Integrated Impact Analysis for Managing Software Changes

Malcom Gethers[1], Bogdan Dit[1], Huzefa Kagdi[2], Denys Poshyvanyk[1]
[1]Computer Science Department
The College of William and Mary
Williamsburg, VA 23185
{mgethers, bdit, denys}@cs.wm.edu

[2]Department of Computer Science
Wichita State University
Wichita, KS 67260-0083
kagdi@cs.wichita.edu

***Abstract* — The paper presents an adaptive approach to perform impact analysis from a given change request to source code. Given a textual change request (*e.g.*, a bug report), a single snapshot (release) of source code, indexed using Latent Semantic Indexing, is used to estimate the impact set. Should additional contextual information be available, the approach configures the best-fit combination to produce an improved impact set. Contextual information includes the execution trace and an initial source code entity verified for change. Combinations of information retrieval, dynamic analysis, and data mining of past source code commits are considered. The research hypothesis is that these combinations help counter the precision or recall deficit of individual techniques and improve the overall accuracy. The tandem operation of the three techniques sets it apart from other related solutions. Automation along with the effective utilization of two key sources of developer knowledge, which are often overlooked in impact analysis at the change request level, is achieved.**

To validate our approach, we conducted an empirical evaluation on four open source software systems. A benchmark consisting of a number of maintenance issues, such as feature requests and bug fixes, and their associated source code changes was established by manual examination of these systems and their change history. Our results indicate that there are combinations formed from the augmented developer contextual information that show statistically significant improvement over stand-alone approaches.

I. INTRODUCTION

Software change requests, such as bug fixes and new features, are an integral part of software evolution and maintenance. Effectively supporting software changes is essential to provide a sustainable high-quality evolution of large-scale software systems, as realizing even a slight change may not be always straightforward. Software-change impact analysis, or simply impact analysis (IA), has been recognized as one such key maintenance activity. IA aims at estimating the potentially impacted entities of a system due to a proposed change [7]. The applications of IA include cost estimation, resource planning, testing, change propagation, managing ripple effects, and traceability [8, 16, 22, 26, 27, 29-31, 35].

In several realistic settings, change requests are typically specified in natural language (*e.g.*, *English*). They include bug reports submitted, by programmers or end users, during the post-delivery maintenance of a product. In the distributed collaborative software development environments, such as the open source software model, change requests are typically managed with issue tracking systems (*e.g.*, *Bugzilla*). These change requests may need to be resolved with the appropriate changes to relevant source code. It is not uncommon in such projects to receive numerous change requests daily that need to be resolved in an effective manner (*e.g.*, within time, priority, and quality factors) [3, 20]. Another factor that adds to the challenge of IA in the maintenance environment is the regular absence of useful intermediate artifacts (*e.g.*, design documents and pertinent traceability information) between the abstractions levels of change requests and source code. It is a common maintenance scenario in which a change request, described in the natural language, is the only source of information available to perform IA and an automatic technique must operate in such a situation. Moreover, developers or development environments may have accumulated valuable sources of information in the context of solving a specific change request (or past change requests). On the other hand, developers cannot be expected to manually provide such information all the time.

In this paper, we present a novel approach for IA that automatically adapts to the specific maintenance scenario at hand. We consider scenarios in which the change request is available at the minimum and is the source of *focus*. Additional forms of developer knowledge may be available or not in the *context* of this change request. Two quantifiable forms of the developer knowledge are considered: a verified source code entity to start performing the change (*e.g.*, a relevant source code method) and run-time information pertinent to the features in change request (*i.e.*, an execution trace for a feature specific scenario). Developers may narrow down to at least one entity to change, e.g., using feature location techniques [23, 28], previous project experience, and/or tacit software development knowledge. Also, developers typically attempt to reproduce the problem reported in the change request; a typical activity during the issue triage process [3, 20]. In some cases, call stacks of the failure are also available in the bug reports.

Our approach uses a scenario-driven combination of information retrieval (IR), dynamic analysis, and mining software repositories techniques (MSR). We chose a history-based mining technique, as we share a prevalent view in MSR that the information in software repositories is an extension of the collective developer or development knowledge [5]. Given a textual change request, an IR (*e.g.*, Latent Semantic Indexing or simply, LSI) indexed single release of source code is used to estimate the impact set. Should the execution information be made available for the

same snapshot associated with the change request, methods in the trace are also obtained. A combination of IR and dynamic analyses is favored over IR to estimate the impact set in such cases.

Furthermore, should a verified start entity of change be available, evolutionary couplings are mined from the commits in software repositories that occur before the snapshot of code used for IR-based indexing (and dynamic analysis). A combination of IR and evolutionary coupling analyses is favored over IR to estimate the impact set in such cases. Evolutionary couplings are used in our approach, as they are derived from the actual changes to artifacts across multiple releases, rather than estimations that are based on the analysis of various structural and semantic dependencies between them in a single snapshot of a system. Also, the commits embody part of the developer's knowledge and experience [2]. The version history may contain domain-specific "hidden" links that are manually created and maintained (*e.g.*, database schema changes [24]), which traditional program-analysis methods may fail to uncover [35]. When both forms of additional developer-information context are available, a combination of IR, dynamic information, and evolutionary couplings supersedes others to estimate the impact set.

Our hypothesis is that such combinations would help counter the accuracy, *i.e.*, precision and/or recall, deficit of individual techniques and improve upon the accuracy collectively. To validate our approach, we first created a benchmark of change requests and their associated source code changes by manual examination of open source projects *ArgoUML, jEdit, muCommander and JabRef.* Empirical evaluation of our approach on this benchmark shows statistically significant gains in precision and recall up to 17% and 41% respectively.

II. Related Work

Several IA approaches ranging from classical static and dynamic analysis techniques [8, 22, 26, 27, 30, 31] to the recent unconventional approaches, such as those based on IR [16, 29] and MSR [15, 19, 35], exist in the literature. In the next subsections, we review some of the related approaches to provide a breadth of the IA solutions; the intent is not to exhaustively discuss every single technique.

A. Software Change IA via Static & Dynamic Analyses

Depending on the type of information available, impact analysis is traditionally performed using static program analysis [7, 9], dynamic program analysis [22, 25, 26] or a combination of these techniques [30]. Static program analysis relies solely on the structure of the program and the relationship between program elements, at different levels of granularity, whereas dynamic program analysis takes into account information gathered from program execution.

Law and Rothermel [22] introduced *PathImpact*, a dynamic IA technique based on whole path profiling (*i.e.*, when a method *m* is changed, any method that calls *m* or is called after *m* is added to the set of potentially impacted methods). Orso et al. [25] proposed *CoverageImpact*, a technique that combines forward static slicing with respect to a modified program element (*i.e.*, method) with dynamic information collected from program instrumentation. Orso et al. [26] conducted an experiment to analyze the tradeoff between in terms of cost and precision for two dynamic IA techniques, *CoverageImpact* and *PathImpact*. These two techniques require data gathered during program execution using various inputs or test cases. Using this information, their technique determines the impact set by locating the program elements that were executed simultaneously with the given program elements used as seeds. It should be noted that we are not using any of these algorithms in this paper. In our solution execution information is rather used as a filter to eliminate methods that were not executed and, as a result, are less likely to be relevant to the change request.

