2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC) | 978-1-6654-2463-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/COMPSAC51774.2021.00137

Code Change Sniffer: Predicting Future Code Changes with Markov Chain

Ekincan Ufuktepe
University of Missouri - Columbia
Columbia, MO, USA
euh46@missouri.edu

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

***Abstract*—Code changes are one of the essential processes of software evolution. These changes are performed to fix bugs, improve quality of software, and provide a better user experience. However, such changes made in code could lead to ripple effects that can cause unwanted behavior. To prevent such issues occurring after code changes, code change prediction, change impact analysis techniques are used. The proposed approach uses static call information, forward slicing, and method change information to build a Markov chain, which provides a prediction for code changes in the near future commits. For static call information, we utilized and compared call graph and effect graph. We performed an evaluation on five open-source projects from GitHub that varies between 5K-26K lines of code. To measure the effectiveness of our proposed approach, recall, precision, and f-measure metrics have been used on five open-source projects. The results show that the Markov chain that is based on call graph can have higher precision compared to effect graph. On the other hand, for small number of cases higher recall values are obtained with effect graph compared to call graph. With a Markov chain model based on call graph and effect graph, we can achieve recall values between 98%-100%.**

***Index Terms*—change impact analysis, markov chains, software evolution, change propagation prediction**

I. INTRODUCTION

Code change prediction is an estimation of to be modified code segments in near future. Change prediction plays an important role in reducing the amount of time and resources that are spent on software maintenance. In this study, we propose a novel approach for change prediction based on (i) Markov chain that uses (ii) forward slicing information for calculating the probability of effect after a change, and as the graph structure from which we construct the Markov chain, we utilized (iii) call graph (CG) and effect graph (EG). CG only includes the change propagation/effect direction in one way and disregards the impacts from callee methods via return values. To include the impacts from callee methods, we introduce EG, which is a bi-directional version of CG. Both CG and EG represent the dependencies between methods and how the change could propagate in the software to cause other methods to change.

For change impact analysis, previous studies have focused on file-level [1], [2] or class-level [3], [4] granularities. However, a class or file-level granularity may not precisely point out which software components are impacted or affected. Files and classes may include multiple classes and methods. Assume that there is a class with ten methods, and the class is predicted as impacted or affected, and assume that only one of the ten methods is impacted. There will be a 1/10 chance to find and know which method is actually impacted. Therefore, in this study, we performed our predictions at a finer granularity level, i.e., at method granularity. To work at method-level granularity, we use CG and EG, of which details are discussed in the following sections.

Previous studies have used Bayesian Network (BN) to calculate change propagation [5] and perform change impact analysis [6], [7]. Even though that BNs are powerful probabilistic reasoning systems under uncertainty, they have very high complexity, and lack representing cycles. For instance, a study [6] had to remove cycles (dependencies) to used BN, which might have caused to lose valuable information. Therefore, in our study we used Markov chain, which is commonly used in making predictions. However, the main factor of using Markov chain is that it allows representing and using cycles, and it has a lower complexity.

Program slicing is also used in change impact analysis [6], [8], [9]. Especially, forward slicing is well-known to be used for finding the affected components after a change. Since our goal is to predict future code changes, we use forward slicing to find the affected statements in the control-flow graph. After the affected statements are found, we calculate the percentage of a method being affected by a change.

To measure and evaluate the success of a code change prediction approach, ground-truth data is required. For instance, a study used mutation testing and failed test cases for ground-truth and evaluation [10]. The drawback of using test cases is that test cases applied on real systems may not have 100% coverage all the time. Furthermore, this approach depends on the quality of the test cases, which could be a case where not all the mutants can be eliminated. Another study employed the diff between two consecutive versions as ground-truth and utilized the commits between the two consecutive versions for analysis [6]. In this study, we have used the diff of two consecutive versions on their bytecode, since that there could be change in due to the compiler.

