Comparison of Different Impact Analysis Methods and Programmer's Opinion — an Empirical Study

Gabriella Tóth, Péter Hegedűs, Árpád Beszédes and Tibor Gyimóthy
University of Szeged
Department of Software Engineering
Árpád tér 2. H-6720 Szeged, Hungary
gtoth,hpeter,beszedes,gyimothy@inf.u-szeged.hu

Judit Jász
MTA-SZTE
Research Group on Artificial Intelligence
Árpád tér 2. H-6720 Szeged, Hungary
jasy@inf.u-szeged.hu

Abstract

In change impact analysis, obtaining guidance from automatic tools would be highly desirable since this activity is generally seen as a very difficult program comprehension problem. However, since the notion of an 'impact set' (or dependency set) of a specific change is usually very inexact and context dependent, the approaches and algorithms for computing these sets are also very diverse producing quite different results. The question 'which algorithm finds program dependencies in the most efficient way?' has been preoccupying researchers for a long time, but there are still very few results published on the comparison of the different algorithms to what programmers *think* are real dependencies. In this work, we report on our experiment conducted with this goal in mind using a compact, easily comprehensible Java experimental software system, simulated program changes, and a group of programmers who were asked to perform impact analysis with the help of different tools and on the basis of their programming experience. We show which algorithms turned out to be the closest to the programmers' opinion in this case study. However, the results also certified that most existing algorithms need to be further enhanced and an effective methodology to use automated tools to support impact analysis still needs to be found.

Categories and Subject Descriptors I.2.2 [*Automatic Programming*]: Program modification

General Terms Experimentation, Human Factors, Algorithms

Keywords Change impact analysis, software dependencies, JRipples, BEFRIEND, software co-change, SEA, static slicing, call graph, Java

1. Introduction

During change impact analysis in software evolution processes, the software engineer tries to assess the extent of potential impact that a particular (intended or performed) change to a software system may have [6]. A change can have impact on various software project artefacts like requirements, software components (at all levels, including the source code itself), test cases, etc., but usually also have secondary effects on business risks, maintenance costs, market value, and so forth. In this work we focus on impacts on software components. The incremental change model [6, 26] establishes a framework for change impact analysis by distinguishing between *concept location*, *impact analysis*, *change propagation* and other necessary tasks like *regression testing*. During these phases of the incremental change process, the initial and secondary impacted components are identified, in a sense estimating the *ripple effect* of the initial change request [10].

The identification of such impacted components basically means finding and following the *dependencies* between software components. But what actually are software dependencies? They can be described (not defined!) in many ways. Determining the relevant dependencies for the task at hand is basically a *program comprehension* problem, and since it depends on the particular situation, it cannot be relied upon a specific program analysis algorithm. A specific, externally observable dependency is always a reflection of the semantics behind, or in other words the *intentions* of the software developer [25]. This means that finding the most relevant dependencies is difficult, and is essentially a creative mental task.

Having said that, tool support is extremely desirable in impact analysis and for determining the dependencies, as it is generally true for any program comprehension task. However, it is naïve to think that there might be one ultimate algorithm that fits all tasks. Over the time, different classes of such algorithms have been developed by researchers. One of the most general ones is the class that we call *computation based*, in which the computation relationships between program elements are followed, most notably program slicing, call graphs, and other similar approaches. Another approach is to analyze different software artefacts to find possible semantic links like the ones based on historically consistent co-change of program elements [17, 18, 28]. A notable property of the methods is that depending on their nature, the dependencies may be found with very different levels of recall and precision with respect to the – only theoretically existing – *relevant* dependencies for the actual task. What we can derive from the above is that there is no universal definition for dependency, only algorithms that approximate dependencies. But then the question arises, which algorithm is most suitable in a specific situation—which one finds the relevant

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PPPJ '10, September 15–17, 2010, Vienna, Austria.
Copyright © 2010 ACM 978-1-4503-0269-2...$10.00

dependencies best? In other words, how do the algorithms differ in terms of the ability to predict relevant dependencies?

A possible way to answer the questions above is to involve programmers, and hear their subjective opinions based on expertise and experience in program comprehension. In this paper, we present such an experiment. We wanted to know what is the difference between some well-known algorithms and programmer's opinion, and hence we conducted a case study. We chose a compact, easily comprehensible experimental Java software system, simulated program changes and a group of programmers who were asked to perform impact analysis with the help of different tools and based on their programming experience. We applied several well-known algorithms (callgraph, program slicing, static execute after, historical co-change), a Java framework for change impact analysis embedded in a development environment (JRipples [12]), and used a tool to evaluate the results given by the programmers (BEFRIEND [16]).

The paper is organized as follows. First, we overview related work in Section 2, then we continue with a motivating example and establish the main research questions in Section 3. In Section 4, we overview the methods that we compared in the study, while Section 5 is devoted to the detailed description of the experiment performed. Section 6 discusses the results of the study, and Section 7 lists any possible threats to validity. Finally, we conclude in Section 8.

2. Related Work

The comparison of different impact analysis techniques is not a novel problem in the field of software maintenance [5]. One reason that these comparisons are very difficult could be the lack of an accepted definition of impact analysis. Although the definition of Arnold and Bohner [5] for impact analysis is accepted in general, the concept is still not included in the latest version of the IEEE Standard Glossary of Software Engineering Terminology [3].

Arnold and Bohner give a framework for the categorization of IA techniques based on the comparisons of the SIS (Starting Impact Set), EIS (Estimated Impact Set) and AIS (Actual Impact Set), but the general comparisons use only the following factors:

- weakness and strength,
- cost-precision tradeoffs,
- precision, space cost, and time overhead,
- Cohen-kappa value [13], which gives an overall indication of the goodness of the prediction.

Many papers in the literature compare the existing impact analysis techniques with the help of another technique. For example, Orso *et al.* compared PathImpact and Coverage impact analysis [24] and gave the relative costs and benefits of the techniques in practice. Breech *et al.* [11] compare online dynamic impact analysis with the PathImpact and Coverage impact analysis. As a particular application of the impact analysis, Bible *et al.* made a comparative study of coarse- and fine-grained safe regression test-selection techniques [9] where two test selection tools were compared with each other.

The methods which take the programmers' opinion about the software impacts into account are much more interesting for us, but unfortunately the number of such paper in the literature is very small. One of the first papers is the work of Lindvall and Sandahl [23]. In this work the authors quantified how well experienced software developers predicted change by conducting RDIA (Requirement-driven impact analysis) where RDIA in their case was the general activity of predicting changes based on the change request. Contrary to our experiments they compared and evaluated the predictions by examining the changes of the concrete implementation. Their results show that the impacted set predicted by the programmers without any help is generally correct but underestimated, which is basically in alignment with our findings.

