IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received October 22, 2018, accepted November 6, 2018, date of publication November 27, 2018,
date of current version December 31, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2883533

Mining Variable-Method Correlation for Change Impact Analysis

CHUNLING HU[1], BIXIN LI[2], AND XIAOBING SUN[2,3], (Senior Member, IEEE)
[1]Department of Computer Science and Technology, Hefei University, Hefei 230026, China
[2]School of Computer Science and Engineering, Southeast University, Nanjing 210096, China
[3]School of Information Engineering, Yangzhou University, Yangzhou 225009, China

Corresponding author: Bixin Li (bx.li@seu.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61806068 and Grant 61672204, in part by the Visiting Scholar at Home and Aboard Funded Project of Universities of Anhui Province under Grant gxfxZD2016209, in part by the Key Technologies R&D Program of Anhui Province under Grant 1804a09020058, in part by the Major Program for Scientific and Technological of Anhui Province under Grant 17030901026, and in part by the Talent Research Foundation Project of Hefei University under Grant 16-17RC23.

ABSTRACT Software change impact analysis (CIA) is a key technique to identify the potential ripple effects of the changes to software. Coarse-grained CIA techniques such as file, class and method level techniques often gain less precise change impacts, which are difficult for practical use. Fine-grained CIA techniques, such as slicing, can be used to gain more precise change impacts, but need more time and space cost. In this paper, by combining the features of the coarse-grained technique and the fine-grained technique, a variable-method (VM) correlation-based CIA technique called VM-CIA is proposed to improve the precision of static CIA. First, the VM-CIA technique uses the abstract syntax tree (AST) of program to construct a novel intermediate representation called variable and method triple (VMT), which is used to analyze the correlation between the variables and methods. Second, the VM-CIA technique proposes the single-change impact analysis algorithm and multi-change impact analysis algorithm to compute the impact set based on the VMT representation. In addition, the VM-CIA technique can get a sorted impact set which is more accurate than the existing CIA techniques. The empirical results show that the VM-CIA technique can greatly improve the precision (19%) over traditional the CIA techniques, while at the cost of a little recall (5%). Moreover, the empirical studies also show that the VM-CIA technique predicts a ranked list of potential impact results according to the distance measure, which can greatly facilitate the practical use.

INDEX TERMS Change impact analysis, variable-method correlation, variable and method triple, impact propagation, call graph, impact set, abstract syntax tree.

I. INTRODUCTION

Software change impact analysis (CIA) is a process used to identify the potential ripple effects of the proposed changes [1]. The source code of software must be changed ultimately in the process of software maintenance. A change to a system, however small, can lead to several unintended effects, which are often not obvious or easy to detect [2]. Therefore, the change impact information is important for software maintenance activities. For example, it can be used to evaluate the feasibility of the change report so that we can select the best decision with minimal change impacts from all alternative change proposals [3]. In addition, it can also be used to judge whether the change impacts need secondary-changes after the changes are implemented to keep consistency [4]. Furthermore, the change impact information is useful for regression testing because the results of CIA is helpful for sorting or selecting the test suites [1], [3], [5], [6].

Most of current CIA techniques have a similar process, i.e., with a set of changed elements in a software system, called the *change set*, CIA attempts to determine a possibly larger set of elements, called the *impact set*, that requires attention or maintenance effort due to these changes [3]. The items in the change set and impact set can be either the specification of design or requirement, or the source code elements. At present, the resultant CIA results are often directly used for the modification task during software maintenance, so most current CIA techniques are performed on the source code [3]. The commonly used CIA techniques can be divided into two categories, 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) [7], [8]. The resultant impact set often has many false-positives, with many of its elements not really impacted [5], [9]. Thus this impact set they compute is very large and difficult for practical use [4]. Whereas dynamic CIA techniques consider some specific inputs, and rely on the analysis of the information collected during program execution to calculate the impact set [5], [9], [10]. Their impact set often includes some false-negatives, i.e., some of the real impacted elements are not identified [3]. 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 [11].

The impact set is usually expected to be safe and easy to compute, so static CIA techniques have advantages in terms of the above aspects. In addition, most of current static CIA techniques compute the impact set at method-granularity level [3]. However, as the programs are always complex, the method-granularity level impact set has large size and low accuracy [3], [5], [9], which obstructs its practical use. At present, among the static method-level CIA techniques, Breech presented a technique, which is simply called as Influence-CIA [12], with higher accuracy than traditional techniques. It uses parameter passing between methods to propagate the impacts of the change. On the one hand, if a method *a* calls method *b*, *a* passes parameter to *b*, and *b* returns value to *a*. Thus *a* and *b* will impact mutually. On the other hand, if method *a* calls method *b* and *a* passes reference parameter to *b*, in this case, *a* and *b* will impact mutually. This CIA technique analyzes fine-grained information between method calls, and has higher precision than transitive closure on call graph. However, it does not take into account the fine-grained dependence relationship within the method. Hence, its precision is also limited.

In this paper, the VM-CIA technique is performed at method and statement level to take into account variable-method correlation within method to improve the accuracy of CIA. More specifically, we combine coarse-grained and fine-grained techniques to improve precision. Firstly, we use the abstract syntax tree (AST) of program [13] to construct a novel intermediate representation structure named variable and method triple(VMT). Secondly, based on VMT representation, we propose single-change impact analysis algorithm Single-CIA to compute single-change impact set, and propose multiple-change impact analysis algorithm Multi-CIA to compute impact set. Moreover, VM-CIA technique can get a sorted impact set which is more accurate than the existing CIA techniques. Finally, an illustrative example and empirical study are presented to show the proposed VM-CIA can greatly improve the precision over traditional CIA techniques.

The rest of this paper is organized as follows: Section II presents the intermediate representation needed for the proposed VM-CIA technique. Section III introduces VM-CIA technique. Section IV presents an illustrative example to clearly show the process of VM-CIA technique Section V presents our empirical evaluation of VM-CIA technique. Section VI explores the related work. Finally, Section VII discusses the conclusions and future work.

II. INTERMEDIATE REPRESENTATION

Call graph has been widely used as the program intermediate representation for method-level CIA [3]. As the relation of method call is needed to analyze the propagation of the method change, call graph is also used as the intermediate representation in this paper, which is defined as follows.

Definition 1 (Call Graph): Call graph $CG = (V, E)$ is composed of a set of vertices and edges. V represents the methods of programs. $E = V \times V$ represents the call relationship. Specifically if method M_i calls method M_j, then a directed edge should be added on the call graph CG.

Figure 1 is a call graph of the example program in the Appendix.

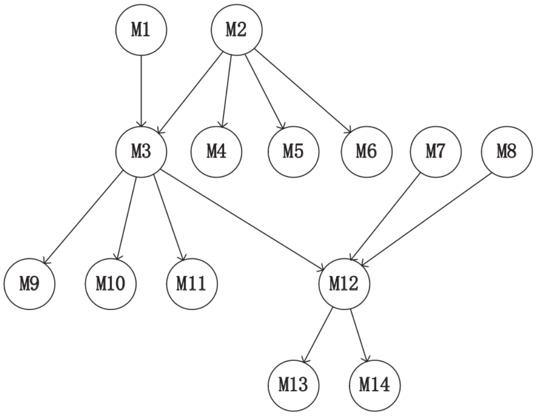


FIGURE 1. Call graph of the example program.

As call graph is a coarse-grained representation, and can only represent the call relationship which does not provide sufficient information to support CIA [2]. The accuracy of such kind of CIA is not satisfactory. Hence, we need to analyze fine-grained dependence information in the program to facilitate CIA. In this paper, we analyze deep into statements level, and extract their dependence relationship by parsing the source codes into the VMT set.

For the statements in the program, we mainly consider the dependence between variables and methods. In the object-oriented program, variables can be divided into *class field*, *method parameter*, and *local variable* within a method. Within a method, variables in different statements have different expressions and meanings. To distinguish them, we define variable *var* as follows.

$$var = cName.mName.vName : vType$$

This structure defines four attributes of the variable *var*, i.e., *class name*, *method name*, *variable name* and *variable type*, respectively. So the qualified name of a variable (the first three attributes of the structure) and variable type uniquely identify a variable. If the *var* is not the local variable of a method but a class field, the value of *mName* is null. Similarly, to uniquely identify a method, we define method as a structure as follows.

$$method = cName.mName.parameterList$$

The structure defines *class name*, *method name*, and *parameter list* of the method respectively. The *method name* and *parameter list* are the signature of a method. Since method call is usually binding with class instance with regards to object-oriented programs, besides the common variables, the class instance also belongs to the variable structure. Based on these structure definitions, the variable-method correlation is defined as follows.

Definition 2 (VariableMethod Correlation): There is a direct correlation between variable and method when their relation satisfies the following five forms:

1) Variable definition

Statement *var* represents the declaration or definition of a variable, and it does not correlate to other variables or methods.

2) Variable assignment

Statement $var1 = var2$ represents that the variable $var1$ correlates to variable $var2$.

Statement $var = f(\,)$ represents that the variable *var* correlates to the method f.

Statement $var = obj.f(\,)$ represents that the variable *var* correlates to the method f and the variable *obj*.

3) Method invocation

Method call statement $obj.f(\,)$ represents that the variable *obj* correlates to the method f.

Statement $f(arg)$ represents that the method f correlates to the parameter *arg*.

Statement $f(\,)$ represents that f does not correlate to any other variables.

4) Method as parameter

Statement $f(obj.g(arg))$ represents that the method f correlates to the variable *obj* and method g, and g correlates to the parameter *arg*.

5) Method chain invocation

Statement $var = f(\,).g(\,).h(\,)$ represents that the variable *var* correlates to method f, g and h.

The forms of variable-method correlations are extracted from method invocation, variable assignment and parameter passing statements in the program. And then, a more formal definition of variable-method correlation, called as variable and method triple, is defined as follows.

