Transfer Defect Learning Using Dependency Clustering

Mathieu Nayrolles, Abdelwahab Hamou-Lhadj
SBA Lab, ECE Dept, Concordia University
Montréal, QC, Canada
{mathieu.nayrolles, wahab.hamou-lhadj}@concordia.ca

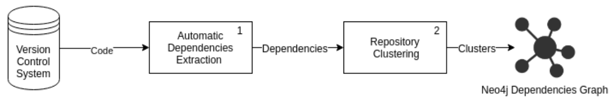

Fig. 1: Clustering by dependency

I. Introduction

Research in software maintenance has evolved over the years to include areas like mining bug repositories, bug analytic, and bug prevention and reproduction. The ultimate goal is to develop better techniques and tools to help software developers detect, correct, and prevent bugs in an effective and efficient manner.

One particular (and growing) line of research focuses on the problem of preventing the introduction of bugs by detecting risky commits (preferably before the commits reach the central repository). Recent approaches (e.g., [1], [2]) rely on training models based on code and process metrics (e.g., code complexity, experience of the developers, etc.) that are used to classify new commits as risky or not. Metrics, however, may vary from one project to another, hindering the reuse of these models. Consequently, these techniques tend to operate within single projects only, despite the fact that many large projects share dependencies, such as the reuse of common libraries. This makes them potentially vulnerable to similar faults.

Another advantage of BIANCA is that it uses commits that are used to fix previous defect-introducing commits to guide the developers on how to improve risky commits. This way, BIANCA goes one step further than existing techniques by providing developers with a potential fix for their risky commits.

We validated the performance of BIANCA on 42 open source projects, obtained from Github. The examined projects vary in size, domain and popularity.

II. The BIANCA Approach

A. Clustering Project Repositories

We cluster projects according to their dependencies. The rationale is that projects that share dependencies are most likely to contain defects caused by misuse of these dependencies. In this step, the project dependencies are analysed and saved into a single NoSQL graph database as shown in Figure 1. Graph databases use graph structures as a way to store and query information. In our case, a node corresponds to a project that is connected to other projects on which it depends. Project dependencies can be automatically retrieved if projects use a dependency manager such as Maven.

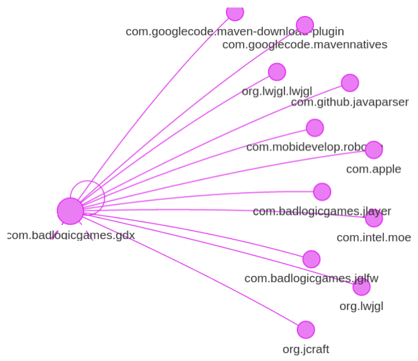

Fig. 2: Simplified Dependency Graph for `com.badlogicgames.gdx`

Figure 2 shows a simplified view of a dependency graph for a project named `com.badlogicgames.gdx`. As we can see, `com.badlogicgames.gdx` depends on projects owned by the same organization (i.e., badlogicgames) and other organizations such as Google, Apple, and Github.

Once the project dependency graph is extracted, we use a clustering algorithm to partition the graph. To this end, we choose the Girvan–Newman algorithm [3], [4], used to detect communities by progressively removing edges from the original network. Instead of trying to construct a measure that identifies the edges that are the most central to communities, the Girvan–Newman algorithm focuses on edges that are most likely "between" communities. This algorithm is very effective at discovering community structure in both computer-generated and real-world network data [4]. Other clustering algorithms can also be used.

B. Building a Database of Code Blocks of Defect-Commits and Fix-Commits

To build our database of code blocks that are related to defect-commits and fix-commits, we first need to identify the respective commits. Then, we extract the relevant blocks of code from the commits.

Extracting Commits: BIANCA listens to bug (or issue) closing events happening on the project tracking system. Every time an issue is closed, BIANCA retrieves the commit that was used to fix the issue (the fix-commit) as well as the one that introduced the defect (the defect-commit). Retrieving fix-commits, however, is known to be a challenging task [5]. This is because the link between the project tracking system and the code version control system is not always explicit. In an ideal situation, developers would add a reference to the issue they work on inside the description of the commit. However, this good practice is not always followed. To make the link between fix-commits and their related issues, we turn to a modified version of the back-end of commit-guru [6]. Commit-guru is a tool, developed by Rosen *et al.* [6] to detect *risky commits*. In order to identify risky commits, Commit-guru builds a statistical model using change metrics (i.e., amount of lines added, amount of lines deleted, amount of files modified, etc.) from past commits known to have introduced defects in the past.

