96.

Regarding the diameter and average shortest path length, GD-GNC generates graphs whose the diameter is within a factor of 2 with regards to the empirical graphs: the correlation between the diameter of the empirical graph and the diameter of the GD-GNC graphs is 0.59. Baxter and Frean's generates graphs whose the diameter is typically very different from the empirical one: the correlation between the diameter of the empirical graph and the diameter of Baxter and Frean's graphs is 0.37. The same sort of observation may be made about the expected relation between the diameter and the number of nodes n of a scale-free graph: the diameter scales with $O(\log(n)/\log(\log(n)))$. GD-GNC graphs follow this relation quite well, which is not the case for Baxter and Frean's graphs. Likewise, the average shortest path length exhibit the same behavior.

Regarding the transitivity or clustering coefficient C, the correlation between the value of C for empirical graphs and its value for graphs generated by GD-GNC is 0.38, the one between empirical graphs and Baxter and Frean's graphs is 0.51.

Regarding the modularity Q, the correlation between the value of Q of empirical graphs and its value for graphs generated by GD-GNC is 0.72, the one between empirical graphs and Baxter and Frean's graphs is 0.23.

To sum up, considering the different metrics we have used to compare the performance of GD-GNC with regards to other models is not an easy task. However, as reported above, most often, for each metric, the properties of graphs generated with GD-GNC are more similar to these generated by Baxter and Frean's model.

6.4 Discussion

We now put aside technical considerations and discuss the meaning and validity of our empirical results.

6.4.1 Threats to Validity

Let us now discuss the threats to the validity of our findings. First, we have optimized our model with respect to the fit to in-degree and out-degree distributions. Even if the degree distributions capture many topological properties of graphs, it is only one feature of the structure of the dependency graph. One threat to the validity of our conclusions is that some other important topological properties of software dependency graphs have minor or no impact to degree distributions.

Second, our experiments are done on a dataset of 50 Java software systems. Our findings may only hold for object-oriented code, Java software or even worse, to this particular dataset only. However, for us, a sign of hope is that the degree distributions on other programming languages and systems that are reported in previous works look qualitatively the same [95, 152, 114].

Third, our evolution model is completely expressed in abstract graph terms. We have reformulated the algorithm from a software engineering perspective in Section 6.2. It may be the case that we have correctly extracted the core operations but that, at the same time, we have misinterpreted their meaning. We look forward to more work in this area, to discuss with the community in order to see the emergence of a consensus on the core software evolution mechanisms.

6.4.2 Practical Implications

For Researchers Our model and experiments have shown that remix is likely a fundamental phenomenon of software evolution. Our model is another piece of evidence suggesting that remix-oriented software engineering is key, strengthening existing arguments [19, 61]. We note that research has already made significant advances in supporting remix of groups of classes. For instance, code-completion can work at the level of groups of classes [121] and documentation can be generated to explain common remix strategies [31]. Our results call for more contributions on that topic.

For Practitioners The generative model is primarily intended to validate fundamental hypotheses about software evolution. As such, no practitioner directly uses the model to generate new graphs. Speculatively, we envision that people who write static analysis based on dependency graphs use synthetic graphs generated by our model to validate the scalability of their technique.

6.5 Conclusion

In this chapter, we wanted to use synthetic software graphs in our experiments. The first step was to determine the possibility to generate such graphs. To this extent, we have studied the existence of common structures in many software dependency graphs and devised an experimental protocol to understand the evolution principles that result in such a common structure.

Once we have identified common properties, i.e., the cumulative degree distribution, we have introduced a new generative model: GD-GNC. This model generates graphs whose degree distribution is very close to that of real software: this closeness is assessed with statistical tests.

This is a piece of evidence that the evolution rules encoded in the generative model resemble the actual ones: new features are based on the perpetual remix of existing

interacting classes and refactoring mostly consists in extracting a reusable class from an existing class.

7

Conclusion

"Well done is better than well said."

— Benjamin Franklin

In this last chapter, we conclude this thesis. This conclusion is divided in two parts: first, in Section 7.1, we propose a summary of the work presented in this thesis. Then, in Section 7.2, we present interesting future directions.

7.1 Summary

In this thesis, we answered our two problems presented in Chapter 1 by the mean of graphs and synthetic data obtained using mutation testing.

The first problem was the absence of a systematic evaluation methodology for change impact analysis. The second was that current fault localization techniques do not consider the whole program, ignoring how elements depend on each other. These problems were addressed during my thesis. To that extent, four contributions were proposed.

The first contribution was a direct answer to the first problem. We proposed an evaluation framework for change impact analysis techniques based on a large number of changes. The approach uses synthetic seeded faults to bypass the limitation of real changes: thanks to mutation testing, we ensure that the inserted change is unique. Hence, we can observe impacts related to this single change. Based on this framework, we conducted a study on four different types of call graphs to determine their potential for change impact analysis. Each call graph expresses a programming feature in a way that we analyzed what are the elements that are most responsible of propagation in the software graph.

The second contribution was Vautrin, a tool intended to filter call sites returned by an IDE based on previous executions of the program. Vautrin relies on the concept of causal graph: a call graph on which edges are decorated with weights ranging from 0 to 1. These weights express the likelihood of an edge to propagate a fault. During a learning phase, real test executions are analyzed to capture the causes-effects of bugs and tests in the whole graph. Based on this phase, graph edges weights are updated. A threshold is used to disable some edges and thus filter out the reported impacts.

The third contribution was Strogoff, a new fault localization technique based on a similar learning phase as Vautrin, but where the causal graph is now used to propose a solution intended for fault localization. Indeed, based on the causal graph, the ability

of determining faulty points based on failing tests enables us to improve the state of the art according to the major evaluation metrics. We used the nodes reported as suspicious based on the causal graph in conjunction with classic spectrum-based fault localization techniques to improve their performance. This third contribution is an answer to our second problem as using the causal graph allows to reason on the program as a whole and not only have a limited view of executed elements.

Our last contribution was a generative model for software dependency graphs. This model was proposed in an effort of finding other synthetic data to use in our research. We first studied the presence of some properties in 50 real Java software graphs. We found that these graphs have a common topology regarding their in- and out-degree distribution. Based on this empirical observation, we proposed GD-GNC, a generative model based on GNC. This generative model is able to generate software graphs with similar degree distribution to real software graphs. This fitness was assessed with real distributions and with an existing model intended for same purposes: the Baxter and Frean's one. We concluded this chapter by a speculative discussion of phenomena which motivate such a structure.