To describe the empirical study, we applied a widely used approach, the Goal-Question-Metrics (GQM) paradigm by Basili *et al.* [7]. This approach has three levels: determine organization goals first, generate questions, whose answers can determine if the goals are met, and find measurements that can answer the questions. We emphasize our goal, define 5 questions and 19 metrics which are associated with the questions in order to answer them. Moreover, we used another guideline by Kitchenham *et al.*[22] to improve the quality of performing and evaluating our empirical research.

3. Motivation

To show that different algorithms can compute really different impact sets, consider the example in Figure 1. Any average programmer can observe that class `BB` is dependent on class `B` due to the inheritance, and class `BB` is dependent on class `A` since the computation of the `get` method of `A` determines the value of the `_x` member attribute of class `BB` through the method `hiddenAndMagicAssign` in class `C`.

However, if we compute the dependencies of these classes with different impact analysis algorithms we can get really different results. Using a call graph [27] only, classes `A`, `B`, and `BB` are dependent on class `C`, and there are no other dependencies in this example. If we determine the impact sets of the particular classes with the help of Static Execute After relations [8] we get that class `A` is dependent on class `C`, classes `B` and `BB` are dependent on classes `A`, `C`, `BB`, and that class `C` is dependent on classes `A`, `B` and `BB`. Finally, if we compute the dependencies of these classes using forward program slices [19], we get that the impact set of `A` contains classes `B` and `BB`, furthermore the impact set of `BB` contains only class `B`, and that the impact set of class `C` contains all the other classes.

```
class C {
  public void hiddenAndMagicAssign() {
      A a = new A();
      B b = new B();
      BB bb = new BB();
      int result = a.get();
      b.set(result);
      bb.set(result);
    }
}

class A {
  public int get() {
      return _someComputation();
    }
}

class B {
  public int _x;
  public void set(int p) {
    // intentionally do nothing
  }
}

class BB extends B {
  public void set(int p) {
    _x = p;
  }
}
```

Figure 1. Motivating example

This example shows how different the behavior of related impact analysis algorithms and, moreover, the opinion of an average

programmer – who was able to take into account other semantic properties like comments in the program – can be.

Our **goal** is to compare the different impact analysis methods and the programmers' opinion. This drives us to raise the following research directions:

- What is the relationship between different kinds of static dependencies, like the ones based on program slices, SEA, co-changing files (classes) in version control repositories, or call graphs, for instance? How different are the so-computed dependency sets?
- What is the relationship between different kinds of dependencies if we consider the dependencies that were evaluated by programmers, and filter out the irrelevant dependencies according to them? How much improvement in the correctness of the algorithms can be observed after such a filtering?
- How much the programmers' opinions change after seeing the results of different impact analysis algorithms over their initial opinions when using no specific algorithms but a general impact analysis tool within the development environment?

According to these research directions, we determine the following questions:

Q1: Do the different algorithms and the programmers identify the same dependencies?

Q2: What is the relationship between the different kinds of dependency sets?

Q3: According to the precision and recall values of the algorithms, what is the rank of the algorithms?

Q4: Do the programmers identify the dependencies with the same confidence levels?

Q5: How much the programmers' opinions change after seeing the results of different impact analysis algorithms over their initial opinions?

4. Impact analysis methods

To answer the questions we applied different impact analysis algorithms. In this section we overview the investigated algorithms.

The algorithms have been implemented within the JRipples Java tool and framework [2, 12], which is an integrated tool in the Eclipse development environment supporting incremental change, and supports relevant program analysis for the programmer and manages the organization of the steps that comprise the impact analysis and the subsequent change propagation. JRipples is based on the philosophy of 'intelligent assistance', which requires cooperation between the programmer and the tool itself. First, the tool analyzes the program, keeps track of the inconsistencies, and then automatically marks the classes/methods which are necessary to be visited by the programmer. Its main advantage is that it covers the algorithmic parts of the change propagation which are often difficult or error-prone for humans.

JRipples is implemented in Java as a plug-in for the Eclipse platform and analyzes Java source code. The tool consists of three Java packages that implement the parser, the database with organizational rules, and the user interface. The parser has two default analyzers, which analyze the project and extract class or method level dependencies of several kinds: for example, class A depends on B if class A refers to class B in a definition of a data member, local variable, argument, data cast; if A refers to class B's static members (static methods or static data members); if A inherits from B; or if A implements interface B. JRipples shows interactions between classes/methods. Classes A and B interact if either A depends on B or B depends on A.

Since it is easy to adapt other analyzer (algorithms) into JRipples, it can serve us as a framework to determine several kinds of static dependency. JRipples itself supports analysis on the granularity of classes and methods. In our experimental study, we determine dependencies on the granularity of methods (except historical cochange) but we lift the granularity to class level for the ability of the comparison.

We implemented the following algorithms within JRipples:

- **Callgraph**. We build a directed graph that represents calling relationships between methods [27]. The graph is built based on AST computed by Eclipse JDT.
- **Static slice**. We apply static forward program slicing [19] (considering data and control dependencies) to determine the impacts of the method modifications. The static forward slices were computed by the Indus Java static slicer API [20]. A slice is determined for every statement.
- **Static Execute After (SEA)**. According to the definition of SEA dependencies, method B depends on method A if and only if B may be executed after A in any possible execution of the program [8, 21]. The computation of SEA relations is an efficient analysis algorithm, which is able to produce conservative impact sets on method level. The determination of these relations is based on the ICCFG representation [8] of the program. We built the ICCFG graph based on AST and computed by Eclipse JDT. We implemented the SEA algorithm on this graph and determined all the method pairs which might be in a SEA relationship.
- **Co-changing files retrieved from SVN repositories**. Some dependencies are not written down in the code; the software engineer only 'knows' which certain set of modules needs to be changed [4, 18, 28] to make a certain type of change. In these cases, to derive the set of source files impacted by a proposed change request, we can use historical data stored in a versioning system, namely SVN. Since this way only the changed files can be retrieved, this analysis has only class granularity. A correlation value can be set between 0 and 1.0, if we would like to filter the found co-changed classes.[1] We determined two result sets, one with a 0.4 correlation value and one with a 1.0 correlation value. We chose the correlation value 40% because we would not like the union of the dependency sets to be too large, the number of the SEA relations and the dependencies determined by programmers were the most extended and finding co-changing classes with a 40% correlation value to get the same amount of dependencies.

