IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 15, 2019, accepted May 8, 2019, date of publication May 15, 2019, date of current version May 28, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2916615

Using Tri-Relation Networks for Effective Software Fault-Proneness Prediction

YIHAO LI[1], W. ERIC WONG[2,3], SHOU-YU LEE[3], AND FRANZ WOTAWA[1]
[1]Institute of Software Technology, Graz University of Technology, 8010 Graz, Austria
[2]Shanghai Business School, Shanghai 201400, China
[3]Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75080, USA

Corresponding author: W. Eric Wong (ewong@utdallas.edu)

ABSTRACT Software modules and developers are two core elements during the process of software development. Previous studies have shown that analyzing relations between 1) software modules; (2) developers; and (3) modules and developers, is critical to understand how they interact with each other, which ultimately affects software quality. Specifically, relations such as developer contribution relation, module dependency relation, and developer collaboration relation have been used independently or in pairs to build networks for software fault-proneness prediction. However, none of them investigate the joint effort of these three relations. Bearing this in mind, in this paper, we propose a tri-relation network, a weighted network that integrates developer contribution, module dependency, and developer collaboration relations to study their combined impact on software quality. Four network node centrality metrics are further derived from the proposed network to predict the fault-proneness of a given software module at the file level. Moreover, we have explored a mechanism to refine current networks in order to further improve the effectiveness of software fault-proneness prediction.

INDEX TERMS Tri-relation network, developer contribution relation, module dependency relation, developer collaboration relation, network node centrality metrics, software fault-proneness prediction.

I. INTRODUCTION

Software failures are becoming increasingly costly: a study by the National Institute of Standards and Technology reports that the annual cost of software bugs in the U.S. is about $59.5 billion [66]. Although software fault localization techniques [74]–[77], [80] are becoming more comprehensive, it is still expensive to precisely locate, let alone fix, bugs[1] in a program. Moreover, software bugs may cause severe impacts on system failure [82]. As a result, we may apply fault-proneness prediction beforehand to alleviate the cost of program debugging [8], [12], [19], [32]–[35], [41], [53]–[58], [64], [73], [78], [79], [81]–[84].

Meanwhile, social network analysis has been frequently applied in software engineering. Ghosh [24] reports that many open source projects at SourceForge are organized as social networks. Xu *et al.* [85] classify people working an open source project at SourceForge into project leader, core developer, co-developer, and active user. Ohira *et al.* [52] apply social network analysis and collaborative filtering to identify experts across different projects. Howison *et al.* [30] also use data collected from SourceForge to investigate how the social structures in projects are changing.

With regard to software quality control, study conducted by Cataldo *et al.* [16] indicates that logical dependency (i.e., two files are modified in the same commit) is a more accurate representation of product dependency affecting the development effort and it also explains most of the variance in fault-proneness [17]. Bird *et al.* [12] and Meneely [46] point out that ownership (e.g., a developer contributes a commit on a software module) can have a strong relationship to software defects. Ell [21] and Simpson [64] use the Failure Index (FI) to determine the failure-inducing possibility of developer pairs in developer social networks. In general, these studies characterize software quality from a specific relation between either developers, modules, or developers and modules.

In this paper, [2] three types of relations during the process of software development are investigated: the developer contribution relation (who works on which software modules), the module dependency relation (which modules are dependent on others), and the developer collaboration relation

The associate editor coordinating the review of this manuscript and approving it for publication was Hui Liu.

[1]In this paper, we use "bugs" and "faults" interchangeably. We also use the terms "software" and "program" interchangeably.

[2]This paper is based on Chapters 3, 4, and 5 of Li's dissertation [42].

(which developers work together on the same modules). These relations have been used independently or in pairs in social network analysis to construct different networks to predict which modules are likely to contain faults at different levels such as developer contribution network (DCN) [59], module dependency network (MDN) [87], socio-technical network (STN) [11], [62], and developer collaboration network (DN) [46]. Encouraging results in prior research indicate that software modules that play key roles and are central in these networks tend to be more fault-prone than modules in the surrounding areas of the network [11], [46], [50], [59], [87]. Although these networks are useful for fault-proneness prediction, they are built either by a single relation or by a pair of relations mentioned above. We therefore propose the Tri-Relation Network (TRN), a weighted social network that integrates all three types of relations. Four network node centrality metrics are correspondingly derived from TRN. The design of TRN not only merges the features of DCN, MDN, STN, and DN, but also includes additional relationship (i.e., logical dependency). Moreover, a calibration mechanism based on developer quality [40] for edge weights on TRN and other four networks is explored for further enhancement as well. After all, it is developers who make mistakes and introduce faults into software. Case studies are conducted on six software projects to evaluate the effectiveness of TRN-based metrics in predicting software fault-proneness. In our study, we answer the following three research questions, which are thoroughly discussed later in Section V.

- R1 Are centrality metrics derived from TRN important indicators for the number of post-released bugs in a file?[3]
- R2 Do centrality metrics derived from TRN effectively improve software fault-proneness prediction models?
- R3 Will the fault-proneness prediction effectiveness be improved if applying the proposed edge calibration mechanism on TRN and other four networks.

The remainder of the paper is structured as follows. Section II presents related work whence the proposed TRN arises. Section III explains the proposed TRN. Four network node centrality metrics used in our study and ten software metrics that are commonly used for predicting fault-proneness are introduced in Section IV. Our case studies are detailed in Section V. Section VI discusses some threats to the validity of our study. Finally, our conclusions and plans for future work appear in Section VII.

II. RELATED WORK

The TRN arises from four existing types of networks that have been developed for predicting software fault-proneness: the developer contribution network (DCN), the module dependency network (MDN), the socio-technical network (STN), and the developer collaboration network (DN).

[3]Although we use the term "software module" in previous paragraphs, we want to emphasize that a software module is a generic term to represent a unit of a software system. It can have different representations depending on how a software system is described on a particular architectural level. For example, it can represent a single function, a single class, or a single file. In our case studies, a software module refers to a file.

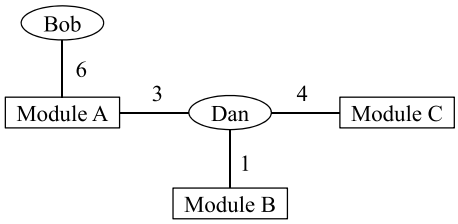


FIGURE 1. A DCN with 2 developers and 3 software modules.

A. DEVELOPER CONTRIBUTION NETWORK (DCN)

In [59], Pinzger et al. represent developer contributions with a developer-module network that is called a contribution network. Case studies based on data collected from Windows Vista indicate that centrality metrics derived from the contribution network are good indicators of the number of post-release faults.

A contribution network is an undirected graph G that is formally defined as $G = (D, N, E)$. D and N are the two sets of vertices, and E is a set of edges between vertices $E \subseteq \{(d, n)|\ d \in D \bigwedge n \in N\}$. D represents the set of developers and N the set of software modules. An edge $e \in E$ denotes a contribution of a developer $d \in D$ to a module $n \in N$. A contribution refers to a commit of a developer to a module. Edges are always between a developer and a module, and there are no self-loops (i.e., neither modules nor developers can contribute to themselves). Edge weights are used to denote the number of commits a developer has made to a module. Figure 1 depicts a sample developer contribution network. Circles represent developers, rectangles represent software modules, and edges represent developer contributions to modules. For example, developer *Bob* has made 6 commits to module A. Developer *Dan* has made 3, 1, and 4 commits to Modules A, B, and C, respectively.

B. MODULE DEPENDENCY NETWORK (MDN)

Zimmermann and Nagappan [87] construct a network from dependency information for software modules in Windows Server 2003. They also find that social network analysis-based metrics derived from the dependency network are good indicators of the number of post-release faults and module fault-proneness, which is consistent with the results presented in [50] and [59]. Generally, a dependency network models the dependency relationships (e.g., call graphs, class inheritance, class coupling, etc.) between software modules within a software system. It is a directed graph that is formally defined as $G = (N, E)$ where N is the set of software modules and E is the set of directed edges such that $(n_1, n_2) \in E$ if Module n_1 has a dependency on Module n_2. Figure 2 shows a simple dependency network where rectangles represent software modules and directed edges represent module dependency relationships. For example, Module A has a dependency on

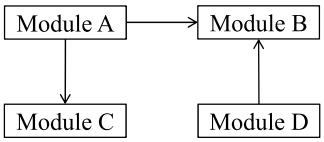


FIGURE 2. A MDN with 4 software modules.

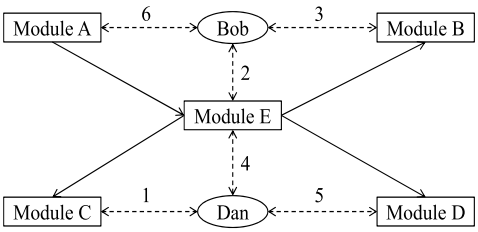


FIGURE 3. A STN with 2 developers and 5 software modules.

Modules *B* and *C* respectively. Modules *A* and *D* are both dependent on Module *B*.

C. SOCIO-TECHNICAL NEWORK (STN)

In [11], Bird et al. argue that the dependency relations and contribution history should be used together for fault-proneness prediction. They construct a socio-technical network by combining the developer contribution network and the module dependency network.

In the socio-technical network, there is a bidirectional dashed edge (denoted as the contribution edge) between a developer and a software module if the developer has made a commit to the module. The weight on the contribution edge is set as the number of commits from a developer to a module, and the weight of module dependencies is set to 1. Figure 3 shows a sample socio-technical network with 2 developers and 5 modules. For example, developer *Bob* has made 6, 2, and 3 commits to Modules *A*, *E*, and *B*, respectively. Module *E* has dependencies on Modules *B*, *C*, and *D*, respectively.

D. DEVELOPER COLLABORATION NETWORK (DN)

Meneeley *et al.* [46] construct a developer collaboration network consisting solely of developers in which edges between developers are based on collaboration on common modules. The authors use social network analysis to assign values of metrics to developers. The value of a metric for a module is based on the values of the developers that contributed to that module (e.g., the sum of a metric for developers for a module).

Figure 4 depicts a sample developer network with 4 developers. Circles represent developers and edges represent common files that two developers have both worked on in a particular release. For example, developers *Bob* and *Pan* have both worked on Module *A* during Release *R*1.

As mentioned above, DCN, MDN, STN, and DN are built either by a single relation or by a pair of relations. These networks also seem to miss an important factor developer quality as it is developers who make mistakes and create bugs during software development. Therefore, we intend to propose an enhanced social network which integrates the features of these four networks with additional adjustments.

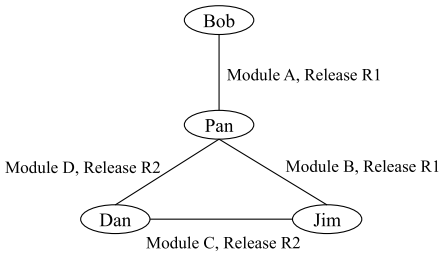


FIGURE 4. A DN with 4 software developers.

III. THE PROPOSED TRI-RELATION NETWORK (TRN)

The motivation behind TRN is that a network integrating developer contribution, module dependency, and developer collaboration can provide a more fully comprehensive insight into the interactions between developers and modules than the use of networks based on either a single or a paired relation. This insight is expected to ultimately enhance the effectiveness of software fault-proneness prediction.

In a TRN, there is a directed edge (denoted as the developer contribution) between a developer and a software module if the developer has made a commit to the module between two consecutive releases (e.g., between Release *R* and Release $R + 1$). The weight on the contribution edge is set as the normalized[4] number of commits made from a developer to a module between Release *R* and Release $R + 1$. Dependencies between modules are represented as directed dash-dot edges with arrows pointing to the modules upon which other modules depend. It is worth noting that we consider two types of dependency: functional dependency (i.e., a function in a file calls another function in a different file) and logical dependency (i.e., two files are modified in the same commit). We believe the use of both dependency types provides a more accurate representation of module dependencies affecting the development effort. The well-known commercialized tool *Understand* from SciTools [69] is used to quantify the normalized dependencies between two modules. The weights for logical dependency between two modules are computed as the normalized number of times that these two modules are modified in the same commit. The resulting module dependency is computed as the sum of normalized functional dependency and normalized logical dependency. In addition, there is a bidirectional dotted edge between one developer and another if these two developers have made at least one commit on the same module between release *R* and Release $R+1$. The weight on collaboration edge is computed as the normalized number of modules two developers have worked on together between Release *R* and Release $R + 1$. Figure 5 presents a TRN with 3 developers and 4 modules. For example, the

[4]In this paper, we apply the Min-Max approach for data normalization.