7.2 Perspectives

We now present the possible future directions. We structure this section in five aspects that are: improving the propagation profile, improving the time performance, considering other types of software graphs, considering propagation profiles obtained using other granularities and using alternative evaluations for generative models.

7.2.1 Better Propagation Profiles

Our contributions are based on propagation profiles using synthetic faults obtained using mutation testing. A first perspective is to better improve the propagation profile obtained from these mutants.

A way to proceed is to mutate in a cleverer manner. Indeed, in Chapter 3, 4 and 5, mutation is made in a random manner. Instead, it would be possible to mutate in a way to ensure the whole call graph is covered. As an example, mutate each line or each block of the program source code is worth considering.

Many different mutation operators could be used instead of the five mutation operators presented by Offutt [122]. We proceeded this way as we wanted to focus on performance measures based on basic and largely studied ones. However, considering other operators can yield to different results and observations on the data as the propagation profile may differ (i.e., depending on the mutation operator, different types of impacts may be observable). As example of other operators, we can cite ones by Kim et al. [78] and Ma et al. [100] who have proposed operators that are better-suited to object-oriented programs. Ma et al. [101] have even proposed operators for Java programming languages composed of a large number of operators.

Another possible direction is to use advanced machine learning techniques, e.g., classification, made on a set of data observations. Therefore, it is possible to define a set of attributes from the software source code such as modifiers, visibility, number of lines of code, number and types of parameters, etc. A classification algorithm could use such information to learn association rules between these attributes and the appearance of a failure in the program. With a large corpus of failures, the model could become more precise and be used to help in change impact analysis or fault localization technique.

7.2.2 Faster Computation of Propagation Profiles

A second perspective is to speed up the computation of our causal graph based on mutations. A possible approach is to tailor mutation analysis, i.e., to determine how to conduct the generation of mutants in order to maximize the quantity of information learned about impact propagation and to minimize the number of required mutants. Consequently, instead of mutating in a totally random manner, one idea is to choose the mutation operator in a clever way, depending on the code element under consideration.

In this way, we could mutate in a manner to better cover the code and improve the quality of learned information for studying propagation. This results in decreasing the number of generated mutants as we avoid to generate mutants which will report similar propagation information in the graph. As a consequence, if less mutants are required, the approach will be faster.

7.2.3 Other Types of Software Graphs

In this thesis, we used a static approach for building the call graphs used in our experiments. Instead of obtaining the graph by statically analyzing the source code, one direction would be to use a dynamic approach which will return a graph containing only nodes/edges really called at run-time. The dynamic call graph would certainly have less edges than the static one as many calls would not occur in real execution. This can lead in an improvement regarding execution time, as well as surprising discoveries. Indeed, edges not called in the dynamic analysis may act as a filter that would remove some false positives reported by the static approach. However, the time required to obtain the data (i.e., the graph) would be longer as dynamic approaches generally instrument and execute the code, which is more expensive to compute than a static approach.

In Chapter 3, 4 and 5, we considered the class hierarchy analysis call graph. In such an approach, the obtained graph is not too dense to be handled by a computer quickly and easily, but not too sparse in order to be able to learn concrete things from it. However, better performances can be obtained by pruning the graph based on phenomena which are not likely to propagate a fault. As an example, all cycles where `A` is connected to `B`, `B` is connected to `C` and `C` is connected to `A` can easily be removed as, in a propagation point of view, they may be useless as such cycles increase the size and complexity of the graph. The same observation can be made for isolated nodes or nodes with only one edge in specific cases. However, their removal should not be systematic. We must promptly understand their usefulness in the approach under investigation. If we want to be able to detect impacts on these nodes or if they play a central role in our impact prediction process, they should not be removed. For instance, we use tests in our approaches, tests are generally isolated nodes with a few (and possibly only one) edges. As the tests are central in our propagation analysis, they should not be removed.

7.2.4 Propagation Profiles at Other Granularities

Graphs studied in this manuscript are not the only good candidates for studying software engineering. Other granularities may be explored to speed up the process on very large projects. Moreover, even if some types of graphs are slow to compute and to work with, due to their size, some types of edges may be extracted from these to work with hybrid granularity graphs. As an example, the program dependence graph can give much more details about the propagation, but the fact they are dense would require more time and memory to work with these. An hybrid approach would have the speed advantage of the

call graph and the details (or at least partial details) of the program dependence graph. As a consequence, discovering some elements in source code which are subject to propagate the error could enable to embed some finer granularity nodes/edges to better materialize the propagation phenomenon.

7.2.5 Alternative Evaluation of Generative Models

A direction which would be interesting is to study graph motifs as an alternative evaluation metric for our generative model. Graph motifs are patterns consisting of a small amount of nodes connected to each other in a certain way.

These may turn to be valuable to determine the appropriateness of generated graphs and to study software shapes in a dependency graph. We hypothesize two kinds of motifs: ones resulting from design patterns introduced at once in software and others that are evolving, kinds of "emergent design patterns". Identifying them would sketch a new interesting light on software evolution.

A

Reproducible Research

Most of the source code can be downloaded from the following URLs:

`https://www.github.com/v-m`
`https://www.github.com/v-m/PropagationAnalysis`
`https://www.github.com/v-m/PropagationAnalysis-dataset`
`https://www.github.com/v-m/gdgnc`

The second and third URLs are respectively the source code for propagation analysis research and the considered dataset. The last URL contains the code and dataset used for the generative model.

The source code related to propagation analysis is made of four tools:

1. *simple mutation framework (smf)*, our mutation tool. Several mutation tools exist, for instance Javalanche[1], or Pitest[2]. However, we need a full control over the mutation process and on extracted information. So, we have implemented ours.

2. *softminer*, a tool for extracting call graphs from Java source code;

3. *pminer* implements the prediction analysis (Algorithms 3.1 and 3.2, propagation prediction from call graphs and accuracy computations);

4. *softwearn* is responsible of the learning part (i.e., learning weights for call graphs).

Both *smf* and *softminer* use *Spoon* [127], an open-source library for analyzing and transforming Java source code.

Graphs used for change impact analysis contributions are obtained using Git tag `g1` of the project.