Altogether, we have 5 kinds of dependency sets (call, static slice, SEA, co-changing 0.4, and co-changing 1.0 relations). Two of these result sets have method, two of them have class, and one of them has statement granularity originally but, of course, we lift all results to class level to be able to compare them. Since the callgraph determines the call dependencies only step by step, we compute the transitive closure of the call relations of each method.

5. Description of the Experiment

The experiment involving the participant programmers was performed in two stages (see Figure 2). First, the participants were asked to use JRipples in 7 different use cases to discover the impact set of the hypothetical changes in some particular methods of our chosen sample project. Secondly, for each scenario we stored their results together with the results produced by the specific algo-

[1] For example, if the correlation value is 0.4, it means that class A depends on class B if in 40% of the commits when A file changed, B file changed as well.

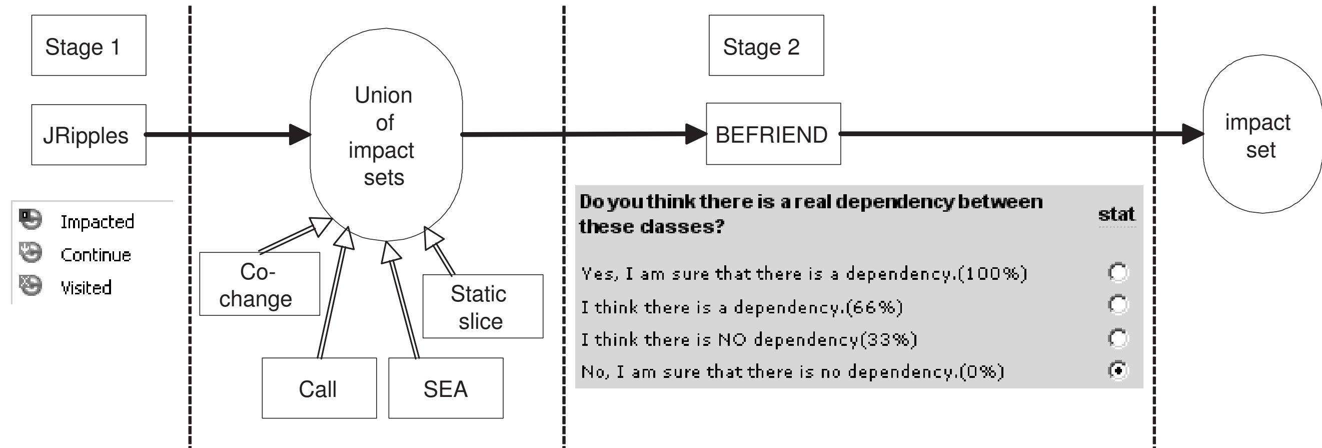

Figure 2. The architecture of the empirical study

rithms mentioned in the previous section in a common repository (BEFRIEND) that served as a control benchmark. Then we asked the participants to evaluate all of the stored dependencies individually. This way we were able to calculate valuable statistics about the usefulness of JRipples itself, the precision and the recall of the examined algorithms with and without respect to the programmers' opinion, and the rate of the changes in programmers' opinions made between the two phases. The following subsections provide detailed description of the above mentioned stages and their preparation.

5.1 Preparation for the first stage

First, we have set up a test environment. We needed a sample project in which the impact sets were defined according to the hypothetical modifications. When choosing the sample project we relied on the following considerations:

- The code is written in Java, since JRipples analyzes only Java code.
- Accessible SVN repository with extended history.
- Relatively complex code but not too large code size – since the Indus slicer could not produce slices for large programs due to the excessive memory consumption.
- Compatible with JRE 1.4., since the Indus static slicer works only on such kind of Java code.
- Code is unknown to the participants but it is easily comprehensible for the programmers.

According to these requirements, we have found an open source Java project called *ownSync*.[2] This is a small Java application to synchronize two folders (on different machines or on the same machine) in both directions. The main characteristics of the sample project can be seen in Table 1.

Nr. of classes	Nr. of methods	LOC	Not empty LOC	Nr. of commits
30	234	3666	3108	92

Table 1. The characteristics of the subject system (ownSync)

[2] http://sourceforge.net/projects/ownsync/

After selecting the sample project, 11 programmers with different qualification and experience levels were asked to contribute to our experiment. The group of participating programmers consisted of 4 computer science students, 5 PhD students, and 2 software developers. Most of them has been working as software developer for years: they have experience in Java and program analysis. Hereafter, we will call them programmers.

Before the first stage, we defined some hypothetic change scenarios. We gave 7 methods from the sample project to the programmers so that they examine them and determine their impact sets. The methods are the following:

- *writeFolderState* (in FolderState class),
- *internalMoveFile* (in SyncTrashbox class),
- *loadConfig* (in OwnSyncConfiguration class),
- *DeleteFolderAction* (in DeleteFolderAction class),
- *forceDelete* (in FileUtils class),
- *getSyncFolder* (in FolderConfiguration class),
- *isAnyActionFailed* (in OwnSyncResult class).

This selection was purely based on investigating the method kinds and their call information. Among these methods there are a constructor, a recursive, a getter method; simple or complex ones, it is called several times or calls several methods. The list of the 7 methods from our sample project was given to each programmer. Then, the task of the programmers was to use JRipples to discover all of the methods impacted by any possible change made in these methods.

We logged the programmers' actions to retrieve the dependencies which were determined by the programmers using JRipples.

5.2 Stage 1: Discovering dependencies step by step with JRipples

As the starting point of the change (*concept location*) is found by the programmers in JRipples, the remaining methods of the impact set are discovered in a step-by-step look at the neighbours in the dependency graph.

JRipples marks every method with a *Next* label that might be affected by the change of the located method. It is then up to the programmer to check all the possibilities offered by JRipples and decide whether there is a real dependency between those methods. After examining a method labelled with *Next*, the programmer

needs to make a decision and apply one of the labels *Visited*, *Continue*, or *Impacted* to this method.

The label *Visited* means that the programmer did not find a real dependency between the starting and the visited method. Applying this label does not bring new possible dependencies into the impact set.

With the label *Continue*, the programmers can mark that they think there is no real dependency between the starting and the visited method, but transitive dependencies could not be ruled out. Therefore, the methods marked with *Continue* will not be a part of the final impact set but JRipples marks them possibly impacted neighbours for further checking. This means that applying the label *Continue* may bring in some new possible dependencies (marking the impacted neighbours with the label *Next*).

