
by
Stephanie Perez Day Galbo
Bachelor of Science
Computer Science
University of Central Florida
2010

A thesis
submitted to the School of Computing
at Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Software Engineering

Melbourne, Florida
April, 2017
Approval Statement

The undersigned committee hereby recommends that the attached document be accepted as fulfilling in part the requirements for the degree of Software Engineering Masters in Computer Science and Cybersecurity.

by
Stephanie Perez Day Galbo

__________________________
Keith B. Gallagher, Ph.D.
Primary Advisor
Program Chair, Software Engineering

__________________________
Deborah S. Carstens, Ph.D.
Outside Committee Member
Program Chair, Aviation Human Factors

__________________________
Walter P. Bond, Ph.D.
Committee Member
Professor, Software Engineering

__________________________
Cem Kaner, Ph.D.
Committee Member
Professor, Software Engineering

__________________________
Ronaldo Menezes, Ph.D.
Professor, School of Computing
Program Chair, Computer Science & Software Engineering
Abstract

Title: A Survey of Impact Analysis Tools for Effective Code Evolution

Author: Stephanie Perez Day Galbo

Major Advisor: Keith B. Gallagher, Ph.D.

Background: Software maintenance and evolution are becoming noticeably more important activities in the software development process, with the growing popularity of Agile/SCRUM and continuous integration. Impact analysis, the activity of identifying the consequences of a change before the change is made, can help the developer understand the consequences of their potential changes and therefore make informed decisions about incremental changes, with the ultimate goal of minimizing negative impacts on the current baseline. To aid developers with impact analysis, there are a number of tools with a variety of capabilities to facilitate this process.

Aim: In order to help developers take advantage of the cost-saving benefits of these tools, we would like to identify the impact analysis tools and plugins that currently exist in academia and the industry. There may be missed opportunities for tools developed for research purposes to be transferred to the industry.

Method: In this paper, we will perform a systematic literature review of the impact analysis tools that exist today, and identify the various analysis types and capabilities offered by the various tools. We have also selected two tools, JRipplles and ImpactMiner for an informal usability inspection, to gauge the industry readiness for these tools.

Results and Conclusion: In this review, 20 impact analysis tools were selected for this literature review, and our results showed a variety of tools with different techniques and capabilities. We also discovered that there is an opportunity for the transferal of impact analysis tools from academia to industry to aid developers in maintenance and evolution.
## Contents

- **Abstract** iii
- **List of Figures** vii
- **List of Tables** viii
- **Abbreviations** ix
- **Acknowledgements** x

### 1 Introduction
1.1 Purpose ........................................ 2
1.2 Scope ........................................... 3
1.3 Organization ..................................... 4

### 2 Background
2.1 Maintenance and Evolution .......................... 6
2.2 Impact Analysis Classification and Comparison ............ 8
2.3 Impact Analysis in the Industry ........................... 11
2.4 Impact Analysis Techniques ........................... 12
   2.4.1 Program Slicing .................................. 12
   2.4.2 IA and Repository Mining ....................... 13
2.5 IA Tool Features .................................. 13

### 3 Method
3.1 Overview ........................................ 15
3.2 Systematic Literature Review ............................. 15
3.3 Literature Search .................................... 16
3.4 Study Selection ..................................... 16
3.5 Data Extraction and Analysis ............................... 18
   3.5.1 A Framework for Classifying Impact Analysis Tools ........ 18
      3.5.1.1 General Tool Information ................ 19

iv
# 4 Study

## 4.1 Tool Descriptions

- **4.1.1 Chianti** ................................................. 24
- **4.1.2 CodeSurfer** ........................................... 25
- **4.1.3 EAT** .................................................. 26
- **4.1.4 eROSE** ............................................... 26
- **4.1.5 FaultTracer** .......................................... 27
- **4.1.6 Frama-C** ............................................. 27
- **4.1.7 Imp** .................................................. 28
- **4.1.8 ImpactMiner** ....................................... 28
- **4.1.9 ImpactViz** ........................................... 29
- **4.1.10 Impala** ............................................... 29
- **4.1.11 Indus** ............................................... 30
- **4.1.12 JArchitect** ......................................... 30
- **4.1.13 JRipples** ........................................... 31
- **4.1.14 JSlice** ............................................... 31
- **4.1.15 JUnit/CIA** ......................................... 32
- **4.1.16 Kaveri** ............................................. 33
- **4.1.17 Lattix** ............................................... 33
- **4.1.18 REST** ............................................... 33
- **4.1.19 SLICE** ............................................... 34
- **4.1.20 Unravel** ........................................... 34

## 4.2 General Information .............................................. 35

## 4.3 Tool Accessibility ............................................... 36

## 4.4 Analysis Types and Techniques ................................ 37

## 4.5 Capabilities ...................................................... 38

## 4.6 Usability Inspection ............................................. 39

### 4.6.1 JRipples ................................................. 40

### 4.6.2 ImpactMiner ............................................ 41

## 4.7 Analysis .......................................................... 43

### 4.7.1 Overview .................................................. 43

### 4.7.2 Tool Accessibility ........................................ 43

### 4.7.3 Analysis Types and Techniques ............................ 44

### 4.7.4 Capabilities ............................................... 45

### 4.7.5 Usability Inspection ........................................ 47
List of Figures

2.1 Best Practices for Committing Code ........................................ 7
4.1 Results of a JRipple Analysis ................................................. 40
4.2 Results of an ImpactMiner IR Analysis ...................................... 42
4.3 Programming Languages Supported by Impact Analysis Tools ....... 44
4.4 Tool Support Lifespan .......................................................... 45
4.5 Availability of Impact Analysis Tools ......................................... 46
4.6 Analysis Types of Impact Analysis Tools ................................... 46
4.7 IA Techniques of Impact Analysis Tools .................................... 47
4.8 Capabilities of Impact Analysis Tools ....................................... 48
## List of Tables

<table>
<thead>
<tr>
<th></th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>General Information on Selected Tools</td>
<td>36</td>
</tr>
<tr>
<td>4.2</td>
<td>Availability of Selected Tools</td>
<td>37</td>
</tr>
<tr>
<td>4.3</td>
<td>Analysis Types of Selected Tools</td>
<td>38</td>
</tr>
<tr>
<td>4.4</td>
<td>Capabilities of Selected Tools</td>
<td>39</td>
</tr>
<tr>
<td>1</td>
<td>Full Results from the Systematic Literature Review, Part 1</td>
<td>54</td>
</tr>
<tr>
<td>2</td>
<td>Full Results from the Systematic Literature Review, Part 2</td>
<td>55</td>
</tr>
</tbody>
</table>
Abbreviations

AIS  Actual Impact Set
AST  Abstract Syntax Tree
CIA  Change Impact Analysis
CVS  Concurrent Versions System
EIS  Estimated Impact Set
NIST National Institute of Standards and Technology
SCM  Software Configuration Management
SIS  Starting Impact Set
SVN  Subversion
TFS  Team Foundation Server
Acknowledgements

I would like to thank Dr. Keith Gallagher, my advisor, for introducing me to this area of study and for guiding me in my research.

Also, I would like to thank my parents for their support, and my husband Vincent, for his patience and encouragement, and reminding me that “the only way to eat an elephant is one bite at a time.”
“How does a project get to be a year late?... One day at a time.”

Fred Brooks
Chapter 1

Introduction

Jerry Weinberg [1] noted in 1982 that improper impact analysis in software systems can lead to billion-dollar errors. More recent work [2] notes that even software system developers are highly unreliable in assessing impacts on the systems they have built. According to claims in a NIST study in 2002, software companies reported that they lost an estimate of over 18 developer hours per error, and for minor errors, firms reported an estimated loss of over $3,000 per error [3].

It is clear that we need to have better support for processes and tools for better error prevention. Even if the process change is something as simple as changing developer habits to perform error prevention techniques before committing, it could potentially save multiple industries hundreds of thousands of dollars. By equipping developers with automated tools to perform error prevention tasks, this minimizes the impact to the development process.

Impact analysis (IA), identifying the potential consequences of a change [4], is one of these error prevention techniques. Impact analysis is a semantic question. The programmer wants to know “What will happen if I make a change to this variable / method / function?” Program slicing, dependency analysis, and measurement of ripple effect are all variations of IA techniques.

There is a variety of impact analysis tools created for various purposes. Most of these tools were created for academic use to demonstrate a particular IA technique, and when the research is completed, they seem to reach a crossroads. Most tools appear to be
abandoned and are never repackaged or transitioned for industry use. Other tools, such as JRipples [5], Kaveri [6], and FaultTracer [7], are made open-source, to allow the software development community to contribute to and take advantage of the benefits of impact analysis.

There are a few commercial tools that offer impact analysis, such as JArchitect [8] and Lattix [9], but there is a noticeable lack of the quantity of tools and the depth of capability demonstrated in the tools created for research purposes. There should be a methodology implemented into the roadmap for experimental academic tools, to ensure that the noted cost-saving techniques are taken advantage of and transitioned accordingly for widespread and long term use.

1.1 Purpose

Many researchers have found that the ability to perform impact analysis during the development and maintenance phases of the software life cycle have great benefits [10] [11]. There have been many tools developed for academic research that encapsulate the benefits of impact analysis, but there is a noticeable lack of these tools made available to developers commercially.

This paper aims to survey a variety of known impact analysis tools that exist, either academically or commercially, and categorize them according to a developed framework. Another goal is to identify the features of these tools that are useful for developers in order to perform preventive maintenance. The outcome of these findings should be able to provide a foundation for future development of an impact analysis tool that developers can use to facilitate code evolution and maintenance.