Other tools, such as *Chianti* [30] support IA by analyzing the changes between two versions of a program (i.e., static information) and a set of tests that execute parts of a program (i.e., dynamic information). Using this information, Chianti suggests a set of regression or unit tests that might have been affected by the changes between the two versions of the program. *JRipples* [9] is an Eclipse plug-in that relies on static information to guide developers while manually locating the impact set, by keeping track of impacted program elements. The comprehensive summary on using dynamic analysis to support program comprehension including IA is reported in Cornelissen et al. [12].

B. Software Change IA via Information Retrieval

IR methods were proposed and used successfully to address tasks of extracting and analyzing textual information in software artifacts, including change impact analysis in source code [10, 19, 29].

Existing approaches to IA using IR operate at two levels of abstraction: change request [10] and source code [19, 29]. In the first case, the technique relies on mining and indexing the history of change requests (*e.g.*, bug reports). In particular, this IA method utilizes IR to link an incoming change request description to similar past change requests and file revisions that were modified to address them [10, 34]. While this technique has been shown to be relatively robust in certain settings, it is entirely dependent on the history of prior change requests. In cases where textually similar change requests cannot be identified (or simply do not exist), the technique may not be able to identify relevant impact sets. Also, these works show that a sizeable change request history must exist to make this approach operational in practice, which may limit the effectiveness. The work in [16] relates to our approach in the use of lexical (textual) clues from the source code to identify related methods.

The other set of techniques to IA that use IR operates at the source code level and requires a starting point (*e.g.*, a source code method that is likely to be modified in response to an incoming change request) [19, 29]. This approach is based on the hypothesis that modules (or classes) in software systems are related in multiple ways. The evident and most explored set of relationships is based on data and control dependencies; however, the classes can be also related conceptually (or textually), as they may contribute to the implementation of similar domain concepts. This

information is derived using IR-based analysis of textual software artifacts that are derived from a single version of software (*e.g.,* comments and identifiers in source code).

Our previous work [19] was consistent with earlier usages of IR in IA [29]; however, it was limited to IA at the source-code level staring point, and the work presented in this paper operates at the change-request level as a starting point. In this paper, we apply IR for IA similar to how it has been used in the context of feature location [23, 28], which is different from two aforementioned approaches. We also use this technique as our *baseline* in our adaptive solution. We do not discuss applications of IR-based techniques in the context of other maintenance tasks due to space limitations; however, such an overview can be found elsewhere [6].

C. Software Change IA via Mining Software Repositories

The term MSR has been coined to describe a broad class of investigations into the examination of software repositories (*e.g.*, *Subversion* and *Bugzilla*). We refer the interested readers to Kagdi et al. [18] literature survey, and Xie's online bibliography and tutorial[1] on MSR. We now briefly discuss some representative works in MSR for mining of evolutionary couplings.

Zimmerman et al. [35] used *CVS* logs for detecting evolutionary coupling between source code entities. Association rules based on itemset mining were formed from the change-sets and used for change-prediction. Canfora et al. [10] used the bug descriptions and the *CVS* commit messages for the purpose of change prediction. An information retrieval method is used to index the changed files, and commit logs, in the *CVS* and the past bug reports from the *Bugzilla* repositories.

In addition, conceptual information has been utilized in conjunction with evolutionary data to support several other tasks, such as assigning incoming bug reports to developers [3, 17], identifying duplicate bug reports [33], estimating time to fix incoming bugs [34] and classifying software maintenance requests [13].

Traditionally, given a proposed change in a given source entity, other change-prone source code entities are estimated using static and/or dynamic analysis based models of a specific snapshot of source code. Traditional techniques largely performed impact analysis at the same level of abstraction and that too mostly on source code. Supporting IA at the change request level has been suggested only recently [10]; an advent of applied IR and MSR methods has provided a renewed interest in cross abstraction IA.

Our combined approach is different from other previous approaches, including those using IR and MSR techniques, for IA that rely solely on the historical account of past change requests and/or source code change history. Our approach is not dependent on past change requests (*e.g.*, repositories of past bug reports, which may not be always available), and only requires source code of a single complete release of the system, source code change history, and access to execution and tracing environment (*e.g.*, JPDA or TPTP). To the best of our knowledge, ours is the only approach that utilizes such a combination for performing IA from change request to source code without the need for a bug/issue history. The selective use of dynamic and evolutionary information along with the textual information has not been used before. Our approach builds on existing solutions, but synergizes them in a new holistic technique.

[1] https://sites.google.com/site/asergrp/dmse

III. An Integrated Approach to Impact Analysis at Change Request Level

Our framework for impact analysis is based on the possible degree of automation and developer augmented information that may be available in a given maintenance scenario. In several realistic settings, change requests are typically specified in natural language (*e.g.*, *English*). It is reasonable to assume that change requests, in several cases, are the only source of available information to conduct the needed maintenance. In such a situation, a high degree of automation in estimating the impact set can be achieved by taking the textual view of source code and applying IR techniques, which are an organic fit to automatic text analysis. This component of our framework assumes that there is no developer or maintenance environment supplied information available. Our framework operates in this default mode, which has the highest degree of automation and the least level of developer supplied information. We refer to this default configuration as IR_{CR}.

The maintenance scenario may not necessarily be as ascetic as depicted in the default IR mode. In several situations, additional pieces of valuable information are also available. We consider two such developer-augmented information cases: 1) a developer somehow narrows down to at least one verified entity that needs a change (*e.g.*, from previous experience of performing similar changes) – seed entity, 2) a developer has executed the feature, inferred by reading the textual change request, and collected the run-time information – executed methods, (*e.g.*, to verify if the issue that was reported can be replicated or collected from the call stack of a failure). For the first case, our framework provides a component $Hist_{seed}$, which mines the past commits (change history) of software entities to estimate the impact set. This component provides medium levels of automation and human intervention is in selecting a starting point of change; then a data mining technique is used to compute the impact set automatically. For the second case, our framework provides the component that uses the methods executed in the run-time scenario. This component requires the most human involvement and the lowest level of automation. We refer to this component as Dyn_{CR}.

Our framework employs the best effort paradigm in an adaptive manner – it selectively employs the best-fit components depending on the type of developer-supplied information before resorting to the default mode. For example, when a seed entity is available along with the change request, a combination of the components IR_{CR} and $Hist_{seed}$ is engaged. Similarly, when the dynamic information is available along with the change request, a combination of the components IR_{CR} and Dyn_{CR} is selected. The premise of our approach is that any combination that involves the human augmented information and (highest or medium)

automation would provide a better impact set than those based on automated components alone.

The impact analysis model presented here defines several sources of information, the analyses used to derive the data, and how the information can be combined to support impact analysis at the change request level.

A. *Analyzing Textual Information via IR*

Textual information in source code and software repositories (*e.g.*, changes requests in Bugzilla), reflected in identifiers and comments, encodes problem domain information about a software project. This unstructured information can be used to support impact analysis through the use of IR techniques [10]. IR works by comparing a set of artifacts (*e.g.*, source code files) to a query (*e.g.*, a change request) and ranking these artifacts by their relevance to the query. IR_{CR} follows five main steps [23]: (1) building a corpus, (2) natural-language processing (NLP), (3) indexing, (4) querying, and (5) estimating an impact set.