The label *Impacted* means that the programmer believes there is a real dependency between this method and the starting method. Every method labelled as *Impacted* will be a part of the final impact set. Furthermore, setting the label to *Impacted* may bring in new possible dependencies since JRipples marks all unmarked, possibly impacted neighbouring methods with the label *Next*. The first stage ends when none of the methods remains in the impact set labelled as *Next*.

5.3 Preparation for the second stage

After everyone finished their first stage task, the logs were collected. Despite the fact that the programmers searched for dependencies on the level of methods, we lifted the results up to class level. If at least one method in the class had the label *Impacted*, the class got the label *Impacted* as well.

The algorithms were implemented within JRipples and before the second stage we collected all the class level impact sets for the same 7 criteria produced by the different algorithms. We got the union of all dependencies in all kinds of impact sets in case of all 7 methods.

A general purpose benchmark tool called **BEFRIEND** (BEnchmark For Reverse engInEering tools workiNg on source coDe) [15] was used to help getting through the second stage efficiently. With BEFRIEND, the results of reverse engineering tools from different domains that recognize the arbitrary characteristics of the source code can be subjectively evaluated and compared with each other through a web-based application. The benchmark was successfully applied for evaluating and comparing design pattern miner tools [14], clone detector tools [15], rule violation checkers, and now impact analysis algorithms. Despite the fact that BEFRIEND is designed to be very general some major improvements were required in order to make it capable of evaluating and comparing impact analysis results. The most important developments of the system are the follows:

- A new file format for describing impact analysis results.
- A BEFRIEND plug-in for uploading results from this new format.
- New sibling connection strategy (for connecting dependencies).
- A new criterion for the evaluation of dependencies resulted by impact analysis.
- Remarkable improvements in the statistics module (allowing sophisticated statistic queries, e.g.: distribution of the tool results, various statistics about the evaluators, enhanced precision and recall calculation, etc.).

In BEFRIEND, we have to create our own evaluation criteria which are basically questions about the results with two or more possible answers. Then the evaluation process is choosing an answer to every question for every single uploaded result which describes it the best. We defined only one evaluation criterion, since we are interested in the correctness of the results. So, the programmers have to answer the following question: 'Do you think there is a real dependency between these classes?' for every uploaded dependency. There are 4 possible answers to be given to this question:

- Yes, I am sure that there is a dependency.(100%)
- I think there is a dependency.(66%)
- I think there is NO dependency.(33%)
- No, I am sure that there is no dependency.(0%)

We give the possibility to the evaluator not only to choose yes/no answers but to describe his/her level of confidence. Each answer is associated with a percentage value forming a numerical scale from the firm negative answer through the solid negative and solid positive answer to the firm positive answer. According to the Likert scale a typical questionnaire should also contain the possibility for a neutral answer (neither agree that there is a dependency nor disagree). This neutral answer is included in BEFRIEND implicitly by allowing users to skip the evaluation of some instances (where they are unsure about their answers). The neutral answers are excluded from the calculation of statistic information making the results more relevant (there are also examples in our experiment for users choosing the possibility of neutral answers).

Using BEFRIEND terminology, there are *instances* in the system which can be evaluated against the evaluation criteria. We chose the obvious approach and defined the instances as a dependency between two classes (where one of the classes contains the method from which the impact analysis is started). As a final step of the preparation, the union of the impact sets computed by algorithms or programmers was uploaded to the benchmark grouped by the scenarios.

5.4 Stage 2: Discovering dependencies all at once with BEFRIEND

In the second stage, the union of the results, either found by a programmer or an algorithm, are provided together for the programmers, who were asked to evaluate the dependencies individually. The programmers evaluated the dependencies grouped by the scenarios but without knowing which tool or programmer found them.

The real work of the contributing programmers in stage two was after the preparation steps are done. They were asked to proceed as follows: vote for every dependency uploaded to the benchmark via the online interface. The voting process was done by opening all the sibling groups (impact sets of a method) and picking an answer of the evaluation criteria for every dependency in this group. The second stage ended when all the contributing programmers evaluated every single dependency.

The BEFRIEND database used in our experiment is publicly available online [1].

6. Results and discussion

In this section, we answer the research questions in two approaches. We calculate statistics from 2 kinds of dependency sets. First, we get the dependencies whose average votes are above 0% threshold value (at least one programmer evaluated the dependency with a 33% value), This means that a dependency is filtered out, if all programmers agreed that it is not a true dependency (all programmers' votes are 0%). So we throw away conservatively the dependencies, which are determined by any algorithms but not by any programmers. Secondly, we considered the dependencies, where the average vote is above 50% threshold value. Hereafter, we will call these dependencies *positive votes*. This means that a dependency is filtered out, if the programmers mostly agreed that it is not a true dependency (the average vote is less than or equal to 50%).

We chose the threshold value 50% to be between 33% and 66%, which means filtering out irrelevant dependencies in BEFRIEND. It could be reasonable to give the threshold value 66%(mainly solid and firm positive answers), but – as you can see later – most of the dependencies have a standard deviation between 30% and 50% in the votes, such a high threshold value can cut off too much dependencies which can be important to us.

Q1: Do the different algorithms and the programmers identify the same dependencies?

In order to answer Q1 question, we calculate M1 and M2 metrics:

- **M1:** the number of algorithms which detected dependencies whose average votes are above 0%,
- **M2:** the number of algorithms which detected dependencies whose average votes are above 50% (positive votes).

When a programmer uses the own analysis of JRipples step by step, the methods/classes are labelled considering undirected dependencies (interactions). Our preexamination showed that for each scenario the dependency set contains all of the classes if the programmers use JRipples user interface. That is why we are sure that all of the possible dependencies can be determined by JRipples, so the programmers have only the role to filter out the less possible dependencies.