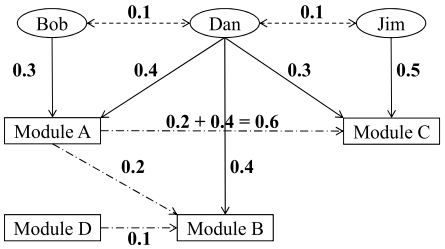


FIGURE 5. A TRN with 3 developers and 4 software modules.

weight on the developer contribution edge *Bob*-to-Module *A* is 0.3. The module dependency edge Module *A*-to-Module *C* is 0.2 (normalized functional dependency) + 0.4 (normalized logical dependency) = 0.6. The developer collaboration edge *Bob*-to-*Dan* is 0.1.

IV. METRICS

In this section, we first introduce four network node centrality metrics that are used in our study. Then we present another ten software metrics that are commonly used for predicting software fault-proneness.

Network node centrality metrics stem from social network theory and are used to quantify the location of a node to the rest of the network. There are three types of network node centrality[5]: (1) degree centrality, (2) closeness centrality, and (3) betweenness centrality. Degree centrality metrics are computed based on the number of edges that a node has. The more edges a node has, the more central is the node. Two degree centrality metrics are used in our study: Freeman degree centrality (denoted as M_{FDC}) and Bonacich's power (denoted as M_{BP}). M_{FDC} here is calculated as the number of direct edges a node has to its neighbors. M_{BP} is based on the adjacencies. It takes into account the connections of one's connections, in addition to one's own connections. M_{FDC} and M_{BP} focus on the number of developers and other modules it directly connects to, the impact of direct interactions on the module. The more people that are working on the module, the higher the probability of introducing faults due to inconsistent coding style, especially when these people have never worked/communicated with each other before. Due to the direct dependency relationship, the more changes that are made on its neighbor modules, the higher the probability that appropriate changes should be made on the module accordingly, thus the more difficult to maintain the module. Closeness centrality emphasizes the distance of a node to other nodes in the network. In this paper, we use one such node distance measure: eigenvector of geodesic distances (denoted as M_{EGD}). M_{EGD} finds the most central nodes (i.e. those with the smallest farness from others) in terms of the ''global'' or ''overall'' structure of the network, and pays less attention to patterns that are more ''local''. Specifically, M_{EGD} applies factor analysis to identify ''dimensions'' of the distances among nodes. The location of each node with respect to each dimension is called an ''eigenvalue'', and the collection of such values is called the ''eigenvector''. Usually, the first dimension captures the ''global'' aspects of distances among nodes; second and further dimensions capture more specific and local sub-structures. Betweenness centrality denotes the extent to which information flows through a node to get from one node to another. The more information flows through a node, the higher its betweenness centrality. For betweenness centrality, we use one such metric, Freeman node betweenness (denoted as M_{FNB}). It counts how frequently each node falls in the geodesic paths between all pairs of nodes. M_{EGD} and M_{FNB} focus on the connection strength of the module to all modules and developers that it either directly or indirectly connects to. The closer the module to other modules and developers, the stronger the connection, the more likely the module can be affected by other modules and developers in a way. In total, these four centrality metrics (i.e., M_{FDC}, M_{BP}, M_{EGD}, and M_{FNB}), which are also widely used in previous studies [11], [46], [50], [59], [87], are used in our study. We use a tool, *Ucinet* [8], to compute the values of these four centrality metrics based on the instructions given by Hanneman and Riddle [29].

[5]For more details about the computation of the centrality metrics introduced in this section, we refer the reader to textbooks and articles such as [13], [14], [22], [29], and [71].

TABLE 1. Ten commonly used software metrics.

Metric	Description
M_{LOC}	Lines of code
M_{McCabe}	McCabe cyclomatic complexity
M_{LCOM}	Lack of cohesion in methods
M_{DIT}	Depth of inheritance tree
M_{CBO}	Coupling between object classes
M_{NOC}	Number of children
M_{RFC}	Response for a class
M_{WMC}	Weighted methods per class
M_{NC}	Number of commits
M_{ND}	Number of developers

Additionally, we introduce another ten software metrics, as shown in TABLE 1, that are commonly used for predicting software fault-proneness, including Lines of Code [51], McCabe Complexity [45], all six CK metrics [18], number of commits [47], and number of developers [58], all of which will be used later in our case studies. We use a tool, *Understand* [69], to compute the values of these ten metrics.

For the sake of both simplicity and consistency, we use the notations provided in TABLE 2. For example, X represents a generic weighted network such as TRN, DCN, MDN, STN, or DN. M_{Cen} represents a generic network code centrality metric (e.g., M_{FDC}, M_{BP}, M_{EGD}, or M_{FNB}). M_{X-Cen} represents a M_{Cen} derived from X. For example, $M_{TRN-FDC}$ represents the FDC network node centrality metric derived from a TRN. Meanwhile, all four network node centrality metrics derived from a TRN (i.e., $M_{TRN-FDC}$, M_{TRN-BP}, $M_{TRN-EGD}$, and $M_{TRN-FNB}$) can now be simplified to M_{TRN}. We use M_{CO} to denote a metric set that contains the ten software metrics described in TABLE 1. In addition, we use Φ (M_X)

TABLE 2. Notations relevant to metrics, networks, and fault-proneness prediction models.

Abbrev	Description
X	a network
M	a generic software metric
Φ	a generic software fault-proneness prediction model
M_{Cen}	a generic network code centrality metric
$M_{X\text{-}Cen}$	a network code centrality metric derived from X
M_X	a set of four network code centrality metrics derived from X
M_{CO}	a set of ten software metrics that are commonly used for software fault-proneness prediction
$\Phi(M_X)$	a software fault-proneness prediction model using M_X
$\Phi(M_{CO})$	a software fault-proneness prediction model using M_{CO}

TABLE 3. Network node centrality metrics derived from TRN, DCN, MDN, STN, and DN.

$M_{TRN\text{-}FDC}$	$M_{DCN\text{-}FDC}$	$M_{MDN\text{-}FDC}$	$M_{STN\text{-}FDC}$	$M_{DN\text{-}FDC}$
$M_{TRN\text{-}BP}$	$M_{DCN\text{-}BP}$	$M_{MDN\text{-}BP}$	$M_{STN\text{-}BP}$	$M_{DN\text{-}BP}$
$M_{TRN\text{-}EGD}$	$M_{DCN\text{-}EGD}$	$M_{MDN\text{-}EGD}$	$M_{STN\text{-}EGD}$	$M_{DN\text{-}EGD}$
$M_{TRN\text{-}FNB}$	$M_{DCN\text{-}FNB}$	$M_{MDN\text{-}FNB}$	$M_{STN\text{-}FNB}$	$M_{DN\text{-}FNB}$

and Φ (M_{CO}) to denote a software fault-proneness prediction model using all four network code centrality metrics derived from X and a prediction model using the ten commonly used metrics, respectively. Since for each network we have four centrality metrics, a total of 20 metrics can be derived as shown in TABLE 3.

V. CASE STUDIES

In this section, we examine the three research questions related to our study, followed by a discussion of the software programs and the data analysis techniques used in our case studies. Results are presented at the end of this section.

A. THREE RESEARCH QUESTIONS

Here we simply revisit the three research questions from Section I.

- R1 Are centrality metrics derived from TRN important indicators for the number of post-released bugs in a file?
- R2 Do centrality metrics derived from TRN effectively improve software fault-proneness prediction models?
- R3 Will the fault-proneness prediction effectiveness improve if applying the proposed edge calibration mechanism on TRN and other four networks?

Answers to these three questions can help determine whether TRN-based centrality metrics are more powerful in building fault-proneness prediction models than not only DCN-, MDN-, STN-, or DN-based centrality metrics, but also software metrics that are commonly used for fault-proneness prediction. Moreover, valuable insights will be gained regarding the contributing factors that can be used to refine current networks in order to further enhance the prediction effectiveness.

B. SIX SOFTWARE PROGRAMS STUDIED

Our experiments use six programs, Camel [1], Flume [2], Tika [3], Gedit [23], Nginx [49], and Redis [61]. Camel is an open-source integration framework to define routing and mediation rules in a variety of domain-specific languages. Flume is a distributed service for collecting, aggregating, and moving log data from different sources to a centralized data store. Tika detects and extracts metadata and text from different file types such as PPT, XLS, and PDF. Gedit is the GNOME text editor. Nginx is an HTTP and reverse proxy server, a mail proxy server, and a generic TCP/UDP proxy server. Redis is an open source, in-memory data structure store, used as a database, cache and message broker. TABLE 4 summarizes the information for these six programs used in our case studies. The columns, starting from the left, give project name, release version, lines of code (including blanks and comments), number of files, and number of faulty files. Each program contains two consecutive releases. The values of all metrics are collected at file level.

TABLE 4. Program information.

Program	Release	LOC	Files	Faulty Files
Camel	1.3.0	110,113	1,245	81
Camel	1.4.0	114,621	1,488	43
Flume	1.4.0	92,437	507	54
Flume	1.5.0	99,127	547	40
Tika	1.5	83,502	472	14
Tika	1.6	87,180	507	18
Gedit	2.25.3	59,830	217	21
Gedit	2.25.4	61,052	219	16
Nginx	1.2.0	80,564	287	23
Nginx	1.3.0	80,672	287	13
Redis	2.6.0.7	44,882	227	24
Redis	2.6.0.8	47,100	228	18

TABLE 5. Correlation analysis using spearman rank correlation coefficient for Camel 1.4.0 where correlation is significant at the 0.05 level.

$M_{TRN\text{-}FDC}$	0.79	$M_{DCN\text{-}FDC}$	0.73	$M_{MDN\text{-}FDC}$	0.62	$M_{STN\text{-}FDC}$	0.71	$M_{DN\text{-}FDC}$	0.63
$M_{TRN\text{-}BP}$	0.55	$M_{DCN\text{-}BP}$	0.5	$M_{MDN\text{-}BP}$	0.47	$M_{STN\text{-}BP}$	0.5	$M_{DN\text{-}BP}$	0.45
$M_{TRN\text{-}EGD}$	0.54	$M_{DCN\text{-}EGD}$	0.46	$M_{MDN\text{-}EGD}$	0.39	$M_{STN\text{-}EGD}$	0.5	$M_{DN\text{-}EGD}$	0.36
$M_{TRN\text{-}FNB}$	0.44	$M_{DCN\text{-}FNB}$	0.37	$M_{MDN\text{-}FNB}$	0.26	$M_{STN\text{-}FNB}$	0.36	$M_{DN\text{-}FNB}$	0.33
M_{LOC}	0.33	M_{CBO}	0.23	M_{NC}	0.38	M_{LCOM}	0.11	M_{RFC}	0.1
M_{McCabe}	0.31	M_{NOC}	0.03	M_{ND}	0.34	M_{DIT}	0.05	M_{WMC}	0.25

C. EXPERIMENTAL METHODOLOGY

In order to answer R1, we use Spearman rank correlation coefficient [65] to measure the correlation between each metric (described in TABLE 5 through TABLE 10) and the number of post-released bugs for Camel 1.4.0, Flume 1.5.0, Tika 1.6, Gedit 2.25.4, Nginx 1.3.0, and Redis 2.6.0.8 respectively. The coefficient is between $+1$ and -1, inclusive, in which $+1$ is total positive correlation, 0 is no correlation, and -1 is total negative correlation.

In order to answer R2, we use a data mining tool, Weka [72], to construct different fault-proneness prediction models. For each program, a total of six datasets are formed based on TRN, DCN, MDN, STN, and DN, as well as a dataset consisting of ten commonly used metrics (denoted as CO-based dataset) from two consecutive software releases

TABLE 6. Correlation analysis using spearman rank correlation coefficient for Flume 1.5.0 where correlation is significant at the 0.05 level.

