DigitalCommons
@WayneState

Wayne State University

Wayne State University Dissertations

January 2019

Integrating Heuristics To Support Impact Analysis In Software Evolution

Yibin Wang
Wayne State University

Follow this and additional works at: https://digitalcommons.wayne.edu/oa_dissertations

Part of the Computer Sciences Commons

Recommended Citation

Wang, Yibin, "Integrating Heuristics To Support Impact Analysis In Software Evolution" (2019). *Wayne State University Dissertations*. 2312.
https://digitalcommons.wayne.edu/oa_dissertations/2312

This Open Access Dissertation is brought to you for free and open access by DigitalCommons@WayneState. It has been accepted for inclusion in Wayne State University Dissertations by an authorized administrator of DigitalCommons@WayneState.

INTEGRATING HEURISTICS TO SUPPORT IMPACT ANALYSIS IN SOFTWARE EVOLUTION

by

YIBIN WANG

DISSERTATION

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

2019

MAJOR: COMPUTER SCIENCE

Approved by:

____

Advisor Date

____

____

____

____

DEDICATION

To my family and friends.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Dr. Václav Rajlich, for his infectious enthusiasm for my research. This thesis would not be possible without a continuous guidance and support of him. His mentoring was invaluable for me while learning the scientific inquiry and principles of software engineering. He not only kept a close eye on me and my work to ensure its quality, but also taught me the characteristics of great researchers.

Also, I thank Dr. Marwan Abi-Antoun for his feedback and instructions on some projects. His expertise in program comprehension and static code analysis was particularly valuable, and his feedback facilitated the development of my research.

I appreciate all the help and feedback provided by the alumni of SEVERE software engineering group. Particularly, I thank Dr. Mohammad Ebrahim Khalaj, Dr. Radu Vanciu and Andrew Giang for their input and collaboration in our research projects. I appreciate Dr. Maksym Petrenko who worked on the previous generations of JRipples.

I am grateful to Dr. Dae-Kyoo Kim, Dr. Alexander Kotov and Dr. Dongxiao Zhu for serving on my dissertation committee.

I am thankful to the Department of Computer Science, Dr. Václav Rajlich, and Dr. Marwan Abi-Antoun for providing financial support during my studies at Wayne State University.

TABLE OF CONTENTS

ACKNOWLEDGMENTS iii

LIST OF TABLES vi

LIST OF FIGURES vii

Chapter 1 Introduction 1

1.1 Motivation and Hypotheses 1

1.2 Contribution 2

1.3 Dissertation Organization 3

1.4 Bibliographical Notes 3

Chapter 2 Background and Related Work 5

2.1 Impact Analysis Taxonomy and Techniques 6

2.2 Iterative IA Process and All-at-once IA Process 10

2.3 Class and Member Dependency Graph and its Application 13

2.4 Ownership Object Graph (OOG) and its Application 13

2.4.1 OOG Applications 14

2.5 Heuristics of Impact Analysis 15

2.5.1 Categorization based on information sources 15

2.5.2 Categorization based on the role in the IA process 16

Chapter 3 Iterative Impact Analysis based on a Global Hierarchical Object Graph 18

3.1 Evaluation Method 18

3.1.1 Environment 18

3.1.2 Procedures 19

3.1.3 Subject System and Tasks 22

3.1.4 Hypothesis and Measures 22

3.2 Evaluation 26

3.3 Discussion....28

Chapter 4 Evaluating Heuristics for Iterative Impact Analysis based on a Weighted Class Propagation Graph....30

4.1 Weighted Class Propagation Graph....30

4.2 Candidate Heuristics....31

4.2.1 Propagation Heuristics....31

4.2.2 Termination Heuristics....33

4.3 Design of Case Study....34

4.3.1 IIA Reenactment Process Overview....34

4.3.2 Simulation Algorithm....36

4.3.3 Simulation Example in MiniDraw....39

4.3.4 Subject Systems....42

4.3.5 Measures....43

4.4 Evaluation....43

4.4.1 Discussion of results....45

4.4.2 Threats to Validity....50

Chapter 5 Conclusions....52

5.1 Future Work....53

Reference....54

ABSTRACT....62

AUTOBIOGRAPHICAL STATEMENT....64

LIST OF TABLES

Table 2-1. Marks In Iterative IA 11

Table 3-1. Marks In ArchSummary 20

Table 3-2. Mapping of Marks between ArchSummary and JRipples 22

Table 3-3. Change Requests of Subject Systems 23

Table 3-4. Comparative Results 27

Table 4-1. Subject Systems 42

Table 4-2. Average Recall of Investigated Heuristics (%) 44

Table 4-3. Standard Deviation of Recall (%) 45

Table 4-4. Average Precision of Investigated Heuristics (%) 46

Table 4-5. Standard Deviation of Precision (%) 47

LIST OF FIGURES

Figure 2-1. Process Model of Software Change 5

Figure 2-2. An IIA instance 12

Figure 3-1. IIA in ArchSummary 19

Figure 3-2. IIA in JRipples 21

Figure 4-1. Overview of the IIA reenactment 35

Figure 4-2. Finding reachable nodes and edges in WCPG based on the IIC and the termination heuristic Top2 37

Figure 4-3. Finding Steiner nodes example. Left part is the directed graph, and the right part is the directed Steiner tree accordingly. 38

Figure 4-4. Partial WCPG of MiniDraw based on DBH heuristic 40

Figure 4-5. Reachable classes and edges based on Top2 for the class GameStub 41

Figure 4-6. Directed Steiner tree to connect GameStub and BoardDrawing 41

Figure 4-7. Visited Set in the reenactment. Nodes in yellow estimates 'Impacted' or 'Propagating' classes, and nodes in green estimates 'Unchanged' classes. 42

Figure 4-8. Average precision (left) and recall (right) of investigated heuristics for JEdit 48

Figure 4-9. Average precision (left) and recall (right) of investigated heuristics for Jhotdraw 48

Figure 4-10. Average precision (left) and recall (right) of investigated heuristics for Quickfix/J 49

Figure 4-11. Average overall precision (left) and recall (right) 49

Figure 4-12. Median precision (left) and recall (right) of investigated heuristics for JEdit 49

Figure 4-13. Median precision (left) and recall (right) of investigated heuristics for Jhotdraw 50

Figure 4-14. Median precision (left) and recall (right) of investigated heuristics for Quickfix/J 50

Chapter 1 Introduction

Software evolution and maintenance is an everlasting topic for software engineers and researchers. During software evolution, programmers continuously make software changes. Impact analysis (IA) is a designing phase for a software change task where programmers plan the units that should be modified in the change [1]. The widely accepted definition of IA is "identifying the potential consequences of a change" [2], and the common measures of IA are precision and recall [1].

Iterative impact analysis (IIA) is a process that allows developers to detect impacted units (e.g. statements, methods, classes) step by step following program dependencies in the program representation of a software system. It starts with an initial impacted unit that is scheduled to be modified; this unit can be identified during a preceding phase named concept location [3]. The programmers inspect other units that interact with the initial impacted unit and determine whether these units are impacted by the change also. This process continues iteratively and has been called ripple effect [4].

All units inspected by the programmers during IIA constitute the *visited set (VS)*, the units that programmers predict to be modified form the *estimated impact set (EIS)*, and the *actual impact set (AIS)* consists of all units that are modified in the real implementation. A few IIA techniques have been investigated in the past [5-8].

Some researchers proposed IA techniques that predict the EIS in a single algorithmic step [9-12]. We call this process *all-at-once impact analysis (AIA)*. Compared to AIA, programmers are "in the loop" during IIA and are expected to correct imperfections that an AIA algorithm accumulates.

1.1 Motivation and Hypotheses

The drawback of IIA techniques is that for a specific change in a unit, not all its interacting units are impacted. Thus, programmers may inspect many irrelevant units and

have unnecessary workload for IA tasks. To mitigate this problem, we explore two approaches.

The first approach is to put forward new program representations that provide more precise dependencies for software change propagation. Our hypothesis for this approach is that the precision of IIA can be improved using such a program representation while developers can still achieve good recall.

The second approach is integrating other IA techniques into a program representation that may assist programmers to make correct decision for which units to inspect during IIA. We treat those additional IA techniques as heuristics and distinguish two different types. One type is *propagation heuristics* that guide the developers towards the units that are likely to be impacted by the change. They play a role in the situation where there are many interacting units for the programmers to inspect. They guide programmers towards the units that are most likely to change. The other type of heuristics is *termination heuristics* that indicate the EIS is complete. The roles of these two kinds of heuristics are complementary and affect both the precision and the recall for the assisted IIA technique. For this approach, we investigate several propagation heuristics adapted from previously published papers and combine them with a practical termination heuristic during IIA that uses a static dependency graph as the program representation.

1.2 Contribution

Our work has the following contributions:

(1) Evaluating a new program representation in the context of IIA and comparing it to an existing IIA technique. Several unique measures have been designed for further comparisons in this circumstance.

(2) Evaluating the performance of a static dependency graph used for IIA enriched by various propagation heuristics along and a practical termination heuristic at the granularity

of classes. Some of the propagation heuristics have been investigated within the context of AIA techniques [9, 11, 13] and we adapted them for IIA.

(3) In order to compare the effectiveness of these heuristics, we develop an empirical approach called reenactment that simulates the actions of developers who are guided by heuristics during IIA. This reenactment is applied to the past changes of open source projects mined from software repositories.

1.3 Dissertation Organization

The rest of the dissertation is organized as follows. Chapter 2 provides the background of current impact analysis techniques, especially two different program representations (i.e. the Class and Member Dependency Graph and the Ownership Object Graph) that can support iterative IA process. Chapter 3 studies the difference of completing IA tasks following dependencies extracted from these two program representations. Chapter 4 investigates the effectiveness of selected heuristics in assisting IIA; this part also provides an empirical study. Finally, Chapter 5 concludes the main findings in our research.

1.4 Bibliographical Notes

Some of the materials that were produced in collaboration with other researchers and/or were published. Chapter 2.4 and Chapter 3 are based on [8], coauthored with Dr. Marwan Abi-Antoun (leading author), Dr. Ebrahim Khalaj, Andrew Giang and Dr. Václav Rajlich. We contribute to unify the IIA process for each tool, propose the measures, instrument the automated logging of human-tool interactions, perform the subject tasks and evaluate the results. More details are depicted in Chapter 3. Chapter 4 is motivated by the prior work described in [14]; portions of this work were based on a tool developed by Dr. Maksym Petrenko. We re-evaluate three propagation heuristics mentioned in [14] along with two new ones based on *Mining Software Repositories (MSR)* techniques. Moreover, we propose a novel IIA reenactment approach to simulate the actions of developers in real

software change IA tasks after combing each propagation heuristic and a practical termination heuristic.

Chapter 2 Background and Related Work

Phased Model of Software Change (PMSC) [15, 16] describes a process of adding new functionality to the existing software systems in terms of phases, as shown in Figure 2-1.

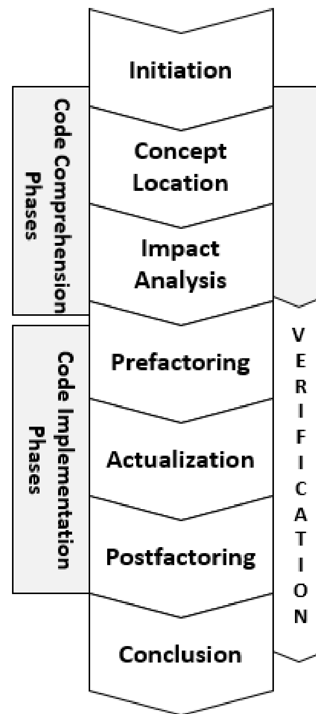

Figure 2-1. Process Model of Software Change

In initiation phase, a certain change request is formed, selected and delivered to the developers. The next two phases concept location (aka. feature location) and impact analysis contribute to the design and comprehension of the change. Concept location finds the initial code units to change, whereas impact analysis estimates the total impact of the change within the source code and/or other related artifacts like documentation and test cases. Such estimation can help developers decide how the change should be implemented in a proper way and avoid expensive late rework. The result of concept location is called *initial impact set (IIS)* and can be treated as the input for impact analysis. The result of Impact Analysis is the estimated impact set. Developers implement the real change in the phase actualization where change propagation is conducted to resolve the

remained inconsistencies of the program introduced by the change. In fact, change propagation is very similar and interchangeable to impact analysis in the literature [14]. Any technique of impact analysis can be adopted by change propagation and vice versa. The units changed in actualization construct the actual impact set. The estimated impact set may contain units that are not modified finally; such units are false positives of IA. It is also possible that the estimated impact set misses some units of the actual impact set, which are false negatives of IA. Prefactoring and postfactoring are refactoring of code with different purposes. Prefactoring aims at facilitating actualization whereas postfactoring cleans up the code of the actualization. During the conclusion phase, the code of the change request is merged to the repository, the paper work is done and the development team is ready for new change request by repeating PMSC. Through all phases that the code is actually changed, the verification such as unit testing and functional testing is performed to ensure the quality.

2.1 Impact Analysis Taxonomy and Techniques

An early classification of impact analysis techniques was discussed by Arnold and Bohner [2]. They also put forward two measures for evaluating IA techniques: precision and recall. Such measurements have been widely used in the literature since then. Lehnert [17] enriched that work of Arnold and Bohner and developed a taxonomy for classifying impact analysis techniques. The new classification criteria included the scope of analysis, the granularity of units, utilized techniques, the style of analysis, tool support, supported languages, scalability and experimental results. Li et al. [18] focused on code-based IA techniques in the literature and categorized them by seven properties, which are object, impact set, type of analysis, intermediate representation, language support, tool support and empirical evaluation.

Early impact analysis techniques include Program Slicing, Call Graphs, Execution Traces, Program Dependence Graphs, Message Dependence Graphs, Traceability, Explicit Rules, Information Retrieval, Probabilistic Models and History Mining, which have been overviewed in [17]. We provide an overview of the more recent works as follows, and most of them integrated different IA techniques to enhance the overall performance.

Aryani et al. [12, 19-23] discussed about using domain concepts to approximate dependencies among software components, especially when the source code is not available. First, they manually assign some domain concepts (namely, domain variables) for each software component based on the comprehension of its functionality. Then software components can have dependencies to each other if they share some common domain variables. These dependencies are also weighted according to how many affected variables from the given IIS are contained. Thus, programmers can compute the EIS based on that. They tested the performance by different thresholds at the granularity of program components.

