SE^2 Model to Support Software Evolution

Huzefa Kagdi
Department of Computer Science
Winston-Salem State University
Winston-Salem, NC 27110
kagdihh@wssu.edu

Malcom Gethers and Denys Poshyvanyk
Computer Science Department
College of William and Mary
Williamsburg, VA 23185
{mgethers, denys}@cs.wm.edu

Abstract—**The paper proposes an integrated approach, namely SE^2, to support three core software maintenance and evolution tasks: feature location, software change impact analysis, and expert developer recommendation. The approach is centered on the combinations of the conceptual and evolutionary relationships latent in structured and unstructured software artifacts. Information Retrieval (IR) and Mining Software Repositories (MSR) based techniques are used for analyzing and deriving these relationships. All the three tasks are supported under a single, common framework by providing systematic combinations of MSR and IR analyses on single and multiple versions of a software system. This combining ability of SE^2 sets it apart from previously reported relevant solutions in the literature. The outlined empirical assessment is aimed at identifying the exclusive and synergistic improvements offered by such combinations for each of the addressed tasks. Preliminary evaluation on a number of open source systems suggests that such combinations do offer improvements over individual approaches.**

I. INTRODUCTION

Software maintenance and evolution is a particularly complex phenomenon in case of long-lived, large-scale systems [11, 14]. It is not uncommon for such systems to progress through years of development history, a number of developers, and a multitude of software artifacts including millions of lines of code. Therefore, realizing even a tad of change may not be always straightforward. Clearly, changes are the central force driving software evolution. Therefore, it is not surprising that a paramount effort has been (and should be) devoted in the software engineering community to systematically understanding, estimating, and managing changes to software artifacts. This effort includes three core change related tasks of concept or feature location (where a particular functionality is implemented in a code or a starting point of a change) - **FL**, impact analysis (which other software entities should be changed given a starting point) - **IA**, and expert developer recommendations (who are the most experienced developers to implement needed changes) – **DR**

In this paper, we propose SE^2 model for comprehensive analysis of **s**oftware **e**volution that combines the **s**emantic (or conceptual) and **e**volutionary relationships in software to directly support the core software maintenance tasks FL, IA, and DR. Conceptual information captures the extent to which domain concepts and software artifacts are related to each other. This information is derived using Information Retrieval based analysis of textual software artifacts that are not limited to a single version of software (*e.g.,* comments and identifiers in a single snapshot of source code), but also across versions (*e.g.,* change logs and bug reports in the change history). Evolutionary information is derived from analyzing relationships and relevant information observed from past changes by mining software repositories. Central to our approach are the information sources that are developer/human centric (*e.g.*, comments and identifiers, and commit practices), rather than (formal)language/artifact centric (*e.g.*, static and dynamic dependencies such as call graphs).

The core research philosophy is that the *past* and *present* of software system leads to its better *future* evolution. For example, the existing methods of FL are largely limited to a single version analysis (typically the latest release) and do not consider the past evolutionary information. For IA, both single and multiple version analysis methods have been utilized independently, but their combined use has not been previously investigated. Overall, a comprehensive change management solution under a single unified umbrella that not only helps with locating the starting point of a change, but also the extent of it, and who should handle it, is currently missing. Our proposed SE^2 solution is an attempt to address these open issues and support software maintenance under one cohesive unit.

The rest of the paper is organized as follows. In Section II we describe the principle components of SE^2. The support for maintenance tasks is discussed in Section III with evaluation plans in Section IV. Finally, we conclude in Section V.

II. SE^2 MODEL PRIMITIVES

We describe the principals underlying the SE^2 model.

A. Conceptual Coupling

A vast amount of *conceptual information* is buried in the documentary or textual elements of software artifacts, *e.g.*, comments and identifiers in source code, and commits logs and bug reports in software repositories. We can infer dependencies or couplings between software entities based on natural, real phenomenon of human driven implicit documentation of application/problem/solution domains. For example, if two methods share a similar vocabulary, a conceptual, change dependency, or *conceptual coupling*, is assumed to be present between them. Therefore, managing changes becomes a discovery process for uncovering patterns, trends, and relationships from documentary constructs and their evolution.

