Generalizing the Analysis of Evolutionary Coupling for Software Change Impact Analysis

Thomas Rolfsnes*, Stefano Di Alesio*, Razieh Behjati*, Leon Moonen* and Dave W. Binkley§

*Simula Research Laboratory, Oslo, Norway §Loyola University Maryland, Baltimore, Maryland, USA

{thomgrol, stefano, raziehb}@simula.no, leon.moonen@computer.org, binkley@cs.loyola.edu

***Abstract*—Software change impact analysis aims to find artifacts potentially affected by a change. Typical approaches apply language-specific static or dynamic dependence analysis, and are thus restricted to homogeneous systems. This restriction is a major drawback given today's increasingly heterogeneous software. Evolutionary coupling has been proposed as a language-agnostic alternative that mines relations between source-code entities from the system's change history. Unfortunately, existing evolutionary coupling based techniques fall short. For example, using Singular Value Decomposition (SVD) quickly becomes computationally expensive. An efficient alternative applies targeted association rule mining, but the most widely known approach (ROSE) has restricted applicability: experiments on two large industrial systems, and four large open source systems, show that ROSE can only identify dependencies about 25% of the time.**

To overcome this limitation, we introduce TARMAQ, a new algorithm for mining evolutionary coupling. Empirically evaluated on the same six systems, TARMAQ performs consistently better than ROSE and SVD, is applicable 100% of the time, and runs orders of magnitude faster than SVD. We conclude that the proposed algorithm is a significant step forward towards achieving robust change impact analysis for heterogeneous systems.

I. Introduction

As a software system evolves, the amount and complexity of interactions in the code grows. For a developer, it therefore becomes increasingly challenging to be in control of the impact of a change made to the system. One potential solution to this problem, change impact analysis [5, 14, 18, 28], aims to find artifacts (e.g., files, methods, classes etc.) affected by a given change. This knowledge can then be used either as direct feedback to the developer, or as the basis for another down-stream task such as test-case selection and prioritization.

Traditionally, change impact analysis has been performed using static or dynamic dependence analysis (e.g., by identifying the methods that call a changed method). One advantage of such approaches is that they are considered *safe*, as all potentially affected artifacts are found [4]. However, in recent years there has been an investigation of alternative approaches. This search is motivated, in part, by limitations in existing techniques. For example, static and dynamic dependence analysis are generally language-specific, making them unsuitable for the analysis of heterogeneous software systems [25]. In addition, they can involve considerable overhead (e.g., dynamic analysis' need for code-instrumentation), and tend to over-approximate the impact of a change.

One alternative is to identify dependencies through *evolutionary coupling*. Such couplings differ from the ones found through static and dynamic dependence analysis, in that they are based on *how* the software system has evolved over time. Using this information in essence attempts to tap into the developer's inherent knowledge of the inner workings of the system. This knowledge can manifest itself in several ways, for example through commit-comments, bug-reports, context-switches in an IDE etc. In this paper we consider *co-change* as a basis for establishing evolutionary couplings. Co-change information can be extracted from a project's version control system, its issue tracking system, or both, depending on how the project maintains its software revision history.

The goal of evolutionary coupling is to mine connections between entities in the software from the co-change data. While several levels of granularity are possible, in this paper we focus on connections at the file level. Observe that this is without loss of generality, as the mining algorithms are agnostic to the choice of granularity. Provided that suitably fine-grained co-change data is obtained, the algorithms will just as well relate methods or variables as files in a system.

There are several options to mine evolutionary coupling from co-change data. For example, although it is computationally expensive, Singular Value Decomposition (SVD) can be used to form clusters of files. These clusters can then be used to predict files likely to change together [20]. A considerably faster approach is based on Srikant et al.'s *targeted association rule mining* algorithm [22].

However, the off-the-self use of this algorithm has a significant limitation causing a lack of *applicability* (a notion formalized in Section IV). Consider ROSE, which uses the algorithm for software change recommendations [30]. Experiments with ROSE on six large systems show that it can only find evolutionary couplings about 25% of the time.

This paper presents TARMAQ, a new generalized algorithm for targeted association rule mining that overcomes this limitation. Moreover, we make the following additional contributions: we classify the limitations of existing algorithms, and we empirically evaluate the performance of TARMAQ on two industrial software systems and four open source software systems. We find that TARMAQ performs consistently better than two popular alternatives, is applicable 100% of the time, and runs orders of magnitude faster than SVD. We conclude that the new algorithm is a significant step forward towards achieving robust change impact analysis for heterogeneous systems.

978-1-5090-1855-0/16 $31.00 © 2016 IEEE
DOI 10.1109/SANER.2016.101

II. Research Questions

Our research is driven by a desire to conduct impact analysis on large heterogeneous systems via evolutionary coupling. Despite its limited applicability, targeted association rule mining has shown promise in addressing software engineering problems. Combined, these two observations lead us to investigate the following research questions:

RQ1 What are the limitations of existing techniques to analyse evolutionary coupling?

RQ2 Can we devise an alternative technique that does not suffer from these limitations?

RQ3 How well does the best alternative perform in comparison to the state-of-the-art?

The remainder of the paper is organized as follows: Section III defines some terminology and provides background on the use of association rule mining. The limitations of existing algorithms (RQ1) are discussed in Section IV. In Section V we address RQ2 by introducing TARMAQ, a new algorithm for targeted association rule mining algorithm in software engineering context. We address RQ3 by empirically evaluating TARMAQ and state-of-the-art algorithms on a series of large software systems in Sections VI – VIII. We discuss related work in Section IX, and we conclude in Section X.

III. Background

Agrawal et al. introduced the concept of *association rule mining* as the discipline aimed at inferring relations between *entities* of a dataset [1]. The relations, called *association rules*, are identified from a collection of transactions where each transaction is a subset of the entities. For example, consider the classic application of analyzing shopping cart data: if multiple transactions include bread and butter then the corresponding association rule would be *bread* $\rightarrow$ *butter*. This can be read as *"if you buy bread, then you are also likely to buy butter"*.

In the context of evolutionary coupling for software systems, the entities are the files of the system and the collection of transactions is the change history $\mathcal{H}$ of the system. Note that, in general, entities in the system can be considered at other levels of granularity, such as method- or procedure-level. Each transaction $T \in \mathcal{H}$ is a *commit* of changed files, i.e., a transaction includes the set of files that were either changed or added while addressing a given bug or feature addition (creating a *logical dependence* [7]).

Finally, an *association rule* is an implication of the form $A \rightarrow B$, where A and B are disjoint, A is referred to as the *antecedent*, and B is referred to as the *consequent*. In our use, the association rule $A \rightarrow B$ denotes that *"if the files in A change, then the files in B are also expected to change"*.