(1) Building a corpus. To use IR on software, a document granularity needs to be defined, so that the corpus can be build. A document contains all the text found in a contiguous section of software artifact, such as a method, class, or package. A corpus consists of all such documents (artifacts). For instance, for impact analysis, we employ the method-level granularity for documents that include contiguous text of each method in a project.

(2) NLP. Once the corpus is created, it is preprocessed using NLP techniques. For source code, operators and programming language keywords are removed. Additionally, identifiers and other compound words are split (*e.g.*, "*impactAnalysis*" becomes "*impact*" and "*analysis*") [14]. Finally, stemming is performed to reduce words to their root forms (*e.g.*, "*impacted*" becomes "*impact*").

(3) Indexing the corpus with IR. The corpus is used to compile a term-by-document matrix (TDM). The matrix's rows correspond to the words from identifiers or comments in the corpus, and the columns represent methods from source code. A cell $m_{i,j}$ in the TDM holds a measure of the weight or relevance of the i^{th} word in the j^{th} method. In particular we use a more complex measure, such as term frequency-inverse-document frequency. Singular Value Decomposition (SVD) [32] is then used to reduce the dimensionality of the TDM by exploiting the co-occurrence of related words across source code methods.

(4) Running a query. The title and description of an incoming change request serve as an input to this technique, that is a query. An example of such a query is the bug #2472 [2] reported in *ArgoUML v0.22*. The query is formulated from its description "*Wrong keyboard focus in Settings dialog after close & reopen […]*".

(5) Estimating an impact set. In the SVD model, each method corresponds to a vector. The query (or change request) is also converted to a vector-based representation, and then the cosine of the angle between the two vectors is used to measure the similarity of the source code method to the change request. The closer the cosine is to one, the more textually similar the method is to the change request. A cosine similarity value is computed between the change request and all the methods in the source code, and then these methods are sorted by their similarity values. The top results from this list constitute an estimated impact set.

For the input query from the bug #2472, the IR_{CR} technique returns a ranked list of methods according to their similarity values in descending order. The top methods in this ranked list are considered based on a cut point, which establishes the size of the estimated impact set. Now, the question is how accurate are these IR_{CR} estimated impact sets. We manually examined the source code methods that were changed to address/fix a specific bug, which we refer to as a gold set. We identified 16 methods that are relevant to the change request for the bug #2472 (*i.e.*, gold sets). When comparing the IR_{CR} estimated impact set with its gold set, the relevant methods appeared at positions 2, 16, 30, 37, 52, 56, 57, and so on. This example shows that although IR can help identify the real impact set, it might produce results that require an examination of several candidates; in some cases it may not be quite practical (*e.g.*, bug #2472).

B. *Analyzing Evolutionary Information via Data Mining*

Broadly, we use a data mining technique to infer evolutionary information, *i.e.*, frequent change patterns of methods, from the commits stored in software repositories. The presented approach for mining fine-grained evolutionary couplings and prediction rules consists of three steps:

(1) Extract Commits from Software Repositories.

Modern version-control systems, such as *Subversion*, preserve the grouping of multiple changed files, *i.e.*, change-sets or commits, as submitted by a committer. These commits can be readily obtained. We perform additional processing in an attempt to group multiple commits forming a cohesive unit of a high-level change. We use a heuristic, namely author-time, to estimate such related commits. The premise is that the change-sets committed by the same committer within a time interval (*e.g.*, same day) are related and are placed in the same group or transaction [21].

(2) Process to Fine-grained Change-sets

The differences in a file of a commit can be easily obtained at a line-level granularity (*e.g.*, *diff* utility). Our approach employs a lightweight methodology for further fine-grained differencing of files in a change-set. Our tool *codediff* is used to process all the files in every change-set for source code differences at a fine-grained syntactic level (e.g., method). It uses a word-differencing tool, namely *dwdiff* (http://os.ghalkes.nl/dwdiff.html) and *srcML* representation for source code [11].

(3) Mine Evolutionary Couplings

We mine the change history of a software system for evolutionary relationships. In our approach, evolutionary couplings are essentially mined patterns of changed entities. We employ *itemset* mining [1], a data mining technique to uncover frequently occurring patterns or itemsets (co-changed entities such as methods) in a given set of transactions (change-sets/commits). The frequency is typically measured by the metric *support* or support value,

[2] http://argouml.tigris.org/issues/show_bug.cgi?id=2472

which simply measures the number of transactions in which an itemset appears. A mining tool, namely *sqminer* [21], was previously developed to uncover evolutionary couplings from the set of commits (processed at fine-granularity levels with *codediff*). These patterns are used to generate association rules that serve as IA rules for source code changes. For example, consider a method named *getType* in *ArgoUML*. The evolutionary coupling

{argouml/model/mdr/FacadeMDRImpl.java/getType, argouml/model/mdr/FacadeMDRImpl.java/isAStereotype}

is mined from the commit history between releases 0.24 and 0.26.2of *ArgoUML* and is supported by three commits with ID's 13341, 12784, and 12810. In these three commits, both *getType()* and *isAStereotype()* are found to co-change.

(4) Estimating an impact set

For any given starting/seed software entity, for impact analysis, we compute all the association rules from the mined evolutionary couplings where it occurs as an antecedent (*lhs*) and another entity as a consequent (*rhs*). Simply put, an association rule gives the conditional probability of the *rhs* also occurring when the *lhs* occurs, measured by a *confidence* value. That is, an association rules is of the form $lhs \Rightarrow rhs$. When multiple rules are found for a given entity, they are first ranked by their confidence values and then by their support values; both in a descending order (higher the value, stronger the rule). We allow a user specified cut-off point to pick the top *n* rules. Thus, the estimated impact set is the set of all consequents in the selected *n* rules. From the above evolutionary coupling example, the association rule

{argouml/model/mdr/FacadeMDRImpl.java/getType} $\Rightarrow$ *{argouml/model/mdr/FacadeMDRImpl.java/isAStereotype}*

is computed. This rule has a confidence value of 1.0 (100%) and it suggests that should the method *getType()* be changed, the method *isASteretype()* is also likely to be a part of the same change with a conditional probability of 100%.

For the bug #2472, using the seed method *org.argouml.ui.SettingsDialog.SettingsDialog* results in the methods in the gold set appearing at positions 1, 4, 5, 7, 11-17, and so on in the estimated impact set.

C. Analyzing Execution Information via Dynamic Analysis

Majority of existing impact analysis techniques rely on post-mortem execution analysis [22, 26]. The approach presented in this paper takes a different approach to applying dynamic analysis for IA. Information collected from execution traces is combined with textual and evolutionary data. Execution information is combined with other types of information by using it as a filter, as in the SITIR approach [23] where methods not executed in a feature or bug-specific scenario are clipped from the ranked list produced by IR_{CR}.