With the rising popularity continuous integration and Agile/SCRUM development environments, effective rapid code evolution is becoming increasingly important. Developers commit their code and a build server subsequently runs builds, tests, and static code analysis checks. In continuous integration, when every developer commits to the same baseline, it is immensely critical that a developer avoids committing bad code if possible. A bad commit could in turn “break the build”, which halts any future tests or static code analysis checks from running (and thus, halts progress for other developers).
Developers have the responsibility to make sure their commit will not break the build, but it still happens, and can be potentially disrupting to the team’s workflow.

Running regression tests locally is one method that developers use to ensure that their commit does not break the build. Developers may choose to run all tests or manually select a subset of tests that appear to be relevant to the change set. One drawback to this is that running tests locally can take a long time, especially if the developer chose to run all tests. A developer could also choose the wrong subset of tests, which could still lead to a broken build.

There are many impact analysis techniques that have been developed and explored, but very few tools are observed to have delivered these techniques to software developers in the industry. These techniques have been shown to improve software maintenance and reduce the quantity of errors introduced into main baseline. It stands to reason that these techniques should be encapsulated in usable tools for developers. There is a recognized need for effective impact analysis tool development in order to ensure widespread and continuing usage of impact analysis techniques. [12] [13]

The goal for effective everyday software maintenance is to give the developer the tools to identify the extent of the propagation of changes made, and allow the developer to make an informed decision on whether or not it is safe to commit their code. By cataloging and analyzing the capabilities that existing impact analysis tools offer, we can establish what exists today, and identify any potential opportunities for future impact analysis tools.

1.2 Scope

There are many tools that exist for the purpose of performing impact analysis, and only a subset of these tools have been chosen to be examined. This study focuses on tools that aid the developer in the pre-commit phase of development, before changes have been integrated into the main project baseline. In this paper, we want to discover tools that help developers answer the question, “What will happen if I make this change?” and have a degree of confidence about the integrity of their outgoing changesets.

It is noted that there are a variety of other tools available that aid software maintenance outside of the realm of impact analysis, such as static code analysis tools. Static code
analysis tools, such as Findbugs\(^1\) or Checkstyle\(^2\), can greatly improve the stability and efficiency of a developer’s code, by alerting the developer to coding standard violations. These tools should be used in conjunction with impact analysis tools in order to ensure that the code being integrated is of high quality and will not negatively impact the existing baseline.

There are also a number of tools that exist to help the developer understand the relationships and architecture of their code, and many of them include visualization components. These tools were considered, but many of them were excluded if they also did not implement a more intelligent method or algorithm involving impact analysis.

It is also noted that the term “impact analysis” is a broad term. Impact analysis can be performed at many levels, from a code level, to a system component, and even at a conceptual level. Many tools were discovered that cover these varieties of analyses, but many were regarded as out of scope for the purpose of this paper. This paper aims to focus on impact analysis at the code level, using algorithms that are capable of parsing and analyzing programming languages.

### 1.3 Organization

This paper is organized into five sections: **Introduction**, **Background**, **Method**, **Study**, and **Discussion**. In the **Introduction**, we have presented an initial background and the purpose for carrying out this work. In the **Background** section, we will discuss software maintenance and evolution, and the techniques used to aid code evolution. In the **Method** section, we present a detailed plan for executing the systematic literature review and the usability inspection. In the **Study** section, we present the results gathered from the systematic literature review and the usability inspection, and provide an analysis of the results. Finally, in the **Discussion** section, we provide some final thoughts on the results of the study and identify opportunities for future work.

\(^1\)http://findbugs.sourceforge.net/
\(^2\)http://checkstyle.sourceforge.net/
Chapter 2

Background

Software maintenance can be categorized into four classes: adaptive, perfective, corrective, and preventive [14]. Many software programs end up spending precious time and money on corrective and adaptive software maintenance, which causes overblown budgets and missed deadlines. These are reactive types of maintenance, and usually performed in response to a discovered fault in the system or process. In order to spend less time doing reactive maintenance, it is important to devote resources to proactive maintenance (perfective and preventive).

The type of maintenance that we are most concerned with in this research is preventive maintenance. Preventive maintenance is defined by IEEE as the following:

Preventive maintenance: modification of a software product after delivery to detect and correct latent faults in the software product before they become operational faults.

In addition to correcting faults in software after delivery, we would like to incorporate these activities into the development process, which is typically referred to as software evolution.
2.1 Maintenance and Evolution

Rajlich [15] has discussed the importance of effective evolutionary software development. Most of software development happens in a state of evolution. Concept location and impact analysis are two important activities involved in effective code evolution. Rajlich notes that continuous improvement is needed in tool support for such a fundamental task.

It is acknowledged that nearly every piece of software will eventually have to be maintained. However, as our software development process improves, it becomes apparent that development goes hand-in-hand with code evolution. The best practices involved in software maintenance should be utilized through the entire software cycle.

The Agile/SCRUM process has slowly gained popularity as the benefits of incremental change are recognized. Development is done in incremental “sprints”, usually lasting 2-4 weeks. The goal at the end of each sprint is to complete a specific component of functionality, which can generally be demonstrated at a “sprint demo”. Because of these short bursts of development, with the need to continuously keep the main threads functional, each developer has a responsibility to maintain high stability for each commit in order to keep the baseline in working order. Bad commits, which can break the build or break integration tests, can cause development to slow down or even come to a halt until the error can be corrected. In some cases, a significant number of hours can be devoted to fixing these errors in order to allow development to continue, and could potentially result in missed deadlines due to the unexpected nature of these problems.

To protect the baseline, software development teams should have best practices in mind to form good habits amongst their developers before they commit code. To practice effective preventive maintenance and stable code evolution, this is more involved than simply making changes and committing them. If possible, developers should make an effort to perform the following activities before committing their code:

- **Identify the extent of impacted code**: The developer should use their best judgment and knowledge of the code to identify any unintended impacts and any co-changes needed to complete the changeset.

- **Run regression/unit tests**: The developer should attempt to determine the relevant unit tests to the change and run them to ensure previous functionality is still working.

6
Figure 2.1: Best Practices for Committing Code

Note: A visual inspection of the code should be performed at each step if possible.

- **Visual inspection of code**: The developer should look over all changes made to verify all changes are intentional and are properly documented. This should be done at each step if possible.

To ease load on developers, especially when they may not be intimately familiar with all aspects of the code, tools can be used to carry out intelligent analysis of the impact of changes, identify any needed co-changes by analyzing commit histories, and identify any relevant regression tests.

Due to industry observations [10] [11], it is clear that implementing effective processes that incorporate impact analysis can result in more discovered errors and fewer schedule impacts, and therefore, less negative impacts to the budget. Because these development environments often work at a fast pace, it is critical that developers have efficient tools at their disposal in order to ensure an error-free commit.

Preventive maintenance and code evolution go hand-in-hand with performing impact analysis (also known as change impact analysis), which involves estimating and identifying the potential consequences of a specified changeset. In order to carry out effective code evolution, a developer must be aware of the relevant relationships within their code.
If the developer was the one who originated the code, this aids the process, but in an evolutionary environment, this is rarely the case. Tools are necessary for programmers to provide insight into the quality and relationships into code that they may not be familiar with.

Impact analysis is a broad area of study that encompasses other areas such as program slicing [16], program dependence graphs, dependence clusters [17], and the “ripple effect” [18]. Each of these areas is concerned with the relationship between statements in a program, and how a given set of statements can impact the rest of the program. At a low level, this is a useful metric or set of metrics for the software developer in order to help with software evolution. Understanding the consequences of making a change can aid the developer in program comprehension at the pre-commit stage.

2.2 Impact Analysis Classification and Comparison

Arnold and Bohner [4] devised a framework for classifying different kinds of impact analysis techniques, at a time when there was not a clear definition for impact analysis. Many techniques existed, but there was no effective framework for comparison. The framework sought to find the relationships between the starting impact set (SIS), the estimated impact set (EIS), and the actual impact set (AIS). Four classification types were developed: SIS and EIS, EIS and System, EIS and AIS, and Granularity. The effectiveness of this approach was tested using impact analysis techniques such as program slicing and manual cross referencing.

Li et al [12] explored an array of change impact analysis (CIA) techniques using a systematic literature review, and categorized each of the identified techniques based on different classifications of impact analysis. A framework was developed to assign properties to each impact analysis technique. These properties are as follows:

- **Object**: The changeset and source for analysis
- **Impact Set**: The impacted elements of the system
- **Type of analysis**: Divided into two main types - static and dynamic
- **Intermediate Representation**: The technique derives an intermediate representation from the program to enhance analysis
• **Language Support**: Programming language and paradigm support

• **Tool Support**: Industry adoption of a impact analysis technique can depend on the type of tool support that exists

• **Empirical Evaluation**: How the IA technique has been empirically evaluated, and the results of the evaluation

This framework was then used to compare and classify 23 different IA techniques.