In SE^2, *conceptual similarity* is a primary mechanism of capturing conceptual relationships among software entities. The conceptual similarity measure is designed to capture the amount of shared conceptual information among software documents. Formally, the conceptual similarity between software entities e_k and e_j (*e.g.*, e_k and e_j are methods), is computed as the cosine between the vectors ve_k and ve_j, corresponding to e_k and e_j in the vector space constructed by an IR method (*e.g.*, Latent Semantic Indexing – LSI):

$$\text{CSE}(e_k, e_j) = \frac{ve_k^T ve_j}{|ve_k|_2 \times |ve_j|_2}$$

The value of $CSE(e_k, e_j) \in [-1, 1]$ because CSE is a cosine in the vector space model. The CSE can be used as a basis for computing conceptual similarities among different documents in the model. For instance, for source code documents, these can be attributes, methods, or classes. Computing attribute-attribute or method-method similarities, CSE is straightforward (*e.g.*, e_k and e_j are substituted by a_k and a_j in the CSE formula), while deriving method-class or class-class CSE requires additional steps. We define the conceptual similarity between the method m_k and the class c_j, CSEMC, which is an average of the conceptual similarities between the method m_k and all the methods from class c_j. Using CSEMC, we define the conceptual similarity between two classes, CSEBC, as the average of the similarity measures between all unordered pairs of methods from the class c_k and class c_j.

While source code is used as an example here, all these measures are directly applicable to other types of software artifacts (*e.g.*, requirements and bug reports). For more details and examples on computing conceptual coupling measures, please refer to [12].

B. *Evolutionary Couplings*

A research direction, rooted in the emerging area of *Mining Software Repositories*, is to analyze *multiple* versions, *i.e.*, actual past changes in software repositories such as *Subversion* and *CVS*, to identify dependencies between software entities that are found to co-change. Such dependencies are termed as *evolutionary dependencies* or *couplings* [5, 6, 17, 18]. The changes observed from the past evolution of a specific system are used as a basis to speculate the change dependencies between any given software entities. For example, if two methods are observed to co-change in a number of change-sets, an inherent change dependency between them is surmised. Therefore, managing changes become a discovery process for detecting patterns, trends, and relationships from past changes.

In SE^2, evolutionary couplings are the patterns mined from itemset and/or sequential-pattern mining of change-sets or commits in the Software Change History (SCH). Formally, an unordered evolutionary coupling is a set of source code entities that are found to be recurring in at least a given number (σ_{min}) of groups of change-sets, $ec_u = \{e_p, e_q, ..., e_o\}$ where each $e \in E$ and there exists a set of related change-sets, $S(ec) = \{c \in SCH \mid ec \subseteq c\}$ with its cardinality, $\sigma(ec) = |S(ec)| \geq \sigma_{min}$. Also, let $EC = \bigcup_{i=1}^{k} eci$ be a set of all the evolutionary couplings observed in *SCH*. If (partial) order is desired, an ordered evolutionary coupling is a sequence of sets of source code artifacts, $ec_o = s_1 \rightarrow s_2 \rightarrow ... \rightarrow s_n$ where each $s = \{e_p, e_q, ..., e_o\} \subseteq cs \in SCH$, and each s_i occurs in a change-set before the change-set s_{i+1}, and each $\in E$. We give examples of ordered evolutionary couplings mined from *KDE* repository. If the order is not desired, the entities can be coalesced into a set (instead of a list).

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*. This coupling 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. Based on this evolutionary coupling, 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%.

In addition to conceptual and evolutionary couplings, we are developing several measures to gauge developer contributions or expertise from the past evolutionary information. In our previous work [7], we presented a developer expertise factor, termed *xFactor*. It is computed using a similarity measure between two vectors representing the change contributions of a developer to a given source code entity and the total changes to that source code entity. This element of *SE^2* is not discussed at length here, as it is already reported previously [7].