Commit-guru's back-end has three major components: ingestion, analysis, and prediction. We reuse the ingestion and the analysis part for BIANCA. The ingestion component is responsible for ingesting (i.e., downloading) a given repository. Once the repository is entirely downloaded on a local server, each commit history is analysed. Commits are classified using the list of keywords proposed by Hindle *et al.* [7]. Commit-guru implements the SZZ algorithm [8] to detect risky changes, where it performs the SCM blame/annotate function on all the modified lines of code for their corresponding files on the fix-commit's parents. This returns the commits that previously modified these lines of code and are flagged as the defect introducing commits (i.e., the defect-commits). Prior work showed that commit-guru is effective in identifying defect-commits and their corresponding fixing commits [9] and the SZZ algorithm, used by commit-guru, is shown to be effective in detecting risky commits [6], [10]. Note that we could use a simpler and more established approach such as Relink [5] to link the commits to their issues and re-implement the classification proposed by Hindle *et al.* [7] on top of it. However, commit-guru has the advantage of being open-source, making it possible to modify it to fit our needs by fine-tuning its performance.

III. Case Study Setup

In this section, we present the setup of our case study in terms of repository selection, dependency analysis, comparison process and evaluation measures.

A. Project Repository Selection

To select the projects used to evaluate our approach, we followed three simple criteria. First, the projects need to be in Java and use Maven to manage dependencies. This way, we can automatically extract the dependencies and perform the clustering of projects. The second criterion is to have projects that enjoy a large community support and interest. We selected projects that have at least 2000 followers. Finally, the projects must have a public issue repository to be able to mine their past issues and the fixes. We queried Github with these criteria and retrieved 42 projects (see Table II for the list of projects), including those from some of major open-source contributors such as Alibaba, Apache Software Foundation, Eclipse, Facebook, Google and Square.

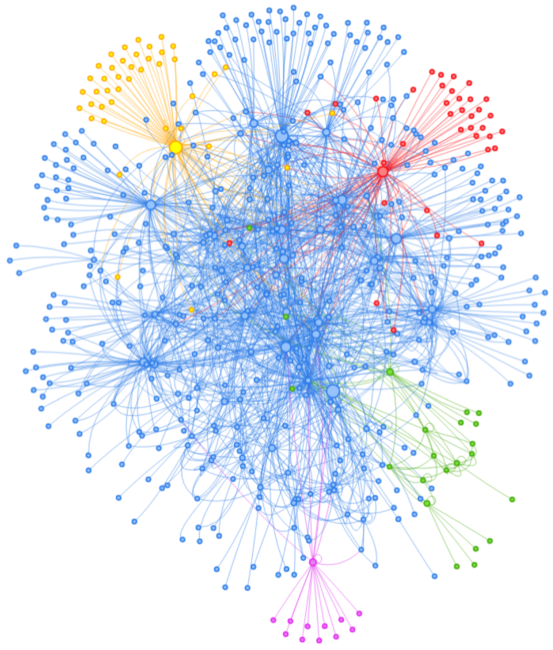
Fig. 3: Dependency Graph

B. Project Dependency Analysis

Figure 3 shows the project dependency graph. The dependency graph is composed of 592 nodes divided into five clusters shown in yellow, red, green, purple and blue. The size of the nodes in Figure 3 is proportional to the number of connections from and to the other nodes.

As shown in Figure 3, these Github projects are very much interconnected. On average, the projects composing our dataset have 77 dependencies. Among the 77 dependencies, on average, 62 dependencies are shared with at least one other project from our dataset.