Lehnert [19] developed a taxonomy to classify different approaches to impact analysis. Building on the work of Arnold and Bohner [4], which did not capture all aspects of IA techniques, the following classification criteria were developed:

• **Scope of Analysis**: Type of analysis performed with respect to code or models (ex: static, dynamic)

• **Granularity of Entities**: The level at which the analysis is applied (ex: granularity of artifacts, changes, or results)

• **Utilized technique**: The type of IA technique used (ex: program slicing, program dependence graphs, history mining)

• **Style of analysis**: The strategy supported by the approach (ex: global, exploratory)

• **Tool Support**: Any tools (if they exist) that implement the approach

• **Supported Languages**: The programming or modeling languages supported by the approach (ex: Java, C)

• **Scalability**: Support for scalability for real-world projects

• **Experimental Results**: Results from evaluating the approach (ex: size of system, precision)

Of particular importance among the subcriteria for Experimental Results are precision and recall, which have commonly been used to evaluate approaches to impact analysis [12] [13] [20] [21]. Both of these terms can be defined by the entities AIS (actual impact set) and EIS (estimated impact set), introduced by Arnold and Bohner[4].
Precision represents the fraction of identified-as-impacted classes that were actually impacted, and can be represented as follows [19]:

\[\text{Precision} = \frac{|EIS| \cap |AIS|}{|EIS|}\]  

(2.1)

Recall represents the fraction of actually impacted classes were correctly identified by the algorithm, and can be represented as follows [19]:

\[\text{Recall} = \frac{|EIS| \cap |AIS|}{|AIS|}\]  

(2.2)

There are several similarities observed when comparing the framework from Li [12] and the taxonomy from Lehnert [19]. For example, both classification schemes include type of analysis, tool support, language support, and evaluation results. Both recognize the importance and necessity of tool support in order for adoption of IA techniques.

Wilkerson [22] also devised a taxonomy for classifying change impact analysis algorithms. The taxonomy developed was focused on classifying the actual algorithms themselves, by examining how the algorithms function on a lower level. Classification at the top level started by identifying techniques that either have a direct or indirect impact on the source code. Subtypes for direct impact include: addition impact, deletion impact, and modification impact. Subtypes for indirect impact include: lookup impact, field value changed impact, access changed field impact, invoked method impact, invoking method method, and external change impact.

Toth et al. [13] performed a comparison of different impact analysis techniques (call-graph, static slicing, static execute-after, and co-changing version-controlled files), and discovered that there is a need to enhance existing techniques. It was also noted that there is a need to create automated impact analysis tools to ensure the longevity of these techniques. To test the different IA techniques, JRipples was used on programmers and gather data on the precision and recall values of the algorithms. They also compared how the programmer’s opinion of the estimated impact matched up against the estimated impact by the algorithms. While opinions varied greatly, they found that programmers were open to accepting the results presented by the tool.
2.3 Impact Analysis in the Industry

Han [23] noted that both impact analysis and change propagation are interrelated and are crucial activities in software engineering environments. Additional tool support is needed for effective management of code changes. This includes understanding the extent and reach of a change before the change is actually made (impact analysis), and carrying out the modifications safely without negatively impacting the rest of the code (change propagation). Tools can help guide developers through performing these activities in an automated fashion to reduce chances of human error.

Ajila [24] also proposes a model for impact analysis and change propagation. The WHAT-IF model is a language-agnostic method for evaluating impacts to a software system through four phases of the software life cycle: requirements, specification, design, and programming. Impact analysis is noted as being necessary in order to estimate the cost of a code change, understand the meaning and relationship of a change, to record the history of change related information, and to identify regression tests that need to be re-run.

De Souza and Redmiles [11] performed an empirical study with software developers in the industry to understand their approaches to change impact analysis. Two studies were performed at two different companies and the teams were observed to understand their methods of impact management. Three aspects to impact management were recognized: impact network (the network of people that could impact or be impacted by your change), forward impact management (managing how your work affects others), and backward impact management (managing how others’ work affects your own). It was observed that different approaches work better for different team roles, and approaches had to be modified as the impact network grew.

Goeritzer [10] performed a case study on a software application called ExactOnline in order to understand how the development team use impact analysis in their process. An IA tool was created for the study, and the teams used it in their development process for 3 months. In the feedback from the study, it was captured that most of the participants used the tool daily, and over half reported that the tool identified impacts that they had not previously considered.
2.4 Impact Analysis Techniques

2.4.1 Program Slicing

Program slicing was first introduced by Mark Weiser [16] as a way of isolating the parts of a program that relate to a given program point. A given program slice is an isolated, executable section of a program. By examining a program slice, a developer can understand the relationships between variables and statements, which can aid in debugging and program optimization.

Silva [25] surveyed previous work performed using program-slicing techniques, and described and compared each of the techniques. Two of the techniques analyzed are static slicing and dynamic slicing, which were first introduced by Weiser. Some of the notable slicing-related vocabulary are as follows:

- **Static slicing**: Using program slicing according to criterion, but independent of program execution
- **Dynamic slicing**: Using program slicing with specific execution, to examine real-time program behavior
- **Backward slicing**: Identifying statements that could potentially influence the slicing criterion
- **Forward slicing**: Identifying statements that are potentially influenced by the slicing criterion
- **Relevant slicing**: Starting with the dynamic slice, separating out the statements that actually affect the slicing criterion
- **Decomposition slicing**: Identifying the statements in a program that are needed to compute a given variable

Program slicing can also provide insight into cohesion and coupling metrics. Coupling is the degree of mutual interdependence between software modules. While it is typically calculated by measuring the information in and out of the given modules, Harman et al. found that using program slicing results in a more precise calculation [26]. Briand [27]
also investigated the use of coupling measurement as a means for discovering the ripple effect throughout an object-oriented program. Slicing can also provide metrics on cohesion, the degree to which elements of software modules belong together [28] [29]. Meyers and Binkley [30] presented an empirical study for coupling and cohesion metrics using program slicing.

In addition to coupling and cohesion measurement, there are many other practical uses for program slicing, such as refactoring [31], parallelization, testing, debugging, program integration, and reverse engineering [32].

Binkley [33] carried out an empirical study of program slice size. It was discovered in the study that the average program slice contains almost one-third of the entire program size, for programs written in the C programming language. A smaller average program slice is generally more desired, since this indicates that program components are less tightly coupled together.

Tip [32] presented a survey of different program slicing techniques, and compared the accuracy and efficiency of the methods. To classify the techniques, the following programming languages features were used: procedures, unstructured control flow, composite variables/pointers, and concurrency.

### 2.4.2 IA and Repository Mining

Zanjani [20] created a method for mining software repositories to facilitate impact analysis. Techniques for information retrieval and machine learning were utilized, as well as data from Mylyn (which monitors programmer activities within Eclipse) to capture data about entities that were previously interacted with. This approach was found to result in better recall gains than other SCM mining techniques.

### 2.5 IA Tool Features

There are many different kinds of capabilities offered by impact analysis tools. Some tools offer more than one of the following capabilities:
• **Visualization of Program Slice or Impact Set** - This gives developers the ability to identify a program slice given a specific point in a program. Many tools that offer this capability have a way of displaying the program slice to the user [34] [35] [36].

• **Dependency Visualization** - This creates a dependence graph as a visual aid to the developer, to better understand the relationships between modules and functions [37] [8].

• **Repository Integration and Mining** - These tools hook into source code repositories in order to perform historical impact analysis [38] [21] [39].

• **Regression Test Selection** - Given a set of incoming changes, these tools will identify a set of affected regression tests for execution [40] [7].

These features offer the developers incentives to pick certain tools over others. Of course, it depends on what the developer’s goals are in performing impact analysis. It may be possible that the ability to perform regression test selection is more valued than a more efficient underlying IA algorithm.
Chapter 3

Method

3.1 Overview

This section will describe the method that will be used in order to perform the survey of the chosen impact analysis tools. This process will involve a systematic literature review, described by Kitchenham [41].

The purpose behind carrying out a systematic literature review is to gain a clear perspective of the impact analysis tools that currently exist. Our aim is not just to identify the tools that exist, but understand the capabilities offered by the tools and analyze the distribution of selected tool properties.

3.2 Systematic Literature Review

For a systematic literature review, it is important to have research questions identified to drive the methodology. The following research questions have been developed:

- RQ1: What tools for impact analysis have been developed?
- RQ2: What capabilities and features are offered by impact analysis tools?
- RQ3: What opportunities exist to improve impact analysis tools in order to aid developers in software maintenance?
The following actions will be performed as part of the systematic literature review to answer the review questions above:

1. **Literature search**: Databases are searched using keywords for relevant articles and publications

2. **Study selection**: The articles are further narrowed down according to inclusion and exclusion criteria

3. **Data Extraction and Analysis**: The content of the articles is analyzed and categorized in order to answer the research questions

### 3.3 Literature Search

The following databases were used to aid in the literature search, in order to establish an initial set of relevant articles.

- ACM Digital Library (http://dl.acm.org/)
- IEEE Xplore Digital Library (http://ieeexplore.ieee.org/)
- Springer Link (https://link.springer.com/)

Some of the query/search terms used in these databases included ‘impact analysis tools’ and ‘program slicing tools’.

In order to provide a more comprehensive view of the availability of impact analysis tools, there were some selected commercial tools that were included in the literature search results as well. These were discovered by searching for ‘impact analysis tools’ through the Google search engine.

### 3.4 Study Selection

To further refine the set of selected articles to a selection more useful for this study, some inclusion and exclusion criteria were established. Firstly, articles that were focused on
discussing or presenting a tool were considered. In some cases, the tool was developed in order to showcase a particular impact analysis or program slicing technique; this was allowed. Additionally, in some cases an impact analysis tool was mentioned by a paper and the primary paper was hosted elsewhere. Tools without identifying names were not considered.

Additionally, it is important to distinguish named “tools” from named “techniques”. The difference identified by this paper comes down to intent. In the software world, a “technique” is a way or method of carrying out a particular task, and a “tool” is a software application that implements techniques. A technique is independent of programming language or platform. A tool may implement one or more techniques, and is a vehicle for bringing techniques and algorithms to users, ideally in usable and intuitive ways.