$M_{TRN-FDC}$	0.76	$M_{DCN-FDC}$	0.71	$M_{MDN-FDC}$	0.61	$M_{STN-FDC}$	0.65	M_{DN-FDC}	0.62
M_{TRN-BP}	0.69	M_{DCN-BP}	0.6	M_{MDN-BP}	0.49	M_{STN-BP}	0.63	M_{DN-BP}	0.52
$M_{TRN-EGD}$	0.6	$M_{DCN-EGD}$	0.5	$M_{MDN-EGD}$	0.4	$M_{STN-EGD}$	0.5	M_{DN-EGD}	0.4
$M_{TRN-FNB}$	0.56	$M_{DCN-FNB}$	0.45	$M_{MDN-FNB}$	0.37	$M_{STN-FNB}$	0.5	M_{DN-FNB}	0.4
M_{LOC}	0.21	M_{CBO}	0.05	M_{NC}	0.58	M_{LCOM}	0.12	M_{RFC}	0.01
M_{McCabe}	0.06	M_{NOC}	0.01	M_{ND}	0.43	M_{DIT}	0.02	M_{WMC}	0.01

TABLE 7. Correlation analysis using spearman rank correlation coefficient for Tika 1.6 where correlation is significant at the 0.05 level.

$M_{TRN-FDC}$	0.65	$M_{DCN-FDC}$	0.53	$M_{MDN-FDC}$	0.46	$M_{STN-FDC}$	0.52	M_{DN-FDC}	0.41
M_{TRN-BP}	0.52	M_{DCN-BP}	0.42	M_{MDN-BP}	0.39	M_{STN-BP}	0.45	M_{DN-BP}	0.3
$M_{TRN-EGD}$	0.56	$M_{DCN-EGD}$	0.47	$M_{MDN-EGD}$	0.36	$M_{STN-EGD}$	0.46	M_{DN-EGD}	0.39
$M_{TRN-FNB}$	0.4	$M_{DCN-FNB}$	0.33	$M_{MDN-FNB}$	0.25	$M_{STN-FNB}$	0.34	M_{DN-FNB}	0.21
M_{LOC}	0.29	M_{CBO}	0.19	M_{NC}	0.62	M_{LCOM}	0.32	M_{RFC}	0.06
M_{McCabe}	0.12	M_{NOC}	0.03	M_{ND}	0.42	M_{DIT}	0.09	M_{WMC}	0.14

TABLE 8. Correlation analysis using spearman rank correlation coefficient for Gedit 2.25.4 where correlation is significant at the 0.05 level.

$M_{TRN-FDC}$	0.53	$M_{DCN-FDC}$	0.50	$M_{MDN-FDC}$	0.42	$M_{STN-FDC}$	0.47	M_{DN-FDC}	0.43
M_{TRN-BP}	0.43	M_{DCN-BP}	0.39	M_{MDN-BP}	0.33	M_{STN-BP}	0.38	M_{DN-BP}	0.32
$M_{TRN-EGD}$	0.40	$M_{DCN-EGD}$	0.33	$M_{MDN-EGD}$	0.26	$M_{STN-EGD}$	0.35	M_{DN-EGD}	0.26
$M_{TRN-FNB}$	0.34	$M_{DCN-FNB}$	0.29	$M_{MDN-FNB}$	0.22	$M_{STN-FNB}$	0.29	M_{DN-FNB}	0.25
M_{LOC}	0.20	M_{CBO}	0.10	M_{NC}	0.33	M_{LCOM}	0.10	M_{RFC}	0.06
M_{McCabe}	0.13	M_{NOC}	0.02	M_{ND}	0.27	M_{DIT}	0.04	M_{WMC}	0.10

TABLE 9. Correlation analysis using spearman rank correlation coefficient for Nginx 1.3.0 where correlation is significant at the 0.05 level.

$M_{TRN-FDC}$	0.73	$M_{DCN-FDC}$	0.65	$M_{MDN-FDC}$	0.54	$M_{STN-FDC}$	0.62	M_{DN-FDC}	0.53
M_{TRN-BP}	0.54	M_{DCN-BP}	0.46	M_{MDN-BP}	0.43	M_{STN-BP}	0.48	M_{DN-BP}	0.38
$M_{TRN-EGD}$	0.57	$M_{DCN-EGD}$	0.48	$M_{MDN-EGD}$	0.38	$M_{STN-EGD}$	0.49	M_{DN-EGD}	0.40
$M_{TRN-FNB}$	0.44	$M_{DCN-FNB}$	0.36	$M_{MDN-FNB}$	0.27	$M_{STN-FNB}$	0.35	M_{DN-FNB}	0.29
M_{LOC}	0.32	M_{CBO}	0.23	M_{NC}	0.51	M_{LCOM}	0.22	M_{RFC}	0.09
M_{McCabe}	0.23	M_{NOC}	0.04	M_{ND}	0.40	M_{DIT}	0.07	M_{WMC}	0.22

TABLE 10. Correlation analysis using spearman rank correlation coefficient for Redis 2.6.0.8 where correlation is significant at the 0.05 level.

$M_{TRN-FDC}$	0.59	$M_{DCN-FDC}$	0.52	$M_{MDN-FDC}$	0.46	$M_{STN-FDC}$	0.48	M_{DN-FDC}	0.41
M_{TRN-BP}	0.50	M_{DCN-BP}	0.44	M_{MDN-BP}	0.37	M_{STN-BP}	0.43	M_{DN-BP}	0.33
$M_{TRN-EGD}$	0.49	$M_{DCN-EGD}$	0.39	$M_{MDN-EGD}$	0.34	$M_{STN-EGD}$	0.39	M_{DN-EGD}	0.33
$M_{TRN-FNB}$	0.39	$M_{DCN-FNB}$	0.34	$M_{MDN-FNB}$	0.29	$M_{STN-FNB}$	0.37	M_{DN-FNB}	0.26
M_{LOC}	0.23	M_{CBO}	0.10	M_{NC}	0.52	M_{LCOM}	0.20	M_{RFC}	0.06
M_{McCabe}	0.10	M_{NOC}	0.05	M_{ND}	0.35	M_{DIT}	0.06	M_{WMC}	0.09

(say Release 1 and Release 2). We use all data points collected from Release 1 as the training set. Because of the class imbalance issue in the training set we apply SMOTE [43], [70], [86] to oversample the minority class (i.e., classified as fault-prone) so that the size of fault-prone samples is equal to the size of samples that are non-fault-prone. For the training set, we randomly split all fault-prone classes in the training set into 30 equal-sized groups. We do the same for all non-fault-prone classes in the training set. Later, we randomly combine one fault-prone group with one non-fault-prone group to make one training subset. In this way, we have

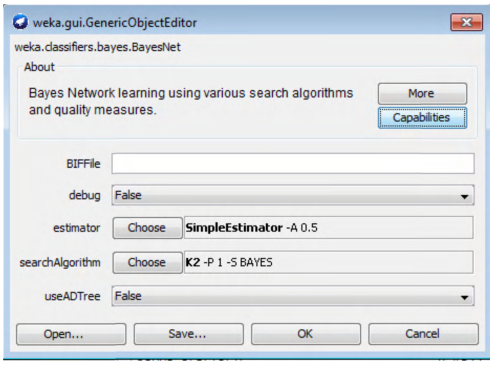


FIGURE 6. BayesNet in Weka.

randomly created 30 training subsets. We use BayesNet[6] as the training algorithm and apply the default setting on Weka as shown in Figure 6. For example, in a TRN-based dataset, each data point is a labeled (i.e., fault-prone or non-fault-prone) file characterized by M_{TRN} (i.e., $M_{TRN-FDC}$, M_{TRN-BP}, $M_{TRN-EGD}$, and $M_{TRN-FNB}$). In a CO-based dataset, each data point is a labeled file characterized by M_{CO}. The same train-predict process is repeated 30 times with each time using a different training subset. We use all data points collected from Release 2 as the test set. For example, we use $\Phi(M_{TRN})$ to represent a fault-proneness prediction model using M_{TRN}. As a result, for each program we have constructed 180 (i.e., 30×6) fault-proneness prediction models including 30 $\Phi(M_{TRN})$, 30 $\Phi(M_{DCN})$, 30 $\Phi(M_{MDN})$, 30 $\Phi(M_{STN})$, 30 $\Phi(M_{DN})$, and 30 $\Phi(M_{CO})$.

We use three measures, recall (defined in (1)), false positive rate (defined in (2)), and F1 score (defined in (4)) to evaluate the fault-proneness prediction effectiveness of each model. In the first three equations, TP (true positive) is the number of fault-prone modules that are correctly predicted, TN (true negative) is the number of non-fault-prone modules that are correctly predicted, FP (false positive) is the number of non-fault-prone modules that are predicted as fault-prone, and FN (false negative) is the number of fault-prone modules that are incorrectly predicted as non-fault-prone.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{1}$$

$$\text{False Positive Rate (FPR)} = \frac{\text{FP}}{\text{FP} + \text{TN}} \tag{2}$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{3}$$

[6]We use BayesNet in our experiment because it is robust to overfitting and does not assume data independence. As a matter of fact, many machine learning techniques such as neural networks [7], [37], [67], decision trees [6], [27], [28], case-based reasoning [36], [38], [55], Naïve Bayes [15], [31], [44], fuzzy logic [56], logistic regression [5], [9], [16], SVM [20], [25], [26], random forests [39], [63], and so on have been used for predicting software fault-proneness in the past. We want to emphasize that the focus of this study is to evaluate prediction effectiveness of the metrics derived from newly designed social networks. All these techniques can be used to train our dataset. However, the selection of the training algorithm used in the experiment is beyond the scope of this study.

$$\text{F1 score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (4)$$

For discussion purposes, let us assume there are 100 software modules, of which 30 are faulty and 70 are non-faulty. Assume also that a fault-proneness prediction model predicts 35 modules as fault-prone, 10 of which are actually non-faulty. This also implies that among the 65 modules predicted as non-fault-prone, 5 are actually fault-prone. Therefore, TP = 25 (25 faulty modules are correctly predicted as fault-prone), TN = 60 (60 non-faulty modules are correctly predicted as non-fault-prone), FP = 10 (ten non-faulty modules are incorrectly predicted as fault-prone), and FN = 5 (five faulty modules are incorrectly predicted as non-fault-prone). Based on Equations (1) and (2), the recall is $25/(25+5) \approx 83.33\%$ and the FPR is $10/(10+60) \approx 14.29\%$. Based on Equations (3) and (4), the precision is $25/(25+10) \approx 71.43\%$ and the F1 is $(2 \times 71.43\% \times 83.33\%)/(71.43\% + 83.33\%) \approx 76.92\%$. For two different fault-proneness prediction models Φ_1 and Φ_2, if Φ_1 has a higher recall or F1 than Φ_2, then it can be said that Φ_1 is more effective than Φ_2 with respect to recall or F1. If Φ_1 has a lower FPR than Φ_2, then it can be said that Φ_1 is more effective than Φ_2 with respect to FPR.

For each program, we compute and compare the respective average recall and FPR of 30 $\Phi(M_{TRN})$, 30 $\Phi(M_{DCN})$, 30 $\Phi(M_{MDN})$, 30 $\Phi(M_{STN})$, 30 $\Phi(M_{DN})$, and 30 $\Phi(M_{CO})$. For example, regarding R2, if $\Phi(M_{TRN})$ has a higher average recall than $\Phi(M_{DCN})$, then $\Phi(M_{TRN})$ is more effective than $\Phi(M_{DCN})$ with respect to average recall.

In addition, we employ the paired Wilcoxon signed-rank test [60] to investigate R1 and R2. For example, regarding R1, we can make the following null hypothesis with respect to the computed Spearman rank correlation coefficient using $M_{TRN-Cen}$ and the same coefficient using $M_{DCN-Cen}$:

H_0: The computed Spearman rank correlation coefficient using $M_{TRN-Cen}$ is equal to or smaller than the computed Spearman rank correlation coefficient using $M_{DCN-Cen}$. If H_0 is rejected (i.e., the alternative hypothesis is accepted), then it implies that $M_{TRN-Cen}$ is more correlated with the number of post-released bugs than $M_{DCN-Cen}$.

Regarding R2, we can make the following null hypothesis with respect to the recall of $\Phi(M_{TRN})$ and $\Phi(M_{DCN})$:

H_0: $\Phi(M_{TRN})$ has equal or lower recall than $\Phi(M_{DCN})$. If H_0 is rejected (i.e., the alternative hypothesis is accepted), then it implies that $\Phi(M_{TRN})$ will correctly predict more fault-prone files than $\Phi(M_{DCN})$. This also implies that $\Phi(M_{TRN})$ is more effective than $\Phi(M_{DCN})$ with respect to recall.