We use two different technologies to collect execution trace: Java Platform Debugger Architecture (*JPDA*[3]) and Test and Performance Tools Platform (*TPTP*[4]), which is a part of *Eclipse*. JPDA and TPTP collect the runtime information (*e.g.*, methods that were executed) about the software system without requiring any source code or byte code instrumentation. Using JPDA, we are able to collect marked traces (*i.e.*, we manually control when to start and stop collecting traces), whereas, while using TPTP, we are able to collect only full traces (*i.e.*, the trace contained all the methods from the program start until the end of the execution scenario). A significant difference between these two techniques is that JPDA exhibits noticeable overhead for large programs, making simple scenarios (*i.e.*, clicking on the menu and navigating through it) time-consuming.

D. Combining different techniques

The main goal of this work is to integrate information from orthogonal sources to attain potentially more accurate results. For change impact analysis, we have defined three information sources derived from three types of analysis: information retrieval (on textual data), data mining (on change data), and dynamic analysis (on execution data). This subsection outlines integrated approaches to provide automated support to software developers in different impact analysis scenarios (depending on the information at hand).

Information Retrieval and Dynamic Analysis. The idea of integrating IR with dynamic analysis was previously defined in the context of feature location [23]; however, it was not used for change impact analysis. A single feature- or bug-specific execution trace is first collected. IR_{CR} then ranks all the methods in the trace instead of all the methods in a software release. Therefore, the run-time information is used as a filter to eliminate methods that were not executed and are less likely to be relevant to the change request. We refer to this integrated approach as $IR_{CR}Dyn_{CR}$. The dynamic information, if and when available, can be used to eliminate some of the false positives produced by IR_{CR}. For the bug #2472, $IR_{CR}Dyn_{CR}$ results in methods in its gold set at positions 1, 3, 5, 7, 11, 12, 14, 29, and so on. Once again, the impact set gleaned via $IR_{CR}Dyn_{CR}$ is more accurate than IR_{CR}.

Information Retrieval and Data Mining. Existing change impact analysis techniques [16, 19, 29] take an initial software entity (*e.g.*, a method) in which a change is identified and estimates other software entities that are probable change candidates, referred to as an estimated impact set. Our approach ($IR_{CR}Hist_{seed}$) not only considers this initial software entity, but also takes into account the textual description of a given change request, which triggers this maintenance task. Our integrated approach computes the estimated impact set with the following steps: (1) selecting the starting point; (2) mining commits for evolutionary couplings; (3) computing change request similarities; and (4) integrating IR and evolutionary coupling results.

(1) Selecting the first relevant entity. This is the initial software entity for which IA needs to be performed. For example, this initial entity (*i.e.*, a method) could be a result of a feature location activity [23].

(2) Mining evolutionary couplings from commits. Mine a set of commits from the source code repository and compute evolutionary couplings for a given software entity. Only the commits that occurred before the software release in the step (1) are considered. Evolutionary couplings are

[3] http://java.sun.com/javase/technologies/core/toolsapis/jpda/
[4] http://www.eclipse.org/tptp/

then used to form association rules that are ranked by the support and confidence values. See details in Section B.

(3) Computing similarities using a change request. Compute conceptual couplings with IR methods from the release of a software system in which the first entity is selected. This process in discussed in depth in Section A.

(4) Integrating IR and data mining results. Like our previous work [19], the resulting impact set is acquired by combining the *N/2* highest ranked elements from each technique (steps 2 and 3). Note that *N* is the desired size of the final impact set. Therefore, each technique equally contributes to the resulting set. If the same method is suggested by both techniques, it will appear only once in the final impact set. Methods will be continuously selected, alternating the source ranked list, until an impact set of size *N* is acquired or the two sources are exhausted. For the bug #2472, $IR_{CR}Hist_{seed}$ showed improvement over IR_{CR}. In this case IR_{CR} returned a few relevant methods in the top positions and $Hist_{seed}$ returned complementary 11 relevant results in the first 18 positions. The two examples depict two different scenarios where the combination improves IA by either alleviating the shortcomings of one source or blending the orthogonal information from the two sources.

Information Retrieval, Data Mining and Dynamic Analysis. We combine all types of analyses: IR, dynamic, and data mining, to perform IA. To integrate these three techniques, we utilize the combination $IR_{CR}Dyn_{CR}$ with the standalone approach $Hist_{seed}$, which yields $IR_{CR}Dyn_{CR}Hist_{seed}$. Although the combination $IR_{CR}Dyn_{CR}$ benefits from the filtering provided by dynamic information, it is also possible that correct methods are eliminated from further consideration; an undesired effect. We augment $IR_{CR}Dyn_{CR}$ with $Hist_{seed}$, with the intent of reducing the impact of erroneously filtered methods. The techniques $IR_{CR}Dyn_{CR}$ and $Hist_{seed}$, are combined using the same heuristic presented for the combination $IR_{CR}Hist_{seed}$. Using the highest ranked *N/2* methods, we strive to leverage the best selection of methods from each technique. Similar to the improvement of $IR_{CR}Hist_{seed}$ over IR_{CR}, $IR_{CR}Dyn_{CR}Hist_{seed}$ produces more accurate impact set than $IR_{CR}Dyn_{CR}$ for the bug #2472. Other combinations are worth investigating, but they present a different focus for future work.

IV. Empirical Case Study

The research hypothesis is that these combinations help counter the precision or recall deficit of individual techniques and improve the overall accuracy. The components IR_{CR} and $Hist_{seed}$ embed automatic elements, whereas, the most developer intensive component is Dyn_{CR}. We posit the following **r**esearch **q**uestions (RQ):

RQ_1: Does the combination of IR_{CR} and Dyn_{CR} provide an improved impact set over the one with the highest automated component IR_{CR}?

RQ_2: Does the combination of IR_{CR} and $Hist_{seed}$ provide an improved impact set over the one with the highest automated component IR_{CR}?

RQ_3: Does the combination of IR_{CR}, Dyn_{CR}, and $Hist_{seed}$ provide an improved impact set over the one with the highest automated component IR_{CR}?

The above three research questions are substantiated with the statistical tests for the following *null* hypotheses:

$\mathbf{H}_{0\,P1}$: The combination of IR_{CR} and Dyn_{CR} (RQ_1) *does not significantly* improve the precision results of impact analysis compared to IR_{CR}.

$\mathbf{H}_{0\,R1}$: The combination of IR_{CR} and Dyn_{CR} (RQ_1) *does not significantly* improve the recall results of impact analysis compared to IR_{CR}.

$\mathbf{H}_{0\,P2}$: The combination of IR_{CR} and $Hist_{seed}$ (RQ_2) *does not significantly* improve the precision results of impact analysis compared to IR_{CR}.

$\mathbf{H}_{0\,R2}$: The combination of IR_{CR} and $Hist_{seed}$ (RQ_2) *does not significantly* improve tje recall results of impact analysis compared to IR_{CR}.

$\mathbf{H}_{0\,P3}$: The combination of $IR_{CR,}$ $Dyn_{CR,}$ and $Hist_{seed}$ (RQ_3) *does not significantly* improve the precision results of impact analysis compared to IR_{CR}.

$\mathbf{H}_{0\,R3}$: The combination of $IR_{CR,}$ $Dyn_{CR,}$ and $Hist_{seed}$ (RQ_3) *does not significantly* improve the recall results of impact analysis compared to IR_{CR}.

Accordingly, we also defined alternative hypotheses for the cases where the null hypotheses can be rejected with high confidence. For example:

$\mathbf{H}_{ALT\ P1}$: The combination of IR_{CR} and Dyn_{CR} (RQ_1) *significantly* improve the precision results of impact analysis compared to IR_{CR}.