Several measures are used to reason about the rules. First, the *frequency*, denoted ϕ, of association rule $A \rightarrow B$ is the number of transactions in $\mathcal{H}$ where items in A and B change together:

$$\phi(A \rightarrow B) \overset{\text{def}}{=} |\{T \in \mathcal{H} : A \cup B \subseteq T\}| \tag{1}$$

Usually, the frequency of a rule is normalized by dividing by the number of transactions. For this purpose, the *support*, denoted σ, of a rule $A \rightarrow B$ is defined as the frequency of a rule divided by the number of transactions in the history:

$$\sigma(A \rightarrow B) \overset{\text{def}}{=} \frac{\phi(A \rightarrow B)}{|\mathcal{H}|} \tag{2}$$

Intuitively, higher support for a rule means that it is more likely to hold. Alternatively, rules with low support identify relations that rarely occur. For this reason, a minimum threshold on support is often used to filter out uninteresting rules.

The final measure used, *confidence*, denoted κ, measures the reliability of the inference made by a rule. The confidence of a rule is defined as its frequency divided by the number of transactions that contain its antecedent.

$$\kappa(A \rightarrow B) \overset{\text{def}}{=} \frac{\phi(A \rightarrow B)}{|\{T \in \mathcal{H} : A \subseteq T\}|} \tag{3}$$

Confidence measures the ratio of the transactions in which the files in A *and* B are present, to the transactions where files in A are present. Consequently, the higher the confidence of a rule $A \rightarrow B$, the higher the chance that when the files in A are modified, then the files of B are also modified.

As originally defined [1], association rule mining generates rules that express patterns in a complete data set. However, some applications can exploit a more focused set of rules. *Targeted association rule mining* [23] is a refinement that focuses the generation of rules on a particular *query* supplied by the user. It does so by removing all transactions that are not related to the query from the database of transactions (the change history $\mathcal{H}$ in our context), which results in a dramatic reduction of execution time [23].

IV. Problem Description

This section characterizes a key limitation of applying off-the-shelf targeted association rule mining to the problem of providing developer change recommendations. As described earlier, the technique removes all transactions that are not related to the query from the database of transactions used for rule generation. However, this strategy can often filter away all the transactions, leaving an empty database, and rendering the technique incapable of producing a recommendation.

Before considering the causes of this limitation, we formalize the notion of *applicability*:

Definition: Given a history $\mathcal{H}$ and a query Q, a targeted association rule mining technique $\mathcal{T}$ is said to be *applicable* to query Q if the filtered history after removing all transactions not related to Q is non-empty. Conversely, we say that $\mathcal{T}$ *is not applicable* when the filtered history for Q is empty.

It turns out that there are two patterns that lead to a lack of applicability of the existing techniques. The first pattern is when a file in the query has never occurred before in any transaction of the history. The second pattern is a query of previously seen files that nonetheless have not been seen together as a proper subset of a single transaction. The second pattern requires a proper subset because a transaction that exactly matches the query involves no other files, and thus there are no other files to recommend. Therefore, nothing can be learned from such a transaction.

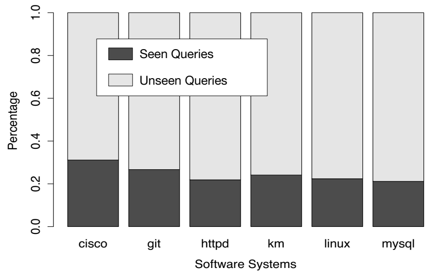

Fig. 1. The percentage of queries that were found to be *seen* or *unseen* in the change histories of each of the software systems.

To illustrate the two patterns, let *a, b, c, d* and *e* represent five source files belonging to an evolving software system. Suppose that the version history of the system, $\mathcal{H}$, contains three transactions:

$$\mathcal{H} = [\{a, b, c\}, \{a, d\}, \{c, d\}]$$

Then, the queries $q_1 = \{e, d\}$ and $q_2 = \{a, c, d\}$ exemplify the two patterns for which existing targeted association rule mining techniques are not applicable. q_1 includes a new file e which does not occur in any transaction of $\mathcal{H}$. In this case techniques such as ROSE [30] fail to provide any recommendation. In the second example, the files in q_2 have all occurred in the history, but never together. Here again ROSE fails to provide a recommendation. In the following, we refer to these queries as *unseen* queries because they include a pattern that is unseen in the change history.

The opposite of an unseen query is a *seen* query. For such queries ROSE is able to provide a recommendation. Two examples of seen queries are $q_3 = \{a\}$ and $q_4 = \{a, b\}$. Because there is at least one transaction in $\mathcal{H}$ that is a proper superset of these queries, ROSE can produce a result.

To find out to what extent this limitation affects the applicability of ROSE and similar approaches, such as that of Ying et al. [26], we studied the distribution of unseen and seen queries in four open source repositories, namely, Git[1], Apache HTTP Server[2], Linux Kernel[3], MySQL[4] as well as two software repositories from our industry partners, Cisco Norway[5] and Kongsberg Maritime (KM)[6]. The results are reported in Figure 1 [7]. The high percentage of unseen queries in all six cases implies that traditional targeted association rule mining cannot produce results a significant amount of the time.

We can answer our first research question, RQ1, in the affirmative; it is possible to characterize the limitations of existing evolutionary coupling based techniques. Specifically, the two patterns described in this section prevent current techniques from returning a result. This is because these techniques use an over-restrictive history filtering, which only keeps transactions where *all* the files in the query are present. This suggests the need for alternative and more relaxed rule filtering methods.

[1] https://www.openhub.net/p/git

[2] https://www.openhub.net/p/apache

[3] https://www.openhub.net/p/linux

[4] https://www.openhub.net/p/mysql

[5] http://www.cisco.com/web/NO/index.html

[6] http://www.km.kongsberg.com/

[7] The study *re-enacted* the change history,basically using commits as a query over the change history up to the point that the commit was made.

Algorithm 1: TARMAQ

```
**Require:** The history: $\mathcal{H}$, and the query: $Q$
**Ensure:** A ranked list of files: $\mathcal{F}$
1  {Filtering Step}
2  $k \leftarrow 0$
3  $\mathcal{H}_f \leftarrow \emptyset$ {$\mathcal{H}_f$: the filtered history}
4  for all $T \in \mathcal{H}$ do
5      if $|Q \cap T| = k$ then
6          $\mathcal{H}_f \leftarrow \mathcal{H}_f \cup \{T\}$
7      else if $|Q \cap T| > k$ then
8          $k \leftarrow |Q \cap T|$
9          $\mathcal{H}_f \leftarrow \{T\}$
10 {Rule Creation Step}
11 $\mathcal{R} \leftarrow \emptyset$ {$\mathcal{R}$: the set of rules}
12 for all $T \in \mathcal{H}_f$ do
13     $Q' \leftarrow Q \cap T$
14     for all $x \in T \setminus Q'$ do
15         $\mathcal{R} \leftarrow \mathcal{R} \cup \{Q' \rightarrow x\}$
16         update($\sigma(Q' \rightarrow x)$)
17         update($\kappa(Q' \rightarrow x)$)
18 {Ranked List Creation Step}
19 $\mathcal{R}_s \leftarrow$ sort($\mathcal{R}$) {sorts using $\sigma$ and $\kappa$}
20 for all $i \in [1..$ length($\mathcal{R}_s$)$]$ do
21     $\mathcal{F}[i] \leftarrow$ consequent($\mathcal{R}_s[i]$)
22 return $\mathcal{F}$
```

V. Proposed Solution: TARMAQ

This section presents TARMAQ, a novel algorithm implementing *Targeted Association Rule Mining for All Queries*. TARMAQ takes as input a transaction history $\mathcal{H}$ and a query Q, and generates a ranked list $\mathcal{F}$ of files where higher ranked files are more related to Q. As shown in Algorithm 1, TARMAQ consists of three steps: transaction filtering, rule creation, and finally the ranked list creation.