Definition 3 (Variable and Method Triple (VMT)): $VMT = <var_1, var_2, MS>$, where var_1 and var_2 represent the variables in the program. *MS* is a set of methods having direct correlation with var_1 or var_2, defined as $MS = \{m|m$ defined as $MT\}$, where $MT = <method, var, VS, MS>$. For *MT, method* represents the identification of *MT var* represents the object called by the method, and the set *VS* represents the set of method parameters.

In the above definition, *MT* defines a method declaration or a method invocation which has direct correlation with variables, parameters and other method invocations Method is the identification of method invocation and *VS* is a variable set directly correlated with method, such as the actual arguments for a method. *MS* represents the methods nested in the parameter list of method or the method invocation chains. The variables in *VMT* include class fields, formal parameters of the method, actual arguments of the method, and local variables of the method. As stated above, the *VMT* is mainly used to represent the definition and assignment of the variable, parameter passing, and the relation of method invocation. To further illustrate the *VMT* clearly, we give six *VMTs* of variable-method correlation which have listed in Definition 2 as follows.

1) $var1 = var2$
$VMT = <var1, var2, null>$
2) $var = f(\,)$
$VMT = <var, null, MS> MS = \{MT\}$,
$MT = <f, null, null, null>$
3) $var = obj.f(\,)$
$VMT = <var, obj, MS>, MS = \{MT\}$,
$MT = <f, obj, null, null>$
4) $obj.\,f(\,)$
$VMT = <null, obj, MS>, MS = \{MT\}$.
$MT = <f, obj, null, null>$
5) $f(par, obj.g(arg))$
$VMT = <null, null, MS>, MS = \{MT\}$,
$MT = <f, null, VS, MS'>, VS = \{par\}$,
$MS' = \{MT'\} MT' = <g, obj, VS', null> VS' = \{arg\}$
6) $var = f(\,).h(\,).g(\,)$
$VMT = <var, null, MS>, MS = \{MT1, MT2, MT3\}$,
$MT1 = <f, null, null, null>$,
$MT2 = <g, null, null, null>$,
$MT3 = <h, null, null, null>$

If there is only a variable declaration without variable definition or initialization in the statement of the program, we define $VMT = <var, null, null>$, for example, as declaration of a class field. Figure 2 is the *VMT* example of statement $v = obj\mathrm{n}_1.f(arg_1 arg_2, obj_2.g(arg_3))$.

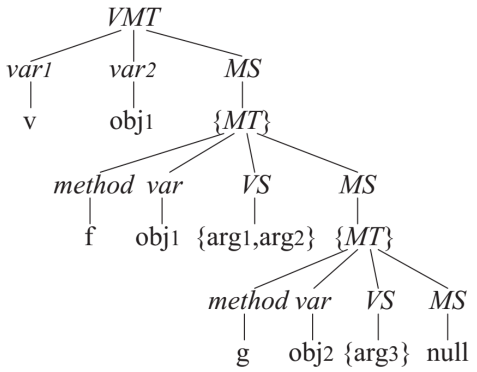


FIGURE 2. A Example of VMT of statement $v = obj_1.f(arg_1, arg_2, obj_2.g(arg_3))$.

With the definition of *VMT*, we can parse the program statements into the *VMT* structure. The parsed program is defined as follows.

Definition 4 (Program Representation): Program $P = \{C_1, C_2, \ldots\}$ represents all the classes of program P, and its class $C_i = \{FieldSet_i, MethodSet_i\}$ where

$FieldSet_i = \{VMT_{i1}, VM\ T_{i2}, \ldots\}$ is a *VMT* set which consists of the parsed fields of class C_i.

$MethodSet_i = \{Method_{i1}, Method_{i2}, \ldots\}$ is a method set of class C_i.

$Method_i = \{method_i, VMT_{i1}, VMT_{i2}, \ldots\}$ is a *VMT* set which consists of the parsed statements related to variable definition and use within $Method_i$.

These intermediate representations are used to perform impact analysis.

III. CHANGE IMPACT ANALYSIS

Given the change set, CIA is employed to estimate the impact set of software. In practical modification process, there are many elements needed to change. VM-CIA technique performs through two steps. First, we consider the single change impact analysis which computes the impacts of single change element in the program. Then, we merge all the single impact sets into the final impact set, which is called multiple change impact analysis. The VM-CIA technique is performed at method and statement level, and takes into account the definition and use of class fields, method parameters and local variables of method as the change source. The change set is composed of a set of variables. In order to evaluate the results of our VM-CIA and compare with other techniques, we compute the impact set at method level. This situation fits to single change impact analysis. That is, if a single statement related to a variable is changed, the methods correlated with this variable will be impacted. When multiple variables are changed, we can compute the impact set through merging all the impact sets of single change into the final impact set.

VM-CIA technique performs CIA after the programs are parsed into the set of *VMTs* We redefine the change set and impact set as follows.

Definition 5 (Change Set, CS): It is a set of *VMT* which consists of parsed statements related to changed variables, defined as: $CS = \{VMT \mid \exists var \in VMT \wedge VMT \in P \wedge var\ is\ changed\}$.

Variable *var* changed in program *P* could be class fields, method parameters and local variables within a method. The change type could be addition, deletion or type change of the variable. As the program has been parsed into the set of *VMTs*, the variables in the program can be mapped to their corresponding *VMTs*.

With the change set, impact set is computed and its form is defined as follows.

Definition 6 (Impact Set, IS): Impact set is defined as a set of two-tuples in which the first item is the *method* impacted by the changes, and the second item is the *distance* between the method and the changed element. Here, the *distance* corresponds to the distance between two vertices v_1 *and* v_2 on the call graph *(CG),* i.e., it is defined as the least number of edges reaching from v_1 to v_2, *or* reaching from v_2 *to* v_1. Hence, the definition of impact set can be denoted as follows.

$IS = \{<Method, distance>|Method \in CG \wedge Method$ is obtained by $CIA \wedge distance$ is the shortest distance between *Method* and the *methods* in which variable is changed$\}$.

Intuitively, the smaller the *distance* between the element in the impact set and the element in the change set is, the bigger the probability of this element will be impacted by the change [15]. The *distance* here can be also used to rank the probability of the elements to be impacted in the impact set. According to the definition of *CS* and *IS*, the input of VM-CIA is a set of *VMTs* and the output is a set of *methods*. The process of VM-CIA is divided into three steps. Figure 3 shows the architecture of the proposed VM-CIA technique.

- Compute the direct impact set of changes
- Perform single change impact analysis
- Perform multiple change impact analysis

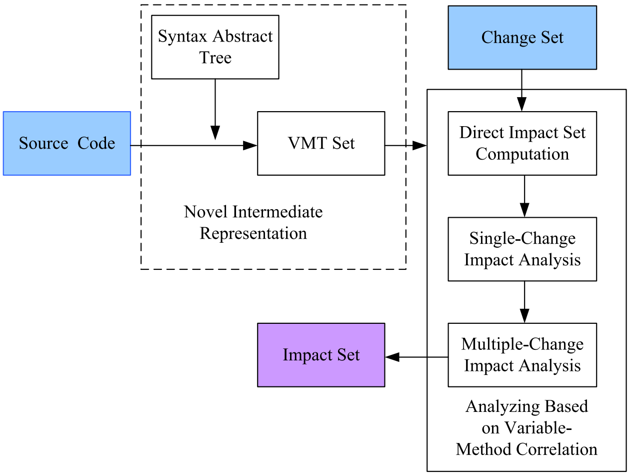


FIGURE 3. **The architecture of proposed VM-CIA technique.**

A. DIRECT IMPACT SET COMPUTATION

First, we compute the direct impact set (*DIS*) defined as follows.

Definition 7 (Direct Impact Set, DIS): $DIS = \{<<Method, 0>, VMTSet > |vmt \in CS \cap Method \bigvee \{vmt \in CS \bigwedge vmt \notin Method \bigwedge \exists vmt' \in Method \bigwedge \exists var \in vmt' \bigwedge var \in vmt \bigwedge vmt \in (CS \cap FieldSet)\}\ for\ \forall vmt \in VMTSet, vmt \in Method\}$.

In the above definition, *vmt* represents an element of *VMT*, and *VMTSet* is the set of *VMT*. The *VMTSet* is a subset of *VMT* of the *Method* in terms of *DIS*. The *Method belonging* to *DIS* needs to satisfy one of the two conditions. *1)* there exists the *VMT* belonging to *Method*, and the *VMT* belongs to *CS*. *2)* there exists the *VMT*′ belonging to *Method*, and it contains the same variable with *VMT* which belongs to both *FieldSet* and *CS*. In short, for the *Method*, either its parameters or local variables are changed or it uses the changed class fields.

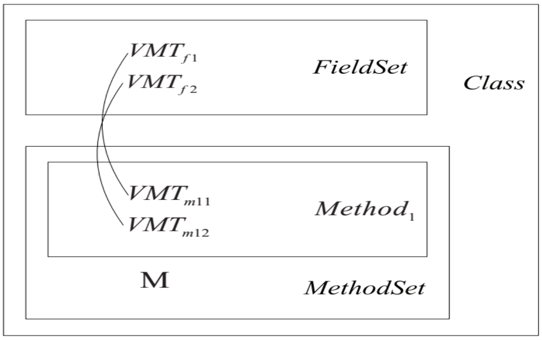


FIGURE 4. The VMT set of parsed class.

For the *VMTSet* of *DIS*, the *vmt* belonging to it needs to satisfy one of the two conditions: *1) vmt* belongs to both *Method* and *CS*. *2) vmt'* belongs to *Method*. At the same time, there exists the *vmt* belonging to *CS*, and the vmt' contains the same variable with *vmt*.