Cai et al. [24, 25] combined sensitivity analysis and execution differencing to rank code dependencies found by static program slicing. Execution differencing compares the differences in a number of executions of a program by only changing the value at a certain statement ST to find out which statements are really impacted by ST. Thus, in their research, large quantity of test suites were required as the partial input of IA. The result showed their technique was more precise than static program slicing.

Kagdi et al. [9, 11] investigated the intersection and union of results from two IA techniques, which are the similarity among units using information retrieval and the association rules among units extracted from the repository. They set different cut points to get the result of each IA technique before combining them.

Gethers et al. [10, 26, 27] enriched the work of [9, 11] by combining one more IA technique based on execution trace with the previous two IA techniques.

Zanjani et al.[28] proposed an approach, InComIA, to integrate information retrieval, machine learning, and lightweight source code analysis techniques into IA based on the resources of developer interaction histories (e.g., Mylyn) and commit histories (e.g., SVN).

Li et al. [29, 30] used the intersection of results from concept lattice and call graph, respectively, as a new IA approach. They computed the temporary EIS based on concept lattice at the granularity of classes first, then they used the call graph to reduce the false positives in the temporary EIS to generate the final EIS.

In [31], Li et al. revised the traditional call graph IA technique. They considered that the interference among multiple methods of the IIS may improve the precision of the prediction, that is, a method that is within a certain distance in the call graph to all methods of the IIS has a higher chance to be impacted. Thus, instead of using the transitive closure by following call relations in the traditional technique using call graphs, their approach generated a smaller EIS for a given IIS.

Abi-Antoun et al. [8] estimated the classes that may be instantiated at runtime by static analysis to construct an Ownership Object Graph (OOG). Then they extracted class dependences for change propagation from OOG and used propagation heuristics that took both ownership information and the number of certain kinds of edges in OOG into account to refine and rank the those dependencies. As a result, programmers could follow those ranked dependencies to explore the EIS of a change request.

Cai and Santelices [32] put forward a new kind of static program dependency graph at granularity of methods, namely Method Dependence Graph (MDG). MDG defines two kinds of edges among methods: (1) A method m’ is data dependent on a method m if m defines a variable that m’ might use or if m returns a value to m’, and the direction of the dependency is from m to m’. (2) A method m’ is control dependent on a method m if a decision in m determines whether m’ (or part of it) executes, and the direction is from m to

m'. They used the transitive closure of a given IIS in MDG via following the edges to predict the EIS, though their original idea is to mimic forward program slicing with less cost.

Cai et al. [33, 34] discussed an impact analysis framework at the granularity of methods, DIAPRO, and compared it to an early dynamic technique PI/EAS of Apiwattanapong et al. [35]. The work studied the combination of static dependencies and multiple forms of dynamic data including method-execution events, statement coverage, and dynamic points-to sets to fill the gap between two extreme conditions for current IA techniques: either fast, but too imprecise, or more precise, yet overly expensive.

Borg et al. [36] provided an industrial study about a tool, ImpRec, which can predict impacted artifacts if the developers input the description of change requests (i.e. issue reports). ImpRec used two existing IA techniques: the information retrieval method to find similar past issue reports (aka. issue duplicate detection) and the Mining Software Repositories (MSR) method to decide impacted non-code artifacts according to the AIS of those similar issue reports. ImpRec was evaluated in a two-phase industrial case study, and the results showed that it could present about 40% of the AIS among the top-10 recommendations for the studied project.

Musco et al. [37, 38] used machine learning by mutation testing to rank the dependencies in the static call graph to predict the impact in the harness code. The artifacts were methods in the object-oriented software source code. The change under study was a change in the code of a method *M* in the production code, and the impacted units are the test methods that failed because of the change in *M*. They run mutation testing with five mutation operators and a number of test cases on *M* to train the weights of the test methods that have call relations to *M*. Also, they put forward two different algorithms in the computation of the weights for ranking and investigated the best threshold. The result showed such technique had better precision and recall than using transitive closures of the call graph for IA in harness code.

2.2 Iterative IA Process and All-at-once IA Process

IA process starts when the IIS is determined and ends when the full EIS is found. The IIS can be a set of units from concept location (e.g., [10, 26, 27]) or the change request itself (e.g., [8, 36]).

Some IA techniques build graph representations for the software systems such as the Class and Member Dependency Graph [14] where nodes of the graph are program units at certain granularities and edges are dependencies among these units that may propagate changes. The EIS by those techniques is usually supposed to be in the part of the transitive closure of the IIS in the graph. However, due to the complexity of software system, the whole transitive closure of a single unit in the dependency graph may consist of a large number of irrelevant units for the change in that unit. To alleviate this, some IA approaches allow programmers exploring the graph of a software system and inspecting units step-by-step to decide whether they are impacted by a change [14]. We call that process iterative IA (IIA). To give a more accurate illustration of IIA, a model is created based on a set of marks in Table 2-1. Those marks include the 'Propagating' mark that is used for units not changed, but still propagate a change to their interacting units (aka. neighbors) [6, 8]. Both 'Propagating' and 'Unchanged' units are inspected by programmers, but they are treated as the false positives that increase the workload of the programmers and lower the precision of impact analysis.

Figure 2-2 is an instance of iterative IA process using a generic class dependency graph (CDG) described in [3]. The nodes of CDG are classes, and the edges are program dependencies among classes. Class A is the neighbor of class B if and only if there is a dependency from B to A. If a node is scheduled as 'Impacted' or 'Propagating', all its neighbors in the dependency graph should be inspected. In that way, the programmer is presented with only a partial set of units compared to the entire transitive closure of the IIS

and not overwhelmed with the information that requires too much effort [14]. Note iterative IA process theoretically finishes when there is no unit scheduled to inspect (i.e. marked as 'Next'). However, in practice, the set of 'Next' units is still often large, so programmers can terminate the process as soon as they conclude that the EIS has been sufficient to some point [39]. In this situation, the order in which the programmers inspect the 'Next' units should impact the number of false negatives and false positives of IIA, and hence the prioritization of those units becomes an issue.

Table 2-1. Marks In Iterative IA

Mark	Meaning
Blank	Unknown status of the unit; the unit was never inspected and is not currently scheduled for inspection.
Impacted	The programmers found the recommended unit was impacted by the change, i.e., the unit belongs to the EIS. All 'Blank' neighbors of this unit must be scheduled for inspection, i.e., they are marked 'Next'.
Unchanged	The programmers found the recommended unit was not impacted by the change. This unit does not propagate the change to any of its neighbors.
Next	The unit is recommended to inspect by the programmers for a possible change.
Propagating	The programmers found the recommended unit was not impacted by the change, but the neighbors of this unit might need to change. All 'Blank' neighbors of this unit must be marked 'Next'.

On the other hand, some IA approaches attempted to predict the full EIS in a single algorithmic step (e.g. [9-12]). We call their process all-at-once IA (AIA). Malik and Hassan [40] combined machine learning with several IA techniques to predict the EIS for C programs. The best result was 78% recall and 64% precision. Several approaches were proposed to divide the IA into several subtasks and applied different IA technique in each subtask [27, 28, 36]. Among them, Zanjani et al. [17] reached 32% recall and 12% precision for IA at the granularity of files, when the size of the visited set (VS) was 20. Borg et al. [36] aimed at predicting the impact on non-code artifacts, and they got 60% recall and 7%

precision when the size of VS was 40. Some researchers investigated the union and intersection of the EIS found by different IA techniques to improve the performance [10, 11], but the recall was still low. Sun et al. [13] compared three tools based on different static IA techniques and studied the union and intersection of the results from each tool. They achieved 61% recall and 38% precision at the granularity of classes. Musco et al. [37, 38] proposed two propagation heuristics based on the execution of test cases to predict how a change in the production code impacts harness code. This resulted in 79% recall and 69% precision at the granularity of methods. To sum up, AIA techniques suffer from low recall.

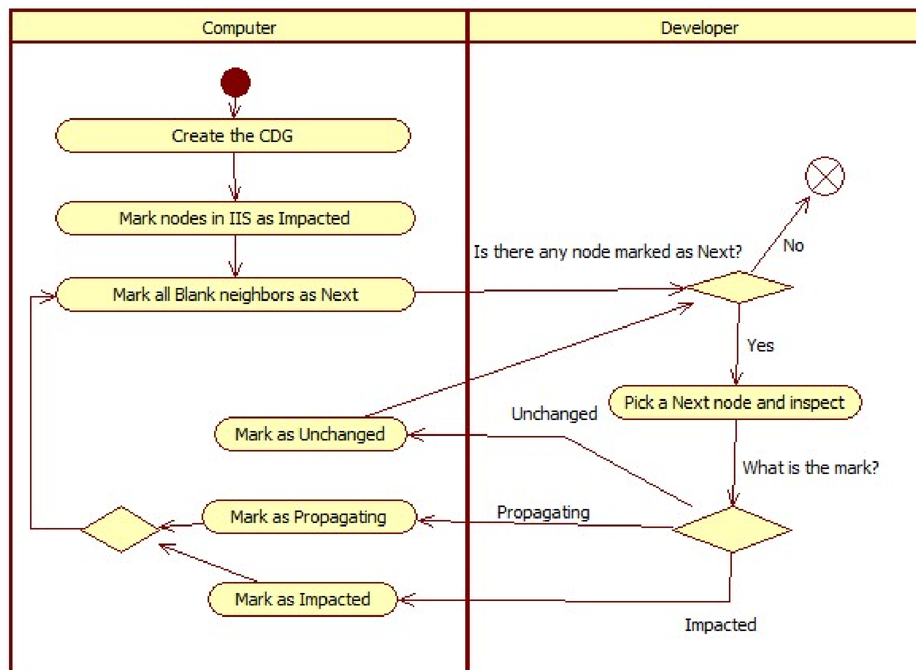

Figure 2-2. An IIA instance

Because AIA and IIA are very different processes, IA techniques designed for IIA are not suitable to apply for AIA by trivial modification and vice versa. The literature still lacks a framework to switch IA techniques between these two processes. So far, few research had been conducted to improve the IIA techniques.

2.3 Class and Member Dependency Graph and its Application

Our work is based on a Class and Member Dependency Graph (CMDG) that glue together the units of a software system at variable granularity and was proposed by Maksym and Rajlich [6, 14]. $G(V,E)$ is defined as a CMDG of the program P where the set of nodes $V = C \cup M \cup F$ representing units, C represents the set of all types (classes and interfaces) of P, M represents the set of all methods in P, and F represents the set of all fields in P. The set of edges is defined as $E = EN \cup ER$ where an edge $(x, y) \in EN$ if and only if the definition of the unit y is nested within the definition of the unit x, and an edge $(x, w) \in ER$ if and only if the unit w is referred within the definition of unit x. During IA, only ER edges are considered as potentially propagating the change among units at a certain granularity, whereas EN edges are used to shift the granularity.

A simple Java code example and its visualized representation of CMDG could be find in [8].

CMDG has been applied for concept location [41], impact analysis and actualization [6, 14] of software changes. A tool named JRipples [42] was developed to extract CMDG from Java source code using type analysis based on the Eclipse Abstract Syntax Tree (AST) and supported all related applications of CMDG.

2.4 Ownership Object Graph (OOG) and its Application

Unlike CMDG, an Ownership Object Graph (OOG) [43] over-approximates what types are created at runtime in terms of abstract objects and how they may communicate using abstract interpretation.

An OOG is defined as an *OGraph* that have two types of nodes: *OObject* represents the set of abstract objects, and *ODomain* represents the set of domains defined by the developers. A domain $D \in ODomain$ is a named, conceptual group of objects. Each domain could have zero or more objects as its children, and each object could include zero more

child domains. As a result, an arbitrary object hierarchy conveying some design intent according to domain names and their containment is achieved. For example, objects related to the architectural domain of software are at the higher levels of the hierarchy, and those related to the implementation details are placed at the lower levels.

The set of edges of *OGraph* is denoted by *OEdge*, which have four different kinds: parent-child, import dataflow, export dataflow and points-to. Parent-child relations are used to show which domain *D*∈*ODomains* contains which object *O*∈*OObjects* or verse visa. An import dataflow communication exists from the source object *m* of type *M* to the destination object *n* of type *N* if *n* receives data from *m*. An export dataflow communication exists from the source object *m* of type *M* to the destination object *n* of type *N* if one of *n*'s field *f* may be modified when one of *m*'s methods is invoked. A points-to communication represents a field reference from the object of class *C* that declares a field *f* to the object of *f*.

In plain code, there is no such information for domains. To construct a useful OOG or OGraph, it requires developers to add domain information in terms of annotations. An annotated Java code example and its OGraph could be find in [8], where the same code was also used to show CMDG. According to that example, OGraph may obtain more precise dependencies among types than CMDG considering the complicated issues caused by subclassing, programming to interfaces, aliasing, and collections of object-oriented programs.

2.4.1 OOG Applications

OOG have been applied in code comprehension, reasoning about security, dealing with distribution node, conformance analysis and impact analysis [44]. The OOG is a global points-to graph that may be able to. Due to its potentially more precise dependencies, Abi-Antoun et al. [8] conducted case studies to compared an IA technique using OGraph to the one using CMDG. They implemented a tool, ArchSummary, as an Eclipse plugin to support

their approach. Four kinds of weighted dependencies are computed based on OGraph: the most important classes (MICs), the most important related classes (MIRCs), the most important methods in a class (MIMs) and the most important classes behind an interface (MCBIs), whose definitions are in [8]. The idea of these dependencies and the tool ArchSummary were presented in [47] at first, but they were not used for IA at that time.

2.5 Heuristics of Impact Analysis

In the literature, no IA approach can guarantee the EIS of a change to be same to the AIS all the time or even be high in both precision and recall, so there is still a need for new IA approaches. Some researchers called IA techniques heuristics [5]. Researchers have attempted to combine two or more heuristics to improve the overall IA performance over the past decade. However, most of them investigated just AIA techniques, and few study was conducted about the heuristics for IIA. Under this condition, we categorize IA heuristics based on their information sources and the role in the IA process in order to provide a solution to combine heuristics for IIA.

