Information and Software Technology 55 (2013) 1437–1449

Contents lists available at SciVerse ScienceDirect

Information and Software Technology

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

FCA–CIA: An approach of using FCA to support cross-level change impact analysis for object oriented Java programs ☆

Bixin Li [a,c], Xiaobing Sun [a,b,*], Jacky Keung [d]

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

ARTICLE INFO

Article history:
Received 21 January 2012
Received in revised form 26 December 2012
Accepted 4 February 2013
Available online 5 March 2013

Keywords:
Formal concept analysis
Change impact analysis
Lattice of class and method dependence
Impact factor

ABSTRACT

Background: *Software Change Impact Analysis* (CIA) is an essential technique in software engineering to identifying the potential influences of a change, or determining change entities to accomplish such a change. The results derived, in many cases, ambiguous for the software maintainers, introduces the problem of unclear starting point of these impacted entities.
Objective: In an attempt to address this issue, this work proposes a novel approach for cross-level CIA, producing a ranked list of potentially impacted methods derived from class-level changes. Moreover, the approach of ranking the impact results is expected to be effective for maintainers to distinguish the probability of the impacted methods to be false-positives. Such results provide an eclectic approach for CIA.
Method: The approach, *FCA–CIA*, uses formal concept analysis (*FCA*) to produce an intermediate representation of the program based on the static analysis of the source code. The representation is called Lattice of Class and Method Dependence (*LoCMD*). *FCA–CIA* takes the changed classes in the change set as a whole, and determines the reachable set from the changed classes on the *LoCMD*. Based on the hierarchical property of the *LoCMD*, the impacted methods are ranked according to the impact factor metric which corresponds to the priority of these methods to be inspected.
Result: Empirical evaluations on four real-world software projects demonstrate the effectiveness of the impact factor metric and the *FCA–CIA* technique. The result shows the predicted impacted methods with higher impact factor values are more likely to be affected by the changes. Our study also shows that the *FCA–CIA* technique generates more accurate impact set than the *JRipples* and *ICP* coupling based CIA technique.

1. Introduction

Software maintenance and change are necessary to cope with new requirements, existing faults and change requests, etc. Changes made to software will inevitably have certain unpredictable and undesirable effects on the other parts of the software. Software Change Impact Analysis (*CIA*) is a technique commonly used to identify the potential effects caused by software changes [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]. It plays an important role in software development, maintenance, and regression testing [11,12,50,64]. CIA can be used before or after a change implementation. Before making changes, we can employ CIA for program comprehension, 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 [50,64,10,49]. The commonly used CIA techniques mainly contain *static CIA* and *dynamic CIA* techniques. Static CIA techniques are often performed by analyzing the syntax and semantic, or evolutionary dependence of the program (or its change history repositories) [2,1,55,44,56,28]. The resultant impact set often has many false-positives, with many of its elements not really impacted [37,43,38]. Thus this impact set they compute is very large and difficult for practical use [11]. Whereas dynamic

☆ 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 at: School of Computer Science and Engineering, Southeast University, Nanjing, China. Tel.: +86 25 5209089; fax: +86 25 52090879.
E-mail addresses: bx.li@seu.edu.cn (B. Li), xbsun@yzu.edu.cn (X. Sun), Jacky.Keung@cityu.edu.hk (J. Keung).

0950-5849/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.infsof.2013.02.003

CIA techniques consider some specific inputs, and rely on the analysis of the information collected during program execution (e.g., execution traces information, coverage information and execution relation information) to calculate the impact set [37,5,43]. Moreover, their impact set often includes some false-negatives, that is, some of the real impacted elements are not identified [11,38]. In addition, the cost of dynamic CIA techniques is usually higher than that of static CIA because of the overhead of expensive dependency analysis during program execution [13,38].

In spite of such a collection of CIA techniques, there still exist several problems with current CIA techniques as follows:

(1) The impact set is expected to have fewer false-positives and false-negatives. To provide an eclectic approach for CIA is necessary.
(2) The impact set computed by most of current CIA techniques is composed of a set of potentially impacted entities (classes, methods or statements). However, software maintainers do not know where to start to inspect the impacted entities in the impact set.
(3) The granularity level of impact set of a CIA technique is often corresponding to that of the change set, i.e., when the change set is at a certain granularity-level (files, classes, class members), the impact set is also at the same granularity-level. Sometimes a cross-level CIA which computes fine-level impact set from coarse-level change set is needed [47,38].
(4) Most current CIA techniques compute the impact set based on computing the union of the impact sets of each changed element in the change set. It does not consider the relation among the changed elements in the change set. But in practice, there exists some relationship among these changed elements.

Faced with these problems, we propose a novel approach, *FCA–CIA*, which uses *Formal Concept Analysis* (*FCA*) to support static CIA for object oriented Java programs in this article. *FCA* is a field of applying mathematics to deal with the study of the relation between entities and entity properties to infer a hierarchy of concepts [23]. For every binary relation between entities and their properties, a lattice can be constructed to provide a remarkable insight into the structure of the original relation [23]. It can be employed to produce a decomposition of an existing program based on the static analysis of the source code [54]. It has been shown that *FCA* is an elegant and powerful code analysis technique for software maintenance in the last few years [54,58]. This article firstly presents a novel representation, called *Lattice of Class and Method Dependence* (LoCMD), a compact and effective representation for CIA. *LoCMD* organizes methods into a hierarchical order. Based on this observation, *FCA–CIA* computes a ranked list of impact set by considering the change set as a whole. In addition, *FCA–CIA* technique is particularly useful for multiple proposed changes. Another advantage of using *LoCMD* is its easy application to CIA. *FCA–CIA* can be easily performed on the *LoCMD* by determining the reachable set from the lattice nodes labeled by the changed classes. The main contributions of this article are as follows:

- Construct a representation *LoCMD* for objected oriented Java programs, which can be easily applied to CIA.
- Propose a cross-level CIA technique, *FCA–CIA*, which starts from class-level changes and produces a ranked list of potentially impacted methods. These methods are ranked according to a novel *impact factor* metric, which displays the priority for inspecting this method.
- Show the effectiveness of the impact factor metric and the *FCA–CIA* technique on four real-world case studies.

This article is organized as follows: in the next section, we propose a novel representation, *Lattice of Class and Method Dependence* (LoCMD), to support *CIA*. Section 3 gives an introduction about the *FCA–CIA* technique based on the *LoCMD*. We conduct some case studies to validate the effectiveness of the *FCA–CIA* technique in Section 4. In Section 5, some related work of *CIA* techniques and applications of *FCA* in software maintenance are presented. Finally, we conclude our CIA technique and show some future work in Section 6.

2. Concept lattice for CIA

2.1. Basics for concept lattice

Formal Concept Analysis (FCA), also called concept lattice, is a field of applying mathematics based on the schematization of *concept* and *conceptual hierarchy* [23]. The basic notions of concept lattice are those of the *formal context* and the *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, which is simply called concept, is a maximal collection of formal objects sharing common formal attributes. It is defined as a pair (O,A) with $O \subseteq \mathcal{O}$, $A \subseteq \mathcal{A}$, $O = \tau(A)$ and $A = \sigma(O)$, where $\tau(A) = \{o \in \mathcal{O} | \forall a \in A : (o,a) \in \mathcal{R}\} \wedge \sigma(O) = \{a \in \mathcal{A} | \forall o \in O : (o,a) \in \mathcal{R}\}$.

In the above definition, $\tau(A)$ is often 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 the following *subconcept* definition to construct a concept lattice [23]:

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 mark such relation as: $(O_1,A_1) \leqslant (O_2,A_2) \Leftrightarrow O_1 \subseteq O_2 \Leftrightarrow A_1 \supseteq A_2$

The set of all concepts of a formal context forms a partial order. Birkhoff has found in 1940 that it was also a complete lattice [9], defined as:

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 very hard to understand. Fortunately, there is a simple way to represent their extents and intents in a more compact and readable form. A lattice element is labeled with $a \in \mathcal{A}(o \in \mathcal{O})$, if it is the most general (special) concept having a (o) in its intent (extent). The lattice element marked with a is thus [23]:

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

In this formula, $int(co)$ 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:

$$\gamma(o) = \wedge\{co \in \mathcal{L}(Co) | o \in ext(co)\}$$

```java
import java.util.*;
public  class Vertex  {

    private String id;
    private ArrayList out_Edges;
    private ArrayList in_Edges;
    private int VertexType;
    public int getVertexType() {
        return VertexType;
    }
    public void setVertexType(int vertexType) {
        VertexType = vertexType;
    }
    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id =0;
    }
    public  ArrayList getInVertex(){
        ArrayList in=new ArrayList();
        for(int i=0;i<in_Edges.size();i++){
            in.add(i, ((Edge)in_Edges.get(i)).getFrom());
        }
        return in;

    }
    public  ArrayList getOutVertex(){
        ArrayList out=new ArrayList();
        for(int i=0;i<out_Edges.size();i++){
            out.add(i, ((Edge)out_Edges.get(i)).getTo());
        }
        return out;

    }

    public boolean hasOut(){
        return this.out_Edges.size()>0;
    }
    public boolean hasIn(){
        return this.in_Edges.size()>0;
    }
    public void addOutEdge(Vertex v,int edgeType){
        Edge edge=new Edge(this,v,edgeType);
        out_Edges.add(edge);
        v.in_Edges.add(edge);
    }
    public Edge getOutEdgeTo(Vertex v){
        Iterator iter=out_Edges.iterator();
        while(iter.hasNext()){
            Edge e=(Edge)iter.next();
            if(e.getTo()==v){
                return e;
            }
        }
        return null;

    }
   public void removeEdgeTo(Vertex v){
        Iterator iter=out_Edges.iterator();
        Edge e=new Edge();
        while(iter.hasNext()){
             e=(Edge)iter.next();
            if(e.getTo()==v){
                break;
            }
        }
        out_Edges.remove(e);
        v.in_Edges.remove(e);
    }
}
```

```java
public class Edge {
    private Vertex from;
    private Vertex to;
    private int EdgeType;
    public Edge(){
        this.from=null;
        this.to=null;
        this.EdgeType=0;
    }
    public Vertex getFrom() {
        return from;
    }
    public void setFrom(Vertex from) {
        this.from = from;
    }
    public Vertex getTo() {
        return to;
    }
    public void setTo(Vertex to) {
        this.to = to;
    }
    public int getEdgeType() {
        return EdgeType;
    }
    public void setEdgeType(int edgeType) {
        this.EdgeType = edgeType;
    }
}
```

```java
import java.util.*;
public class Graph {
    protected int vertexCount;
    protected int edgeCount;
    protected Map vertexes;
    public Graph(){
        vertexCount=0;
        edgeCount=0;
        vertexes=new HashMap();
    }
    public int sizeVertexes(){
        return vertexCount;
    }
    public int sizeEdges(){
        return edgeCount;
    }
    public boolean containsEdge(Vertex from,Vertex to){
        return getEdge(from,to)!=null;
    }
     public Edge getEdge(Vertex from,Vertex to){
       return from.getOutEdgeTo(to);
    }
    public void addEdge(Vertex from,Vertex to,int edgeType){
        from.addOutEdge(to,edgeType);
        edgeCount++;
    }
    public void addVertex(Vertex  v){
        vertexes.put(v.getId(),v);
        vertexCount++;
    }
    public void removeVertex(Vertex v){
        Vertex nextVertex;
        Iterator iter=vertexes();
        while(iter.hasNext()){
            nextVertex=(Vertex)iter.next();
            if(containsEdge(v,nextVertex)){
                v.removeEdgeTo(nextVertex);
                 edgeCount--;
            }
            if(containsEdge(nextVertex,v)){
                nextVertex.removeEdgeTo(v);
                 edgeCount--;
            }
        }
        String str=v.getId();
        vertexes.remove(str);
        vertexCount--;
    }
}
```

Fig. 1. A simple Java example program.

Here, $ext(co)$ represents the extent of the concept co. This formula indicates that all concepts greater than $\gamma(o)$ have o in its extent. With such labeling approach, suprema in the lattice denotes that certain objects have common attributes while the infima shows that some attributes fit to common formal objects. In the next section, we proposed a representation to represent the object oriented Java program.

2.2. LoCMD for CIA

Class is the basic element in object oriented programming environment, which is also one of the key factors to transform from the requirement or design domain to code domain. In addition, the granularity level of most current CIA techniques are method level [38]. Therefore, the representation of object oriented Java program is also focused on class and method level. *FCA* is instantiated as follows: formal objects are the classes while formal attributes are their member methods. The binary dependence relation between them forms a pair if some class depends on some method. We define the *dependence relation* between class and member method as follows:

Definition 5 (*Dependence between class and method*). Given that a class c and a method m in a program, class c depends on method m, if and only if, at least one of the following conditions is satisfied:

1. m belongs to c;
2. m belongs to any superclass of c;
3. c depends on another method k calling m;
4. c depends on another method k called by m.

With the instantiation of formal context, concept lattice is generated based on the bottom-up lattice constructing algorithm [23]. We call our lattice as *Lattice of Class and Method Dependence* (*LoCMD*).

A formal context can be easily represented by a relation table, in which rows are headed by classes and columns headed by methods. Fig. 1 gives a simple Java example, which describes basic data structure of the *Graph*. This example is composed of three classes and 26 methods. Then according to Definition 5, the dependence between classes and methods is extracted, as shown in Table 1. The left column shows the classes of the program, and the right column shows the methods that have dependence with the class in the left corresponding column. Such a table forms the formal context to be analyzed. By applying *FCA* technique to the formal context in Table 1, *LoCMD* of the *Graph* program is generated as shown in Fig. 2. On the *LoCMD*, nodes are associated with the formal concepts while edges correspond to the containment relation between concept intents. Each lattice node on the *LoCMD* in Fig. 2 is marked with the $\mu(a)$ (*I Set*) labeling and $\gamma(o)$ (*E Set*) labeling. From the definitions above, we can obtain the following corollary about the lattice $\mathcal{L}(Co)$:

Corollary 1. *Given LoCMD:* $\mathcal{L}(Co) = (N, E)$. *C represents a set of classes. For* $c \in C$, *M(c) represents a set of methods that class c depends on. Then, we have*

$$n \in N \Longleftrightarrow ext[n] = \{c \in C | \forall m \in int[n] : m \in M(c)\} \wedge int[n] = \{m \in M | \forall c \in ext[n] : m \in M(c)\}$$

$$(n, m) \in E \Longleftrightarrow int[n] \subset int[m] \wedge \nexists k \in N : int[n] \subset int[k] \subset int[m]$$

This corollary in fact corresponds to the formal notion of concept in Definition 2, which represents the content of each node and the relation between the nodes on the LoCMD. According to the hierarchical properties of concept lattice, we can give the following three explanations for the resulting *LoCMD*:

- The lattice presents a hierarchical classification of formal objects and formal attributes, for example, $co2 \leqslant co0$ (a.k.a., $\mu(addEdge\ (), addOutEdge\ (), removeVertex\ ()) \leqslant \mu(removeEdgeTo\ ())$) indicates that all the classes depending on the methods *addEdge (), addOutEdge (), removeVertex ()* also depend on the method *removeEdgeTo ()*.

Table 1
Formal context for the simple Java example.

Formal objects	Formal attributes
Vertex	getVertexType (), setVertexType (), getId (), setId (), getInVertex (), getFrom (), getOutVertex (), getTo (), hasOut (), hasIn (), addOutEdge (), Edge (), getOutEdgeTo (), removeEdgeTo (), addEdge (), removeVertex ()
Edge	Edge (), setFrom (), setTo (), getEdgeType (), setEdgeType (), getFrom (), getTo (), getInVertex (), getOutVertex (), getOutEdgeTo (), removeEdgeTo ()
Graph	Graph (), sizeVertexes (), sizeEdges (), containsEdge (), addEdge (), addVertex (), addOutEdge (), addVertex (), removeVertex (), removeEdgeTo ()

- The supremum of $co4$ (γ(*Vertex*)) and $co5$ (γ(*Graph*)) is labeled by the methods *addEdge (), addOutEdge (), removeVertex ()*, which means that both *Graph* and *Vertex* depend on these three methods; similarly, the infimum of some concepts labeled by some methods are marked by some class c, which indicates that class c (and all classes below $\gamma(c)$, but no other) depends on these methods.
- These methods (namely *Graph (), sizeVertexes (), sizeEdges (), containsEdge (), addEdge (), addOutEdge (), addVertex (), removeVertex (), removeEdgeTo ()*) upward reachable from γ(*Graph*) (corresponding to $co5$) are potentially used by the class *Graph*.

In this article, CIA is just performed based on these explanations.

2.3. Discussions

LoCMD is a kind of concept lattice, which describes the dependences between classes and methods for the object oriented Java programs. It is really a compact and effective representation, in which one node can express either the classes in its extent or the methods in its intent. Thus the size of the *LoCMD* is smaller than traditional dependence graph [30,44,22,55], on which each class and method are expressed with one node. In spite of the loss of an amount of information on our representation, our representation can meet the demands that CIA needs. The one thing we need to do is to provide the formal context of the software. Then a concept lattice is automatically generated. Additionally, *LoCMD* organizes methods into a hierarchical order, which can help define the probability of the methods belonging to the impact set. With these impact results, maintainers can be effective and motivated to inspect which elements are more probably affected. Therefore, *LoCMD* is suitable for its application in *CIA*.