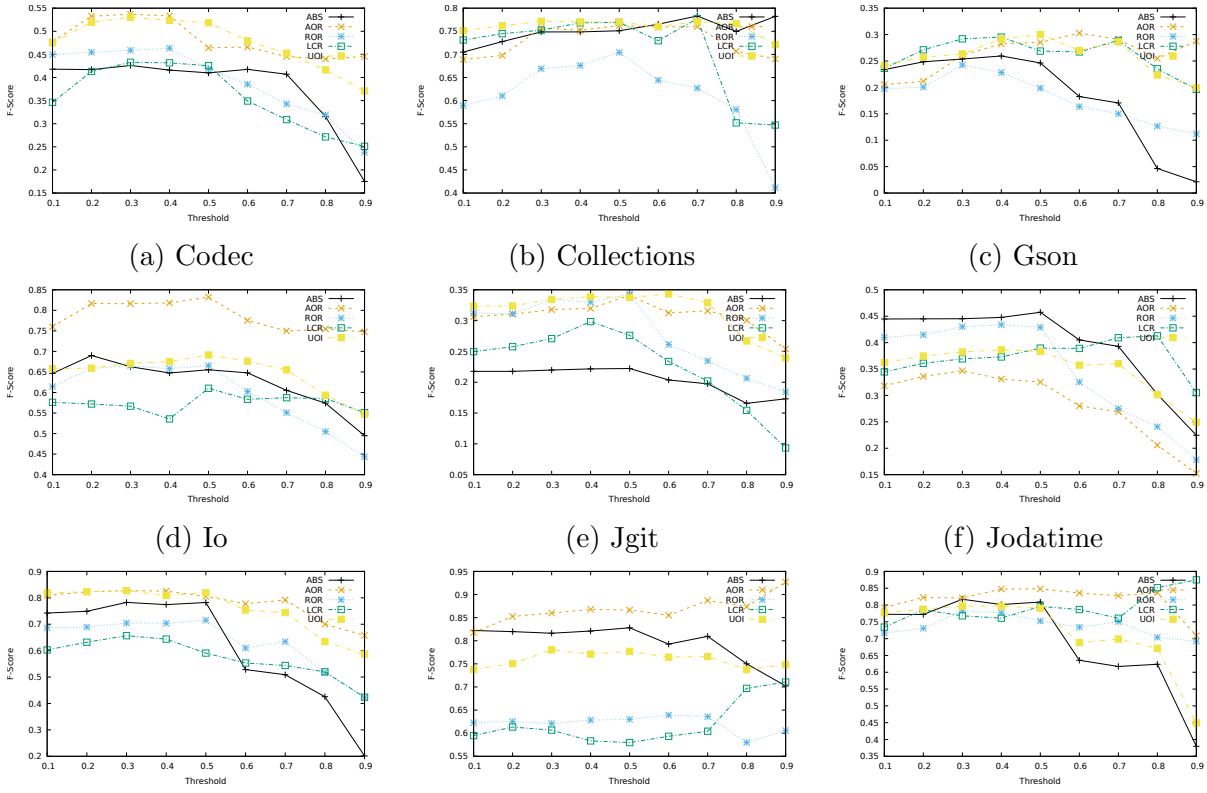
Dependency graphs used in Chapter 6 are obtained using the Dependency Finder tool[3].

[1] `http://javalanche.org`
[2] `http://pitest.org`
[3] `http://depfind.sourceforge.net/`

B

Details on Parameter Optimization

In this appendix, we report detailed plots related to Figure 4.4 in Chapter 4. Here, we show the threshold searched for each project and each mutation operator considering the Dichotomic algorithm.



(a) Codec (b) Collections (c) Gson

(d) Io (e) Jgit (f) Jodatime

(g) Lang (h) Shindig (i) Sonarqube

Figure B.1: The impact of the prediction threshold (x-axis) on the F-score (y-axis) for each considered mutation operator.

C

Generative Model – Other directions

C.1 Generic Graphs

In Chapter 6, we have made a preliminary study with other generative models in order to ensure none of them are already well suited to generate software dependency graphs. We want to answer the question: "Do existing generative models of directed graphs fit the topology of software graphs at the class granularity ?"

To answer this question, we searched in the literature for various digraphs generative models and generated graphs with the same number of nodes and/or edges numbers depending on the required model parameters. Parameters are configured using the parameter optimization approach presented on Section 6.3.2.2.

We execute the described procedure for some programs and notice same observations. We use the following generative models (the given parameters are optimized in order to have degree distributions as close as possible of the software applications – here values for Ant 1.9.2 are shown as an example):

- Erdös and Rényi [48] with parameter $p = 0.1$;
- Dorogvtsev et al. [45] with parameters $m = 4$, $A = 1$;
- Kumar et al. [83] with parameters $copyfactor = 0.2$, $d = 5$;
- Vazquez [154] with parameter p = 0.3,
- Grindrod [55] with parameters $\alpha = 0$, $\lambda = 0$;
- R-MAT of Chakrabarti et al. [37] with parameters $a = 0$, $b = 0.3$, $c = 0.6$, $d = 0.1$;
- GNC of Krapivsky and Redner [82];
- Baxter and Frean [21] with parameters $\gamma = 0.3$.

A visual representation of the generated graph degree distributions is shown on Figure C.1. Each line plots the inverse cumulative degree distribution for in- (C.1a) and out- (C.1b) degrees. On x-axis are the degrees and on y-axis are the cumulative frequency. The goal of this figure is to compare the shape of the synthetic distributions against the shape of empirical software data. We see that most models produce graphs whose degree distributions do not fit at all our data.

(a) in-degree (b) out-degree

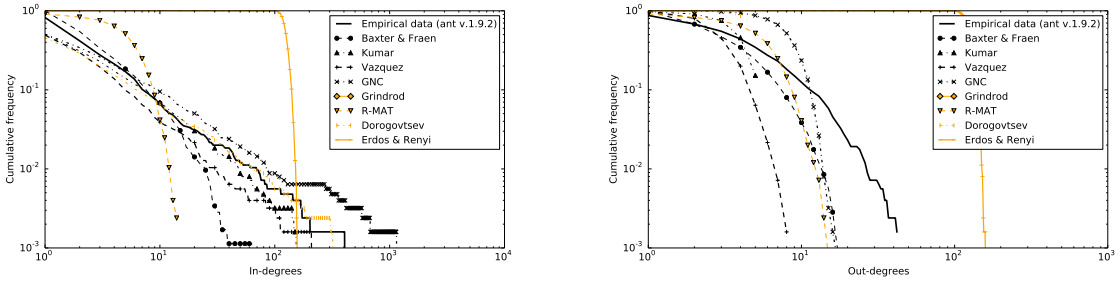


Figure C.1: Plot of the inverse cumulative degree distributions for ant 1.9.2 and for graphs generated using various models.

The model proposed by Baxter & Frean in 2008 [48] is the only one which is intended to generate software graphs. The overall fit of this model is good. However, some other models such as ones proposed by Dorogovtsev [45] or even by Kumar [83] can have a better fit than Baxter ones on in-degree.