2.5.1 Categorization based on information sources

From the view of information sources, existing heuristics can be from:

Structural information: Such heuristics utilize the information extracted from the static program dependencies such as counting certain dependencies among units. An earlier survey of heuristics based on static dependencies was presented in [45]. Li et al. [31] computed the distance of units in the call graph to decide the range of EIS. Abi-Antoun et al. [8] counted the number of dataflow and points-to communications to instruct the programmers during IA.

Execution information: Such heuristics use program traces or other runtime information. Cai et al. [24, 25] used execution differencing at statement level, and Musco et al. [37, 38] run mutation testing at method level to refine the EIS found by static analysis.

In [33, 34], Cai et al. investigated the combination of method-execution events, statement coverage and dynamic points-to sets in supporting IA tasks.

Historical information: Units that are changed together (aka. co-change) or modified frequently by the same developer may imply a close relationship and serve as a heuristic. Mining Software Repositories (MSR) methods have been applied to extract such relations, which are called evolutionary couplings [10, 26, 27]. Hassan and Holt [5, 40] studied the frequency and recency of co-changed units in predicting the EIS for C programs. Tóth et al. [7] used the correlation value of the co-changed units. Zimmermann et al. extracted association rules based on co-change information to predict the EIS [46].

Textual information: Heuristics can take into account the similarities of the text in unit names, annotations, comments, documentations, logs and change requests. In the recent work summarized in Chapter 2.1, Zanjani et al.[28], Kagdi et al. [9, 11], Gethers et al. [10, 26, 27], Borg et al. [36] and Sun et al. [13] considered information retrieval (IR) methods in finding similar units or similar change requests for a given change. Aryani et al. [12, 24-28] used common domain variables to compute the impact of a change in software components. The relations gained using IR are called conceptual couplings [10, 26, 27].

Heuristics based on the same information source need a thoroughly comparison before combining. For instance, the call relations in the call graph are included in the edges of CMDG, so it is not that helpful to combine CMDG and the call graph.

2.5.2 Categorization based on the role in the IA process

According to the role in the IA process, we divide heuristics into clustering heuristics, propagation heuristics and termination heuristics.

Clustering Heuristic: It clusters the units of the program. Given an IIS, the EIS must be in the same cluster of the IIS. In other words, the clustering heuristic determines the

best recall that an IA approach can achieve. All program representations belong to this type.

Propagation Heuristic: It guides the programmers towards the units that are likely to be impacted by the change. Propagation heuristics rank the units of a cluster from the clustering heuristic in order to improve the precision.

Termination Heuristic: Termination heuristics indicate that the EIS is complete. A termination heuristic can be used with a propagation heuristic or without.

It is worth noting that propagation heuristics and termination heuristics assist the clustering heuristic and cannot increase the recall of IA tasks once the clustering heuristic is determined.

Chapter 3 Iterative Impact Analysis based on a Global Hierarchical Object Graph

Impact analysis techniques rely on dependencies between different program units. Simple static analysis cannot expose some subtle dependencies due to interfaces, collections, and possible aliasing. As discussed in Chapter 2.4, instead of considering classes and computing dependencies based on visiting the program's Abstract Syntax Tree such as CMDG, the IA technique based on the Ownership Object Graph may provide a more precise result.

To evaluate such new approach, we conducted two case studies on two systems and five completed code modification tasks to compare the precision of dependencies extracted from OOG to those extracted from the CMDG at the granularity of types. Since CMDG has been implemented in the tool JRipples, our evaluators enacted these five tasks using ArchSummary and JRipples, respectively. The result is a detailed evaluation consists of a step-by-step comparison between ArchSummary and JRipples as both of them adopt the iterative IA process.

The work mentioned in this chapter was led by Dr. Marwan Abi-Antoun and partially collaborated with Dr. Ebrahim Khalaj.

3.1 Evaluation Method

To evaluate the approach, during each task of each case study, our evaluators completed code modification tasks after IA to get the actual impact set. The evaluation design is summarized in this section. Further details could be found in [8].

3.1.1 Environment

Two graduate students completed impact analysis for all tasks of the subject systems using ArchSummary and JRipples, respectively. Neither evaluator had prior knowledge of

the design of subject systems. All the evaluation was done under Eclipse 4.2. Both JRipples and ArchSummary are plugins for Eclipse. The JRipples evaluator used the latest version of JRipples at that time (version 3.2.2).

An additional logging functionality was added to JRipples and ArchSummary. It recorded the types that were involved during the interaction between the evaluator and the tool every step. We defined a step as each time the tool recommends a set of types based on the evaluator's operation. Because MIMs in ArchSummary are at a granularity of methods rather than types, our study did not record them.

3.1.2 Procedures

1. ArchSummary Procedure

ArchSummary adopts iterative IA process with the help of a marking system, as shown in Figure 3-1 and Table 3-1. The tool displays MICs, MIRCs and MCBIs in terms of Eclipse views. The procedure of using this tool was mainly contributed by our co-authors.

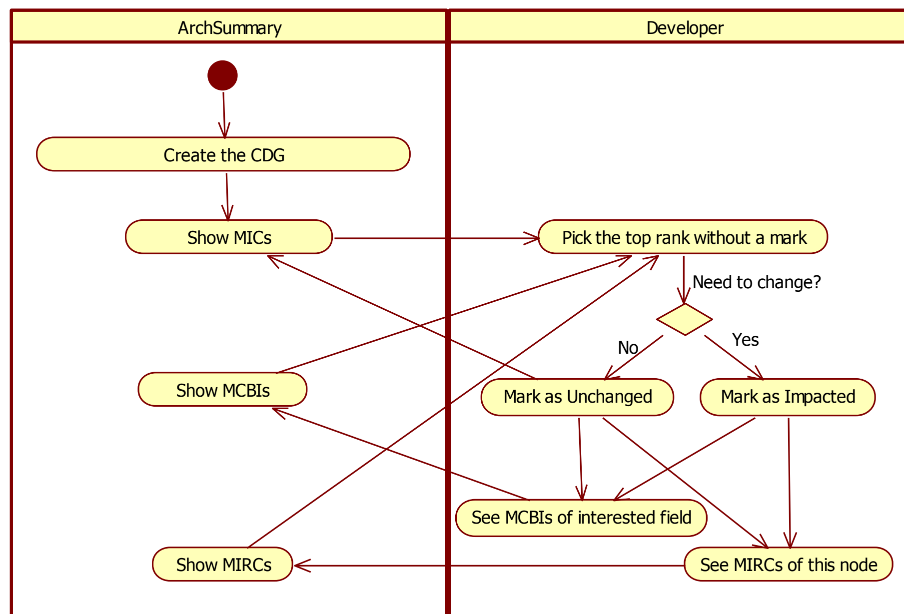

Figure 3-1. IIA in ArchSummary

The ArchSummary evaluator always started from the top-ranked class in MICs for impact analysis. ArchSummary would mark a type as 'Visited' if it was the first time that the evaluator explored it. After inspecting a class, the evaluator should set the mark to 'Impacted' if the class was scheduled to change, or 'Unchanged' otherwise. As soon as MICs has a class without any mark, the evaluator could do any of the following and jump between them:

(1) Explore a class that does not have a mark but with a highest rank from MICs.

(2) For a 'Visited' class C, pick a new class in its MIRCs that does not have a mark but with a highest rank.

(3) For a field f declared with an interface or abstract class T in class C, pick a new class that does not have a mark but with a highest rank from the MCBIs of this field.

Table 3-1. Marks In ArchSummary

Mark	Meaning
No mark	Unknown status of the type; the type was never inspected
Visited	Automatically marked if the type is visited by developer
Unchanged	A visited class that is not scheduled to change
Impacted	A visited class that is scheduled to change

ArchSummary hides the interfaces or abstract classes from all its Eclipse views such as MICs, MIRCs and MCBIs. However, the evaluator was always able to see the superclass or implemented interfaces directly when exploring a class. Moreover, the Eclipse Type Hierarchy feature provides all subtypes of a class or an interface in case the evaluator really needs it.

2. JRipples Procedure

Because we evaluate CMDG and JRipples at class level, the information of CMDG at variable granularity is summarized into class level to formulate its corresponding class dependency graph, as described in the following definitions.

Definition 1: *Let P be a program and let G = (V, E) be a directed graph where V is the set of all classes in P and E is the set of directed edges. An edge (x, y) ∈ E if and only if the class y or any member of y is referred (e.g., called, inherited, extended, instantiated, etc.) within the definition of the class x. Then, G is the Class Dependency Graph (CDG) of the program P.*

Definition 2: *For an edge (x, w) ∈ER, the type x is the neighbor type to the type w and vice versa.*

The procedure to use JRipples in this study is shown by Figure 3-2. It is a bit more constrained with the help of the marking system described in Table 2-1 from Chapter 2.2.

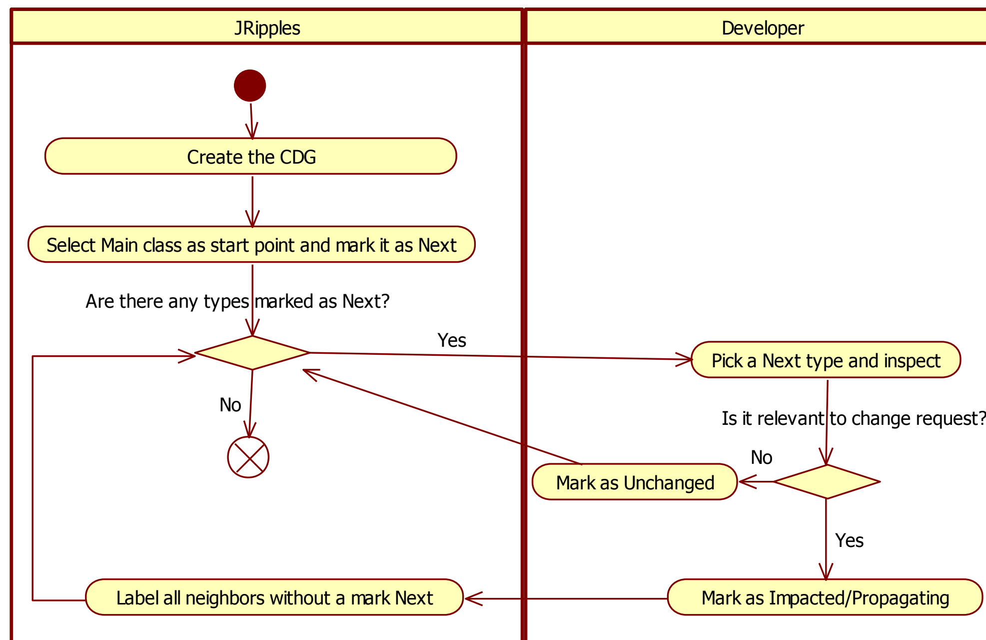

Figure 3-2. IIA in JRipples

To be short, the JRipples evaluator always started impact analysis from the main class of the subject system. The rest of the procedure is same to that in Figure 2-2. Thus, the IA result was strictly due to the dependencies of CMDG.

Table 3-2 shows the correspondence between the marks used in ArchSummary and JRipples. Such differences were taken into account for the computation of the measures in Chapter 3.1.4.

Table 3-2. Mapping of Marks between ArchSummary and JRipples

Mark	ArchSummary	JRipples
Impacted	Impacted	Impacted
Unchanged	Unchanged	Unchanged/Propagating
Next	N/A	Next
Visited	Visited	Impacted/Unchanged/Propagating

3.1.3 Subject System and Tasks

For the first study, an object-oriented framework to develop board games, MiniDraw (MD) [47], was used to conduct T1 – T4. MD includes 68 classes and interfaces, and its overall size is about 1400 lines of code. For the second case study, DrawLets (DL) is used to conduct the last task T5. DL has 138 classes and interfaces with a size of 8800 lines of code. This system supports a drawing canvas that holds figures and lets users interact with them, and it was previously studied by others [48, 49]. Table 3-3 lists the change requests of these tasks.

3.1.4 Hypothesis and Measures

The following hypothesis is proposed:

Following dependencies extracted from OOG leads to a higher precision in impact analysis compared to following dependencies from CMDG at the granularity of types.

Several measures are used to test the hypothesis:

(1) **Distinct Recommended Types (DRT)**: It is the accumulated number of distinct types that each tool recommends for the task. Given a task, the DRT for ArchSummary is the sum of the MICs of the project, the MIRCs of each 'Visited' class and the MCBIs the

evaluator invokes after eliminating duplicates. The DRT for JRipples is the union of all the types that have a mark (non-Blank).

(2) **Recommended Types per Step (RTS)**: For ArchSummary, RTS is the number of recommended classes in MIRCs when the evaluator views MIRCs of a 'Visited' class. For JRipples, RTS is the number of neighbors of a type marked as 'Propagating' or 'Impacted'. We finally compute the average (denoted by Avg) and the maximum (denoted by Max) of RTS for the given task.

Table 3-3. Change Requests of Subject Systems

Task	Change Request
T1	**Validate piece movement on the board**: the board piece can move one square straight or diagonally towards the opponent home row.
T2	**Implement the capture of a board piece:** a board piece can only capture another board piece on a diagonal move. The attacker piece takes the position of the captured one that is removed from the board.
T3	**Implement an undo feature for a piece movement on the board:** add a menu item to invoke the functionality.
T4	**Implement a status bar to be updated on each piece movement:** add a status bar to the framework.
T5	**Implement an "owner" for each figure:** an owner is a user who puts that figure onto the canvas, and only the owner is allowed to move and modify it. At the beginning, each session declares a session owner, and this session owner will own all new figures created in that session. No other user will be allowed to manipulate them. At the beginning of a session, user inputs an ID and a password. Any function that attempts to modify a figure must check that the figure owner and the current session owner are the same.

(3) **#Visited**: It measures the size of the VisitedSet. With the help of marking system, #Visited for each tool could be easily computed by the number of all types that have any mark except 'Next' for the given task.

(4) **Precision**: It is one of the most common measures for IA. The formula of precision in this evaluation is the following:

Precision = $\frac{|EIS \cap AIS|}{\#Visited}$ * 100%

In our evaluation, the types marked as 'Impacted' in ArchSummary and JRipples form the EIS. The AIS is collected manually based on the modifications the evaluators make to the code after impact analysis.

Because in our evaluation evaluators performed and verified IA manually, they were required to guarantee the highest recall of each IA task unless the actual impacted types were not detectable using the evaluated tools. In this circumstance, we did not use recall as a measure to compare these tools.