Although *LoCMD* here is used for object oriented Java programs, it can be also extended to most of other object oriented programs, such as *C++*. In other object oriented programs, basic entities also include classes, class methods. In addition, the dependence relationship between them can be defined similarly. Therefore, our *LoCMD* to represent object oriented Java programs can be easily fitted to other object oriented programs.

3. Change impact analysis

CIA is employed to estimate the potential ripple effects of the changes made to the software. The input of the *CIA* is often identified using the feature location technique [48,19]. Programmers use feature location to find where in the source code the initial change needs to be made [19]. 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 [38]. The output of many feature location techniques is

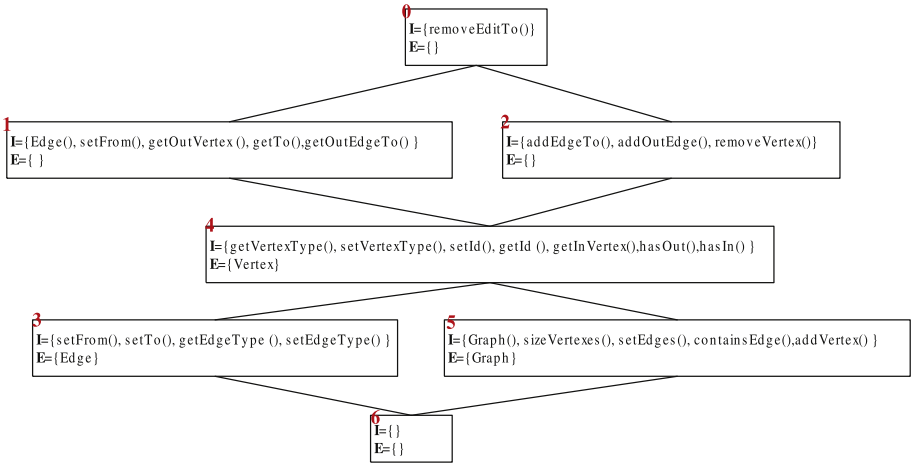

Fig. 2. LoCMD for the simple Java example program.

composed a set of classes [19], so we assume the change set to be composed of a set of classes. In this article, a cross-level CIA technique, *FCA–CIA*, is proposed with support of *FCA*, which generates method-level impact set from class-level change set. The process of using *FCA* to support *CIA* includes three steps:

Step 1: A formal context with formal object and formal attribute is provided. It is based on the analysis of the dependence between classes and methods, and such dependence is defined as Definition 5.
Step 2: Then concept lattice (*LoCMD*) is generated by applying concept lattice construction algorithm to the formal context obtained from Step 1 [23].
Step 3: Finally, CIA is conducted based on the hierarchical property of the concept lattice.

3.1. Computing impact factor based on LoCMD

The output of the CIA is the impact set, which includes the potentially impacted elements. In practice, the elements in the impact set are expected to be a ranked list, which facilitates users to focus on the elements with higher priority. Since *LoCMD* structures methods into a hierarchical order, we compute a ranked list of potentially impacted methods based on the following two assumptions:

- Upward reachable lattice nodes include methods that are shared by an increasing number of classes, hence they are expected to be less and less impacted by the change.
- Lattice nodes upward reachable from the nodes labeled by an increasing number of *changed classes* (*least common ancestors* for these *changed classes*) are more probably affected by these changed classes.

The first assumption considers the distance on the lattice, which implies that the probability of a method impacted by the changed classes decreases as we move from lattice nodes labeled by these changed classes to upper reachable nodes. And the second assumption tells us that methods upward reachable from more *joint changed classes* are more probably impacted. This assumption considers the relation among the changed classes in the change set, rather than arbitrary classes of the software. According to these two assumptions, a metric, *Impact Factor* (IF_j) of the lattice node j is defined to show the probability of its labeling methods to be impacted. The *IF* metric is defined as follow:

$$IF_j = n + \frac{1}{\sum_{i=1}^{n} min(dist(j,i)) + 1}$$

In this formula, n is the number of multiple *changed classes* upward reachable to lattice node j; $min(dist(j,i))$ is the least number of edges needing to traverse upward from lattice node i to node j. In the latter part of this formula, 1 is added to the sum of the distance to prevent the denominator from ever becoming 0. From the formula, we know that the *IF* values are always bigger than 1 and smaller (or equal to) than $n+1$, because the methods are considered to be always affected by at least one changed class. Another obvious corollary is that if a method is upward reachable from k classes, its *IF* value must be between k and $k+1$, and vice versa.

The *IF* formula is clearly accorded with these two assumptions proposed above: $\frac{1}{\sum_{i=1}^{n} min(dist(j,i))+1}$ in the formula expresses the first assumption, and n expresses the second assumption. Methods in the impact set with higher *impact factor* may be more probably impacted. Thus, maintainers should first inspect these methods to see whether they need corresponding modifications.

Consider our Java program in Fig. 1 as an example. Classes *Vertex* and *Graph* are assumed to be changed. According to our definition of the *impact factor*, we can compute the impact factor of upward reachable concepts labeling the potentially impacted methods in Table 2. Columns in Table 2 represent the priority to check the impacted methods (*Priority*), lattice nodes reachable from the concepts labeled by changed classes (*Node*), potentially impacted methods labeling these reachable concepts (*IM*), and the impact factor (*IF*), respectively. From Table 2, we see a ranked list of methods with their *IF* values. Take the concept node *co*2 as an example. *co*2 is reachable from both these two changed classes, so $n=2$. Then, we compute the least number of edges needing to traverse upward from lattice node *co*4 to node *co*2 and *co*5 to node *co*2. They both are 1. So the sum of their distance is 2. Finally, we obtain the *IF* value of node *co*2, which is 2.3. The concept node *co*2 is labeled by the methods *addEdge ()*, *addOutEdge ()*, *removeVertex ()*. And the *IF* values of other lattice nodes (methods) are computed in a similar way.

3.2. FCA–CIA process

In this article, *FCA–CIA* is conducted in a straightforward upward way on the *LoCMD*. The input of our *FCA–CIA* is composed of a set of changed classes and the output is a ranked list of impacted methods. Given the classes to be changed are $\{c_1, c_2, \ldots, c_n\}$, *FCA–CIA* is performed by means of the following way:

Table 2
Impact factor computation for the *Vertex* and *Graph* changes.

Priority	Node	IM	IF
1	*co*2	*addEdge (), addOutEdge (), removeVertex ()*	2.3
2	*co*0	*removeEdgeTo ()*	2.2
3	*co*4, *co*5	*Graph (), sizeVertexes (), sizeEdges (), containsEdge (), addVertex (), getVertexType (), setVertexType (), getId (), setId (), getInVertex (), hasOut (), hasIn ()*	2
4	*co*1	*Edge (), setFrom (), getOutVertex (), getTo (), getOutEdgeTo ()*	1.5

1. Determine the set of lattice nodes $\{co_1, co_2, \ldots, co_k\}$, which are labeled by $\{c_1, c_2, \ldots, c_n\}$ on the *LoCMD*.
2. Compute the set of lattice nodes upward reachable from $\{co_1, co_2, \ldots, co_k\}$.
3. Calculate the *impact factor* of those lattice nodes obtained in previous two steps.
4. Prioritize the set of lattice nodes according to their *impact factor* results, and provide the potentially impacted methods labeling these ordered lattice nodes to maintainers for inspection.

The *FCA–CIA* procedure described above firstly requires the location on the *LoCMD* which is directly affected by the *changed classes*. Then, methods labeling those (upward) reachable concept nodes are potentially affected by these changes. Finally, these methods are prioritized according to their *impact factor* results, and those with larger *IF* values are firstly provided to maintainers for inspection.

For our Java example, with reference to the *LoCMD* in Fig. 2, the classes *Vertex* and *Graph* are proposed to be changed. First, we identify the set of nodes labeling these two classes (*co*4 and *co*5). Then, according to Step 2, we compute the (upward) reachable concept nodes, which can be referred to Column 2 in Table 2. After this, *impact factor* values of these concept nodes are calculated, which are shown in the last column of Table 2. From these *IF* values, we prioritize these impacted methods as presented in Column 1 in this table. Finally, the ranked list of impacted methods is submitted to maintainers for inspection. The results show that methods *addEdge (), addOutEdge (), removeVertex ()* have the highest *IF* value, thus they are firstly provided to maintainers for inspection. And other methods with *IF* values from high to low are sequentially submitted for check.

3.3. Discussion

FCA–CIA is a cross-level technique, which computes a finer method level impact set according to a given coarser class level change set, and it is performed relying on two assumptions. One considers the relationship among the changed classes in the change set, and the other considers the distance between the potentially impacted method and the change set. To the best of our knowledge, current approaches are mainly focused on compute the impact set from single element in the change set, and there have been no approaches that take all the elements in the change set as a whole. Therefore, *FCA–CIA* is particularly suitable for multiple changes.