Figure 4 is an example of *VMT* set which consists of all *VMTs* from a parsed class. VMT_{f1} and VMT_{f2} are the parsed statements which relate to class fields. VMT_{m11} and VMT_{m12} are the parsed statements which relate to the variable and method correlation in $Method_1$. VMT_{f2} and VMT_{m12} contain the same variable. The *CS* is proposed to be $\{VMT_{f1}, VMT_{f2}, VMT_{m11}\}$. Since VMT_{m11} belongs to $Method_1$, and VMT_{f2} and VMT_{m12} contain a same variable, $Method_1$ is included in the *DIS* and the *VMTSet* related to $Method_1$ in *DIS* is $\{VMT_{m11}, VMT_{m12}\}$.

After the source code is parsed into the set of *VMTs*, we can start the impact propagation based on the correlation between different *VMTs* containing the same variable. The explanations of the relationship among the *VMTs*, *variables* and *VMTs*, *methods* and *VMTs*, *VMT* and *VMTSet* are given below.

1) If *VMT* and VMT' contain the same variable, *VMT* is correlated with VMT'.
2) If variable *var* belongs to *VMT*, *var* is correlated with *VMT*.
3) If there exists VMT' belonging to *VMTSet*, and *VMT* contains the same variable with VMT', *VMT* is correlated with VMT'.
4) If *Method* and *VMT* contain the same method, the Method is correlated with *VMT*.

B. SINGLE-CHANGE IMPACT ANALYSIS

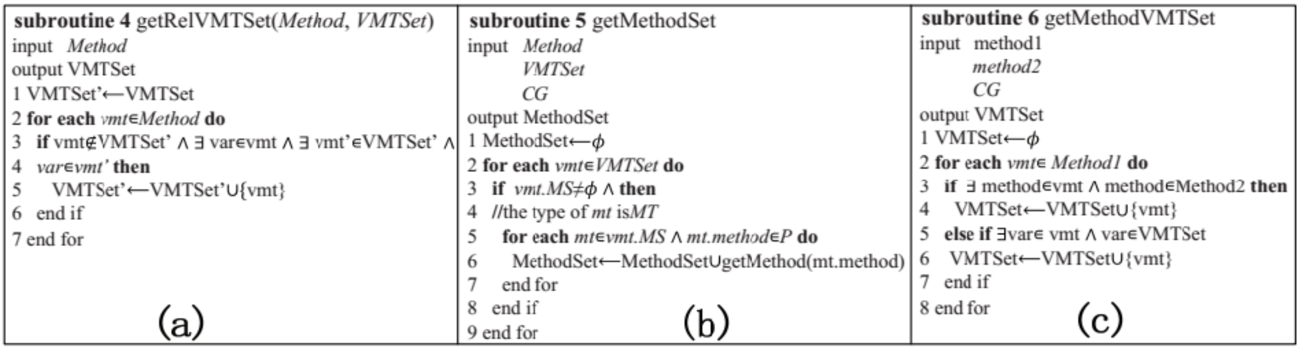
After computing the direct impact set, we can perform single change impact analysis which computes the impacts of single changed element in the program. First, we see how change impact is propagated on the call graph, and we define the impact propagation process as follows.

Definition 8 (Impact Propagation Process): On the call graph, impact propagation process is proceeded in two steps.

1) If method *a* calls method *b* and *b* is impacted by the change, the impact will propagate from *b* to *a*.
2) If method *a* calls both methods *b* and *c*, at the same time, *b* is impacted by the change and method *c* is correlated with *b*, the impact will propagate from *b* to *a* and continue to propagate to *c*.

The process of impact set computation needs seven routines to get the information of impact propagation, and the information takes into account the correlations between variables and methods. For each impacted method, seven subroutines are defined as follows.

1) subroutine 1. *getCaller(method)* returns a set of methods which call the *method* on the call graph.
2) subroutine 2. *getCallee(method)* returns a set of methods which are called by the *method* on the call graph.
3) subroutine 3. *getMethod(method)* returns a set of methods with the name of *method* on the call graph.
4) subroutine 4. *getRelVMTSet(Method, VMTSet)* returns the union of *VMTSet* and a set of *VMT* which are correlated with the elements of *VMTSet* in the *Method*. Details of this subroutine is shown in Figure 5(a).
5) subroutine 5. *getMethodSet(Method, VMTSet)* returns a set of methods which are correlated with *VMTSet* in *Method*. Detail of this subroutine is shown in Figure 5(b).
6) subroutine 6. *getMethodVMTSet(Method1,Method2)* returns a set of *VMT* which has correlation with *Method*2 in *Method*1. Detail of this sub-routine is shown in Figure 5(c).



```
subroutine 4 getRelVMTSet(Method, VMTSet)
input   Method
output VMTSet
VMTSet'←VMTSet
for each vmt∈Method do
  if vmt∉VMTSet' ∧ ∃ var∈vmt ∧ ∃ vmt'∈VMTSet' ∧
  var∈vmt' then
    VMTSet'←VMTSet'∪{vmt}
  end if
end for
```

(a)

```
subroutine 5 getMethodSet
input   Method
        VMTSet
        CG
output MethodSet
MethodSet←ϕ
for each vmt∈VMTSet do
  if vmt.MS≠ϕ ∧ then
  //the type of mt isMT
    for each mt∈vmt.MS ∧ mt.method∈P do
      MethodSet←MethodSet∪getMethod(mt.method)
    end for
  end if
end for
```

(b)

```
subroutine 6 getMethodVMTSet
input   method1
        method2
        CG
output VMTSet
VMTSet←ϕ
for each vmt∈ Method1 do
  if ∃ method∈vmt ∧ method∈Method2 then
    VMTSet←VMTSet∪{vmt}
  else if ∃var∈ vmt ∧ var∈VMTSet
    VMTSet←VMTSet∪{vmt}
  end if
end for
```

(c)

FIGURE 5. Three subroutines of the impact propagation process.

Algorithm 1 Single-CIA

```
Input:
    eDIS: an element of Direct Impact Set
    CG: call graph
Declare:
    calleeSet: a set of methods that are called by a method and
    extracted from the impacted VMTs
    callerSet: a set of methods that call a method
Use:
    addCallerIS(IS, m, eDIS.Method,2): it is used to traverse
    the unvisited caller methods on the call graph, and details
    are shown in Algorithm 2.
    addCalleeIS(IS, m,2): it is used to traverse the unvisited
    callee methods on the call graph, and details are shown
    in Algorithm 3.
Output:
    IS: impact set
1: $VMTSet' \leftarrow getRelVMTSet(eDIS.Method, eDIS.VMTSet)$
2: $calleeSet \leftarrow getMethodSet(eDIS.Method, VMTSet')$
3: $callerSet \leftarrow getCaller(eDIS.Method)$
4: for each $m \in callerSet$ do
5:     $IS \leftarrow IS \cup \{<m, 1>\}$
6: end for
7: for each $m \in calleeSet$ do
8:     $IS \leftarrow IS \cup \{<m, 1>\}$
9: end for
10: for each $m \in callerSet$ do
11:     addCallerIS(IS, m, eDIS.Method, 2)
12: end for
13: for each $m \in calleeSet$ do
14:     addCalleeIS(IS, m, 2)
15: end for
16: return IS
```

Algorithm 2 addCallerIS (*IS, m1, m2, Distance*)

```
Input:
    IS: impact set
    m1: a method
    m2: a method
    distance: distance between m1 and the changed method
    on call graph
Declare:
    calleeSet: a set of methods that are called by a method and
    extracted from the impacted VMTs
    callerSet: a set of methods that call a method
    pMethodSet: a set of methods
    cMethodSet: a set of methods
1: $VMTSet \leftarrow getMethodVMTSet(m1, m2)$
2: $VMTSet' \leftarrow getRelVMTSet(m1, VMTSet)$
3: $calleeSet \leftarrow getMethodSet(m1, VMTSet')$
4: $callerSet \leftarrow getCaller(m1)$
5: $pMethodSet \leftarrow \phi$
6: $cMethodSet \leftarrow \phi$
7: for each $m \in callerSet$ do
8:     if $m \notin IS$ then
9:         $pMethodSet \leftarrow pMethodSet \cup \{m\}$
10:        $IS \leftarrow IS \cup <m, distance>$
11:    end if
12: end for
13: for each $m \in calleeSet$ do
14:     if $m \notin IS$ then
15:         $cMethodSet \leftarrow cMethodSet \cup \{m\}$
16:         $IS \leftarrow IS \cup <m, distance>$
17:     end if
18: end for
19: for each $m' \in pMethodSet$ do
20:     addCallerIS (IS,m', m1, distance +1)
21: end for
22: for each $m' \in cMethodSet$ do
23:     addCalleeIS(IS, m', distance +1)
24: end for
```

7) *subroutine* 7. *getVMTSet(Method)* returns all the *VMTs* of the *Method*.

For a single changed method, we can compute its impact set through single-change impact analysis. The algorithm 1 is used to compute the impact set of the method.

The computation of the impact set is based on the impact propagate process. It uses the method in the *DIS* as a starting point on the call graph and propagates the impacts layer by layer from two directions, i.e. method calling and method being called on the call graph. Algorithm 1 is divided into two parts.

1) Line 3 computes the method set *callerSet* in which method calls *eDIS*. Lines 4-6 merge the methods in the *callerSet* into the impact set *IS*. In lines 10-12, each method of *callerSet* is used as the starting point and calls the *addCallerIS* procedure (Algorithm 2) to compute the impact set recursively from two directions.

2) Lines 1-2 compute the method set *calleeSet* in which method is called by *eDIS* and is correlated with *eDIS*. Lines 7-9 merge the methods in the *calleeSet* into *IS*. In lines 13-15, each method of *calleeSet* is used as the starting point and calls the *addCalleeIS* procedure (Algorithm 3) to compute the impact set recursively from two directions.