The GNC model is different from others as it requires no external parameter at all. Its implementation is uniformly random and simple to understand and implement. Despite of its simplicity, it has a high ability to fit in-degrees distributions of our software graphs. The out-degree distribution is far from being perfect but compared to other models, and keeping in mind the lightweight of its logic, the fitness is not so bad.

To sum up, among the considered models; Baxter & Frean's one has the best fit of degree distributions with the software dependency graphs of our dataset. GNC is very simple, yet has a good fit for the in-degree distributions.

C.2 Different Versions

We also began to study another idea. The question we wanted to answer was "Is the GD-GNC model able to simulate the graph evolution that occurs between two versions of a same software application?"

To answer this question, we adapt our generation algorithm by passing as input a software class graph from an older release of the program. The model logic remains the same as previously, we pass to it the number of nodes contained in the last graph version so it will stop once the total number of nodes in the graph is equal to the expected number of nodes.

The adapted algorithm is executed on some programs and we observe an improvement for each new generated graph. As an example, Figure C.2 shows the inverse cumulative degree distribution from two generations for the Jtds application, one starting from scratch, and the other starting from the software dependency graph of a previous version of jtds (v.0.1). The x-axis is degrees and y-axis is the cumulative frequency. The dashed line shows when the generation starts from scratch and the solid thin line from a previous version of Jtds. In order to generate accurate graph, we optimize parameters as done in Section 6.3.2.2 before for each version (we notice parameters are globally the same for both versions of a same program).

We see that starting from a previous version improves the fit at the end of the generation. It means that our model does not break the existing topology but more importantly, it uses the existing topology to direct the creation and connection of new nodes. In other

(a) in-degree (b) out-degree

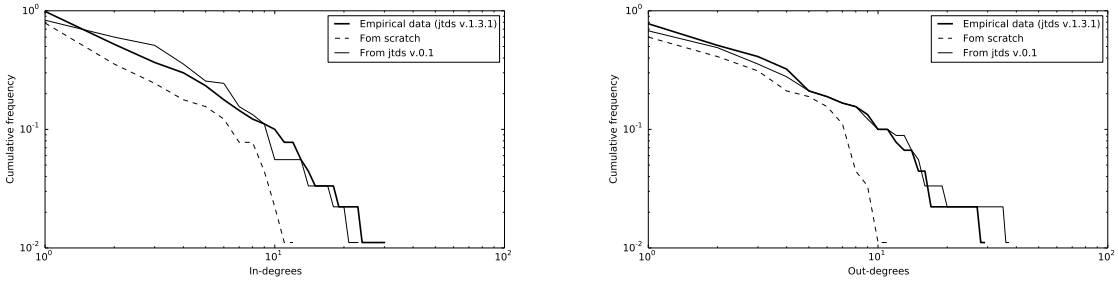


Figure C.2: Plot of the inverse cumulative in- and out- degree distribution for generated graphs using GD-GNC. These generations produce graphs having the same nodes number as Jtds v.1.3.1 at class granularity when starting the generation from scratch (dashed line) and from a previous version 0.1 (solid thin line)

words, from the point of view of degree distributions, the model simulates the evolution of jtds between v0.1 and v1.3.1.

To sum up, our model is able to simulate the evolution which occurs between two dependency graphs of two different versions of a software application. The synthesized graphs better fit when starting from a past real software graph instead of starting from scratch.

Bibliography

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. An Evaluation of Similarity Coefficients for Software Fault Localization. In *Proceedings of the Pacific Rim International Symposium on Dependable Computing*, pages 39–46, 2006.

[2] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. On the Accuracy of Spectrum-based Fault Localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 89–98, 2007.

[3] Allen T Acree, Timothy A Budd, Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Mutation Analysis. Technical report, DTIC Document, 1979.

[4] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault Localization using Execution Slices and Dataflow Tests. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 143–151, October 1995.

[5] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson, 2 edition, September 2006.

[6] Syed Nadeem Ahsan and Franz Wotawa. Impact Analysis of SCRs Using Single and Multi-label Machine Learning Classification. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 51:1–51:4, New York, NY, USA, 2010. ACM.

[7] Shaimaa Ali, James H. Andrews, Tamilselvi Dhandapani, and Wantao Wang. Evaluating the Accuracy of Fault Localization Techniques. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 76–87, Washington, DC, USA, 2009. IEEE Computer Society.

[8] Frances E. Allen. Control Flow Analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[9] J. H. Andrews, L. C. Briand, and Y. Labiche. Is Mutation an Appropriate Tool for Testing Experiments? In *Proceedings of the International Conference on Software Engineering*, pages 402–411, May 2005.

[10] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, and Andrea de Lucia. Identifying the Starting Impact Set of a Maintenance Request: A Case Study. In *Proceedings of the Conference on Software Maintenance and Reengineering*, CSMR '00, pages 227–, Washington, DC, USA, 2000. IEEE Computer Society.

[11] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who Should Fix This Bug? In *Proceedings of the 28th International Conference on Software Engineering*, ICSE'06, pages 361–370, New York, NY, USA, 2006. ACM.

[12] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and Precise Dynamic Impact Analysis Using Execute-after Sequences. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 432–441, New York, NY, USA, 2005. ACM.

[13] Robert S. Arnold and Shawn A. Bohner. Impact Analysis - Towards a Framework for Comparison. In *Proceedings of the Conference on Software Maintenance*, ICSM '93, pages 292–301, Washington, DC, USA, 1993. IEEE Computer Society.

[14] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January 2004.

[15] George K. Baah, Andy Podgurski, and Mary Jean Harrold. Causal Inference for Statistical Fault Localization. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 73–84, 2010.

[16] George K. Baah, Andy Podgurski, and Mary Jean Harrold. Mitigating the Confounding Effects of Program Dependences for Effective Fault Localization. In *Proceedings of the Symposium and the European Conference on Foundations of Software Engineering*, pages 146–156, 2011.

[17] Linda Badri, Mourad Badri, and Daniel St-Yves. Supporting Predictive Change Impact Analysis: A Control Call Graph Based Technique. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference*, APSEC '05, pages 167–175, Washington, DC, USA, 2005. IEEE Computer Society.

[18] Albert-László Barabási and Réka Albert. Emergence of Scaling in Random Networks. *Science*, 286(5439):509–512, October 1999.

[19] Ohad Barzilay. Example Embedding. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2011, pages 137–144, New York, NY, USA, 2011. ACM.

[20] Benoit Baudry, Franck Fleurey, and Yves Le Traon. Improving Test Suites for Efficient Fault Localization. In *Proceedings of the International Conference on Software Engineering*, pages 82–91, 2006.