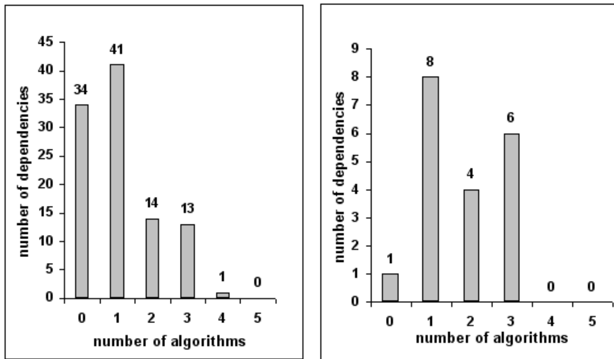

Figure 3. Number of algorithms which detected dependencies whose average votes are above 0% (left side) and 50% (right side)

The left hand side of Figure 3 shows how many algorithms detected how many dependencies. The algorithms and the programmers recognized 118 dependencies altogether. In the first case, we filtered out 15 dependencies with the 0% threshold value. Most of the dependencies are found by only one algorithm and there are 34 dependencies which are determined only by the programmers, and not by any of the algorithms. This diagram shows that the different algorithms find different kinds of dependencies which are rarely the same. While less and less dependencies are found by 2 or 3 algorithms, there is one dependency which is determined by 4 algorithms. The only algorithm which did not find it was the co-changing algorithm with a 1.0 correlation value. This dependency is between the *loadConfig* method and the *FolderConfiguration* class. The *loadConfig* method collects and stores properties about folders, states of the folders and regular expression patterns. The *FolderConfiguration* class gets these properties.

In the right part of the Figure 3, only positive votes were considered. 99 dependencies are ruled out by this filtering, and only 19 real dependencies remain according to the programmers' evaluation. There is only one dependency which was determined only by the programmers. This dependency is between the *loadConfig* method and the *OwnSyncTestCase* class. The class has a *createOwnSyncConfiguration* method whose body is similar to the *loadConfig*. So the programmers thought if the *loadConfig* method would change, they had to change the *createOwnSyncConfiguration* method as well. The static analyzers did not determine this dependency because there is no data or control dependency between the method and the class. The co-change algorithm did not determine it because the *OwnSyncTestCase* class was added to history at one of the last commits, and we were interested only in modified classes and not added classes. The only dependency which were determined by 4 algorithms disappeared. The programmers may have thrown away this dependency because the *loadConfig* method sets only attributes in the Property object which determine on which directory the file synchronization works and what kind of patterns it uses. Although the *FolderConfiguration* class uses this Property object, its retrieved value is only delegated to other classes, so the potential change of the *loadConfig* does not affect the functionality of the *FolderConfiguration* class.

We can conclude from these metrics that the different algorithms and the programmers determine almost different dependency sets, only few dependencies are determined by more than one algorithms.

Q2: What is the relationship between the different kinds of dependency sets?

Six metrics are associated with Q2 question in order to answer it in a quantitative way:

- **M3:** the normalized size of the intersection of two dependency sets ($A \cap B$), where the average votes of the dependencies are above 0%,
- **M4:** the normalized size of the difference of two dependency sets ($A \setminus B$), where the average votes of the dependencies are above 0%,
- **M5:** the normalized size of the difference of two dependency sets in the opposite direction ($B \setminus A$), where the average votes of the dependencies are above 0%,
- **M6:** the normalized size of the intersection of two dependency sets ($A \cap B$), where the average votes of the dependencies are above 50%,
- **M7:** the normalized size of the difference of two dependency sets ($A \setminus B$), where the average votes of the dependencies are above 50%,
- **M8:** the normalized size of the difference of two dependency sets in the opposite direction ($B \setminus A$), where the average votes of the dependencies are above 50%.

Table 2 presents **M3, M4, M5** metrics, the relationships between each dependency set pair. We determined the intersection and the differences of the dependency sets for every algorithm pair. The values in the table are normalized by the size of the union of the given sets. In most cases there is no complete containment and almost every set has intersection with the others. There are only 2 cases where one of the sets is a subset of the other. These were expected due to the definition of the algorithms: the co-changing classes with a 1.0 correlation value are a subset of the co-changing classes with a 0.4 correlation value; and calling dependency set is a subset of the SEA relation set. This shows that no matter how different these algorithms are, some dependencies are mutually found by more than one algorithm.

In Table 2, we considered the dependencies with 0% threshold value. This results in that the union of these dependency sets can contain the dependencies that are not relevant, due to the imprecision of the algorithms.

Algorithms(A,B)	**M3** (A∩B)	**M4** (A\B)	**M5** (B\A)
programmers, call	16.28%	82.56%	1.16%
programmers, slice	11.36%	85.23%	3.41%
programmers, SEA	37.38%	42.06%	20.56%
programmers, co-change (1)	2.3%	95.4%	2.3%
programmers, co-change (0.4)	24.49%	62.24%	13.27%
call, SEA	24.19%	**0%**	75.8%
call, co-change (1)	**0%**	78.95%	21.05%
call, co-change (0.4)	8.33%	22.92%	68.75%
call, slice	47.37%	31.58%	21.05%
SEA, co-change (1)	4.76%	93.65%	1.59%
SEA, co-change (0.4)	23.75%	53.75%	22.5%
SEA, slice	13.64%	80.30%	6.06%
slice, co-change (1)	**0%**	76.47%	23.53%
slice, co-change (0.4)	2.04%	24.49%	73.47%
co-change (1), co-change (0.4)	10.81%	**0%**	89.19%

Table 2. Comparison of dependencies sets detected by different algorithms

According to the **M6, M7, M8** metrics, Figure 4 shows the relationships among the dependencies with 50% threshold value. By filtering dependencies, the containing relations became much simpler.

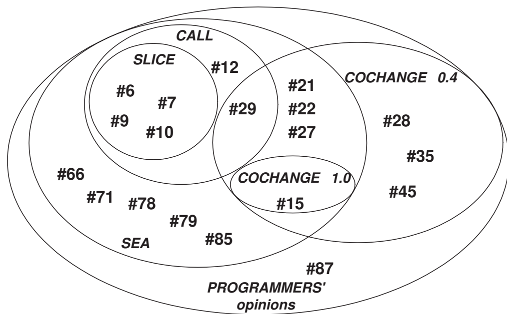

Figure 4. Comparison of dependencies sets detected by different algorithms filtered out by the programmers' evaluation (only positive votes). We represent the id-s of the dependencies which are belong to the certain set.