In order to further refine the list of selected studies, our definitions of “software maintenance” and “software evolution” had to be made clear. While some tools did not specifically mention improving software maintenance or software evolution as an overarching goal, they may certainly be used towards that end, such as tools for regression test selection or coupling measurement. Additionally, some tools used impact analysis or slicing techniques, but for the purposes of post-analysis, such as history slicing on software repositories. Therefore, the following criteria were used:

- **Inclusion Criterion 1**: Utilizes an impact analysis technique
- **Inclusion Criterion 2**: Able to be used in the “pre-submit” stage of development, before changes are officially integrated into the main baseline
- **Inclusion Criterion 3**: Utilized at the code level (as opposed to impact analysis techniques used at the service or component level)
- **Inclusion Criterion 4**: A goal for the tool is to help developers understand the impact of a change before the change is made
- **Inclusion Criterion 5**: Conference paper or articles must be written in English
- **Inclusion Criterion 6**: The tool must be able to perform analysis on Java, C, or C++ programs
There are many tools that can be classified as software maintenance or software evolution tools. For example, one of the applications of program slicing is for measurement of cohesion and coupling [29]. While this does give some insight into the quality of the code, it does not directly aid the programmer in understanding the extent of the impact of changes made.

These criteria also exclude static analysis tools, such as Checkstyle and Findbugs. These tools are useful in identifying problem areas and potential bugs in application code, but they give the developer no information on the estimated impact of implemented changes.

Additionally, only conference papers and articles written in English were selected from the literature review.

### 3.5 Data Extraction and Analysis

For the data extraction and analysis phase, each of the selected articles and tools were documented, and a list of properties were checked against each one. These properties pertain to the features and capabilities offered by the tool, and information about the tool itself. The full set of data collected was detailed in a spreadsheet, and can be found in Appendix A.

#### 3.5.1 A Framework for Classifying Impact Analysis Tools

To help with the data extraction process, a framework had to be developed to better classify tools that perform impact analysis. To our knowledge, a current framework does not yet exist. There are several frameworks for classifying and comparing impact analysis techniques [4] [12] [19] [22]. These taxonomies and frameworks helped serve as an inspiration for this classification framework. It is important to note however, that the nature of techniques and tools are different. In our framework, we are focusing on the qualities of the tool itself, rather than the techniques employed by the tool.

This framework is divided into the following sections: general tool information, tool accessibility, impact analysis type, impact analysis technique, and tool features and capabilities.
3.5.1.1 General Tool Information

Some general high-level information was collected about each of the tools. This is to get a general idea of when the tool was developed and how the tool may be used. The authors of the tool and the database the original article was published in (if applicable) was also recorded.

The following general information properties were extracted for each tool:

- The year the tool was created
- The year the tool was last updated
- The programming languages applicable to the tool

If the year that the tool was last updated could not be determined (particularly if no further information of the tool could be traced), then “N/A” was recorded, rather than assuming development had ended. It is possible that there may be some inaccuracies in the data extraction for this property, particularly for tools that are not open source, since many tools are not well publicized.

It is assumed that for commercial tools, the year that the tool was last updated is the current year. Commercial tools are generally expected to be updated with the latest versions of languages and frameworks.

3.5.1.2 Tool Accessibility

Next, the accessibility of the tool should be determined. It can be argued that an inaccessible tool is an unusable tool. Therefore, it is important to know if the tool is available for external use, and how it is made available for others to use.

The properties used to record accessibility include the following:

- Whether the tool is available for external use
- Whether the tool is available commercially
- Whether the tool is open source
To determine if the tool was available for external use, if it was not explicitly stated in the given publication, online research was performed to determine if the author(s) or university involved had posted any additional information on the tool.

### 3.5.1.3 Impact Analysis Type and Technique

The impact analysis technique employed by the tool is also investigated and classified. This classification framework will not go in-depth into the underlying details of the technique (we will leave that to the designated IA technique frameworks and taxonomies). However, the type of technique used should be made known to a developer looking to select a tool, since this can result in different impact sets and granularity of results.

The properties used to record impact analysis type and technique include the following:

- Whether the tool employs static impact analysis, dynamic impact analysis, or both
- The type of technique utilized by the tool, using the suggested techniques outlined in Lehnert’s taxonomy for Criterion 3 \[19\]
  
  - Program Slicing
  - Call Graphs
  - Execution Traces
  - Program Dependence Graphs
  - Message Dependence Graphs
  - Traceability
  - Explicit Rules
  - Information Retrieval
  - Probabilistic Models
  - History Mining

### 3.5.1.4 Tool Features and Capabilities

Next, the different capabilities and features were then determined. It was observed that many of these tools presented similar features. These common features were included in
the data extraction properties to better understand the types of impact analysis tools that currently exist. The features are:

- able to display a program slice or impact set
- able to display a relevant dependence graph
- able to utilize software configuration management (SCM) mining in order to better understand the impact
- able to perform regression test selection

These features may be considered as enhancements to the tool that may entice users to choose one tool over another. Not all tools have extra features or enhancements (only the impact analysis technique is offered), and some tools have multiple features.

### 3.6 Usability Inspection

In addition to performing a systematic literature review, an informal usability inspection will also be performed on a couple selected impact analysis tools. A usability inspection is a cost-effective method for assessing the usability of a particular tool and identifying significant user experience issues early. The inspection carried out follows the usability heuristics developed by Nielsen [42], which are as follows:

1. Visibility of system status
2. Match between system and the real world
3. User control and freedom
4. Consistency and standards
5. Error prevention
6. Recognition rather than recall
7. Flexibility and efficiency of use
8. Aesthetic and minimalist design

9. Help users recognize, diagnose, and recover from errors

10. Help and documentation

Due to the availability and the age of some tools, it was not practical to perform a comprehensive comparison of all of the tools. Many tools were available for external use, but some required commercial licenses. Some tools were not able to be installed, due to evolving technologies and lack of tool updates.

For each of the tools analyzed, the Apache Commons Lang\(^1\) (commons-lang3) project was used to perform the usability inspection. As an active and well-known open source project, Commons Lang makes a practical test subject to simulate pre-commit, incremental changes.

It is acknowledged that many of the tools selected were developed for primarily academic purposes and perhaps were not designed with usability as a motivating factor. These tools were nevertheless still inspected and analyzed in order to provide better feedback for future tool development.

There are several reasons to performing a usability inspection over usability testing. Because of the experimental nature of the tools selected, they are still considered in the early developmental stages. Therefore it may be unfair to perform a full usability test with end users since it may not be intended as a finished product. By performing an initial inspection and providing feedback, significant issues can be identified and fixed early. There are also much less resources required in order to perform a usability inspection, at the cost of relying on the experience of the evaluator.

### 3.6.1 Tool Selection for Inspection

In selecting tools for the usability inspection, it was discovered that many openly available tools are no longer compatible with current technologies (i.e., Eclipse Neon, Java 8/9). These tools were removed from consideration, since there is little point in evaluating the usability of a tool that no longer makes an effort to stay compatible with the current technologies.

\(^1\)https://commons.apache.org/proper/commons-lang/
Two tools were hand-selected for the usability inspection: JRipples [5] and Impact-Miner [38]. These two tools were chosen for several reasons:

- They are compatible with Java 8 and Eclipse Neon
- They are open source (easily accessible) and well-publicized
- They were designed with the end user in mind

We aim to gather and provide usability feedback on these tools that may be useful for tool developers. By providing feedback on usability, we can potentially improve the user experience of these tools and help them achieve widespread use throughout the industry.
Chapter 4

Study

Using the selection criteria and methods described in Section 3, 18 academic papers were selected for review, describing 18 tools total. To provide a more comprehensive look at the variety of tools that exist, commercial tools were researched and considered as well. There were 2 commercial tools selected based on the criteria, bringing the total to 20 tools selected for the systematic literature review.

4.1 Tool Descriptions

In this section, we will provide some background information and details on each of the 20 impact analysis tools under observation. While we have a framework developed to classify each tool according to common properties and traits, each tool is unique in its own way.

4.1.1 Chianti

Chianti [40] is a change impact analysis tool available as an Eclipse plugin \(^1\) that identifies a set of atomic changes representing the difference between two versions of a given program. From this set of changes, a set of potentially affected regression tests are identified.

\(^1\)http://aria.cs.vt.edu/projects/chianti/
The method behind Chianti is capable of performing static or dynamic analysis, but dynamic analysis is used to illustrate the examples. The first step in the method is to compile a set of the atomic changes, which are categorized by the type of change, such as “added class”, “changed methods” or “deleted fields” (however, changes to individual program statements are not covered). Then, call graphs are constructed for each regression test, and the set of atomic changes is correlated with their counterparts in the constructed call graphs. As a final step, for each regression test, the set of relevant atomic changes is identified and correlated.

This tool is useful to developers attempting to analyze impact at the code level, depending on how fine-grained of an analysis the developer needs. A user interface is provided which shows the affected regression tests and associated relevant atomic changes. Clicking on an element in the results window will direct the user to the associated location in the source code.

4.1.2 CodeSurfer

CodeSurfer [34] was originally developed as a research tool at the University of Wisconsin, and was eventually packaged into a commercial tool offered by GrammaTech ². The tool offers a wide range of features to help developers better understand C and C++ code, including pointer analysis, call graphs, dataflow analysis, and of course, impact analysis.

A program dependence graph is constructed, which can be visualized on a user interface. Program slicing is also utilized (both forward and backward), with the relevant parts highlighted for simple analysis. These static analysis capabilities help the developer understand the complex relationships in their code and prevent errors before they are potentially integrated with a main baseline.

An API is also provided to developers, to allow them to directly integrate the analysis capabilities and take advantage of CodeSurfer’s program representations and AST functions.

Grammattech also offers another tool, CodeSonar, which has static analysis functions borrowed from CodeSurfer. CodeSonar helps developers identify potential bugs and

²https://www.grammatech.com/
inconsistencies, and also recommends optimizations with respect to performance and security.