Given a query $Q = \{file_1, file_2, \ldots file_n\}$, the transaction filtering extracts from $\mathcal{H}$ those transactions that have the largest intersection with Q. More formally, transaction $T \in \mathcal{H}$ is kept if $|T \cap Q| = k$, where k is the size of the largest subset of Q seen in the history. In contrast, ROSE keeps only those transactions where $Q \subseteq T$.

Revisiting the example from Section IV, for q_2 the filtering removes none of the three transactions as each includes two of the files from q_2. However, for q_3 the final transaction, $\{c, d\}$, is removed by the filtering because $\{a\} \cap \{c, d\} = \emptyset$.

After filtering, the second step is rule creation. TARMAQ generates rules of the form $Q' \rightarrow x$, where x represents a single file and Q' is a subset of Q with $|Q'| = k$. Such a rule is created if and only if there exists a transaction $T \in \mathcal{H}$ such that $Q' \cup \{x\} \subseteq T$. Considering the example from Section IV, TARMAQ generates three rules for q_3: $\{a\} \rightarrow \{b\}$, $\{a\} \rightarrow \{c\}$, and $\{a\} \rightarrow \{d\}$.

Note that when $\mathcal{H}$ contains at least one transaction T such that $Q \subset T$, TARMAQ generates the same set of rules as ROSE. However, unlike ROSE, when Q is not contained in any T, TARMAQ generates rules whose antecedent is as close to Q as possible. In contrast, ROSE fails to generate any rules.

In the final step TARMAQ produces the ranked list $\mathcal{F}$ of recommended files. This is done by first sorting on the support of each rule, and in the case of ties (which are likely) on the confidence of each rule. The final list of files is then produced by mapping each rule to its consequent.

The rational for having a single file as the consequent includes both efficiency and utility. More general rules would involve Q' implying that a set of files should be included rather than a single file. Algorithms for generating such rules are computationally more expensive and may require a search through all possible subsets, which is an exponential computation and thus quickly becomes a performance concern. Furthermore, there is no loss of utility because in a software context it is sufficient to recommend files independently. The independence of software entities was exploited by ROSE, which also produces singleton consequents [30].

By looking for transactions that contain subsets of Q instead of Q itself, we obtain some recommendation evidence in the absence of transactions involving all of Q. For example, consider the situation where an engineer has modified files a, b, and c to fix a bug, but errantly forgot to modify file x. In this case, we want to mine the rule $\{a, b, c\} \rightarrow x$ with high confidence. However, this is not possible if c is new or has not been previously changed. Nevertheless, assuming that a, b, and x have frequently changed together before, TARMAQ produces the rule $\{a, b\} \rightarrow x$ with high confidence. Thus, in answer to RQ2, using the largest intersection, we can develop a technique that does not suffer from a lack of applicability.

VI. Evaluation

The evaluation compares TARMAQ, ROSE, and SVD by emulating a developer's need for a change-recommendation tool. To facilitate the three experiments we assume that the files of a transaction are related and evaluate performance by partitioning transactions into a query and an expected output. To describe the evaluation we first describe the subject systems used as test subjects in the experiments. We then detail the query generation process, the query evaluation, and finally, the implementation and execution environment.

A. *Subject Systems*

To assess the algorithms in a variety of conditions, we selected six systems having varying size and frequency of commits. Two of these are systems from our industry partners, Kongsberg Maritime (KM) and Cisco Norway. KM is a leading company in the production of systems for positioning, surveying, navigation, and automation of merchant vessels and offshore installations. Cisco Norway is the Norwegian division of Cisco Systems, a worldwide leader in the production of networking equipment. We validated TARMAQ in the common software platform KM uses in applications in the maritime and energy domain, and in the Cisco software product line for professional video conferencing systems. The other four systems are part of well known open-source projects, namely Apache HTTP Server, Linux Kernel, MySQL, and Git. Table I summarizes descriptive characteristics of the software systems used in the evaluation. The table shows that the systems we selected vary from medium to large size, with up to forty thousand different files committed in the transaction history. Furthermore, the oldest transactions from the system histories are fifteen years old in the case of KM. Note that all the systems are heterogeneous, i.e., they are implemented in more than a single programming language.

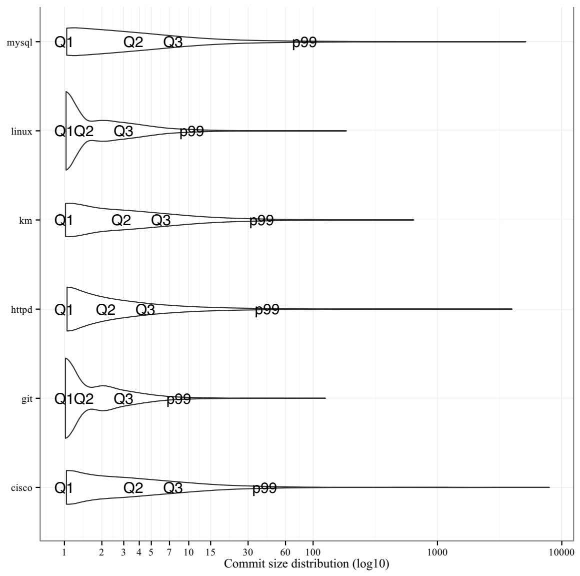

Fig. 2. Distribution of commit sizes for the different cases

A key aspect of each system is the way its authors interact with the version control system. This usage typically depends on the software process adopted by the developer's organization. For example, agile teams tend to frequently commit small incremental changes to the project artifacts following the *"commit early, commit often"* philosophy. On the other hand, in more traditional software processes developers might commit only on a monthly basis, with each commit changing a large number of files. To gain some understanding of the commit patterns used, Figure 2 shows violin plots of the six software systems. For each plot the x-axis shows commit size using a log scale. Each violin includes quartile markers, Q1, Q2, and Q3, and a marker for the 99^{th} percentile, p99. From these plots certain patterns emerge. For example, Linux and Git are dominated by small commits while MySQL and KM include considerably more larger commits. This variety of patterns is relevant to our evaluation.