III. SUPPORTING TASKS WITH SE^2

We now describe the specifics of SE^2 in supporting the core software maintenance tasks FL, IA, and DR

A. *Feature location (FL)*

A feature represents in a program some functionality that is accessible and visible to the developers. Identifying the parts of the source code that correspond to a specific functionality is a prerequisite to several maintenance tasks. This process is referred to as *feature location* [13] and it is a part of the incremental change [15]. For example, assume a developer is working on text editor software and needs to modify the *file printing* feature to ensure the files can be also printed to PDF format. The developer first needs to find the existing source code that implements *file printing* before making any further changes. If the developer is unfamiliar with this particular feature before, he might not know the exact location and may spend considerable time and effort searching for relevant source code before making any changes. In SE^2, FL is supported via the following steps:

Step 1: **Create** a corpus of the retrieved software system. To analyze conceptual information in a given release of a software system, the source code and accompanying artifacts (*e.g.*, requirements, design documentation, and bug reports) are parsed using a developer-defined granularity level (that is methods or files). A corpus is created, so that each software artifact will have a corresponding document in it. We rely on *srcML* [4] as the underlying representation for source code and textual information. *srcML* is an XML representation for C/C++/Java source code with selective AST embedded and documentary constructs preserved.

Step 2: **Index** software using IR methods**.** The corpus is indexed using advanced IR methods, such as LSI and Latent Dirichlet Allocation, for indexing software. If LSI is used for indexing, dimensionality reduction is performed to capture important conceptual information about identifiers, comments and their relationships in the source code.

Step 3: **Expand** the original query using terms from evolutionary sources.

Evolutionary information at this point can be used in at least three different ways. Firstly, the user query may be expanded with similar words from commit logs (which, in turn should improve expressiveness of user queries). Secondly, the corpus of software, which is built in Step 1, may be augmented with information from commit logs pertaining to source code entities. For instance, all the commit logs involving method *foo()* are added to the document representing method *foo()* in a corpus of a software system. The idea behind this approach is to capture design decisions and rationale encoded by developers while modifying code entities, which in turn should improve the expressiveness of IR-based FL. Thirdly, the developer can also utilize evolutionary couplings, which are obtained via MSR-based analysis, to inspect other methods, which have high evolutionary couplings with the methods in the ranked list. We will compare how effective these three integration scenarios are with respect to other potential combinations, and develop and evaluate tool prototypes.

B. *Change Impact Analysis (IA)*

A typical IA technique takes a software entity in which a change is proposed, and estimates other entities that are also potentially change candidates, referred to as an estimated impact set. Bohner and Arnold surveyed IA methodologies in 1996 [2], and a number of approaches based on improved static and dynamic analyses are proposed thereafter [1, 10, 16]. Our general approach consists of the following steps:

Step 1: **Select** the first software entity, e_s, for which IA needs to be performed. For example, this first entity could be a result of FL for a feature request. Note that IA starts with a given entity.

Step 2: Compute conceptual couplings for the release of a software system in which the first entity is selected with IR methods. Let $EI(e_s)$ be the set of entities that are conceptually related to the entity from Step 1, *i.e.*, Conceptual Coupling Between two Entities, $CCBE(e_s, e_j)$ is within a user specified value, R_i. Let $EI(e_s) = \{ e_j \mid CCBE(e_s, e_j) \leq R_i \in [0, 1]\}$.

Step 3: Mine a set of commits from the source code repository and compute evolutionary coupling metrics. Here, only the commits that occurred before the release in the above step are considered. Let $EM(e_s)$ be the set of entities that are evolutionary coupled to the entity from Step 1, *i.e.*, Evolutionary Coupling Between two Entities, $ECBE(e_s, e_j)$ is within a user specified value, R_m. Let $EM(e_s) = \{ e_k \mid ECBE(e_s, e_k) \leq R_m \in [0, 1]\}$.

