2018 IEEE International Conference on Software Quality, Reliability and Security

A Program Slicing-based Bayesian Network Model for Change Impact Analysis

Ekincan Ufuktepe
Department of Computer Engineering
Izmir Institute of Technology
Izmir, Turkey
ekincanufuktepe@iyte.edu.tr

Tugkan Tuglular
Department of Computer Engineering
Izmir Institute of Technology
Izmir, Turkey
tugkantuglular@iyte.edu.tr

***Abstract*—Change impact analysis plays an important role in identifying potential affected areas that are caused by changes that are made in a software. Most of the existing change impact analysis techniques are based on architectural design and change history. However, source code-based change impact analysis studies are very few and they have shown higher precision in their results. In this study, a static method-granularity level change impact analysis, that uses program slicing and Bayesian Network technique has been proposed. The technique proposes a directed graph model that also represents the call dependencies between methods. In this study, an open source Java project with 8999 to 9445 lines of code and from 505 to 528 methods have been analyzed through 32 commits it went. Recall and f-measure metrics have been used for evaluation of the precision of the proposed method, where each software commit has been analyzed separately.**

Keywords—change impact analysis; program analysis; bayesian network

I. INTRODUCTION

Software modification and evolution has become an integral part of software development process. It has become competitive and challenging among organizations to keep up with the frequent changes in technologies, trends and fulfilling customer's frequently changing demands. These frequent changes cause serious changes in source codes. For instance, showing that change is continual in programming and intensive, it has been mentioned that Google performs more than 20 code changes per minute [1]. Furthermore, 50% of their code changes every month. On the other hand, in a software development life cycle, software maintenance has been described as the most difficult, expensive and labor-intensive process [2]. Therefore, in terms of time, effort and cost, it is important to locate and predict the possible effected codes after a change is committed. To locate the possible effected codes, change impact analysis is used.

In literature, it has been observed that, proposed approaches on change impact analysis (CIA) performed their prediction by focusing on mining change history and previous version of repositories [3], [4] . However, late and few studies have shown that dependency-based CIA techniques have focused on source code level [5]. The advantage of source code-based CIA techniques is that they identify the change impact in the final program source code. Furthermore, it is possible that they can improve the precision of the change impact analysis results because they directly analyze implementation details.

In this paper, a new model called Change Effect Graph (CEG) has been proposed. Change Effect Graph is a source code level, dependency-based model that is used for CIA. It uses program slicing, call graph and change information in a Bayesian Network.

The paper is organized as follows. In Section II, a summary of related work on CIA that is based on dependency and static analysis is shown. In Section III, a theoretical background that is required to understand concepts of CEG, backward slicing and Bayesian Networks is given. In Section IV, the CEG model and its formal definition has been proposed. In Section V, the experimental setup of our study is shown. In Section VI, the case study and selected open source project for our study has been described. In Section VII, the results of our study are given. In Section VIII, threats of our study and results has been discussed. Finally, Section IX, concludes the paper and mentions about the future work on CEG.

II. RELATED WORK

Badri et al. [6] proposed a static approach based on dependency analysis. By combining static program analysis and call graphs, they have formed a directed graph model called "*control call graph*" to support and predict change impact analysis. Since that call graphs are at method-level granularity, the proposed approach is at method-level granularity. The *control call graph* is based on "*if*" (also include *while*) conditions in the source codes and the method interactions between them. Thereby, when a method is changed, a control call graph allows to find the possible impacted methods through reachable methods. This provides to exclude methods from the estimated impact set that has no dependency relationship between that change method.

Sun et al. [7] provided a static CIA approach, by focusing on the change types. They have given a taxonomy of change types that some of them were referred from [8]. To improve the precision of their estimation on impacted sets, their approach relies on three factors. The first one is the change types of a modified entity that they have classified. Second factor is the dependencies between the changed entity and other entities. Third is finding the initial impact set. The initial impact set has

978-1-5386-7757-5/18/$31.00 ©2018 IEEE
DOI 10.1109/QRS.2018.00062

an important effect on the final impact. The more accurate estimation on initial impact, the more precise final impact set become. Their CIA technique has concentrated on the class and call member level granularities.

Tonella [9] proposed an approach that uses both program slicing and concept analysis. By combining the program representation of program slicing called decomposition slice concept analysis, a lattice called concept lattice of decomposition slice has been generated. Using lattices has allowed to provide more information about the relationships between slices.

Bayesian Networks have already been used in CIA. However, instead of using it on source code and its information, it has been used in architectural design and history. For instance, Tang et al. [10] proposed a CIA technique that uses Bayesian Networks in architectural design. Based on the design decisions and design elements, the Bayesian Network used to quantify these design relationships and elements. An example of Bayesian Network usage in change history was proposed by Mirarab et al. [11] called that uses a Bayesian Dependency History Model. They have used two sources of information that are; change history from CVS (Concurrent Version System) and dependency metrics which are extracted and calculated with static analysis. As for dependency metrics, coupling information and package relationship information have been used. This information is then used in Bayesian Network for training, to predict changes.

Ren et al. [12] proposed a CIA technique that is based on test cases and call graphs. Their technique assumes that a test suit of regression tests, which that have access to the original and modified versions of the source code. First their technique analyzes the changed source code in a method level granularity. Then, for each test case a call graph is constructed. After the call graphs are generated, their analysis determines a subset of the test suit that is affected by the changes that are made. Using the subset of test suit, an analysis is performed on each test case's call graph. The analysis determines a subset of affected parts of source codes that are by the changes.

III. Theoretical Background

A. Bayesian Networks

Bayesian Networks (BN) are sometimes known as Bayes Nets or Bayesian Belief Networks. They are probabilistic directed acyclic graphical models. To express any conditional dependencies between variables (nodes) they use directed acyclic graphs. Each node is encoded with probabilistic information related from its parent nodes. If the node does not have any parent node, then probabilistic information that is directly related to the node is encoded. The nodes in the BN have their own Node Probability Table (NPT) and these tables are where the probabilistic information are stored. Each node's NPT size change by the number of parent nodes that it has. On the other hand, BN is extensively used in many fields. However, they are generally used for prediction or for reasoning. In this study, we use BNs for predicting the change impact set.