B. *Determining transactions*

While our approach is not dependent on the versioning system used, adaptors still need to be written for each versioning system. These adaptors take a change-history from a specific versioning system as input, and output a set of transactions

TABLE I
CHARACTERISTICS OF THE EVALUATED SOFTWARE SYSTEMS

Software System	Unique Files	Avg. transaction size (# files)	History covered by 10 000 transactions	Languages used
MySQL	21854	10.1	2.34 years	C++ (54%), C (19%), JavaScript (17%), 23 other (10%)
Git	2141	1.9	4.2 years	C (45%), shell script (35%), Perl (9%), 14 other (11%)
Apache HTTP Server	7905	6.9	7.18 years	XML (56%), C (32%), Forth (8%), 19 other (4%)
Linux Kernel	9021	2.2	0.15 years	C (94%), 16 other (6%)
Kongsberg Maritime	35111	5.1	15.97 years	C++, C, XML, other build/config (% undisclosed)
Cisco Norway	41701	6.2	1.07 years	C++, C, C#, Python, Java, XML, other build/config (% undisclosed)

conforming to a common format. Our prototype tool uses this common format.

Depending on the versioning system however, it might not always be perfectly clear what constitutes a "transaction", as defined as "a set of related changes"; For each versioning system, a best approximation should be made. For example, in the Concurrent Versions System (CVS), changes are not explicitly grouped into commits, and it is therefore necessary to consider time-stamps (e.g., sliding time windows), log messages and developer identity to define transactions [29].

Another aspect that needs to be considered is if previous commits can be modified (i.e., history rewriting). If this is not possible in the versioning system, a developer who has forgotten to commit a relevant file needs to make an additional commit. Again, a sliding time window is a potential solution here.

For this paper, all chosen software-systems use the Git version control system.[8] Our adaptor for Git treats the files changed in one commit as one transaction. Merge commits however, are ignored through the use of the *'–no-merges'* option. Additionally it should be noted that Git supports history rewriting through the *'–amend'* option and the *'rebase'* command. These commands obviate the need for employing time windows in the Git adaptor. Finally, it should also be noted that only transactions from the *main branch* are considered.

C. Query Generation Process

Conceptually, a query Q represents a set of files that a developer changed since the last synchronization with the version control system. The key idea behind our evaluation is to generate, starting from a transaction T, a set of queries that emulate a developer errantly forgetting to update some subset of T.

The first step in the process is to select a set of transactions. The distribution plots are clearly skewed towards small commits. In fact, 75% of the commits have ten or fewer files while 90% have thirty or fewer files. For this reason, and because we assume that larger commits often consist of unrelated files committed together because of a directory reshuffle or license change, we follow the work of Zimmerman et al. [30] and remove transactions of more than thirty files.

From the remaining commits we sample forty commits for each transaction size between two and thirty. We use the resulting 1160 transactions to form the queries used to investigate how the three algorithms behave over a range of transaction sizes.

To mimic a developer forgetting files, we partition each of the 1160 transaction T into a non-empty query Q and a non-empty expected outcome $E \overset{\text{def}}{=} T \setminus Q$. In this way, we can evaluate the ability of an algorithm to infer E from Q. To investigate a range of query sizes, we generate one query for each size from one to $|T| - 1$. For example, for a transaction of size 4, we generate three queries of size 1, 2, and 3, whose expected outcomes thus have sizes 3, 2 and 1, respectively. Note that we do not sample commits of size one because they cannot be split into a query and expected outcome.

D. Query Evaluation

Evaluating the generated queries requires executing each query and then comparing the resulting ranked lists. To execute each query Q that was generated from transaction T, we use the 10 000 commits prior to T as the history. In these experiments 10 000 represents a balance between to short a history, which would lack sufficient connections, and to long a history, which is inefficient and can even be misleading when previously connected files are not longer connected.

Comparing the ranked lists produced by TARMAQ, ROSE, and SVD requires an appropriate performance measure. Prior work on association rule mining typically uses *precision* and *recall* as performance metrics. The *precision* of a recommendation is the ratio of the number of correct items recommended to the total number of items recommended. The *recall* of a recommendation is the ratio of the number of correct items recommended to the total number of correct items.

As a practical consideration, while precision and recall are designed for unordered results a recommendation tool's output is a *ranked list*. Consider the difference between a single correct recommendation occurring at the beginning of the list and a single correct recommendation occurring further down the list. These two have the same precision and recall. However, the correct recommendation at the beginning of the list is far more valuable, because it is far more likely to be considered. This is a well known phenomenon in information retrieval where, for example, internet searchers rarely look past the first ten results [9].

A more appropriate performance measure in the context of a ranked list is the *average precision* (AP). For query Q producing ranked recommendation list R, AP is defined as

[8]With the exception of Kongsberg Maritime, where a special adaptor was written.

$$AP(Q,R) \overset{\text{def}}{=} \sum_{k=1}^{|R|} P(k) * \triangle r(k) \quad (4)$$

where $P(k)$ is the precision calculated on the first k files in the list, (i.e., the precision@k) and $\triangle r(k)$ is the *change in recall* calculated only on the $k-1^{\text{th}}$ and k^{th} files, i.e., the fractional increase in true positives compared to the previous rank. Table II illustrates the computation of AP, $P(k)$, and $\triangle r(k)$ given the ranked list $[c, a, f, g, d]$ and the expected outcome $\{c, d, f\}$.

Finally to compare the performance of two tools, we use the *mean average precision* (MAP) computed over a set of queries. A tool producing a higher MAP value is, on average, producing better results. In addition, we compare the tools based on the total wall-clock time taken to execute a collection of queries.

E. Implementation and Execution Environment

We implemented all three algorithms in RUBY, using LAPACKE C to implement SVD through the NMatrix gem [24]. Our implementation is open-source and can be found online.[9] We performed the experiment executing the algorithms, one at a time, on a *c4.2xlarge* Amazon EC2 instance.[10]

VII. Results

This section addresses RQ3 by reporting objective measures of the data from our evaluation. Our interpretation of the data can be found in Section VIII. The results are organized according to the following questions:

- RQ3.1 What is the overall performance of all algorithms?
- RQ3.2 What is the performance on seen queries of all algorithms?
- RQ3.3 What is the performance on unseen queries of TARMAQ and SVD?
- RQ3.4 Is there a significant difference in performance on seen and unseen?
- RQ3.5 How do the algorithms perform on the individual software systems?

While not unexpected, we found that the average precision distribution for the different software systems and algorithms were not normally distributed, we therefore use the *Friedman Test*, a non-parametric test for differences between several samples.