RQ1: Does use of effect graph instead of call graph in predicting future code changes make a difference? How are the precision, recall, and f-measure values affected? We use our approach both on CG and EG and compare their

978-1-6654-2463-9/21/$31.00 ©2021 IEEE
DOI 10.1109/COMPSAC51774.2021.00137

Authorized licensed use limited to: ULAKBIM UASL - IZMIR YUKSEK TEKNOLOJI ENSTITUSU. Downloaded on May 26,2022 at 11:41:18 UTC from IEEE Xplore. Restrictions apply.

effectiveness by their recall, precision, and f-measure values.

RQ2: Does applying a threshold on the effect probabilities of methods improve the recall, precision, and f-measure values? We assume that a changed method can affect all its reachable methods based on a CG and an EG. Since our approach provides probabilistic values for reachable methods, in predicting future code changes, we have the opportunity not to include the methods that are under some probabilistic value. This way, we investigate whether applying a threshold to disregard some of the methods makes a difference in the precision, recall, and f-measure values.

This paper makes the following main contributions:

(i) Method: We present an end-to-end pipeline for predicting future code changes that might occur in a software evolution process. For our prediction system, our method uses Markov chain. Then, with respect to the relationships between methods (CG and EG), the transition matrix of the Markov chain is filled with probabilistic information that is obtained from forward slicing. The initial vector uses the calculated diff ratio for each method. After the multiplication between the initial vector and transition matrix the impact vector is calculated, in other words the predicted methods that will change in future.

(ii) Tool: We developed a tool called Code Change Sniffer (CCS)[1] and made it publicly available for predicting future code changes. The tool implements the end-to-end pipeline for predicting future code changes with Markov chain.

(iii) Dataset: We provide our evaluation dataset to other researcher and practitioners to provide a better reproducibility of our study. Our dataset contains data for each studied project that includes ground-truth data, predicted changed methods with probabilistic values. Furthermore, the type of changes for each commit are tagged by types given in [11].

II. Proposed Approach

In this section, we describe our proposed approach, and how the change and program slicing information are utilized.

A. Change Information

Our approach is based on an assumption that to predict the method(s) that will change in future commits, it is necessary to know which method(s) has already been changed in the past and current commits. The change information is obtained by using diff with a tool called reJ. It is a code diff visualization tool that shows the code changes in bytecode at class-level granularity. Obtaining the changed code in bytecode is very important, since different styles of coding can be represented in different sizes of LOC (Lines of Code) and visually can be interpreted as a change.

Since our proposed approach is at method-level granularity, we modified *reJ* to provide change information at method-level granularity. To extract the change information and use it in Markov chain, we calculate the percentage of change performed on each method as given in Equation 1, where M is a set of changed methods that exists in a software, where cm_i is a changed method and an element of set M. The n is the total number of changed methods between two commits. In other words, n is the size of set M. The $P(cm_i)$ represents the probability of method cm_i being affected by the changes that are made internally.

$$cm_i \in M, 0 < i \leq n$$
$$P(cm_i) = \frac{\textit{Total num. of changed stmts}}{\textit{Total Num. of changes and unchanged stmts}} \quad (1)$$

The information extracted through this process is assigned to the initial (impact) vector. The change information is also used in transition matrix of CG, where there is a self-loop represented in change prediction model.

B. Effect Probability Information

Forward slicing information is used for methods that has call relationship, which can affect each other through parameter passing or method returns. Therefore, we apply forward slicing on two types of variables for each method: method parameters and method returns. The forward slicing on the method parameter represents the change flow from caller method to the callee method. The forward slicing applied on the return value represents the change flow from callee method to the caller method. The forward slicing is used on the CFG of methods. Thereby, we calculate the probability of change based on the sliced CFG.