In addition, for *addCallerIS* procedure, methods m_1 and m_2 are used as parameters, where m_1 is used as the starting point to compute the impact set. The relation of m_1 and m_2 is that m_1 calls m_2. This procedure is also divided into two parts:

Lines 1-3 compute the method set *calleeSet* in which method has correlation with m_2 in m_1. In lines 13-18, the methods not belonging to *IS* are included in *IS*. In lines 22-24, each method which belongs to *calleeSet* and included in *IS* is used as the starting point. Then, it calls the *addCalleeIS* procedure to compute the impact set recursively from two directions.

Line 4 computes the method set *callerSet* in which method calls m_1. In lines 7-12, the methods not belonging to *IS* are

Algorithm 3 addCalleeIS (*IS, m, Distance*)

```
Input:
    $IS$: impact set
    $m$ : a method
    $distance$: distance between $m$ and the changed method on call graph
Declare:
    $calleeSet$: a set of methods that are called by a method and extracted from the impacted $VMTs$
    $callerSet$: a set of methods that call a method
    $pMethodSet$: a set of methods
    $cMethodSet$: a set of methods
1: $VMTSet \leftarrow getVMTSet(m)$
2: $calleeSet \leftarrow getMethodSet(m, VMTSet)$
3: $callerSet \leftarrow getCaller(m)$
4: $pMethodSet \leftarrow \phi$
5: $cMethodSet \leftarrow \phi$
6: for each $m' \in callerSet$ do
7:    if $m' \notin IS$ then
8:        $pMethodSet \leftarrow pMethodSet \cup \{m'\}$
9:        $IS \leftarrow IS \cup <m', distance>$
10:   end if
11: end for
12: for each $m' \in calleeSet$ do
13:    if $m' \notin IS$ then
14:        cMethodSet $\leftarrow$cMethodSet $\cup \{m'\}$
15:        $IS \leftarrow IS \cup <m', distance>$
16:    end if
17: end for
18: for each $m' \in pMethodSet$ do
19:    $addCallerIS\ (IS, m', m, distance + 1)$
20: end for
21: for each $m' \in cMethodSet$ do
22:    $addCalleeIS(IS, m', distance + 1)$
23: end for
```

Algorithm 4 Multi-CIA

```
Input:
    $DIS$: direct impact set
    $CG$: call graph
Output:
    $IS$: impact set
$IS \leftarrow \phi$
for each $eDIS \in DIS$ do
    $IS \leftarrow IS \cup \{<eDIS.Method, 0>\}$
    $IS' \leftarrow singleCIA(eDIS, CG)$
    for each $m \in IS'$ do
        if $m \notin IS$ then
            $IS \leftarrow IS \cup \{m\}$
        else if $\exists m' \in IS \wedge m' == m \wedge m.distance < m'.distance$ then
            $m'.distance = m.distance$
        end if
    end for
end for
return $IS$
```

included in *IS*. In lines 19-21, each method which belongs to *callerSet* and *IS* is used as the starting point and calls the *addCallerIS* procedure to compute the impact set recursively from two directions.

Finally, for the *addCalleeIS* procedure, it uses the method *m* as its parameter and the starting point of the impact set computation. This procedure is similar to the *addCallerIS* procedure. The only difference is that the *calleeSet* of *m* is composed of all methods which are called by *m* (Lines 1-2).

C. MULTI-CHANGE IMPACT ANALYSIS

The impact set of multi-change impact analysis can be obtained by computing the union of impact sets from single-change impact analysis. Algorithm 4 shows the detailed process of multi-CIA, which merges the direct impact sets (*DIS*) into the final impact set.

Algorithm 4 uses the method of direct impact set as the starting point of impact set computation and uses Algorithm 1 to get the single impact set. Finally, all the impacted methods are included in the final impact set. For a method M_i, we assume that P_i and C_i are the number of *Caller* and *Callee* of this method, respectively. The time complexities of Subroutine 1 and Subroutine 2 are $O(P_i)$ and $O(C_i)$, respectively. Assuming the number of statements within this method is S_i, and the number of *VMT* of the method is not more than the number of statements, therefore, the time complexity of Subroutine 3 and Subroutine 4 is $O(S_i)$. Since the number of method callers which relate to the statements of the method is not more than the number of *Callee* of this method, the time complexity of Subroutine 5 is $O(C_i)$. Obviously, the time complexity of Subroutines 6 and 7 is $O(S_i)$. Algorithm 1 calls Subroutines 4, 5, 1, the size of *callerSet* is not more than P_i, and the size of *calleeSet* is not more than C_i, so the time complexity of Algorithm 1 is $O(S_i + 2\,C_i + 2\,P_i)$, similarly, the time complexity of the *addCallerIS* procedure is $O(2\,S_i + 2\,P_i + 2\,C_i)$ and the time complexity of the *addCalleeIS* procedure is $O(S_i + 2\,P_i + 2\,C_i)$. Assuming the number of methods on the call graph is V, the number of call edges on the call graph is E and the number of statements of the program is S, the time complexity of Algorithm 1 is

$$O(\sum_{i=1}^{V} 2(S_i + P_i + C_i)) + O(2E) = O(S + V + E)$$

The time complexity $O(E)$ in the above formula is based on this fact that impact propagating from a method to its caller and callee needs to traverse its call edges on the call graph. Assuming the size of the change set is N, the ultimate time complexity of the VMCIA is $O\ (N\ (S + V + E))$.

The proposed technique VM-CIA takes into account variable-method correlation within method *and* parses source code into VMT set as a novel intermediate representation, and then proposes single impact analysis algorithm 1 and multiple impact analysis algorithm 4 to improve the accuracy of impact analysis through mining the correlation between variable-method. An illustrative example and empirical study in the following Section IV-V will further show the proposed technique VM-CIA can greatly improve the precision over traditional CIA techniques through mining variable-method correlation.

IV. AN ILLUSTRATIVE EXAMPLE

To illustrate the VM-CIA process, an example is presented in this section.

In the program of the Appendix, the parameter *i* of method M_3 in class C_1 (Figure 6(b)) is assumed to be changed. Then, we can get the direct impact set $DIS = \{<<C_1.M_3, 0>, VMTSet>\}$, where $VMTSet = \{VMT_1\}$, $VMT_1 = <C_1.M_3.i, null, null>$. Thus, *DIS* is first added to the impact set *IS*, i.e. $IS = \{<M_3, >\}$. Using M_3 as the starting point, we perform the impact propagation analysis from two directions on the call graph.

A. METHOD CALLING

Method M_3 calls methods M_9, M_{10}, M_{11}, and M_{12}, as shown in Figure 6(c). The statement in Line 5 within M_3 is correlated with *i*, and the statements in Line 9 and Line 5 are correlated with v_2. We denote the *VMTs* of Line 5 and Line 9 as follows

$$VMT_2 = <C_1.v_2, C_1.o_2, MS_1>, \quad MS_1 = \{MT_1\},$$
$$MT_1 = <C_1.M_{10}, C_1.o_2\{C_1.M_3.i\}, null>.$$
$$VMT_3 = <null, C_1.o_2, MS_2>,$$
$$MS_2 = \{MT_2\}, \quad MT_2 = <C_1.M_{12}, C_1.o_2, \{C_1.v_2\}, null>$$

VMT_3 is correlated with VMT_2 based on $C_1.v_2$, and VMT_2 is correlated with VMT_1 based on variable $C_1.M_3.i$. Then, we get the *calleeSet* of M_3 from VMT_1, VMT_2, and VMT_3, i.e. $calleeSet = \{M_{10}, M_{12}\}$. At this time, we get the *IS* by merging the *caleeSet* into *IS*, $IS = \{<M_3, 0>, <M_{10}, 1>, <M_{12}, 1>\}$. In the *IS*, M_3 belongs to the direct impact set and the distance between M_3 and M_{10}, M_{12} is 1. Next, every method of *calleeSet* is used as the starting point of impact set computation and calls the *addCalleeIS* procedure to compute the impact set on two directions on the call graph. For example, using M_{12} as the starting point, we need to judge whether M_{13} and M_{14} are impacted by M_{12} from the direction of method calling. We also need to judge whether M_7 and M_8 are impacted on the direction of method being called.

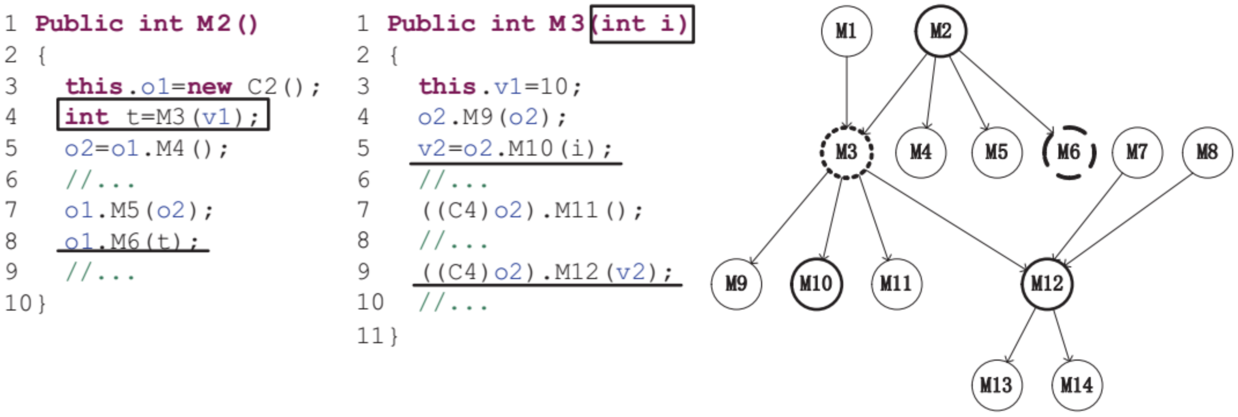
B. METHOD BEING CALLED

Methods M_1 and M_2 calls M_3, as shown in Figure 6(c) and Appendix. Based on the impact propagation process, the *callerSet* of M_3 is $\{M_1, M_2\}$ and the distance between M_3 and M_1, M_2 is 1. After merging *callerSet* into the *IS*, we have $IS = \{<M_3, 0>, <M_{10}, 1>, <M_{12}, 1>, <M_1, 1>, <M_2, 1>\}$. Then, we use the *callerSet* as the starting point of impact set computation, and the *addCallerIS* procedure is called to compute the impact set from two directions. As shown in Figure 6(c), we compute the impact set of M_2 on the direction of method calling. In class C_1, M_2 calls the methods M_3, M_4, M_5 and M_6. Based on the impact propagation process, M_2 is impacted because M_2 calls M_3. Therefore, judging whether methods M_4, M_5 and M_6 are impacted needs to consider whether they are correlated with M_3. The *VMTs* of the statements (Line 4 and Line 8 in Figure 6(a)) which are correlated with M_3 are as follows.