It was a surprising observation that the slice dependency set is a subset of the dependency set of the callgraph. The dependency set retrieved from the callgraph shows relationships where in the given method declaration one of the methods of the class is invoked. The slice dependency set does not contain #12 and #29 dependencies which are *super* method calls, they call the constructor of the ancestor class. We use the Indus static slicer API, which is the only available static slicer for Java programs. These two dependencies show that the slicer is not complete. For example, in the case of #22 (dependency between the *getSyncFolder* method and the *FolderSynchronizer* class), the *getSyncFolder* method is invoked from the *getFolderLocation* method of the *FolderState* class (#27), which is invoked from the *evaluateActions* method of the *FolderSynchronizer* class. We can mention here #15, #78, #79, and #85 as well for the same reason. We considered only the forward slice, so the dependencies #21, #27, #66 and #71 are contained only in dependency set computed by SEA. #15 is a special dependency since every time the *FolderState* class changed, the *OwnsyncStarter* class has changed as well according to the history retrieved from SVN repository. The dependency #87 is between the *loadConfig* method and the *OwnSyncTestCase* class. The class has a *createOwnSyncConfiguration* method whose body is similar to the body of the *loadConfig*. So that the programmers thought if *loadConfig* method would change, they had to change the *createOwnSyncConfiguration* method as well, but none of the static analyzers or examining co-changing detect it. The dependencies retrieved from co-changing classes were examined and we noticed that they were changed together but not for the same reason according to the change log. The users erroneously determined them as dependencies. For example, the dependency #28 (between *DeleteFolderAction* constructor and *FolderState* class) might be evaluated as a dependency because the constructor has two parameters and the type of one of them is FolderState.

The Venn diagram on Figure 4 depicts that there are a lot of complete containments among the dependency sets. By raising the threshold to 50%, the less relevant dependencies – according to the programmers' evaluation – have disappeared. The probable explanation to this is that the programmers may have found some errors in the results of the algorithms due to their imprecisions.

Q3: According to the precision and recall values of the algorithms, what is the rank of the algorithms?

In our next experiment we verified how close were the algorithms' dependency sets to the the programmers' opinion and each others. For this, we used precision and recall values, taking the collective programmers' opinion as the correct results.

From the precision and recall values, we determine 2 metrics for each algorithm:

- **M9:** the harmonic mean of the precision and recall value of the given algorithm, where the average votes of the dependencies are above 0%,
- **M10:** the harmonic mean of the precision and recall value of the given algorithm, where the average votes of the dependencies are above 50%.

Algorithm	Precision	Recall	**M9** (Harmonic mean)
SEA	80.65%	48.54%	60.6%
cochange(0.4)	89.19%	32.04%	47.14%
call	86.67%	12.62%	22.03%
slice	92.31%	11.65%	20.69%
cochange(1.0)	100%	3.88%	7.47%

Table 3. Precision and recall value of different algorithms

Table 3 shows the precision and recall values of the algorithms. We calculated precision as the rate of the number of dependencies computed by the given algorithm with at least one positive vote (at least one programmer evaluated it with a 33% value) and the number of all detected dependencies by the given algorithm. We calculated recall as the rate of the number of dependencies computed by the given algorithm with at least one positive vote (at least one programmer evaluated it with a 33% value) and the union of the dependencies computed by the algorithms which are evaluated at least once with at least with a 33% value.

The dependency sets computed by SEA have a relatively high precision and recall values, and hence it produces the best result according to the harmonic mean value. Although the precision of a slice is very high, it has a very weak recall value. It is the same for all the other algorithms as well. They found a small number 'real' dependencies but the ones they found are mostly right. The set of the 100% co-changed classes contains 4 dependencies and all the four dependencies proved to be real dependencies by the programmers, however, these 4 dependencies are just a small part of the union of the dependency sets.

Algorithm	Precision	Recall	**M10** (Harmonic mean)
SEA	24.19%	78.95%	37.03%
call	40%	31.58%	35.29%
cochange(0.4)	21.62%	42.11%	28.57%
slice	30.77%	21.05%	25%
cochange(1.0)	25%	5.26%	8.69%

Table 4. Precision and recall value of different algorithms filtered out according to the programmers' evaluation

Table 4 presents also the precision and the recall values of the algorithms with the difference that we considered only dependencies with positive votes. By filtering out the irrelevant dependencies, the precision decreased and the recall increased on average. These change directions are based on the increased threshold value: the number of dependencies evaluated as real dependencies by the programmers radically decreased. The number of true positive and false negative dependencies decreased and the false positive ones increased. The SEA dependency set has the highest overall ranking, and it has a specifically high recall value compared to the others in this case. This value is not 100% because the measurement is based on programmers' evaluation, and they confirmed several false dependencies.

Q4: Do the programmers identify the dependencies with the same confidence levels?

We compared the programmers' evaluation for each dependency to examine how different they evaluated the dependencies. We can measure it with the **M11** metric, the deviation of the programmers' votes.

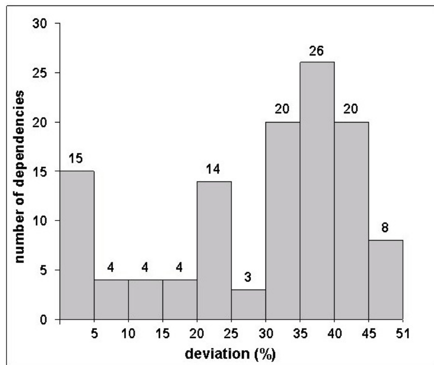

Figure 5. The deviation of the programmers' votes.

Figure 5 depicts the deviation of the programmers' votes. When the programmers evaluated the relevance of the dependencies, they could choose from four levels (0%, 33%, 66%, and 100%). This figure shows how close the answers are. Most of the dependencies have a standard deviation between 30% and 50% in the votes, which means that there are not so many dependencies which are evaluated similarly by the different programmers. In the case of 15 dependencies, the deviation was zero. The programmers were sure that there was no dependency in these cases (they all voted with 0% value). The dependency whose evaluation was the most diverse is #5 with a 50.45% deviation value. This dependency was determined by 3 algorithms and on the average it received a positive vote from the programmers, only the co-changing algorithms with both correlation values missed it. This dependency is between the *writeFolderState* method and the *OwnSyncStatus* class. The method calls the *setMessage* and the *appendToMessage* methods from the *OwnSyncStatus* class to prepare a message about how long it takes to write a folder state into a file. 7 programmers were sure that it was not a real dependency (they gave a 0% vote) but the remaining 4 programmers were sure that it was a dependency (a 100% vote).

Q5: How much the programmers' opinions change after seeing the results of different impact analysis algorithms over their initial opinions?

We can compare the programmers' evaluation not only for each dependency, but for each programmer as well. First, we considered the programmers' precision and recall values in the two stages. The metrics are the following:

- **M12:** the precision value of the given programmer in the first stage,
- **M13:** the recall value of the given programmer in the first stage,
- **M14:** the precision value of the given programmer in the second stage,
- **M15:** the recall value of the given programmer in the second stage.