Table I shows the result of the Girvan–Newman clustering algorithm in terms of centroids and betweenness. The blue cluster is dominated by Storm from The Apache Software Foundation. Storm is a distributed real-time computation system. Druid by Alibaba, the e-commerce company that provides consumer-to-consumer, business-to-consumer and business-to-business sales services via web portals, dominates the yellow cluster. In recent years, Alibaba has become an active member of the open-source community by making some of its projects publicly available. The red cluster has Hadoop by the Apache Software Foundation as its centroid. Hadoop is an open-source software framework for distributed storage and distributed processing of very large datasets on computer clusters built

TABLE I: Communities in terms of ID, Color code, Centroids, Betweenness and number of members

#ID	Community	Centroids	Betweenness	# Members
1	Blue	Storm	24,525	479
2	Yellow	Alibaba	24,400	42
3	Red	Hadoop	16,709	37
4	Green	Openhab	3,504	22
5	Purple	Libdx	6,839	12

from commodity hardware. The green cluster is dominated by the Persistence project of OpenHab. OpenHab proposes home automation solutions and the Persistence project is their data access layer. Finally, the purple cluster is dominated by Libdx by Badlogicgames, which is a cross-platform framework for game development.

A review of each cluster shows that this partitioning divides projects in terms of high-level functionalities. For example, the blue cluster is almost entirely composed of projects from the Apache Software Foundation. Projects from the Apache Software Foundation tend to build on top of one another. We also have the red cluster for Hadoop, which is by itself an ecosystem inside the Apache Software Foundation. Finally, we obtained a cluster for e-commerce applications (yellow), real-time network application for home automation (green), and game development (purple).

C. Building a Database of Defect-Commits and Fix-Commits for Performances Evaluation

To build the database that we can use to assess the performance of BIANCA, we use the same process as discussed in Section II-B. We used Commit-guru to retrieve the complete history of each project and label commits as defect-commits if they appear to be linked to a closed issue. The process used by Commit-guru to identify commits that introduce a defect is simple and reliable in terms of accuracy and computation time [10]. We use the commit-guru labels as the baseline to compute the precision and recall of BIANCA. Each time BIANCA classifies a commit as *risky*, we can check if the *risky* commit is in the database of defect-introducing commits. The same evaluation process is used by related studies [11]–[14].

D. Process of Comparing New Commits

Because our approach relies on commit pre-hooks to detect risky commits, we had to find a way to *replay* past commits. To do so, we *cloned* our test subjects, and then created a new branch called *BIANCA*. When created, this branch is reinitialized at the initial state of the project (the first commit) and each commit can be replayed as they have originally been. For each commit, we store the time taken for *BIANCA* to run, the number of detected clone pairs, and the commits that match the current commit. As an example, let's assume that we have three commits from two projects. At time t_1, commit c_1 in project p_1 introduces a defect. The defect is experienced by an user that reports it via an issue i_1 at t_2. A developer fixes the defect introduced by c_1 in commit c_2 and closes i_1 at t_3. From t_3 we known that c_1 introduced a defect using the process described in Section III-C. If at t_4, c_3 is pushed to p_2 and c_3 matches c_1 after preprocessing, pretty-printing and formatting, then c_3 is classified as *risky* by BIANCA and c_2 is proposed to the developer as a potential solution for the defect introduced in c_3.

E. Evaluation Measures

Similar to prior work focusing on risky commits (e.g., [10], [15]), we used precision, recall, and F-measure to evaluate our approach. They are computed using TP (true positives), FP (false positives), FN (false negatives), which are defined as follows:

- TP: is the number of defect-commits that were properly classified by BIANCA
- FP: is the number of healthy commits that were classified by BIANCA as risky
- FN: is the number of defect introducing-commits that were not detected by BIANCA
- Precision: TP / (TP + FP)
- Recall: TP / (TP + FN)
- F-measure: 2.(precision.recall)/(precision+recall)

It is worth mentioning that, in the case of defect prevention, false positives can be hard to identify as the defects could be in the code but not yet reported through a bug report (or issue). To address this, we did not include the last six months of history. Following similar studies [6], [16]–[18], if a defect is not reported within six months then it is not considered.

IV. Case Study Results

A. Baseline Classifier Comparison

Although our average F-measure of 52.72% may seem low at first glance, achieving a high F-measure for unbalanced data is very difficult [19]. Therefore, a common approach to ground detection results is to compare it to a simple baseline.

To the best of our knowledge, this is the first approach that relies on code similarity instead of code or process metrics for the detection of risky commits. Comparing it to other approaches will not be accurate. In addition, existing metric-based techniques (e.g., [2]) detect risky commits within single projects only. BIANCA, on the other hand, operates across projects. We compared BIANCA with a random classifier to have a baseline and show that we perform better than a simple baseline.

