Advances in Engineering Software 53 (2012) 1–13

Contents lists available at SciVerse ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

Combining concept lattice with call graph for impact analysis ☆

Bixin Li [a,b,*], Xiaobing Sun [a,b], Hareton Leung [c]

[a] School of Computer Science and Engineering, Southeast University, Nanjing, China
[b] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China
[c] Department of Computing, Hong Kong Polytechnic University, Hong Kong, China

ARTICLE INFO

Article history:
Received 12 May 2012
Received in revised form 12 June 2012
Accepted 1 July 2012
Available online 3 August 2012

Keywords:
Formal concept analysis
Change impact analysis
Call graph
Impact factor
Hierarchical impact results
Case study

ABSTRACT

Software change impact analysis (CIA) is a key technique to identify unexpected and potential effects caused by software changes. Given a changed entity, most of current CIA techniques compute the change effect composed of some potentially impacted entities. The generated results are often of no help to the maintainers in starting the analysis of impacted entities. In this article, we combine concept lattice with call graph together to obtain a ranked list of potentially impacted methods from the proposed changed methods and/or classes. These impacted methods are ranked based on the hierarchical feature of concept lattice, represented by an impact factor, which can then be used to prioritize these methods to be inspected. Case studies based on four real-world programs show that our approach can improve the precision of the impact result without severely decreasing its recall, when compared with results from either concept lattice or call graph used independently. In addition, the predicted impacted methods with higher impact factor values are also shown to have higher probability to be affected by the changes. Our study also shows that our approach is better than the JRipples CIA approach in removing the false-positives, but at the cost of losing more false-negatives and much more time overhead.

1. Introduction

Software change is unavoidable and contributes to the high cost of software maintenance. Lehman and Belady proposed and refined five laws that characterize the dynamics of program evolution, in which the first law is – *change is continual* [31]. Changes can be stemmed from new requirements, fixing existing faults, change requests, etc. When changes are made to software, they often produce some unexpected and potential ripple effects to other parts of the software. Software change impact analysis (*CIA*), often simply called *impact analysis*, is an approach used to identify such potential effects caused by changes made to software [11]. *CIA* starts with a set of changed elements in a software system, called the *change set*, and attempts to determine a possibly larger set of elements, called the *impact set*, which requires attention or maintenance effort due to these changes [11]. The change set is often identified using the feature location technique [40,17]. Programmers use feature location to find where in the source code the initial change needs to be made [17]. The full extent of the change is then determined by CIA, which contains a collection of techniques for determining the effects on other parts of the software due to proposed changes [33]. CIA plays an important role in software development, maintenance, and regression testing [11,12,42,56]. CIA can be used before or after a change implementation. Before making changes, we can employ CIA for program understanding, change impact prediction and cost estimation [11,12]. After changes have been implemented, CIA can be applied to trace ripple effects, select test cases, and perform change propagation [42,56,9,41].

The accuracy of CIA can be determined by its resulting impact set, which often contains some false-positives (i.e., the elements in the estimated impact set are not really impacted) and false-negatives (i.e., some of the real impacted elements are not identified in the impact set). A good CIA should provide an accurate impact set with few false-positives and false-negatives. The commonly used *CIA* techniques can be categorized into *static CIA* and *dynamic CIA* techniques. Static *CIA* techniques often analyze the dependencies of the program (or its change history repositories), construct an intermediate representation (e.g., call graph), and then conduct analysis based on these representations [52,47]. The resulting impact set often has too many false-positives [30]. Thus it is not suitable for practical use [30,33]. Whereas dynamic *CIA* techniques consider part of the inputs, and often rely on the analysis of the information collected during the execution (e.g., execution traces

☆ This work is supported partially by National Natural Science Foundation of China under Grant No. 60973149, partially by the Open Funds of State Key Laboratory of Computer Science of Chinese Academy of Sciences under Grant No. SYSKF1110, partially by Doctoral Fund of Ministry of Education of China under Grant No. 20100092110022, and partially by the Scientific Research Foundation of Graduate School of Southeast University under Grant No. YBJJ1102.

* Corresponding author.
E-mail addresses: bx.li@seu.edu.cn (B. Li), sundomore@seu.edu.cn (X. Sun), cshleung@comp.polyu.edu.hk (H. Leung).

0965-9978/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.advengsoft.2012.07.001

information, coverage information) to calculate the impact set [30,35,5]. The resulting impact set often includes some false-negatives [33].

Given a change set, most of current *CIA* techniques compute the impact set composed of a set of potentially impacted entities. This presents two challenges to maintainers. First, as all the entities in the impact set are assumed to be equally affected, software maintainers do not know which impacted entities should be inspected first. Second, the impact result may contain many false-positives that will waste the maintainers' effort in analysis. But in practice, some entities in the impact set are really affected and need corresponding modifications while some others may be false-positives, which are over-estimated by the CIA technique. Hence, a better CIA technique should compute an impact set that includes the probability of being affected associated with each potentially impacted entity and also achieves a higher precision over traditional CIA techniques, that is, the impact set has fewer false-positives.