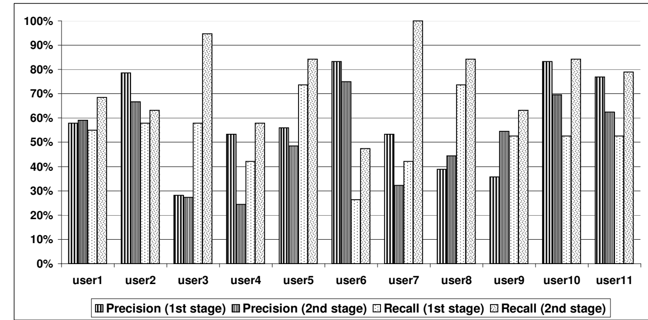

Figure 6. Comparison of programmers' precision and recall in the first stage and the second stage

Figure 6 presents how much the programmers' evaluation changed in the second stage compared to the first stage. The bar-chart shows the programmers' individual precision and recall values in each stage with respect to the collective opinion. The calculation of the precision and recall values for the first stage is based on the results produced by the programmers using JRipples. On the other hand, in the second stage we have considered those dependencies as positive instances that the programmer evaluated by a 66% or 100% value. In both cases the dependencies with positive votes were used as the etalon. Generally we can say that the programmers' precision value decreased, while the recall values increased. This is particularly true when a programmer introduced many new dependencies in the second stage compared to his/her first stage results. In view of dependencies determined by algorithms, they found more 'relevant' dependencies and throw away less 'relevant' dependencies.

A more detailed evaluation of the programmers' opinion change is presented in Figure 7[3]. For each programmer four kinds of rate were calculated:

[3] While in the first stage the programmers can mark a given method with *Visited* or *Impacted* label, in the second stage, BEFRIEND gives 4 possible values (0%, 33%, 66%, and 100%). In order to compare these two different evaluation values, we mapped *Visited* label to 0% and *Impacted* label to a 100% value.

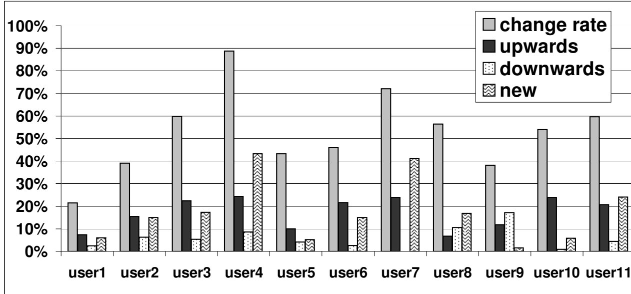

Figure 7. Comparison of programmers' evaluation in the first stage and the second stage

- **M16:** changing rate,
- **M17:** upwards changing rate,
- **M18:** downwards changing rate,
- **M19:** new dependencies rate.

We calculate a rate for each scenario and for each programmer: the number of the dependencies where the evaluation value changed (the programmer changed his opinion) divided by the total number of the dependencies identified by the given programmer for the given scenario. The first bars show these rates averaging for scenarios. These values are very different: from 20% to 90%. The values show the number of changes upwards and downwards, and the number of the dependencies which were not evaluated in the first stage but were evaluated in the second stage with at least a 33% value. The second bars show only the rate of upward changes (the evaluation of a certain dependency was changed at least 2 levels upwards), the third ones show the rate of downward changes (the evaluation of a certain dependency was changed at least 2 levels downwards), and the fourth bars show the rate of dependencies which are not evaluated in the first stage but were evaluated with at least 33% value in the second stage. Generally, when the change rate is high, the last bars are high as well. In most cases, the number of downward changes are negligible, while the upward changes are more significant. As a general rule, we can establish that there are several dependencies appearing only in the second stage and the programmers typically changed their mind upwards and not downwards. They insisted on their previously observed dependencies but they were open to new dependencies, too.

7. Threats to Validity

This paper is an empirical study, and it has limitations that must be considered during evaluating the result and generalizing in other contexts.

First, our hypothetical change requests may not equally represent real maintenance situations. If a programmer has a maintenance task, there is a certain program point in a method which needs modification. Contrarily, in our experiment the programmer had to find all the methods/classes impacted by *any* changes of the forward defined methods. It is necessary because the algorithms have method or class granularity, not a statement one. However, for testers this can represent a real maintenance situation: if the developers give only the names of the changed methods to the testers, they must proceed from these methods to determine their impact sets and the necessary test cases.

There is another factor which is not common in software maintenance: the programmers were not familiar with the sample project. No project exists that the participants equally know, so that is why we chose a project which is not known for each of them. And generally, the participants were all computer science students, PhD students, or software engineers: most of them are not familiar with impact analysis, they are common programmers who determine dependencies according to their best knowledge.

We have considered algorithms which determine impact sets to only one program. The constraints has been mentioned in Section 5 (only Java code, memory consumption limitation, extended SVN history, etc.). In this case, the programmer can understand the project much better despite understanding several projects a bit. However, this project is an actual, non-trivial software system with real SVN history. While only one subject system was examined, the empirical study has low statistic power. We cannot claim that the results are generalizable.

Next, comparing the programmers' evaluation brought in some difficulties. The two stages are very different: tools with different user interfaces and different confidence levels. In the first stage, the programmers determined the dependencies step by step where only one piece of source code at the same time could be seen and only 'dependency' or 'not dependency' could be stated. Contrary to JRipples, at the second stage, BEFRIEND shows not only all the dependency candidates but both of the sources of classes of a certain dependency could be seen simultaneously. BEFRIEND gives 4 possible values (0%, 33%, 66%, and 100%) in order to represent the programmers' level of confidence. When we compared the programmers' evaluation, we noticed that they insisted on their previously observed dependencies, so it is possible that they remembered their opinions from the first stage. The solution can be to split the programmers into two groups and one of the groups determine dependencies only in the first stage, while the others do it only in the second stage.

8. Conclusion

In this paper, we presented an empirical comparison of four static impact analysis techniques (call information, static slice, SEA relation and co-changing files retrieved from SVN repositories) and dependencies which are determined by programmers. We designed and performed an empirical study to find out how much the different kinds of dependency sets supported program comprehension, and made finding the impact sets of several hypothetical modifications more successful.

Our first finding was that the algorithms produce quite different results. There are a lot of dependencies that are found only by one of the algorithms, and only one was found by almost all of them. This suggests that the algorithms should be treated as complementing each other.

The overall opinion of the programmers showed quite large deviation, which also shows that no single algorithm can be relied upon, and that the human opinion may depend on many different things. We compared the precision and recall rates of algorithms to the collective programmer opinion, and found that the results produced by Static Execute After algorithm were the closest.