The baseline classifier first generates a random number n between 0 and 1 for the 165,912 commits composing our dataset. For each commit, if n is greater than 0.5, then the commit is classified as risky and vice versa. As expected by a random classifier, our implementation detected ~50% (82,384 commits) of the commits to be *risky*. It is worth mentioning that the random classifier achieved 24.9% precision, 49.96% recall and 33.24% F-measure. Since our data is unbalanced (i.e., there are many more *healthy* than *risky* commits) these numbers are to be expected for a random classifier. Indeed, the recall is very close to 50% since a commit can take on one of two classifications, risky or non-risky. While analysing the precision, however, we can see that the data is unbalanced (a random classifier would achieve a precision of 50% on a balanced dataset).

It is important to note that the purpose of this analysis is not to say that we outperform a simple random classifier, rather to shed light on the fact that our dataset is unbalanced and achieving an average F-= 52.72% is non-trivial, especially when a baseline only achieves an F-measure of 33.24%.

V. Discussion

In this section we propose a discussion on limitations and threats to validity.

A. Limitations

B. Threats to Validity

VI. Related Work

The work most related to ours come from two main areas, work that aims to predict future defects in files, modules and changes and work that aims to propose or generate patches for buggy software.

A. File, Module and Risky Change Prediction

The majority of previous file/module-level prediction work used code or process metrics. Approaches using code metrics only use information from the code itself and do not use any historical data. Chidamber and Kemerer published the well-known CK metrics suite [20] for object oriented designs and inspired Moha *et al.* to publish similar metrics for service-oriented programs [21]. Another famous metric suite for assessing the quality of a given software design is Briand's coupling metrics [22].

The CK and Briand's metrics suites have been used, for example, by Basili *et al.* [23], El Emam *et al.* [11], Subramanyam *et al.* [24] and Gyimothy *et al.* [25] for object-oriented designs. Service oriented designs have been far less studied than object oriented design as they are relatively new, but, Nayrolles *et al.* [26], [27], Demange *et al.* [28] and Palma *et al.* [29] used Moha *et al.* metric suites to detect software defects. All these approaches, proved software metrics to be useful at detecting software fault for object oriented and service oriented designs, respectively. More recently, Nagappan *et al.* [30], [31] and Zimmerman *et al.* [32], [33] further refined metrics-based detection by using statical analysis and call-graph analysis.

Other approaches use historical development data, often referred to as process metrics. Naggapan and Ball [34] studied the feasibility of using relative churn metrics to prediction buggy modules in the Windows Server 2003. Other work by Hassan *et al* and Ostrand *et al* used past changes and defects to predict buggy locations (e.g., [35], [36]). Hassan and Holt proposed an approach that highlights the top ten most susceptible locations to have a bug using heuristics based on file-level metrics [35]. They find that locations that have been recently modified and fixed locations are the most defect-prone. Similarly, Ostrand *et al.* [36] predict future crash location by combining the data from changed and past defect locations. They validate their approach on industrial systems at AT&T. They showed that data from prior changes and defects can effectively defect-prone locations for open-source and industrial systems. Kim *et al.* [37] proposed the bug cache approach, which is an improved technique over Hassan and Holt's approach [35]. Rahman and Devanbu found that, in general, process-based metrics perform as good as or better than code-based metrics [38].

Other work focused on the prediction of risky changes. Kim et al. proposed the change classification problem, which predicts whether a change is buggy or clean [15]. Hassan [39] used the entropy of changes to predict risky changes. They find that the more complex a change is, the more likely it is to introduce a defect. Kamei *et al.* performed a large-scale empirical study on change classification [10]. They aforementioned studies find that size of a change and the history of the files being changed (i.e., how buggy they were in the past) are the best indicators of risky changes.

Our work shares a similar goal to the work on the prediction of risky changes, however, BIANCA takes a different approach in that it leverages dependencies of a project to determine risky changes.

Our work differs from the work on automated patch generation in that we do not generate patches, rather we use clone detection to determine the similarity of a change to a previous risky change and suggest to the developer the fixes of the prior risky changes.