The *IF* metric is important to rank the methods in the impact set. There are two assumptions behind impact factor computation. For most object oriented programs, these two assumptions are reasonable. However, they may be inaccurate in some special environment. For example, for the system with god classes or functional designed system implemented with object oriented languages, the effectiveness of the assumptions is affected. In this case, the probability of a method with low *IF* value may be more probably affected. On the other hand, the *IF* metric is defined on the concept nodes rather than the methods, i.e., it does not rank the methods that are inside the nodes. For example, in Table 2 for node *co*4 and *co*5, it is not clear which methods will be ranked on the second, third, and etc., as they all have the same *IF* value. This is because the change set provided is only known at class level, which does not consider the finer-granularity level and the type of a change, which is future work.

4. Case study

In this section, we present some case studies to evaluate the effectiveness of the *IF* metric and the *FCA–CIA* technique. In our studies, we address the following research questions:

RQ1: Does impacted methods with higher *IF* values have higher probability to be affected?
RQ2: Does the *FCA–CIA* technique improve the accuracy of the impact set over the other CIA techniques?

4.1. Setup

4.1.1. Environment

We implemented the *FCA–CIA* technique with Java language in the *Eclipse* environment.[1] All the case studies were also conducted in the *Eclipse* environment.

4.1.2. Subject programs

As the *FCA–CIA* technique is implemented with Java language in the Eclipse environment, we select four open source Java projects for our case studies as shown in Table 3. The table shows some basic information of these projects, including: name of the project (*Name*), the evaluation period (*Period*), the number of classes (*Class*), the number of methods (*Method*), the number of lattice nodes generated for the project (*Lattice*), and lines of code (*LoC*). The number of classes, methods, lattice nodes, and lines of code are averaged across different versions for these projects. These projects are general enough to represent real-world software systems and their history of changes is available. Moreover, they have been widely used as case study in the context of CIA [25,38].

The first subject is *Rhino*,[2] which is an open source JavaScript engine and managed by the Mozilla Foundation. The second subject is *Ant*,[3] which is a Java library and command-line tool to drive processes described in build files as targets and extension points dependent upon each other. *XML – security*[4] is a component library implementing XML signature and encryption standards. The final subject is *JMeter*,[5] which is a Java desktop application designed to load test functional behavior and measure performance. We extract five consecutive versions ($v0 \rightarrow v5$) from their CVS repositories for our case studies.

[1] http://www.eclipse.org/.
[2] www.mozilla.org/rhino.
[3] http://ant.apache.org.
[4] http://santuario.apache.org/.
[5] http://jakarta.apache.org/jmeter.

Table 3
Subject programs.

Name	Period	Class	Method	Lattice	LoC
Rhino	2012-01-10 to 2012-04-23	142	1916	1274	36,223
Ant	2004-02-12 to 2005-06-02	316	2691	1566	149,191
XML – security	2010-09-15 to 2010-11-18	192	1564	2067	59,905
Jmeter	2003-02-03 to 2005-10-03	189	1237	595	73,850

Table 4
The number of bugs and the correlated changed classes of these four projects during their consecutive evolution.

System	The number of bugs and the correlated changed classes							
	$v0 \rightarrow v1$		$v1 \rightarrow v2$		$v2 \rightarrow v3$		$v4 \rightarrow v5$	
	Bug	CS	Bug	CS	Bug	CS	Bug	CS
Rhino	3	14	6	25	5	16	5	19
Ant	7	39	6	29	8	36	7	23
XML – security	4	18	3	14	6	20	3	11
Jmeter	3	16	5	22	5	19	4	15

4.1.3. Process

The process of conducting our case study includes three main stages: change set extraction, change impact analysis, and results collection. The details of these stages are as follows:

- *Change set extraction.* As the input for the CIA is the change set, we should first identify the change set. The change set of the *FCA–CIA* is composed of a set of changed classes. We obtain the change set in this way. For a given subject system, a set of bug reports is mined from the bug tracking system, such as Bugzilla.[6] The classes which had been changed to fix each bug are then mined from their software history repositories. Specific details on the process of the identification of the bug reports and changed classes can refer to [46]. During the process of collecting a set of bug reports and sets of changed classes respectively, we only include the bugs which contained more than three modified classes. As a result, there are 80 bugs for these four subject programs during their five consecutive evolution. And for these bugs, the overall size of the changed classes are 336. Details of the bugs (*Bug*) and the correlated changed classes (*CS*) are shown in Table 4.
- *Change impact analysis.* Having the change set, we apply the *FCA–CIA* technique to compute the impact set. In addition, we use another two comparative *CIA* techniques (*JRipples* [44], *ICP* coupling based CIA [15]) for comparison. For *JRipples*, it can tackle different granularity-level (class, method and block) changes and is implemented as a plug-in[7] for the Eclipse platform which can be easily run for the case study, it is selected for our comparative study. This technique is performed based on class and member dependency graph (*CMDG*). The components of *CMDG* including classes and their members of an arbitrary nesting level, are at the disposal of the users. Thus *JRipples* allows the user to interact with the iterative *CIA* process to switch from one granularity level to another. In our study, this tool is used to generate the method-level impact set from the class-level change set. The methods in the impact set produced by *JRipples* are ranked by their corresponding levels. Hence, the output of these two CIA techniques are method-level impact set, in which methods are ranked with their respective parameters. For coupling based CIA [15], we used the *ICP* coupling (a.k.a., information-flow-based coupling) to collect the impact set coupled with the changed classes, as this coupling has higher precision and recall values over the other coupling-based CIA techniques [46]. *ICP* represents the number of method invocations in a class *C*, of methods in class *D*, weighted by the number of parameters of the invoked methods. It can only compute the coupled classes in the impact set. So for comparison, we need to make some changes. After *FCA–CIA* computes the method-level impact set, it also needs to compute the enclosing classes of the methods in the impact set, which is easily done. These impacted classes are also ranked by the *IF* values of their enclosing methods. If the enclosing methods of a class have different *IF* values, we compute their average value for comparison. Hence, for this comparative study, the output of these two CIA techniques are class-level impact set, in which classes are ranked with their respective measures.
- *Results collection.* The output of these three CIA techniques is a ranked list of impacted methods or classes. To facilitate comparison, we employ a specific cut point criterion which ranges from 10% to 40% of potentially impacted methods. For each change set, we assess the accuracy of their impact sets via rankings of specific cut point.

4.1.4. Measures

Accuracy is a critical factor to evaluate the *CIA* technique. To evaluate the accuracy of a technique, two widely used metrics are *precision* and *recall* [65]. The combination of these two measures are used to assess the accuracy of an impact analysis technique. *Precision* is an inverse measure of false-positives while *recall* is an inverse measure of false-negatives. These two evaluative metrics are defined as follows:

$$P = \frac{|Actual\ \ Set\ \cap\ Estimated\ \ Set|}{|Estimated\ \ Set|} \times 100\%$$

$$R = \frac{|Actual\ \ Set\ \cap\ Estimated\ \ Set|}{|Actual\ \ Set|} \times 100\%$$

Here, *Actual Set* is the set of methods/classes which are actually changed during bug fixing. *Estimated Set* is the impact set predicted by the CIA techniques.

4.2. Results and analysis

In this section, we gather and analyze the results collected from the case studies to answer *RQ1* and *RQ2*.

[6] http://bugzilla.mozilla.org/.
[7] http://jripples.sourceforge.net/.

4.2.1. RQ1

In this section, we validate the first research question. As higher precision corresponds to higher probability of the impact set to be really affected, we see the precision of the impact set of different *IF* values. Fig. 3 shows the precision values of different *IF* values of four programs. From the variation tendency, we see that the bigger the *IF* value is, the higher the precision it corresponds to. When $4 < IF < 5$, most of the precision values for these four programs are above 30%. When $3 < IF < 4$, most of the precision values decrease to 20–30%. With continuous decrease of the *IF* value, the precision values also decrease. When $1 < IF < 2$, the precision values decrease to 10% below. In addition, we also show the precision of finer-level selection of *IF* values as in Fig. 4. The results also show the similar phenomenon, i.e., impact set with higher *IF* value has higher probability to be affected. Hence, we observe that prioritizing the impacted methods according to the *IF* values is helpful. When the maintainer traces the change effects or implements modification tasks, he can preferentially inspect those methods with high *IF* values first.

