

# An Adaptive Approach to Impact Analysis from Change Requests to Source Code

Malcom Gethers<sup>1</sup>, Huzefa Kagdi<sup>2</sup>, Bogdan Dit<sup>1</sup>, Denys Poshyvanyk<sup>1</sup>

<sup>1</sup>Computer Science Department  
The College of William and Mary  
Williamsburg, VA 23185  
{mgethers, bdit, denys}@cs.wm.edu

<sup>2</sup>Department of Electrical Engineering and  
Computer Science  
Wichita State University  
Wichita, KS 67260  
kagdi@cs.wichita.edu

**Abstract** — The paper presents an adaptive approach to perform impact analysis from a given change request (*e.g.*, a bug report) to source code. Given a textual change request, a single snapshot (release) of source code, indexed using Latent Semantic Indexing, is used to estimate the impact set. Additionally, the approach configures the best-fit combination of information retrieval, dynamic analysis, and data mining of past source code commits to produce an improved impact set. The tandem operation of the three techniques sets it apart from other related solutions.

## I. INTRODUCTION

Software-change impact analysis, or simply impact analysis (IA), has been recognized as a key maintenance activity. IA aims at estimating the potentially impacted entities of a system due to a proposed change [6]. In several realistic settings, change requests are typically specified in natural language (*e.g.*, *English*). For example, bug reports submitted during the post-delivery maintenance by programmers or end users. These change requests may need to be resolved with the appropriate changes to relevant source code. It is not uncommon in large-scale software 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]. It is not an uncommon 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.

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* with or without other forms of additional developer knowledge that maybe also available 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.*, relevant source code method) and run-time information pertinent to the features in change request (*i.e.*, execution trace for a feature specific scenario).

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 repositories is an extension of the collective developer or development knowledge [4]. 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. When both forms of additional developer-information context are available, a combination of IR, dynamic information, and evolutionary couplings supersedes others.

## II. RELATED WORK

Several IA approaches ranging from classical static and dynamic analysis techniques [7, 21, 25, 26, 29, 30] to the recent unconventional approaches, such as those based on IR [15, 28] and MSR [14, 19, 34], exist in the literature.

Impact analysis is traditionally performed using static program analysis [6, 8], dynamic program analysis [21, 24, 25], or a combination of these techniques [29]. 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. Cornelissen et al. [11] provide a comprehensive summary on using dynamic analysis to support program comprehension including IA.

IR methods address tasks of extracting and analyzing textual information in software artifacts, including change impact analysis in source code [9, 19, 28]. Existing approaches to IA using IR operate at two levels of abstraction: change request [9] and source code [19, 28]. In the first case, the technique relies on mining and indexing the history of change requests (*e.g.*, bug reports) [9, 33]. While this technique has been shown to be relatively robust in certain settings, it is entirely dependent on the history of prior change requests. The work in [15] 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, 28]. Our previous work [19] was consistent with earlier usages of IR in IA [28]; however, it was limited to the source-code level starting point. Our new adaptive approach operates at the change-request level as a starting point. IR is a *baseline* technique in our adaptive solution.

The term MSR has been coined to describe a broad class of investigations into the examination of software repositories (*e.g.*, *Subversion* and *Bugzilla*). Zimmerman et al. [34] 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. We refer the interested readers to Kagdi et al. [17] literature survey, and Xie’s online bibliography and tutorial on MSR<sup>1</sup>. In addition, conceptual information has been utilized in conjunction with evolutionary data to support other tasks, such as assigning incoming bug reports to developers [3, 16], identifying duplicate bug reports [32], estimating time to fix incoming bugs [33] and classifying maintenance requests [12].

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 [9]; the advent of applied IR and MSR methods has 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, such as Java Platform Debugger Architecture (JPDA) or Test and Performance Tools Platform (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.

### III. AN INTEGRATED APPROACH TO IMPACT ANALYSIS

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<sub>CR</sub>*.

The maintenance scenario may not necessarily be as austere 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<sub>seed</sub>*, 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<sub>CR</sub>*.

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<sub>CR</sub>* and *Hist<sub>seed</sub>* is engaged. Similarly, when the dynamic information is available along with the change request, a combination of the components *IR<sub>CR</sub>* and *Dyn<sub>CR</sub>* 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.

#### A. Analyzing Textual Information via IR

Textual information in software repositories (*e.g.*, change requests in *Bugzilla*) and source code, 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 [9]. 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<sub>CR</sub>* follows five main steps [22]: (1) building a corpus, (2) natural-language processing (NLP), (3) indexing, (4) querying, and (5) estimating an impact set.

An example of such a query is the bug #2472 reported in *ArgoUML v0.22*. The query is formulated from its

<sup>1</sup> <https://sites.google.com/site/asergpr/dmse>

description “Wrong keyboard focus in Settings dialog after close & reopen [...]”. 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 induce an examination of several candidates; in some cases it may not be quite practical (*e.g.*, bug #4349).

### B. Analyzing Evolutionary Information via Data Mining

Our approach to mining fine-grained evolutionary couplings and prediction rules consists of four steps: (1) extract commits from software repositories (*e.g.*, *Subversion*), (2) process commits to fine-grained change-sets, (3) mine evolutionary couplings using itemset/association rule mining, (4) estimating an impact set.

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 [18]. 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).

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, which simply measures the number of transactions in which an itemset appears. These patterns are used to generate association rules that serve as IA rules for source code changes. 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 0.26.2 of *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.

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 rule 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. Thus, the estimated impact set is the set of all consequents in the (user) 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 *isAStereotype()* 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

The majority of existing IA techniques rely on post-mortem execution analysis [21, 25]. In our approach, 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 [22] where methods not executed in a feature or bug-specific scenario are clipped from the ranked list produced by  $IR_{CR}$ .