Finally, we checked how much the programmers changed their initial opinion based on JRipples only, after seeing all the possible dependencies computed by the different algorithms. Although the rate of changes and consequently the precision and recall values of the programmers' decisions did not change so much, we found that if there was a change it was mainly in an upward direction, meaning that in principle the programmers observed the dependencies which they determined at the first stage, but they were open to accept new dependencies, too.

The results are promising and raising further interesting questions. Of course, a bigger case study (with more and bigger programs and more participants) could be useful to verify the findings. An important enhancement to the experiment design would be to start from concrete change requests and not only pointing to the

starting concept methods. In that case the oracle can be a manually defined impact set, not the programmers opinions.

As the sideeffect of this empirical study, we have information about the programmers' decisions. Here, we did not process them individually, but it would be interesting to compare the programmers to each other according to their answers and qualification.

Another topic for the future work is to compare static and dynamic impact sets. In this article we used only static analysis to determine the dependency sets, however, we can determine dynamic impact sets as well (after we implement several dynamic algorithms) or combine static and dynamic impact sets to retrieve 'hybrid' impact sets.

Finally, it would be interesting to perform a similar experiment based on actual programmer activities within a tool like JRipples using different algorithms, instead of investigating the complete impact sets. This would make the comparison of the appropriateness of the different algorithms in this environment more complete.

Acknowledgments

This research was supported in part by the grants OTKA K-73688 and GOP -1.1.2/1-2008-0007.

Thanks to the 11 participants without whom the experiment could not have been conducted.

References

[1] The BEFRIEND homepage. `http://www.inf.u-szeged.hu/befriend/`.

[2] JRipples tool for Incremental Change. `http://jripples.sourceforge.net/`.

[3] IEEE std 610.12-1990: IEEE standard glossary of software engineering terminology, 1990.

[4] G. Antoniol, V. F. Rollo, and G. Venturi. Detecting groups of co-changing files in cvs repositories. In *IWPSE '05: Proceedings of the Eighth International Workshop on Principles of Software Evolution*, pages 23–32, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2349-8. doi: http://dx.doi.org/10.1109/IWPSE.2005.11.

[5] R. S. Arnold and S. A. Bohner. Impact analysis - towards a framework for comparison. In *ICSM '93: Proceedings of the Conference on Software Maintenance*, pages 292–301, Washington, DC, USA, 1993. IEEE Computer Society. ISBN 0-8186-4600-4.

[6] R. S. Arnold and S. A. Bohner. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996. ISBN 0818673842.

[7] V. R. Basili, G. Caldiera, and H. D. Rombach. The goal question metric approach. In *Encyclopedia of Software Engineering*. Wiley, 1994.

[8] Á. Beszédes, T. Gergely, J. Jász, G. Tóth, T. Gyimóthy, and V. Rajlich. Computation of static execute after relation with applications to software maintenance. In *Proceedings of the 2007 IEEE International Conference on Software Maintenance (ICSM'07)*, pages 295–304. IEEE Computer Society, Oct. 2007.

[9] J. Bible, G. Rothermel, and D. S. Rosenblum. A comparative study of coarse- and fine-grained safe regression test-selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):149–183, 2001. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/367008.367015.

[10] S. A. Bohner. Impact analysis in the software change process: a year 2000 perspective. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, pages 42–51, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7677-9.

[11] B. Breech, M. Tegtmeyer, and L. Pollock. A comparison of online and dynamic impact analysis algorithms. In *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, pages 143–152, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2304-8. doi: http://dx.doi.org/10.1109/CSMR.2005.1.

[12] J. Buckner, J. Buchta, M. Petrenko, and V. Rajlich. JRipples: A tool for program comprehension during incremental change. In *IWPC*, pages 149–152, 2005.

[13] J. Cohen. A coefficient of agreement for nominal scales. *Psychological Bulletin*, 20:37–46, 1960.

[14] L. J. Fülöp, R. Ferenc, and T. Gyimóthy. Towards a Benchmark for Evaluating Design Pattern Miner Tools. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR 2008)*. IEEE Computer Society, Apr. 2008. doi: http://dx.doi.org/10.1109/CSMR.2008.4493309.

[15] L. J. Fülöp, P. Hegedűs, R. Ferenc, and T. Gyimóthy. Towards a Benchmark for Evaluating Reverse Engineering Tools. In *Tool Demonstrations of the 15th Working Conference on Reverse Engineering (WCRE 2008)*, Oct. 2008. doi: http://dx.doi.org/10.1109/WCRE.2008.18.

[16] L. J. Fülöp, P. Hegedűs, and R. Ferenc. BEFRIEND - a Benchmark for Evaluating Reverse Engineering Tools, accepted, to appear. *Periodica Polytechnica*, 2009.

[17] D. M. German, A. E. Hassan, and G. Robles. Change impact graphs: Determining the impact of prior codechanges. *Information and Software Technology*, 51(10):1394 – 1408, 2009. ISSN 0950-5849. Source Code Analysis and Manipulation, SCAM 2008.

[18] A. E. Hassan and R. C. Holt. Predicting change propagation in software systems. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 284–293, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2213-0.

[19] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[20] INDUS Homepage. Indus project: Java program slicer and static analyses tools. `http://indus.projects.cis.ksu.edu/`. URL `http://indus.projects.cis.ksu.edu/`.

[21] J. Jász, Á. Beszédes, T. Gyimóthy, and V. Rajlich. Static execute after/before as a replacement of traditional software dependencies. In *Proceedings of the 2008 IEEE International Conference on Software Maintenance (ICSM'08)*, pages 137–146. IEEE Computer Society, Oct. 2008.

[22] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Softw. Eng.*, 28(8):721–734, 2002. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.2002.1027796.

[23] M. Lindvall and K. Sandahl. How well do experienced software developers predict software change? *J. Syst. Softw.*, 43(1):19–27, 1998. ISSN 0164-1212. doi: http://dx.doi.org/10.1016/S0164-1212(98)10019-5.

[24] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. J. Harrold. An empirical comparison of dynamic impact analysis algorithms. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 491–500, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2163-0.

[25] V. Rajlich. Intensions are a key to program comprehension. In *ICPC*, pages 1–9, 2009.

[26] V. Rajlich and P. Gosavi. Incremental change in object-oriented programming. *IEEE Software*, 21:62–69, 2004. ISSN 0740-7459. doi: http://doi.ieeecomputersociety.org/10.1109/MS.2004.17.

[27] B. G. Ryder. Constructing the call graph of a program. *IEEE Trans. Softw. Eng.*, 5(3):216–226, 1979. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.1979.234183.

[28] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Trans. Software Eng.*, 31(6):429–445, 2005.