B. Program Slicing

Program slicing is a technique for simplifying programs that is performed on a set of program statements. The slicing process basically deletes the statements of the program that have no effect the values at a point of interest, which is also known as slicing criterion. Program slicing is categorized as static analysis-based and dynamic analysis-based program slicing.

In this paper, static analysis-based program slicing is used. There are two fundamental static analysis-based program slicing approaches. Backward slicing was proposed by Weiser [13]. Backward slicing performs its analysis on control-flow graphs (CFGs) and it is used to assist developers by helping to locate the parts of the program, which contain a bug. The second slicing technique is forward slicing and was proposed by Horwitz et al. [14]. Rather than performing its slicing on a CFG, they do it on system dependency graphs.

C. Call Graph

Call graphs are a directed graph representation that shows the call relationships between procedures. It is defined as a set of directed edges and each edge is a call to a target function. In general, there are two types of call graphs; *dynamic* and *static* call graphs. Dynamic call graph is constructed during an execution of a program, while static call graph is constructed by a program's source code. The *ideal* call graph [15] is the union of dynamic call graphs that is obtained from all possible executions of the program. Therefore, we can say that every dynamic call graph is a subset of the ideal call graph. On the other hand. Static call graph is a superset of ideal call graph.[16]. In this study, we have used a static call graph.

D. Change Impact Analysis EIS and AIS Relationships

This study aims to find all the affected methods and achieve a recall value 1.0, before half of the software development is complete. Arnold and Bohner [17] have defined seven different possibilities of Estimated Impact Set (EIS) and Actual Impact Set (AIS) relationships. EIS is a set of affected methods/class that is estimated by the CIA approach. AIS is a set of the actual set of affected methods. # represents changes in a set.

1. ***Best:***
   $EIS\# = AIS\#,$
   $|AIS\#| / |EIS\#| = 1$
2. ***Safe:***
   $|EIS\#| > |AIS\#|,$
   $AIS\# \subset EIS\#,$
   $0 < |AIS\#| / |EIS\#| < 1$
3. ***Safe (Not so good):***
   $|EIS\#| >> |AIS\#|,$
   $AIS\# \subset EIS\#,$
   $0 < |AIS\#| / |EIS\#| < 1$
4. ***Expected:***
   $|AIS\#| > |EIS\#|,$
   $EIS\# \subset AIS\#,$
   $0 < |EIS\#| / |AIS\#| < 1$

5. ***Not so Good:***
   AIS#| > |EIS#|,
   EIS# ⊂ AIS#,
   0 < |EIS#| / |AIS#| < 1

6. ***Not so Good:***
   |AIS# ∩ EIS#| > 0,
   AIS# ≠ EIS#

7. ***Not so Good:***
   |AIS# ∩ EIS#| = 0,

Based on the seven possibilities, the first, second and third possibilities are considered as *Best* and *Safe*. Furthermore, they can detect all the affected methods, while the other possibilities cannot find all the affected methods.

IV. CHANGE EFFECT GRAPH

Change effect graph (CEG) is a model that is used in the BN, which we use to detect to be affected methods in a version change. To construct a CEG, first, changed code lines are detected between two consecutive versions of software. The change information is gathered at method granularity level. This change information is combined with the call graph (CG) of the methods involved in change to create a CEG, which will be described in detail in Figure 1 and Table I. The created CEG presents the layout of the Bayesian Network.

A CEG is defined as a directed edge-labeled graph as $G(N, E, w)$, where:

1. Each node $n_i \in N$, where N is the set of methods of a software.
2. For each node $n_i, n_k \in N$, if there is a directed edge $e_j \in E$ exists representing $< n_i n_k >$, it corresponds to the effect relationship from n_i to n_k, where n_i effects n_k.
3. $w : E \rightarrow R$ is a function mapping each edge $e_j \in E$ to a label $R_l \in R, where\ l = \{1, 2, 3, 4, 5, 6\}$ and R is the set of rules, which is given in Table I. Set R consists of six rules $R = \{R_1, R_2, R_3, R_4, R_5, R_6\}$. Each rule contains information of the nodes represented by e_j $<n_i, n_k >$, which are caller method and callee method, respectively.

In shown Figure 1, where creation of a change effect graph is shown, where a change effect graph is built on method change information from difference of versions and caller-callee information from the call graph of the latter version.

A. Rule Definifions for Change Effect Graph

As in Table I, for the construction of the CEG, we define six rules. These rules, forms the direction of the edges of a CEG. In the first column of Table I, the rules are given. The second column represents caller method's change status (*C1*). Likewise, the third column represents callee method's change status (*C2*). Fourth column shows the change rate relationships between *C1* and *C2* methods. Furthermore, in the fourth column, where S(C1) and S(C2) are functions that return the number of statements that *C1* and *C2* has. In the last column, the edge of effect direction is shown.

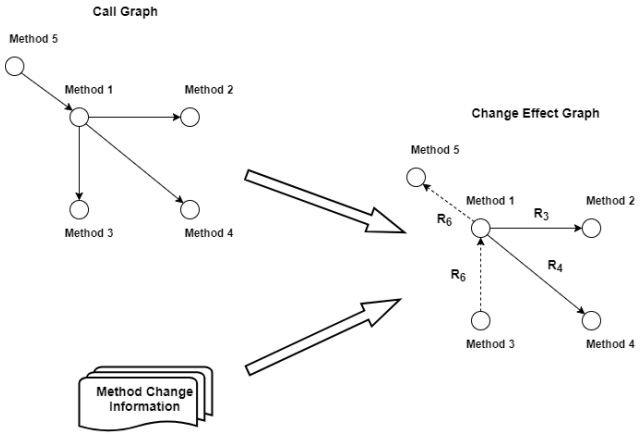

Fig 1. Creation of Change Effect Graph

The complete rule definitions are given below in Table I:

TABLE I. Rules for constructing change effect graph

Rule	*Caller (C1) Status*	*Callee (C2) Status*	*Change Rate Relationships*	*Effect Direction*
R_1	Unchanged	Unchanged	***C1 = C2***	C1 → C2
R_2	Unchanged	Changed	***C1 < C2***	C1 ← C2
R_3	Changed	Unchanged	***C1 > C2***	C1 → C2
R_4	Changed	Changed	***C1 > C2***	C1 → C2
R_5	Changed	Changed	***(C1 = C2)*** S(C1) ≥ S(C2) S(C1) < S(C2)	 C1 → C2 C1 ← C2
R_6	Changed	Changed	***C1 < C2***	C1 ← C2

- ***Rule R_1:*** The caller method *C1* and callee method *C2* are unchanged between two versions. Since there are no changes in both methods, then the call graph relationship (edge direction) is preserved as given in Table I in ***R_1***.
- ***Rule R_2:*** If callee method *C2* is changed and caller method *C1* is unchanged, we assume that the changes that are made in callee method *C2* will affect the caller method *C1*. Therefore, the edge <*caller method, callee method*> in call graph, has been transformed to <*callee method, caller method*> in change effect graph.
- ***Rule R_3:*** If the caller method *C1* is changed and callee method *C2* is unchanged, we assume that the changes that are made in caller method *C1* will affect the callee method *C2*. Therefore, the edge <*caller method, callee method*> in call graph, has been kept same in change effect graph.

Before explaining rules ***R_4***, ***R_5*** and ***R_6***, we introduce *CEG Precondition* that is used in defining for them. In this precondition, three cases are observed, where both caller *C1* and callee *C2* methods are changed at the same time.

CEG Precondition: If both *caller method* and *callee method* are changed, we assume that the changes that are made in *callee method* and *caller method* could both affect each other. However, if we define two edges in a change effect graph such as <*caller method*, *callee method*> and <*callee method*, *caller method*> we will create a cyclic graph, which does not satisfy the definition of a Bayesian network. Therefore, we must assume that one of the methods has a higher effect than the other method. We select the method with the highest change percentage to have higher effect on the other method. This precondition applies to rules **Rule R_4, Rule R_5** and **Rule R_6**.

- ***Rule R_4:*** If Precondition is satisfied and if caller method *C1* has a higher change percentage than callee method's *C2* change percentage, we assume that the changes that are made in caller method *C1* will affect the callee method *C2* more than the changes in callee method *C2* affecting caller method *C1*. Therefore, the edge <*caller method*, *callee method*> in call graph, has been kept same in change effect graph.
- ***Rule R_5:*** If Precondition is satisfied and if callee method *C2* has the same bytecode-wise change percentage with caller method's *C1* change percentage, we assume that the changes that are made in caller method and callee method could affect each other equally. However, if the number of statements in *C1* (*S(C1)*) is greater or equal to *C2* (*S(C2)*), then the edge <*caller method*, *callee method*> in call graph, has been kept same in change effect graph. If the number of statements in *C2* (*S(C2)*) is more than *C1*'s number of statements (*S(C1)*), then the edge <*caller method*, *callee method*> in call graph is transformed into <*callee method*, *caller method*> in change effect graph. The reason is, if a method has more statements, this means that more statements have been changed. For instance, let's assume that we have two methods, *C1* and *C2* methods. In addition, assume that they have the same amount of change rate 0.25. However, the statements numbers are different. Let, *C1* have 4 statements in total and only 1 statement has been changed and let *C2* have 20 statements at total and 5 of its statements have been changed. Both of their changed rates are same, however, the total number of changed statements can be different. Therefore, when there is an equality between change rates, we change the effect direction based on the amount of changed statements.
- ***Rule R_6***: If Precondition is satisfied and if callee method *C2* has a higher change percentage than caller method's *C1* change percentage, we assume that the changes that are made in callee method will affect the callee method more than the changes in callee method affecting caller method. Therefore, the edge <*caller method*, *callee method*> in call graph, has been transformed to <*callee method*, *caller method*> in change effect graph.

B. Converting Directed Cyclic CEG to Directed Acyclic CEG

Applying the rules in Section IV-A and Table I might end up with creating directed cyclic graphs. Therefore, CEGs can have cycles and it is a graphical representation that shows the effect relationships between method. Meanwhile, BNs only uses the effect dependencies of CEG by using its graphical layout. However, according to BNs properties, a BN must be an acyclic graph. Thereby, if a CEG contains cycles, for a BN to use a CEG's layout the cycles must be removed from the CEG.

We recall that CEGs are used in BNs to obtain change impact analysis results and one of the fundamentals of BN is that they are directed acyclic graphs. Therefore, after applying rules, the initial CEG is analyzed if it contains any cycles. If any cycle exists in the initial CEG, *Feedback Arc Set Algorithm* [18] is used to remove cycles from initial CEG.

The main objective of the Feedback Arc Set Algorithm is to find a set of arcs (edges) that causes feedback, i.e. edges that generates cycles in the graph. These edges are found by first deciding an initial node on the graph, then the graph is traversed starting from that initial node. While traversing the graph, the first node that is revisited is where the edge creates a cycle. Then this edge is added into a set that is called Feedback Arc Set. After the graph is completely traversed, i.e. all nodes visited, all the edges in the Feedback Arc Set are removed from the graph to obtain final CEG. If there are no cycles in the initial CEG, the Feedback Arc Set Algorithm is not used and initial CEG becomes final CEG automatically.

C. Encoding Bayesian Network Nodes with Backward Slicing Probabilistic Information

With final CEG, the layout of the acyclic CEG is used for the construction of the BN by encoding the nodes, i.e. by filling in the Node Probabilistic Table (NPT) for each node, of the final CEG. NPTs require probabilistic values. Our proposal and one of the contribution of the paper is to obtain those probabilistic values through backward slicing and change rate.