In order to answer R3, we first propose an approach to calibrate the edge weight of TRN and other networks. The modified networks with calibrated edges are denoted as CaX (e.g., CaTRN). Later, we investigate the relationship between the centrality metrics derived from CaX and the number of post-released bugs, as well as the performance of predicting fault-proneness using CaX-based metrics.

TABLE 11. Confidence that $M_{TRN-Cen}$ is more correlated with the number of bugs than the corresponding $M_{DCN-Cen}$, $M_{MDN-Cen}$, $M_{STN-Cen}$, and M_{DN-Cen}.

	$M_{DCN-Cen}$	$M_{MDN-Cen}$	$M_{STN-Cen}$	M_{DN-Cen}
Camel 1.4.0	98.98%	99.98%	99.98%	99.98%
Flume 1.5.0	97.97%	99.98%	98.97%	99.97%
Tika 1.6	98.98%	99.97%	99.98%	99.98%
Gedit 2.25.4	98.48%	99.98%	99.48%	99.98%
Nginx 1.3.0	98.98%	99.98%	99.99%	99.99%
Redis 2.6.0.8	98.48%	99.98%	99.48%	99.98%

D. RESULTS

To answer R1, we use the Spearman rank correlation coefficient to measure the correlation between each metric and the number of post-released bugs in a file.[7] The results are shown in TABLE 5 through TABLE 10. Each entry in the tables gives the coefficient between a metric and the number of bugs. For example, let us look at the first row of TABLE 5. The correlation between metric $M_{TRN-FDC}$ and the number of bugs is 0.79, and the correlation between $M_{DCN-FDC}$ and the number of bugs is 0.73. The corresponding correlations between metrics $M_{MDN-FDC}$, $M_{STN-FDC}$, and M_{DN-FDC} and the number of bugs are 0.62, 0.71, and 0.63, respectively. Therefore, the centrality metric, M_{FDC}, derived from TRN (i.e., $M_{TRN-FDC}$) has the strongest correlation with the number of bugs compared to the corresponding M_{FDC} derived from DCN (i.e., $M_{DCN-FDC}$), MDN (i.e., $M_{MDN-FDC}$), STN (i.e., $M_{STN-FDC}$), and DN (i.e., M_{DN-FDC}). Let us now look at the second column of the same table. The correlation between M_{TRN-BP} and the number of bugs is 0.55, the correlation between $M_{TRN-EGD}$ and the number of bugs is 0.54, and the correlation between $M_{TRN-FNB}$ and the number of bugs is 0.44. Therefore, the M_{FDC} derived from TRN (i.e., $M_{TRN-FDC}$) has the strongest correlation with the number of bugs compared to M_{TRN-BP}, $M_{TRN-EGD}$, and $M_{TRN-FNB}$ which are derived from the same TRN.

In general, from TABLE 5 to TABLE 10, we observe that: (1) $M_{TRN-FDC}$ has the strongest correlation (i.e., 0.79) with the number of bugs among all metrics; (2) for any M_{Cen} derived from TRN, $M_{TRN-Cen}$ has the strongest correlation with the number of bugs compared to the corresponding M_{Cen} derived from DCN (i.e., $M_{DCN-Cen}$), MDN (i.e., $M_{MDN-Cen}$), STN (i.e., $M_{STN-Cen}$), and DN (i.e., M_{DN-Cen}).

In addition, we use the paired Wilcoxon signed-rank test to investigate R1 from a statistical point of view. TABLE 11 presents the results of a Wilcoxon signed-rank test showing the confidence with which it can be claimed that $M_{TRN-Cen}$ is more correlated with the number of bugs than the corresponding $M_{DCN-Cen}$, $M_{MDN-Cen}$, $M_{STN-Cen}$, and M_{DN-Cen}. Each entry in the table strengthens the conviction that the alternative hypothesis stands. Furthermore, for each program in TABLE 11, at the 0.05 level, the

[7] Hereinafter, "bugs" will be used to refer specifically to post-release bugs.

TABLE 12. Prediction effectiveness evaluation of $\Phi(M_{CaTRN})$, $\Phi(M_{CaDCN})$, $\Phi(M_{CaMDN})$, $\Phi(M_{CaSTN})$, and $\Phi(M_{CaDN})$ with respect to average Recall, average FPR, and average F1 score for Camel 1.4.0.

Prediction Model	Average Recall	Average FPR	Average F1 score
$\Phi(M_{TRN})$	69.79%	4.60%	43.86%
$\Phi(M_{DCN})$	62.95%	4.86%	36.07%
$\Phi(M_{MDN})$	26.49%	6.23%	15.40%
$\Phi(M_{STN})$	44.09%	5.66%	26.92%
$\Phi(M_{DN})$	38.47%	6.63%	22.43%
$\Phi(M_{CO})$	54.15%	4.90%	34.57%

TABLE 13. Prediction effectiveness evaluation of $\Phi(M_{TRN})$, $\Phi(M_{DCN})$, $\Phi(M_{MDN})$, $\Phi(M_{STN})$, $\Phi(M_{DN})$, and $\Phi(M_{CO})$ with respect to average Recall, average FPR, and average F1 score for Flume 1.5.0.

Prediction Model	Average Recall	Average FPR	Average F1 score
$\Phi(M_{TRN})$	87.01%	4.51%	72.19%
$\Phi(M_{DCN})$	76.91%	6.17%	60.88%
$\Phi(M_{MDN})$	44.71%	9.96%	33.61%
$\Phi(M_{STN})$	64.79%	7.05%	51.54%
$\Phi(M_{DN})$	53.22%	9.58%	39.20%
$\Phi(M_{CO})$	68.14%	6.39%	55.51%

TABLE 14. Prediction effectiveness evaluation of $\Phi(M_{TRN})$, $\Phi(M_{DCN})$, $\Phi(M_{MDN})$, $\Phi(M_{STN})$, $\Phi(M_{DN})$, and $\Phi(M_{CO})$ with respect to average Recall, average FPR, and average F1 score for Tika 1.6.

Prediction Model	Average Recall	Average FPR	Average F1 score
$\Phi(M_{TRN})$	64.60%	2.10%	59.42%
$\Phi(M_{DCN})$	63.78%	2.38%	57.09%
$\Phi(M_{MDN})$	40.67%	3.79%	33.97%
$\Phi(M_{STN})$	52.78%	2.78%	46.72%
$\Phi(M_{DN})$	45.06%	2.97%	40.51%
$\Phi(M_{CO})$	53.97%	2.52%	49.58%

TABLE 15. Prediction effectiveness evaluation of $\Phi(M_{TRN})$, $\Phi(M_{DCN})$, $\Phi(M_{MDN})$, $\Phi(M_{STN})$, $\Phi(M_{DN})$, and $\Phi(M_{CO})$ with respect to average Recall, average FPR, and average F1 score for Gedit 2.25.4.

Prediction Model	Average Recall	Average FPR	Average F1 score
$\Phi(M_{TRN})$	78.40%	4.56%	58.03%
$\Phi(M_{DCN})$	69.93%	5.52%	48.48%
$\Phi(M_{MDN})$	35.60%	8.10%	24.51%
$\Phi(M_{STN})$	54.44%	6.36%	39.23%
$\Phi(M_{DN})$	45.85%	8.11%	30.82%
$\Phi(M_{CO})$	61.15%	5.65%	45.04%

correlation coefficient distributions are significantly different between (1) $M_{TRN-Cen}$ and $M_{DCN-Cen}$, (2) $M_{TRN-Cen}$ and $M_{MDN-Cen}$, (3) $M_{TRN-Cen}$ and $M_{STN-Cen}$, and (4) $M_{TRN-Cen}$ and M_{DN-Cen}. For example, for Camel 1.4.0, it can be said with 98.98% confidence that $M_{TRN-Cen}$ is more correlated with the number of bugs than the corresponding $M_{DCN-Cen}$. Let us look at the third row of TABLE 11; for Flume 1.5.0, it can be said with 97.97%, 99.98%, 98.97%, and 99.97% confidence that $M_{TRN-Cen}$ is more correlated with the number of bugs than the corresponding $M_{DCN-Cen}$, $M_{MDN-Cen}$, $M_{STN-Cen}$, and M_{DN-Cen}. In general, from TABLE 11 it can be claimed with high confidence (at least 97%) that $M_{TRN-Cen}$ is more correlated with the number of bugs than the corresponding $M_{DCN-Cen}$, $M_{MDN-Cen}$, $M_{STN-Cen}$, and M_{DN-Cen} for all six programs. If we change our alternative hypothesis to "$M_{TRN-Cen}$ is equally/more correlated with the number of bugs as/than the corresponding $M_{DCN-Cen}$, $M_{MDN-Cen}$, $M_{STN-Cen}$, and M_{DN-Cen}," then the confidence is 100% for almost very scenario.

Summary With Respect to R1:

- Metrics derived from the proposed TRN are significant indicators for the number of bugs in a file.
- Metrics derived from the proposed TRN are generally more correlated to the number of bugs than corresponding metrics derived from DCN, MDN, STN, and DN.
- The FDC metric derived from TRN, $M_{TRN-FDC}$, has the strongest correlation with the number of bugs among all metrics used in our case studies. This also indicates that for a software module (a file in our case), (1) the number of direct interactions with its contributing software developers, (2) the contribution frequency of these developers, (3) the number of modules with which it has a direct dependency relationship (both functional and logical), and (4) their mutual dependence intensity, jointly have a significant impact on the quality of the module itself.

To answer R2, for each program we compute and compare the average recall, FPR, and F1 score of $\Phi(M_{TRN})$, $\Phi(M_{DCN})$, $\Phi(M_{MDN})$, $\Phi(M_{STN})$, $\Phi(M_{DN})$, and $\Phi(M_{CO})$. The results are shown in TABLE 12 through TABLE 18. For example, in TABLE 17, the average recall, FPR, F1 score of $\Phi(M_{TRN})$ are 69.79%, 4.60%, and 43.86%, respectively. In the same table, the average recall, FPR, F1 score of $\Phi(M_{DCN})$ are 62.95%, 4.86%, and 36.07%, respectively. Therefore, $\Phi(M_{TRN})$ has a larger average recall, a lower average FRP, and a higher F1 score compared to $\Phi(M_{DCN})$. From TABLE 17, we observe that $\Phi(M_{TRN})$ has the highest average recall (i.e., 69.79%), the lowest average FPR (i.e., 4.60%), and the highest average F1 score (i.e., 43.86%) among all fault-proneness prediction models in the table. The same also applies to TABLE 18 and TABLE 17 where $\Phi(M_{TRN})$ has the highest average recall (i.e., 87.01%, 64.60%, 78.40%, 61.09%, and 72.20%), the lowest average FPR (i.e., 4.51%, 2.10%, 4.56%, 3.05%, and 3.15%), and the highest F1 score (i.e., 72.19%, 59.42%, 58.03%, 46.95%, and 62.67%) among all fault-proneness prediction models in these two tables.

Once again, from a statistical point of view, we employ the paired Wilcoxon signed-rank test to compare the recall and FPR of $\Phi(M_{TRN})$ against $\Phi(M_{DCN})$, $\Phi(M_{MDN})$, $\Phi(M_{STN})$, $\Phi(M_{DN})$, and $\Phi(M_{CO})$. TABLE 18 presents the results of a Wilcoxon signed-rank test showing the confidence with which it can be claimed that $\Phi(M_{TRN})$ is more effective (in terms of recall, FPR, and F1 score) than $\Phi(M_{DCN})$, $\Phi(M_{MDN})$, $\Phi(M_{STN})$, $\Phi(M_{DN})$, and $\Phi(M_{CO})$. Each entry in the table gives the assurance with which the alternative hypothesis stands. Furthermore, for each program in TABLE

TABLE 16. Prediction effectiveness evaluation of $\Phi(M_{TRN})$, $\Phi(M_{DCN})$, $\Phi(M_{MDN})$, $\Phi(M_{STN})$, $\Phi(M_{DN})$, and $\Phi(M_{CO})$ with respect to average Recall, average FPR, and average F1 score for Nginx 1.3.0.