We used JPDA and TPTP to collect execution traces because they do not require any source code or byte code instrumentation. Using JPDA, we collected marked traces (*i.e.*, we manually controlled when to start and stop collecting traces), whereas, using TPTP, we collected full traces (*i.e.*, the trace contained all the methods from the start until the stop of the program).

### 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. Our integrated approach provides an automated support to software developers in different impact analysis scenarios.

**Information Retrieval and Dynamic 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  $IR_{CR}^{Dyn_{CR}}$ . This dynamic information 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  $IR_{CR}$ .

**IR and Data Mining.** Existing change impact analysis techniques [15, 19, 28] take an initial software entity (*e.g.*, a method) in which a change is identified and estimates other change-prone 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 approach computes the estimated impact set with the following steps:

**(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 [22].

**(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 then used to form association rules and are ranked by the support and confidence values.

**(3) Computing similarities using a change request.** Compute conceptual couplings with IR methods from the release of a system in which the first entity is selected.

**(4) Integrating IR and data mining results.** The resulting impact set, similar to our previous work [19], 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 some relevant methods in the top positions and  $Hist_{seed}$  returned the complementary other 11 relevant results in the first 18 positions. The two examples depict scenarios where the combinations improve IA by either alleviating the shortcomings of one source or blending the orthogonal information from the two sources.

**IR, 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.

#### IV. 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. We identified example cases from real systems, which suggest that our integrated approach could offer accuracy improvements for IA.

#### V. ACKNOWLEDGEMENTS

This work is supported in part by NSF CCF-1063253, NSF CCF-1016868, and NSF CCF-0916260 grants. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

#### VI. REFERENCES

- [1] R. Agrawal and R. Srikant, "Mining sequential patterns," in *ICDE'95*, 1995.
- [2] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?," in *ICSE'06*, 2006, pp. 361-370.
- [3] A. Begel, K. Y. Phang, and T. Zimmermann, "Codebook: Discovering and exploiting relationships in software repositories," in *ICSE'10*, 2010, pp. 125-134.
- [4] S. Bohner and R. Arnold, *Software change impact analysis*. Los Alamitos, CA: IEEE Computer Society, 1996.
- [5] L. Briand, J. Wust, and H. Louinis, "Using coupling measurement for impact analysis in object-oriented systems," in *ICSM'99*, 1999, pp. 475-482.
- [6] J. Buckner, J. Buchta, M. Petrenko, and V. Rajlich, "Jripples: A tool for program comprehension during incremental change," in *IWPC'05*, St. Louis, Missouri, USA, 2005, pp. 149-152.
- [7] G. Canfora and L. Cerulo, "Fine grained indexing of software repositories to support impact analysis," in *MSR'06*, 2006, pp. 105 - 111.
- [8] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *TSE*, vol. 35, pp. 684-702, 2009.
- [9] G. A. Di Lucca, M. Di Penta, and S. Gradara, "An approach to classify software maintenance requests," in *ICSM'02*, 2002, pp. 93-102.
- [10] H. Gall, Hajek, K., Jazayeri, M., "Detection of logical coupling based on product release history," in *ICSM'98*, 1998, pp. 190 - 198.
- [11] E. Hill, L. Pollock, and K. Vijay-Shanker, "Exploring the neighborhood with dora to expedite software maintenance," in *ASE'07*, 2007, pp. 14-23.
- [12] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *ESEC/FSE'09*, 2009.
- [13] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *JSME*, vol. 19, pp. 77-131, March/April 2007.
- [14] H. Kagdi, J. I. Maletic, and B. Sharif, "Mining software repositories for traceability links," in *ICPC'07*, 2007, pp. 145-154.
- [15] H. Kagdi, M. Gethers, D. Poshyanyk, and M. Collard, "Blending conceptual and evolutionary couplings to support change impact analysis in source code," in *WCRE'10*, 2010, pp. 119-128.
- [16] H. Kagdi, M. Gethers, D. Poshyanyk, and M. Hammad, "Assigning change requests to software developers," *JSME*, 2011.
- [17] J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," in *ICSE'03*, 2003, pp. 308-318.
- [18] D. Liu, A. Marcus, D. Poshyanyk, and V. Rajlich, "Feature location via information retrieval based filtering of a single scenario execution trace," in *ASE'07*, 2007, pp. 234-243.
- [19] A. Orso, T. Apiwattanapong, and M. J. Harrold, "Leveraging field data for impact analysis and regression testing," in *ESEC/FSE'03*, 2003, pp. 128-137.
- [20] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. J. Harrold, "An empirical comparison of dynamic impact analysis algorithms," in *ICSE'04*, 2004, pp. 776-786.
- [21] M. Petrenko and V. Rajlich, "Variable granularity for improving precision of impact analysis," in *ICPC'09*, 2009, pp. 10-19.
- [22] D. Poshyanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, "Using information retrieval based coupling measures for impact analysis," *EMSE*, vol. 14, pp. 5-32, 2009.
- [23] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: A tool for change impact analysis of java programs," in *OOPSLA '04*, 2004, pp. 432-448.
- [24] M. Robillard, "Automatic generation of suggestions for program investigation," in *ESEC/FSE'05*, 2005, pp. 11 - 20.
- [25] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *ICSE'08*, 2008, pp. 461-470.
- [26] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller "How long will it take to fix this bug?," in *MSR'07*, Minneapolis, MN, 2007, pp. 1-8.
- [27] T. Zimmermann, A. Zeller, P. Weißgerber, and S. Diehl, "Mining version histories to guide software changes," *TSE*, vol. 31, pp. 429-445, 2005.