Step 4: Compute the estimated impact set, $E(e_s)$, from the metrics computed in steps 3 and 4. With regards to combining conceptual and evolutionary dependencies, there are quite a few possibilities. Should the union or intersection of the two estimations be considered, *i.e.*, $EI(e_s) \cup EM(e_s)$ or $EI(e_s) \cap EM(e_s)$? This question may not be an issue, if both $EI(e_s)$ and $EM(e_s)$ predict the same estimation set. In a different situation, taking their union could result in increased recall; however, at the expense of decreased precision (if a large number of false-positive estimates). On the other hand, taking only the intersection imposes a stricter constraint that could result in increased precision; however, at the expense of decreased recall. Our initial finding shows that the conceptual and evolutionary couplings provide orthogonal information (i.e., they tend to give impact sets with only a slight overlap) [9]. Therefore, we are focusing on taking the union of the two couplings for impact sets and refer to this combination as *disjunctive* approach. Furthermore, we have devised schemes based on equal (and adaptive weights given to the contribution of the two types of couplings in the combinations. For example, in one equal combination, both the coupling types contribute half the elements in the impact set. In another adaptive combination, the contribution of evolutionary couplings is parameterized to the amount and period of the considered change history (e.g., a week from the previous release).

We illustrate the mechanics of our disjunctive approach with an equal combination scheme. In Apache httpd, commit# *888310* is a fix for the bug# *47087*[1]: "*Incorrect request body handling with Expect: 100-continue…. response prior to sending its body*." In this revision, three source code files were changed: (*/http/http_filters.c, /http/http_protocol.c, /server/protocol.c*). For this example, let us assume the developer discovers, through feature location, that fixing the problem requires modifying *http/http_filters.c*. From this point, the developer can perform impact analysis to discover other entities that also require modification. As standalone techniques neither conceptual nor evolutionary coupling were capable of establishing a 100% recall. Conceptual coupling ranked */server/protocol.c* as first in the ranked list, but ranked http/*http_protocol.c* as 91st, whereas evolutionary coupling ranked *http/http_protocol.c* second in the ranked list, but ranked */server/protocol.c* as 16th. Here, we observed that when combined, the couplings identify all methods requiring modification within an impact set within a cut point of five methods (i.e., improving both recall and precision).

[1] https://issues.apache.org/bugzilla/show_bug.cgi?id=47087

C. Developer recommendations (DR)

Our approach to recommending expert developers two steps [8]. In the first step, given a concept description, LSI is used to locate a ranked list of relevant units of source code (*e.g.*, files, classes, and methods) that implement that concept in a version (typically the release in which an issue is reported) of the software system, *i.e.*, FL. In the second step, we use *xFinder* [7] (a tool that is based on *xFactor)* to suggest a ranked list of developers to assist with a change in a given file. Our approach differs from previous approaches, as it does not need the history of past bug reports. Furthermore, our approach not only supports bug assignments, but also extends to features or any change request (concept) in general.

For example, our approach correctly recommended a ranked list of developers, [*jaham, boemann*], knowledgeable in source code files related to a bug, from its description "*splitting views duplicates the tool options docker*", *in KOffice*, an open source office productivity suite.

IV. EVALUATION PLANS AND PRELIMINARY RESULTS

We conducted our first set of evaluations for IA on hundreds of changes from open source systems *Apache httpd*, *ArgoUML*, *iBatis*, and *KOffice*. The results show that combining the two couplings provides statistically significant improvements in precision and recall values over the two couplings used individually. In some cases an improvement of 20% in recall is achieved. We included code granularity levels of files and methods, and several impact set sizes.

We have also conducted a preliminary evaluation for DR on change requests from three open source systems *ArgoUML*, *Eclipse*, and *KOffice*. The overall accuracies of the correctly recommended developers are between 47% and 96% for bug reports, and between 43% and 60% for feature requests. Furthermore, our approach outperformed two other recommendation alternatives with a substantial margin.