The CEG is formed by call graphs, for this reason the backward slicing should use data flow information between method to method. The data could flow in between methods from both sides; 1) data flow from caller method to callee method, 2) data flow from callee method to caller method. Details of the two direction of data flows are given below:

1. ***Type 1: Data Flow from Caller Method to Callee Method:*** This data flow assumes that the callee method is affected by the caller method. For a caller method to affect callee method could happen through passed parameter. Therefore, for this type of relationship in the CEG, the backward slicing uses the passed parameters as variables for the slicing criterion and investigate them. If, callee method does not have any parameter, then we assume that caller method does not affect the callee method.
2. ***Type 2: Data Flow from Callee Method to Caller Method:*** This data flow assumes that the caller method is affected by the callee method. For a callee method to affect a caller method could happen through returned value from the callee method. Thereby, for this type of relationship in CEG, the backward slicing uses the returned variable for its slicing criterion. If the callee method does not return anything (*void method*), then in CFG there will be no newly created variable and no transition to any other statements in the CFG.

On the other hand, global variables are another option that caller and callee method could affect each other. However, backward slicing uses CFGs, which has a modular approach on each method. Therefore, it is very difficult to track global variables in CFGs and hard to differentiate which variables are global variables and which are not.

Before explaining how the probabilistic values are calculated and assigned to their respective NPTs, we require two definitions. Let S_X be a set of statements of a CFG (control-flow graph) of method X. Then let, S_X^Y be a subset of S_X, that represents the affected statements from backward slicing, where Y is the affecting method and X is the affected method. There are two types of slicing criteria; slicing by callee method's parameters (*Type 1*) and slicing by returned callee methods (*Type 2*). We define two different sets of statements based on slicing criteria. For *Type 1*, let $SP_X^Y \subseteq S_X$ be a set of sliced statements based on method's parameters. For *Type 2*, let $SR_X^Y \subseteq S_X$ be a set of sliced statements based on callee method's returning value.

To calculate the probability of *Type 1*, by using backward slicing, we divide the statements of callee method that are affected by its parameters to the total number of statements of callee method's CFG. The equation for *Type 1* is given in Equation 1.

$$\frac{|SP_X^Y|}{|SP_X|} \tag{1}$$

Similar to *Type 1*, we use backward slicing to calculate the probability of *Type 2*. However, we divide the statements of caller method that are affected by callee method's return value to the total number of statements of caller method's CFG. Therefore, the rather than using set SP we use set SR. The equation for *Type 2* is given in Equation 2.

$$\frac{|SR_X^Y|}{|SR_X|} \tag{2}$$

D. Application of Proposed Method

In this section, we discuss how we apply the rule definitions that form the CEG and how the backward slice information is represented with probabilistic values on a running example. Finally, we show how the probabilistic values are encoded into the nodes of the BN.

To determine which edge directions should be changed, by using the change information and call relationships we find the rules that should be applied on the edges. In Figure 2, on the left-hand side, we have a call graph of a program. Each node represents a unique method and each edge direction describes a *<from, to>* relationship, which means that *from method* calls *to method*. The nodes that are filled are the methods that have been changed. Therefore, the methods *A*, *C*, *D*, *E* and *H* in Figure 2 are changed. Let's assume that the change percentage in terms of the amount of changed lines of bytecode relationship between methods are given below:

- ***Assumption 1:*** Change % of *A* > Change % of *C*
- ***Assumption 2:*** Change % of *C* = Change % of *E*
- ***Assumption 3:*** Change % of *D* > Change % of *E*
- ***Assumption 4:*** Change % of *E* < Change % of *H*

For each edge by using call graph information and change percentage relationships between methods, the rules are applied below:

- Due to ***Assumption 1*** and rule $\boldsymbol{R_4}$ in Table I, the edge *<A,C>* in call graph has been kept same with in CEG.
- The call graph relationship between method *B* and *C* corresponds to the rule $\boldsymbol{R_2}$ in Table, where method *B* is *unchanged* and method *C* is *changed*. Therefore, the edge *<B,C>* in call graph has been has been transformed to *<C,B>* in CEG and shown with a dotted edge.
- Due to ***Assumption 2*** and rule $\boldsymbol{R_5}$ in Table I, the edge *<C,E>* in call graph has been kept same with in CEG.
- Due to ***Assumption 3*** and rule $\boldsymbol{R_4}$ in Table I, the edge *<D,E>* in call graph has been kept same with in CEG.
- The call graph relationship between method *E* and *G* corresponds to the rule $\boldsymbol{R_3}$ in Table I, where method *E* is *changed,* and method *G* is *unchanged*. Therefore, the edge *<E,G>* in call graph has been kept same with in CEG.
- The call graph relationship between method *F* and *G* corresponds to the rule $\boldsymbol{R_1}$ in Table I, where methods *F* and *G* are *unchanged*. Therefore, the edge *<F,G>* in call graph has been kept same with in CEG.
- The call graph relationship between method *E* and *H* corresponds to the rule $\boldsymbol{R_6}$ in Table I, where methods *E* and *G* are *changed*. Therefore, the edge *<E,H>* in call graph has been has been transformed to *<H,E>* in CEG and shown with a dotted edge.

Once final CEG is ready, the CEG is used for the construction of the BN. It is important to recall that CEGs and BNs are not same. CEGs are directed graphs that can have cycles and a model that represents the affecting relationships between methods. BNs are directed acyclic graphs and their nodes contain probabilistic information. A BN only uses the affecting relationships between methods (graph layout) from final CEG, which does not contain cycles.