VII. Conclusion

In this paper, we presented BIANCA (Bug Insertion ANticipation by Clone Analysis at commit time), an approach that detects risky commits (i.e., a commit that is likely to introduce a bug) with an average of 90.75% precision and 37.15% recall. BIANCA uses clone detection techniques and project dependency analysis to detect risky commits within and across projects. BIANCA operates at commit-time, i.e., before the commits reach the code central repository. In addition, because it relies on code comparison, BIANCA does not only detect risky commits but also makes recommendations to developers on how to fix them. We believe that this makes BIANCA a practical approach for preventing bugs and proposing corrective measures that integrates well with the developers workflow through the commit mechanism.

To build on this work, we need to conduct a human study with developers in order to gather their feedback on the approach. The feedback obtained will help us fine-tune the approach. Also, we want to examine the relationship between project cluster measures (such as betweenness) and the performance of BIANCA. Finally, another improvement to BIANCA would be to support Type 4 clones.

VIII. Reproduction Package & Dataset

As described in Section III-D, we rely heavily on virtual machines instrumentation and coordination to run our experiments. Providing a straightforward reproduction package is therefore very challenging. However, we are happy to share our consolidated dataset: https://github.com/MathieuNls/tdl-data. The dataset is composed of three compressed PostgresSQL formatted tables: clones, commits and repository. The clone table stores the relationship between set of similar commits. The commits themselves are in the commit table with details about their author, repository, commit message and all the metrics found in commit guru [6]. Finally, the repository table describes the repository used in terms of url, name and ingestion status.

TABLE II: BIANCA results in terms of organization, project name, a short description, number of class, number of commits, number of defect introducing commits, number of risky commit detected, precision (%), recall (%), F-measure (%), the average similarity of first 3 and 5 proposed fixes with the actual fix and the average time difference between detected and original. "−" are reported when the size of the history was not sufficient to train a model.