4.1.3 EAT

Execute-After Tool (EAT) [43] was developed to evaluate the CollectEA technique, which aims to showcase the benefits of dynamic impact analysis over static impact analysis. Written for Java programs, EAT is made up of three components: an instrumentation module, a set of runtime monitors, and an analysis module.

The CollectEA technique is shown to be as precise as other dynamic impact analysis techniques, and is also space and time efficient. The granularity of the technique only goes down to the method level, but improvements can be made to capture changes down to the statement level. As far as the tool itself, the tool exists to demonstrate and gather data on the CollectEA technique.

4.1.4 eROSE

eROSE [39] (Reengineering of Software Evolution) is an Eclipse plugin 3 that integrates with CVS to mine the version history of a given project and guide the user into understanding the consequences of making (or not making) a change. Similar to ImpactMiner [38], eROSE has the ability to suggest changes that may be necessary to prevent errors, based on previous commits to version control. The suggestions are ranked by support (number of relevant historical transactions) and confidence level of the suggestion.

To perform the analysis, eROSE uses a server to collect change transactions from source control (CVS), and the relevant files are parsed. The next step is to mine rules from the transactions. The frequency of detected rules is also determined, and this is used to assign a confidence level to each rule. The more often a pattern or rule appears in the repository, the more confident the algorithm is that it is an appropriate suggested change.

When the programmer modifies code in Eclipse, eROSE applies any applicable rules and makes suggestions to the programmer. This is compared to functionality made popular

3https://www.st.cs.uni-saarland.de/softevo
by Amazon, where items may be suggested to you based on your previous purchases. This concept is introduced to software evolution, to help developers optimize their code evolution processes and prevent potential errors.

4.1.5 FaultTracer

FaultTracer [7] is an open-source change impact analysis tool implemented as a plugin for Eclipse. Utilizing the Java Abstract Syntax Tree (AST), this tool compares two given versions of a program and after extracting the edits, identifies any potentially affected regression tests. In many ways, FaultTracer is similar to Chianti [40] in goals and underlying techniques, but it claims to outperform Chianti’s ranking heuristic by over 50%.

FaultTracer is made up of three views: the atomic-change view, the extended-call-graph view, and the testing-debugging view. The atomic-change view allows developers to view and interact with the differences between two selected versions. In the extended-call-graph (ECG) view, developers can see the ECGs for each test, which can help facilitate understanding about the affecting changes. The testing-debugging view displays the final output of FaultTracer’s algorithms, including the affected tests, and a ranking of the relevant atomic changes for each test.

The developer must have two different versions of a project in the same workspace in order to run FaultTracer. A regression test suite must be provided as well.

4.1.6 Frama-C

Frama-C [35] is a static analysis platform for C programs which hosts an impact analysis plugin. The plugin performs slicing and dependency analysis to allow developers to visualize the impact of a given variable, and can be run either through a graphical user interface or from the command line. It is meant to be used for industrial-size C programs, but can be used on programs of any size and for any purpose.

The main function of the tool is to help developers understand how data flow throughout the program, by examining variable contents at different points of execution and also

---

allows slicing. The impact analysis plugin allows developers to see what would be impacted by a variable modification (also called forward slicing). Other plugins are offered as well, which perform source code metrics calculation, variable occurrence browsing, and data-flow browsing.

Frama-C also happens to be only tool in the set of selected tools that is openly available both as a commercial tool and as an open source tool. The community support for this tool is also well established, with an actively updated blog, wiki, public discussion forum, and bug tracking database.

4.1.7 Imp

Imp [36] is a change impact analysis tool that is available as a plugin for Visual Studio to analyze programs written in C or C++. Previous work on static program slicing and dependence clusters [33] was noted as related to the work carried out by the Imp plugin. Imp aims to improve on the results carried out by Binkley et al. by addressing the performance and accuracy issues with static program slicing. CodeSurfer [34] was used to apply the static program slicing, and integration with version control (Microsoft Team Foundation Server) was also built into the tool.

Imp can be used for a variety of different scenarios, such as dependency analysis, “what-if” analysis, regression testing, and risk analysis. Impacts are highlighted in the Visual Studio editor and a summary of the analysis is presented to the developer in a popup. While it is built over the analysis techniques used in CodeSurfer, Imp claims to offer improvements in performance and accuracy.

4.1.8 ImpactMiner

ImpactMiner [38] is a change impact analysis tool \(^5\) that utilizes source code repository mining and offers three analysis techniques: Information Retrieval (IR), Dynamic Analysis (Dyn), and Historical Analysis (Hist). Implemented as an Eclipse plugin, ImpactMiner can analyze Java programs and using the integration with Subversion (SVN), notify the developer of other methods that are typically edited in conjunction with the currently identified changes.

\(^5\)http://www.cs.wm.edu/semeru/ImpactMiner/
Developers have the options to choose one of the three techniques (IR, Dyn, or Hist), or a combination of the three techniques to perform impact analysis. IR is the easiest method for developers to perform, but may not offer the best performance or accuracy. Dit et al. [38] found that performing analysis using multiple techniques resulted in much better accuracy than using a single technique by itself.

Developers also have the option to configure association rules to refine the analysis results, and seed the repository mining algorithm to get different results.

### 4.1.9 ImpactViz

ImpactViz [37] is an Eclipse plugin that allows developers to visualize class dependencies, including information mined from the source code repository (SVN). This capability can help the developer pinpoint the source of an error, and understand the potential impact of a given set of changes. The tool organizes the mined results into change impact regions, and allows the developer to zoom into areas of interest, enhanced with the history of changes for each particular class.

The tool uses call graphs representing the method call graph and relationships between classes in preparation for impact discovery. These graphs also form the basis of the visualized color-coded graphs that form the main function of the tool. Developers may use filters to trim the graph to a desired size and scope, and interact with the graph, to trace program dependencies and program flow. These features are meant to help developers in the debugging process by narrowing down the source of an error. Integration with version control (SVN) allows analysis to start from a known bug-free state, and traverse through versions until the source of the error is discovered.

### 4.1.10 Impala

Impala [21] is an Eclipse plugin that utilizes data mining algorithms to perform impact analysis on Java programs, before the changes are executed. Comparing two versions of a program, Impala generates a dependence graph and creates a change set of all the detected changes, and potentially impacted entities. Impala is integrated with CVS, a version control tool, that allows for the ability to traverse through the project history and understand the detailed evolution of each class.
The tool generates call graphs for the given program, and changes are classified into different types, such as “adding/removing classes or methods” or “changing the visibility of classes or methods”. The impact algorithms are then applied to get the affected changes.

Impala strives to improve on the limitations of previous static impact analysis algorithms, which don’t produce the most accurate results. By focusing on improving the precision and recall of the algorithms, a higher accuracy can be achieved.

4.1.11 Indus

Indus [44] is an open source 6 Java program slicer tool that provides analysis methods to help developers evolve Java programs. There are two main functions of the tool: a collection of static analysis capabilities, and the Java program slicer.

The tool provides a user interface that allows developers to specify slicing parameters such as the slice type (forward, backward, or complete) or whether or not to search for an executable slice. While a GUI is available, Indus is meant to be used as a library. Kaveri [6] is one of the best well-known tools that implements the Indus API.

4.1.12 JArchitect

JArchitect [8] is a commercial 7 static analysis tool for Java programs. It offers a wide range of features, including code quality metrics, trend monitoring, diagramming capabilities, and an interactive dependence graph. Developers can select an entity within their project, and JArchitect will display the dependence graph showing potentially affected projects, packages, methods, or fields. Other types of graphs are able to be generated as well, in order to understand the relationships between elements, such as a coupling graph, path graph, all paths graph, and cycle graph.

To perform impact analysis, developers can view the generated dependency graphs to get an idea of the program relationships to make informed decisions about change propagation. JArchitect also generates a dependency structure matrix, which can be used

---

6http://indus.projects.cs.ksu.edu/
7http://www.jarchitect.com/

30
to gather data about coupling between entities. It should be noted that the impact analysis performed offers a granularity down to the method level.

JArchitect is available under two different types of commercial licenses. Developers also have the option to choose a free 14-day trial.

4.1.13 JRipples

JRipples [5] is an Eclipse plugin ⁸ that aims to supports developers with the tasks of change propagation and incremental change. Made up of three components (a parser, a database with organizational rules, and a user interface), it analyzes the dependencies within a Java project and returns a result set of potentially affected classes for the developer to review. The developer then reviews the results, one by one, and marks them as “Impacted” or “Visited”.

JRipples implements a combination of a dependency search technique and information retrieval that is termed DepIR to perform the impact analysis. This is meant to mimic the actions of a programmer performing impact analysis. When the analysis is complete, results are shown either in a hierarchical view or a table view. From there, the developer can iterate through the changes to determine what needs to be fixed.

Multiple analysis modes are offered. The default is Concept Location, but the developer may also choose from Impact Analysis or Change Propagation. Apache Lucene (text search engine library) and grep (regular-expression-enabled querying) searches are available, to help developers refine their searches.

There are a host of other features offered as well. Developers may choose to save or load program states, for more in-depth analysis. The results of the analysis can also be exported. A dependency manager also allows developers to customize dependencies for better analysis.

4.1.14 JSlice

JSlice [45] is an open source ⁹ Eclipse plugin that supports dynamic program slicing. The Kaffe Virtual Machine is used on the backend of the tool in order to execute the

---

⁸http://jripples.sourceforge.net/
⁹http://jslice.sourceforge.net/
Java program in preparation for retrieving the dynamic slice. The developer’s specified criterion is applied, and the results are translated back to the source code for visual inspection on the user interface. The tool also offers slicing flexibility to the developer, allowing the option to slice on the last execution or all executions of a given statement.