(5) In addition, we measure specific outputs for a tool and their closest counterparts in the other tool:

#MCBIs vs. #AllTypes: Given a task, every time the ArchSummary evaluator invokes MCBIs for a field declaration of an interface type, we record the classes in MCBIs and compute its size as #MCBIs. In JRipples, for every interface the evaluator explores, we record all its subtypes in Eclipse Type Hierarchy and compute its size as #AllTypes. Then, we find the intersection of interfaces in the logs of ArchSummary and JRipples, because the set of interfaces in the logs of the two tools could be very different. Next, we compute the average (denoted by Avg) and the maximum (denoted by Max) for #MCBIs and #AllTypes across this intersection to achieve a more accurate comparison. To clarify, dependencies from JRipples provide only the direct subtypes of a type, so we have to collect all subtypes using the Eclipse Type Hierarchy.

MCBIs_Invoked vs. Interfaces_Visited: From #MCBIs and #AllTypes, we compare JRipples and ArchSummary for only the same set of interfaces. We also concern about how many interfaces the ArchSummary evaluator explores and the JRipples evaluator

inspects per task. MCBIs_Invoked is the number of distinct interfaces of the field declarations on which the ArchSummary evaluator reviews MCBIs, whereas Interfaces_Visited is the number of all the interfaces in the VisitedSet of JRipples.

As we record all the information during the impact analysis automatically in CVS files, we design the following schema for the logging system of JRipples and ArchSummary:

- **ClassName:** The full name of the type (class/interface) being visited. DRT counts all names of the same task in this column after filtering out duplications. Interface_Visited counts all names of the same task in this column whose ClassType is 'INTERFACE' after filtering out duplications.
- **MethodName:** The simple name of method.
- **FieldName(ParamsName):** The simple name of the field/method parameter. Currently it only records the field name but it may also record method parameters in the future for deeper studies. **MCBI_Invoked** counts all names of the same task in this column in ArchSummary's log.
- **ClassType:** The value is 'CLASS' or 'INTERFACE'.
- **Mark:** This is used to record the mark of a visited type. For JRipples, it can be 'Next', 'Propagating', 'Located', 'Impacted' or 'Unchanged'. For ArchSummary, it can be 'Visited', 'Unchanged' or 'Impacted'. We count the set of all types with a mark except 'Next' as VisitedSet after filtering out duplications, and the set of all types with the mark 'Impacted' after filtering out duplications as the EIS.
- **Order:** This is to track the sequence of steps during the IA process.
- **Rank:** In ArchSummary, this represents the position of a type in the recommendations, and 0 means it is the currently visited type/field.
- **Comment:** The evaluator can write comments during the IA process.

- **ARS:** It records the number of All Recommended Types per Step (RTS) to compute **RTS Max** and **RTS Avg**.
- **NewTypes: New Recommendations per Step**, which records ONLY the set of new types added to the recommendations.
- **NewTypesNum:** The number of types in NewTypes.
- **AllTypes:** The list of all the subtypes in the Eclipse Type Hierarchy for the target interface or abstract class.
- **AllTypesNum:** The number of types in AllTypes.
- **MCBIs:** The list of all the types shown in the MCBI view of ArchSummary (invoked when selecting a field declaration).
- **MCBIsNum:** The number of types in MCBIs.
- **TimeStamp:** The time stamp for the current log record.

More details of the logs, raw data and the detailed reports for the navigation of the evaluators are available on http://www.cs.wayne.edu/~mabianto/arch_summary/.

3.2 Evaluation

Table 3-4 shows the results of the measures. Further analysis could be found in [8]. According to the verification after IA, both evaluators achieved 100% recall for each IA task they performed.

As shown in Table 3-4, according to #Visited of each tool, it needed to visit double or triple the number of types to complete each task using compared to that using ArchSummary.

The Max and Avg for #MCBIs and #AllTypes show the clear difference between ArchSummary and JRipples for the same set of visited interfaces. Since #MCBIs is always smaller than #AllTypes per task, this may be one scenario that ArchSummary provides more precise recommendations.

Table 3-4. Comparative Results

Task	ArchSummary		JRipples	
T1	DRT	21	DRT	24
	RTS Avg/Max	7.4 / 14	RTS Avg/Max	2.3 / 8
	#Visited	5	#Visited	12
	#MCBIs Avg	1	#AllTypes Avg	2
	#MCBIs Max	1	#AllTypes Max	2
	Precision	20%	Precision	8.3%
	MCBI_Invoked	3	Interface_Visited	4
T2	DRT	21	DRT	41
	RTS Avg/Max	6 / 14	RTS Avg/Max	8.4 / 17
	#Visited	8	#Visited	21
	#MCBIs Avg	1	#AllTypes Avg	2
	#MCBIs Max	1	#AllTypes Max	2
	Precision	25%	Precision	9.5%
	MCBI_Invoked	2	Interface_Visited	8
T3	DRT	22	DRT	30
	RTS Avg/Max	7 / 14	RTS Avg/Max	6.4 / 10
	#Visited	7	#Visited	19
	#MCBIs Avg	1	#AllTypes Avg	2
	#MCBIs Max	1	#AllTypes Max	2
	Precision	28.6%	Precision	10.5%
	MCBI_Invoked	2	Interface_Visited	8
T4	DRT	22	DRT	25
	RTS Avg/Max	7 / 14	RTS Avg/Max	8.7 / 15
	#Visited	8	#Visited	18
	#MCBIs Avg	1	#AllTypes Avg	2
	#MCBIs Max	1	#AllTypes Max	2
	Precision	25%	Precision	11.1%
	MCBI_Invoked	3	Interface_Visited	7
T5	DRT	53	DRT	100
	RTS Avg/Max	23 / 46	RTS Avg/Max	17 / 58
	#Visited	37	#Visited	97
	#MCBIs Avg	2.9	#AllTypes Avg	6.5
	#MCBIs Max	12	#AllTypes Max	19
	Precision	35.1%	Precision	16.5%
	MCBI_Invoked	8	Interface_Visited	20

MCBIs of ArchSummary allow developers to concentrate on the concrete classes that implement interfaces. During all the tasks, the ArchSummary evaluator invoked MCBIs far less times (shown by MCBIs_Invoked) compared to the number of interfaces that the JRipples evaluator had to inspect (shown by Interfaces_Visited).

Besides, the DRT of ArchSummary is always smaller, which means ArchSummary recommended fewer types in total every task compared to JRipples. From the results, the RTS values of ArchSummary could be larger or smaller than the JRipples ones. This indicates that dependencies from OOG lead to different results than those from CMDG.

To clarify, the RTS Max of ArchSummary stays at 14 for MD because from T1 to T4 the evaluator repeatedly inspected the same class that recommended the largest number of types.

For JRipples, the value of Interface_Visited is not trivial considering its corresponding DRT. This confirms that the JRipples evaluator struggled with interfaces for each task.

Overall, ArchSummary always achieved better precision as it led to a smaller VisitedSet while maintaining the highest recall. The difference between ArchSummary and JRipples was more distinguishable when the complexity of the change and the subject system increased, as shown by the data of T5.

It is worth noting that when we compared the EIS of T5 from each evaluator, we found that the JRipples evaluator detected three more classes. These classes were not in any view of MICs, MIRCs or MCBIs for ArchSummary. After investigation, we found that they were not instantiated in the subject system, and thus, they did not exist in the OOG/OGraph or any other kind of object graphs. We did not treat such classes as false negatives of ArchSummary this time as they were not used actually. Moreover, the missing classes are the subtypes of some 'Impacted' classes for the ArchSummary evaluator, so he could use the Eclipse Type Hierarchy to discover them.

3.3 Discussion

We described how the tool ArchSummary, which used new static program dependencies based on a Global Hierarchical Object Graph (i.e. OOG), supported impact analysis for a given change request after adopting an iterative IA process. The new IA

technique was evaluated by the precision and other characteristics according to case studies as well as compared with JRipples, which used static program dependencies from CMDG. For a fair comparison, we reformed the iterative IA process in JRipples such that the evaluator finds out the EIS of a given change request by merely following program dependencies the tool utilizes. The results showed that following dependencies extracted from OOG leaded to a higher precision in impact analysis compared to following dependencies from CMDG at the granularity of types.

3.3.1 Limitations of OOG

1. OOG overhead

The main problem, which limits the usage of OOG or OGraph, is that getting a meaningful OOG is somehow hard and overwhelming for developers.

When we conducted this study, the annotations of source code had to be refined manually to get a usable hierarchy for OOG. Abi-Antoun et al. [50] measured that the effort of adding annotations manually was about 1 hour/KLOC, assuming that the programmer or the evaluator has learned what the annotations mean. Thus, one can estimate the effort based on the system size.

Khalaj proposed an approach in [44, 52] that enables developers to refine an initially flat object graph into an OOG directly, which is easier than refining the annotations. However, the time effort for refining OOG is still not trivial. Furthermore, after evolving the software, developers have to evolve annotations/OOG accordingly.

2. Threat to recall

Any kind of object graphs does not contain types that are not instantiated, and OOG still belongs to object graphs. If a software change affects such un-instantiated types, they are not reachable using ArchSummary. This may lower down the recall of IA tasks.

The threats to validity of our work and other issues were discussed by our co-authors in [8].

Chapter 4 Evaluating Heuristics for Iterative Impact Analysis based on a Weighted Class Propagation Graph

Researchers have integrated different IA techniques to enhance the overall performance over the past decade. Most of these publications studied AIA techniques, and they suffered from low recall, as summarized in Chapter 2.2.

In the literature, IA techniques based on program slicing are believed to have high recall, though they are very costly in practice [24, 53]. Several approaches were proposed to approximate program slicing at the granularity of methods with lower cost [32, 54]. However, Toth et al. [7] found that at the granularity of classes, program slicing has very low recall (i.e. 11.65%) and does not meet the needs of developers.

In this context, we select a few propagation heuristics and a termination heuristic to assist IIA and evaluate their performance at the granularity of classes. Due to the limitations of OOG and ArchSummary, we adapt JRipples and the CMDG as the IIA technique in this study. Past results showed that IIA supported by JRipples can reach 100% recall [6, 8], though the precision was low. For a system with 500 classes, JRipples usually finishes building the CMDG in a minute.

In the work of Petrenko [14], several propagation heuristics were investigated. That work compared the average precision of IIA tasks when the recall is 100%. Instead of this assumption, we introduce termination heuristics. We investigate both the precision and recall for a set of propagation heuristics combined with the termination heuristic. This makes the reenactment more realistic, compared to that in [14].

4.1 Weighted Class Propagation Graph

The class dependency graph (CDG) adapted from CMDG has been defined in Chapter 3.1.2.

For impact analysis, we use the symmetric closure for the edges in CDG because the change can propagate in both directions through an edge (i.e. dependency), and we add supports for propagation heuristics.

Definition 3: *Let G = (V, E) be a CDG, then* G_H *= (V, E', W_H) is a Weighted Class Propagation Graph (WCPG) with a set of classes V, a set of edges E' where E' is a symmetric closure of E, and a set of weights* W_H *where each weight* $w_H(x, y)$ *is produced by a propagation heuristic H for an edge (x, y)* $\in$ *E'. Note* $w_H(x, y)$ *could be different to* $w_H(y, x)$*. We say y is a neighbor of x if there is an edge(x, y)* $\in$ *E'.*

During IIA, if a class *x* is marked as 'Impacted' or 'Propagating', $w_H(x, y)$ will be substituted for the value from a specific propagation heuristic to rank its neighbor *y.* We assume that a higher ranked neighbor is more likely to change or propagate the change.

4.2 Candidate Heuristics

After defining WCPG, we select the several representative heuristics adapted from previously published papers for our study.

4.2.1 Propagation Heuristics

Our study evaluates the following propagation heuristics:

(1) Dependency Based Heuristic (DBH)

An extensive survey of heuristics based on static dependencies was conducted in [45], and the PIM heuristic (i.e. the number of method invocations between classes, taking into account the polymorphism) had particularly good performance among them. PIM was also used in more recent IA studies [13, 55].

Thus, we derive our DBH from this heuristic. If call(x,y) denotes the number of times the class x calls any method of the class y, including polymorphic calls, then DBH(x,y) = call(x,y) + call(y,x). The value of DBH(x, y) is always a natural number.

(2) Class to Class similarity by Information Retrieval (CCIR)

Latent Semantic Indexing is an information retrieval method that constructs a vector space model (VSM) for texts where each text is represented by a vector. Before constructing VSM, the source code of classes is pre-processed to identify meaningful words; this may include splitting composite identifiers, removing language-specific stop-words, and so forth. A non-negative cosine value of the angle between the corresponding vectors of two texts in VSM indicates their similarity and serves as the conceptual coupling [11, 55]. Our CCIR(x, y) is the non-negative cosine value of the angle between the vectors representing classes x and y.

(3) Change Request to Class similarity by Information Retrieval (RCIR)

A change request is a text that describes the required modification to the program. The heuristic in [56] compares the text of a change request to the text in the source code of classes, and is based on the assumption that terms appearing in the text of the change request also appear in the source code of the impacted classes.

Let IR(r, x) be the non-negative cosine value of the angle between the vectors representing a class x and a change request r. We adjust it as a propagation heuristic in the following way: For every edge from a class x to a class y in G_H, RCIR(x, y) = IR(r, y).

(4) Evolutionary Coupling between Classes by Mining Software Repositories (Hist1 and Hist2)

Mining Software Repositories (MSR) methods can uncover unique relations for change propagation among program units [57], which are not detectable by program analysis. Such relations are called evolutionary couplings [10, 11]. Association rules are specific kinds of evolutionary couplings between a resource unit m and a destination unit n determined by a set of commits in a training set. Each rule comes with a *support value*, which indicates how frequently both m and n appear together in a single commit in the training set, denoted by $Asso_S(m, n)$. There is also a *confidence value*, which indicates how often the commits

containing m also contain n, denoted by $Asso_C(m, n)$. $Asso_S(m, n)$ is always symmetric but $Asso_C(m, n)$ can be different from $Asso_C(n, m)$. Zimmermann et al. explored an AIA technique based on association rules [46]. That is, for the given initial impacted unit u, all association rules using u as the resource unit with a non-zero support value are collected. Then the destination units of those rules construct the EIS and are ranked by the corresponding confidence value. This AIA technique was combined with other IA techniques in more recent studies [10, 11, 13].