[9] https://bitbucket.org/evolveIT/tarma
[10] https://aws.amazon.com/

TABLE II
Example of average precision calculation

Consider as relevant files: c, d, f

Rank (k)	File	$P(k)$	$\triangle r(k)$
1	c	1/1	1/3
2	a	1/2	0
3	f	2/3	1/3
4	g	2/4	0
5	d	3/5	1/3

$AP = 1/1 * 1/3 + 1/2 * 0 + 2/3 * 1/3 + 2/4 * 0 + 3/5 * 1/3 \approx 0.75$

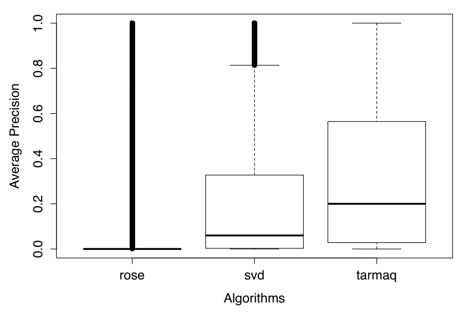

Fig. 3. Overall distribution of average precision for each algorithm

A. Overall Performance on Average Precision (RQ3.1)

The overall performance implies what is to be expected of each algorithm when the type of query is unknown (seen vs unseen), which would normally be the case. We test the following hypothesis:

- H_0 The average precision distribution generated by each algorithm is the same.
- H_1 The average precision distribution generated by each algorithm is different.

Figure 3 shows the overall distribution of average precision for each of the investigated algorithms. Here it is clear that ROSE cannot produce results for a significant number of the queries, and therefore ends up with a high percentage of 0 AP values. Furthermore we see that TARMAQ has a higher median than SVD and also a larger interquartile range.

A Friedman rank sum test[11] of the distributions yields a p-value < 0.00001, and we therefore reject H_0 and conclude that there is a significant difference between the algorithms.

Since we accept H_1, we can do a post-hoc test to actually look at which algorithms were different, to this end we use individual Wilcoxon signed rank tests[12]. Since we do multiple comparisons (tests) we have to use Bonferroni adjustment on what should be considered significant p-values. With three factor levels (three algorithms) it is sufficient with two directional tests to establish TARMAQ's place in the ordering of algorithms. The Bonferroni adjustment is given by dividing the desired alpha level by the number of performed tests, we thus get an adjusted alpha of 0.05/2 = 0.025.

We have the following hypotheses:

- H_0^{TvR} The average precision distribution generated by TARMAQ is less than that of ROSE.
- H_1^{TvR} The average precision distribution generated by TARMAQ is greater than that of ROSE.
- H_0^{TvS} The average precision distribution generated by TARMAQ is less than that of SVD.
- H_1^{TvS} The average precision distribution generated by TARMAQ is greater than that of SVD.

In Table III we provide the p-values of the two Wilcoxon tests. In both cases we can safely reject the null-hypothesis, and conclude that on overall, TARMAQ performs better than both ROSE and SVD.

[11] We used friedman.test in R.
[12] We used wilcox.test in R with: alternative = 'greater', paired = TRUE.

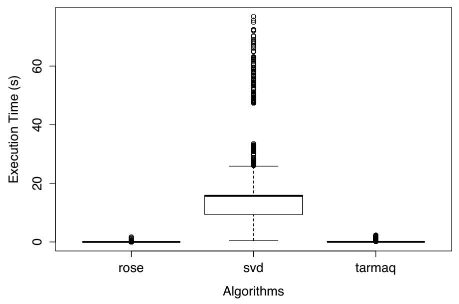

Fig. 4. Overall distribution of execution time for each algorithm. Note that ROSE executes faster because it does not produce results in all cases.

B. Overall Performance on Time (RQ3.1)

To investigate the execution cost of each algorithm, we perform the same analysis as we did for average precision. The overall execution time distributions can be found in Figure 4. At once we see that SVD have an obviously larger spread in execution times than ROSE and TARMAQ, and a Friedman rank sum test does indeed show significant difference between the distributions (p-value < 0.00001). The more interesting analysis is to compare ROSE and TARMAQ, as they have a much closer distribution. A Wilcoxon test showed that ROSE was significantly faster than TARMAQ (p-value < 0.00001), a large part of this however, might be attributed to the large number of queries where ROSE does not produce results (cannot find relevant transactions), and therefore does not need to perform any rule creation. The mean execution time for TARMAQ was 0.09 s, 0.003 s for ROSE, and 17.5 s for SVD.

C. Performance on seen queries (RQ3.2)

In this section and in Section VII-D we look at algorithm performance on either seen or unseen queries respectively. This division gives us a better view into how the two types of queries differ.

In Figure 5 we provide the distribution of average precision on all seen queries for all algorithms. We can see that, as expected, ROSE and TARMAQ have equal distributions on seen queries, while SVD performs worse. A Friedman test on the distributions was found significant (p-value < 0.00001), i.e., there is a difference between the algorithms. A test of significant difference between ROSE and TARMAQ is unneeded, as they produce exactly the same result here, but a Wilcoxon test of TARMAQ and SVD was found significant (p-value < 0.00001). We can therefore conclude that both TARMAQ and ROSE perform better than SVD on seen queries.

D. Performance on unseen queries (RQ3.3)

When a query consists of files that have never changed together before, we consider the query to be unseen. Since ROSE cannot produce recommendations on unseen queries, only TARMAQ and SVD are evaluated in this section. With only two subjects, we can use the Wilcoxon test directly without Bonferroni correction.

Figure 6 shows a boxplot of the average precision distribution of TARMAQ and SVD on unseen queries. A Wilcoxon test on TARMAQ and SVD distributions proved to be significant (p-value < 0.00001), we can therefore conclude that TARMAQ performs better than SVD on unseen queries.

TABLE III
P-VALUES FOR WILCOXON PAIRWISE MULTIPLE COMPARISONS

	TARMAQ	conclusion
ROSE	p-value < 0.00001	reject H_0^{TvR}
SVD	p-value < 0.00001	reject H_0^{TvS}

TABLE IV
THE MEAN AVERAGE PRECISION OF EACH ALGORITHM ON EACH SOFTWARE SYSTEM

	ROSE	SVD	TARMAQ	Friedman p-value
cisco	0.2063	0.1387	**0.3864**	< 0.00001
git	0.1042	0.1183	**0.2412**	< 0.00001
httpd	0.1035	0.2194	**0.3049**	< 0.00001
km	0.1711	0.3774	**0.3842**	< 0.00001
linux	0.1300	**0.5496**	0.3367	< 0.00001
mysql	0.1256	0.2062	**0.2958**	< 0.00001

E. Seen vs. unseen queries (RQ3.4)

In this section we investigate the difference in performance on seen and unseen queries, i.e., if the distributions shown in Figure 5 and Figure 6 significantly differ. To get a good overview we have plotted the overall distributions in Figure 7. To compare the distributions we use an unpaired Wilcox test, as the query population obviously is different for the seen and unseen queries. The test yields a p-value < 0.00001, and we can therefore conclude that unseen queries are significantly harder than the seen queries.