$$VMT_4 = <C_1.M_2.t, null, MS_3>, \quad MS_3 = \{MT_3\},$$
$$MT_3 = <C_1.M_3, this, C_1.v_1, null>$$
$$VMT5 = <null, C_1.o_1, \quad MS_4>,$$
$$MS_4 = \{MT_4\}, \quad MT_4 = <C_1.M_6, C_1.o_1\{C_1.M_2.t\}, null>$$

VMT_4 has correlation with VMT_5 due to $C_1.M_2.t$. Then, the *calleeSet* of M_2 is M_6, and distance between M_3 and M_6 is 2. After merging *calleeSet* into *IS*, we have $IS = \{<M_3, 0>, <M_{10}, 1>, <M_{12}, 1>, <M_1, 1>, <M_2, 1>, <M_6, 2>\}$.

In a similar way, using M_{12} as the starting point, a set of methods $\{M_7, M_8, M_{13}, M_{14}\}$ is added to the impact



```
1 Public int M2()
2 {
3   this.o1=new C2();
4   int t=M3(v1);
5   o2=o1.M4();
6   //...
7   o1.M5(o2);
8   o1.M6(t);
9   //...
10}
```

```
1 Public int M3(int i)
2 {
3   this.v1=10;
4   o2.M9(o2);
5   v2=o2.M10(i);
6   //...
7   ((C4)o2).M11();
8   //...
9   ((C4)o2).M12(v2);
10  //...
11}
```

(a) (b) (c)

FIGURE 6. **Process of change propagation.**

set. Finally, $IS = \{<M_3, >, <M_{10}, 1>, <M_{12}, 1>, <M_1, 1>, <M_2, 1>, <M_6, 2>, <M_7, 2>, <M_8, 2>, <M_{13}, 2>, <M_{14}, 2>\}$, as shown in Figure 7.

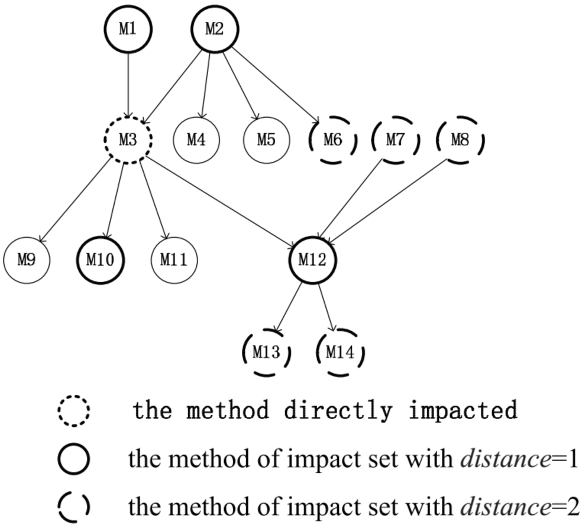


FIGURE 7. **The impact set of example program.**

The VM-CIA technique proposed in this paper can improve the accuracy of the impact results by analyzing the relation between variables and methods in the statements of a program. The Influence-CIA algorithm uses the parameters and return values of the method to construct the influence graph to compute the impact set in [12]. In this example, impact set computed by Influence-CIA is $\{M_1, M_2, M_3, M_4, M_5, M_6, M_7, M_8, M_9, M_{10}, M_{11}, M_{12}, M_{13}, M_{14}\}$.

From this example, we see that the size of the impact set computed by our approach is smaller than the size of the Influence-CIA. The difference between these two CIA techniques is that Influence-CIA takes into account all the cases of parameter passing and control flow of the program. In the method M_2, object o_1 calls the methods M_4, M_5 and M_6, that is, the o_1 is the reference parameter of M_4, M_5 and M_6. According to the Influence-CIA process, the methods are mutually impacted because they have reference parameter passing. Thus, all methods in M_2 are included in the impact set. We will further show the effectivity of the VM-CIA technique in the next section on some real open projects.

V. EMPIRICAL STUDY

A. RESEARCH QUESTIONS

The CIA proposed in this paper is closely related to the Influence-CIA. We compare our VM-CIA with Influence-CIA. In addition, we would like to see the effect of the distance on the impact set. The research questions we seek to answer are:

RQ1: Compared to Influence-CIA, can VM-CIA compute a smaller impact set?

RQ2: Compared to Influence-CIA, can VM-CIA compute an impact set with fewer false-positives without severely missing the false-negatives?

RQ3: How does the distance affect the accuracy of impact set?

B. RESEARCH SUBJECTS

To evaluate the effectivity of the VM-CIA technique presented in this paper, we select five open source Java projects from some software application domains. These Java projects are *NanoXML*,[1] *log4j*[2] *JUnit*[3] *HttpCore*[4] and *Heritrix*[5] *NanoXML* is a small XML parser. *log*4 *j* is an open source project of Apache and used to control the output of log information. *JUnit* is a simple framework to write repeatable tests. It is an instance of the *xUnit* architecture for unit testing frameworks. *HttpCore* is a set of low level HTTP transport components that can be used to build custom client and server side *HTTP* services with a minimal footprint. *Heritrix* is the Internet Archive's open-source, extensible, web-scale, archival-quality web crawler project.

We select some successive releases of these Java projects to perform empirical studies. Some basic statistics of these projects, including the number of versions (N_v), the number of classes (N_c), the number of methods (N_m) and the lines of code (*LOC*) are presented in Table 1.

TABLE 1. **Research subjects.**

Name	Statistics	V_0	V_1	V_2	V_3	V_4
NanoXML	N_v	2.0	2.1	2.1.1	2.2	2.2.1
	N_c	24	26	26	29	29
	N_m	308	392	397	431	432
	LOC	8255	9643	9730	11583	11412
Log4j	N_v	1.0	2.0	3.0		
	N_c	124	134	198		
	N_m	805	711	1653		
	LOC	17950	16262	32806		
JUnit	N_v	3.4	3.5	3.6	3.7	3.8
	N_c	33	52	53	52	56
	N_m	356	461	469	472	514
	LOC	3776	4682	4860	4833	5217
HttpCore	N_v	0.9	1.0	1.1	1.2	1.3
	N_c	351	365	414	424	430
	N_m	2193	2459	2781	2886	3065
	LOC	70925	78715	87078	91345	92532
Heritrix	N_v	0.2.0	0.4.0	0.6.0	0.8.0	1.10.0
	N_c	140	208	238	271	569
	N_m	1261	1901	1954	2228	5021
	LOC	25585	39957	43781	49433	110831

C. MEASURES

We try to evaluate the effectivity of VM-CIA technique from multiple aspects. First, we focus on the size and accuracy of its impact set. If the size of the impact set is too large, the precision of the impact set will be influenced. That is,

---

[1] https://sourceforge.net/projects/nanoxml/

[2] http://logging.apache.org/log4j/

[3] http://sourceforge.net/projects/junit/

[4] http://hc.apache.org/httpcomponents-core-ga/index.html

[5] http://sourceforge.net/projects/archive-crawler/

many false-positives are probably included in the impact set, which limits its practical application.

In addition, to evaluate the accuracy of the VM-CIA technique, two widely used metrics are *precision* and *recall* [14]. The combination of these two measures is 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 which are actually changed during bug fixing. *Estimated Set* is the impact set predicted by the CIA techniques.

D. PROCESS

First, we use the JDT (Java Development Tools)[6] to parse the Java source code into AST (abstract syntax tree). Then, we parse the AST related to the variable method correlation into *VMT* set. And we can get the actual change set by comparing the releases in *CVS* or *SVN* and map the changed elements into the corresponding *VMTs*[15]. In empirical study, we select the change set and impact set according to the sorted change set queue based on the distance between the impact element and the change element. The process of the empirical study is as follows.

1) CHANGE SET AND ACTUAL IMPACT SET SELECTION

For the input of CIA, there are three kinds of elements in the change set, i.e. class fields, method parameters, and local variable of a method. Due to the difficulty of identifying the relations among the change actions, we mainly select the simple changes in the program, such as declaration of class fields, method parameter list. In addition, we cannot get the accurate actual impact set. A widely used approach is that the actual impact set consists of actual changed methods in the program. However, we can know which methods are changed or deleted through version comparison during their evolution. Simply, the change of methods is due to the variable changes [7].

2) IMPACT SET COLLECTION

The main task of CIA is to predict the potential impacts of the changes made to the software. In our experiments, we use our VM-CIA technique and InfluenceCIA to compute the impact set from the change set obtained in the above step. In addition, we also collect the accuracy of the impact sets with different distance values.

E. RESULTS AND ANALYSIS

The empirical studies analyze each two consecutive versions ($V_i \rightarrow V_{i+1}$) of these five Java projects and identify the change set from V_i. Then, we compute the impact set of the change set on V_i, and compare the impact set with the actual impact set of V_i to evaluate the accuracy of the CIA. In this section, we report the results collected from the empirical studies to answer these three research questions.