In our study, we extract change history from a selected period of commits in the repositories as the training set. Then we build association rules among all classes. Next, for an edge from a class x to a class y in G_H, we investigate two propagation heuristics $Hist1(x, y) = Asso_C(x, y)$ and $Hist2(x, y) = Asso_S(x, y)$.

(5) Random Propagation Weight between Classes (RND)

We add RND, i.e. random weights ranging from 0 to 1, for all edges in G_H. RND is used as the baseline for assessing the performance of all propagation heuristics in order to show whether they are better than a completely random inspection. We generate RND weights only once for each subject system in our experiment and reuse those weights to reenact all cases of that system.

4.2.2 Termination Heuristics

Our termination heuristic, denoted by TopN, is based on the idea that a developer would inspect no more than N neighbors of every ‘Impacted’ or ‘Propagating’ class. Those are the neighbors that are ranked highest by the specific propagation heuristic that we are exploring. Similar heuristic, cut point, was used in many research papers such as [9, 11, 13]. Without termination heuristics, programmers have to inspect all neighbors iteratively from the initial impact set until all reachable classes have been inspected.

TopN is defined by the following way:

Definition 4: *Let x be a class in G_H, then neighbors(x) = { y | (x, y)$\in$E' } and weights(x) = {w_H(x, y) | y$\in$neighbors(x) }.*

Definition 5: *Given a natural number N and a class x in G_H, let y$\in$neighbors(x) and w_H(x, y) is the N'th largest weight in weights(x), then TopN(x) is a set of reachable neighbors for x where TopN(x) = {v |v$\in$neighbors(x) such that $w_H(x, v) \geq w_H(x, y)$ }.*

In order to adjust this heuristic to the subject systems of different size, we decided to use the percentage of the total number of classes in the subject system to determine the actual value N of the heuristic. We selected different percentages in this study, which are 0.5%, 1%, 2%, 3%, 4% and 5%.

4.3 Design of Case Study

The empirical method used in our study is reenactment. We integrate the selected propagation heuristics into JRipples (namely, enhanced JRipples) and instrument the most part of the reenactment as a tool to apply the termination heuristic. We run JRipples, our reenactment tool as well as subject systems in Eclipse 4.2. We assume developers using IIA would strictly follows the process described in Chapter 2.2.

4.3.1 IIA Reenactment Process Overview

An overview of the reenactment is given in Figure 4-1.

We manually mines real changes of each subject system from its repository and related information on SourceForge to compute the AIS and other input for certain heuristics. Given a subject system, we visit its closed tickets on SourceForge to extract the description of change request, change ID and resolved date of the ticket. Then we find the related commits according to the change ID and resolved date. Only if a complete and final implementation of that change request is done in a single revision, we consider it as a candidate change request and record its revision id. This is because the presence of multiple revisions for a single change request indicates that some revisions could be

incorrect, incomplete, buggy, or were about refactoring. The discrimination between these cases would require an extensive manual analysis and potentially introduce bias into our study.

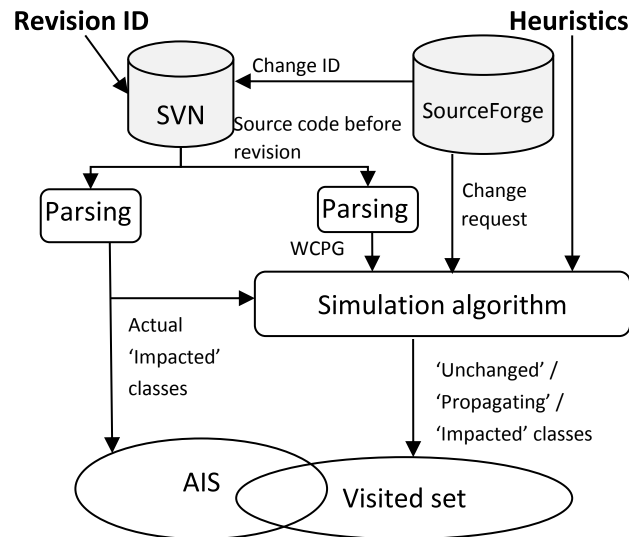

Figure 4-1. Overview of the IIA reenactment

Due to the characteristics of selected propagation heuristics and IIA, we manually parse the change details of the corresponding revision of each candidate change request and perform further filtering based on necessary criteria: (1) To fairly use RCIR, the change request should be described from the view of users instead of programmers. If a candidate change request explicitly mentions which program units should be modified or other clear information about implementation, we remove it from our study. (2) Each change request has to change at least two classes. (3) Each change request involves a single feature only.

After mining the change request and its AIS, we check out the source code of the previous revision and use the enhanced JRipples to generate the WCPG for that code based on a propagation heuristic.

The repositories do not provide information on the 'Propagating' and 'Unchanged' classes that were inspected during the IA by the d. We reconstruct the sets of these classes

by an algorithm that simulates actions of the programmers during the IIA in our reenactment tool.

4.3.2 Simulation Algorithm

The simulation algorithm used in our reenactment tool consists of the following steps:

(1) Select the initial impacted class (IIC)

From the information in the repositories and SourceForge, the actual starting point of the change is not available. Thus, for a software change that has multiple classes in its AIS, the tool repeats the simulation by selecting each class of the AIS as the IIC.

(2) Build a subgraph of WCPG based on the IIC and the termination heuristic

In the WCPG weighted by a selected propagation heuristic, the tool uses the IIC as the root and construct a subgraph of WCPG such that it contains all classes and edges reachable from IIC in WCPG after applying a specific TopN:

Definition 6: *Let $G_H = (V, E', W_H)$ be a WCPG, $c \in V$ is the IIC and N is a natural number, then $G_c = (V_c, E_c, W_c)$ is a weighted subgraph based on c with a set of reachable classes V_c for c where $V_c = \{ x \mid x=c$ or there exists $y \in V_c$ such that $x \in TopN(y)\}$. The set of reachable edges E_c is defined as $E_c = \{ (x, y) \mid$ there exists $x \in V_c$ and $y \in TopN(x)$ such that $(x, y) \in E' \}$, and for the edge $(x, y) \in E_c$ $w_c(x, y) = w_H(x, y)$.*

As an example, a G_H weighted by propagation heuristic is shown in the left part of Figure 4-2 where the members of AIS have the black filling. The initial impacted class is indicated as ‘IIC’. Suppose N=2, then the right part of Figure 4-2 shows the corresponding G_c that contains all the reachable nodes and edges of the IIC after applying the termination heuristic Top2. It can be seen that some classes of the AIS are no longer reachable from the IIC, due to termination heuristics.

(3) Identification of ‘Propagating’ Classes

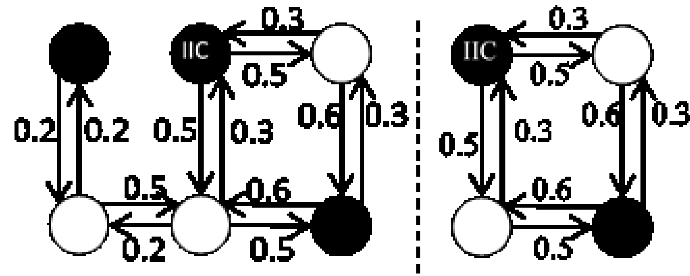

Figure 4-2. Finding reachable nodes and edges in WCPG based on the IIC and the termination heuristic Top2

After constructing G_c, the intersection between AIS and V_c represents the *reachable part of AIS* for the given propagation and termination heuristics. Then the reenactment algorithm simulates the inspections that the original developer made.

If the reachable part of AIS is disconnected, the developer must have visited ‘Propagating’ classes during IIA. The reenactment assumes that the developer visited the minimal number of ‘Propagating’ classes. For the simulation algorithm, this is equivalent to resolve the graph-theoretical directed Steiner tree problem in a weighted directed graph [58] where all edges share an identical weight.

Definition 7: *Let $G_c = (V_c, E_c, W_c)$ be a weighted subgraph rooted on $c \in V_c$, then $G'_c = (V_c, E_c)$ is a converted graph from G_c with the same set of classes V_c and same set of edges E_c as G_c and an identical weight on every edge.*

Then, the graph-theoretical Steiner tree problem is formulated in the following way: given G'_c and $M = V_c \cap AIS$, find the sub-tree T of G'_c where the root c has a path to every node in M and the sum of the weights on the paths is the minimum. Note that T may include several interconnecting nodes that are not in M; these nodes are known as the Steiner nodes.

As an example, a G'_c with all weights equal to 1 is shown in the left part of Figure 4-3 where the nodes in the reachable AIS have the black filling. The root vertex is indicated

with ‘IIC’. Then the directed Steiner tree is depicted in the right part of Figure 3, and the Steiner nodes are indicated by the letter ‘P’.

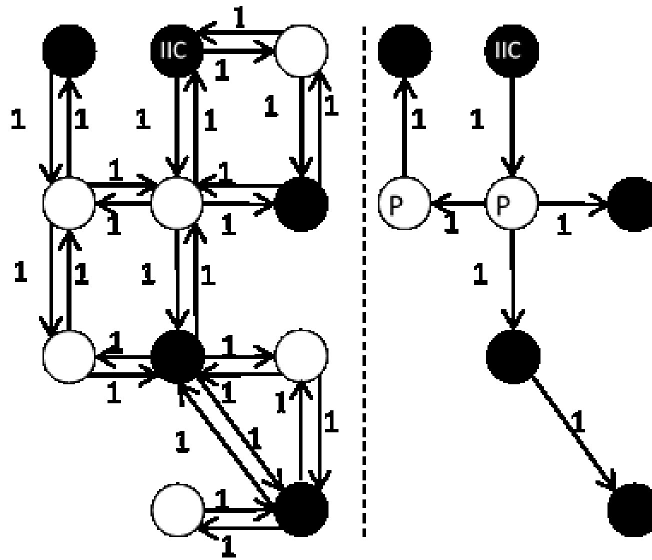

Figure 4-3. Finding Steiner nodes example. Left part is the directed graph, and the right part is the directed Steiner tree accordingly.

Since the problem of resolving directed Steiner tree is NP hard, we use an approximation solution [59] taking G'_c and the reachable AIS as the input. For a directed graph G'_c with n vertices, let X represents the set of given vertices that must be contained in the final directed Steiner tree, m is the size of X, and S represents the set of vertices in the current directed Steiner tree. Then the solution in [59] is as follows:

```
Initialize S as an empty set
For each vertex k in X
  remove k from X: X = X – {k}
    insert k into S: S = S + {k}
    for each vertex d in X
        for each node s in S
              compute the shortest path to d from s
    find the vertex in X that has the minimum sum of paths to all nodes in S, denoted by v
    record all vertices on the shortest path from k to v as P
    for each vertex u in P:
        insert u into S: S = S + {u}
        if u is in X, remove u from X: X = X – {u}
Output S, which contains all vertices of the final directed Steiner tree
```

(4) Reenactment of the visited set

When using an IIA technique, the visited set contains all inspected units, i.e. 'Impacted', 'Propagating' and 'Unchanged' units. The simulation algorithm uses G'_c to compute the visited set. For every 'Impacted' or 'Propagating' class (i.e. every class in the directed Steiner tree), it marks all neighbors as "Unchanged." The assumption of this step is that the developer inspected all these units because of the guidance by both propagation and termination heuristics. The algorithm of the last step is as follows:

```
Initialize visited impact set, visited propagating set and unchanged set as empty
Select the root r of the Steiner tree // i.e. the initial impact class
Add r to visited impact set
Call visitGraph(G'c, r)
visitGraph (Graph G, vertex v)
        for each neighbor v' of v in G
                if v' belongs to Actual Impact Set
                        add v' to visited impact set
                        call visitGraph (G, v')
                else if v' belongs to reenacted 'Propagating' Classes
                        add v' to visited propagating set
                        call visitGraph (G, v')
                else
                        add v' to unchanged set
VS = visited impact set ∪ unchanged set ∪ visited propagating set
```

4.3.3 Simulation Example in MiniDraw

To further illustrate our simulation algorithm, we reuse the board game framework MiniDraw and the task T2 we had implemented in Chapter 3. The change request is "Implement the capture of a board piece: a board piece can only capture another board piece on a diagonal move. The attacker piece takes the position of the captured one that is removed from the board."

Due to the Observer design pattern used in MiniDraw, programmers need to modify both the method pieceMovedEvent() of the class BoardDrawing and the method move() of the class GameStub to complete this change request. Thus, the actual impact set contains two classes, which serve as one input for our reenactment tool.

MiniDraw has 68 classes and interfaces. Figure 4-4 shows a partial Weighted Class Propagation Graph based on DBH heuristic using the code before the implementation of this task. The classes filled by red represent the AIS, and the class filled by blue is the class containing the main function of this program.

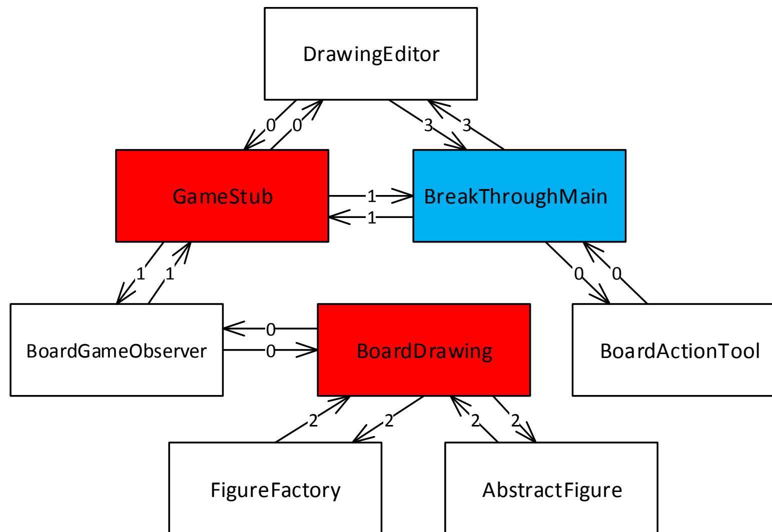

Figure 4-4. Partial WCPG of MiniDraw based on DBH heuristic

The first step of our simulation is to pick up the IIC from the AIS and load the actual value of N for the termination heuristic. **Suppose GameStub is the IIC and N value is 2.**

Then, Figure 4-5 is the subgraph G_c accordingly where the IIC is highlighted by the color yellow. Fortunately, the whole AIS is reachable from the IIC in this case.