The forward slicing applied on the method parameters has a straightforward approach. Since the method parameters are defined in the first statements of the CFG, the forward slicing starts from the first statement to the last statements (leaf nodes). After the sliced CFG is obtained, we calculate the probability of the possible change that could occur in the method. The probability is calculated by dividing the remaining statements in CFG after slicing to the total number of statements that exist in the CFG before slicing. We recall that each method has its own CFG, which is consisted of statements. Thereby, let CFG_{m_i} be a set of statements of method m_i and let $stm_{m_{ik}}$ be the statements in the CFG. Then, in Equation 2, let $pCFG_{m_i}$ be the parameter-based sliced CFG of CFG_{m_i}, which is also a subset of CFG_{m_i}. The calculation of change probability for parameter-based forward slicing is given in Equation 3. For methods that does not have any parameters, the probability of change is set to 0.

$$CFG_(m_i) = \{stm_{m_{ik}} : 0 < k \leq |CFG_{m_i}|, m_i \in M\}$$
$$\textit{where } pCFG_{m_i} \subseteq CFG_{m_i} \quad (2)$$

$$P(m_i) = \frac{|pCFG_{m_i}|}{|CFG_{m_i}|} \quad (3)$$

For the return-based forward slicing, we require the callee methods and where the method is called in the CFG. The next information of where callee methods are called in the CFG, requires a parsing process, to locate the statements (nodes) in the CFG. After where the callee methods are called in

[1]https://github.com/ekincanufuktepe/code-change-sniffer

Authorized licensed use limited to: ULAKBIM UASL - IZMIR YUKSEK TEKNOLOJI ENSTITUSU. Downloaded on May 26,2022 at 11:41:18 UTC from IEEE Xplore. Restrictions apply.

the CFG are located, we use located statements in the slicing criterion for each callee method separately. For instance, if a caller method calls three different methods, the forward slicing runs once for each callee method, which makes three forward slicing executions in total.

We formalize the calculated probabilities for each callee method's effect on the caller method as follows. Let m_i be the caller method and m_j the callee method. Then, let $rCFG_{m_{i_{m_j}}}$ be the return-based sliced CFG of method m_i, which is also a subset of CFG_{m_i}. To calculate the change probability from the possibility effect of return values from callee methods, we divide the remaining statements in the CFG after the return-based forward slicing by the total number of statements before slicing. The probability calculation is given in Equation 4. In some cases, the callee method may not have a return type (i.e., void methods). In such cases, the CFG is not be assigned to a variable or return a value. Therefore, since there is no assignment the forward slicing algorithm will not proceed, and the probability will be set to 0.

$$\begin{gathered} where\ rCFG_{m_{i_{m_j}}} \subseteq CFG_{m_i} \\ P(m_{i_{m_j}}) = \frac{|rCFG_{m_{i_{m_j}}}|}{|CFG_{m_i}|} \end{gathered} \tag{4}$$

After all the slicing information is collected and probabilities are calculated, we assign all the probabilities to the transition matrix. For the CG, all the edge directions are from caller method to callee method, which means the effect direction is from caller method to callee method Therefore, only necessary information that is assigned to the transition matrix will be the probabilities that are calculated from parameter-based forward slicing information. On the other hand, EG contain possible change effects from caller to callee method and callee method to caller method. Thereby, both parameter-based and return-based forward slicing information will be assigned to the transition matrix.

C. Proposed Code Change Prediction Method

In Fig. 1, we provide the architecture and pipeline of our code change prediction method. Our approach first starts by extracting the CG of one of the commits after the Nth version (within $(N+1)^{th}$ version) and finding the code changes between the Nth and within $(N+1)^{th}$ versions. Then, we generate CFGs for each method in and perform a forward slicing on the commit within $(N+1)^{th}$ version. After the necessary information is collected, we create two alternative code change prediction models based on Markov chain, namely CG and EG. To create an EG, we utilize already extracted CG. When both Markov chain models are constructed, we create the transition matrix, which is also similar to an adjacency matrix, only that the connected methods (nodes) are encoded with probabilistic information (Equations 2-4).

The transition matrix for CG only uses probabilistic information from Equation 2, since the CG only contains effect from caller to callee via parameters. The transition matrix for EG uses probabilistic information from Equations 2-4, since that EG includes change affects from caller to callee, and from callee to caller (return methods).

Finally, we set the initial (impact) vector based on probabilistic information obtained from Equation 2 and multiply the initial vector with the transition matrix. Since that the multiplication is between a vector and a square matrix, the complexity is $O(n^2)$. The output of the multiplication provides a vector output, which represents a list of code change probabilities, with respect to the changes that are made in the software.