Prediction Model	Average Recall	Average FPR	Average F1 score
$\Phi(M_{TRN})$	61.09%	3.05%	46.95%
$\Phi(M_{DCN})$	57.60%	3.29%	42.35%
$\Phi(M_{MDN})$	30.53%	4.55%	22.44%
$\Phi(M_{STN})$	44.03%	3.84%	33.47%
$\Phi(M_{DN})$	37.97%	4.36%	28.61%
$\Phi(M_{CO})$	49.15%	3.37%	38.25%

TABLE 17. Prediction effectiveness evaluation of $\Phi(M_{TRN})$, $\Phi(M_{DCN})$, $\Phi(M_{MDN})$, $\Phi(M_{STN})$, $\Phi(M_{DN})$, and $\Phi(M_{CO})$ with respect to average Recall, average FPR, and average F1 score for Redis 2.6.0.8.

Prediction Model	Average Recall	Average FPR	Average F1 score
$\Phi(M_{TRN})$	72.20%	3.15%	62.67%
$\Phi(M_{DCN})$	67.00%	4.07%	56.18%
$\Phi(M_{MDN})$	40.66%	6.55%	32.18%
$\Phi(M_{STN})$	55.99%	4.68%	46.79%
$\Phi(M_{DN})$	46.80%	5.98%	37.96%
$\Phi(M_{CO})$	58.15%	4.24%	50.04%

TABLE 18. Confidence that $\Phi(M_{TRN})$ is more effective than $\Phi(M_{DCN})$, $\Phi(M_{MDN})$, $\Phi(M_{STN})$, $\Phi(M_{DN})$, and $\Phi(M_{CO})$ with respect to Recall, FPR, and F1 score.

		$\Phi(M_{DCN})$	$\Phi(M_{MDN})$	$\Phi(M_{STN})$	$\Phi(M_{DN})$	$\Phi(M_{CO})$
Camel 1.4.0	Recall	99.98%	99.99%	99.99%	99.98%	99.98%
	FPR	99.99%	99.99%	99.99%	99.98%	99.97%
	F1 score	99.64%	99.86%	99.99%	99.78%	99.94%
Flume 1.5.0	Recall	99.98%	99.99%	99.98%	99.99%	99.98%
	FPR	99.98%	99.99%	99.99%	99.98%	99.99%
	F1 score	99.96%	99.91%	99.99%	99.93%	99.88%
Tika 1.6	Recall	99.96%	99.99%	99.97%	99.98%	99.98%
	FPR	99.96%	99.99%	99.97%	99.98%	99.99%
	F1 score	99.94%	99.99%	99.98%	99.97%	99.79%
Gedit 2.25.4	Recall	99.98%	99.99%	99.99%	99.99%	99.98%
	FPR	99.99%	99.99%	99.99%	99.98%	99.98%
	F1 score	99.80%	99.89%	99.99%	99.86%	99.91%
Nginx 1.3.0	Recall	99.97%	99.99%	99.98%	99.98%	99.98%
	FPR	99.98%	99.99%	99.98%	99.98%	99.98%
	F1 score	99.79%	99.93%	99.99%	99.88%	99.87%
Redis 2.6.0.8	Recall	99.97%	99.99%	99.98%	99.99%	99.98%
	FPR	99.97%	99.99%	99.98%	99.98%	99.99%
	F1 score	99.95%	99.95%	99.99%	99.95%	99.84%

18, at the 0.05 level, the recall/FPR/F1 score distributions are significantly different between (1) $\Phi(M_{TRN})$ and $\Phi(M_{DCN})$, (2) $\Phi(M_{TRN})$ and $\Phi(M_{MDN})$, (3) $\Phi(M_{TRN})$ and $\Phi(M_{STN})$, (4) $\Phi(M_{TRN})$ and $\Phi(M_{DN})$, and (5) $\Phi(M_{TRN})$ and $\Phi(M_{CO})$, respectively. To take an example from TABLE 18, it can be said with 99.98%, 99.99%, 99.64% confidence that $\Phi(M_{TRN})$ has a higher recall, lower FPR, and higher F1 score respectively, than $\Phi(M_{DCN})$ for Camel 1.4.0. It also implies that $\Phi(M_{TRN})$ is more effective than $\Phi(M_{DCN})$ in terms of recall, FPR, and F1 score, respectively. In general, from TABLE 18 we observe that it can be said with at least 99% confidence that $\Phi(M_{TRN})$ has a higher recall, lower FPR, and higher F1 score than the corresponding $\Phi(M_{DCN})$, $\Phi(M_{MDN})$, $\Phi(M_{STN})$, $\Phi(M_{DN})$, and $\Phi(M_{CO})$ for all six programs. If we change our alternative hypothesis to consider equalities, then the confidence is 100% for almost every scenario.

Summary With Respect to R2:

Fault-proneness prediction models using network node centrality metrics derived from the proposed TRN are more effective than prediction models using the same metrics derived from DCN, MDN, STN, and DN as well as prediction models using the ten common metrics, in terms of recall, FPR, and F1 score.

To answer R3, we propose CaTRN. The motivation behind the construction of the CaTRN is to investigate whether integrating additional factors that describe the development effort in the current TRN will better present the interactions between developers and modules and therefore further improve the fault-proneness prediction using the metrics derived from CaTRN. Consequently, in order to construct a CaTRN, for each type of relation in a TRN, a particular mechanism is applied to further calibrate the corresponding relation strength (i.e., the weight on the corresponding edges).

Specifically, we introduce developer risk score (DRS) [40], which computes the risk of a developer working on the modules, and use it for further edge weight calibration. DRS is based on two heuristics: (1) with respect to a given program, the more frequently a developer has introduced bugs in past releases, and the greater the severity of those bugs, the higher the risk that this program will contain a bug if this same developer makes a commit on the current release; and (2) the greater the complexity of a program, the greater the difficulty a developer has in working on this program and the higher the risk that the developer will introduce a bug into the program. For a given software system, assume that m_j is the j^{th} module in the system, and c_k is the k^{th} bug-introducing commit made by developer d in the j^{th} module. We retrieve the bug severity of each bug-introducing commit (i.e., critical, major, minor, or trivial) from JIRA [33] and use the function SeverityScore(f_j, c_k, d, R−1) to map it to one of the following scores: 4 (critical), 3 (major), 2 (minor), and 1 (trivial). A score of 4 is assigned to the variable MaxSeverityScore. The bug severity ratio of the k^{th} bug-introducing commit made by developer d in the j^{th} module in release R−1 is defined as:

$$SeverityRatio(m_j, c_k, d, R-1) = \frac{SeverityScore(m_j, c_k, d, R-1)}{MaxSeverityScore} \quad (5)$$

The overall complexity value of the j^{th} module in release R−1 is computed by Complexity(m_j, R−1) as the sum of normalized LOC [51], McCabe Complexity [45], and all six CK metrics [18]. TotalCommits(d, R−1) gives the total number of commits made by developer d in release R−1. DRS(d, R), the developer risk score of developer d at release R, is defined as:

$$DRS(d, R) = \frac{\sum \frac{SeverityRatio(m_j, c_k, d, R-1)}{Complexity(m_j, R-1)}}{TotalCommits(d, R-1)} \quad (6)$$

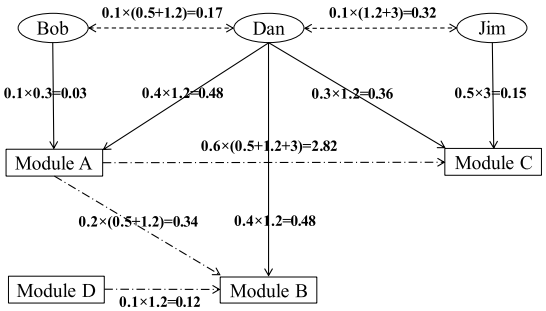


FIGURE 7. A CaTRN with three developers and four modules.

To calibrate the weight of a developer contribution edge, we multiply the original weight (i.e., the number of commits made by a developer) by the DRS value[8] of this developer. To calibrate the weight of a module dependency edge, we multiply the original weight (i.e., the quantified dependency value for a pair of modules) by the sum of DRS values of distinct developers who have worked on the two modules. To calibrate the weight of a developer collaboration edge, we multiply the original weight (i.e., the number of modules on which two developers have both worked) by the sum of DRS values of these two developers. Let us assume the DRS values for developers Bob, Dan, and Jim are 0.5, 1.2, and 3, respectively. The CaTRN is shown in Figure 7. Compared to TRN, CaTRN contains additional information by considering developer risk, program complexity, and bug severity, thus describing the development effort from a more comprehensive perspective.

The same calibrating strategy can also apply to DCN, MDN, STN, and DN. As a result, a total of seven modified networks are obtained (i.e., CaTRN, CaDCN, CaMDN, CaSTN, and CaDN). Once we have these modified networks, the corresponding network centrality metrics from each modified network are derived, respectively. Then, we investigate R3 by re-conducting similar data analysis which has been used to investigate R1 and R2. TABLE 19 through TABLE 24 present the Spearman rank correlation coefficient used to measure the correlation between each metric derived from the corresponding modified networks and the number of bugs in a file. For example, in the first row of TABLE 19, the correlation between the FDC metric derived from CaTRN (i.e., $M_{CaTRN-FDC}$) and the number of bugs is 0.87. As you may recall, the same FDC metric derived from TRN (i.e., $M_{TRN-FDC}$) in TABLE 5 is 0.79. This indicates that the FDC metric derived from the modified TRN (i.e., $M_{CaTRN-FDC}$) that consider calibrated edge weight has a stronger correlation with the number of bugs than the same FDC metric derived from the original TRN that does not.

In general, from TABLE 19 to TABLE 24, we observe that: (1) the metrics derived from modified networks (i.e., $M_{CaX-Cen}$) have stronger correlation with the number of bugs than the corresponding metrics derived from the original networks (i.e., M_{X-Cen}) as shown from TABLE 5 to TABLE 10; (2) $M_{CaTRN-FDC}$ has the strongest correlation with the number of bugs among all metrics derived from modified networks; and (3) for any M_{Cen} derived from CaTRN, $M_{CaTRN-Cen}$, it has the strongest correlation with the number of bugs compared to the corresponding M_{Cen} derived from CaDCN (i.e., $M_{CaDCN-Cen}$), CaMDN (i.e., $M_{CaMDN-Cen}$), CaSTN (i.e., $M_{CaSTN-Cen}$), and CaDN (i.e., $M_{CaDN-Cen}$).

In addition, the results of the paired Wilcoxon signed-rank test in TABLE 25 indicate that with high confidence (at least 98%), $M_{CaX-Cen}$ is more correlated with the number of bugs than M_{X-Cen}. The confidence increases to 100% for almost every scenario when considering equalities.

[8]For a newly joined developer, its DRS value is set to the median of the DRS value set which is currently available.

TABLE 19. Correlation analysis using spearman rank correlation coefficient for Camel 1.4.0 where correlation is significant at the 0.05 level.

$M_{CaTRN-FDC}$	0.87	$M_{CaDCN-FDC}$	0.80	$M_{CaMDN-FDC}$	0.67	$M_{CaSTN-FDC}$	0.75	$M_{CaDN-FDC}$	0.69
$M_{CaTRN-BP}$	0.60	$M_{CaDCN-BP}$	0.51	$M_{CaMDN-BP}$	0.50	$M_{CaSTN-BP}$	0.55	$M_{CaDN-BP}$	0.47
$M_{CaTRN-EGD}$	0.57	$M_{CaDCN-EGD}$	0.53	$M_{CaMDN-EGD}$	0.40	$M_{CaSTN-EGD}$	0.51	$M_{CaDN-EGD}$	0.40
$M_{CaTRN-FNB}$	0.52	$M_{CaDCN-FNB}$	0.42	$M_{CaMDN-FNB}$	0.31	$M_{CaSTN-FNB}$	0.37	$M_{CaDN-FNB}$	0.40

TABLE 20. Correlation analysis using spearman rank correlation coefficient for Flume 1.5.0 where correlation is significant at the 0.05 level.