JSlice aims to improve over static slicing and traditional dynamic slicing by computing the “relevant slice”. In traditional dynamic slicing, the algorithm starts from the last executed bytecode and works backwards, collecting executed statements that may have directly contributed to the slicing criterion. To compute the relevant slice, dependencies are laid out in an “Extended Dynamic Dependence Graph”. This allows the developer to see statements in the slice that may have affected the slicing criterion. This can greatly help in debugging in situations where un-executed code may have affected the program execution.

4.1.15 JUnit/CIA

JUnit/CIA [46] is a change impact analysis tool implemented as an extension to JUnit within Eclipse. It also utilizes the impact analysis tool Chianti [40], in order to identify the program’s atomic changes, identify tests affected by the atomic changes, and determine any affecting changes for each of the tests. The tool is envisioned to be used in order to identify the cause of an unexpectedly failing test.

A classification schema was developed to help pinpoint the change that most likely contributed to the failing test. The classifiers are as follows:

- **Red**: High likeliness of being the source of failure
- **Green**: Low likeliness of being the source of failure
- **Yellow**: Between low and high likeliness of being failure-inducing

The goal for the tool is to be able to correctly identify the changes contributing to the failing tests and label them as “Red”. This draws the attention of the developer directly to the potential source of the problem.
4.1.16  Kaveri

Kaveri [6] is a program slicing tool offered as an Eclipse plugin 10. It is built off of Indus [44], a Java program slicing library, which can perform backward or forward slicing. Kaveri offers these slicing capabilities to the developer by abstracting away any unnecessary details involved in working with Indus, and simplifying the analysis process. Slice criteria can be chosen, and the resulting slice is then highlighted in the Eclipse editor.

Developers can utilize Kaveri to trace program dependencies to enhance their general program comprehension, prepare for change propagation, or identify the potential source of an error. The slice is displayed in the Eclipse editor, and the developer also has the option to perform additive slicing, or intersect slices based on multiple criteria.

4.1.17  Lattix

Lattix [9] is a commercial tool 11 that can be utilized by architects, developers, managers, and QA engineers to better understand the project architecture and impact of changes throughout the entire software life cycle. An Enterprise System Solution is offered that allows developers to perform impact analysis and understand how potential changes could affect the rest of the system.

In addition to supporting a variety of languages beyond Java, C, and C++, Lattix also has the capability to integrate with Klocwork 12, a popular static analysis framework for identifying potential failures.

4.1.18  REST

The Ripple Effect and Stability Tool (REST) [47] is a tool that implements an algorithm that calculates the ripple effect that a given change has on the rest of the program. Implemented for programs written in C, the tool helps developers with four main activities: determine possible impacts, identify known impacts, determine the stability of the software, and examine the requirements.

11http://lattix.com/
12http://lattix.com/products/klocwork
To calculate the ripple effect, propagation information is collected and organized into matrices. By connecting the information between the matrices, the algorithm can determined how values are propagated throughout different modules. With this information, developers can make informed decisions to safely design potentially invasive updates to programs.

4.1.19 SLICE

SLICE [48] is a dynamic program slicer for C programs. The tool supports both backward and forward slicing, and also offers four kinds of slices to aid the developer in the debugging process: data dependence, data closure, data and control closure, and executable.

To perform the dynamic analysis, a given input is applied to the program to be executed, and slices are constructed for any relevant variables in the program according to the slicing criterion. In the evaluation of SLICE’s performance, it was confirmed that the resulting dynamic slices are relatively small compared to the size of the program (as opposed to static slices, which are typically larger). Slice sizes can vary from program to program, but this can be affected by the slicing criterion chosen.

SLICE offers two modes to the developer: an interactive mode where the developer can choose a variable to slice on and see the results displayed, and a batch mode, where slicing operations can be performed on multiple programs with a summary of results.

4.1.20 Unravel

Unravel [49] is an open source program slicing tool for C programs available through NIST. The goals for the tool were to help developers with debugging and program comprehension by identifying program slices with a given slicing criterion. Additionally, the tool was evaluated in order to determine if the slices produced were of a useful size to the programmer, if the tool computed slices quickly and efficiently, and if the tool was considered to be usable by the average programmer.

13https://www.nist.gov/itl/ssd/unravel-project
The tool is made up of three parts: the analyzer, the linker, and the slicer. The analyzer parses the source code, and keeps track of the counts and types of source code elements, and the linker maps the relevant source code files to their respective configuration files.

By helping developers focus on relevant subsections of a program, code inspection and maintenance is made a more manageable task. Also, by specifying usability requirements for the tool, this allows developers to focus on comprehending the program rather than spending more time than necessary on how to use the slicing tool. This is especially beneficial to developers who may be unfamiliar with the program being analyzed and are seeking to increase their comprehension of the program.

4.2 General Information

There were 20 impact analysis tools selected for this review. Each of the tools were catalogued, categorized and analyzed according to the criteria previously identified.

These tools were developed over a range of years (1994-2017), and only a few have been recently updated. Below lists some of the basic information on the tools, such as the year the tool was developed, the last time the tool was updated, and the languages supported by the tool.

The year recorded for “Year Created” is the earliest year that published information was available for the tool. For example, if the paper introducing the tool was published in 2003, then the year recorded for “Year Created” was 2003, even if development on the tool may have begun prior to 2003. Of the commercial tools reviewed, each one is the flagship tool for the company. Therefore, if the creation year of the tool could be not determined, the year that the company was established was recorded.

In some cases, the year that the tool was last updated was unable to be obtained. In other cases, the year that the tool was last updated was discerned from externally available information. Tools available commercially are assumed to be continuously updated.

It is also worth noting that 10 of the selected tools were implemented as Eclipse plugins.
<table>
<thead>
<tr>
<th>Tool Name</th>
<th>Year Created</th>
<th>Last Updated</th>
<th>Language(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chianti</td>
<td>2004</td>
<td>2008</td>
<td>Java</td>
</tr>
<tr>
<td>CodeSurfer</td>
<td>1999</td>
<td>2017</td>
<td>C</td>
</tr>
<tr>
<td>EAT</td>
<td>2005</td>
<td>n/a</td>
<td>Java</td>
</tr>
<tr>
<td>eROSE</td>
<td>2004</td>
<td>2005</td>
<td>Java</td>
</tr>
<tr>
<td>FaultTracer</td>
<td>2012</td>
<td>2016</td>
<td>Java</td>
</tr>
<tr>
<td>Frama-C</td>
<td>2008</td>
<td>2016</td>
<td>C</td>
</tr>
<tr>
<td>Imp</td>
<td>2012</td>
<td>n/a</td>
<td>C/C++</td>
</tr>
<tr>
<td>ImpactMiner</td>
<td>2014</td>
<td>2015</td>
<td>Java</td>
</tr>
<tr>
<td>ImpactViz</td>
<td>2010</td>
<td>2011</td>
<td>Java</td>
</tr>
<tr>
<td>Impala</td>
<td>2008</td>
<td>n/a</td>
<td>Java</td>
</tr>
<tr>
<td>Indus</td>
<td>2005</td>
<td>2009</td>
<td>Java</td>
</tr>
<tr>
<td>JArchitect</td>
<td>2006</td>
<td>2017</td>
<td>Java</td>
</tr>
<tr>
<td>JRipples</td>
<td>2005</td>
<td>2016</td>
<td>Java</td>
</tr>
<tr>
<td>JSlice</td>
<td>2008</td>
<td>2008</td>
<td>Java</td>
</tr>
<tr>
<td>JUnit/CIA</td>
<td>2008</td>
<td>n/a</td>
<td>Java</td>
</tr>
<tr>
<td>Kaveri</td>
<td>2006</td>
<td>2014</td>
<td>Java</td>
</tr>
<tr>
<td>Lattix</td>
<td>2004</td>
<td>2017</td>
<td>Java/C/C++</td>
</tr>
<tr>
<td>REST</td>
<td>2001</td>
<td>2008</td>
<td>C</td>
</tr>
<tr>
<td>SLICE</td>
<td>1995</td>
<td>n/a</td>
<td>C</td>
</tr>
<tr>
<td>Unravel</td>
<td>1994</td>
<td>1996</td>
<td>C</td>
</tr>
</tbody>
</table>

Table 4.1: General Information on Selected Tools

*aOther languages supported include: .NET, Actionscript, Ada, Fortran, Javascript.

### 4.3 Tool Accessibility

The accessibility of impact analysis tools varied greatly. Some tools were developed for strictly academic purposes only, in order to showcase a particular slicing technique or algorithm. These tools did not appear to be available for external use, and in some cases this was explicitly stated. It is possible that the tools were made available when the accompanying article was published, and since have been taken down.

There are a handful of tools that are available with commercial licenses, such as JArchitect, CodeSurfer, and Lattix. There are a few tools that have been made open source,
<table>
<thead>
<tr>
<th>Tool Name</th>
<th>Externally Available</th>
<th>Commercial</th>
<th>Open Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chianti</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CodeSurfer</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>EAT</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eROSE</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FaultTracer</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Frama-C</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Imp</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>ImpactMiner</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>ImpactViz</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Impala</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indus</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>JArchitect</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>JRipples</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>JSlice</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JUnit/CIA</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kaveri</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Lattix</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>REST</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLICE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unravel</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4.2: Availability of Selected Tools

such as FaultTracer and JRipples. The source code for FaultTracer is available on BitBucket \(^{14}\), and the source code for JRipples is available on SourceForge \(^{15}\).