III. Evaluation

In this section, we introduce the open-source projects we selected for evaluation and provide the results for our approach. We first mention how we select the version control system and then, describe how we extracted the commits for each project. Finally, we describe the metrics that are used for evaluation and discuss the results obtained.

A. Open Source Project Selection and Evaluation Metrics

We selected five popular open-source Java projects from GitHub, with different project sizes: commons-codec, commons-csv, JJWT, Jsoup, and JUnit4. We chose consecutive two versions of each selected project, where there is a significant amount of commits in between two consecutive versions with significant amount of source code changes.

Table I provides details of the selected open-source projects. The fourth column in Table I gives the total number of commits between two consecutive versions of each project. The data in the fifth column represents the total number of commits that contain source code changes among the commits given in the fourth column. The fifth column data is collected manually on GitHub. However, for JUnit4 we have selected the commits that merged the source code changes, hence the changes were made independently that caused next commits to omit the previous commits. Since the other projects were not that big, merge was not used, and every commit included previous commits. After all the source code change included commits are determined, these commits are downloaded based on the hash value of the commit and analyzed. The fifth column also represents the number of analysis we performed for evaluation of our approach.

TABLE I: Selected open-source projects

Project	LOC	Analyzed versions	Total commits	Commit w/ Code Changes
commons-codec	23K	1.14–1.15	91	34
commons-csv	7K	1.7-1.8	82	34
JJWT	5K	0.6.0-0.7.0	80	31
Jsoup	18K	1.10.3–1.11.1	92	70
JUnit4	26K	r4.11–r4.12	700	54

There are three commonly used metrics in evaluating the success of prediction results. These metrics are Recall, Precision, and F-Measure. Recall is a metric used to quantify the number of positive class predictions made from all positive examples in the dataset. On the other hand, Precision is a metric used to quantify the number of positive predictions

Authorized licensed use limited to: ULAKBIM UASL - IZMIR YUKSEK TEKNOLOJI ENSTITUSU. Downloaded on May 26,2022 at 11:41:18 UTC from IEEE Xplore. Restrictions apply.

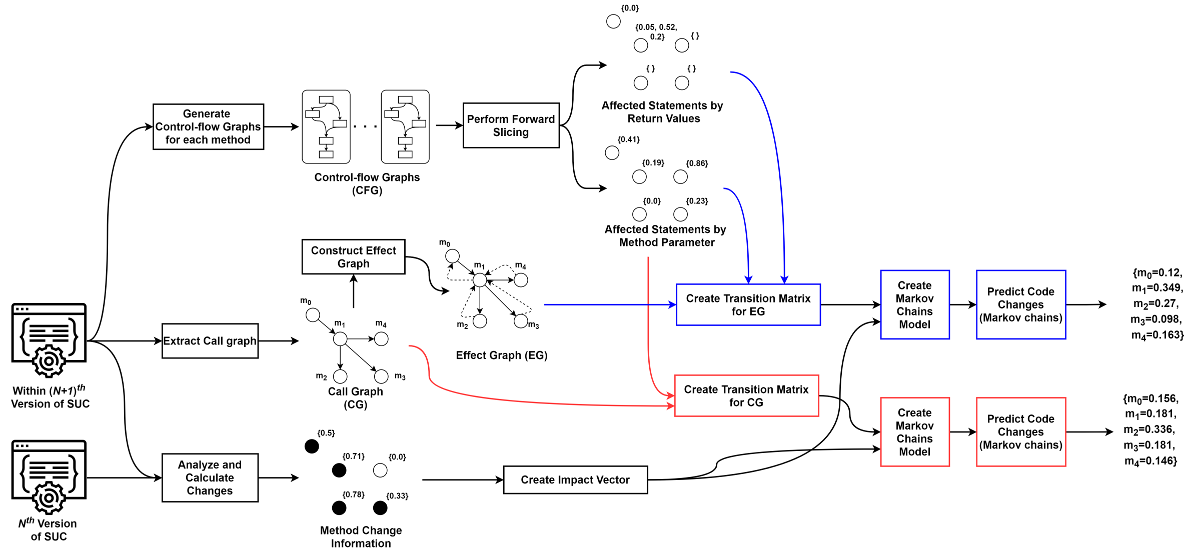

Fig. 1: Code Change Sniffer: future code change prediction pipeline

that actually belong to the positive class. Finally, we used F-Measure for balancing the concerns of recall and precision in a single numeric representation. However, according to a study, having a higher recall value is always more desired than Precision [12]. The reason is, even if a 100% precision is achieved, but with a low recall value, then this means that an impacted method is missed. A missed impacted method is not tolerable in terms of the cost of maintenance.