F. Performance of algorithms on individual systems (RQ3.5)

In this final results section we look closer at how performance varies over the individual software systems. To this end we present the *mean average precision* for each software system and algorithm combination in Table IV. The last column in Table IV lists the p-value for a Friedman test of equal distributions between the algorithms. In all cases the p-value was found significant. To determine which algorithm did the best for each software system, we performed post-hoc Wilcoxon tests on each algorithm pair. The best performing algorithm is shown in bold. For all software systems, with the exception of linux, TARMAQ is the top performer.

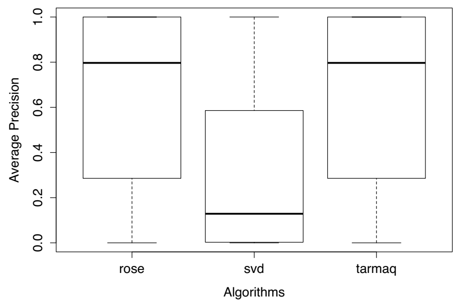

Fig. 5. Distribution of average precision on seen queries for each algorithm

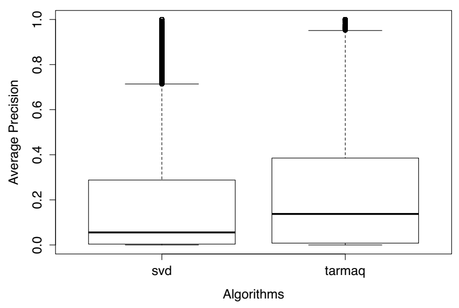

Fig. 6. Distribution of average precision on unseen queries for each algorithm

VIII. Discussion

In this section we will discuss some implications of the results presented in Section VII.

A. Overall Performance on Average Precision (RQ3.1)

To answer RQ3.1 we measured the overall performance of ROSE, TARMAQ and SVD over all software systems, regardless of the type of query (seen vs unseen). We argue that this is the most realistic comparison, as it is hard to know the query type a priori. We therefore claim that a good evolutionary coupling based algorithm for change impact analysis should be agnostic to the type of query. The same argument also holds in terms of queries on different software systems. While there might be systems where evolutionary couplings are harder to analyze because of developer practices, the general algorithm should not be too sensitive to specific software systems.

B. Overall Performance on Time (RQ3.1)

When implementing a change recommendation engine in an industrial process, the time needed to generate recommendations is an important factor to consider when it comes to industry adoption, as a slow tool can deter users from frequent use.

In Section VII-B we found that ROSE and TARMAQ executed in a fraction of a second on average, while the SVD algorithm had an average execution time of 17 seconds. We observe that SVD's execution time is unsatisfactory in a change-recommendation context, although it is certainly acceptable in other applications, such as for test-selection.

We also argue that the difference in execution time between ROSE and TARMAQ is significant only statistically, not practically, and to this end both can be considered as providing real-time feedback.

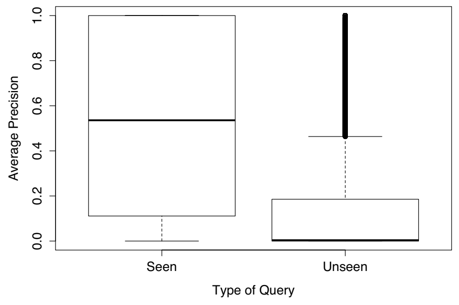

Fig. 7. Distribution of average precision on seen and unseen queries

TABLE V
TOP10-TRUEPOSITIVE: THE PERCENTAGE OF TIMES THAT EACH ALGORITHM PREDICTS AT LEAST ONE CORRECT FILE IN THE TOP 10 ON EITHER SEEN OR UNSEEN QUERIES

	seen	unseen
ROSE	92%	0%
SVD	56%	50%
TARMAQ	92%	65%

C. Performance on seen and unseen queries (RQ3.2-4)

In addition to investigating the overall performance, we also considered performance on seen and unseen queries separately. The discussion in Section VII-C and Section VII-D states the best-performing algorithm based on average precision, but does not give sufficient insights on the performance of each algorithm in practice. While the average precision is a good evaluator of the complete recommendation list produced by the algorithms, for this purpose, we introduce a quantification of practical use, which is easier to interpret than average precision. We define *top10-truepositive* as the number of times an algorithm correctly predicts at least one file in the top-10 of its recommendation list. The top10-truepositive metric, given in percentage, is shown in Table V.

We see that in over 90% of cases, at least one correctly predicted file will be in the top 10 for ROSE and TARMAQ on seen queries. For SVD the same is true only in about 50% of cases.

For all algorithms, performance measured by top10-truepositive degrades on unseen queries, which should be expected; however, given the average precision degradation we also saw in Section VII-D. TARMAQ and SVD are closer in performance here, but for about every 6th query, TARMAQ will predict at least one correct file in the top 10 that SVD did not catch.

D. Performance of algorithms on individual systems (RQ3.5)

In our final evaluation in Section VII-F we looked at each algorithm's performance on the individual software systems. Here we found that, with the exception of linux, TARMAQ performed best on all systems. Finally, we remark the diversity expressed by the different system in terms of number of unique files, size of the average transaction, time between first and last transaction in the history, and the number of different languages used (Table I), and we argue that TARMAQ's performance on such diverse systems is satisfactory.

However, in the future we plan to achieve a better understanding of how individual software system characteristics might affect TARMAQ, especially in the case of linux where SVD achieves higher average precision than TARMAQ. A possible explanation for this result is the relatively short time frame of the history used with linux, where the older commits are only a month old. Future work will consider if this short time frame could cause TARMAQ to assign low confidence and support to correct association rules.

E. Threats to Validity

We identified a set of threats that could affect the construct, internal, and external validity of our experimental results.

1) Threats to Construct Validity: One main threat could negatively affect the extent to which our experimental design measures the effectiveness of TARMAQ for the purpose of generating change recommendations.

Using Evolutionary Coupling for Software Change Impact Analysis: The main underlying assumption behind our experimental design is that evolutionary coupling infers meaningful dependencies from the transaction histories, which can in turn be used to generate effective change recommendations. While this has not been proven on a universal basis, research in the field showed that evolutionary coupling is an effective strategy for software change impact analysis (Section IX).

2) Threats to Internal Validity: One main threat could negatively affect the conclusions on the cause-effect relationships derived from the experimental results.

Algorithms implementation: We compared the effectiveness of TARMAQ to that of the most commonly used alternatives for deriving change recommendations based transaction history, namely ROSE and SVD. However, we could not find publicly available implementations of these algorithms, and we re-implemented them based on the specification in the papers where such algorithms were introduced [30, 20]. In particular, we implemented the calculation of the decomposition matrices of SVD using standard linear algebra libraries, which ensure number overflows are properly avoided. Repeated executions on abstract examples show that our implementations of TARMAQ, ROSE and SVD are correct.

3) Threats to External Validity: Four main threats could negatively affect the generalizability of the conclusions drawn.