[21] Gareth. J. Baxter and Marcus R. Frean. Software Graphs and Programmer Awareness. In *ArXiv e-prints*, volume 0802, page 2306, February 2008.

[22] Pamela Bhattacharya, Marios Iliofotou, Iulian Neamtiu, and Michalis Faloutsos. Graph-based Analysis and Prediction for Software Evolution. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 419–429, Piscataway, NJ, USA, 2012. IEEE Press.

[23] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and Shin Yoo. ORBS and the Limits of Static Slicing. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–10, September 2015.

[24] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. ORBS: Language-independent Program Slicing. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 109–120, New York, NY, USA, 2014. ACM.

[25] S.A. Bohner. Software Change Impacts - An Evolving Perspective. In *Proceedings of the International Conference on Software Maintenance*, ICSM '02, pages 263–272, 2002.

[26] Shawn A. Bohner and Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[27] Béla Bollobás. *Modern Graph Theory*, volume 184. Springer Science & Business Media, 1998.

[28] John Adrian Bondy and Uppaluri Siva Ramachandra Murty. *Graph Theory with Applications*, volume 290. Macmillan London, 1976.

[29] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 455–464, New York, NY, USA, 2010. ACM.

[30] Ben Breech, Mike Tegtmeyer, and Lori Pollock. Integrating Influence Mechanisms into Impact Analysis for Increased Precision. In *Proceedings of the 22Nd IEEE International Conference on Software Maintenance*, ICSM '06, pages 55–65, Washington, DC, USA, 2006. IEEE Computer Society.

[31] Marcel Bruch, Mira Mezini, and Martin Monperrus. Mining Subclassing Directives to Improve Framework Reuse. In *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, 2010.

[32] Timothy A Budd, Richard A DeMillo, Richard J Lipton, and Fred G Sayward. The Design of a Prototype Mutation System for Program Testing. In *Proceedings of the AFIPS National Computer Conference*, volume 74, pages 623–627, 1978.

[33] Timothy A Budd, Richard J Lipton, Richard A DeMillo, and Frederick G Sayward. Mutation Analysis. Technical report, Yale University, Department of Computer Science, 1979.

[34] Haipeng Cai, Siyuan Jiang, Raul Santelices, Ying-Jie Zhang, and Yiji Zhang. SENSA: Sensitivity Analysis for Quantitative Change-Impact Prediction. In *Proceedings of the 14th International Working Conference on Source Code Analysis and Manipulation*, SCAM '14, pages 165–174, Washington, DC, USA, 2014. IEEE Computer Society.

[35] D. Callahan. The Program Summary Graph and Flow-sensitive Interprocedual Data Flow Analysis. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 47–56, New York, NY, USA, 1988. ACM.

[36] T. Chaikalis and A. Chatzigeorgiou. Forecasting Java Software Evolution Trends Employing Network Models. *IEEE Transactions on Software Engineering*, 41(6):582–602, June 2015.

[37] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *Proceedings of the SIAM International Conference on Data Mining*, pages 442–446. Society for Industrial and Applied Mathematics, April 2004.

[38] Damien Challet and Andrea Lombardoni. Bug Propagation and Debugging in Asymmetric Software Structures. *Physical Review E*, 70(4):046109, October 2004.

[39] Jacob Cohen. *Statistical power analysis for the behavioral sciences (revised ed.)*. New York: Academic Press, 1977.

[40] A. De Lucia. Program Slicing: Methods and Applications. In *Proceedings of the 1st IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149, 2001.

[41] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 77–101, London, UK, UK, 1995. Springer-Verlag.

[42] Nicholas DiGiuseppe and James A. Jones. Semantic Fault Diagnosis: Automatic Natural-language Fault Descriptions. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, pages 23:1–23:4, 2012.

[43] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering*, 10(4):405–435, October 2005.

[44] Hyunsook Do and Gregg Rothermel. A Controlled Experiment Assessing Test Case Prioritization Techniques via Mutation Faults. In *Proceedings of the 21st International Conference on Software Maintenance*, ICSM '05, pages 411–420, Washington, DC, USA, 2005. IEEE Computer Society.

[45] Sergey N. Dorogovtsev, José F. F. Mendes, and A. N. Samukhin. Structure of Growing Networks with Preferential Linking. *Physical Review Letters*, 85(21):4633–4636, November 2000.

[46] Allen B. Downey. Lognormal and Pareto distributions in the Internet. *Computer Communications*, 28(7):790–801, May 2005.

[47] Karim O. Elish and Mahmoud O. Elish. Predicting Defect-prone Software Modules using Support Vector Machines. *Journal of Systems and Software*, 81(5):649–660, May 2008.

[48] Paul Erdős and Alfréd Rényi. On Random Graphs. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.

[49] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[50] Gordon Fraser and Andrea Arcuri. Sound Empirical Evidence in Software Testing. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 178–188. IEEE, 2012.

[51] Daniel M. German, Ahmed E. Hassan, and Gregorio Robles. Change Impact Graphs: Determining the Impact of Prior Code Changes. *Information and Software Technology*, 51(10):1394–1408, October 2009.

[52] Malcom Gethers, Bogdan Dit, Huzefa Kagdi, and Denys Poshyvanyk. Integrated Impact Analysis for Managing Software Changes. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 430–440, Piscataway, NJ, USA, 2012. IEEE Press.

[53] Martin Charles Golumbic. *Algorithmic Graph Theory and Perfect Graphs*, volume 57. Elsevier, 2004.

[54] A Gonzalez. *Automatic Error Detection Techniques based on Dynamic Invariants*. PhD thesis, MS thesis, Delft University of Technology, The Netherlands, 2007.

[55] Peter Grindrod. Range-dependent random graphs and their application to modeling large small-world Proteome datasets. *Physical Review E*, 66(6):066702, December 2002.

[56] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call Graph Construction in Object-oriented Languages. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 108–124, 1997.

[57] Frank Harary. *Graph Theory*. Addison-Wesley, Reading, MA, 1969.

[58] Mark Harman, David Binkley, Keith Gallagher, Nicolas Gold, and Jens Krinke. Dependence Clusters in Source Code. *ACM Transactions on Programming Languages and Systems*, 32(1):1:1–1:33, November 2009.

[59] Mary Jean Harrold and Brian Malloy. A Unified Interprocedural Program Representation for a Maintenance Environment. *IEEE Transactions on Software Engineering*, 19(6):584–593, June 1993.

[60] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. An Empirical Investigation of Program Spectra. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering*, pages 83–90, 1998.

[61] B. Hartmann, S. Doorley, and S. R. Klemmer. Hacking, Mashing, Gluing: Understanding Opportunistic Design. *IEEE Pervasive Computing*, 7(3):46–54, July 2008.