B. Results and Discussion

Table II presents the mean and standard deviation values of precision, recall, f-measure for both CG-based (MC_CG), and EG-based (MC_EG) Markov chain code change prediction techniques. To answer our research question RQ1, we have compared two Markov chain-based code change prediction techniques. The mean and standard deviation values are calculated based on the precision, recall, and f-measure for each commit of all five projects. Furthermore, to answer our research question RQ2, we performed an empirical study using three different thresholds by only accepting the higher probability than the threshold, which is also given in Table II. The highest mean values in Table II are shown in bold font.

Table II shows that, in terms of precision, using CG as Markov chain model performs mostly better than using EG. On the other hand, using EG as a Markov chain model mostly performed better than CG since it also includes the change propagation that might occur from callee (return).

Here, we describe our observations and answer our research questions given in Section I.

RQ1: Does use of effect graph instead of call graph in predicting future code changes make a difference? How are the precision, recall, and f-measure values affected? To observe the differences between two different graphical models of change effect relationship, we used CG and EG on five different open-source Java projects with three different thresholds. As seen in Table II, there is a result for each one of the 90 (5 projects, 2 models, 3 thresholds, 3 metrics) cases. Out of the 90 cases, 45 of them belong to the model EG, and while we compare the 45 cases of EG with the other 45 cases of CG, using model EG has performed better on 14 out of the 45 cases, and performed equal with CG on 3 of the cases.

According to the results in Table II, where the cases that EG did not perform well than CG has a very small difference between EG, except for project JUnit4, where a threshold 0.2 is used. JUnit4 is the largest project we have studied, and it contains too many changes. The applied threshold 0.2 is high for JUnit4, which discards the actual changes that are even found by diff and causes the recall to decrease. For instance, changes that are made in the method that are below the 0.2 ratio will be discarded.

Using the CG model provides a better precision result compared to using the EG. Comparing the two models with respect to the f-measure, we can say that CG has performed better than EG except for the project JJWT. However, we have observed different development style on the project JJWT compared to the other projects. This outcome can be interpreted that the JJWT project has been developed in way that the changes made in the callee methods have affected the caller methods. In other words, the change affect is not only limited by the flow of calling methods, but also affected by the returning values from callee methods, which can be observed by the precision, recall and f-measure results. Thereby, with respect to the results, the usage of the model may depend on how the software has been implemented, to make such generalization we require an extensive case study with a larger set of projects. If a software is implemented in a way that relies on returning values, using EG might be a better option than using CG. However, if the implementation does not rely on the

Authorized licensed use limited to: ULAKBIM UASL - IZMIR YUKSEK TEKNOLOJI ENSTITUSU. Downloaded on May 26,2022 at 11:41:18 UTC from IEEE Xplore. Restrictions apply.

TABLE II: Mean and Standard Deviation values of Precision, Recall and F-Measure for CG and EG Markov Chain Code Prediction Techniques on each Project with Different Thresholds