$\mathbf{H}_{ALT\ R1}$: The combination of IR_{CR} and Dyn_{CR} (RQ_1) *significantly* improve the recall results of impact analysis compared to IR_{CR}.

The remaining five alternative hypotheses are defined in an analogous fashion; however, their formulation is not shown here due to space limitations.

The *Wilcoxon* signed-rank test, a non-parametric paired samples test, is applied to test for the statistical significance in the improvement obtained using the combinations of IA techniques. The results of the test determine whether the improvement obtained using a given combination over the baseline approach (*i.e.*, IR_{CR}) is statistically significant. Prior work [29] shows that a technique based on information retrieval yields better results than approaches that leverage structural information.

We describe our empirical study using the Goal-Question-Metrics paradigm [4], which includes *goals*, *quality focus,* and *context*. In the context of our case study we aim at addressing our three research questions. The *goal* of the empirical case study is to determine if it is beneficial to combine the various techniques when performing impact analysis, while the *quality focus* is on acquiring improved accuracy. The *perspective* is of a software developer addressing a change request, which demands developers to perform a thorough impact analysis of related source code entities. With regards to accuracy, it is desirable to have a technique that provides all, and only, the true impacted entities, *i.e.*, alleviates the impact of false positives and false negatives. It is important to provide the developers with the

Table I. Summary of the benchmarks: bugs (B), features (F), and patches (P) with changed methods

	#change reqs			methods in gold set: descriptive stats					
System	**B**	**F**	**P**	**min**	**25**	**med**	**75**	**max**	**Total**
jEdit	51	30	22	2	3	5	9	41	701
ArgoUML	50	8	23	2	3	5	12	72	673
muCom	55	10	0	2	3	4	11	104	691
JabRef	25	3	0	2	3	5.5	11	33	269

highest accuracy using the sources of information available (*e.g.*, static and dynamic). Our approach considers various sources of information; however, an important issue is to compare performances of different analysis combinations.

A. Accuracy Metrics

1) Precision and Recall

In order to evaluate impact analysis techniques we use *precision* (*i.e.*, an inverse measure of false positives) and *recall* (*i.e.*, an inverse measure of false negatives), two widely accepted metrics for accuracy assessment. Given an estimated impact set acquired from a technique and the actual impact set (*e.g.*, a set of entities actually modified to address a given change request), the metrics *precision* and *recall* can be computed.

For a given **i**mpact **s**et (*IS*) of entities and a set of actual or **c**orrectly changed entities **s**et (*CS)*, the precision, P_{IS}, is defined as the percentage of correctly estimated changed entities over the total estimated entities. The recall, R_{IS}, is defined as the percentage of correctly estimated changed entities over the total correctly changed entities.

$$P_{IS} = \frac{|IS \cap CS|}{|IS|} \times 100\% \qquad R_{IS} = \frac{|IS \cap CS|}{|CS|} \times 100\%$$

B. Evaluated Subject Systems

The *context* of our study is characterized by four open source *Java* systems, namely *jEdit v4.3*, a popular text editor, *ArgoUML v0.22*, a well-known UML editor, *muCommander v0.8.5*, a cross-platform file manager, and *JabRef v2.6*, a BibTeX reference manager software. The sizes of these considered systems range from 75K to 150K LOC and contain between 4K and 11K methods. The characteristics of these systems are detailed in Table II.

C. Building the benchnmarks

For each of the subject systems, we created a benchmark to evaluate the impact analysis techniques. The benchmark consists of a set of change requests that has the following information for each change request: a natural language query (change request summary) and a gold set of methods that were modified to address the change request.

The benchmark was established by a human investigation of the change requests (done by one of the authors), source code, and their historical changes recorded in version-control repositories. *Subversion* (SVN) repository commit logs were used to aid this process. For example, keywords such as *Bug Id* in the commit messages/logs were used as starting points

Table II. Characteristics of the subject systems considered in the case study.

System	Ver	LOC	Files	Methods	Terms
jEdit	4.3	103,896	503	6,413	4,372
ArgoUML	0.22	148,892	1,439	11,000	5,488
muCommander	0.8.5	76,649	1,069	8,187	4,262
JabRef	2.6	74,182	577	4,604	5,104

to examine if the commits were in fact associated with the change request in the issue tracking system that was indicated with these keywords. The files changes in those commits, which can be readily obtained from SVN, were processed to identify the methods that were changed, *i.e.*, gold set, which forms our actual impact set for evaluation.

Our technique operates at the change request level, so we also need input queries to test. These queries were constructed by concatenating the title and the description of the change requests referenced from the SVN logs.

D. Evaluation Procedure (for all systems)

Our evaluation procedure consists of the following steps:

1. *Acquire Conceptual Training Set* - Compute conceptual/textual similarities between change requests and methods on a release (*e.g.*, *ArgoUML 0.22*) of a subject system.
2. *Acquire Evolutionary Training Set* - Mine evolutionary couplings (and association rules) from a set of commits in a history period prior to the selected release in Step 1. We mined over 7,000 commits between releases 0.14 and 0.22 of *ArgoUML*, over 1,800 commits between releases 4.0 and 4.3 of *jEdit,* over 2,500 commits from the change history before the release 0.8.5 of *muCommander*, and over 2,400 commits from the change history before the release 2.6 of *JabRef*. Both the trunk and branches of the change history were considered while choosing the appropriate commits.
3. *Extract Testing Set* – Pick the gold set of methods associated with every change request in Step 1 from the benchmark described in Section C. This gold set is considered as an actual impact set, *i.e.*, the ground truth, for evaluation purposes.
4. *Acquire Dynamic Information* – Obtain execution traces related for each change request in the testing set. A profiler tool was used on the subject system to generate the execution trace for every change request. Every attempt was made to follow the *steps to reproduce* described in the change request, which are typically the steps described in natural language to reproduce the issue that was reported, so that it can be verified. For the *jEdit* system, we collected traces using JPDA, whereas for the other three systems, we used TPTP.
5. *Generate Impact Sets* - Derive impact sets for the different combinations and the baseline technique of our approach, for each commit in the testing set.
6. *Compute* Results - Compute accuracy metrics for all the estimated impact set in Step 5.
7. *Evaluate Results* - Compare the accuracy results of the combinations over the baseline in Step 6.

Table III. Precision (P) and recall (R) percentages results of IR_{CR}, combination $IR_{CR}Dyn_{CR}$, combination $IR_{CR}Hist_{seed}$, and combination $IR_{CR}Dyn_{CR}Hist_{seed}$ approaches to IA for all systems using various cut points.