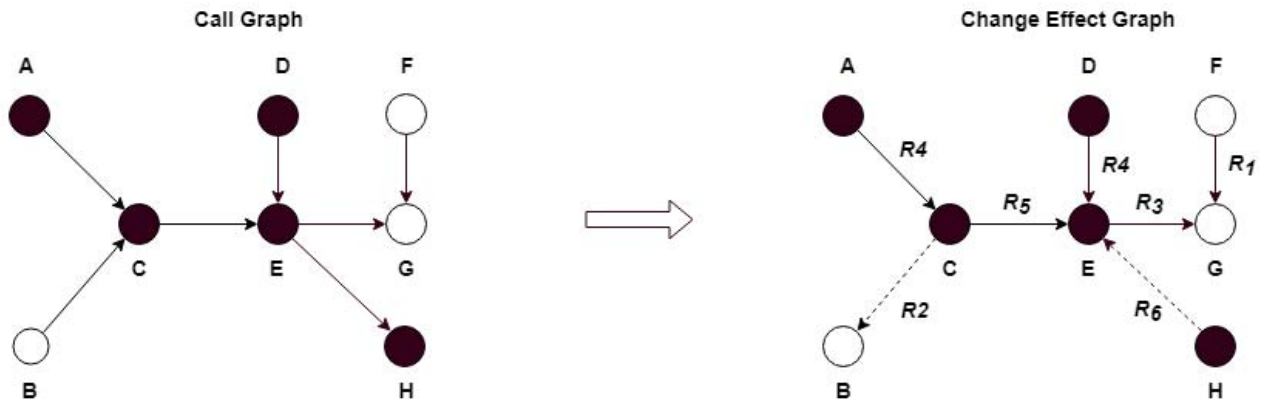

Fig 2. An example of call graph and change effect graph

When the graph layout of final CEG is used for the construction of BN, the next step is to encode the BN nodes, in other words, encoding the NPTs with probabilistic information. For each node in final CEG, *Type1* and *Type2* probability calculations with respect to relationships in Figure 2 are shown in Tables II-IX:

TABLE II. NPT of Method A

	A	
	True	False
	$ChangeRate(A)$	$1 - ChangeRate(A)$

TABLE III. NPT of Method B

C	B	
	True	False
T	$\frac{\lvert SR_B^C \rvert}{\lvert S_B \rvert}$	$1 - \frac{\lvert SR_B^C \rvert}{\lvert S_B \rvert}$
F	$ChangeRate(B)$	$1 - ChangeRate(B)$

TABLE IV. NPT of Method C

A	C	
	True	False
T	$\frac{\lvert SP_C^A \rvert}{\lvert S_C \rvert}$	$1 - \frac{\lvert SP_C^A \rvert}{\lvert S_C \rvert}$
F	$ChangeRate(C)$	$1 - ChangeRate(C)$

TABLE V. NPT of Method D

	D	
	True	False
	$ChangeRate(D)$	$1 - ChangeRate(D)$

TABLE VI. NPT of Method E

C	D	H	E	
			True	False
T	T	T	$\frac{\lvert SP_E^A \cup SP_E^D \cup SR_E^H \rvert}{\lvert S_E \rvert}$	$1 - \frac{\lvert SP_E^A \cup SP_E^D \cup SR_E^H \rvert}{\lvert S_E \rvert}$
T	T	F	$\frac{\lvert SP_E^A \cup SP_E^D \rvert}{\lvert S_E \rvert}$	$1 - \frac{\lvert SP_E^A \cup SP_E^D \rvert}{\lvert S_E \rvert}$
T	F	T	$\frac{\lvert SP_E^A \cup SR_E^H \rvert}{\lvert S_E \rvert}$	$1 - \frac{\lvert SP_E^A \cup SR_E^H \rvert}{\lvert S_E \rvert}$
T	F	F	$\frac{\lvert SP_E^A \rvert}{\lvert S_E \rvert}$	$1 - \frac{\lvert SP_E^A \rvert}{\lvert S_E \rvert}$
F	T	T	$\frac{\lvert SP_E^D \cup SR_E^H \rvert}{\lvert S_E \rvert}$	$1 - \frac{\lvert SP_E^D \cup SR_E^H \rvert}{\lvert S_E \rvert}$
F	T	F	$\frac{\lvert SP_E^D \rvert}{\lvert S_E \rvert}$	$1 - \frac{\lvert SP_E^D \rvert}{\lvert S_E \rvert}$
F	F	T	$\frac{\lvert SR_E^H \rvert}{\lvert S_E \rvert}$	$1 - \frac{\lvert SR_E^H \rvert}{\lvert S_E \rvert}$
F	F	F	$ChangeRate(E)$	$1 - ChangeRate(E)$

TABLE VII. NPT of Method F

	F	
	True	False
	$ChangeRate(F)$	$1 - ChangeRate(F)$

TABLE VIII. NPT of Method G

E	F	G	
		True	False
T	T	$\frac{\lvert SP_G^E \cup SP_G^F \rvert}{\lvert S_G \rvert}$	$1 - \frac{\lvert SP_G^E \cup SP_G^F \rvert}{\lvert S_G \rvert}$
T	F	$\frac{\lvert SP_G^E \rvert}{\lvert S_G \rvert}$	$1 - \frac{\lvert SP_G^E \rvert}{\lvert S_G \rvert}$
F	T	$\frac{\lvert SP_G^F \rvert}{\lvert S_G \rvert}$	$1 - \frac{\lvert SP_G^F \rvert}{\lvert S_G \rvert}$
F	F	$ChangeRate(G)$	$1 - ChangeRate(G)$

TABLE IX. NPT of Method H

	H	
	True	False
	$ChangeRate(H)$	$1 - ChangeRate(H)$

V. Experimental Setup

The experimental setup follows four steps of the proposed method.

1. Call graph extraction
2. Method change information extraction
3. Backward slicing implementation
4. Bayesian Network integration

All these steps are built using Java language. In addition, the open source project that is used for our case study is written in Java language as well.

A. Call Graph Extraction

To extract the call graph of a Java source project, we have used java-callgraph[1], which is developed by Georgios Gousios and available as open source. Java-callgraph, provides both static and dynamic call graphs. In this study, we have only used the static call graph.

Normally, *java-callgraph* only provides an output of the call graph. Since that we need to process on the call graph, we need to store the call graph in a data structure. Therefore, we have modified *java-callgraph* so that we can perform our operations (change edge directions, remove edges) to convert the call graph into a CEG.

B. Method Change Information Extraction

It is important that we find the changed components between two different versions of a project. Thereby, we can have our initial point where are the affected parts of the project. Then, from this point of view we locate the other parts of the project that are affected by these changes. These predictions will then construct our EIS.