We are planning on rigorously validating the proposed SE^2-based techniques for FL, IA and DR using empirical techniques, such as case studies and controlled experiments. For instance, we will perform case studies to determine to what extent SE^2 can be used to support IA tasks and compare instantiated techniques with alternative approaches [3, 19]. Also, we will use the software change-history for preliminary controlled experiments. Large open-source systems, such as the *KDE*, *Apache*, *jEdit*, and *GCC,* will be used as subject systems, as we have previous experience with them in our preliminary case studies[2]. Moreover, they provide a variety of applications, domains, programming languages, development practices, and sizes.

V. CONCLUDING REMARKS

The contributions of this investigation are a step towards answering our overarching research question as to what are the exclusive and potentially synergistic benefits of integrating conceptual and evolutionary information with regards to key software maintenance tasks. While both these sources have been studied independently before, their combined use for tasks such as the ones studied here has not been scientifically investigated. The proposed evaluation will provide an empirical basis to help answer this question, and define tools and techniques based on their integration. Our preliminary evaluation does indicate that there are improvements offered by the proposed SE^2 model.

VI. 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.

REFERENCES

[1] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Efficient and precise dynamic impact analysis using execute-after sequences," in *27th ICSE'05*, St. Louis, MO, USA 2005, pp. 432 - 441.

[2] S. Bohner and R. Arnold, *Software change impact analysis*. Los Alamitos, CA: IEEE Computer Society, 1996.

[3] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb, "Software dependencies, work dependencies, and their impact on failures " *IEEE Transactions on Software Engineering (TSE),* vol. 35, pp. 864-878, 2009.

[4] M. L. Collard, H. H. Kagdi, and J. I. Maletic, "An xml-based lightweight c++ fact extractor," in *11th IWPC'03*, Portland, OR, 2003, pp. 134-143.

[5] H. Gall, Hajek, K., Jazayeri, M., "Detection of logical coupling based on product release history," in *ICSM'98*, pp. 190 - 198.

[6] H. Kagdi, "Improving change prediction with fine-grained source code mining," in *22nd ASE'07*, Atlanta, Georgia, USA, 2007, pp. 559-562.

[7] H. Kagdi, M. Hammad, and J. I. Maletic, "Who can help me with this source code change?," in *ICSM'08*, Beijing, China, 2008.

[8] H. Kagdi and D. Poshyvanyk, "Who can help me with this change request?," in *17th ICPC'09*, Vancouver, BC, Canada, 2009, pp. 273-277.

[9] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Collard, "Blending conceptual and evolutionary couplings to support change impact analysis in source code," in *17th WCRE'10*, Boston, USA, 2010, pp. 119-128.

[10] J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," in *25th ICSE,* Portland, Oregon, 2003, pp. 308-318.

[11] M. M. Lehman and L. A. Belady, *Program evolution: Processes of software change*: Academic Press Professional, Inc., 1985.

[12] D. Poshyvanyk and A. Marcus, "The conceptual coupling metrics for object-oriented systems," in *22nd ICSM'06*, Philadelphia, PA, pp. 469 - 478.

[13] D. Poshyvanyk, Y. G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *IEEE Transactions on Software Engineering,* vol. 33, pp. 420-432, June 2007.

[14] V. Rajlich and K. Bennett, "A staged model for the software lifecycle," *Computer,* vol. 33, pp. 66-71, July 2000.

[15] V. Rajlich and P. Gosavi, "Incremental change in object-oriented programming," in *IEEE Software*, 2004, pp. 2-9.

[16] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: A tool for change impact analysis of java programs," in *19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications(OOPSLA '04)*, Vancouver, BC, Canada, 2004, pp. 432-448.

[17] M. P. Robillard and B. Dagenais, "Retrieving task-related clusters from change history," in *15th Working Conference on Reverse Engineering (WCRE'08)*, 2008, pp. 17-26.

[18] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering,* vol. 31, pp. 429-445, June 2005 2005.

[19] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs " in *30th International Conference on Software Engineering (ICSE'08)*, 2008, pp. 531-540.

[2] http://www.cs.wm.edu/semeru/data/jsme09-bugs-devs/