### 4.4 Analysis Types and Techniques

The tools covered a wide range of impact analysis types and techniques. Our survey has identified whether the tools selected use static analysis, dynamic analysis, or both.

\(^{14}\)https://bitbucket.org/zhanglm10/faulttracer/

\(^{15}\)https://sourceforge.net/projects/jripples/
Additionally, the type of impact analysis technique (as recommended by Lehnert [19]) is identified.

## 4.5 Capabilities

Each of the impact analysis tools selected boast a variety of capabilities, aimed at improving the programmer’s experience for effective maintenance and evolution.
As previously stated, some tools may offer none of the features, and some tools may offer multiple features. These are merely features that have been identified to be commonly offered features in impact analysis tools.

### 4.6 Usability Inspection

This section will cover the results of the informal usability inspection on a few selected tools, using the heuristics developed by Nielsen [42].
The usability inspection was carried out using the following technical environment:

- Mac OS X El Capitan
- Eclipse Neon 4.6.2
- Java 8

### 4.6.1 JRipples

JRipples [5] is an interactive developed to aid developers with change propagation and impact analysis. It runs as an Eclipse plugin, and is made up of three components: a parser, a database with organizational rules, and a user interface.

To run an analysis in JRipples, the developer selects a project and a main class, and has the option to change other parameters such as Incremental Change (ex: Concept Location, Change Propagation), Analysis (ex: Lucene Analysis, grep Analysis), Presentation (ex: Hierarchy Presentation, Table Presentation), and Dependency Graph (ex: Dependency Builder, Dependency Builder with polymorphic dependencies).

When the analysis is complete, a list of potentially affected classes is displayed, and the developer marks each one as Located, Propagating, or Unchanged.

The results of the analysis are as follows:
• The concept of marking impacts individually as Visited or Impacted may not be a realistic expectation for developers. Many development environments are fast-paced and even in the maintenance phase, time is valuable. [System vs. Real World, Flexibility and Efficiency]

• The meaning and relevance of some terms such as “Concept Location” and “Dependency Builder with polymorphic dependencies” may not be immediately recognizable or understood by the average developer. The meanings and outcomes of these choices should be made obvious to the user. [System vs. Real World]

• While there exists a well-documented user manual online, there is very little help documentation within the plugin itself. This could be enhanced by including help buttons or explanations for different features or options. [Help and Documentation]

• On the Advanced Options screen, it is unclear which combination of options is allowed, or if all combinations are allowed. If two options are incompatible and would cause an error if chosen, this should be made clear to the user. [Error Prevention]

• It takes at least 5 clicks minimum to run an analysis. It is best practice in user experience to reduce the number of clicks it takes to complete a task. [Flexibility and Efficiency]

• Occasionally when an analysis is run, no results appear in the table and no reason is given. A message with an explanation would help new users without having to spend time perusing documentation. [Error Diagnosis and Recovery]

4.6.2 ImpactMiner

ImpactMiner [38] is another impact analysis tool with the added benefit of mining source code repositories for a more intelligent analysis of potentially affected code. Developers have the option of choosing between three techniques based on their needs and resources: Information Retrieval (IR), dynamic (Dyn), and historical (Hist). ImpactMiner is available as an Eclipse plugin.

To perform impact analysis with ImpactMiner, the developer selects a class, and then choose “Impact Analysis” through the context menu. A form is shown where the user
## Figure 4.2: Results of an ImpactMiner IR Analysis

Inputs various analysis criteria and whether or not to perform repository mining. The results are shown in the “Results View” window show a list of potentially affected methods.

The plugin version used for this analysis is version 0.6.0.

The results of the analysis are as follows:

- The user is required to select a potential problem area to discover previous links and impacts, but this may not be the most useful method for a fast-paced industry. It would be beneficial to developers to discover impacts that may not have occurred to them. [System vs. Real World]

- It is not immediately clear to the user how to utilize the plugin. For example, it may not be intuitive to the user to right-click on a result and Link it as a concern for further analysis. [Flexibility and Efficiency]

- There is a noticeable lack of official documentation for the tool. There is a brief walkthrough of the tool online, as well as a video, but the tool would benefit from an official user’s manual describing each of the features and expected use. [Help and Documentation]

- There are 5 views associated with this plugin. For in-depth impact analysis this is useful, but may not be practical for everyday use by a developer, if the developer
is expected to utilize all the views to make full use of the tool. [System vs. Real World]

- Several misspellings were found on the user interface. For example, “slected” and “Sepcify” on the IA parameter form. User interfaces should have language checked for spelling and grammar to maintain professionalism and avoid ambiguities [Consistency and Standards]

4.7 Analysis

4.7.1 Overview

Overall, 20 impact analysis tools were analyzed and catalogued. Most of these tools were created for Java, but some tools support multiple languages. One tool, a commercial tool named Lattix, supports multiple languages. 16 A chart showing a distribution of programming languages support is shown in Figure 4.3.

The oldest tool in our survey is Unravel, created in 1994. There are four tools that are considered updated to the present year: CodeSurfer, Frama-C, JArchitect, and Lattix. These all also happen to be commercial tools. The tool with the longest observed lifespan is CodeSurfer, with a current lifespan of 18 years. A chart showing the lifespan ranges of the selected impact analysis is shown in Figure 4.4.

4.7.2 Tool Accessibility

Most tools have been made available externally, as opposed to merely developed for demonstration purposes and abandoned. Very few however are available as open source projects, and even fewer are available as commercial products. There is one tool that is available both as open source and with a commercial license, which is Frama-C. The distribution of tool availability is shown in Figure 4.5.

Making these tools available externally would go a long way towards bridging the gap between the academic and industry worlds. Many authors [38] [5] have noted the great

16 Other languages supported include: .NET, Actionscript, Ada, Fortran, Javascript.
advantages of using their tools, and how these tools would significantly improve incremental changes in the software maintenance process.

Packaging the tool for external use is a start. From there, tool developers can choose whether to make it available on a commercial license, or open source (or even both). A commercial license helps gives the tool credibility in the industry. Making the project open source makes it an attractive option for low budget projects, and also can inspire like-minded developers to contribute to the project and continue improving the tool.

### 4.7.3 Analysis Types and Techniques

All of the tools surveyed performed at least static or dynamic impact analysis. Three of the tools surveyed performed both types of analysis [40] [7] [38].

A variety of impact analysis techniques were found to be used, but most tools implemented the following three techniques: call graphs, program dependence graphs, and program slicing. It should be noted that some tools implemented multiple techniques, which were also counted in this data.
4.7.4 Capabilities

There are many capabilities and features offered by the selected impact analysis, and it was observed that a pattern of common features emerged. Some tools offered additional capabilities beyond identifying potential impact and aiding program comprehension. Other tools did not offer any of the identified capabilities, and merely offered a pure static/dynamic analysis capability. A distribution of the features offered by the impact analysis tools is shown in Figure 4.8.
**Figure 4.5:** Availability of Impact Analysis Tools

**Figure 4.6:** Analysis Types of Impact Analysis Tools
4.7.5 Usability Inspection

Two tools, JRipples and ImpactMiner were informally inspected according to Nielsen’s 10 Usability Heuristics. Both of these tools are well known and recognized as some of the notable impact analysis tools that are available for use today.

It is apparent from the results of the inspection that these tools may not be industry-ready. In some cases, there is a noticeable lack of documentation, and there is a significant amount of subject matter knowledge required to effectively use the tools.

In the case of JRipples, the developer is expected to walk through the potentially affected classes one-by-one, determine if they are affected or not, and assign a classification label. This is not a realistic expectation for developers in the fast-paced software development industry.

In the case of ImpactMiner, the user interface could benefit from more polish, and more intuitive guidance for a developer using the tool. Developers may be unaware of the particular benefits offered by the different impact analysis techniques. Additionally, it
may not be clear to the user on the purpose of the multiple views of ImpactMiner. Navigating between multiple views for a routine task may not be desirable.
Chapter 5

Discussion

The tools discussed in this paper have covered a wide variety of impact analysis tools in the academic world and the industry. Most tools were developed for research purposes, and there is a noticeable lack of sophisticated impact analysis tools available for commercial use.

Most impact analysis tools were developed solely for Java applications (and most of those were Eclipse plugins). This is not an unexpected finding, as Java is a very popular language (at the time of this writing, Java is #1 in the TIOBE index \(^1\), and C and C++ are #2 and #3 respectively). As the benefits of impact analysis are realized however, it would be beneficial to the software industry as a whole to ensure these tools and techniques could be generalized to other languages.

The longevity of tools also provides some interesting feedback. It was observed that commercial and open source tools tended to have a longer lifespan. This implies that if the tool is packaged and documented for commercial use, or simply made available on an open source sharing platform such as BitBucket or GitHub, this is beneficial to the longevity of the tool. This may not be practical in all situations, of course, since funding and schedules are typically limitations when it comes to developing software tools. However, making projects open source takes very little overhead, and can go a long way towards providing publicity for the tool.

\(^1\)https://www.tiobe.com/tiobe-index/
Static analysis techniques were also more prevalent than dynamic analysis techniques. While the two types of techniques can be used for different purposes and towards different goals, dynamic analysis is typically recognized as able to provide more accurate results than static analysis, due to working with the actual program execution. It is important to note however, that not all techniques are created equal, and many static analysis techniques offered performance and accuracy improvements over previous versions of static IA techniques.

Call graphs, program dependence graphs, and program slicing were the most common identified impact analysis techniques in the selected tool set. Some of these are variations on the standard technique that offer certain optimizations (such as extended call graphs and dynamic program dependence graphs). Nevertheless, there are many tried and true impact analysis techniques [12] [19] [32], and the absence of greater variety of techniques reveals a risk for these techniques being lost. It has been noted [12] [13] that there is an importance for tools to be developed around techniques to improve usage of the technique.