After getting G_c, the weights on all edges are replaced by the same value 1 in order to compute the directed Steiner tree that interconnects classes GameStub and BoardDrawing in G_c. The result is shown in Figure 4-6 where all Steiner nodes have the yellow filling. In other words, the class BoardGameObserver serves as a 'Propagating' class in this simulation if the programmer starts IA from GameStub and applies heuristics DBH and Top2.

The last step of the simulation is reenacting the visited set. This would include all classes in the directed Steiner tree as well as related 'Unchanged' classes. In this example, the

'Unchanged' classes are shown in Figure 4-7 using have the green filling. Thus, in this reenactment, we assume the developer visits six classes in total assisted by the selected heuristics.

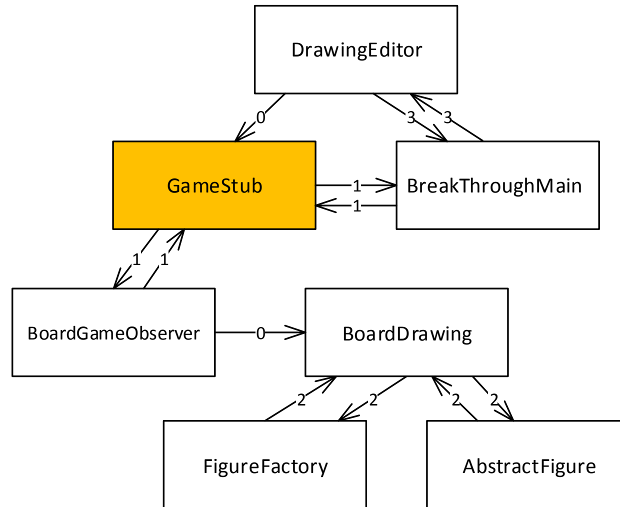

Figure 4-5. Reachable classes and edges based on Top2 for the class GameStub

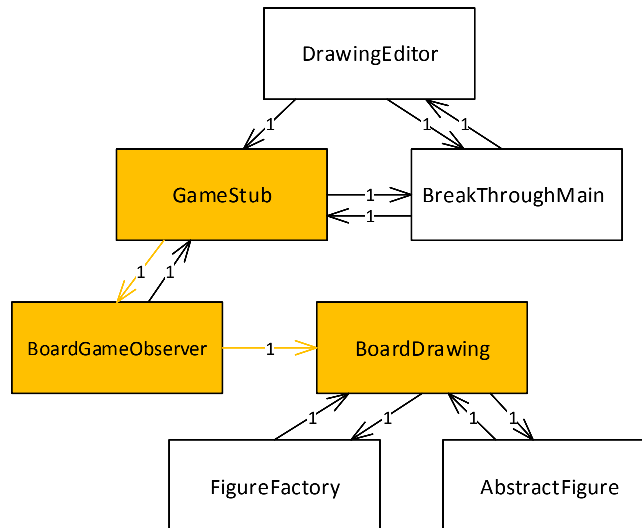

Figure 4-6. Directed Steiner tree to connect GameStub and BoardDrawing

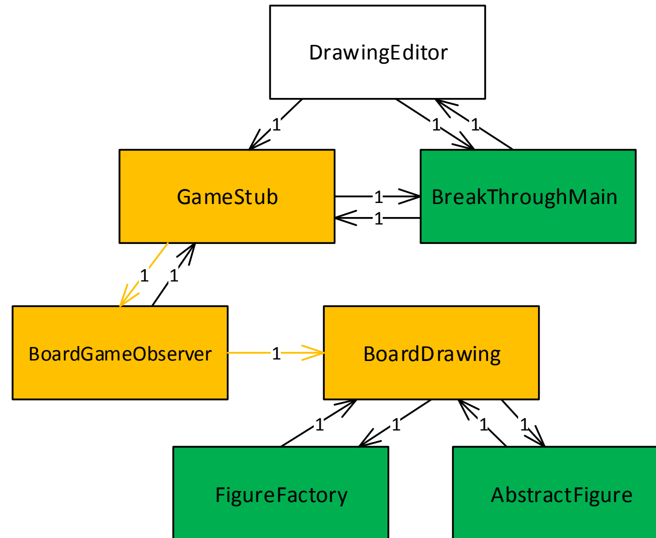

Figure 4-7. Visited Set in the reenactment. Nodes in yellow estimates 'Impacted' or 'Propagating' classes, and nodes in green estimates 'Unchanged' classes.

4.3.4 Subject Systems

Our study is conducted on three different open source java projects, which are listed in Table 4-1.

To get association rules used by heuristics Hist1 and Hist2, we select all commits in SVN repository during a certain date interval, as shown by the columns *Date Interval* and *#Commits* of Table 4-1. Some of systems may be selected by other researcher in the literature of impact analysis. For instance, JEdit and the time interval (2004-12-31 to 2009-12-22) were used in [9, 11] for evaluating some AIA techniques at the granularity of methods.

Table 4-1. Subject Systems

System	**Version**	**LOC**	**Classes**	**History for Association Rules**		**# Requests**
				Date Interval	**# Commits**	
JEdit[1]	4.3	109k	531	[2004-12-31, 2009-12-22]	2051	15
JHotdraw[2]	7	83k	568	[2006-11-1, 2010-8-1]	411	10
QuickFIX/J[3]	1.5.3	30k	281	[2005-2-28, 2011-11-2]	1187	11

[1] https://sourceforge.net/projects/jedit/
[2] https://sourceforge.net/projects/jhotdraw/
[3] https://sourceforge.net/projects/quickfixj/

For JEdit, we extract 15 change requests that were completed right after the selected time interval. The AIS of each change request involves 2 to 6 classes.

For JHotdraw, we finally extract 10 change requests that were completed between 2010-8-2 and 2017-1-25. The AIS of each change request includes 2 to 7 classes.

For QuickFIX/J, we are able to extract 11 change requests that were completed between 2011-11-3 and 2014-5-9. The AIS of each change request covers 2 to 5 classes.

4.3.5 Measures

For each change request *cr* with a specific initial impacted class *c*, we collect its precision P(cr, c) and recall R(cr, c):

$$\text{P(cr, c)} = \frac{|VS \cap AIS - c|}{|VS - c|}$$

$$\text{R(cr, c)} = \frac{|VS \cap AIS - c|}{|AIS - c|}$$

Next, the precision P(cr) and recall R(cr) for each change request is the average of precisions and recalls calculated for each possible initial impacted class:

$$\text{P(cr)} = \sum_{c \in AIS} \frac{\text{P(cr, c)}}{|AIS|}$$

$$\text{R(cr)} = \sum_{c \in AIS} \frac{\text{R(cr, c)}}{|AIS|}$$

For each subject system, let CR denote the set of its change requests, then we measure the average precision P_{avg} and average recall R_{avg}:

$$P_{avg} = \sum_{cr \in CR} \frac{\text{P(cr)}}{|CR|}$$

$$R_{avg} = \sum_{cr \in CR} \frac{\text{R(cr)}}{|CR|}$$

The results of P(cr), R(cr), P_{avg} and R_{avg} are discussed in Chapter 4.3.

4.4 Evaluation

Table 4-2 to Table 4-5 show P_{avg} and R_{avg} of investigated heuristics of each subject system, along with the corresponding standard deviations according to involved P(cr) and

R(cr). The column *ActN* of these tables lists the actual N value corresponding to a specific percentage given in the column *TopN(%)* for a system. At the end of each table, we also compute the overall average performance of all change requests from these three systems after classifying the results by the same percentage value of TopN.

Figure 4-8 to Figure 4-11 depict how the performance changes with the increase of TopN for JEdit, JHotdraw, QuickFIX/J and overall, respectively.

In addition, Figure 4-12 to Figure 4-14 show how the median of P(cr) and R(cr) changes with the increase of TopN for JEdit, JHotdraw and QuickFIX/J, respectively.

Table 4-2. Average Recall of Investigated Heuristics (%)

TopN (%)	ActN	RND	DBH	Hist1	CCIR	Hist2	RCIR	Subject
0.5	11	87.5	40	60.8	40.8	60.8	35.8	JEdit
1	16	97.5	92.5	87.5	82.5	87.5	87.5	
2	22	100	100	95	95	95	100	
3	27	100	100	95	100	95	100	
4	11	100	100	100	100	100	100	
5	16	100	100	100	100	100	100	
0.5	3	75	70.5	31.8	59.1	31.8	45.6	JHotdraw
1	6	90.9	93.2	84.1	95.5	84.1	66.3	
2	11	97.7	95.5	97.7	97.7	97.7	95.5	
3	17	97.7	97.7	97.7	97.7	97.7	97.7	
4	23	97.7	97.7	97.7	97.7	97.7	97.7	
5	28	97.7	97.7	97.7	97.7	97.7	97.7	
0.5	2	68.1	45.8	36	68.1	36	37.5	QuickFIX/J
1	3	82.5	72.5	71.3	82.5	71.3	58.1	
2	6	100	95	90	97.5	90	90	
3	9	100	95	97.5	100	97.5	100	
4	12	100	100	100	100	100	100	
5	15	100	100	100	100	100	100	
0.5	-	78.1	50.2	45.2	54.2	45.2	39.0	Overall
1	-	91.1	86.6	81.6	86.1	81.6	72.6	
2	-	99.4	97.2	94.2	96.5	94.2	95.7	
3	-	99.4	97.8	96.5	99.4	96.5	99.4	
4	-	99.4	99.4	99.4	99.4	99.4	99.4	
5	-	99.4	99.4	99.4	99.4	99.4	99.4	

4.4.1 Discussion of results

According to Table 4-2, IIA combined with RND provides a better recall than many IA techniques in the literature, which have been summarized in Chapter 2.2. However, when TopN is low, RND also reaches a better recall compared to the propagation heuristics that we investigated in our study. In addition, propagation heuristics based on information retrieval or evolutionary couplings lead to low recall for low TopN, especially when it is Top0.5%.

Table 4-3. Standard Deviation of Recall (%)

TopN (%)	ActN	RND	DBH	Hist1	CCIR	Hist2	RCIR	Subject
0.5	11	30	43	37	38	37	39	JEdit
1	16	13	28	36	30	36	28	
2	22	0	0	26	26	26	0	
3	27	0	0	26	0	26	0	
4	11	0	0	0	0	0	0	
5	16	0	0	0	0	0	0	
0.5	3	24	20	33	25	33	27	JHotdraw
1	6	14	13	19	7	19	25	
2	11	5	11	5	5	5	11	
3	17	5	5	5	5	5	5	
4	23	5	5	5	5	5	5	
5	28	5	5	5	5	5	5	
0.5	2	36	38	27	36	27	35	QuickFIX/J
1	3	29	36	36	29	36	36	
2	6	0	11	18	10	18	31	
3	9	0	11	6	0	6	0	
4	12	0	0	0	0	0	0	
5	15	0	0	0	0	0	0	
0.5	-	31	36	33	34	33	35	Overall
1	-	20	28	32	25	32	30	
2	-	3	8	20	18	20	18	
3	-	3	7	17	3	17	3	
4	-	3	3	3	3	3	3	
5	-	3	3	3	3	3	3	

It is worth noting that in JHotdraw, the highest recall on average stops at 97.7% for any investigated propagation heuristic. This is caused by a specific change request related to Revision ID 783: “It should distinguish between large icon and small icon. This way, an Action can use different icons for buttons and menu items.”

Table 4-4. Average Precision of Investigated Heuristics (%)

TopN (%)	ActN	RND	DBH	Hist1	CCIR	Hist2	RCIR	Subject
0.5	11	11.9	10.3	11.3	13.1	11.3	12.3	
1	16	10.3	12.1	9.8	9.8	9.8	12.3	
2	22	7.3	8.2	7.4	7.8	7.4	8.6	
3	27	6	6.4	6.2	6.4	6.2	6.8	JEdit
4	11	5.2	5.6	5.6	5.7	5.6	5.8	
5	16	5	5.2	5.1	5.2	5.1	5.3	
0.5	3	10.8	16.9	10.1	12.6	10.1	19.9	
1	6	9.7	10.8	9.3	11.2	9.3	13.4	
2	11	7.1	7.9	7.1	7.7	7.1	8.7	
3	17	5.6	5.9	6	6.3	6	6.8	JHotdraw
4	23	4.9	5.3	5.3	5.6	5.3	5.9	
5	28	4.8	4.9	4.9	5.1	4.9	5.4	
0.5	2	12.1	46.1	24.8	29.9	24.8	35	
1	3	12.5	23.5	22.3	24.1	22.3	33.9	
2	6	9.4	13.5	14.6	13.7	14.6	16.5	
3	9	7.7	9.8	10.8	9.7	10.8	11.7	QuickFIX/J
4	12	6.4	8.4	8.8	7.8	8.8	9.4	
5	15	5.9	7.6	7.4	6.8	7.4	8.2	
0.5	-	11.7	23.1	15.1	18.1	15.1	21.3	
1	-	10.8	15.2	13.5	14.6	13.5	19.2	
2	-	7.9	9.7	9.5	9.6	9.5	11.0	
3	-	6.4	7.3	7.6	7.4	7.6	8.3	Overall
4	-	5.5	6.4	6.5	6.3	6.5	6.9	
5	-	5.2	5.9	5.7	5.7	5.7	6.2	

This change affects 7 classes. One class of the AIS, CrossPlatformApplication, is only interacting with another class named ResourceBundleUtil. Unfortunately, in the neighborhood of ResourceBundleUtil by every propagation heuristic including RND, CrossPlatformApplication is ranked lower than Top5%, which leads to a situation that our

reenactment is not able to reach 100% recall unless using CrossPlatformApplication as the IIC.

In JEdit and QuickFIX/J, RCIR always get the better precision compared to any other propagation heuristic for the same TopN, as shown in Table 4-4.

Similar performance of RCIR continues in JHotdraw, However, DBH provides good precision, which is lower only than RCIR, while maintaining far better recall.