[62] Ahmed E Hassan and Richard C Holt. ADG: Annotated Dependency Graphs for Software Understanding. In *Proceedings of 2nd International Workshop on Visualizing Software For Understanding And Analysis*, VISSOFT '03, 2003.

[63] Lile Hattori, Gilson dos Santos Jr, Fernando Cardoso, and Marcus Sampaio. Mining Software Repositories for Software Change Impact Analysis: A Case Study. In *Proceedings of the 23rd Brazilian Symposium on Databases*, SBBD '08, pages 210–223, Porto Alegre, Brazil, Brazil, 2008. Sociedade Brasileira de Computação.

[64] Lile Hattori, Dalton Guerrero, Jorge Figueiredo, João Brunet, and Jemerson Damásio. On the Precision and Accuracy of Impact Analysis Techniques. In *Proceedings of the Seventh IEEE/ACIS International Conference on Computer and Information Science (Icis 2008)*, ICIS '08, pages 513–518, Washington, DC, USA, 2008. IEEE Computer Society.

[65] Matthew S Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Inc., 1977.

[66] John L. Hennessy. *Computer Architecture: A Quantitative Approach, 3rd Edition*. Morgan Kaufmann, San Francisco, CA, 3 edition edition, May 2002.

[67] Susan Horwitz and Thomas Reps. The Use of Program Dependence Graphs in Software Engineering. In *Proceedings of the 14th International Conference on Software Engineering*, ICSE '92, pages 392–411, New York, NY, USA, 1992. ACM.

[68] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[69] Lulu Huang and Yeong-Tae Song. Precise Dynamic Impact Analysis with Dependency Analysis for Object-oriented Programs. In *Proceedings of the 5th ACIS International Conference on Software Engineering Research, Management & Applications*, SERA '07, pages 374–384, Washington, DC, USA, 2007. IEEE Computer Society.

[70] Lulu Huang and Yeong-Tae Song. A Dynamic Impact Analysis Approach for Object-Oriented Programs. In *Advanced Software Engineering and Its Applications, 2008. ASEA 2008*, pages 217–220, December 2008.

[71] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. Improving Bug Triage with Bug Tossing Graphs. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 111–120, New York, NY, USA, 2009. ACM.

[72] Yue Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, September 2011.

[73] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the International Conference on Software Engineering*, pages 467–477, 2002.

[74] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are Mutants a Valid Substitute for Real Faults in Software Testing? In *Proceedings of the Symposium on the Foundations of Software Engineering*, 2014.

[75] Vigdis By Kampenes, Tore Dybå, Jo E. Hannay, and Dag I. K. Sjøberg. A Systematic Review of Effect Size in Software Engineering Experiments. *Information and Software Technology*, 49(11–12):1073–1086, November 2007.

[76] Malia Sofia Kilpinen. *The Emergence of Change at the Systems Engineering and Software Design Interface*. PhD thesis, University of Cambridge, 2008.

[77] Sunghun Kim, T. Zimmermann, and N. Nagappan. Crash Graphs: an Aggregated View of Multiple Crashes to Improve Crash Triage. In *Proceedings of the 41st International Conference on Dependable Systems Networks*, DSN '11, pages 486–493, June 2011.

[78] Sunwoo Kim, John A Clark, and John A McDermid. Class mutation: Mutation testing for object-oriented programs. In *Proc. Net. ObjectDays*, pages 9–12, 2000.

[79] K. N. King and A. Jefferson Offutt. A Fortran Language System for Mutation-based Software Testing. *Software: Practice and Experience*, 21(7):685–718, July 1991.

[80] Andrew J. Ko and Brad A. Myers. Debugging Reinvented: Asking and Answering Why and Why Not Questions About Program Behavior. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 301–310, New York, NY, USA, 2008. ACM.

[81] R. Kohavi. A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection. *International joint conference on artificial intelligence, 1995*, pages 1137–1143, 1995.

[82] Pavel L. Krapivsky and Sidney Redner. Network Growth by Copying. *Physical Review E*, 71(3):036118, March 2005.

[83] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, D. Sivakumar, Andrew Tomkins, and Eli Upfal. Stochastic Models for the Web Graph. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS '00, pages 57–65, 2000.

[84] William Landi and Barbara G. Ryder. Pointer-induced Aliasing: A Problem Classification. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 93–103, New York, NY, USA, 1991. ACM.

[85] James Law and Gregg Rothermel. Whole Program Path-Based Dynamic Impact Analysis. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 308–318, Washington, DC, USA, 2003. IEEE Computer Society.

[86] Steffen Lehnert. A Review of Software Change Impact Analysis. *Ilmenau University of Technology, Tech. Rep*, 2011.

[87] Steffen Lehnert. A Taxonomy for Software Change Impact Analysis. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, IWPSE-EVOL '11, pages 41–50, New York, NY, USA, 2011. ACM.

[88] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. A Survey of Code-based Change Impact Analysis Techniques. *Software Testing, Verification and Reliability*, 23(8):613–646, December 2013.

[89] Hui Li, Hai Zhao, Wei Cai, Jiu-Qiang Xu, and Jun Ai. A modular attachment mechanism for software network evolution. *Physica A: Statistical Mechanics and its Applications*, 392(9):2025–2037, May 2013.

[90] L. Li and A.J. Offutt. Algorithmic Analysis of the Impact of Changes to Object-Oriented Software. In *Proceedings of the International Conference on Software Maintenance 1996*, ICSM '06, pages 171–184, November 1996.

[91] Lun Li, David Alderson, John C. Doyle, and Walter Willinger. Towards a Theory of Scale-Free Graphs: Definition, Properties, and Implications. *Internet Mathematics*, 2(4):431–523, January 2005.

[92] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable Statistical Bug Isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 15–26, New York, NY, USA, 2005. ACM.

[93] Zhongpeng Lin and Jim Whitehead. Why Power Laws?: An Explanation from Fine-grained Code Changes. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 68–75, Piscataway, NJ, USA, 2015. IEEE Press.

[94] C. Liu, X. Yan, H. Yu, J. Han, and P. Yu. Mining Behavior Graphs for "Backtrace" of Noncrashing Bugs. In *Proceedings of the 2005 SIAM International Conference on Data Mining*, Proceedings, pages 286–297. Society for Industrial and Applied Mathematics, April 2005.

[95] Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos. Power Laws in Software. *ACM Transactions on Software Engineering and Methodology*, 18(1):2:1–2:26, October 2008.