	Technique	commons-codec	commons-csv	JJWT	Jsoup	JUnit4
Threshold=0.0	MC_CG_Precision	**0.1215±0.0185**	**0.3742±0.2571**	0.5750±0.2827	**0.5319±0.1326**	**0.9541±0.0352**
	MC_EG_Precision	0.0927±0.0155	0.0930±0.0762	**0.6158±0.2549**	0.4231±0.1116	0.7809±0.0333
	MC_CG_Recall	**1.0000±0.0000**	**1.0000±0.0000**	**1.0000±0.0000**	**0.9322±0.1276**	**1.0000±0.0000**
	MC_EG_Recall	**1.0000±0.0000**	**1.0000±0.0000**	**1.0000±0.0000**	0.9270±0.1252	**1.0000±0.0000**
	MC_CG_F-Measure	**0.2162±0.0300**	**0.4919±0.2817**	0.6922±0.2131	**0.6663±0.1373**	**0.9762±0.0183**
	MC_EG_F-Measure	0.1692±0.0264	0.1613±0.1279	**0.7335±0.1829**	0.5697±0.1126	0.8766±0.0208
Threshold=0.1	MC_CG_Precision	**0.1378±0.0234**	**0.3366±0.1852**	0.5681±0.2874	**0.5510±0.1305**	**0.9565±0.0351**
	MC_EG_Precision	**0.1378±0.0248**	0.2224±0.1575	**0.6102±0.2585**	0.5022±0.1298	0.9357±0.0354
	MC_CG_Recall	0.9531±0.0281	0.7486±0.2523	**0.9710±0.0341**	0.8566±0.1383	0.9326±0.0101
	MC_EG_Recall	**0.9814±0.0168**	**0.8189±0.1828**	0.9655±0.0218	**0.8707±0.1482**	**0.9331±0.0096**
	MC_CG_F-Measure	0.2402±0.0372	**0.3873±0.1231**	0.6804±0.2209	**0.6619±0.1361**	**0.9439±0.0135**
	MC_EG_F-Measure	**0.2407±0.0390**	0.2998±0.1766	**0.7176±0.1798**	0.6267±0.1352	0.9339±0.0140
Threshold=0.2	MC_CG_Precision	**0.1334±0.0333**	**0.3294±0.1655**	0.5251±0.3123	**0.6250±0.1277**	**0.9564±0.0366**
	MC_EG_Precision	0.1308±0.0401	0.2294±0.1545	**0.5689±0.2825**	0.5522±0.1321	0.9409±0.0405
	MC_CG_Recall	0.6944±0.0963	0.6998±0.3005	0.6784±0.0908	**0.7424±0.1380**	**0.8610±0.0163**
	MC_EG_Recall	**0.7027±0.1326**	**0.7714±0.2290**	**0.6984±0.1059**	0.7352±0.1483	0.8213±0.0416
	MC_CG_F-Measure	**0.2233±0.0523**	**0.3632±0.0798**	0.5242±0.1376	**0.6722±0.1321**	**0.9055±0.0109**
	MC_EG_F-Measure	0.2199±0.0640	0.2953±0.1540	**0.5689±0.1034**	0.6223±0.1344	0.8752±0.0120

return values, but relies on the parameters that are passed, then there is a higher likelihood to achieve better recall, precision, and f-measure results with by using CG.

RQ2: Does applying a threshold on the effect probabilities of methods improve the recall, precision, and f-measure values? We obtained low precision result for two projects; namely commons-codec, where the precision ranged between 0.05-0.18 for the complete software evolution process from one version to another, and commons-csv, where the precision has ranged between 0.017-0.17 for the first half of the evolution process. Therefore, we observe the changes in f-measure, recall, and precision measures to see if it can be improved by applying a threshold. We used three thresholds: 0, 0.1, and 0.2. The thresholds are used for discarding methods that has a probability that are equal or less than the specified threshold. We observed that applying thresholds 0.1 and 0.2 decreased or did not affect improved the precision, recall, and f-measure results for CG model. Therefore, using the CG model with a threshold above 0 may not improve the effectiveness of precision, recall and f-measure for change prediction that much. On the other hand, we used the same thresholds for with the EG model, and we observed improvements especially on the precision, but decrease on the recall values. However, since that both recall and precision measurements represent different aspects, we observed if there is an actual improvement in overall, by observing the changes in f-measure. For the f-measure, our observations found that, among the three studied thresholds, 0.1 has performed the best with EG. However, we do not have sufficient information if EG would perform with another threshold. Therefore, we used an adaptive threshold calculation that used the median of the dataset of calculated method change effect probabilities. The median for each project was calculated 0, which is exactly one of the same thresholds we used in our evaluation. To find the global optima that works efficiently with EG requires further study on a larger set of projects.