Table 4-5. Standard Deviation of Precision (%)

TopN (%)	ActN	RND	DBH	Hist1	CCIR	Hist2	RCIR	Subject
0.5	11	5	11	8	13	8	12	JEdit
1	16	5	6	6	5	6	6	
2	22	4	4	4	4	4	3	
3	27	3	3	3	3	3	3	
4	11	3	3	3	3	3	3	
5	16	3	3	3	3	3	3	
0.5	3	2	8	8	6	8	8	JHotdraw
1	6	2	5	3	4	3	6	
2	11	2	3	2	2	2	3	
3	17	2	2	2	2	2	3	
4	23	1	2	2	2	2	2	
5	28	2	2	2	2	2	2	
0.5	2	9	39	19	21	19	33	QuickFIX/J
1	3	6	18	13	13	13	22	
2	6	3	5	6	5	6	8	
3	9	2	3	4	3	4	3	
4	12	3	2	3	2	3	3	
5	15	2	2	2	2	2	3	
0.5	-	6	28	14	17	14	22	Overall
1	-	5	12	10	10	10	16	
2	-	3	5	5	5	5	6	
3	-	3	3	4	3	4	4	
4	-	3	3	3	3	3	3	
5	-	3	3	3	3	3	3	

Though the confidence value and support value of an association rule are different, they rank the neighborhood of an 'Impacted' or 'Propagating' class in the same order. As a

result, the simulation using Hist1 has the exactly same performance as that using Hist2 in all the cases.

Figures 4-8 to 4-11 show that for each propagation heuristic, both precision and recall become stable after TopN reaches 2% for all subject systems. This may imply that Top2% is a sufficient termination heuristic and there is no need to investigate termination heuristics that require an inspection of the larger number of neighbors.

Also note that when the value of TopN is low, reenactment requires a lot of 'Propagating' classes in order to achieve the best recall. This is why for some propagation heuristics, the precision is improved when TopN increases at the beginning.

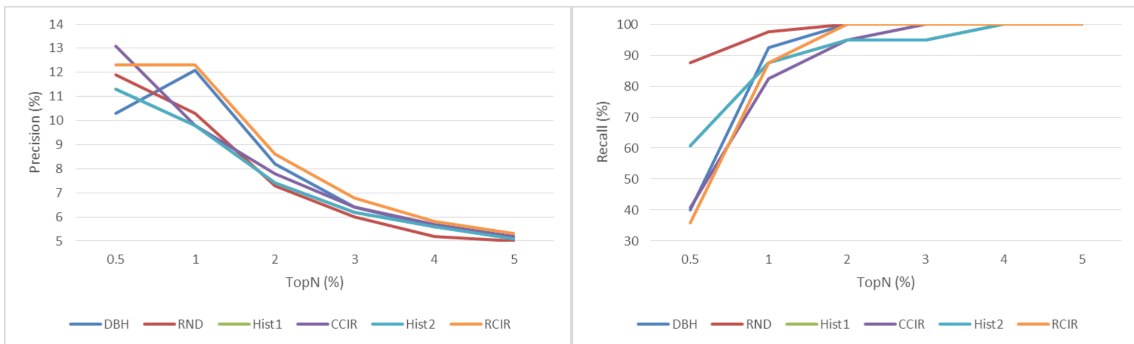

Figure 4-8. Average precision (left) and recall (right) of investigated heuristics for JEdit

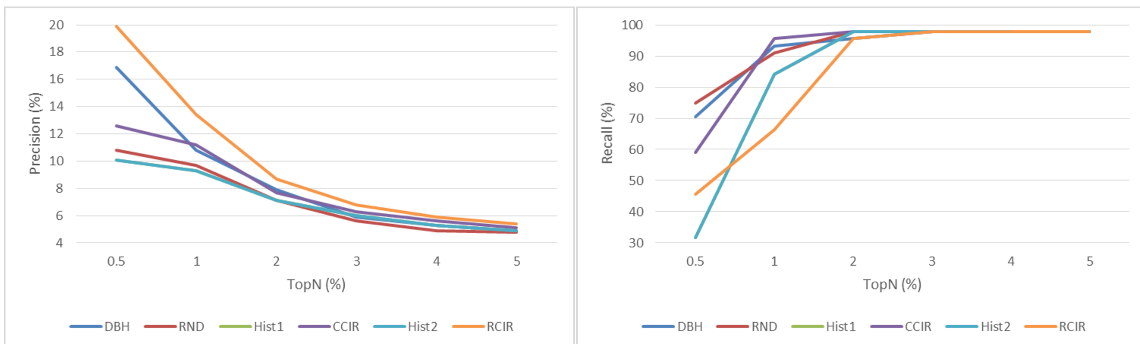

Figure 4-9. Average precision (left) and recall (right) of investigated heuristics for Jhotdraw

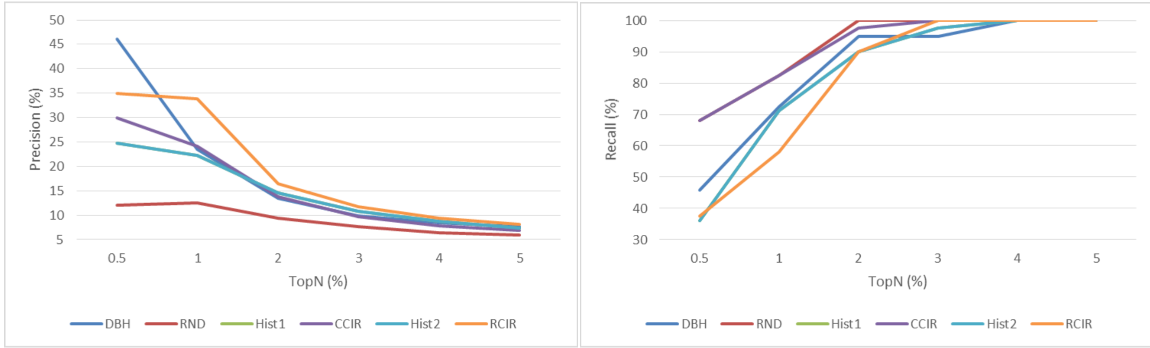

Figure 4-10. Average precision (left) and recall (right) of investigated heuristics for Quickfix/J

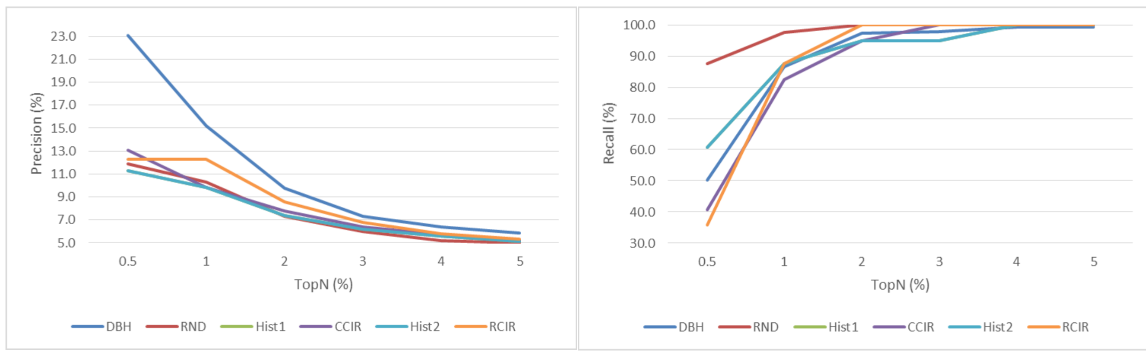

Figure 4-11. Average overall precision (left) and recall (right)

According to Figure 4-12 to Figure 4-14, once the median of recall reached 100%, any investigated propagation heuristic leads to slightly better median precision compared to random inspection.

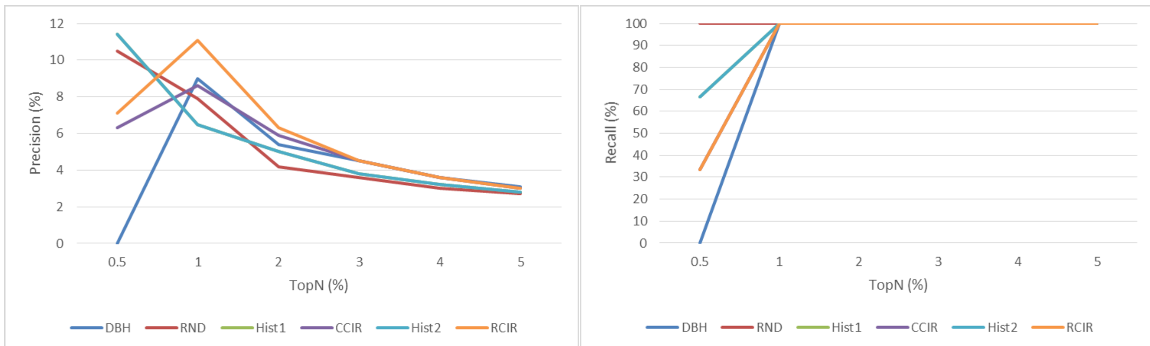

Figure 4-12. Median precision (left) and recall (right) of investigated heuristics for JEdit

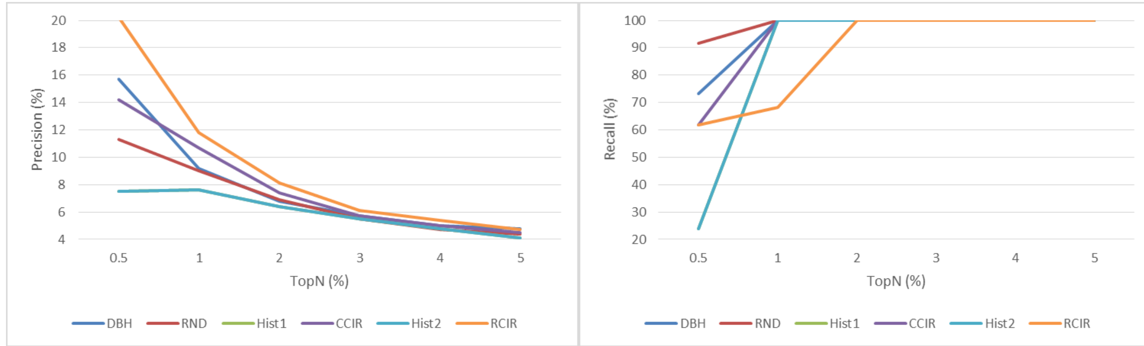

Figure 4-13. Median precision (left) and recall (right) of investigated heuristics for Jhotdraw

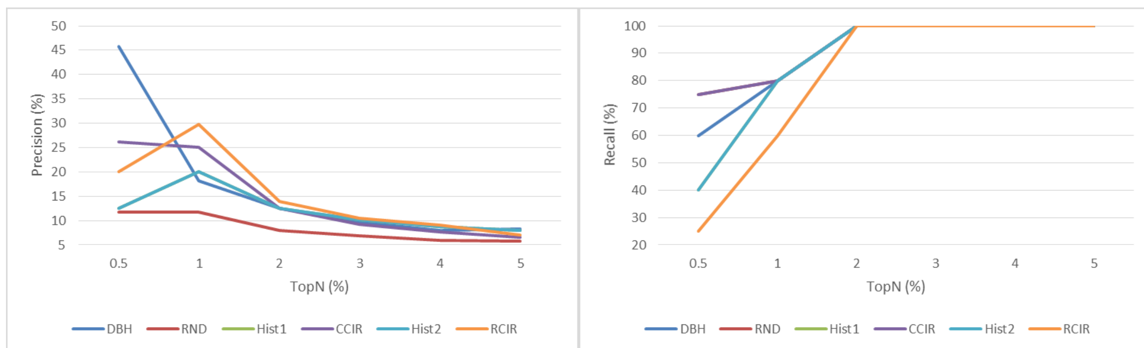

Figure 4-14. Median precision (left) and recall (right) of investigated heuristics for Quickfix/J

4.4.2 Threats to Validity

Our study deals with the granularity of classes. Different results may be obtained for other granularities.

We evaluate only Java programs. Different results may be obtained for programs implemented by other programming languages.

Some investigated heuristics such as Hist were adapted from AIA techniques. Those techniques are very different compared to IIA, as described in section III. In our solution, we convert them into propagation heuristics. Thus, some advantages of such heuristics may not be maintained during this conversion. It is possible that a different solution to adapt AIA techniques into IIA may provide different results.

Software repositories do not provide information about the initial impacted class. In the study, we consider every possible starting point of an impact analysis task. The performance of an IA using a particular propagation heuristic may vary slightly based on the selected IIC; however, this threat to validity is minor and does not affect the relative rankings of the investigated propagation heuristics.

Reenactment, as presented in this paper, may have certain built-in biases. Other empirical techniques, for example, empirical study of human developers, may provide different results.

Chapter 5 Conclusions

Software changes occur frequently in modern software development. Impact analysis helps programmers to understand the system and estimate what units should be modified in order to ensure the quality of changes.

In this thesis, we investigate iterative impact analysis techniques. They use program representations consisting of program units and program dependencies. Starting from the initial impacted unit, programmers inspect other units of the program following those dependencies iteratively to identify the consequence of the change.

This thesis established two novel approaches to improve IIA. The first approach is based on an assumption that a new program representation, Ownership Object Graph, could increase the precision of IIA while keeping the high recall. To evaluate this approach, we conducted case studies on two Java systems and five complete change tasks. Moreover, we designed various measures to provide quantitative and qualitative comparisons for IIA based on these two program representations. The results showed that IIA based on OOG led to a much better precision and maintained 100% recall for each change task.

In the second approach, we evaluated the performance of IIA combined with several representative propagation heuristics adapted from previously published papers and one termination heuristic. Those propagation heuristics include: Dependency Based Heuristic (DBH), Class to Class similarity by Information Retrieval (CCIR), Change Request to Class similarity by Information Retrieval (RCIR) and Evolutionary Coupling between Classes by Mining Software Repositories (Hist1 and Hist2). To support the evaluation, we designed a novel empirical method based on the reenactment of IIA that simulates the actions of developers and reenacted the past changes of open source projects mined from software repositories. As a result, IIA combined with all the propagation heuristics that we explored performed better than other techniques known from the literature in terms of recall.

However, all these heuristics fell short of expectations as they did not provide a convincing improvement when compared to the random inspection.

5.1 Future Work

The investigated approaches lay out a foundation of our future work.

In our first approach, OOG achieves better performance for IA tasks. However, producing OOG from the source code is costly, and lowering the cost is a research issue [44, 52].