To find the differences between two versions we used an open source diff-tool called *reJ*[2]. *reJ*, is a graphical tool that shows the changed lines (added and deleted) on its Java bytecode and does not provide any numerical change information. To automatize our system and to apply our rules that were described in section IV, we have made three major modifications on *reJ*. To view changes between two versions, *reJ* required multiple user interactions. Our first modification was to eliminate all these user interactions and obtain all the changes at once. This modification enabled us to construct an automatized system and save time consumption. Our second modification was, converting *reJ*'s visual change information into numerical information that represents the change rate. The change rate is calculated by dividing the total number of changed bytecode lines to total bytecode line. Third is changing the granularity level of change information. *reJ* provides a change information at class-granularity level. We have modified the granularity level to method-granularity. This modification is necessary because of the aim of this study targets impact analysis on methods.

C. Backward Slicing Implementation

There are few open source tools for Backward slicing that supports Java programming language. One of the popular ones are Indus by Ranganath et al. [19] and Kaveri by Jayaraman et al [20]. These tools are available as open source. However, Kaveri and Indus support Java 4 and the current open source project that we have used in our case study support Java 8 and 9. Therefore, we have implemented our own Backward Slicing.

In Section III we have that Backward Slicing works on and requires CFGs. Kaveri and Indus have used Soot [20],[21] to generate CFGs. In this study, we have used Soot as well.

D. Bayesian Network Integration

To execute the BN, the academic version of GeNIe [23],[24] has been used. GeNIe is a graphical tool of SMILE [24] engine. GeNIe supports multiple commercial and non-commercial BN tools' file formats that include information about the BN. This information includes; node names, directed edges between nodes, node probability table values.

We have implemented a feature that automatically converts the BN we have constructed through our analysis to ".net" file format, which is a BN file format of Hugin[3]. Among other file formats, we have selected ".net" because of simple structure of

[1] *java-callgraph - https://github.com/gousiosg/java-callgraph*
[2] *reJ - http://rejava.sourceforge.net/*
[3] *Hugin - https://www.hugin.com/*

its file format. The file structure is very similar to XML format. Thereby, this allows us to easily implement the BN file and easy to parse these files for later usage.

While analyzing the open source projects, due to their size and large amount of methods, the BNs could be very large and complex. These large and complex BNs are unfortunately an obstacle in terms of memory usage efficiency. However, in the BN there could be disconnected subgraphs. Therefore, for memory efficiency, every disconnected sub-graph is treated separately. First, the disconnected subgraphs are detected. Then, each subgraph is processed. After each subgraph is constructed and its "*.net*" file is generated, then that subgraph is removed from the memory, because the BN is already stored in a file.

VI. Case Study

Previous studies on CIA have selected their open source project from Sourceforge[4] and used CVS[5] for obtaining change history of these open source projects. Unfortunately, CVS is no longer maintained. Therefore, in this study we have used more up to date and popular technologies that are preferred by developers. We select our open source Java project from Github[6] among trending Java projects. It is also possible to extract each version's commit information (change history) and download each version of commit.

We have selected *Java JWT*[7], which is that provides a JSON Web Token for Java and Android. It is a project that has released 10 versions and has 292 commits. We have tried to select the latest version of *Java JWT* with many commits. Thereby, we have performed our case studies for versions v0.6.0 and v0.7.0. In between versions v0.6.0 and v0.7.0 there are 80 commits. Among 80 commits only 32 of them contain Java source code changes. The size for the open source project *Java JWT* is given in Table X.

TABLE X. Java JWT Project Information

	jjwt-0.6.0	**jjwt-0.7.0**
LOC	8999	9445
# of Classes	86	91
# of Methods	505	528
# of Statements	2797	2956

As mentioned before, out of 80 commits only 32 of them had Java source code changes. Therefore, in this study 32 of these commits have been downloaded and analyzed separately. Each of the commit is analyzed in chronological order and each of their *precision*, *recall* and *f-measure* values are calculated. It is expected that, this investigation will provide us a brief explanation about, when is it effective perform CIA after how much of the development has been completed.

VII. Results

To calculate the recall, precision and f-measure of our approach, the AIS is first required. The AIS is a set of changed methods between the current version (*jjwt-0.6.0*) and next version (*jjwt-0.7.0*). The commits are the changes that are made between current version and next version. Therefore, AIS is the ground truth of our study. In between versions *jjwt-0.6.0* and *jjwt-0.7.0*, 49 methods have been changed and newly implemented methods. Depending on the commit, out of the 49 changed methods newly (not implemented yet) methods are removed. The *recall*, *precision* and *f-measure* calculations are calculated by the formulas given below in Equations (3), (4) and (5). In those equations R stands for *recall*, P stands for *precision*, and F stands for *f-measure*.

$$R = \frac{|AIS \cap EIS|}{|AIS|} \tag{3}$$

$$P = \frac{|AIS \cap EIS|}{|EIS|} \tag{4}$$

$$F = 2 \times \frac{(\mathrm{P} \times R)}{P+R} \tag{5}$$

The EIS results contain methods with probabilistic values. Methods that have a very low probability value are the methods that are not affected or least affected. However, the higher probability the method has, the higher it is affected by changes. Therefore, to observe the changes in precision, there are three types of investigations performed. The first is filtering the EIS results by only accepting the methods that have 0.02 higher probabilistic values. The second if filtering the EIS results by only accepting methods that has higher probability value than 0.01. The last is no performing any filtering on the EIS results.

Another investigation that has been performed is manipulating the AIS. Normally the AIS, is the set of changed methods from a version to its next version. However, the drawback of using the set of total changed methods also includes methods that are newly implemented methods. Since it is impossible to predict unimplemented and not defined methods, this causes a serious decrease in recall and precision results. Thereby, to solve this issue, we have removed the methods that are not yet implemented in the currently analyzed commit. Otherwise, without removing the unimplemented methods we would be trying to predict methods that does not exists in the source code and the precision recall values would be misleading.

First, the methods in AIS that are not implemented yet in the current commit under analysis are eliminated. In Figure 3, there are no filtering applied on method probabilities in EIS. Therefore, the recall value on each commit resulted with 1.0. This shows that, every affected method has been detected. However, the maximum precision value that we could get is 0.11, which is very low. This means that, developers have to

[4] *Sourceforge - https://sourceforge.net/*
[5] *CVS - http://www.nongnu.org/cvs/*
[6] *Gitub - https://github.com/*
[7] *Java JWT - https://github.com/jwtk/jjwt*

spend a lot of effort on finding and maintaining the real impacted methods in the software.