1) RQ1

In this paper, we analyze the variable and method correlation to perform proposed VM-CIA. We first focus on the comparison of the size of the impact set of VM-CIA and the Influence-CIA. Table 2 reports the size of the impact sets of these two CIA techniques, i.e., *VM-IS* and *Influence-IS*. During the evolution of all versions for these five Java projects, the size of *VM-IS* is smaller than that of *Influence-IS*. The third column in Table 2 is the size of direct impact set which consists of the directly impacted methods due to class field change and parameter change. In addition, those methods which call the deleted methods must be impacted, so the deleted methods are also included in the *DIS*. From the results of Table 1 and Table 2, we can see that the larger the size of the program is, the larger the sizes of direct impact set and the impact set are On average, the size of *VM-IS* is two to three times bigger than the size of *DIS*, but the size of *Influence-IS* is three to six times bigger than the size of *DIS*. Therefore, from the perspective of the size of impact set, we see that our VM-CIA can generate a smaller impact set than the Influence-CIA, which will be more practical.

2) RQ2

In the previous section, we see that our VM-CIA can compute the impact set with smaller size. Moreover, if the size of

TABLE 2. **Size of the impact sets of VM-CIA and influence-CIA respectively.**

Name	Transaction	*DIS*	*VM-IS*	*Influence-IS*
NanoXML	$V_0 \rightarrow V_1$	41	128	167
	$V_1 \rightarrow V_2$	21	21	205
	$V_2 \rightarrow V_3$	61	128	225
	$V_3 \rightarrow V_4$	23	117	222
	AVG	36.5	98.5	204.8
Log4j	$V_0 \rightarrow V_1$	245	375	583
	$V_1 \rightarrow V_2$	355	485	550
	AVG	300.0	430.0	566.5
JUnit	$V_0 \rightarrow V_1$	69	217	270
	$V_1 \rightarrow V_2$	59	249	352
	$V_2 \rightarrow V_3$	53	241	320
	$V_3 \rightarrow V_4$	76	263	363
	AVG	64.3	242.5	326.3
HttpCore	$V_0 \rightarrow V_1$	201	831	1410
	$V_1 \rightarrow V_2$	133	332	1448
	$V_2 \rightarrow V_3$	335	752	1694
	$V_3 \rightarrow V_4$	362	934	1883
	AVG	257.8	712.3	1608.8
Heritrix	$V_0 \rightarrow V_1$	293	630	907
	$V_1 \rightarrow V_2$	483	886	1307
	$V_2 \rightarrow V_3$	100	744	1208
	$V_3 \rightarrow V_4$	841	1362	1701
	AVG	429.3	905.5	1280.8

---

[6] http://www.eclipse.org/jdt/

the impact set is smaller and accuracy is higher, the CIA technique is better. In this section, we evaluate the two CIA techniques in a more quantitative way. We use the precision and recall to assess the quality of the impact set. The number of false-positive and number of false-negative are closely related to the precision and recall measure respectively. The fewer false-positives the CIA produces, the higher its precision is. Similarly, the fewer false-negatives the CIA produces, the higher its recall is.

TABLE 3. **The precision and recall of VM-CIA and influence-CIA respectively.**

Name	Transaction	*VM-IS*		*Influence-IS*		*Gain*	
		P	*R*	*P*	*R*	Δ*P*	Δ*R*
NanoXML	$V_0 \rightarrow V_1$	0.41	0.96	0.31	0.96	0.1	0
	$V_1 \rightarrow V_2$	1.00	0.95	0.12	1.00	0.88	-0.05
	$V_2 \rightarrow V_3$	0.61	0.85	0.38	0.93	0.23	-0.08
	$V_3 \rightarrow V_4$	0.24	0.93	0.13	0.97	0.11	-0.04
	AVG	0.56	0.92	0.24	0.97	0.32	-0.05
Log4j	$V_0 \rightarrow V_1$	0.78	0.94	0.53	1.00	0.25	-0.06
	$V_1 \rightarrow V_2$	0.82	0.96	0.73	0.98	0.09	-0.02
	AVG	0.80	0.95	0.63	0.99	0.17	-0.04
JUnit	$V_0 \rightarrow V_1$	0.47	0.89	0.39	0.91	0.08	-0.02
	$V_1 \rightarrow V_2$	0.35	0.88	0.26	0.91	0.09	-0.03
	$V_2 \rightarrow V_3$	0.34	0.85	0.26	0.87	0.08	-0.02
	$V_3 \rightarrow V_4$	0.42	0.81	0.33	0.88	0.09	-0.07
	AVG	0.39	0.86	0.31	0.89	0.08	-0.03
HttpCore	$V_0 \rightarrow V_1$	0.33	0.86	0.20	0.90	0.13	-0.04
	$V_1 \rightarrow V_2$	0.42	0.80	0.12	0.97	0.30	-0.17
	$V_2 \rightarrow V_3$	0.48	0.92	0.22	0.96	0.26	-0.04
	$V_3 \rightarrow V_4$	0.43	0.83	0.24	0.92	0.19	-0.09
	AVG	0.41	0.85	0.19	0.94	0.22	-0.09
Heritrix	$V_0 \rightarrow V_1$	0.53	0.93	0.39	0.99	0.14	-0.06
	$V_1 \rightarrow V_2$	0.66	0.96	0.46	0.98	0.20	-0.02
	$V_2 \rightarrow V_3$	0.26	0.87	0.18	0.99	0.08	-0.12
	$V_3 \rightarrow V_4$	0.68	0.93	0.57	0.98	0.11	-0.05
	AVG	0.53	0.92	0.40	0.98	0.13	-0.06
AVG		0.54	0.90	0.35	0.95	0.19	-0.05

The precision and recall results of *VM-IS* and *Influence-IS* are compared in Table 3. For all these five Java projects, the precision of *VM-IS* is higher than that of *Influence-IS* but with a little lower recall value Specifically, on average, the recall of *VM-IS* is about 5% lower than that of *Influence-IS*. However, the precision of *VM-IS* is about 19% higher than that of *Influence-IS*, which is a great improvement. Form these results, we see that the gap of the precision between these two CIA techniques is bigger than the gap of their recall values.

Therefore, from the accuracy perspective, we can also conclude that our VM-CIA can be more precise to identify these actually impacted elements, which is more suitable for practical use.

3) RQ3

The above section shows that our VM-CIA technique is more practical than the Influence-CIA technique. For our VM-CIA technique, we used a distance measure. Intuitively, the further the distance between the impacted element and the changed element is, the less likely the element is impacted by the change. Generally, most changes are performed in the local part of a program. Therefore, we have one question, i.e., whether the impacts result from changes are local? In this section, we aim to answer this question.

We first define a Limited Impact Set (LIS) as follows

$$LIS(dist) = \{m | m \in IS \wedge m.distance \leq dist\}$$

The *dist* represents the predefined shortest distance between the method and the changed elements on the call graph, *LIS*(*dist*) is a set of methods reachable to the changes and the distance to the changes is less than *dist*. The maximal dist denoted as *maxDist* is the maximal distance of all elements of <Method, distance> in the impact set (*IS*). In practical, *Dist* could be defined as *maxDist* when precision and recall of the *LIS*(*dist*) are converged, otherwise, we could predefined an appropriate threshold for *Dist* according to source code and change set.

As shown in Figure 8, by increasing the *dist* value, the precision and recall values of *LIS*(*dist*) are also changing. More specifically, with the increase of the *dist* values, the precision of the *LIS*(*dist*) is reduced but its recall is increased. Thus, it shows that the nearer distance between the method to the change point on the call graph is, the more likely the method is impacted. On the contrary, the further distance between the method to the change point is, the less likely the method is changed. This shows that ranking the possibility of the potential impacts according to the distance is reasonable.

In addition, Figure 8 also shows some interesting phenomena. First, the *maxDist* value of *VM-IS* is small, while the *maxDist* value of *Influence-IS* is twice bigger, which shows that the *Influence-IS* has larger size and needs more time to check the impacts in practical application. In addition, with the increase of *dist* values, the precision and recall of *VM-IS* converge quickly, while precision and recall of *Influence-IS* converge slowly. It shows that VM-CIA is focused on the change point and computes a highly accurate impact set in the local range of the program. Actually, Influence-CIA uses the change point as its center, and propagates the impacts to all parts of the program continuously. Therefore, the recall value of Influence-CIA is approximating to 1, but the precision value is very low.

From above, we can conclude that the impacts are usually local for most changes to the program. CIA over traditional intermediate representation usually does not take into account the fine-grained dependence relationship within the method. Hence, its precision is limited. The proposed technique VM-CIA combines uses VMT set as intermediate representation to analyze deep into statement level and mine the correlation between variable-method to improve the accuracy of VM-CIA.

F. THREATS TO VALIDITY

Like most empirical studies, our study also has its limitations. Three threats to the validity of our study are discussed as follows.