In view of the negative result of the second approach, searching for good IIA heuristics is still on. In the future, we would like to build on the experience from this thesis to seek for more effective propagation heuristics. For example, classes ranked higher by multiple propagation heuristics may be more likely correct for IA compared to the ones ranked higher only by a single propagation heuristic. Furthermore, we plan to explore additional termination heuristics.

The methodology that we developed – reenactment – is giving us a clear comparison between different heuristics, and hence it will help us assess whether the newly proposed heuristics are an improvement compared to the old ones. Hopefully in the future, a better IIA heuristics will emerge and help the developers to make more predictable and safe changes in software.

REFERENCE

[1] S. A. Bohner, "Software Change Impact Analysis," presented at the IEEE CS, 1996.

[2] R. S. Arnold and S. A. Bohner, "Impact Analysis-Towards a Framework for Comparison," in Proceedings of Conference on Software Maintenance, Sept. 1993, pp. 292-301.

[3] V. Rajlich, Software Engineering: The Current Practice: CRC Press, 2011.

[4] S. S. Yau, J. S. Collofello, and T. M. MacGregor, "Ripple Effect Analysis of Software Maintenance," in Software engineering metrics I, S. Martin, Ed., ed: McGraw-Hill, Inc., 1993, pp. 71-82.

[5] A. E. Hassan and R. C. Holt, "Replaying development history to assess the effectiveness of change propagation tools," Empirical Software Engineering, vol. 11, pp. 335-367, 2006/09/01 2006.

[6] M. Petrenko and V. Rajlich, "Variable Granularity for Improving Precision of Impact Analysis," in IEEE 17th International Conference on Program Comprehension (ICPC '09), 2009, pp. 10-19.

[7] G. Tóth, P. Hegedűs, Á. Beszédes, T. Gyimóthy, and J. Jász, "Comparison of different impact analysis methods and programmer's opinion: an empirical study," presented at the Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java, Vienna, Austria, 2010.

[8] M. Abi-Antoun, Y. Wang, E. Khalaj, A. Giang, and V. Rajlich, "Impact Analysis Based on a Global Hierarchical Object Graph," in 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'15), Montréal, 2015, pp. 221-230.

[9] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. L. Collard, "Blending Conceptual and Evolutionary Couplings to Support Change Impact Analysis in Source Code," in 17th Working Conference on Reverse Engineering (WCRE'10), 2010, pp. 119-128.

[10] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk, "Integrated Impact Analysis for Managing Software Changes," in 34th International Conference on Software Engineering (ICSE), 2012, pp. 430-440.

[11] H. Kagdi, M. Gethers, and D. Poshyvanyk, "Integrating Conceptual and Logical Couplings for Change Impact Analysis in Software," Empirical Software Engineering, vol. 18, pp. 933-969, 2013/10/01 2013.

[12] A. Aryani, "Predicting change propagation using domain-based coupling," Ph.D dissertation, Computer Science and Information Technology, RMIT University, 2013.

[13] X. Sun, B. Li, H. Leung, B. Li, and J. Zhu, "Static change impact analysis techniques: A comparative study," Journal of Systems and Software, vol. 109, pp. 137-149, 11// 2015.

[14] M. Petrenko, "On use of dependency and semantics information in incremental change," Ph.D dissertation, Wayne State University, 2009.

[15] V. Rajlich, Software Engineering: The Current Practice: Taylor & Francis, 2012.

[16] L. A. Wilson, Y. Senin, Y. Wang, and V. Rajlich. (2019, April 01). Empirical Study of Phased Model of Software Change. arXiv e-prints. Available: https://ui.adsabs.harvard.edu/abs/2019arXiv190405842W

[17] S. Lehnert, "A taxonomy for software change impact analysis," presented at the Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution, Szeged, Hungary, 2011.

[18] B. Li, X. Sun, H. Leung, and S. Zhang, "A survey of code-based change impact analysis techniques," Software Testing, Verification and Reliability, vol. 23, pp. 613-646, 2013.

[19] A. Aryani, I. D. Peake, M. Hamilton, H. Schmidt, and M. Winikoff, "Change Propagation Analysis Using Domain Information," in Australian Software Engineering Conference (ASWEC '09), 2009, pp. 34-43.

[20] A. Aryani, I. D. Peake, and M. Hamilton, "Domain-based change propagation analysis: An enterprise system case study," in IEEE International Conference on Software Maintenance (ICSM'10), 2010, pp. 1-9.

[21] A. Aryani, F. Perin, M. Lungu, A. N. Mahmood, and O. Nierstrasz, "Can We Predict Dependencies Using Domain information?," in 18th Working Conference on Reverse Engineering (WCRE'11), 2011, pp. 55-64.

[22] M. S. Rahman, A. Aryani, C. K. Roy, and F. Perin, "On the relationships between domain-based coupling and code clones: An exploratory study," in 35th International Conference on Software Engineering (ICSE'13), 2013.

[23] A. Aryani, F. Perin, M. Lungu, A. N. Mahmood, and O. Nierstrasz, "Predicting dependences using domain-based coupling," Journal of Software: Evolution and Process, vol. 26, pp. 50-76, 2014.

[24] H. Cai, S. Jiang, R. Santelices, Y.-J. Zhang, and Y. Zhang, "SENSA: Sensitivity Analysis for Quantitative Change-Impact Prediction," in IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2014, pp. 165-174.

[25] H. Cai, R. Santelices, and S. Jiang, "Prioritizing Change-Impact Analysis via Semantic Program-Dependence Quantification," IEEE Transactions on Reliability, 2015.

[26] M. Gethers, H. Kagdi, B. Dit, and D. Poshyvanyk, "An Adaptive Approach to Impact Analysis from Change Requests to Source Code," presented at the Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, 2011.

[27] B. Dit, M. Wagner, S. Wen, W. Wang, M. L. Vásquez, D. Poshyvanyk, et al., "ImpactMiner: A Tool for Change Impact Analysis," in ICSE Companion, 2014, pp. 540-543.

[28] M. B. Zanjani, G. Swartzendruber, and H. Kagdi, "Impact Analysis of Change Requests on Source Code Based on Interaction and Commit Histories," presented at the Proceedings of the 11th Working Conference on Mining Software Repositories, Hyderabad, India, 2014.

[29] B. Li, X. Sun, and H. Leung, "Combining concept lattice with call graph for impact analysis," Advances in Engineering Software, vol. 53, pp. 1-13, 11 2012.

[30] B. Li, X. Sun, and J. Keung, "FCA–CIA: An approach of using FCA to support cross-level change impact analysis for object oriented Java programs," Information and Software Technology, vol. 55, pp. 1437-1449, 2013/08/01/ 2013.

[31] B. Li, Q. Zhang, X. Sun, and H. Leung, "WAVE-CIA: A Novel CIA Approach Based on Call Graph Mining," presented at the Proceedings of the 28th Annual ACM Symposium on Applied Computing, Coimbra, Portugal, 2013.

[32] H. Cai and R. Santelices, "Abstracting Program Dependencies Using the Method Dependence Graph," in 2015 IEEE International Conference on Software Quality, Reliability and Security (QRS), 2015, pp. 49-58.

[33] H. Cai and R. Santelices, "A Framework for Cost-effective Dependence-based Dynamic Impact Analysis," in Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on, 2015, pp. 231-240.

[34] H. Cai, R. Santelices, and D. Thain, "DiaPro: Unifying Dynamic Impact Analyses for Improved and Variable Cost-Effectiveness," ACM Transactions on Software Engineering and Methodology, vol. 25, pp. 1-50, 2016.

[35] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Efficient and precise dynamic impact analysis using execute-after sequences," presented at the Proceedings of the 27th international conference on Software engineering, St. Louis, MO, USA, 2005.

[36] M. Borg, K. Wnuk, B. Regnell, and P. Runeson, "Supporting Change Impact Analysis Using a Recommendation System: An Industrial Case Study in a Safety-Critical Context," IEEE Transactions on Software Engineering, vol. 43, pp. 675-700, 2017.

[37] V. Musco, A. Carette, M. Monperrus, P. Preux, and F. Lille, "A Learning Algorithm for Change Impact Prediction," in 5th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, 2016.

[38] V. Musco, A. Carette, M. Monperrus, and P. Preux, "A Learning Algorithm for Change Impact Prediction: Experimentation on 7 Java Applications," arXiv preprint arXiv:1512.07435, 2015.

[39] C. Dorman and V. Rajlich, "Software Change in the Solo Iterative Process: An Experience Report," in 2012 Agile Conference, 2012, pp. 21-30.

[40] H. Malik and A. E. Hassan, "Supporting Software Evolution Using Adaptive Change Propagation Heuristics," in 24th IEEE International Conference on Software Maintenance (ICSM 2008), 2008, pp. 177-186.

[41] M. Petrenko and V. Rajlich, "Concept location using program dependencies and information retrieval (DepIR)," Information and Software Technology, vol. 55, pp. 651-659, 2013.

[42] J. Buckner, J. Buchta, M. Petrenko, and V. Václav, "JRipples: A Tool for Program Comprehension during Incremental Change," presented at the Proceedings of the 13th International Workshop on Program Comprehension, 2005.

[43] M. Abi-Antoun and J. Aldrich, "Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure using Annotations," presented at the Proceedings of the 24th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA), Orlando, Florida, USA, 2009.

[44] M. E. Khalaj, "Automated Refinement Of Hierarchical Object Graphs," Wayne State University, 2017.

[45] L. C. Briand, J. Wüst, and H. Lounis, "Using Coupling Measurement for Impact Analysis in Object-Oriented Systems," in IEEE International Conference on Software Maintenance (ICSM '99), 1999, pp. 475-482.

[46] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining Version Histories to Guide Software Changes," IEEE Transactions on Software Engineering, vol. 31, pp. 429-445, 2005.

[47] H. B. Christensen, Flexible, Reliable Software: Using Patterns and Agile Development: CRC Press, 2011.

[48] M. Skoglund and P. Runeson, "A Case Study on Regression Test Suite Maintenance in System Evolution," in 20th IEEE International Conference on Software Maintenance, 2004, pp. 438-442.

[49] V. Rajlich and P. Gosavi, "Incremental Change in Object-Oriented Programming," IEEE Software, vol. 21, pp. 62-69, 2004.

[50] M. Abi-Antoun, N. Ammar, and Z. Hailat, "Extraction of Ownership Object Graphs from Object-Oriented Code: an Experience Report," presented at the Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures, Bertinoro, Italy, 2012.

[51] R. Vanciu and M. Abi-Antoun, "Object Graphs with Ownership Domains: an Empirical Study," in Aliasing in Object-Oriented Programming. Types, Analysis and Verification, ed: Springer, 2013, pp. 109-155.

[52] E. Khalaj and M. Abi-Antoun, "Inferring Ownership Domains from Refinements," presented at the Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, Boston, MA, USA, 2018.

[53] M. Acharya and B. Robinson, "Practical Change Impact Analysis Based on Static Program Slicing for Industrial Software Systems," presented at the Proceedings of the 33rd International Conference on Software Engineering, Waikiki, Honolulu, HI, USA, 2011.

[54] A. Beszedes, T. Gergely, J. Jasz, G. Toth, T. Gyimothy, and V. Rajlich, "Computation of Static Execute After Relation with Applications to Software Maintenance," in IEEE International Conference on Software Maintenance (ICSM 2007), 2007, pp. 295-304.

[55] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, "Using information retrieval based coupling measures for impact analysis," Empirical Software Engineering, vol. 14, pp. 5-32, 2009.

[56] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia, "Identifying the Starting Impact Set of a Maintenance Request: a Case Study," in European Conference on Software Maintenance and Reengineering, 2000, pp. 227-230.

[57] H. Kagdi, M. L. Collard, and J. I. Maletic, "A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution," Journal of Software Maintenance and Evolution: Research and Practice, vol. 19, pp. 77-131, 2007.

[58] F. K. Hwang and D. S. Richards, "Steiner Tree Problems," Networks, vol. 22, pp. 55-89, 1992.

[59] H. Takahashi and A. Matsuyama, "An approximate solution for the Steiner problem in graphs," Mathematica Japonica, vol. 24, pp. 573-577, 1980.

ABSTRACT

INTEGRATING HEURISTICS TO SUPPORT IMPACT ANALYSIS IN SOFTWARE EVOLUTION

by

YIBIN WANG

December 2019

Advisor: Dr. Václav Rajlich

Major: Computer Science

Degree: Doctor of Philosophy

Iterative impact analysis (IIA) is a process that allows developers to estimate the impacted units of a software change. Starting from a single impacted unit, the developers inspect its interacting units via program dependencies to identify the ones that are also impacted, and this process continues iteratively. Experience has shown that developers often miss impacted units and inspect many irrelevant units.

In order to enhance IIA, first we put forward a new program representation that provides more precise dependencies for software change propagation. Our study showed that the precision of IIA was indeed improved using such a program representation while the high recall was maintained.

Second, we distinguished propagation heuristics that guide developers to find the actual impacted units and termination heuristics that help to decide whether the estimated impact is complete. The roles of these two kinds of heuristics are complementary and affect both the precision and recall when used during IIA. We investigated several propagation heuristics adapted from previously published papers and combined them with a practical termination heuristic. We developed a reenactment process that simulates the actions of

developers who use those heuristics during IIA, and we assessed their performance. The software changes for our reenactment were mined from the repositories of open source projects. We found that IIA provides better recall than the other known impact analysis techniques. However, the IIA with the propagation heuristics that we investigated does not supersede IIA combined with a random inspection, and hence these heuristics do not help the IIA.

AUTOBIOGRAPHICAL STATEMENT

Yibin Wang is a Ph.D. candidate in the Department of Computer Science at Wayne State University in Detroit. He received his M.S. in Computer Science and Technology from China University of Petroleum, Shandong, China in 2013.

Wang's research interests lie in software evolution, program comprehension and static program analysis. Specifically, he aims at supporting developers during the software changes to facilitate the analysis and estimations. He has published and co-authored a conference paper in these areas in IEEE International Conference on Software Analysis, Evolution, and Reengineering.

He served as a co-reviewer for peer-reviewed journals and conferences, including Journal of Software: Evolution and Process, Transactions on Software Engineering, ACM Transactions on Software Engineering and Methodology, and International Conference on Program Comprehension.