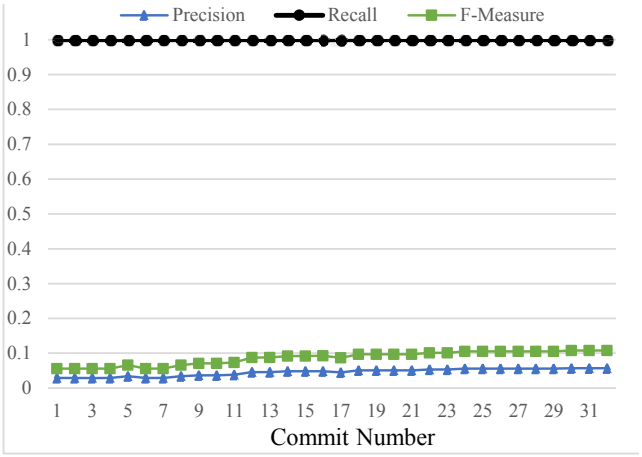

Fig 3. No filtering applied on method probabilities and all methods in EIS are selected.

On the other hand, in Figure 4 when the methods with probability that are lower than 0.01 are removed from EIS, a slightly change on the precision and noticeable change on recall values has been observed. Comparing to Figure 3, the max precision value that has been observed is 0.06, while in Figure 4 the max precision value has increase to 0.19. However, the important jump that we have captured is, at the 12th commit, when 37.5% of the software implementation is complete. Before the project has completed half of its implementation, our approach is able to find all of the affected method. In terms precision, at the 24th commit, when 75% of the software implementation is complete, we have obtained the highes precision value 0.19. Since that all of the affected methods are detected, we have both satisfied the 2nd and 3rd possibilities that Arnold and Bohner [17] have defined, which are "*Safe*".

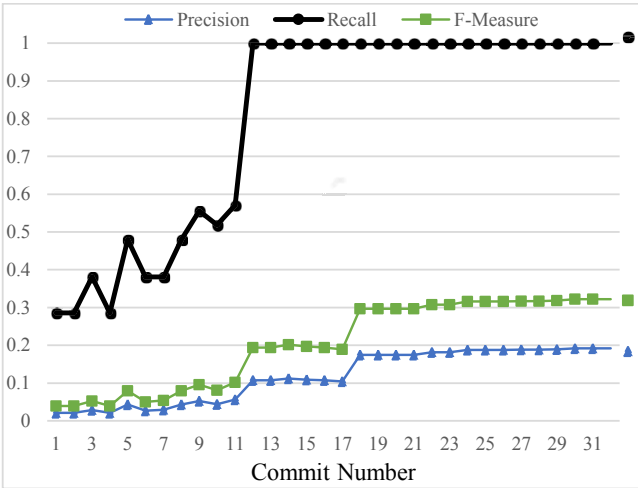

Fig 4. Probabilities under 0.01 are removed from EIS.

In Figure 5, a more selective filtering has been applied. Compared to Figures 3 and 4 a higher precision has been obtained, which is 0.48. However, by using a more selective filtering caused in missing some of the affected methods that has decreased the recall value. While we were able to detect all the affected methods by only accepting the methods that have higher probability than 0.01. Using a filter that accepts the method probabilities that are greater and equal to 0.02 caused by only capturing 88% of the affected methods. This situation corresponds to Arnold and Bohner's [17] 5th possibility of AIS and EIS relationship. The 5th possibility is where all the affected methods are not estimated, and the estimated affected methods have false positives. This possibility is considered as "*Not so good*". It is observed that removing the methods with probability that are lower than 0.01 from EIS is the best parameter choice.

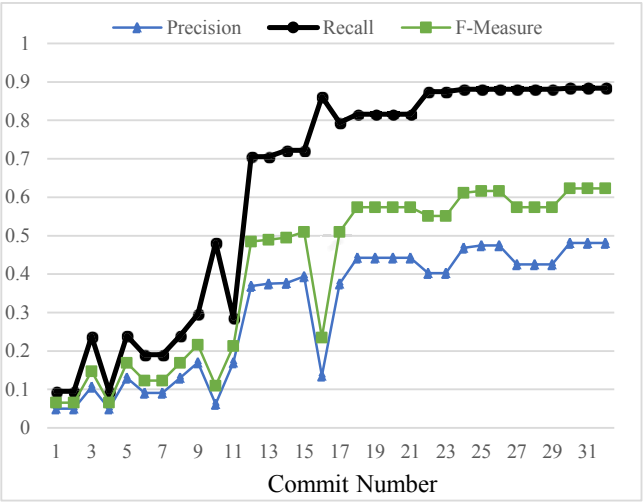

Fig 5. Probabilities under 0.02 are removed from EIS.

VIII. THREATS TO VALIDTY

In this section, we discuss the limitations of our overall experimental design and setting.

We have conducted our research and evaluation on single open source Java project. In addition, the size of the open source project is not large. Therefore, we cannot claim generality for our results. However, the open source project that we have selected is a real project that is frequently used and available on current repository hosting service. Furthermore, the subject project is an up-to-date project with ten versions released. Nevertheless, additional studies with other and larger open source project is required to answer such questions of external validity.

Other limitations involve internal validity. In the basis of our study we used call graphs and change amount of methods to construct CEG, which is built upon six rules. Then, to satisfy the properties of BNs, the edges that cause cycles in the CEG are removed. Recall that edges show dependencies between method and when edges are removed from CEG, we might be losing a valuable dependency information. To reduce the

amount of loss in dependency, we have defined the six rules. However, this might have lead us to low precision results.

In addition, using backward slicing has its own limitations. Using backward slicing is a modular, it performs slicing by on methods' control-flow graphs independently. Thereby, we were only able to focus on method parameters and method returns. However, methods could be affected by global variables as well and our experimental design does not perform its slicing over global variables. This cloud lead to imprecise change impact analysis results.