Overall, using thresholds 0.1 and 0.2 did not improved the CG model results. However, using a 0.1 threshold has improved the EG model results in predicting code changes. While comparing the effectiveness of the code change prediction results of the models CG and EG, we have observed that the way how the software is developed may influence results. For instance, based on the f-measure, using the EG on project JJWT is found effective than CG, while CG is found more effective in code change prediction for the remaining four projects. However, for both models, using two of the models, we can detect 98%-100% of the future code changes after half of the software evolution process is complete.

IV. Threats to Validity

In this section, we discuss the limitations of our overall evaluation that involves our external and internal validity.

External Threats: In our evaluation, where we evaluated recall, precision, and f-measure, we obtained similar graph results for recall. However, we could not make any generalization on the precision results. This is because of the variety of results we obtained for the five projects studied on. For instance, precision results for commons-codec are ranging between 0.07 and 0.14, while ranging between 0.77 and 0.89 for JUnit4. We believe that the types of changes that are made in the projects influences the precision results. However, to make such conclusion, further experiments on multiple projects are required with detailed investigation on the method change types, which we have initiated and included the change types for each commit and shared the data publicly.

Internal Threats: Our other limitation involves internal validity. We encode the Markov chain with probabilistic information that is obtained from forward slicing. We used forward slicing on CFG, that we used focuses only on the method parameters and return values. This approach disregards the changes made on the class attributes and global variables. The changed class attributes could be changed by a method, and these changed attributes could be used in crucial parts of another method that could affect the method. This study

Authorized licensed use limited to: ULAKBIM UASL - IZMIR YUKSEK TEKNOLOJI ENSTITUSU. Downloaded on May 26,2022 at 11:41:18 UTC from IEEE Xplore. Restrictions apply.

does not take into consideration these types of changes. To consider such types of changes for prediction, rather than CFGs, program dependence graphs (PDGs) could be more useful. However, PDGs are huge and complex graphs. Applying forward slicing and performing a Markov chain prediction will increase the complexity.

V. Related Work

Several approaches attempted to predict code changes by mining software repositories on concurrent versions system (CVS) logs. For instance, Zimmermann et al. [13] used CVS logs to detect evolutionary coupling between fine-grained source code entities such as functions or variables, where evolutionary coupling was the implicit relationship between at least two software entities that are frequently changed together [14]. They used association rules on an itemset of change sets to predict code changes. Before evolutionary coupling, there has been studies on detecting logical coupling and change patterns [15] and German et al. [16] studied the characteristics of the change types. Hassan and Holt [17] analyzed CVS logs by using heuristics to predict software changes.

Some studies focused on using probabilistic approaches for predicting software code changes. Tang et al. [7] proposed a change impact analysis technique that is based on architectural design decisions and elements. They have used BN to quantify the relationships and elements of the design. On the other hand, Abdi et al. [18] proposed a change impact analysis technique that estimates the overall impact of an object-oriented system by using BNs. They used BN by using coupling metrics that refer to two other metrics Design and Implementation. Then, based on the Design and Implementation they quantify the impact on the software. Mirarab et al. [5] used BNs in a different approach, by using a dependency history model. They used change history from CVS and dependency metrics, which they collected with static analysis.

VI. Conclusion

In this study, we proposed a code change prediction approach with a pipeline based on Markov chain that uses forward slicing, CG, and method change information. We have used two models call graph (CG) and effect graph (EG) to compare their effectiveness. We have also shared publicly the code of our approach and pipeline, and the dataset. The effectiveness of each model has been compared by their recall, precision, and f-measure results.