Organization	Project Name	Short Description	NoC	#Commits	Bug Introducing Commit	Detected	Precision	Recall	F_1	Top 5 Fixes Similarity	Top 3 Fixes Similarity
Alibaba	druid	Database connection pool	3,309	4,775	1,260	787	88.44	62.46	73.21	39.97	46.69
	dubbo	RPC framework	1,715	1,836	119	61	96.72	51.26	67.01	60.01	57.14
	fastjson	JSON parser/generator	2,002	1,749	516	373	95.71	72.29	82.37	18.19	15.23
	jstorm	Stream Process	1,492	215	24	21	90.48	87.50	88.96	22.38	30.48
Apache	hadoop	Distributed processing	9,108	14,154	3,678	851	86.84	23.14	36.54	38.94	47.68
	storm	Realtime system	2,209	7,208	951	444	86.26	46.69	60.58	53.03	61.10
Clojure	clojure	Programming language	335	2,996	596	46	86.96	7.72	14.18	53.61	59.52
Dropwizard	dropwizard	RESTful web services	964	3,809	581	179	96.65	30.81	46.72	47.54	53.56
	metrics	JVM metrics	335	1,948	331	129	95.35	38.97	55.33	22.53	31.82
Eclipse	che	Eclipse IDE	7,818	1,826	169	9	88.89	5.33	10.05	31.01	39.04
Excilys	Android Annotations	Android Development	1,059	2,582	566	9	100.00	1.59	3.13	25.60	32.13
Facebook	fresco	Images Management	1,007	744	100	68	92.65	68.00	78.43	64.14	71.03
Gocd	gocd	Continuous Delivery server	16,735	3,875	499	297	91.58	59.52	72.15	21.62	30.59
Google	auto	source code generators	257	668	124	95	100.00	76.61	86.76	47.66	55.70
	guava	Google Libraries for Java 6+	1,731	3,581	973	592	98.48	60.84	75.22	23.74	23.59
	guice	Dependency injection	716	1,514	605	104	85.58	17.19	28.63	34.77	34.53
	iosched	Android App	1,088	129	9	6	100.00	66.67	80.00	16.50	24.97
Gradle	gradle	Build system	11,876	37,207	6,896	1,557	97.50	22.58	36.67	23.58	19.93
Jankotek	mapdb	Concurrent datastructures	267	1,913	691	440	94.32	63.68	76.03	63.16	72.48
Jhy	jsoup	Parser	136	917	254	153	87.58	60.24	71.38	46.41	44.59
Libdx	libgdx	Java game development	4,679	12,497	3,514	1,366	87.70	38.87	53.87	57.70	56.31
Netty	netty	Event-driven application	2,383	7,580	3,991	1,618	89.43	40.54	55.79	63.41	62.67
Openhab	openhab	Home Automation Bus	5,817	8,826	28	2	100.00	7.14	13.33	28.46	30.66
Openzipkin	zipkin	Distributed tracing system	397	799	176	73	87.67	41.48	56.31	55.92	51.90
Orfjackal	retrolambda	Backport of Java 8's lambda	171	447	97	35	94.29	36.08	52.19	34.69	42.06
OrientTechnologie	orientdb	Multi-Model DBMS	2,907	13,907	7,441	2,894	86.77	38.89	53.71	62.20	70.00
Perwendel	spark	Sinatra for java	205	703	125	82	97.56	65.60	78.45	21.88	28.00
PrestoDb	presto	Distributed SQL query	4,381	8,065	2,112	991	90.62	46.92	61.83	23.34	20.64
RoboGuice	roboguice	Google Guice on Android	1,193	1,053	229	70	91.43	30.57	45.82	53.81	56.55
Lombok	lombok	Additions to the Java language	1,146	1,872	560	212	91.98	37.86	53.64	58.94	57.49
Scribejava	scribejava	OAuth library	218	609	72	16	93.75	22.22	35.93	30.05	38.16
Square	dagger	Dependency injector	232	697	144	84	90.48	58.33	70.93	64.29	64.97
	javapoet	Java API	66	650	163	113	100.00	69.33	81.88	51.04	53.20
	okhttp	HTTP+HTTP/2 client	344	2,649	592	474	93.04	80.07	86.07	29.09	24.91
	okio	I/O API for Java	90	433	40	24	100.00	60.00	75.00	31.51	35.50
	otto	Guava-based event bus	84	201	15	15	93.33	100.00	96.55	54.11	49.94
	retrofit	Type-safe HTTP client	202	1,349	151	111	99.10	73.51	84.41	49.88	45.46
StephaneNicolas	robospice	Android library	461	865	113	39	87.18	34.51	49.45	60.90	65.04
ThinkAurelius	titan	Graph Database	2,015	4,434	1,634	527	90.13	32.25	47.51	48.64	50.59
Xetorthio	jedis	Redis client	203	1,370	295	226	92.04	76.61	83.62	25.69	29.45
Yahoo	anthelion	Plugin for Apache Nutch	1,620	7	0	-	-	-	-	-	-
Zxing	zxing	1D/2D barcode image	3,030	3,253	791	123	94.31	15.55	26.70	29.35	37.96
Total			**96,003**	**165,912**	**41,225**	**15316**	**90.75**	**37.15**	**52.72**	**40.78**	**44.17**

References

[1] D. Lo, "A Comparative Study of Supervised Learning Algorithms for Re-opened Bug Prediction," in *Proceedings of the european conference on software maintenance and reengineering*, 2013, pp. 331–334.

[2] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proceedings of the international conference on software engineering*, 2013, pp. 382–391.

[3] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. 12, pp. 7821–7826, Jun. 2002.

[4] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical Review E*, vol. 69, no. 2, p. 026113, Feb. 2004.

[5] R. Wu, H. Zhang, S. Kim, and S. Cheung, "Relink: recovering links between bugs and changes," in *Proceedings of the european conference on foundations of software engineering*, 2011, pp. 15–25.

[6] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: analytics and risk prediction of software commits," in *Proceedings of the joint meeting on foundations of software engineering*, 2015, pp. 966–969.

[7] A. Hindle, D. M. German, and R. Holt, "What do large commits tell us?: a taxonomical study of large commits." in *Proceedings of the international workshop on mining software repositories*, 2008, pp. 99–108.

[8] S. Kim, T. Zimmermann, K. Pan, and E. Jr. Whitehead, "Automatic Identification of Bug-Introducing Changes," in *Proceedings of the international conference on automated software engineering*, 2006, pp. 81–90.

[9] Y. Kamei, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, Jun. 2013.