Variation in software systems: We validated TARMAQ in two industrial systems from our user partners, and four large open source systems (Section VI-A). These systems considerably vary in size and frequency of transactions, and have been selected in order to investigate the effectiveness of TARMAQ in a variety of software systems. However, even though our selected software systems display good variation, we very likely have not captured all variations.

Query Generation Process: When generating queries from the systems histories, we removed transactions larger than 30 files that could contain dependent files (Section VI-C). However, as mentioned earlier, similar experimentation in the literature does not consider these large transactions, because on average they are likely to contain for the largest part unrelated files which would introduce noise when inferring dependencies [30].

Incomplete data from our industrial partners: We were not able to include the full history of transactions in the Kongsberg Maritime system. This is because such transactions were parsed from semi-structured free-text fields, which in a small number of cases contained incomplete data. However, the transactions we excluded for this reason constitute less than 0.05% of the total history of the KM system.

Length of history: We evaluated all algorithms using the last 10000 commits from each software system, rather than the entire available history. While this ensures consistency over the different systems, the included length might have also affected our evaluation of seen and unseen queries. We had to limit the number of commits for two reasons. (1) This was the number of commits provided to us by our industry partners, and we wanted to be consistent to this number also for the open-source systems. (2) For the SVD algorithm, the overhead for generating singular value decompositions of large co-change matrices proved to be very high, and we saw the need to limit the number of commits to keep experiment execution times to a manageable level.

IX. Related Work

We distinguish related work on association rule mining, change impact analysis, evolutionary coupling, and mixed approaches.

Association Rule Mining: Since Agrawal et al's seminal paper introducing the concept [1], many techniques have been proposed, generally aimed at improving execution and memory efficiency. The most widely known include Apriori [2], which uses an efficient pre-computation of rule generation candidates, Eclat [27], which partitions the search space into smaller independent subspaces that can more efficiently be analyzed, and FPGrowth [11], which uses a compact tree structure (the FPtree) to encode the database and enable frequent patterns mining without candidate generation. All these apply frequent pattern mining on the complete dataset. A refinement is brought by so called *targeted association rule mining* techniques, which focus the generation of rules on a particular *query* supplied by the user [23, 10, 16]. These techniques filter transactions that are not related to the query from the database used for rule generation, enabling a drastic reduction of execution time [23]. Furthermore, these approaches are very suitable for evolving data like software change histories, because association rules are generated on a per-query basis, and are always "up-to-date" with the latest repository status. A more detailed discussion of advances in pattern mining is outside the scope of this paper. For more details, we refer to a recent survey by Silva et al. [21].

Change Impact Analysis: Zimmerman et al. introduce ROSE [30], the work most closely related to ours. ROSE applies targeted association rule mining to the problem of deriving developer change recommendations for a user specified query. It uses constraints to filter out transactions that do not contain any of the files of the query. The same constraint is used to generate only rules whose antecedent contains *all* the files in the query. As discussed in Section IV, the downside of this approach is that ROSE generates *no* rules if no transaction in the history is a superset of the query.

Ying et al. [26] describe a technique that mines frequent patterns in the change history of a system to recommend potentially relevant source code to a developer that is performing a software maintenance task. Their algorithm uses a *FPtree* structure to efficiently represent the set of files frequent in the history [11]. Similar to ROSE, this algorithm generates no rules if no transaction in the history is a superset of the query, i.e. it suffers from the limitation discussed in Section IV.

Sherriff et al. [20] present an approach to change recommendation that identifies couplings of related files using a SVD of the co-change matrix. This matrix encodes the number of times any two files changed in the same transaction. This algorithm does not suffer from the limited applicability discussed in Section IV, but it is rather computationally expensive, as demonstrated by the results of our empirical evaluation (Section VII). Moreover, the SVD has to be recomputed after an update of the change history.

Evolutionary Coupling: There is a body of work on identifying *evolutionary coupling* (also referred to as *logical coupling*). All these have in common that they are based on some measure of co-change. Example measures include course-, and fine-grained co-change information [7, 3, 19], code-churn [8], and interaction with an IDE [28].

Gall et al. used release information to detect logical coupling between 20 releases of a large Telecommunication Switching System [7]. They later continued this analysis to discover architectural weaknesses in source code (e.g., amount of modularization) [8]. The coupling were primarily found through analyzing sequences of releases in which modules were changed together. Furthermore, couplings were also identified on a class level through analyzing when and who (author/date) that made class changes.

Hassan and Holt present several heuristics for predicting change-propagation/ripple effects that result from source code changes [12]. In addition to evolutionary coupling (described as "historical co-changes" in the paper), three other heuristics were also investigated, whereas one used static dependencies such as *Call/Use/Define* relations, and another used code-layout to identify couplings. Of all 4 heuristics, the use of evolutionary couplings gave the highest recall score, meaning it correctly identified the most couplings (avg. 87%).

Jafar et al. [13] perform an exploratory study on co-changes at file level granularity. They introduce two timing related patterns for co-changes that can help to more accurately mine transactions in a change history.

Mixed Approaches: Hipikat [6] integrates various developer related artifacts, such as change history, email discussions and issue tracking systems. Vector-based information retrieval techniques are used to mine relations between artifacts. Additional relations are created using heuristics, such as the matching of issue IDs in commit messages to issue reports in bugzilla. Hipikat uses this cross-indexed *project memory* to recommend relevant artifacts for a task , either directly from a query, or automatically based on a developer's working context (e.g., documents open in an IDE).

Kagdi et al. [15] combine history based evolutionary coupling with so called *conceptual coupling* which is derived using information retrieval techniques on a single version (i.e, a release) of a software system. They show that the combination of these two techniques provides statistically significant improvements in accuracy over the individual techniques. Mondal et al. [17] combine association rule mining with *change correspondence*, a measure for the extent to which identifiers and constants in co-changed entities overlap. This is a lightweight form of conceptual coupling, which is then used to prioritize association rules, as a more source-code aware version of the standard support and confidence measures.

Although originally developed for other contexts, there is no reason why these orthogonal measures and additional sources could not be used in combination with the technique we propose in this paper, and achieve similar benefits as in their original application.

X. Concluding Remarks

With new techniques and data-sources, progress is being made on improving Change Impact Analysis (CIA). The use of evolutionary coupling as a driver of CIA is a promising direction that can address some caveats of traditional static/dynamic dependency analysis. In particular, the use of evolutionary coupling is inherently language agnostic, and in general can potentially find couplings where static/dynamic approaches cannot find a coupling because of a lack of explicit data/control flow. This is a considerable advantage given the increasing heterogeneity of today's software systems.