5.1 Recommendations for Tool Longevity

One of the issues observed in this literature review was the lack of tools that are actively updated to the present day. While funding may be primary reason (and an understandable reason) for lack of updates, there are some best practices that can be implemented when it comes to general experimental tool development.

- **Make the project open source**: If possible, make the project available on an open source license and allow others to contribute. This keeps the project alive even after research may be finished and allows the software development community to benefit from advances that have been made.

- **Keep the project well documented**: Host a project web page and online documentation. This can be integrated into the open source platform host (ex: BitBucket, GitHub) if necessary. Keeping a web page online allows others to easily search for it and understand what the project was about. Having a detailed user’s manual also gives the project a professional edge and can help users who may not find the tool intuitive.
• **Follow user experience best practices**: Having good user experience designed into the tool from the beginning can have long term benefits. While the tool may be used purely for research or demonstration purposes, an intuitive interface can draw in potential contributors after the project is completed. Additionally if the tool is intended to be used by the everyday developer, effective user experience is a must. Even if the tool offers sophisticated, performance-enhancing algorithms, developers will tire of using it if it is too cumbersome to navigate or if it is too time-consuming to fit into their typical development process.

### 5.2 Future Work

The results from the literature review and usability inspection have uncovered many promising areas of future work. To build on the results of the informal usability inspection, this area of research would benefit from a full usability analysis involving multiple users to fully understand the realistic needs of the present-day developer.

Some additional research could also be carried out on the tools identified in this paper to discover the true lifetime and status of the tool. It is possible that the tool may still be actively developed, but is not well publicized.

It was also observed that while many impact analysis tools exist, there are no open source impact analysis API libraries. There are libraries to analyze the Java Abstract Syntax Tree \(^2\), but no open source APIs to perform impact analysis. By repackaging the impact analysis algorithms implemented in the tools previously discussed into an open source API, this would go a long way towards making impact analysis easily accessible to the industry.

There is a noticeable lack of research surrounding the measurement of impact. Binkley \(^3\) presented an analysis and findings on a study of program slice size, and Stoterzer \(^4\) developed a classification method for ranking likeliness of being a source of failure, but we have yet to discover a technique that has devised a useful and dependable metric for extent of impact. Assigning a metric to impact results can save developers hours of analysis time by helping them answer the question, “Should I commit this changeset or not?”

\(^{2}\text{https://github.com/javaparser/javaparser}\)

\(^{3}\text{Binkley [33]}\)

\(^{4}\text{Stoterzer [46]}\)
Perhaps the most important opportunity of improvement, it is apparent that there is a desire and a need to bridge the gap between the impact analysis tools developed in academia, and the tools used by the software development industry. It should be a best practice for tool planning and development to lay out a roadmap and plan for the future of the tool even after the research is completed. While the research at the time is important, the follow-on usage and replication of the research is crucial in order to be truly beneficial.

5.3 Conclusion

A framework for classifying impact analysis tools for the purpose of software maintenance and evolution was developed and utilized in this paper to perform a systematic literature review, in addition to an informal usability inspection. The results of this review and analysis led the discovery of some opportunities to improve impact analysis tools for industry use.

Software maintenance and evolution will remain a crucial part of the development process, and the necessity for safe and stable code evolution is a critical goal for software architects, managers, and developers. Even on the small scale of changing code line by line, the maintenance effort can add up over time and make a significant financial impact.

With the discoveries being made in research tools, there is an opportunity for tool developers to migrate these beneficial and cost-saving capabilities to the industry. When academic research give empirical studies showing the tangible benefits of using these tools, there should be a plan in place to migrate these technologies to a more lasting venue.

By equipping developers with effective tools to understand impacts of potential changes, it is hoped that the overall errors introduced into the product will decrease, which prevents impacts to team productivity and the project schedule.
Appendix

This appendix contains the full data collected from the systematic literature review. The full table can be viewed below, with the following data about the selected tools:

- Tool Name
- Authors/Creators of the tool
- Database of the article (if applicable)
- Year tool was created
- Last Year tool was updated
- Programming Language(s)Supported
- Whether or not the tool is externally available
- Whether or not the tool is available on a commercial license
- Whether or not the tool is open source
- Whether or not the tool uses static impact analysis
- Whether or not the tool uses dynamic impact analysis
- The impact analysis technique implemented by the tool
- Whether or not the tool utilizes program slicing or displays a program slice
- Whether or not the tool displays a relevant dependency graph
- Whether or not the tool utilizes SCM mining to better understand the impact
- Whether or not the tool is able to perform regression test selection
<table>
<thead>
<tr>
<th>Tool Name</th>
<th>Authors</th>
<th>Database</th>
<th>Year Created</th>
<th>Last Upd.</th>
<th>Lang(s)</th>
<th>Ext. Avail.</th>
<th>Comm.</th>
<th>Open Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chianti[40]</td>
<td>Ren et al. Teitelbaum/</td>
<td>ACM</td>
<td>2004</td>
<td>2008</td>
<td>Java</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CodeSurfer[34]</td>
<td>Grammatech</td>
<td>ACM</td>
<td>1999</td>
<td>2017</td>
<td>C</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>EAT[43]</td>
<td>Apiwattanapong et al.</td>
<td>ACM</td>
<td>2005</td>
<td>n/a</td>
<td>Java</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>eROSE[39]</td>
<td>Zimmerman et al.</td>
<td>ACM</td>
<td>2004</td>
<td>2005</td>
<td>Java</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Imp[36]</td>
<td>Acharya and Robinson</td>
<td>ACM</td>
<td>2012</td>
<td>n/a</td>
<td>C/C++</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ImpactMiner[38]</td>
<td>Dit et al.</td>
<td>ACM</td>
<td>2014</td>
<td>2015</td>
<td>Java</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Impala[21]</td>
<td>Hattori et al.</td>
<td>IEEE</td>
<td>2008</td>
<td>n/a</td>
<td>Java</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>JArchitect[8]</td>
<td>CoderGears</td>
<td>n/a</td>
<td>2006</td>
<td>2017</td>
<td>Java</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>JUnit/CIA[46]</td>
<td>Stoerzer et al.</td>
<td>ACM</td>
<td>2008</td>
<td>n/a</td>
<td>Java</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Lattix[9]</td>
<td>Lattix</td>
<td>n/a</td>
<td>2004</td>
<td>2017</td>
<td>/C/C++</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>REST[47]</td>
<td>Bilal and Black</td>
<td>ACM</td>
<td>2001</td>
<td>2008</td>
<td>C</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SLICE[48]</td>
<td>Venkatesh</td>
<td>ACM</td>
<td>1995</td>
<td>n/a</td>
<td>C</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 1: Full Results from the Systematic Literature Review, Part 1
<table>
<thead>
<tr>
<th>Tool Name</th>
<th>Authors</th>
<th>Static</th>
<th>Dynamic</th>
<th>Technique</th>
<th>Prog Slice</th>
<th>Dep. Graph</th>
<th>SCM Mining</th>
<th>Test Sel.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chianti[40]</td>
<td>Ren et al.</td>
<td>✓</td>
<td>✓</td>
<td>Dynamic</td>
<td>Call graphs</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Teitelbaum/</td>
<td></td>
<td></td>
<td></td>
<td>PDG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodeSurfer[34]</td>
<td>Grammathe</td>
<td>✓</td>
<td></td>
<td></td>
<td>Prog. slicing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CollectEA (Execute-After)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EAT[43]</td>
<td>Apiwattanapong et al.</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eROSE[39]</td>
<td>Zimmerman et al.</td>
<td>✓</td>
<td></td>
<td></td>
<td>History mining</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Frama-C[35]</td>
<td>Cuq et al.</td>
<td>✓</td>
<td></td>
<td></td>
<td>Program slicing</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Imp[36]</td>
<td>Acharya and Robinson</td>
<td>✓</td>
<td></td>
<td></td>
<td>PDG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Impact Miner[38]</td>
<td>Dit et al.</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>IR, History mining</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Impact Viz[37]</td>
<td>Follett</td>
<td>✓</td>
<td></td>
<td></td>
<td>Call graphs</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Impala[21]</td>
<td>Hattori et al.</td>
<td>✓</td>
<td></td>
<td></td>
<td>Call graphs</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Indus[44]</td>
<td>Ranganath</td>
<td>✓</td>
<td></td>
<td></td>
<td>Prog. slicing, call graphs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JArchitect[8]</td>
<td>CoderGears</td>
<td>✓</td>
<td></td>
<td></td>
<td>PDG</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>JRipples[5]</td>
<td>Buckner et al.</td>
<td>✓</td>
<td></td>
<td></td>
<td>IR, PDG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JSlice[45]</td>
<td>Wang and Roychoudhury</td>
<td>✓</td>
<td></td>
<td></td>
<td>Dynamic PDG</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>JUnit/CIA[46]</td>
<td>Stoezer et al.</td>
<td>✓</td>
<td></td>
<td></td>
<td>Dynamic call graphs</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Lattix[9]</td>
<td>Lattix</td>
<td>✓</td>
<td></td>
<td></td>
<td>PDG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REST[47]</td>
<td>Bilal and Black</td>
<td>✓</td>
<td></td>
<td></td>
<td>Ripple effect</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLICE[48]</td>
<td>Venkatesh</td>
<td></td>
<td>✓</td>
<td></td>
<td>Prog. slicing</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Unravel[49]</td>
<td>Lyle et al.</td>
<td>✓</td>
<td></td>
<td></td>
<td>Prog. slicing</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 2: Full Results from the Systematic Literature Review, Part 2
Bibliography