[10] Y. Kamei, "Studying re-opened bugs in open source software," *IEEE Transactions on Software Engineering*, vol. 18, no. 5, pp. 1005–1042, 2013.

[11] K. El Emam, W. Melo, and J. C. Machado, "The prediction of faulty classes using object-oriented design metrics," *Journal of Systems and Software*, vol. 56, no. 1, pp. 63–75, Feb. 2001.

[12] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In, "Micro interaction metrics for defect prediction," in *Proceedings of the european conference on foundations of software engineering*, 2011, pp. 311–231.

[13] P. Bhattacharya and I. Neamtiu, "Bug-fix time prediction models: can we do better?" in *Proceeding of the iIternational conference on mining software repositories*, 2011, pp. 207–210.

[14] S. Kpodjedo, F. Ricca, P. Galinier, Y.-G. Guéhéneuc, and G. Antoniol, "Design evolution metrics for defect prediction in object oriented systems," *Empirical Software Engineering*, vol. 16, no. 1, pp. 141–175, Dec. 2010.

[15] S. Sunghun Kim, E. Whitehead, and Y. Yi Zhang, "Classifying Software Changes: Clean or Buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, Mar. 2008.

[16] T.-h. Chen, M. Nagappan, E. Shihab, and A. E. Hassan, "An Empirical Study of Dormant Bugs Categories and Subject Descriptors," in *Proceedings of the international conference on mining software repository*, 2014, pp. 82–91.

[17] S. Shivaji, S. Member, S. Member, R. Akella, and S. Kim, "Reducing Features to Improve Code Change-Based Bug Prediction," *IEEE Transactions on Software Engineering*, vol. 39, no. 4, pp. 552–569, 2013.

[18] Y. Kamei, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, Jun. 2013.

[19] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald, "Problems with Precision: A Response to‘ Comments on’Data Mining Static Code Attributes to Learn Defect Predictors"," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, p. 637, 2007.

[20] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[21] N. Moha, F. Palma, M. Nayrolles, and B. J. Conseil, "Specification and Detection of SOA Antipatterns," in *International conference on service oriented computing*, 2012, pp. 1–16.

[22] L. Briand, J. Daly, and J. Wust, "A unified framework for coupling measurement in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 25, no. 1, pp. 91–121, 1999.

[23] V. Basili, L. Briand, and W. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751–761, 1996.

[24] R. Subramanyam and M. Krishnan, "Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects," *IEEE Transactions on Software Engineering*, vol. 29, no. 4, pp. 297–310, Apr. 2003.

[25] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897–910, Oct. 2005.

[26] M. Nayrolles, A. Maiga, A. Hamou-lhadj, and A. Larsson, "A Taxonomy of Bugs : An Empircial Study," pp. 1–10.

[27] M. Nayrolles, N. Moha, and P. Valtchev, "Improving SOA Antipatterns Detection in Service Based Systems by Mining Execution Traces," in *Working conference on reverse engineering*, 2013, pp. 321–330.

[28] A. Demange, N. Moha, and G. Tremblay, "Detection of SOA Patterns," in *Proceedings of the international conference on service-oriented computing*, 2013, pp. 114–130.

[29] F. Palma, "Detection of SOA Antipatterns," PhD thesis, Ecole Polytechnique de Montreal, 2013.

[30] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proceedings of the international conference on software engineering*, 2005, pp. 580–586.

[31] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceeding of the international conference on software engineering*, 2006, pp. 452–461.

[32] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting Defects for Eclipse," in *Proceedings of the international workshop on predictor models in software engineering*, 2007, p. 9.

[33] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proceedings of the 13th*

international conference on software engineering - iCSE '08, 2008, p. 531.

[34] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the international conference on software engineering*, 2005, pp. 284–292.

[35] A. Hassan and R. Holt, "The top ten list: dynamic fault prediction," in *Proceedings of the international conference on software maintenance*, 2005, pp. 263–272.

[36] T. Ostrand, E. Weyuker, and R. Bell, "Predicting the location and number of faults in large software systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340–355, Apr. 2005.

[37] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting Faults from Cached History," in *Proceedings of the international conference on software engineering*, 2007, pp. 489–498.

[38] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *Proceedings of the international conference on software engineering*, 2013, pp. 432–441.

[39] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the international conference on software engineering*, 2009, pp. 78–88.