[96] Joseph P. Loyall and Susan A. Mathisen. Using Dependence Analysis to Support the Software Maintenance Process. In *Proceedings of the Conference on Software Maintenance*, ICSM '93, pages 282–291, Washington, DC, USA, 1993. IEEE Computer Society.

[97] J.P. Loyall, S.A. Mathisen, P.J. Hurley, and J.S. Williamson. Automated maintenance of avionics software. In *Aerospace and Electronics Conference, 1993. NAECON 1993., Proceedings of the IEEE 1993 National*, pages 508–514 vol.1, May 1993.

[98] J.P. Loyall, S.A. Mathisen, and C.P. Satterthwaite. Impact analysis and change management for avionics software. In *Aerospace and Electronics Conference, 1997. NAECON 1997., Proceedings of the IEEE 1997 National*, volume 2, pages 740–747 vol.2, July 1997.

[99] A Luqi. A Graph Model for Software Evolution. *IEEE Transactions on Software Engineering*, 16(8):917–927, 1990.

[100] Yu-Seung Ma, Yong-Rae Kwon, and J. Offutt. Inter-class mutation operators for Java. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 352–363, 2002.

[101] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. MuJava: A Mutation System for Java. In *Proceedings of the International Conference on Software Engineering*, pages 827–830, New York, NY, USA, 2006. ACM.

[102] Chris Maddison and Daniel Tarlow. Structured Generative Models of Natural Source Code. In Tony Jebara and Eric P. Xing, editors, *Proceedings of the 31st International Conference on Machine Learning*, ICML '14, pages 649–657. JMLR Workshop and Conference Proceedings, 2014.

[103] M.C.O. Maia, R.A. Bittencourt, J.C.A. De Figueiredo, and D.D.S. Guerrero. The Hybrid Technique for Object-Oriented Software Change Impact Analysis. In *2010 14th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 252–255, March 2010.

[104] José Carlos Maldonado, Márcio Eduardo Delamaro, Sandra C. P. F. Fabbri, Adenilso da Silva Simão, Tatiana Sugeta, Auri Marcelo Rizzo Vincenzi, and Paulo Cesar Masiero. Proteum: A Family of Tools to Support Specification and Program Testing Based on Mutation. In W. Eric Wong, editor, *Mutation Testing for the New Century*, number 24 in The Springer International Series on Advances in Database Systems, pages 113–116. Springer US, 2001. DOI: 10.1007/978-1-4757-5939-6_19.

[105] Christoph C. Michael and Ryan C. Jones. On the Uniformity of Error Propagation in Software. In *Proceedings of the 12th Annual Conference on Computer Assurance*, COMPASS'97, pages 68–76, June 1997.

[106] Brian S. Mitchell and Spiros Mancoridis. On the Automatic Modularization of Software Systems using the Bunch Tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, March 2006.

[107] Michael Mitzenmacher. A Brief History of Generative Models for Power Law and Lognormal Distributions. *Internet Mathematics*, 1(2):226–251, January 2004.

[108] Mark Moriconi and Timothy C. Winkler. Approximate Reasoning About the Semantic Effects of Program Changes. *IEEE Transactions on Software Engineering*, 16(9):980–992, September 1990.

[109] R. E. Mullen and S. S. Gokhale. Software defect rediscoveries: a discrete lognormal model. In *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, pages 10 pp.–212, November 2005.

[110] Robert E. Mullen and Swapna S. Gokhale. A Discrete Lognormal Model for Software Defects Affecting Quality of Protection. In Dieter Gollmann, Fabio Massacci, and Artsiom Yautsiukhin, editors, *Quality of Protection*, number 23 in Advances in Information Security, pages 37–47. Springer US, 2006. DOI: 10.1007/978-0-387-36584-8_4.

[111] Seokhyeon Mun, Yunho Kim, Moonzoo Kim, and Shin Yoo. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *Proceedings of the International Conference on Software Testing, Verification, and Validation*, pages 153–162, 2014.

[112] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. An Empirical Study of Static Call Graph Extractors. *ACM Transactions on Software Engineering and Methodology*, 7(2):158–191, April 1998.

[113] Vincenzo Musco, Martin Monperrus, and Philippe Preux. A Generative Model of Software Dependency Graphs to Better Understand Software Evolution. *ArXiv e-prints, 1410.7921v2*, October 2014.

[114] Christopher R. Myers. Software Systems as Complex Networks: Structure, Function, and Evolvability of Software Collaboration Graphs. *Physical Review E*, 68(4):046116, October 2003.

[115] Glenford J Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. John Wiley & Sons, 2011.

[116] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A Model for Spectra-based Software Diagnosis. *ACM Transactions on Software Engineering and Methodology*, 20(3):11:1–11:32, August 2011.

[117] Akbar Siami Namin and Sahitya Kakarla. The Use of Mutation in Testing Experiments and Its Sensitivity to External Threats. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 342–352, New York, NY, USA, 2011. ACM.

[118] M. E. J. Newman. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics*, 46(5):323–351, September 2005.

[119] Mark Newman. The Structure and Function of Complex Networks. *SIAM Review*, 45(2):167–256, January 2003.

[120] Mark Newman. *Networks: An Introduction*. Oxford University Press, Inc., New York, NY, USA, 2010.

[121] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, and Tien N Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *Proceedings of the 34th International Conference on Software Engineering*, pages 69–79. IEEE Press, 2012.

[122] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An Experimental Determination of Sufficient Mutant Operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, April 1996.

[123] Keziban Orman, Vincent Labatut, and Hocine Cherifi. An Empirical Study of the Relation between Community Structure and Transitivity. In Ronaldo Menezes, Alexandre Evsukoff, and Marta C. González, editors, *Complex Networks*, number 424 in Studies in Computational Intelligence, pages 99–110. Springer Berlin Heidelberg, 2013. DOI: 10.1007/978-3-642-30287-9_11.

[124] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging Field Data for Impact Analysis and Regression Testing. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, pages 128–137, New York, NY, USA, 2003. ACM.

[125] David A. Padua, David J. Kuck, and Duncan H. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Transactions on Computers*, (9):763–776, 1980.

[126] Mike Papadakis and Yves Le Traon. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability*, 25(5-7):605–628, August 2015.

[127] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, page na, 2015.

[128] M. Petrenko and V. Rajlich. Variable Granularity for Improving Precision of Impact Analysis. In *IEEE 17th International Conference on Program Comprehension, 2009. ICPC '09*, pages 10–19, May 2009.

[129] A. Podgurski and L. A. Clarke. A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, September 1990.

[130] Alex Potanin, James Noble, Marcus Frean, and Robert Biddle. Scale-free Geometry in OO Programs. *Communications of the ACM - Adaptive complex enterprises*, 48(5):99–103, May 2005.