In addition, to compare values of precision for the impact sets of different *IF* values and conclude whether or not the difference is statistically significant, the Kruskal–Wallis statistical test is executed [52]. It is a nonparametric statistical test used to compare three or more groups of sample data. We used the *SPSS*[8] software to execute Kruskal–Wallis test for precision values of the different *IF*-level impact sets. For more details on the Kruskal–Wallis test, readers can refer to the work of Siegel and Castellan [52]. The statistical results are shown in Table 5. The results show that at the level of significance for alpha = 0.05, the decision is to reject the null hypothesis of absence of differences between the different *IF*-level impact sets. That is to say, differences between precision values of different *IF* values are statistically significant. We can conclude that the *IF* metric is effective in distinguishing the probability of the methods belonging to the true-positives. In other words, methods in the impact set with higher *IF* values are more likely to be really affected.

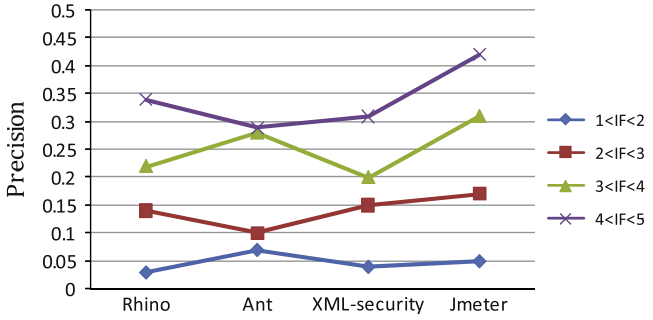

Fig. 3. Precision of the impact set in different *IF* ranges.

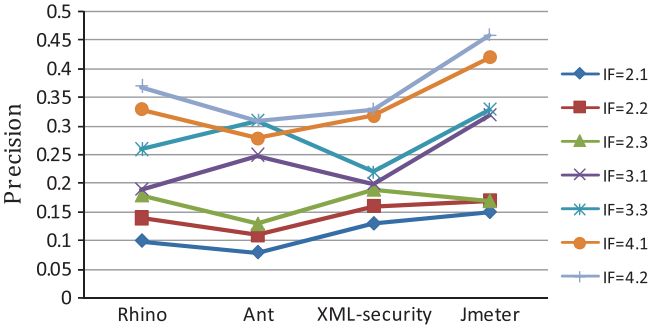

Fig. 4. Precision of the impact set of different *IF* values.

Table 5
The Kruskal–Wallis test results for the precision of different *IF* values.

Statistical results	Precision
DF (degrees of freedom)	6
Chi-square	23.198
Asymp. sig.	0.001
Alpha	0.05

4.2.2. RQ2

The *FCA–CIA* technique is a novel cross-level technique, which computes the method-level impact set from class-level change set. At present, there is few technique focusing on such kind of impact analysis. And we compare our techniques with the other CIA-techniques with some adjustment. First, we make some changes on *JRipples* to generate method-level impact set from class-level changes for comparison. Then, we modify our own CIA technique to generate the class-level impact set from class-level changes for comparison with *ICP* coupling based CIA. Here we resort to these two techniques for comparison to see whether the *FCA–CIA* technique improves the accuracy. In the following, we discuss their comparison results, respectively.

As both *FCA–CIA* and *JRipples* can generate a ranked list of impact set, in our case study, different cut points ranging from 10% to 40% of potentially impacted methods are selected. Then, we compute precision and recall values for each cut point for each CIA technique. The accuracy (precision and recall) results of using these two CIA techniques to rank potentially impacted methods during impact analysis for these four subject programs are presented in Fig. 5. First, consider the precision of the impact sets of these two approaches. For all four subject programs, we see that the *FCA–CIA* technique has better precision results over the *JRipples* technique. For example, in case of the cut point of 10% of the *Rhino* system, the average precision of the *FCA–CIA* technique is 34%, and for *JRipples* CIA technique, the precision is 22%. Increasing the cut point to 20% (or above) of the impact set, the precision of both CIA techniques decrease. But, whatever the cut point is, the precision values of the *FCA–CIA* technique are always better than the *JRipples* technique. Then, we see the recall results of these two CIA techniques in Fig. 5. The results show that sometimes the recall of the *FCA–CIA* is better than the *JRipples* technique, for example, in case of the cut point of 20% (30%) of the *Ant* system. Yet, sometimes the *JRipples* technique has better recall, for example, in case of the cut point of 10% (20%) of the *Ant* system. However, the gap of the recall values between these two techniques is not big. Table 6 presents a quantitative comparison of the accuracy of these two CIA techniques, i.e., the absolute and relative gain of the accuracy results of the *FCA–CIA* and *JRipples* approach using various cut points. The results show that the gap of the precision between these two CIA techniques is bigger than the gap of their recall values. For example, for the cut point of 10% of the *Ant* system, the absolute precision and relative precision of the *FCA–CIA* over the *JRipples* are 6% and 33%, while their absolute recall and relative recall values are only −2% and −7%. And for other cut points, a similar phenomenon can be found. In order to compare values of precision and recall for these two CIA techniques and conclude whether or not the difference is statistically significant, we executed the Mann–Whitney U test [18], which is a nonparametric approach to assess whether a difference exists between two independent observations. We still used the *SPSS* software to execute Mann–Whitney U test for precision and recall values of different cut points of

[8] http://www-01.ibm.com/software/analytics/spss/.

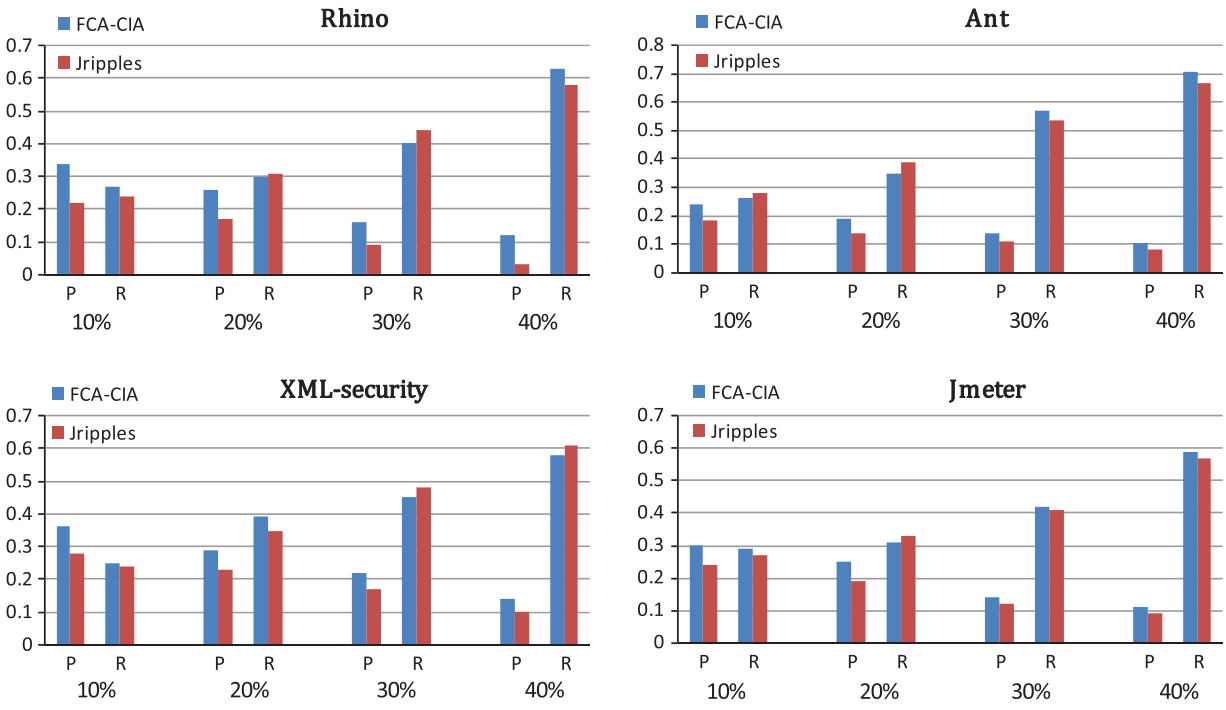

Fig. 5. The accuracy comparison results of the FCA–CIA and the JRipples approach.

Table 6
The absolute and relative gain of the accuracy results of the *FCA–CIA* and *JRipples* approach using various cut points.