In this paper we present an algorithm (TARMAQ) for CIA using a *generalized analysis* of evolutionary coupling with respect to some change-scenario (changed files). The use of TARMAQ for CIA promises a best effort analysis of the evolutionary coupling given any change-scenario. The contributions of this paper are the following: (1) We provided a classification of two different change-scenarios for change impact analysis. (2) We provided an empirical evaluation of the frequency of these change-scenarios. (3) We introduced an algorithm (TARMAQ) that can analyze the evolutionary coupling of any change-scenario, and therefore can support CIA for most change-scenarios. (4) We provided a comprehensive evaluation of TARMAQ on two industrial software systems from our industry partner and four open source systems.

Directions for Future Research: In future work we would like to address the following: (1) We plan to conduct a larger empirical evaluation of TARMAQ, both in the number of software systems and in the number of evaluated factors. We would, for example, like to explore the effect of history-size and query-size on performance, and attempt to classify software systems in terms of how applicable our approach might be. (2) We also plan to explore methods for further increasing overall performance, and especially performance on unseen queries. (3) Third, we hope to directly compare CIA based on evolutionary coupling with static/dynamic dependency analysis. (4) Next we plan to apply TARMAQ on other problems, for example test-selection. (5) Finally, future work will explore existing alternative interestingness measures that might replace support/confidence, and see how they perform.

Acknowledgement: This work is supported by the Research Council of Norway through the EvolveIT project (#221751/F20) and the Certus SFI. Dr. Binkley is supported by NSF grant IIA-1360707 and a J. William Fulbright award.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. "Mining association rules between sets of items in large databases". In: *ACM SIGMOD International Conference on Management of Data*. ACM, 1993, pp. 207–216.

[2] R. Agrawal and R. Srikant. "Fast Algorithms for Mining Association Rules". In: *20th International Conference on Very Large Data Bases (VLDB)*. 1994, pp. 487–499.

[3] D. Beyer and A. Noack. "Clustering Software Artifacts Based on Frequent Common Changes". In: *13th International Workshop on Program Comprehension (IWPC)*. IEEE, 2005, pp. 259–268.

[4] D. Binkley et al. "ORBS and the Limits of Static Slicing". In: *IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2015, pp. 1–10.

[5] G. Canfora and L. Cerulo. "Impact Analysis by Mining Software and Change Request Repositories". In: *11th IEEE International Software Metrics Symposium (METRICS)*. IEEE, 2005, pp. 29–37.

[6] D. Cubranic et al. "Hipikat: a project memory for software development". In: *IEEE Transactions on Software Engineering* 31.6 (2005), pp. 446–465.

[7] H. Gall, K. Hajek, and M. Jazayeri. "Detection of logical coupling based on product release history". In: *IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 1998, pp. 190–198.

[8] H. Gall, M. Jazayeri, and J. Krajewski. "CVS release history data for detecting logical couplings". In: *Sixth International Workshop on Principles of Software Evolution (IWPSE)*. IEEE, 2003, pp. 13–23.

[9] L. A. Granka, T. Joachims, and G. Gay. "Eye-tracking analysis of user behavior in WWW search". In: *Proceedings of the 27th annual international conference on Research and development in information retrieval - SIGIR '04*. New York, New York, USA: ACM Press, 2004, p. 478.

[10] A. Hafez, J. Deogun, and V. V. Raghavan. "The Item-Set Tree: A Data Structure for Data Mining". In: *DataWarehousing and Knowledge Discovery*. Vol. 1676. LNCS. Springer, 1999, pp. 183–192.

[11] J. Han et al. "Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach". In: *Data Mining and Knowledge Discovery* 8.1 (2004), pp. 53–87.

[12] A. Hassan and R. Holt. "Predicting change propagation in software systems". In: *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 2004, pp. 284–293.

[13] F. Jaafar et al. "An Exploratory Study of Macro Co-changes". In: *Working Conference on Reverse Engineering (WCRE)*. IEEE, Oct. 2011, pp. 325–334.

[14] M.-A. Jashki, R. Zafarani, and E. Bagheri. "Towards a more efficient static software change impact analysis method". In: *8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. ACM, 2008, pp. 84–90.

[15] H. Kagdi et al. "Blending conceptual and evolutionary couplings to support change impact analysis in source code". In: *Proceedings - Working Conference on Reverse Engineering, WCRE*. 2010, pp. 119–128.

[16] M. Kubat et al. "Itemset trees for targeted association querying". In: *IEEE Transactions on Knowledge and Data Engineering* 15.6 (Nov. 2003), pp. 1522–1534.

[17] M. Mondal, C. K. Roy, and K. A. Schneider. "Improving the detection accuracy of evolutionary coupling by measuring change correspondence". In: *IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 2014, pp. 358–362.

[18] X. Ren et al. "Chianti: a tool for change impact analysis of java programs". In: *ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2004, pp. 432–448.

[19] R. Robbes, D. Pollet, and M. Lanza. "Logical Coupling Based on Fine-Grained Change Information". In: *Working Conference on Reverse Engineering (WCRE)*. IEEE, 2008, pp. 42–46.

[20] M. Sherriff and L. Williams. "Empirical Software Change Impact Analysis using Singular Value Decomposition". In: *International Conference on Software Testing, Verification, and Validation (STVV)*. IEEE, 2008, pp. 268–277.

[21] A. Silva and C. Antunes. "Constrained pattern mining in the new era". In: *Journal of Knowledge and Information Systems* online (July 2015), pp. 1–28.

[22] R. Srikant and R. Agrawal. "Mining generalized association rules". In: *21st International Conference on Very Large Data Bases (VLDB)*. 1995, pp. 407–419.

[23] R. Srikant, Q. Vu, and R. Agrawal. "Mining Association Rules with Item Constraints". In: *International Conference on Knowledge Discovery and Data Mining (KDD)*. AASI, 1997, pp. 67–73.

[24] J. O. Woods and the Ruby Science Foundation. *NMatrix: A dense and sparse linear algebra library for the Ruby programming language*. 2013.

[25] A. R. Yazdanshenas and L. Moonen. "Crossing the boundaries while analyzing heterogeneous component-based software systems". In: *IEEE International Conference on Software Maintenance (ICSM)*. ICSM '11. Washington, DC, USA: IEEE, Sept. 2011, pp. 193–202.

[26] A. Ying et al. "Predicting source code changes by mining change history". In: *IEEE Transactions on Software Engineering* 30.9 (2004), pp. 574–586.

[27] M. Zaki. "Scalable algorithms for association mining". In: *IEEE Transactions on Knowledge and Data Engineering* 12.3 (2000), pp. 372–390.

[28] M. B. Zanjani, G. Swartzendruber, and H. Kagdi. "Impact analysis of change requests on source code based on interaction and commit histories". In: *Working*

Conference on Mining Software Repositories (MSR) (2014), pp. 162–171.

[29] T. Zimmermann. "Preprocessing CVS data for fine-grained analysis". In: *International Workshop on Mining Software Repositories (MSR)*. Vol. 2004. IEE, 2004, pp. 2–6.

[30] T. Zimmermann et al. "Mining version histories to guide software changes". In: *IEEE Transactions on Software Engineering* 31.6 (2005), pp. 429–445.