$M_{CaTRN-FDC}$	0.83	$M_{CaDCN-FDC}$	0.79	$M_{CaMDN-FDC}$	0.64	$M_{CaSTN-FDC}$	0.71	$M_{CaDN-FDC}$	0.69
$M_{CaTRN-BP}$	0.71	$M_{CaDCN-BP}$	0.64	$M_{CaMDN-BP}$	0.56	$M_{CaSTN-BP}$	0.69	$M_{CaDN-BP}$	0.55
$M_{CaTRN-EGD}$	0.64	$M_{CaDCN-EGD}$	0.58	$M_{CaMDN-EGD}$	0.48	$M_{CaSTN-EGD}$	0.57	$M_{CaDN-EGD}$	0.44
$M_{CaTRN-FNB}$	0.61	$M_{CaDCN-FNB}$	0.50	$M_{CaMDN-FNB}$	0.40	$M_{CaSTN-FNB}$	0.51	$M_{CaDN-FNB}$	0.43

TABLE 21. Correlation analysis using spearman rank correlation coefficient for Tika 1.6 where correlation is significant at the 0.05 level.

$M_{CaTRN-FDC}$	0.67	$M_{CaDCN-FDC}$	0.57	$M_{CaMDN-FDC}$	0.52	$M_{CaSTN-FDC}$	0.54	$M_{CaDN-FDC}$	0.45
$M_{CaTRN-BP}$	0.55	$M_{CaDCN-BP}$	0.49	$M_{CaMDN-BP}$	0.41	$M_{CaSTN-BP}$	0.47	$M_{CaDN-BP}$	0.33
$M_{CaTRN-EGD}$	0.61	$M_{CaDCN-EGD}$	0.51	$M_{CaMDN-EGD}$	0.42	$M_{CaSTN-EGD}$	0.51	$M_{CaDN-EGD}$	0.40
$M_{CaTRN-FNB}$	0.47	$M_{CaDCN-FNB}$	0.37	$M_{CaMDN-FNB}$	0.32	$M_{CaSTN-FNB}$	0.39	$M_{CaDN-FNB}$	0.25

TABLE 22. Correlation analysis using spearman rank correlation coefficient for Gedit 2.25.4 where correlation is significant at the 0.05 level.

$M_{CaTRN-FDC}$	0.57	$M_{CaDCN-FDC}$	0.55	$M_{CaMDN-FDC}$	0.44	$M_{CaSTN-FDC}$	0.50	$M_{CaDN-FDC}$	0.48
$M_{CaTRN-BP}$	0.44	$M_{CaDCN-BP}$	0.40	$M_{CaMDN-BP}$	0.35	$M_{CaSTN-BP}$	0.42	$M_{CaDN-BP}$	0.36
$M_{CaTRN-EGD}$	0.41	$M_{CaDCN-EGD}$	0.38	$M_{CaMDN-EGD}$	0.29	$M_{CaSTN-EGD}$	0.38	$M_{CaDN-EGD}$	0.29
$M_{CaTRN-FNB}$	0.38	$M_{CaDCN-FNB}$	0.33	$M_{CaMDN-FNB}$	0.24	$M_{CaSTN-FNB}$	0.30	$M_{CaDN-FNB}$	0.28

TABLE 23. Correlation analysis using spearman rank correlation coefficient for Nginx 1.3.0 where correlation is significant at the 0.05 level.

$M_{CaTRN-FDC}$	0.79	$M_{CaDCN-FDC}$	0.69	$M_{CaMDN-FDC}$	0.60	$M_{CaSTN-FDC}$	0.65	$M_{CaDN-FDC}$	0.58
$M_{CaTRN-BP}$	0.59	$M_{CaDCN-BP}$	0.51	$M_{CaMDN-BP}$	0.46	$M_{CaSTN-BP}$	0.52	$M_{CaDN-BP}$	0.42
$M_{CaTRN-EGD}$	0.60	$M_{CaDCN-EGD}$	0.54	$M_{CaMDN-EGD}$	0.43	$M_{CaSTN-EGD}$	0.52	$M_{CaDN-EGD}$	0.40
$M_{CaTRN-FNB}$	0.51	$M_{CaDCN-FNB}$	0.40	$M_{CaMDN-FNB}$	0.32	$M_{CaSTN-FNB}$	0.40	$M_{CaDN-FNB}$	0.33

TABLE 24. Correlation analysis using spearman rank correlation coefficient for Redis 2.6.0.8 where correlation is significant at the 0.05 level.

$M_{CaTRN-FDC}$	0.61	$M_{CaDCN-FDC}$	0.58	$M_{CaMDN-FDC}$	0.47	$M_{CaSTN-FDC}$	0.52	$M_{CaDN-FDC}$	0.46
$M_{CaTRN-BP}$	0.53	$M_{CaDCN-BP}$	0.48	$M_{CaMDN-BP}$	0.42	$M_{CaSTN-BP}$	0.48	$M_{CaDN-BP}$	0.38
$M_{CaTRN-EGD}$	0.52	$M_{CaDCN-EGD}$	0.46	$M_{CaMDN-EGD}$	0.37	$M_{CaSTN-EGD}$	0.45	$M_{CaDN-EGD}$	0.35
$M_{CaTRN-FNB}$	0.46	$M_{CaDCN-FNB}$	0.38	$M_{CaMDN-FNB}$	0.33	$M_{CaSTN-FNB}$	0.38	$M_{CaDN-FNB}$	0.29

TABLE 25. Confidence that $M_{CaX-Cen}$ is more correlated to the number of bugs than the corresponding M_{X-Cen}.

	$M_{CaX\text{-}Cen} > M_{X\text{-}Cen}$
Camel 1.4.0	98.98%
Flume 1.5.0	99.96%
Tika 1.6	99.97%
Gedit 2.25.4	99.47%
Nginx 1.3.0	99.48%
Redis 2.6.0.8	99.97%

TABLE 26. Prediction effectiveness evaluation of $\Phi(M_{CaTRN})$, $\Phi(M_{CaDCN})$, $\Phi(M_{CaMDN})$, $\Phi(M_{CaSTN})$, and $\Phi(M_{CaDN})$ with respect to average Recall, average FPR, and average F1 score for Camel 1.4.0.

Prediction Model	Average Recall	Average FPR	Average F1
$\Phi(M_{CaTRN})$	71.50%	4.59%	44.82%
$\Phi(M_{CaDCN})$	66.59%	4.69%	41.78%
$\Phi(M_{CaMDN})$	27.77%	5.48%	18.36%
$\Phi(M_{CaSTN})$	45.65%	5.53%	28.40%
$\Phi(M_{CaDN})$	38.86%	6.47%	22.57%

TABLE 27. Prediction effectiveness evaluation of $\Phi(M_{CaTRN})$, $\Phi(M_{CaDCN})$, $\Phi(M_{CaMDN})$, $\Phi(M_{CaSTN})$, and $\Phi(M_{CaDN})$ with respect to average Recall, average FPR, and average F1 score for Flume 1.5.0.

Prediction Model	Average Recall	Average FPR	Average F1
$\Phi(M_{CaTRN})$	88.80%	3.93%	75.53%
$\Phi(M_{CaDCN})$	80.73%	5.18%	66.22%
$\Phi(M_{CaMDN})$	46.92%	9.10%	36.45%
$\Phi(M_{CaSTN})$	65.75%	6.43%	54.25%
$\Phi(M_{CaDN})$	54.61%	8.94%	41.48%

In general, metrics derived from the modified networks that consider calibrated edge weights are more correlated with the number of bugs than the same metrics derived from the corresponding networks that do not.

Similarly, we also compute the average recall, the average FPR, and the average F1 score of $\Phi(M_{CaTRN})$, $\Phi(M_{CaDCN})$, $\Phi(M_{CaMDN})$, $\Phi(M_{CaSTN})$, and $\Phi(M_{CaDN})$. The results are shown in TABLE 26 through TABLE 31. For example, in TABLE 26, the average recall, FPR, and F1 score of $\Phi(M_{CaTRN})$ are 71.50%, 4.59%, and 44.82%, respectively. As stated previously, the average recall, FRP, and F1 score of the corresponding $\Phi(M_{TRN})$ in TABLE 12 are 69.79%, 4.60%, and 43.86%, respectively. This indicates that the fault-proneness prediction models based on modified TRN that consider calibrated edge weights (i.e., $\Phi(M_{CaTRN})$) are more effective than the models based on the corresponding TRN that do not (i.e., $\Phi(M_{TRN})$) in terms of average recall, FPR, and F1 score.

In general, from TABLE 26 to TABLE 31, we observe that: (1) fault-proneness prediction models based on modified networks that consider calibrated edge weights (i.e., $\Phi(M_{CaX})$) are more effective than the prediction models based on the corresponding networks that do not (i.e., $\Phi(M_X)$) as shown from TABLE 12 to TABLE 17 in terms of average recall, FPR, and F1 score; and (2) $\Phi(M_{CaTRN})$ has the highest average recall, FPR, and F1 score among all modified networks. Moreover, the results of the paired Wilcoxon signed-rank test in TABLE 32 indicate that with high confidence (at least 96%), $\Phi(M_{CaX})$ is more effective than $\Phi(M_X)$ in terms of recall, FPR, and F1 score. If we change our alternative hypothesis to consider equalities, then the confidence is 100% for almost every scenario.

TABLE 28. Prediction effectiveness evaluation of $\Phi(M_{CaTRN})$, $\Phi(M_{CaDCN})$, $\Phi(M_{CaMDN})$, $\Phi(M_{CaSTN})$, and $\Phi(M_{CaDN})$ with respect to average Recall, average FPR, and average F1 score for Tika 1.6.

Prediction Model	Average Recall	Average FPR	Average F1
$\Phi(M_{CaTRN})$	65.73%	1.73%	62.39%
$\Phi(M_{CaDCN})$	65.01%	2.25%	57.64%
$\Phi(M_{CaMDN})$	41.66%	3.00%	37.92%
$\Phi(M_{CaSTN})$	56.77%	2.24%	52.89%
$\Phi(M_{CaDN})$	45.45%	1.85%	47.20%

TABLE 29. Prediction effectiveness evaluation of $\Phi(M_{CaTRN})$, $\Phi(M_{CaDCN})$, $\Phi(M_{CaMDN})$, $\Phi(M_{CaSTN})$, and $\Phi(M_{CaDN})$ with respect to average Recall, average FPR, and average F1 score for Gedit 2.25.4.

Prediction Model	Average Recall	Average FPR	Average F1
$\Phi(M_{CaTRN})$	80.15%	4.26%	60.18%
$\Phi(M_{CaDCN})$	73.66%	4.94%	54.00%
$\Phi(M_{CaMDN})$	37.35%	7.29%	27.41%
$\Phi(M_{CaSTN})$	55.70%	5.98%	41.33%
$\Phi(M_{CaDN})$	46.74%	7.71%	32.03%

TABLE 30. Prediction effectiveness evaluation of $\Phi(M_{CaTRN})$, $\Phi(M_{CaDCN})$, $\Phi(M_{CaMDN})$, $\Phi(M_{CaSTN})$, and $\Phi(M_{CaDN})$ with respect to average Recall, average FPR, and average F1 score for Nginx 1.3.0.

Prediction Model	Average Recall	Average FPR	Average F1
$\Phi(M_{CaTRN})$	62.38%	2.87%	48.73%
$\Phi(M_{CaDCN})$	59.82%	3.15%	45.19%
$\Phi(M_{CaMDN})$	31.56%	3.85%	25.58%
$\Phi(M_{CaSTN})$	46.55%	3.53%	36.95%
$\Phi(M_{CaDN})$	38.32%	3.78%	31.71%

TABLE 31. Prediction effectiveness evaluation of $\Phi(M_{CaTRN})$, $\Phi(M_{CaDCN})$, $\Phi(M_{CaMDN})$, $\Phi(M_{CaSTN})$, and $\Phi(M_{CaDN})$ with respect to average Recall, average FPR, and average F1 score for Redis 2.6.0.8.

Prediction Model	Average Recall	Average FPR	Average F1
$\Phi(M_{CaTRN})$	73.59%	2.70%	65.68%
$\Phi(M_{CaDCN})$	69.40%	3.54%	58.98%
$\Phi(M_{CaMDN})$	42.18%	5.76%	35.41%
$\Phi(M_{CaSTN})$	58.34%	4.13%	51.02%
$\Phi(M_{CaDN})$	47.65%	5.14%	42.23%

Summary With Respect to R3:

- The developer risk score (DRS), which takes bug severity, program complexity, and development difficulty into account, contributes to the network refinement and improves the effectiveness of software-fault proneness prediction.