Cut Points		5		10		20		30		40			5		10		20		30		40	
Precision (P) and Recall (R)		**P**	**R**	**P**	**R**	**P**	**R**	**P**	**R**	**P**	**R**		**P**	**R**	**P**	**R**	**P**	**R**	**P**	**R**	**P**	**R**
IR_{CR}	**ArgoUML**	7	4	6	6	5	12	4	14	4	18	**jEdit**	10	7	9	13	6	20	5	26	5	30
$IR_{CR}Dyn_{CR}$		11	7	8	10	6	19	6	26	5	28		17	14	14	25	10	35	8	23	7	50
$IR_{CR}Hist_{seed}$		15	14	12	19	9	25	7	28	6	33		11	11	9	22	7	34	5	43	5	47
$IR_{CR}Dyn_{CR}Hist_{seed}$		17	16	13	22	10	31	8	37	7	41		18	23	14	37	9	53	8	64	7	75
IR_{CR}	**JabRef**	9	4	11	11	8	22	7	25	5	28	**muComander**	7	9	6	13	5	19	4	20	4	24
$IR_{CR}Dyn_{CR}$		14	9	11	14	8	24	6	29	5	31		11	11	9	17	7	22	5	27	5	34
$IR_{CR}Hist_{seed}$		9	4	11	14	9	24	7	38	6	40		8	14	6	22	5	30	4	32	4	36
$IR_{CR}Dyn_{CR}Hist_{seed}$		14	15	11	21	8	33	6	45	5	48		12	19	9	25	6	34	5	37	5	46

E. Results

1) RQ1: Comparing $IR_{CR}Dyn_{CR}$ against IR_{CR}

Combining multiple analysis techniques has been shown useful for impact analysis [19]. Our first RQ focuses on a combination of IR and dynamic analysis techniques, which has not been considered for the task of impact analysis in the literature previously. We investigate the likely benefits of combining IR_{CR} and Dyn_{CR} for IA in our approach.

Table III presents the results for IR_{CR} as well as the results for the combination $IR_{CR}Dyn_{CR}$. The results indicate a positive improvement for all four systems considered. The table indicates an improvement of as much as 7% in precision and up to 20% in recall for the software systems considered. Based on these results, the combination of $IR_{CR}Dyn_{CR}$ is shown to be superior to the standalone technique IR_{CR}. Additionally, the results in Table IV for the hypotheses $H_{0\,P1}$ and $H_{0\,R1}$ indicate that the improvement is statistically significant for all the systems, with the exception of *JabRef*. These two null hypotheses were rejected based on the *p* values for all the systems, but *JabRef*.

An example of this combination improvement can be seen in the *ArgoUML* bug #2472, described in Section III D. It is evident here that the dynamic information helped eliminate the false positives that were ranked at the top by IR_{CR} and helped to bubble up the relevant methods buried at the bottom. The ranking of relevant methods is drastically improved with this combination over that of IR_{CR} (*i.e.*, a number of method were promoted to the top 10 list).

2) RQ2: Comparing $IR_{CR}Hist_{seed}$ against IR_{CR}

We explore the combination IR_{CR} and $Hist_{seed}$ and compare its performance to that of IR_{CR}. We used a 50:50 combination ratio of IR_{CR} and change history for *ArgoUML* (*i.e.*, 50% of the method in the estimated impact set were selected from IR_{CR} and the other 50% from $Hist_{seed}$) and a 75:25 combination ratio for the other systems. The choice of these ratios was driven by the system sizes and their historical information. These results also appear in Table III. Our findings reveal that this combination is quite useful in several cases. For example, when performing impact analysis on *ArgoUML*, this combination always yields an improvement in accuracy. The improvement of 8% in precision and 25% in recall, on average, is observed across all the change requests in *ArgoUML*. These results are rather promising. The results for other systems also indicate an improvement yielded by this combination for certain cut points, but there exist cases where the combination results in a decrease in accuracy. It is interesting to note the results of hypotheses $H_{0\,P2}$ and $H_{0\,R2}$ in Table IV, which show that only the improvement in recall is statistically significant across all the systems; however, note that the gain in precision acquired for *ArgoUML* is still statistically significant, which is the largest system in our evaluation.

We present examples from *ArgoUML* that show the benefits of using historical change records in conjunction with the textual information analyzed with IRCR. For example, feature #1641[5] "*Explorer option for creating diagrams from elements useful where you lost/never had a class diagram for a particular package*". This text was used as a query for IR. The issue contains two methods in the gold set. Using IR, the first method *ExplorerPopup.initMenuCreate* in the gold set is ranked at the position 12 and the second method *ExplorerPopup.ExplorerPopup*) is at the position 179. With the history information available, the method *ExplorerPopup.ExplorerPopup* is used as a seed. The other method *ExplorerPopup.initMenuCreate* in the gold set appears on the position 1, using a confidence of 1 and a support value of 2. This example shows that combining IR and history information can yield better results than using IR alone.

For the bug #2144[6], using the query "*Use Case property tab: Operations are not listed in the Use Case Property Tab furthermore, there is no possibility to create operations on use cases*", IR_{CR} ranked the first relevant method 206th. When the method *PropPanelClassifier.getAttributeScroll* was used as a seed with $Hist_{seed}$, three out of the four methods it returned were in the gold set: *PropPanelClassifier.getOperationScroll*, *ActionNewExtensionPoint.actionPerformed*, and

[5] http://argouml.tigris.org/issues/show_bug.cgi?id=1641
[6] http://argouml.tigris.org/issues/show_bug.cgi?id=2144

Table IV. Results of Wilcoxon signed-rank test ($\mu = 40$). The p values indicate that the provided improvement by combined IA approaches is not by chance.

System	$H_{0\,P1}$	$H_{0\,R1}$	$H_{0\,P2}$	$H_{0\,R2}$	$H_{0\,P3}$	$H_{0\,R3}$
ArgoUML	**< 0.001**	**< 0.001**	**< 0.001**	**< 0.001**	**< 0.001**	**< 0.001**
JabRef	0.266	0.324	0.381	**< 0.001**	0.091	**< 0.002**
jEdit	**< 0.001**	**< 0.001**	0.068	**< 0.001**	**< 0.001**	**< 0.001**
muCommander	**< 0.001**	**< 0.001**	0.425	**< 0.001**	**< 0.001**	**< 0.001**

ActionNewExtensionPoint constructor at the first, second, and fourth position respectively. This example demonstrates cases where $Hist_{seed}$ returns relevant methods in the top positions, whereas IR_{CR} returns false positives, possibly due to the lack of specific expressiveness of the query. That is, change information compensates for the deficiency of IR_{CR}.

For the feature #1942[7], which has 20 methods in the gold set, IR_{CR} produces relevant methods ranked positions 1, 3-7, 11, and so on. Furthermore, using the method *FacadeMDRImpl.getImportedElement* as a seed, $Hist_{seed}$ returns a ranked list of relevant methods with positions 1, 2, 4-7, 11, and so on. The first 10 results returned by both IR_{CR} and $Hist_{seed}$ have 12 relevant methods (11 are unique and one method appears in both lists). Combining the results of these two techniques increases the recall of the returned set of methods. This example demonstrates situations where both IR_{CR} and $Hist_{seed}$ can complement each other with a relevant set of methods can operate in tandem.