Briefly, our results support that, using backward slicing and CEGs in BN could produce benefits in change impact analysis, such that all affected methods could be found before half of the implementation is complete. Therefore, the results we have obtained motivates us to perform further studies, followed by a detailed and carefully controlled experimentation, to investigate whether results will generalize.

IX. Conclusion and Future Work

In this study, first a directed graphical model called Change Effect Graph has been proposed. This model is created by using call graph and change information. In the proposed approach, the Change Effect Graph is used in a Bayesian Network and the nodes are encoded with probabilistic values that are obtained from program slicing and change information. The approach is applied to an open source project called *Java JWT*, which is obtained from Github. *Java JWT* has 32 commits between version 0.6.0 and 0.7.0 that included Java source code changes. Each commit is investigated separately to observe at which commit we can detect all to be affected methods with respect to already started changes. The proposed method detects all to be affected methods in 12th commit. Our experimental results have shown that, by using CEG, it is possible to detect all to be affected methods before half of the implementation is complete.

To increase precision results, change types could be investigated deeply. Each different type of change can have different effects on different methods and these changes could be weighted differently. In addition, CEG uses call dependency and data dependency. The data dependency is consisted of parameter passing and method return values. Other types of dependencies could be investigated further. In addition, we are aware to satisfy a BN, removing cycles is one of the drawbacks of our study. To create an acyclic graph, we remove edges, which could remove a valuable dependency information as well. Therefore, we have constructed the six rules to minimize the loss of dependency information. However, in further studies we plan to use Markov chains which allow us to include cycles in graph.

References

[1] A. Kumar, "Development at the Speed and Scale of Google," in *International Software Development Conference (QCon)*, 2010.

[2] F. Norman, N. F. Schneidewind, and S. Member, "The state of software maintenance The State of Software Maintenance," no. 3, pp. 303–310, 1987.

[3] G. Canfora and L. Cerulo, "Impact Analysis by Mining Software and Change Request Repositories," *Proc. IEEE Int. Symp. Softw. Metrics*, no. Metrics, pp. 29–38, 2005.

[4] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Trans. Softw. Eng.*, vol. 30, no. 9, pp. 574–586, 2004.

[5] B. Li, X. Sun, H. Leung, and S. Zhang, "A survey of code-based change impact analysis techniques," *Softw. Testing, Verif. Reliab.*, vol. 23, no. 8, pp. 613–646, Dec. 2013.

[6] L. Badri, M. Badri, and D. St-Yves, "Supporting predictive change impact analysis: a control call graph based technique," in *12th Asia-Pacific Software Engineering Conference (APSEC'05)*, 2005, p. 9 pp.

[7] X. Sun, B. Li, C. Tao, W. Wen, and S. Zhang, "Change Impact Analysis Based on a Taxonomy of Change Types," in *2010 IEEE 34th Annual Computer Software and Applications Conference*, 2010, pp. 373–382.

[8] B. Fluri and H. C. Gall, "Classifying Change Types for Qualifying Change Couplings," in *14th IEEE International Conference on Program Comprehension (ICPC'06)*, 2006, vol. 2006, pp. 35–45.

[9] P. Tonella, "Using a concept lattice of decomposition slices for program understanding and impact analysis," *IEEE Trans. Softw. Eng.*, vol. 29, no. 6, pp. 495–509, 2003.

[10] A. Tang, A. Nicholson, Y. Jin, and J. Han, "Using Bayesian belief networks for change impact analysis in architecture design," *J. Syst. Softw.*, vol. 80, no. 1, pp. 127–148, 2007.

[11] S. Mirarab, A. Hassouna, and L. Tahvildari, "Using Bayesian Belief Networks to Predict Change Propagation in Software Systems," in *15th IEEE International Conference on Program Comprehension (ICPC '07)*, 2007, pp. 177–188.

[12] Xiaoxia Ren, B. G. Ryder, M. Stoerzer, and F. Tip, "Chianti: a change impact analysis tool for Java programs," in *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, 2005, pp. 664–665.

[13] M. Weiser, "Program Slicing," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 352–357, Jul. 1984.

[14] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM SIGPLAN Not.*, vol. 39, no. 4, p. 229, Apr. 2004.

[15] O. Lhoták, "Comparing call graphs," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering - PASTE '07*, 2007, pp. 37–42.

[16] B. G. Ryder, "Constructing the Call Graph of a Program," *IEEE Trans. Softw. Eng.*, vol. SE-5, no. 3, pp. 216–226, May 1979.

[17] R. S. Arnold and S. A. Bohner, "Impact Analysis - Towards a Framework for Comparison," *Proc. Conf. Softw. Maint.*, pp. 292–301, 1993.

[18] V. Ramachandran, "Finding a minimum feedback arc set in reducible flow graphs," *J. Algorithms*, vol. 9, no. 3, pp. 299–313, 1988.

[19] V. P. Ranganath and J. Hatcliff, "Slicing concurrent Java programs using Indus and Kaveri," *Int. J. Softw. Tools Technol. Transf.*, vol. 9, no. 5–6, pp. 489–504, Oct. 2007.

[20] G. Jayaraman, V. P. Ranganath, and J. Hatcliff, "Kaveri: Delivering the Indus Java Program Slicer to Eclipse," in *Fundamental Approaches to Software Engineering*, vol. 3442, 2005, pp. 269–272.

[21] P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge, "A framework for optimizing java using attributes," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2001, vol. 2027, pp. 334–354.

[22] R. Vallée-Rai, P. Co, É. M. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan, "Soot: A Java Bytecode Optimization Framework," *Proc. 2010 Cent. Adv. Stud. Conf.*, pp. 214–224, 2010.

[23] "GeNIe." [Online]. Available: https://www.bayesfusion.com/genie-modeler.

[24] M. J. Druzdzel, "SMILE : Structural Modeling, Inference, and Learning Engine and GeNIe: A Development Environment for Graphical Decision-Theoretic Models," *Proc. Sixt. Natl. Conf. Artif. Intell.*, no. May, pp. 342–343, 1999.