[131] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Sieve: A Tool for Automatically Detecting Variations Across Program Versions. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 241–252. IEEE Computer Society, 2006.

[132] Xiaoxia Ren and Barbara G. Ryder. Heuristic Ranking of Java Program Edits for Fault Localization. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 239–249, New York, NY, USA, 2007. ACM.

[133] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: A Tool for Change Impact Analysis of Java Programs. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 432–448, New York, NY, USA, 2004. ACM.

[134] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G Ryder, Ophelia Chesley, and Julian Dolby. Chianti: A Prototype Change Impact Analysis Tool for Java. *Rutgers University, Department of Computer Science, Tech. Rep. DCS-TR-533*, 2003.

[135] M. Renieres and S. P. Reiss. Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings*, pages 30–39, October 2003.

[136] Martin P. Robillard and Gail C. Murphy. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 406–416, New York, NY, USA, 2002. ACM.

[137] Barbara G. Ryder and Frank Tip. Change Impact Analysis for Object-oriented Programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 46–53, New York, NY, USA, 2001. ACM.

[138] B.G. Ryder. Constructing the Call Graph of a Program. *IEEE Transactions on Software Engineering*, SE-5(3):216–226, May 1979.

[139] Raul Santelices, James A. Jones, Yanbing Yu, and Mary Jean Harrold. Lightweight Fault-localization Using Multiple Coverage Types. In *Proceedings of the International Conference on Software Engineering*, pages 56–66, 2009.

[140] David Schuler and Andreas Zeller. Javalanche: Efficient Mutation Testing for Java. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 297–298, New York, NY, USA, 2009. ACM.

[141] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. Programmers' Build Errors: A Case Study (at Google). In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, pages 724–734, New York, NY, USA, 2014. ACM.

[142] Gang Shu, Boya Sun, T.A.D. Henderson, and A. Podgurski. JavaPDG: A New Platform for Program Dependence Analysis. In *Proceedings of the 6th International Conference on Software Testing, Verification and Validation*, ICST'13, pages 408–415, March 2013.

[143] Gang Shu, Boya Sun, Andy Podgurski, and Feng Cao. MFL: Method-Level Fault Localization with Causal Inference. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 124–133, 2013.

[144] F. Steimann and M. Frenkel. Improving Coverage-Based Localization of Multiple Faults Using Algorithms from Integer Linear Programming. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pages 121–130, November 2012.

[145] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. Threats to the Validity and Value of Empirical Assessments of the Accuracy of Coverage-based Fault Locators. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 314–324, 2013.

[146] Ma łgorzata Steinder and Adarshpal S. Sethi. A survey of fault localization techniques in computer networks. *Science of Computer Programming*, 53(2):165–194, November 2004.

[147] Maximilian Stoerzer, Barbara G. Ryder, Xiaoxia Ren, and Frank Tip. Finding Failure-inducing Changes in Java Programs Using Change Classification. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 57–68, New York, NY, USA, 2006. ACM.

[148] Xiaobing Sun, Bixin Li, Hareton Leung, Bin Li, and Junwu Zhu. Static Change Impact Analysis Techniques: A Comparative Study. *Journal of Systems and Software*, 109:137 – 149, 2015.

[149] Liang Tian and Afzel Noore. Evolutionary Neural Network Modeling for Software Cumulative Failure Time Prediction. *Reliability Engineering & System Safety*, 87(1):45–51, January 2005.

[150] Frank Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[151] Frank Tip and Jens Palsberg. Scalable Propagation-based Call Graph Construction Algorithms. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 281–293, New York, NY, USA, 2000. ACM.

[152] Sergi Valverde and Ricard V. Solé. Logarithmic Growth Dynamics in Software Networks. *EPL (Europhysics Letters)*, 72(5):858, December 2005.

[153] Sergi Valverde and Ricard V. Solé. Network Motifs in Computational Graphs: A case Study in Software Architecture. *Physical Review E*, 72(2):026107, August 2005.

[154] Alexei Vazquez. Knowing a network by walking on it: emergence of scaling. *ArXiv e-prints*, page 6132, June 2000.

[155] Robert J. Walker, Reid Holmes, Ian Hedgeland, Puneet Kapur, and Andrew Smith. A Lightweight Approach to Technical Risk Estimation via Probabilistic Impact Analysis. In *Proceedings of the International Workshop on Mining Software Repositories*, MSR '06, pages 98–104, New York, NY, USA, 2006. ACM.

[156] Duncan J. Watts and Steven H. Strogatz. Collective Dynamics of 'Small-world' Networks. *Nature*, 393(6684):440–442, June 1998.

[157] Douglas Brent West. *Introduction to Graph Theory*, volume 2. Prentice hall Upper Saddle River, 2001.

[158] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2016.

[159] Tao Xie and David Notkin. An empirical study of Java dynamic call graph extractors. 2002.

[160] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-based Fault Localization. *ACM Transactions on Software Engineering and Methodology*, 22(4):31:1–31:40, October 2013.

[161] Jian Xu, Zhenyu Zhang, W. K. Chan, T. H. Tse, and Shanping Li. A general noise-reduction framework for fault localization of Java programs. *Information and Software Technology*, 55(5):880–896, May 2013.

[162] J. Xuan and M. Monperrus. Learning to Combine Multiple Ranking Metrics for Fault Localization. In *Proceedings of the International Conference on Software Maintenance and Evolution*, pages 191–200, September 2014.

[163] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. Localizing Failure-inducing Program Edits Based on Spectrum Information. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 23–32, Washington, DC, USA, 2011. IEEE Computer Society.

[164] Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. Injecting Mechanical Faults to Localize Developer Faults for Evolving Software. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages & Applications*, pages 765–784, 2013.

[165] Lei Zhao, Lina Wang, Zuoting Xiong, and Dongming Gao. Execution-Aware Fault Localization Based on the Control Flow Analysis. In Rongbo Zhu, Yanchun Zhang, Baoxiang Liu, and Chunfeng Liu, editors, *Information Computing and Applications*, number 6377 in Lecture Notes in Computer Science, pages 158–165. Springer Berlin Heidelberg, October 2010. DOI: 10.1007/978-3-642-16167-4_21.

[166] Lei Zhao, Lina Wang, and Xiaodan Yin. Context-aware fault localization via control flow analysis. *Journal of Software*, 6(10):1977–1984, 2011.

[167] Thomas Zimmermann and Nachiappan Nagappan. Predicting Defects Using Network Analysis on Dependency Graphs. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 531–540, New York, NY, USA, 2008. ACM.