3) RQ_3: Comparing $IR_{CR}Dyn_{CR}Hist_{seed}$ against IR_{CR}

Given the promising results of combining two techniques at a time, *i.e.*, $IR_{CR}Dyn_{CR}$ and $IR_{CR}Hist_{seed}$, we also evaluated the combination of all the three techniques, *i.e.*, $IR_{CR}Dyn_{CR}Hist_{seed}$. Table III provides the results for the combination of the three types of analyses for IA. For *ArgoUML*, the results indicate that we are able to achieve precision and recall gains as high as 17% and 41%, respectively. Additionally, combining these techniques shows that after the cut point 20 $IR_{CR}Dyn_{CR}Hist_{seed}$ provides results superior to any other technique considered. Similar trends are also observed for the other three considered software systems. In the context of these results, it is clear that that combining the three types of analyses yields the best performance. Similar to the combination $IR_{CR}Hist_{seed}$, the results for the hypotheses $\mathbf{H}_{0\,P1}$ and $\mathbf{H}_{0\,R1}$ in Table IV reveal that only the improvement in recall demonstrates statistical significance (at the p values of 0.05 or smaller). Also, all the systems, but *JabRef*, yield a statistically significant improvement in precision.

We illustrate examples from *ArgoUML* where a combination of textual, historical, and execution information sources generates better results than techniques with fewer types of information. For the bug #3164[8], IR_{CR} returns the top relevant methods with ranks 36, 40, 44, and so on, whereas $IR_{CR}Dyn_{CR}$ eliminates some false positives and returns the relevant methods with positions 25, 27, 29, and so on. Historical information further improves the results. Using the method *FigState.addListenersForTransition* as a seed with $Hist_{seed}$, three relevant methods are found at positions 1, 2 and 6 in the returned top 6 methods.

For the bug #2618, $IR_{CR}Dyn_{C}$ returns the first relevant method *FigAssociation.updateEnds* ranked at the position 30; however, the historical information, when added, further improves the results. Using the method *FigAssociation.initNotationProviders* as a seed, a list of 7 methods was returned. This list contained 5 relevant methods at the positions 1-4 and 7.

For the bug #4101[9], IR_{CR} produces a ranked list where the relevant methods in the gold set appear at positions 7, 27, 61, and so on; however, $IR_{CR}Dyn_{CR}$ produces a ranked list where the gold set methods appear at positions 1, 11, 17, 27, and so on. Moreover, using the seed method/constructor *UMLComboBoxNavigator*, $Hist_{seed}$ returns three methods, which are all in the gold set.

We evaluate the strength of our scenario driven approach to impact analysis. A comparison of the baseline (IR_{CR}) to the combination $IR_{CR}Dyn_{CR}$ indicates a clear improvement, both in terms of precision and recall, when execution information is available. In the scenario where a start entity is identified, using the evolutionary coupling analysis ($Hist_{seed}$) yields higher precision but suffers in recall, as illustrated by the rapid decline in recall after the cut point three. Our results indicate that the combination $IR_{CR}Hist_{seed}$ overcomes the limitations associated with each individual technique. More specifically, the integration of the two techniques overcomes the low precision of IR_{CR} as well as the rapid decline in recall, which hinders $Hist_{seed}$. Finally, the scenario when dynamic analysis is also obtainable, $IR_{CR}Dyn_{CR}Hist_{seed}$ further demonstrates the benefit of our adaptive framework. Including the execution information considerably builds upon the improvement of $IR_{CR}Hist_{seed}$, leading to a technique that returns results superior to all other considered techniques.

V. THREATS TO VALIDITY

We identify threats to validity that could influence the results of our empirical study and limit our ability to generalize our findings. We demonstrated the benefits of different types of analysis for IA, but our empirical study is performed using four open source *Java* software systems. Although we used a diverse set of software systems in application domains, to claim generalization and external validity of our results would require further empirical

[7] http://argouml.tigris.org/issues/show_bug.cgi?id=1942
[8] http://argouml.tigris.org/issues/show_bug.cgi?id=3164
[9] http://argouml.tigris.org/issues/show_bug.cgi?id=4101

evaluation on systems implemented in other programming languages and different development paradigm.

In the empirical evaluation, we derived our testing set using commits stored in version control systems of the software systems considered in our empirical study. This strategy is similar to what researchers have previously used in MSR-based case studies [18]. Analogous to the work of others, we acknowledge the possibility that entities within a commit may not be all related. Additionally, commits may not fully encapsulate all the entities related to specific change requests. Therefore, the quality of the data stored in version control system may have influenced the results of our study. To lessen the impact of this threat, we evaluated the commits and manually included the entities in our testing set to ensure the quality of the data. The quality of the data in the version control system also impacts one of our underlying types of analysis (*i.e.,* data mining of source code changes). Inadequate historical information could potentially limit data mining techniques to accurately predict relevant methods when given an initial starting point [19].

We apply an IR technique to textual information extracted from the source code of software systems. Therefore, our findings may have been impacted by the consistency of variable naming and commenting performed by the software developers. Furthermore, we used the descriptions of change request as queries, which may have also affected the performance of our techniques.

The use of dynamic information introduces a threat related to the quality of the dynamic traces obtained for a given change request. For each entity in the testing set we manually exercised the feature described in the corresponding change request. It is possible that insufficient or inaccurate details appeared in the change request, which could have led to methods being erroneously filtered from the impact set. To address the issue we thoroughly inspected each change request to safeguard against inappropriate filtering of methods.

VI. Conclusions

The paper presents a novel approach to IA at change request level that automatically adapts to the specific software maintenance scenario at hand. Our approach uses a scenario-driven combination of IR, dynamic analysis, and MSR techniques to analyze incoming change requests, execution traces and prior changes to estimate an impact set. The empirical results on four open source systems support our premise that combining IA techniques help counter the precision or recall deficit of individual ones and improve the accuracy collectively. Our findings indicate that in certain cases an improvement of 17% in precision and 41% in recall is gained while combining, IR_{CR}, Dyn_{CR}, and $Hist_{seed}$. Moreover, the overall improvement obtained while combining these IA techniques is generally statistically significant. Approaches to impact analysis have most likely not reached the optimal levels of accuracy desired by practitioners. Nonetheless, our technique provides improved accuracy over previously published work. Our work provides a noteworthy step forward towards achieving acceptance from practitioners. Finally, the data used in producing the results in this paper is publicly available and other researchers are encouraged to reproduce or verify our results[10].

VII. Acknowledgments

We thank the anonymous reviewers for pertinent comments, which helped us to improve the quality of the paper. This work is supported in part by NSF CCF-1156401, NSF CCF-1016868, and NSF CCF-0916260 grants. Any opinions, findings and conclusions expressed herein are those of the authors and do not necessarily reflect those of the sponsors.

VIII. References

[1] Agrawal, R. and Srikant, R., "Mining Sequential Patterns", in Proc. of 11th International Conference on Data Engineering, Taipei, Taiwan, March 1995.

[2] Alali, A., Kagdi, H., and Maletic, J. I., "What's a Typical Commit? A Characterization of Open Source Software Repositories", in Proc. of 16th IEEE International Conference on Program Comprehension (ICPC'08), Amsterdam, The Netherlands, June 2008.

[3] Anvik, J., Hiew, L., and Murphy, G. C., "Who should fix this bug?" in Proc. of 28th International Conference on Software Engineering (ICSE'06), 2006, pp. 361-370.

[4] Basili, V. R., Caldiera, G., and Rombach., D. H., *The Goal Question Metric Paradigm*, John W & S, 1994.

[5] Begel, A., Phang, K. Y., and Zimmermann, T., "Codebook: Discovering and Exploiting Relationships in Software Repositories", in Proc. of 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10), 2010, pp. 125-134.