System	Gain	Cut point							
		10%		20%		30%		40%	
		P (%)	*R* (%)	*P* (%)	*R* (%)	*P* (%)	*R* (%)	*P* (%)	*R* (%)
Rhino	Absolute	12	3	9	−1	7	−4	9	5
	Relative	55	13	53	−3	78	−9	300	9
Ant	Absolute	6	−2	5	−4	3	3	2	4
	Relative	33	−7	36	−10	27	26	25	6
XML – security	Absolute	8	1	6	4	5	−3	4	−3
	Relative	29	4	26	11	29	−6	40	−5
Jmeter	Absolute	6	2	6	−2	2	1	2	2
	Relative	25	7	32	−6	17	2	22	4

methods for these four subjects. The statistical results are shown in Table 7. First, we see the statistical results of the precision for different cut points. The results show that at the level of significance for *alpha* = 0.05, the decision is to reject the null hypothesis of absence of differences between these two CIA techniques. That is to say, differences of the precision between these two CIA techniques are statistically significant. Second, we see the statistical results of the recall for different cut points. The results show that the null hypothesis of absence of differences between these two CIA techniques is accepted. That is to say, difference between the recall of these two CIA techniques is not statistically significant. Therefore, from the analysis above, our CIA technique is better than the *JRipples* approach in terms of precision, but no obvious difference can be found between their recall.

Then, we see the comparison between *FCA–CIA* and *ICP* coupling based CIA. They both generate the class-level impact sets, in which classes are ranked with their respective measures. So we also select the impact sets with different cut points ranging from 10% to 40%. Then, we compute precision and recall values for each cut point for each CIA technique. The results are presented in Fig. 6. From the results, we see that the *FCA–CIA* technique has both higher precision and recall values than that of the *ICP* coupling based CIA technique. For example, in case of the cut point of 10% of the *Rhino* system, the average precision and recall of the *FCA–CIA* technique are 38% and 22%, respectively, and for *ICP* coupling based CIA technique, the precision and recall are 15%. Increasing the cut point to 20% (or above) of the impact set, the precision of both CIA techniques decrease, and their recall values increase. But, whatever the cut point is, both the precision and recall values of the *FCA–CIA* technique are always better than the *ICP* coupling based CIA technique. Table 8 presents a quantitative comparison of the accuracy of these two CIA techniques. The results also show a similar phenomenon that the *FCA–CIA* is better in terms of the precision and recall. And sometimes the gap is big, and sometimes the gap is low. We also use the Mann–Whitney U test for precision and recall values of different cut points of impacted classes to show whether significant differences exist between these two CIA techniques. The statistical results show that at the level of significance for *alpha* = 0.05, the decision is to reject the null hypothesis of absence of differences between these two CIA techniques for both precision and recall values. That is to say, differences of the precision and recall between these two CIA techniques are statistically significant.

Table 7
The Mann–Whitney U test results for the precision and recall of different cut points of methods.

System	Asymp. sig.	Cut point			
		10%	20%	30%	40%
Rhino	Precision	0.011	0.023	0.018	<0.001
	Recall	0.604	0.711	0.659	0.782
Ant	Precision	0.026	0.016	0.039	0.047
	Recall	0.576	0.601	0.214	0.593
XML – security	Precision	0.032	0.024	0.043	0.019
	Recall	0.717	0.159	0.612	0.359
Jmeter	Precision	0.021	<0.001	0.044	0.011
	Recall	0.607	0.728	0.811	0.703

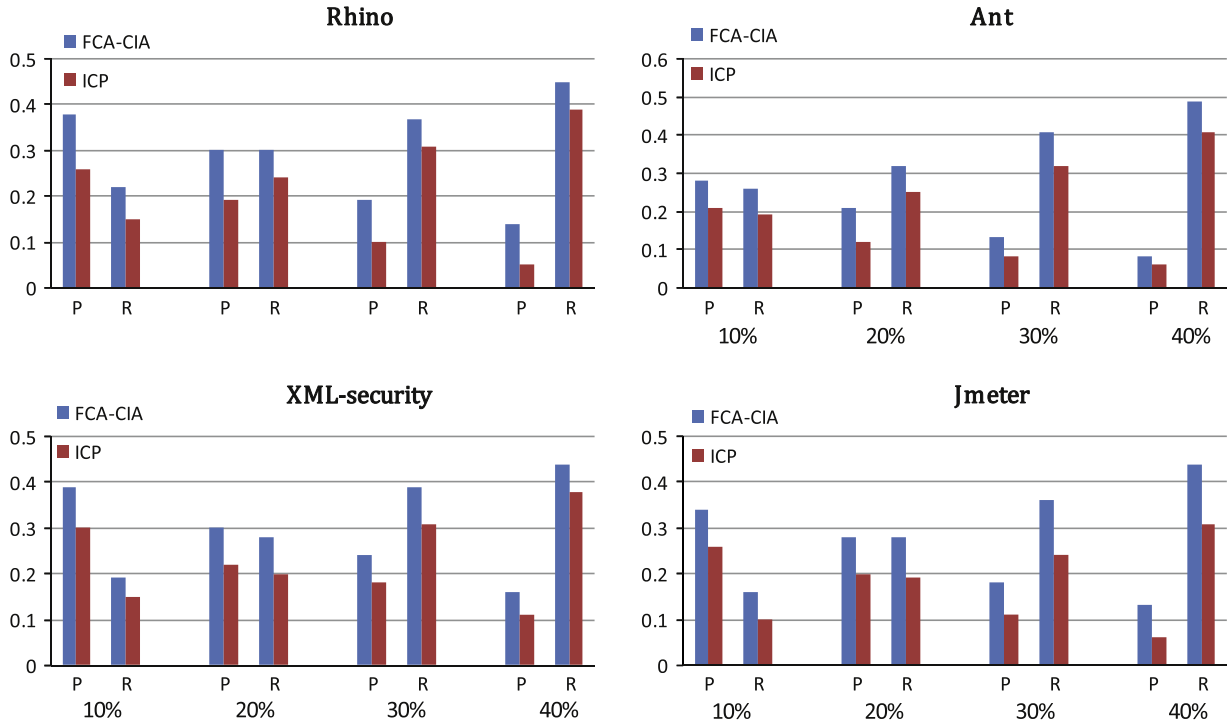

Fig. 6. The accuracy comparison results of the FCA–CIA and the ICP coupling approach.

Therefore, from the analysis above, our CIA technique is better than the *ICP* coupling based CIA technique in terms of precision and recall.

4.3. Threats to validity

Like any empirical validation, ours has its limitations. In the following, some threats to the validity of our case study are discussed.

First, we only apply our technique to four subject programs. Thus we cannot guarantee that the results in our case studies can be generalized to other more complex or arbitrary subjects. However, these subjects are selected from open source projects and widely employed for experimental studies [38]. In addition, the real impact set is obtained by selecting the differences (method/class changes) between consecutive versions. These differences may not be the actual impacts in real programs version, this may affect the evaluating measures of the impact results.

A second concern is the comparative study. *JRipples* was designed to be used manually and systematically by developers, and not in an automatic way to recommend set of methods. Here, we just used it to produce a set of methods from the changed classes. This may affect its merits. In addition, we only select the *ICP* coupling based CIA technique for comparison with our CIA technique for four projects. Some other coupling based CIA techniques, for example, *CBO*, *PIM*, and *MPC* [15], may generate different results. Here selection of this coupling is based on the empirical results from the work of Poshyvanyk et al. [46]. In the future, we hope to find a more justified comparative CIA technique for comparison.

5. Related work

In this section, we introduce some related work from two aspects: (1) change impact analysis and (2) applications of *FCA* in software maintenance.

5.1. Change impact analysis

Current researches in CIA have varied from relying on static information [2,1,56,28] to dynamic information [43,37,5] in working out the impact set. Some also utilized both static and dynamic information in combination [49,67,24]. *FCA–CIA* mainly focuses on static analysis of the program. Static CIA techniques take all

Table 8
The absolute and relative gain of the accuracy results of the *FCA–CIA* and *ICP* coupling approach using various cut points.

System	Gain	Cut point							
		10%		20%		30%		40%	
		P (%)	*R* (%)	*P* (%)	*R* (%)	*P* (%)	*R* (%)	*P* (%)	*R* (%)
Rhino	Absolute	12	7	11	6	9	6	9	6
	Relative	46	47	58	25	90	19	180	15
Ant	Absolute	7	7	9	7	5	9	2	8
	Relative	33	37	75	28	63	28	33	20
XML – security	Absolute	9	4	8	8	6	8	5	6
	Relative	30	27	36	40	33	26	45	16
Jmeter	Absolute	8	6	8	9	7	12	7	13
	Relative	31	60	40	47	64	50	117	42

possible behaviors and inputs into account, and they are often performed by analyzing the syntax and semantic dependence of the program [11,38]. The static analysis includes textual analysis, historical analysis and structural static analysis.

Textual analysis is an approach which extracts the conceptual dependence (conceptual coupling) based on analysis of the non-source code (comments) [46,25]. These coupling measures provide a new perspective to traditional structural coupling measures [14]. Conceptual coupling is based on measuring the degree to which the identifiers and comments from different classes relate to each other. The coupling measures can help rank classes in software systems based on different types of dependencies among classes [46,25]. Those classes strongly coupled with changed class are thought as the most probable impacted classes which need inspection. The granularity level they analyzed is at the class level, while ours is at cross level.