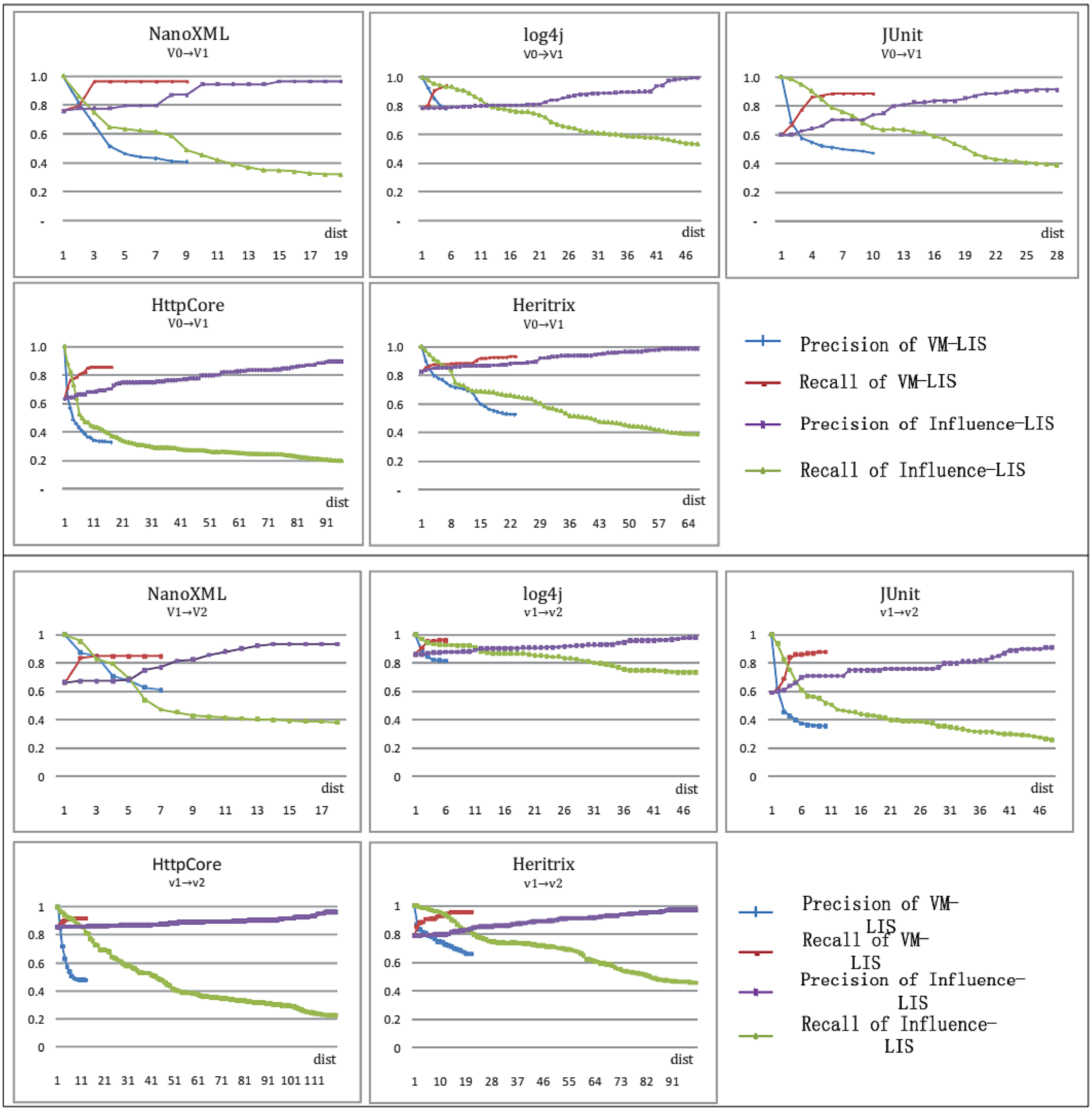


FIGURE 8. The accuracy of impact set with different dist values.

Firstly, we apply our technique to five subject programs in this study. We cannot guarantee that the results from our case studies can be generalized to other more complex or arbitrary programs. However, our subject programs are selected from open source projects and have been widely adopted for experimental studies [3]. Secondly, the change sets are obtained by randomly selecting the changes of the program, such as the declaration of class fields, method parameter list. But in practice, the change set is provided by feature location techniques [16], [17] or users. So the change set used in our study may not be the actual proposed changes in these programs versions. This may affect the evaluation results. Finally, for precision and recall measures, the real impact set is obtained by selecting the differences (method 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.

VI. RELATED WORK

During software maintenance, CIA can be performed on software design or source code level. In terms of source code, commonly used CIA techniques include dynamic techniques [9], [10] and static techniques [7], [18]–[22] Some also utilized both static and dynamic information in combination [23]–[26]

The dynamic CIA techniques are performed mainly based on analyzing the execution sequence of methods or dynamic method call graph through collecting information at runtime environment of the program. The classical dynamic techniques include *PathImpact CoverageImpact*, and *CollectEA* [27]. These techniques need to collect the information when executing the program at high cost. Furthermore, the execution traces cannot cover all the paths of the program, which will reduce the recall of the impact set. On the other hand, dynamic execution needs input of the test cases. A technique which uses different test cases to assist impact analysis is presented in [23]. It depends on the dependence graph (e.g. call graph) and takes into account different atomic changes, such as addition, deletion or change of variable and method.

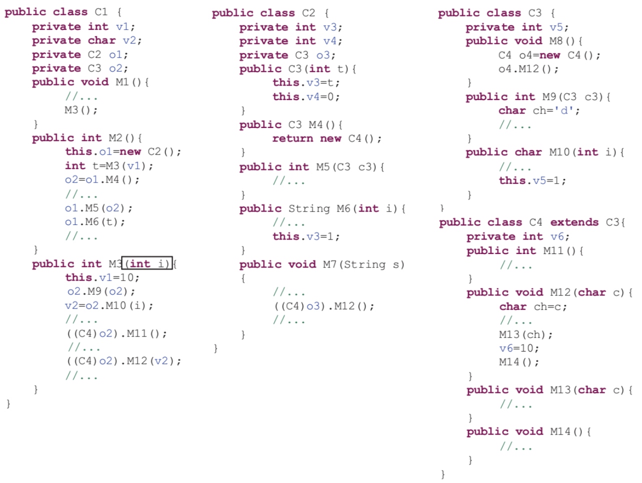
Static CIA techniques can be divided into structure static analysis, textual analysis, and historical analysis. The structure based techniques [7], [18]–[20], [28] need to construct the dependence relation of different entities (classes or methods). Petrenko and Rajlich develop a tool named JRipples, which can be used in the software incremental development. It allows software engineer to interactively perform CIA when developing software (the circulation processes of change, impact analysis, manual check and change again) [18] Hattori presents a depth based impact analysis algorithm on the call graph [29] As a classical intermediate representation of the program, call graph is widely used in CIA techniques [3], [9], [10], [12], [29]–[32]. The VM-CIA technique proposed in this paper also belongs to this category. Our technique can obtain a more precise impact set over traditional call graph based techniques through parsing the source core into VMT set The textual based techniques perform impact analysis mainly through coupling measure of modules or textual match [26], [33]–[35]. It is an approach which extracts the conceptual dependence (conceptual coupling) based on analysis of the nonsource code (comments). Poshyvanyket.al. use the information retrieval techniques, such as Latent Semantic Indexing, to analyze the similarity of two different texts, and then compute the degree of similarity of texts with coupling measure [26], [33]–[35]. The technique computes the impact set only through computing the similarity of two classes or methods, so the accuracy of impact set is not high. Since software development is managed with the version control system, with the prevalence of the technique of Mining Software Repository (MSR), mining the historical information of software to perform change impact analysis has been widely concerned by the research community [8], [26], [34], [36]–[39]. This kind of technique uses data mining and statistic analysis to mine the sequence relation, correlation of files, classes and methods. 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 these 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 by these co-change dependencies.

VII. CONCLUSION AND FUTUREWORK

Static CIA has been widely used to estimate the impact set due to the changes of source code. However, the impact set is larger than the actual impact set in most cases. How to improve precision is a big threat to the effectivity of static CIA technique. This paper attempts to narrow this gap. We present VM-CIA technique which parses the source code into variable and method triples (an intermediate representation) and perform CIA on this intermediate representation. This VM-CIA technique can obtain the impact set which consists of the methods with small distance to the changed elements and has high accuracy. The empirical results show that our VM-CIA technique can get a smaller impact set with high accuracy. Furthermore, we also get another conclusion that the impact results of the changes have the local feature which is expressed by the distance between the impacted element and the changed element in source code in this paper. In addition, the empirical studies also show the reasonability of the distance measure to rank the impact results, which can greatly facilitate practical use.

Although empirical studies are selected form the real world, the size of projects still cannot stand for the general project. And we will conduct experiments on other arbitrary and larger programs to evaluate the generality of our technique. In addition, we would like to compare our VM-CIA technique to other static CIA techniques, for example, textual based, historical based, and etc.

APPENDIX



```
public class C1 {
    private int v1;
    private char v2;
    private C2 o1;
    private C3 o2;
    public void M1(){
        //...
        M3();
    }
    public int M2(){
        this.o1=new C2();
        int t=M3(v1);
        o2=o1.M4();
        //...
        o1.M5(o2);
        o1.M6(t);
        //...
    }
    public int M3(int i){
        this.v1=10;
        o2.M9(o2);
        v2=o2.M10(i);
        //...
        ((C4)o2).M11();
        //...
        ((C4)o2).M12(v2);
        //...
    }
}
```

```
public class C2 {
    private int v3;
    private int v4;
    private C3 o3;
    public C3(int t){
        this.v3=t;
        this.v4=0;
    }
    public C3 M4(){
        return new C4();
    }
    public int M5(C3 c3){
        //...
    }
    public String M6(int i){
        //...
        this.v3=1;
    }
    public void M7(String s)
    {
        //...
        ((C4)o3).M12();
        //...
    }
}
```

```
public class C3 {
    private int v5;
    public void M8(){
        C4 o4=new C4();
        o4.M12();
    }
    public int M9(C3 c3){
        char ch='d';
        //...
    }
    public char M10(int i){
        //...
        this.v5=1;
    }
}
public class C4 extends C3{
    private int v6;
    public int M11(){
        //...
    }
    public void M12(char c){
        char ch=c;
        //...
        M13(ch);
        v6=10;
        M14();
    }
    public void M13(char c){
        //...
    }
    public void M14(){
        //...
    }
}
```

FIGURE 9. A simple Java example program.

REFERENCES

[1] B. Li, X. Sun, and J. Keung, ‘‘FCA–CIA: An approach of using FCA to support cross-level change impact analysis for object oriented Java programs,’’ *Inf. Softw. Technol.*, vol. 55, no. 8, pp. 1437–1449, Aug. 2013.

[2] K. A. Alam, R. Ahmad, A. Akhunzada, M. H. N. M. Nasir, and S. U. Khan, ‘‘Impact analysis and change propagation in service-oriented enterprises: A systematic review,’’ *Inf. Syst.*, vol. 54, pp. 43–73, Dec. 2015.