TABLE 32. Confidence $\Phi(M_{CaX})$ is more effective than the corresponding $\Phi(M_X)$.

		$\Phi(M_{CaX}) > \Phi(M_X)$
Camel 1.4.0	Recall	98.99%
	FPR	99.68%
	F1 score	97.83%
Flume 1.5.0	Recall	96.88%
	FPR	99.99%
	F1 score	98.22%
Tika 1.6	Recall	97.93%
	FPR	97.98%
	F1 score	98.03%
Gedit 2.25.4	Recall	97.94%
	FPR	99.84%
	F1 score	98.03%
Nginx 1.3.0	Recall	98.46%
	FPR	98.83%
	F1 score	97.93%
Redis 2.6.0.8	Recall	97.41%
	FPR	98.99%
	F1 score	98.13%

- Metrics derived from the modified network that considers calibrated edge weight using DRS are generally more correlated to the number of bugs than the same metrics derived from the corresponding networks that do not consider calibrated edge weight.
- Software fault-proneness prediction models using metrics derived from the modified networks that consider calibrated edge weight are more effective than prediction models using the same metrics derived from the corresponding networks that do not.

VI. THREATS TO VALIDITY

A. INTERNAL VALIDITY

The data analysis techniques used in Section V are suitable and commonly adopted for measuring the effectiveness of metrics in predicting software fault-proneness, but by themselves they do not provide a complete picture of the prediction performance. Therefore, we can only observe correlation through statistical measures, not causation. In order to investigate possible causal effects, a root-cause analysis along each variable still needs to be carried out. During the experiment we apply oversampling using SMOTE which may lead to overfitting. First, unlike traditional oversampling methods which simply duplicates minority classes, SMOTE operates in the ''feature space'' rather than the ''data space'' and generates synthetic samples of the minority classes. In this way, it effectively forces the decision region of the minority class to become more general, partially solving the generalization/overfitting problem. Second, we only apply SMOTE on the training data while the test data remain untainted. Third, we have tried random oversampling and the prediction performance is not ideal due to the impact of overfitting. In other words, SMOTE is more appropriate and effective to handle data imbalance and overfitting in our case. Our work lays the foundation for future investigations, by identifying potential connections between social interactions and software quality.

B. EXTERNAL VALIDITY

Only six open-source programs are used to investigate the four research questions. Therefore, our conclusion may not be generalized either to projects developed by other programming languages or to those that are commercialized. Our program selection is based on the information availability for the construction of TRN. To mitigate these threats, our study needs to be repeated on a wider variety of programs. Additionally, all data are collected from an issue tracking system, JIRA [33]. However, the information extracted from the issue tracking system may be incomplete. This potential threat can be alleviated by conducting more thorough data collection in future work. The developers may not be as well trained and experienced as average professional programmers. In this paper, we also use consecutive releases which may prompt replication of same bugs. However, using consecutive releases has been a common and successful practice in SFP. Our intention has never been catching similar or same bugs (according to some static code metrics extracted from modules) in the next release but by studying the impact of interaction between connected software developers and modules as well as the interaction within connected developers/modules (i.e., building a TRN) on module fault-proneness to guide us identify fault-prone modules which do not necessarily belong the same type but are due to a serial of combined social and technical influence during development.

VII. CONCLUSIONS AND FUTURE WORK

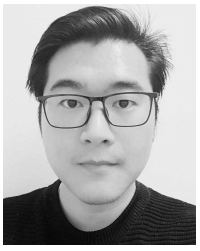
Previous studies have shown that the developer contribution relation, module dependency relation, and developer collaboration relation have been used to build networks for software fault-proneness prediction. However, none of these studies consider the combined influence of all three relations. Motivated by this, we integrate all three relations into one comprehensive network, our proposed tri-relation network (TRN). In addition, four network node centrality metrics (i.e., M_{FDC}, M_{BP}, M_{EDG}, and M_{FNB}) are derived from the corresponding network to predict the fault-proneness of a given file on six programs. The results of our study indicate that (1) TRN-based centrality metrics are more correlated with the number of bugs than the corresponding DCN-, MDN-, STN-, and DN-based centrality metrics as well as the ten software metrics that are commonly used for software fault-proneness prediction; (2) fault-proneness prediction models using TRN-based centrality metrics outperform the models using DCN-, MDN-, STN-, and DN-based centrality metrics as well as the models based on ten commonly used software metrics; (3) centrality metrics derived from a modified network that consider calibrated edge weight using developer risk score are more correlated to the number of bugs than those derived from the same network that does not; and (4) fault-proneness prediction models using centrality metrics derived from a modified network outperform the models using centrality

metrics derived from the same network that does not. In the future, we plan to repeat our study on a wider variety of programs and include additional software metrics for comparison to further validate the effectiveness of our TRN-based metrics. We also intend to search for potential intelligent algorithms for better prediction model training and further network refinement. In addition, it would be interesting to investigate whether our TRN-based centrality metrics can be used for cross-project software fault-proneness prediction when the historical information is limited or unavailable. Last but not least, we also plan to tailor TRN to the needs of industrial enterprise.

REFERENCES

[1] *Apache Camel*. Accessed: Mar. 23, 2017. [Online]. Available: http://camel.apache.org/
[2] *Apache Flume*. Accessed: Mar. 23, 2017. [Online]. Available: http://flume.apache.org/
[3] *Apache Tika*. Accessed: Mar. 23, 2017. [Online]. Available: http://tika.apache.org/
[4] E. Arisholm and L. C. Briand, "Predicting fault-prone components in a java legacy system," in *Proc. 5th ACM/IEEE Int. Symp. Empirical Softw. Eng.*, Rio de Janeiro, Brazil, Sep. 2006, pp. 8–17.
[5] P. Bellini, I. Bruno, P. Nesi, and D. Rogai, "Comparing fault-proneness estimation models," in *Proc. 10th IEEE Int. Conf. Eng. Complex Comput. Syst.*, Shanghai, China, Jun. 2005, pp. 205–214.
[6] A. Bernstein, J. Ekanayake, and M. Pinzger, "Improving defect prediction using temporal features and non linear models," in *Proc. 9th Int. Workshop Princ. Softw. Evol., Conjunction 6th ESEC/FSE Joint Meeting*, Dubrovnik, Croatia, Sep. 2007, pp. 11–18.
[7] M. E. R. Bezerra, A. L. I. Oliveira, and S. R. L. Meira, "A constructive RBF neural network for estimating the probability of defects in software modules," in *Proc. Int. Joint Conf. Neural Netw.*, Orlando, FL, USA, Aug. 2007, pp. 2869–2874.
[8] S. P. Borgatti, M. G. Everett, and L. C. Freeman, "UCINET for windows: Software for social network analysis," Analytic Technol., Harvard, MA, USA, Tech. Rep., 2002.
[9] L. C. Briand, S. Morasca, and V. R. Basili, "Defining and validating measures for object-based high-level design," *IEEE Trans. Softw. Eng.*, vol. 25, no. 5, pp. 722–743, Sep./Oct. 1999.
[10] C. L. Briand, J. Wüst, J. W. Daly, and D. V. Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," *J. Syst. Softw.*, vol. 51, no. 3, pp. 245–273, May 2000.
[11] C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu, "Putting it all together: Using socio-technical networks to predict failures," in *Proc. 20th Int. Symp. Softw. Rel. Eng.*, Mysuru, India, Nov. 2009, pp. 109–119.
[12] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code!: Examining the effects of ownership on software quality," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng.*, Szeged, Hungary, Sep. 2011, pp. 4–14.
[13] P. Bonacich, "Power and centrality: A family of measures," *Amer. J. Sociol.*, vol. 92, no. 5, pp. 1170–1182, Mar. 1987.
[14] U. Brandes and T. Erlebach, *Network Analysis: Methodological Foundations*. Berlin, Germany: Springer, 2005.
[15] C. Catal, U. Sevim, and B. Diri, "Practical development of an eclipse-based software fault prediction tool using Naive Bayes algorithm," *Expert Syst. Appl.*, vol. 38, no. 3, pp. 2347–2353, Mar. 2011.
[16] M. Cataldo, J. Herbsleb, and K. M. Carley, "Socio-technical congruence: A framework for assessing the impact of technical and work dependencies on software development," in *Proc. 2nd ACM-IEEE Int. Symp. Empirical Softw. Eng. Meas.*, Kaiserslautern, Germany, Oct. 2008, pp. 2–11.
[17] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb, "Software dependencies, work dependencies, and their impact on failures," *IEEE Trans. Softw. Eng.*, vol. 35, no. 6, pp. 864–878, Nov./Dec. 2009.
[18] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.
[19] G. Denaro and M. Pezz, "An empirical evaluation of fault-proneness models," in *Proc. 24th Int. Conf. Softw. Eng.*, Orlando, FL, USA, May 2002, pp. 241–251.
[20] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *J. Syst. Softw.*, vol. 81, no. 5, pp. 649–660, May 2008.
[21] J. Ell, "Identifying failure inducing developer pairs within developer networks," in *Proc. Int. Conf. Softw. Eng.*, San Francisco, CA, USA, May 2013, pp. 1471–1473.
[22] L. C. Freeman, "Centrality in social networks: Conceptual clarification," *Social Netw.*, vol. 1, no. 3, pp. 215–239, 1978.
[23] *Gedit*. Accessed: Feb. 11, 2017. [Online]. Available: https://wiki.gnome.org/Apps/Gedit
[24] R. A. Ghosh, "Clustering and dependencies in free/open source software development: Methodology and tools," *First Monday*, vol. 8, no. 4, pp. 1–27, Apr. 2003.
[25] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall, "Method-level bug prediction," in *Proc. ACM-IEEE Int. Symp. Empirical Softw. Eng. Meas.*, Lund, Sweden, Sep. 2012, pp. 171–180.
[26] I. Gondra, "Applying machine learning to software fault-proneness prediction," *J. Syst. Softw.*, vol. 81, vol. 2, pp. 186–195, Feb. 2008.
[27] L. Guo, Y. Ma, B. Cukic, and H. Singh, "Robust prediction of fault-proneness by random forests," in *Proc. 15th Int. Symp. Softw. Rel. Eng.*, Saint-Malo, France, Nov. 2004, pp. 417–428.
[28] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Trans. Softw. Eng.*, vol. 31, no. 10, pp. 897–910, Oct. 2005.
[29] R. A. Hanneman and M. Riddle, "Introduction to social network methods," Univ. California, Riverside, CA, USA, Tech. Rep., 2005.
[30] J. Howison, K. Inoue, and K. Crowston, "Social dynamics of free and open source team communications," in *Proc. Int. Conf. Open Source Syst.*, Jun. 2006, pp. 319–330.
[31] Y. Jiang, B. Cukic, and T. Menzies, "Fault prediction using early lifecycle data," in *Proc. 18th IEEE Int. Symp. Softw. Rel.*, Trollhattan, Sweden, Nov. 2007, pp. 237–246.
[32] Y. Jiang, B. Cuki, T. Menzies, and N. Bartlow, "Comparing design and code metrics for software quality prediction," in *Proc. 4th Int. Workshop Predictor Models Softw. Eng.*, Leipzig, Germany, May 2008, pp. 11–18.
[33] *JIRA*. Accessed: May 2, 2017. [Online]. Available: https://www.atlassian.com/software/jira
[34] T. M. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan, and N. Goel, "Early quality prediction: A case study in telecommunications," *IEEE Softw.*, vol. 13, no. 1, pp. 65–71, Jan. 1996.
[35] T. M. Khoshgoftaar, E. B. Allen, and J. Deng, "Using regression trees to classify fault-prone software modules," *IEEE Trans. Rel.*, vol. 51, no. 4, pp. 455–462, Dec. 2002.
[36] T. M. Khoshgoftaar, N. Seliya, and N. Sundaresh, "An empirical study of predicting software faults with case-based reasoning," *Softw. Qual. J.*, vol. 14, no. 2, pp. 85–111, Jun. 2006.
[37] T. M. Khoshgoftaar and R. M. Szabo, "Using neural networks to predict software faults during testing," *IEEE Trans. Rel.*, vol. 45, no. 3, pp. 456–462, Sep. 1996.
[38] T. M. Khoshgoftaar, Y. Xiao, and K. Gao, "Software quality assessment using a multi-strategy classifier," *Inf. Sci.*, vol. 259, pp. 555–570, Feb. 2014.
[39] P. Knab, M. Pinzger, and A. Bernstein, "Predicting defect densities in source code files with decision tree learners," in *Proc. Int. Workshop Mining Softw. Repositories*, Shanghai, China, May 2006, pp. 119–125.
[40] S.-Y. Lee and Y. Li, "DRS: A developer risk metric for better predicting software fault-proneness," in *Proc. 2nd Int. Conf. Trustworthy Syst. Appl.*, Hualien, Taiwan, Jul. 2015, pp. 120–127.
[41] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 485–496, Jul./Aug. 2008.
[42] Y. Li, "Applying social network analysis to software fault-proneness prediction," Ph.D. dissertation, Dept. Comput. Sci., Univ. Texas Dallas, Richardson, TX, USA, Aug. 2017. [Online]. Available: http://libtreasures.utdallas.edu/xmlui/handle/10735.1/5486
[43] R. Li and S. Wang, "An empirical study for software fault-proneness prediction with ensemble learning models on imbalanced data sets," *J. Softw.*, vol. 9, no. 3, pp. 697–704, Mar. 2014.
[44] H. Lu and B. Cukic, "An adaptive approach with active learning in software fault prediction," in *Proc. 8th Int. Conf. Predictive Models Softw. Eng.*, Lund, Sweden, Sep. 2012, pp. 79–88.
[45] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.
[46] A. Meneely, L. Williams, W. Snipes, and J. Osborne, "Predicting failures with developer networks and social network analysis," in *Proc. 16th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Atlanta, GA, USA, Nov. 2008, pp. 13–23.