Historical analysis is performed by mining the information from multiple evolutionary versions in software historical repositories [68,16,51,31,33,42,24]. With this technique, some *evolutionary* dependencies between program entities that cannot be distilled by traditional program analysis technique can be mined from these repositories. Evolutionary dependencies suggest that for those entities that are (historically) changed together in software repositories, i.e., co-changes, they may need to change when one (some) of the entities are changed during future software evolution. CIA is then supported based on these co-change dependencies. These *CIA* techniques need multiple versions of the subject programs from the software historical repository, and usually generate the impact set at a coarse class (file) level while our technique needs only the current version of program, and produces the impact set at the method level.

Structural static dependencies between program entities are very crucial to CIA, i.e., if a program entity changes, other dependent entities might also have to change [44,56,57,55,44]. Structural static analysis usually analyzes the structural dependencies to build an intermediate representation for the program, and then computes the impact set based on transitive closure on the representation. Our CIA technique also performs structural static analysis, and obtains a more precise impact set by considering the interferences among the proposed changes.

CIA techniques discussed above all belong to static analysis. Due to the imprecision of the impact set computed by these techniques [37,43], some researchers proposed dynamic CIA techniques which focused on a set of specific program executions. Dynamic CIA techniques rely on analysis of the information collected during the execution to calculate the impact set [43,37,5,8]. Their impact sets are more precise. However, the cost of dynamic CIA techniques is higher for its overhead in collecting the execution information. Moreover, the impact set they compute often includes some false-negatives.

5.2. FCA applied in software maintenance

FCA is used to deal with the binary relation between entities and entity properties to infer a hierarchy of concepts [23]. Such a binary relationship usually occurs in software field, which attracts a batch of researchers to focus on its applications in software engineering field [61,29,58], especially in software maintenance, such as software comprehension, change impact analysis, refactoring, debugging and testing.

An amount of work of *FCA* applied in software maintenance is used to support program comprehension. Many of the work using *FCA* to support program comprehension focuses on feature location [20,36,45,27], class hierarchies identification [54,59], modularity assessment [53,6], reengineering pattern recognition (e.g., design patterns [62,40], reuse interfaces [66], co-change patterns [26]), dynamic analysis [7,26], etc.

As is presented in this article, some work uses concept analysis for impact analysis. Tonella provided a novel representation, called the concept lattice of decomposition slices by relating program variables to the statements, and applied concept analysis in such context to support *CIA* [60]. This article also employs concept lattice to perform impact analysis. The difference is that Tonella's CIA better fits the intraprocedural level and the granularity level is at statement level. Here the *FCA–CIA* technique is a cross-level technique. The input is changed classes and output is a ranked list of potential impacted methods. Our CIA technique better fits to object oriented programs.

In addition, some work utilizes formal concept analysis to extract the candidate bad-smell source code for refactoring. The refactoring techniques of applying concept lattice mainly include class hierarchies refactoring [54,39,41,32], module refactoring [59,3,35], and low-level to high-level programming refactoring [63,21].

Also, some researchers apply formal concept analysis to debugging [4,17] and testing [34]. Ammons et al. provided a novel approach for debugging temporal specifications with concept analysis by grouping the traces into highly similar clusters, which can save much effort for the users [4]. In addition, Khor and Grogono combined the genetic algorithm with concept lattice to automatically generate branch coverage test data [34].

6. Conclusion and future work

This article proposed a novel cross-level *FCA–CIA* technique based on FCA. *FCA–CIA* computes a ranked list of potential impacted methods from class-level changes. And methods in the impact set are ranked by an *IF* metric, which shows the probability of the methods to be affected. Finally, case studies on four open programs well show the effectiveness of the *IF* metric and an improvement of accuracy over that of the *JRipples* and *ICP* coupling based CIA techniques.

Though we have shown the effectiveness of our technique through real case studies, it cannot indicate its generality for other real environment. And we will conduct experiments on other arbitrary programs to evaluate the generality of our technique. In addition, we would like to research on how to further refine the *IF* metric when changes are known at method level. Finally, the impacted methods with their *impact factor* values in the impact set are objective and quantitative result, and we are planning on investigating how to apply the results to software maintenance activities concretely, such as change cost estimation and regression testing.

Acknowledgments

The authors thank in particular Paolo Tonella, Vaclav Rajlich, Sai Zhang, Radu Vanciu for their constructive comments and advice on the early version of this article. We would also like to thank Qiandong Zhang for his cooperation in developing the tool and conducting the experimental study. Special thanks to the anonymous reviewers who provided useful suggestions to make the article clearer and stronger.