[6] Binkley, D., Davis, M., Lawrie, D., and Morrell, C., "To CamelCase or Under_score", in Proc. of 17th IEEE International Conference on Program Comprehension (ICPC'09), May 17-19 2009, pp. 158-167.

[7] Bohner, S. and Arnold, R., *Software Change Impact Analysis*, Los Alamitos, CA, IEEE CS, 1996.

[8] Briand, L., Wust, J., and Louinis, H., "Using Coupling Measurement for Impact Analysis in Object-Oriented Systems", in Proc. of IEEE ICSM'99, August 30 - September 3, 1999, pp. 475-482.

[9] Buckner, J., Buchta, J., Petrenko, M., and Rajlich, V., "JRipples: A Tool for Program Comprehension during Incremental Change", in Proc. of 13th IEEE International Workshop on Program Comprehension (IWPC'05), St. Louis, Missouri, USA, May 15-16 2005, pp. 149-152.

[10] Canfora, G. and Cerulo, L., "Fine Grained Indexing of Software Repositories to Support Impact Analysis", in Proc. of International Workshop on Mining Software Repositories (MSR'06), 2006, pp. 105 - 111.

[11] Collard, M. L., Kagdi, H. H., and Maletic, J. I., "An XML-Based Lightweight C++ Fact Extractor", in Proc. of 11th IEEE International Workshop on Program Comprehension (IWPC'03), Portland, OR, May 10-11 2003, pp. 134-143.

[10] http://www.cs.wm.edu/semeru/data/icse2012-impact-analysis

[12] Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., and Koschke, R., "A Systematic Survey of Program Comprehension through Dynamic Analysis", *IEEE Transactions on Software Engineering (TSE)*, vol. 35, no. 5, 2009, pp. 684-702.

[13] Di Lucca, G. A., Di Penta, M., and Gradara, S., "An Approach to Classify Software Maintenance Requests", in Proc. of IEEE International Conference on Software Maintenance (ICSM'02), Montréal, Québec, Canada, 2002, pp. 93-102.

[14] Dit, B., Guerrouj, L., Poshyvanyk, D., and Antoniol, G., "Can Better Identifier Splitting Techniques Help Feature Location?" in Proc. of 19th IEEE International Conference on Program Comprehension (ICPC'11), Kingston, Ontario, Canada, June 22-24 2011, pp. 11-20.

[15] Gall, H., Hajek, K., Jazayeri, M., "Detection of Logical Coupling Based on Product Release History", in Proc. of Proceedings of the International Conference on Software Maintenance (ICSM'98), March 16-19, pp. 190 - 198.

[16] Hill, E., Pollock, L., and Vijay-Shanker, K., "Exploring the Neighborhood with Dora to Expedite Software Maintenance", in Proc. of 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07), November 2007, pp. 14-23.

[17] Jeong, G., Kim, S., and Zimmermann, T., "Improving Bug Triage with Bug Tossing Graphs", in Proc. of 7th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2009), Amsterdam, The Netherlands, August 2009.

[18] Kagdi, H., Collard, M. L., and Maletic, J. I., "A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution", *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, vol. 19, no. 2, March/April 2007, pp. 77-131.

[19] Kagdi, H., Gethers, M., Poshyvanyk, D., and Collard, M., "Blending Conceptual and Evolutionary Couplings to Support Change Impact Analysis in Source Code", in Proc. of 17th IEEE Working Conference on Reverse Engineering (WCRE'10), Beverly, Massachusetts, USA, October 13-16 2010, pp. 119-128.

[20] Kagdi, H., Gethers, M., Poshyvanyk, D., and Hammad, M., "Assigning Change Requests to Software Developers", *Journal of Software Maintenance and Evolution: Research and Practice (JSME)* 2011.

[21] Kagdi, H., Maletic, J. I., and Sharif, B., "Mining Software Repositories for Traceability Links", in Proc. of 15th IEEE International Conference on Program Comprehension (ICPC'07), Banff, Canada, June 26-29 2007, pp. 145-154.

[22] Law, J. and Rothermel, G., "Whole Program Path-Based Dynamic Impact Analysis", in Proc. of 25th International Conference on Software Engineering, Portland, Oregon, May 03 - 10, 2003 2003, pp. 308-318.

[23] Liu, D., Marcus, A., Poshyvanyk, D., and Rajlich, V., "Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace", in Proc. of 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07), Atlanta, Georgia, November 5-9 2007, pp. 234-243.

[24] Maule, A., Emmerich, W., and Rosenblum, D. S., "Impact Analysis of Database Schema Changes", in Proc. of 30th IEEE/ACM Inernational Conference on Software Engineering (ICSE'08), Leipzig, Germany, 2008, pp. 451-460.

[25] Orso, A., Apiwattanapong, T., and Harrold, M. J., "Leveraging Field Data for Impact Analysis and Regression Testing", in Proc. of 9th European Software Engineering Conference and 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'03), Helsinki, Finland, September 1-5 2003, pp. 128-137.

[26] Orso, A., Apiwattanapong, T., Law, J., Rothermel, G., and Harrold, M. J., "An empirical comparison of dynamic impact analysis algorithms", in Proc. of IEEE/ACM International Conference on Software Engineering (ICSE'04), 2004, pp. 776-786.

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

[28] Poshyvanyk, D., Guéhéneuc, Y. G., Marcus, A., Antoniol, G., and Rajlich, V., "Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval", *IEEE Transactions on Software Engineering*, vol. 33, no. 6, June 2007, pp. 420-432.

[29] Poshyvanyk, D., Marcus, A., Ferenc, R., and Gyimóthy, T., "Using Information Retrieval based Coupling Measures for Impact Analysis", *Empirical Software Engineering*, vol. 14, no. 1, 2009, pp. 5-32.

[30] Ren, X., Shah, F., Tip, F., Ryder, B. G., and Chesley, O., "Chianti: a Tool for Change Impact Analysis of Java Programs", in Proc. of 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications(OOPSLA '04), Vancouver, BC, Canada, 2004, pp. 432-448.

[31] Robillard, M., "Automatic Generation of Suggestions for Program Investigation", in Proc. of Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, Lisbon, Portugal, September 2005, pp. 11 - 20

[32] Salton, G. and McGill, M., *Introduction to Modern Information Retrieval*, New York, NY, USA, McGraw-Hill, 1986.

[33] Wang, X., Zhang, L., Xie, T., Anvik, J., and Sun, J., "An Approach to Detecting Duplicate Bug Reports using Natural Language and Execution Information", in Proc. of 30th International Conference on Software Engineering (ICSE'08), Leipzig, Germany, May 10-18, pp. 461-470.

[34] Weiss, C., Premraj, R., Zimmermann, T., and Zeller , A., "How Long Will It Take to Fix This Bug?" in Proc. of 4th IEEE International Workshop on Mining Software Repositories (MSR'07), Minneapolis, MN, 2007, pp. 1-8.

[35] Zimmermann, T., Zeller, A., Weißgerber, P., and Diehl, S., "Mining Version Histories to Guide Software Changes", *IEEE Transactions on Software Engineering*, vol. 31, no. 6, 2005, pp. 429-445.