[47] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proc. 27th Int. Conf. Softw. Eng.*, St. Louis, MO, USA, May 2005, pp. 284–292.
[48] N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality," in *Proc. 30th Int. Conf. Softw. Eng.*, Leipzig, Germany, May 2008, pp. 521–530.
[49] *Nginx*. Accessed: Jan. 12, 2017. [Online]. Available: https://nginx.org/en/
[50] T. H. D. Nguyen, B. Adams, and A. E. Hassan, "Studying the impact of dependency network measures on software quality," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Timisoara, Romania, Sep. 2010, pp. 1–10.
[51] V. Nguyen, S. Deeds-Rubin, T. Tan, and B. Boehm, "A SLOC counting standard," in *Proc. 22nd Int. Forum COCOMO Syst./Softw. Cost Modeling*, Los Angeles, CA, USA, Oct. 2007, pp. 1–16.
[52] M. Ohira, N. Ohsugi, T. Ohoka, and K.-I. Matsumoto, "Accelerating cross-project knowledge collaboration using collaborative filtering and social networks," in *Proc. Int. Workshop Mining Softw. Repositories*, St. Louis, MO, USA, May 2005, pp. 1–5.
[53] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Trans. Softw. Eng.*, vol. 22, no. 12, pp. 886–894, Dec. 1996.
[54] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Trans. Softw. Eng.*, vol. 31, no. 4, pp. 340–355, Apr. 2005.
[55] E. Paikari, B. Sun, G. Ruhe, and E. Livani, "Customization support for CBR-based defect prediction," in *Proc. 7th Int. Conf. Predictive Models Softw. Eng.*, Banff, AB, Canada, Sep. 2011, p. 16.
[56] A. K. Pandey and N. K. Goyal, "Predicting fault-prone software module using data mining technique and fuzzy logic," *Int. J. Comput. Commun. Technol.*, vol. 2, no. 2, pp. 56–63, Dec. 2010.
[57] E. J. Weyuker, T. J. Ostrand, and R. M. Bell, "Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models," *Empirical Softw. Eng.*, vol. 13, no. 5, pp. 539–559, Oct. 2008.
[58] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Programmer-based fault prediction," in *Proc. 6th Int. Conf. Predictive Models Softw. Eng.*, Timisoara, Romania, Sep. 2010, p. 19.
[59] M. Pinzger, N. Nagappan, and B. Murphy, "Can developer-module networks predict failures?" in *Proc. 16th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Atlanta, GA, USA, Nov. 2008, pp. 2–12.
[60] R. L. Ott and M. l. Longnecker, *An Introduction to Statistical Methods and Data Analysis*, 4th ed. Independence, KY, USA: Duxbury Press, 1993.
[61] *Redis*. Accessed: Feb. 12, 2018. [Online]. Available: https://redis.io/
[62] A. Sarma, L. Maccherone, P. Wagstrom, and J. Herbsleb, "Tesseract: Interactive visual exploration of socio-technical relationships in software development," in *Proc. IEEE 31st Int. Conf. Softw. Eng.*, Vancouver, BC, Canada, May 2009, pp. 22–33.
[63] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 772–787, Nov./Dec. 2011.
[64] A. Simpson, "Changeset based developer communication to detect software failures," in *Proc. 35th IEEE Int. Conf. Softw. Eng.*, San Francisco, CA, USA, May 2013, pp. 1468–1470.
[65] C. Spearman, "The proof and measurement of association between two things," *Amer. J. Psychol.*, vol. 15, no. 1, pp. 72–101, Jan. 1904.
[66] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Planning Rep. 02-3, May 2002.
[67] M. M. T. Thwin and T.-S. Quah, "Application of neural networks for software quality prediction using object-oriented metrics," *J. Syst. Softw.*, vol. 76, no. 2, pp. 147–156, May 2005.
[68] A. Tosun, B. Turhan, and A. Bener, "Validation of network measures as indicators of defective modules in software systems," in *Proc. 5th Int. Conf. Workshop Predictor Models Softw. Eng.*, Vancouver, BC, Canada, May 2009, p. 5.
[69] *Understand*. Accessed: Nov. 23, 2018. [Online]. Available: https://scitools.com/
[70] S. Wang and X. Yao, "Using class imbalance learning for software defect prediction," *IEEE Trans. Rel.*, vol. 62, no. 2, pp. 434–443, Jun. 2013.
[71] S. Wasserman and K. Faust, *Social Network Analysis: Methods and Applications*. New York, NY, USA: Cambridge Univ. Press, 1994.
[72] *WEKA*. Accessed: Aug. 7, 2017. [Online]. Available: http://www.cs.waikato.ac.nz/ml/weka/
[73] E. J. Weyuker, T. J. Ostrand, and R. M. Bell, "Comparing the effectiveness of several modeling methods for fault prediction," *Empirical Softw. Eng.*, vol. 15, no. 3, pp. 277–295, Jun. 2010.
[74] W. E. Wong, V. Debroy, and B. Choi, "A family of code coverage-based heuristics for effective fault localization," *J. Syst. Softw.*, vol. 83, no. 2, pp. 188–208, Feb. 2010.
[75] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The DStar method for effective software fault localization," *IEEE Trans. Rel.*, vol. 63, no. 1, pp. 290–308, Mar. 2014.
[76] W. E. Wong, V. Debroy, R. Golden, X. Xu, and B. Thuraisingham, "Effective software fault localization using an RBF neural network," *IEEE Trans. Rel.*, vol. 61, no. 1, pp. 149–169, Mar. 2012.
[77] W. E. Wong, V. Debroy, and D. Xu, "Towards better fault localization: A crosstab-based statistical approach," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 42, no. 3, pp. 378–396, May 2012.
[78] W. E. Wong and S. Gokhale, "Static and dynamic distance metrics for feature-based code analysis," *J. Syst. Softw.*, vol. 74, no. 3, pp. 283–295, Feb. 2005.
[79] W. E. Wong, S. S. Gokhale, and J. R. Horgan, "Metrics for quantifying the disparity, concentration, and dedication between program components and features," in *Proc. 6th Int. Softw. Metrics Symp.*, Nov. 1999, pp. 189–198.
[80] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, Aug. 2016.
[81] W. E. Wong, J. R. Horgan, M. Syring, W. Zage, and D. Zage, "Applying design metrics to predict fault-proneness: A case study on a large-scale software system," *Softw. Pract. Exper.*, vol. 30, no. 14, pp. 1587–1608, Nov. 2000.
[82] W. E. Wong, X. Li, and P. A. Laplante, "Be more familiar with our enemies and pave the way forward: A review of the roles bugs played in software failures," *J. Syst. Softw.*, vol. 133, pp. 68–94, Nov. 2017.
[83] W. E. Wong, Y. Qi, and K. Cooper, "Source code-based software risk assessing," in *Proc. ACM Symp. Appl. Comput.*, Santa Fe, NM, USA, Mar. 2005, pp. 1485–1490.
[84] W. E. Wong, J. Zhao, and V. K. Y. Chan, "Applying statistical methodology to optimize and simplify software metric models with missing data," in *Proc. ACM Symp. Appl. Comput.*, Dijon, France, Apr. 2006, pp. 1728–1733.
[85] J. Xu, Y. Gao, S. Christley, and G. Madey, "A topological analysis of the open souce software development community," in *Proc. 38th Annu. Hawaii Int. Conf. Syst. Sci.*, Big Island, HI, USA, Jan. 2005, pp. 1–10.
[86] C. W. Yohannese and T. Li, "A combined-learning based framework for improved software fault prediction," *Int. J. Comput. Intell. Syst.*, vol. 10, no. 1, pp. 647–662, Jan. 2017.
[87] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proc. ACM/IEEE 30th Int. Conf. Softw. Eng.*, Leipzig, Germany, May 2008, pp. 531–540.



YIHAO LI received the B.S. degree in software engineering from the East China Institute of Technology, the M.S. degree in computer science from Southeastern Louisiana University, and the Ph.D. degree from the Department of Computer Science, The University of Texas at Dallas, Richardson, TX, USA. He currently holds a postdoctoral position at the Graz University of Technology. His research interests include software risk analysis, software fault-proneness prediction, software fault localization, and program debugging.

W. ERIC WONG received the M.S. and Ph.D. degrees in computer science from Purdue University. He is a full professor and the founding director of the Advanced Research Center for Software Testing and Quality Assurance in Computer Science, University of Texas at Dallas (UTD). He also has an appointment as a guest researcher with National Institute of Standards and Technology (NIST), an agency of the US Department of Commerce. Prior to joining UTD, he was with Telcordia Technologies (formerly Bellcore) as a senior research scientist and the project manager in charge of Dependable Telecom Software Development. In 2014, he was named the IEEE Reliability Society Engineer of the Year. His research focuses on helping practitioners improve the quality of software while reducing the cost of production. In particular, he is working on software testing, debugging, risk analysis/metrics, safety, and reliability. He has very strong experience developing real-life industry applications of his research results. Professor Wong is the Editor-in-Chief of IEEE TRANSACTIONS ON RELIABILITY. He is also the Founding Steering Committee Chair of the IEEE International Conference on Software Quality, Reliability, and Security (QRS) and the IEEE International Workshop on Debugging and Repair (IDEAR).

SHOU-YU LEE received the B.S. degree in computer science from National Tsing Hua University, Taiwan, and the M.S. degree in computer science from Tunghai University. He is currently pursuing the a Ph.D. degree with The University of Texas at Dallas, Richardson, TX, USA, under the supervision of Prof. W. E. Wong. His current research interests include software fault localization, context-sensitive computing, and software risk analysis.

FRANZ WOTAWA received the M.Sc. degree in computer science and the Ph.D. degree from the Vienna University of Technology, in 1994 and 1996, respectively. He is currently a Professor of software engineering with the Graz University of Technology. He was the Head of the Institute for Software Technology, from 2003 to 2009. His research interests include model-based and qualitative reasoning, theorem proving, mobile robots, verification and validation, and software testing and debugging. Beside theoretical foundations, he has been always interested in closing the gap between research and practice. Since 2017, he has been the Head of the Christian Doppler Laboratory for Quality Assurance Methodologies for Autonomous Cyber-Physical Systems. During his career, he has written over 360 peer-reviewed papers in journals, books, conferences, and workshops. He has supervised 86 master's and 35 Ph.D. students. For his work on diagnosis, he has received the Lifetime Achievement Award from the International Diagnosis Community, in 2016. He has been a member of various number of program committees and organized several workshops and special issues of journals. He is a member of the Academia Europaea, the IEEE Computer Society, ACM, the Austrian Computer Society (OCG), and the Austrian Society for Artificial Intelligence, and a Senior Member of the AAAI.