References

[1] M.K. Abdi, H. Lounis, H.A. Sahraoui, Predicting change impact in object-oriented applications with bayesian networks, in: Proceedings of the IEEE Conference on Computer Software and Applications, 2009, pp. 234–239.
[2] M.K. Abdi, H. Lounis, H.A. Sahraoui, A probabilistic approach for change impact prediction in object-oriented systems, in: Proceedings of the Workshops of the 5th Conference on Artificial Intelligence Applications and Innovations, 2009, pp. 89–200.
[3] R. Al-Ekram, K. Kontogiannis, Source code modularization using lattice of concept slices, in: Proceedings of the European Conference on Software Maintenance and Reengineering, 2004, pp. 195–203.
[4] G. Ammons, D. Mandelin, R. Bodik, J. Larus, Debugging temporal specifications with concept analysis, in: Proceedings of the Conference on Programming Language Design and Implementation, 2003, pp. 182–195.
[5] T. Apiwattanapong, A. Orso, M.J. Harrold, Efficient and precise dynamic impact analysis using execute after sequences, in: Proceedings of the International Conference on Software Engineering, 2005, pp. 432–441.
[6] G. Arvalo, S. Ducasse, S. Gordillo, O. Nierstrasz, Generating a catalog of unanticipated schemas in class hierarchies using formal concept analysis, Information and Software Technology 52 (11) (2010) 1167–1187.
[7] T. Ball, The concept of dynamic analysis, in: Proceedings of ACM SIGSOFT Symposium on the Foundations of Software Engineering, 1999, pp. 216–234.
[8] A. Beszedes, T. Gergely, S. Farago, T. Gyimothy, F. Fischer, The dynamic function coupling metric and its use in software evolution, in: Proceedings of the 11th European Conference on Software Maintenance and Reengineering, 2007, pp. 103–112.
[9] G. Birkhoff, Lattice Theory, American Mathematical Society Colloquium Publications, 1940.
[10] S. Black, Deriving an approximation algorithm for automatic computation of ripple effect measures, Information and Software Technology 50 (7–8) (2008) 723–736.
[11] S. Bohner, R. Arnold, Software Change Impact Analysis, IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.
[12] S.A. Bohner, Impact analysis in the software change process: a year 2000 perspective, in: Proceedings of the International Conference on Software Maintenance, 1996, pp. 42–51.
[13] B. Breech, A. Danalis, S. Shindo, L Pollock, Online impact analysis via dynamic compilation technology, in: Proceedings of the International Conference on Software maintenance, 2004, pp. 453–457.
[14] L.C. Briand, J. Daly, J. Wst, A unified framework for coupling measurement in object oriented systems, IEEE Transactions on Software Engineering 25 (1) (1999) 91–121.
[15] L.C. Briand, J. Wust, H. Lounis, Using coupling measurement for impact analysis in object-oriented systems, in: Proceedings of the International Conference on Software Maintenance, 1999b, pp. 475–482.
[16] G. Canfora, L. Cerulo, Fine grained indexing of software repositories to support impact analysis, in: Proceedings of the International Workshop on Mining Software Repositories, 2006, pp. 105–111.
[17] P. Cellier, Formal concept analysis applied to fault localization, in: Proceedings of the International Conference on Software Engineering, 2008, pp. 991–994.
[18] W.J. Conover, Practical Nonparametric Statistics, John Wiley & Sons, 1998.
[19] B. Dit, M. Revelle, M. Gethers, D. Poshyvanyk, Feature location in source code: A taxonomy and survey, Journal of Software: Evolution and Process 25 (1) (2013) 53–95.
[20] T. Eisenbarth, R. Koschke, D. Simon, Locating features in source code, IEEE Transactions on Software Engineering 29 (3) (2003) 195–209.
[21] A. Elkharraz, P. Valtchev, H. Mili, Concept analysis as a framework for mining functional features from legacy code, in: Proceedings of the International Conference on Formal Concept Analysis, 2010, pp. 267–282.
[22] J. Ferrante, K. Ottenstein, J. Warren, The program dependence graph and its use in optimization, ACM Transactions on Programming Languages and Systems 9 (3) (1987) 319–349.
[23] B. Ganter, R. Wille, Formal Concept Analysis: Mathematical Foundations, Springer-Verlag, Berlin, 1986.
[24] M. Gethers, B. Dit, H. Kagdi, D. Poshyvanyk, Integrated impact analysis for managing software changes, in: Proceedings of the 2012 International Conference on Software Engineering, 2012, pp. 430–440.
[25] M. Gethers, D. Poshyvanyk, Using relational topic models to capture coupling among classes in object-oriented software systems, in: Proceedings of the 2010 IEEE International Conference on Software Maintenance, 2010, pp. 1–10.
[26] T. Girba, S. Ducasse, A. Kuhn, Using concept analysis to detect co-change patterns, in: Proceedings of the International Workshop on Principles of Software Evolution, 2007, pp. 83–89.
[27] H.S. Hamza, A semi-automated approach for analyzing, separating, and modeling of concerns in evolving systems, in: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, 2005, pp. 220–221.
[28] S. Hassaine, F. Boughanmi, Y. Gueheneuc, S. Hamel, G. Antoniol, Change impact analysis: an earthquake metaphor, in: Proceedings of the IEEE International Conference on Program Comprehension, 2011, pp. 209–210.
[29] W. Hesse, T. Tilley, Formal concept analysis used for software analysis and modelling, in: B. Ganter, G. Stumme, R. Wille (Eds.), Formal Concept Analysis, Lecture Notes in Computer Science, vol. 3626, Springer, Berlin/Heidelberg, 2005, pp. 259–282.
[30] S. Horwitz, T. Reps, D. Binkley, Interprocedural slicing using dependence graphs, ACM Transactions on Programming Languages and Systems 12 (1) (1990) 27–60.
[31] M.A. Jashki, R. Zafarani, E. Bagheri, Towards a more efficient static software change impact analysis method, in: Proceedings of the ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, 2008, pp. 84–90.
[32] P. Joshi, R.K. Joshi, Concept analysis for class cohesion, in: Proceedings of the European Conference on Software Maintenance and Reengineering, 2009, pp. 237–240.
[33] H. Kagdi, M. Gethers, D. Poshyvanyk, M. Collard, Blending conceptual and evolutionary couplings to support change impact analysis in source code, in: Proceedings of the IEEE Working Conference on Reverse Engineering, 2010, pp. 119–128.
[34] S. Khor, P. Grogono, Using a genetic algorithm and formal concept analysis to generate branch coverage test data automatically, in: Proceedings of the International Conference on Automated Software Engineering, 2004, pp. 346–349.
[35] H.H. Kim, D.H. Bae, Object-oriented concept analysis for software modularization, IET Software 2 (2) (2008) 134–148.
[36] R. Koschke, J. Quante, On dynamic feature location, in: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, 2005, pp. 86–95.
[37] J. Law, G. Rothernel, Whole program path-based dynamic impact analysis, in: Proceedings of the International Conference on Software Engineering, 2003, pp. 308–318.
[38] B. Li, X. Sun, H. Leung, S. Zhang, A survey of code-based change impact analysis techniques, Journal of Software Testing, Verification and Reliability (2012). doi:10.1002/stvr.1475. <http://dx.doi.org/10.1002/stvr.1475>.
[39] A. Lienhard, S. Ducasse, G. Arvalo, Identifying traits with formal concept analysis, in: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, 2005, pp. 66–75.
[40] K. Mens, T. Tourw, Delving source code with formal concept analysis, Journal of Computer Languages, Systems and Structures 31 (3–4) (2005) 183–198.
[41] N. Moha, J. Rezgui, Y.G. Guéhéneuc, P. Valtchev, G.E. Boussaidi, Using FCA to suggest refactorings to correct design defects, in: Proceedings of the 4th international conference on Concept lattices and their applications, 2008, pp. 269–275.
[42] A.T. Nguyen, T.T. Nguyen, J. Al-Kofahi, H.V. Nguyen, T.N. Nguyen, A topic-based approach for narrowing the search space of buggy files from a bug report, in: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, 2011, pp. 263–272.
[43] A. Orso, M.J. Harrold, Leveraging field data for impact analysis and regression testing, in: Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering, 2003, pp. 128–137.
[44] M. Petrenko, V. Rajlich, Variable granularity for improving precision of impact analysis, in: Proceedings of the International Conference on Program Comprehension, 2009, pp. 10–19.
[45] D. Poshyvanyk, A. Marcus, Combining formal concept analysis with information retrieval for concept location in source code, in: Proceedings of IEEE International Conference on Program Comprehension, 2007, pp. 37–48.
[46] D. Poshyvanyk, A. Marcus, R. Ferenc, T. Gyimothy, Using information retrieval based coupling measures for impact analysis, Empirical Software Engineering 14 (1) (2009) 5–32.

[47] V. Rajlich, P. Gosavi, Incremental change in object-oriented programming, IEEE Software 21 (4) (2004) 62–69.
[48] V. Rajlich, N. Wilde, The role of concepts in program comprehension, in: Proceedings of the 10th International Workshop on Program Comprehension, 2002, pp. 271–278.
[49] X. Ren, F. Shah, F. Tip, B.G. Ryder, O. Chesley, Chianti: a tool for change impact analysis of Java programs, in: Proceedings of the International Conference on Object Oriented Programming, Systems, Languages and Applications, 2004, pp. 432–448.
[50] P. Rovegard, L. Angelis, C. Wohlin, An empirical study on views of importance of change impact analysis issues, IEEE Transactions on Software Engineering 34 (4) (2008) 516–530.
[51] M. Sherriff, L. Williams, Empirical software change impact analysis using singular value decomposition, in: Proceedings of the International Conference on Software Testing, Verification, and Validation, 2008, pp. 268–277.
[52] S. Siegel, N.J. Castellan, Nonparametric Statistics for the Behavioral Sciences, Second ed., McGraw-Hill, 1988.
[53] M. Siff, T. Reps, Identifying modules via concept analysis, IEEE Transactions on Software Engineering 25 (6) (1999) 749–768.
[54] G. Snelting, F. Tip, Reengineering class hierarchies using concept analysis, ACM Transactions on Programming Languages and Systems 22 (3) (2000) 540–582.
[55] X. Sun, B. Li, S. Zhang, C. Tao, HSM-based change impact analysis of object-oriented java programs, Chinese of Journal Electronics 20 (2) (2011) 247–251.
[56] X.B. Sun, B.X. Li, C.Q. Tao, W.Z. Wen, S. Zhang, Change impact analysis based on a taxonomy of change types, in: Proceedings of the International Conference on Computer Software and Applications, 2010, pp. 373–382.
[57] X.B. Sun, B.X. Li, S. Zhang, C.Q. Tao, X. Chen, W.Z. Wen, Using lattice of class and method dependence for change impact analysis of object oriented programs, in: Proceedings of the Symposium on Applied Computing, 2011, pp. 1444–1449.
[58] T. Tilley, R. Cole, P. Becker, P. Eklund, A survey of formal concept analysis support for software engineering activities, in: G. Bernhard, S. Gerd, W. Rudolf (Eds.), Formal Concept Analysis, Lecture Notes in Computer Science, vol. 3626, Springer, Berlin/Heidelberg, 2005, pp. 250–271.
[59] P. Tonella, Concept analysis for module restructuring, IEEE Transactions on Software Engineering 27 (4) (2001) 351–363.
[60] P. Tonella, Using a concept lattice of decomposition slices for program understanding and impact analysis, IEEE Transactions on Software Engineering 29 (6) (2003) 495–509.
[61] P. Tonella, Formal concept analysis in software engineering, in: Proceedings of the International Conference on Software Engineering, 2004, pp. 743–744.
[62] P. Tonella, G. Antoniol, Inference of object oriented design patterns, Journal of Software Maintenance 13 (5) (2001) 309–330.
[63] P. Tonella, M. Ceccato, Aspect mining through the formal concept analysis of execution traces, in: Proceedings of the Working conference on Reverse Engineering, 2004, pp. 112–121.
[64] R.J. Turver, M. Munro, Early impact analysis technique for software maintenance, Journal of Software Maintenance: Research and Practice 6 (1) (1994) 35–52.
[65] C.J. van Rijsbergen, Information Retrieval, Butterworths, London, 1979.
[66] J. Viljamaa, Reverse engineering framework reuse interfaces, in: Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2003, pp. 217–226.
[67] S. Zhang, Z. Gu, Y. Lin, J.J. Zhao, Change impact analysis for AspectJ programs, in: Proceedings of the International Conference on Software Maintenance, 2008, pp. 87–96.
[68] T. Zimmermann, M.A. Zeller, P. Weissgerber, S. Diehl, Mining version histories to guide software changes, IEEE Transactions on Software Engineering 31 (6) (2005) 429–445.