[3] B. Li, X. Sun, H. Leung, and S. Zhang, ‘‘A survey of code-based change impact analysis techniques,’’ *Softw. Test., Verification Rel.*, vol. 23, no. 8, pp. 613–646, 2013.

[4] M. Abi-Antoun, Y. Wang, E. Khalaj, A. Giang, and V. Rajlich, ‘‘Impact analysis based on a global hierarchical object graph,’’ in *Proc. IEEE 22nd Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Mar. 2015, pp. 221–230.

[5] Q. Wang, C. Parnin, and A. Orso, ‘‘Evaluating the usefulness of IR-based fault localization techniques,’’ presented at the Int. Symp. Softw. Test. Anal., Baltimore, MD, USA, 2015.

[6] C.-H. Liu, S.-L. Chen, and W.-L. Jhu, ‘‘Change impact analysis for object-oriented programs evolved to aspect-oriented programs,’’ presented at the ACM Symp. Appl. Comput., Taichung, Taiwan, 2011.

[7] X. Sun, B. Li, S. Zhang, C. Tao, X. Chen, and W. Wen, ‘‘Using lattice of class and method dependence for change impact analysis of object oriented programs,’’ presented at the ACM Symp. Appl. Comput., Taichung, Taiwan, 2011.

[8] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, ‘‘Mining version histories to guide software changes,’’ *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 429–445, Jun. 2005.

[9] H. P. Cai and R. Santelices, ‘‘Diver: Precise dynamic impact analysis using dependence-based trace pruning,’’ presented at the 29th ACM/IEEE Int. Conf. Automated Softw. Eng., Vasteras, Sweden, 2014.

[10] P. Meananeatra, S. Rongviriyapanish, and T. Apiwattanapong, ‘‘Identifying refactoring through formal model based on data flow graph,’’ in *Proc. Malaysian Conf. Softw. Eng.*, 2011, pp. 113–118.

[11] H. P. Cai and D. Thain, ‘‘DistIA: A cost-effective dynamic impact analysis for distributed programs,’’ in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng., (ASE)*, Singapore, Sep. 2016, pp. 344–355.

[12] B. Breech, M. Tegtmeyer, and L. Pollock, ‘‘Integrating influence mechanisms into impact analysis for increased precision,’’ in *Proc. 22nd IEEE Int. Conf. Softw. Maintenance, (ICSM)*, Philadelphia, PA, USA, Sep. 2006, pp. 55–64.

[13] E. Fauzi, B. Hendradjaya, and W. D. Sunindyo, ‘‘Reverse engineering of source code to sequence diagram using abstract syntax tree,’’ in *Proc. Int. Conf. Data Softw. Eng. (ICoDSE)*, 2016, pp. 1–6.

[14] W. M. Bramer, D. Giustini, and B. M. R. Kramer, ‘‘Comparing the coverage, recall, and precision of searches for 120 systematic reviews in Embase, MEDLINE, and Google scholar: A prospective study,’’ *Syst. Rev.*, vol. 5, no. 1, p. 39, Mar. 2016.

[15] W. Haifeng, Z. Kun, and L. Xia, ‘‘Teaching study of programming courses based on the SVN version control,’’ in *Recent Developments in Intelligent Computing, Communication and Devices*. Springer, 2019, pp. 487–493.

[16] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, ‘‘A systematic survey of program comprehension through dynamic analysis,’’ *IEEE Trans. Softw. Eng.*, vol. 35, no. 5, pp. 684–702, Sep./Oct. 2009.

[17] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, ‘‘Feature location in source code: A taxonomy and survey,’’ *J. Softw., Evol. Process*, vol. 25, no. 1, pp. 53–95, 2013.

[18] M. Petrenko and V. Rajlich, ‘‘Variable granularity for improving precision of impact analysis,’’ in *Proc. IEEE 17th Int. Conf. Program Comprehension (ICPC)*, May 2009, pp. 10–19.

[19] J. A. Dallal and A. Abdin, ‘‘Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review,’’ *IEEE Trans. Softw. Eng.*, vol. 44, no. 1, pp. 44–69, Jan. 2018.

[20] X. Sun, B. Li, C. Tao, W. Wen, and S. Zhang, ‘‘Change impact analysis based on a taxonomy of change types,’’ in *Proc. IEEE 34th Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, Jul. 2010, pp. 373–382.

[21] X. Sun, B. Li, S. Zhang, and C. Tao, ‘‘HSM-based change impact analysis of object-oriented Java programs,’’ *Chin. J. Electron.*, vol. 20, no. 2, pp. 247–251, 2011.

[22] S. Hassaine, F. Boughanmi, Y. Guéhéneuc, S. Hamel, and G. Antoniol, ‘‘A seismology-inspired approach to study change propagation,’’ in *Proc. 27th IEEE Int. Conf. Softw. Maintenance (ICSM)*, Sep. 2011, pp. 53–62.

[23] L. Zhang, M. Kim, and S. Khurshid, ‘‘Localizing failure-inducing program edits based on spectrum information,’’ in *Proc. 27th IEEE Int. Conf. Softw. Maintenance (ICSM)*, Sep. 2011, pp. 23–32.

[24] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst, ‘‘Combined static and dynamic automated test generation,’’ in *Proc. Int. Symp. Softw. Test. Anal.*, 2011, pp. 353–363.

[25] M. C. O. Maia, R. A. Bittencourt, J. C. A. de Figueiredo, and D. D. S. Guerrero, ‘‘The hybrid technique for object-oriented software change impact analysis,’’ in *Proc. 14th Eur. Conf. Softw. Maintenance Reeng. (CSMR)*, 2010, pp. 252–255.

[26] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk, ‘‘Integrated impact analysis for managing software changes,’’ in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, 2012, pp. 430–440.

[27] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta, ‘‘Symbolic PathFinder: Integrating symbolic execution with model checking for Java bytecode analysis,’’ *Automated Softw. Eng.*, vol. 20, no. 3, pp. 391–425, 2013.

[28] M. Acharya and B. Robinson, ‘‘Practical change impact analysis based on static program slicing for industrial software systems,’’ presented at the 33rd Int. Conf. Softw. Eng., Honolulu, HI, USA, 2011, pp. 746–755.

[29] L. Hattori, M. D'Ambros, M. Lanza, and M. Lungu, ‘‘Answering software evolution questions: An empirical evaluation,’’ *Inf. Softw. Technol.*, vol. 55, no. 4, pp. 755–775, 2013.

[30] N. Rungta, S. Person, and J. Branchaud, ‘‘A change impact analysis to characterize evolving program behaviors,’’ in *Proc. 28th IEEE Int. Conf. Softw. Maintenance (ICSM)*, Sep. 2012, pp. 109–118.

[31] B. Li, X. Sun, and H. Leung, ‘‘Combining concept lattice with call graph for impact analysis,’’ *Adv. Eng. Softw.*, vol. 53, pp. 1–13, Nov. 2012.

[32] B. Li, Q. Zhang, X. Sun, and H. Leung, ‘‘Using water wave propagation phenomenon to study software change impact analysis,’’ *Adv. Eng. Softw.*, vol. 58, pp. 45–53, Apr. 2013.

[33] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. D. Lucia, ‘‘On the equivalence of information retrieval methods for automated traceability link recovery,’’ in *Proc. IEEE 18th Int. Conf. Program Comprehension*, Jun. 2010, pp. 68–71.

[34] H. Kagdi, M. Gethers, and D. Poshyvanyk, ‘‘Integrating conceptual and logical couplings for change impact analysis in software,’’ *Empirical Softw. Eng.*, vol. 18, no. 5, pp. 933–969, 2013.

[35] M. Gethers and D. Poshyvanyk, ‘‘Using relational topic models to capture coupling among classes in object-oriented software systems,’’ in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2010, pp. 1–10.

[36] S. Panichella, A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall, ‘‘How can i improve my app? Classifying user reviews for software maintenance and evolution,’’ in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep./Oct. 2015, pp. 281–290.

[37] C. Gregg and M. Sherriff, ‘‘Teaching track faculty in computer science,’’ presented at the 49th ACM Tech. Symp. Comput. Sci. Educ., Baltimore, MD, USA, 2018.

[38] K. Shu, S. Wang, J. Tang, R. Zafarani, and H. Liu, ‘‘User identity linkage across online social networks: A review,’’ *ACM SIGKDD Explor. Newslett.*, vol. 18, no. 2, pp. 5–17, 2017.

[39] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, ‘‘Duplicate bug report detection with a combination of information retrieval and topic modeling,’’ presented at the 27th IEEE/ACM Int. Conf. Automated Softw. Eng., Essen, Germany, 2012.

CHUNLING HU received the Ph.D. degree in computer science from the Hefei University of Technology in 2011. She is currently an Associate Professor with the Department of Computer Science and Technology, Hefei University, China. Her research interests include software analysis and testing, complex system, and evolving systems etc. She is a CCF and ACM member.

BIXIN LI received the Ph.D. degree in software engineering from Nanjing University in 2001. He is currently a Full Professor with the School of Computer Science and Engineering. He is also the Director of the Software Engineering Institute, Southeast University, China. His main research interests include software analysis, testing, and verification of complex system, and evolving systems. He is a senior CCF member.

XIAOBING SUN received the bachelor's degree in computer science and technology from the Jiangsu University of Science and Technology in 2007, and the Ph.D. degree from the School of Computer Science & Engineering, Southeast University in 2012. Then, he joined the School of Computer Science & Engineering, Southeast University. He is currently an Associate Professor with the School of Information Engineering, Yangzhou University, China. His research interests include software maintenance and evolution, software repository mining, and intelligence analysis. He has been authorized over 20 patents, and published over 80 papers in referred international journals such as STVR, IST, JSS, SCIS, and FCS, and conferences, including ICSE, ASE, ICSME, SANER, and ICPC. He is a senior CCF and ACM/IEEE member.

• • •