In our evaluation, we investigated five open-source Java projects. Each project had different number of commits between two consecutive versions, with different types of changes. For each commit, the type of changes was tagged by the change types. We used three different thresholds on two models, to observe the changes in the precision, recall, and f-measure results. Our observations shown that the selection of model depends on how the software has been implemented. If the software is implemented in a way where methods mostly rely on the returns of callee methods, then the model EG should be selected. However, if the methods mostly rely on the parameters that are passed, then the model CG should be selected. In terms of using a threshold, CG is not affected by applying a threshold. On the other hand, the model EG produces more data. Therefore, using a threshold for EG increases the precision, but decreases the recall value. Regarding the studied three thresholds, based on the f-measure results, using 0.1 is the best threshold for EG. Our results have also shown that after 50% of the software evolution process from one version to another is complete, both EG and CG models can detect 98%-100% of the changed methods. Our future plans to investigate the type of changes that are made in the code, to understand and observe their impacts to enhance our approach, achieve higher effectiveness.

References

[1] G. Canfora, M. Ceccarelli, L. Cerulo, and M. Di Penta, "Using multivariate time series and association rules to detect logical change coupling: An empirical study," in *IEEE Int. Conf. on Softw. Maintenance*, pp. 1–10, 2010.

[2] M. Ceccarelli, L. Cerulo, G. Canfora, and M. Di Penta, "An eclectic approach for change impact analysis," in *ACM/IEEE Int. Conf. on Softw. Eng.*, vol. 2, pp. 163–166, 2010.

[3] L. C. Briand, J. Wust, and H. Lounis, "Using coupling measurement for impact analysis in object-oriented systems," in *IEEE Int. Conf. on Softw. Maintenance*, pp. 475–482, 1999.

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

[5] S. Mirarab, A. Hassouna, and L. Tahvildari, "Using bayesian belief networks to predict change propagation in software systems," in *IEEE Int. Conf. on Program Comprehension*, pp. 177–188, 2007.

[6] E. Ufuktepe and T. Tuglular, "A program slicing-based bayesian network model for change impact analysis," in *IEEE Int. Conf. on Softw. Quality, Reliability and Security*, pp. 490–499, 2018.

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

[8] E. Alves, M. Gligoric, V. Jagannath, and M. d'Amorim, "Fault-localization using dynamic slicing and change impact analysis," in *IEEE/ACM Int. Conf. on Automated Softw. Eng.*, pp. 520–523, 2011.

[9] M. Acharya and B. Robinson, "Practical change impact analysis based on static program slicing for industrial software systems," in *IEEE Int. Conf. on Softw. Eng.*, pp. 746–755, 2011.

[10] V. Musco, M. Monperrus, and P. Preux, "A large-scale study of call graph-based impact prediction using mutation testing," *Softw. Quality Journal*, vol. 25, no. 3, pp. 921–950, 2017.

[11] X. Sun, B. Li, C. Tao, W. Wen, and S. Zhang, "Change impact analysis based on a taxonomy of change types," in *IEEE Annual Computer Softw. and Applications Conf.*, pp. 373–382, 2010.

[12] R. S. Arnold and S. A. Bohner, "Impact analysis-towards a framework for comparison," in *IEEE Conf. on Softw. Maintenance*, pp. 292–301, 1993.

[13] 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, 2005.

[14] S. Kirbas, B. Caglayan, T. Hall, S. Counsell, D. Bowes, A. Sen, and A. Bener, "The relationship between evolutionary coupling and defects in large industrial software," *Journal of Softw.: Evolution and Process*, vol. 29, no. 4, p. e1842, 2017.

[15] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *IEEE Int. Conf. on Softw. Maintenance (Cat. No. 98CB36272)*, pp. 190–198, 1998.

[16] D. M. German, "An empirical study of fine-grained software modifications," *Empirical Softw. Eng.*, vol. 11, no. 3, pp. 369–393, 2006.

[17] A. E. Hassan and R. C. Holt, "Predicting change propagation in software systems," in *IEEE Int. Conf. on Softw. Maintenance*, pp. 284–293, 2004.

[18] M. Abdi, H. Lounis, and H. Sahraoui, "Predicting change impact in object-oriented applications with bayesian networks," in *IEEE Int. Computer Softw. and Applications Conf.*, vol. 1, pp. 234–239, 2009.

Authorized licensed use limited to: ULAKBIM UASL - IZMIR YUKSEK TEKNOLOJI ENSTITUSU. Downloaded on May 26,2022 at 11:41:18 UTC from IEEE Xplore. Restrictions apply.