In this article, we propose a new CIA technique to solve these challenges. Our CIA technique combines concept lattice [18] with call graph [32] to compute a ranked list of impact set for method and/or class changes. Given class or method changes, our proposed CIA consists of the following four steps:

(1) All proposed changes (method changes or class changes) are mapped to class-level changes to facilitate the class-level impact analysis.
(2) Concept lattice is used to predict class-level changes. According to the inherent hierarchical feature of concept lattice, a ranked list of potentially impacted methods are generated. Each method in the generated impact set is assigned an *impact factor* (*IF*) value, which represents its probability to be affected by these class changes.
(3) Call graph is used to predict the impact set of the method-level changes. This step removes the methods which are not dependent on the changed methods from the impact set obtained in Step 2.
(4) The intersection of the two sets obtained from Step 2 and Step 3 gives the final impact set. The methods in this final impact set are also labeled with the *IF* values.

The main contributions of this article are as follows:

- We provide a combined approach based on concept lattice and call graph that can remove more false-positives over these two standalone approaches.
- From case studies on four real-world programs, it is shown that our CIA approach improves the precision of the impact set without severely decreasing its recall, when compared to other two standalone approaches.
- The case studies also show the usefulness of the impact factor metric. Impacted methods with higher impact factor values have a lower probability of belonging to false-positives.
- Finally, the accuracy of our CIA approach is empirically compared with the *JRipples* approach, and the results show that our approach has better precision but a little worse recall and more time overhead.

The rest of the article is organized as follows: in the next section, we provide the background of formal concept analysis and call graph, and the precision and recall measure to assess the CIA technique. In Section 3, we present the analysis method of concept lattice and call graph to support *CIA*. In Section 4, we conduct some case studies to validate the effectiveness of our technique. Then, some related work of *CIA* techniques and applications of *FCA* in software maintenance are introduced in Section 5. Finally, we present our conclusion and future work in Section 6.

2. Background

We first introduce some basic knowledge about concept lattice and call graph, two common representations of a system. Then, we will discuss two measures from information retrieval [57], namely precision and recall, used to compare the accuracy of CIA techniques.

2.1. Formal concept analysis

Formal Concept Analysis (FCA) is a field of applying mathematics based on the schematization of *concept* and *conceptual hierarchy* [18]. *FCA* is used to study the relation between entities and entity properties to infer a hierarchy of concepts [18]. The basic notions of *FCA* include *formal context* and *formal concept*, which are defined as follows:

Definition 1. [*Formal Context*] *A formal context is defined as a triple* $\mathbb{K} = (\mathcal{O}, \mathcal{A}, \mathcal{R})$*, where* $\mathcal{R}$ *is a binary relation between a set of formal objects* $\mathcal{O}$ *and a set of formal attributes* $\mathcal{A}$*. Thus* $\mathcal{R} \subseteq \mathcal{O} \times \mathcal{A}$*.*

Definition 2. [Formal Concept] A formal concept is a maximal collection of formal objects sharing common formal attributes, defined as a pair (O,A) with $O \subseteq \mathcal{O},\ A \subseteq \mathcal{A},\ O = \tau(A)$ and $A = \sigma(O)$, where $\tau(A) = \{o \in \mathcal{O} | \forall a \in A : (o,a) \in \mathcal{R}\} \wedge \sigma(O) = \{a \in \mathcal{A} | \forall o \in O : (o,a) \in \mathcal{R}\}$.

$\tau(A)$ is said to be the *extent* of the concept and $\sigma(O)$ is said to be its *intent*. Relation between concepts often forms a partial order on the set of all concepts. We often use *subconcept* to construct a concept lattice [18]:

Definition 3. [Subconcept] Given two concepts (O_1,A_1) and (O_2,A_2) of a formal context, (O_1,A_1) is called the subconcept of (O_2,A_2), provided that $O_1 \subseteq O_2$ (or $A_1 \supseteq A_2$). We usually denote such relation as: $(O_1,A_1) \leqslant (O_2,A_2) \iff O_1 \subseteq O_2 \iff A_1 \supseteq A_2$

The set of all concepts of a formal context forms a partial order. Birkhoff has found that it was also a complete lattice [8], defined as follows.

Definition 4. [*Concept Lattice*] *The concept lattice* $\mathcal{L}(Co)$ *is a complete lattice.* $\mathcal{L}(Co) = \{(O,A) \in 2^{\mathcal{O}} \times 2^{\mathcal{A}} | O = \tau(A) \wedge A = \sigma(O)\}$*, where infimum and supremum of two concepts* (O_1,A_1) *and* (O_2,A_2) *are defined as:* $(O_1,A_1) \wedge (O_2,A_2) = (O_1 \cap O_2, \sigma(O_1 \cap O_2))$*, and* $(O_1,A_1) \vee (O_2,A_2) = (\tau(A_1 \cap A_2), A_1 \cap A_2)$*, respectively.*

The complete information for each concept of $\mathcal{L}(Co)$ is given by their extents and intents. However, if the concepts are all labeled with such complete information, the lattice is really hard to understand. Fortunately, there is a simple way to represent their extents and intents in a more compact form. A lattice element is labeled with $a \in \mathcal{A}$ $(o \in \mathcal{O})$, if it is the most general (specific) concept having a (o) in its intent (extent). The lattice element marked with a is [18]:

$$\mu(a) = \vee\{co \in \mathcal{L}(Co) | a \in intent(co)\} \quad (1)$$

In (1), $intent(co)$ represents the intent of the concept co. And it indicates that all concepts smaller than $\mu(a)$ have a in its intent. Similarly, the lattice element marked with o is:

Table 1
Formal context.

	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12
C1	×	×	×		×	×						
C2	×	×	×	×	×			×			×	
C3	×					×	×		×			
C4				×				×				
C5						×	×		×	×	×	
C6				×						×	×	×