Mining Concurrency Bugs

Paolo Ciancarini*‡, Francesco Poggi*, Davide Rossi*, and Alberto Sillitti†‡
*University of Bologna, Italy
{paolo.ciancarini, francesco.poggi5, daviderossi}@unibo.it
†Center for Applied Software Engineering, Italy
alberto@case-research.it
‡Consorzio Interuniversitario Nazionale per l'Informatica, Italy

***Abstract*—Concurrent programming is becoming more and more popular due to the wide availability and low prices of multi-core CPUs. However, writing concurrent code is difficult and debugging it is even more complex due to a frequent non-deterministic behavior at run-time. Identifying pieces of code that are more prone to include concurrency bugs is important to focus testing activities and improve the overall code quality. Many research papers investigate approaches to predict the quality of the code but just a very limited amount address specifically concurrency problems in multi-core environments. This paper aims at investigating the use of bug mining techniques to link together concurrency-related code revisions with the corresponding issues to characterize concurrency bugs in complex code bases. Our approach is able to identify such links with high precision.**

***Keywords* – Concurrency, defects, bugs, empirical study, prediction model.**

I. Introduction

CPUs manufactures are now focusing on the development of technologies that are able to store more and more cores inside a single chip, while the computational power of a single core is almost constant or even decreasing. This is a revolution from the software point of view because, for the first time since the introduction of the microprocessor, running software on new hardware without any modification does not produce any performance increment. To exploit new CPUs, it is required to redesign the software to adapt it to multi-core, therefore redesign algorithms in a parallel way that can be executed by the different cores at the same time.

Concurrent code is known to be complex and debugging it means facing several challenges due to the non deterministic behavior of the code in some cases. This non deterministic behavior is also present in bug detection, making testing even more challenging. Testing code can be performed using very different approaches but all of them belongs to the families of the static (analyzing the source code without executing it) or dynamic (executing the code in a controlled environment with well known inputs) ones.

Therefore, identifying code that is more prone to include concurrency bugs is important to help developers in finding such bugs and focus the testing effort.

This aspect is even more important when developers have to deal with code developed by third-parties [28] in which there is a very limited knowledge of the details of the code and when analyzing the code provided is a preliminary step to integre it in the code base [12] [20].

The paper investigates the evolution of the source code to identify code that is potentially affected by concurrency problems. This is a preliminary study aimed at analyzing the characteristics of concurrency bugs in relation to source code revisions to help developers in the identification of concurrency-related pieces of code.

The paper is organized as follows: Section II briefly presents the state of the art; Section III analyzes the motivation of our study; Section IV introduces our approach and its application in a case study; Section V analyzes our findings; Section VI investigates the main validity threats of our study; finally, Section VII draws the conclusions and presents future work.

II. Related work

The amount of literature about bugs analysis and prediction is huge. However, in this study we focus on a specific subset of bugs that are related to the concurrency aspects of the code. This kind of bugs is more difficult to study due to the relative low frequency of such bugs and the difficulty of a proper identification.

Many studies in this area deal with the detection [19] [27] and prediction of bugs [5] [15] [25] [37]. Some studies focus on specific kinds of concurrency bugs such as data race [7] [23] [31] [36], atomicity violations [9], or deadlocks [3] [7]. Other studies are more comprehensive and try to include all categories of concurrency bugs [8] [18].

Some additional studies have been conducted on the propagation of concurrency bugs in the source code [26] [33].

Another important aspect in this kind of research is the identification of links between the source code (in particular the specific code revisions) and the issues in the issue tracking system to locate in a precise way the bug in the code [2][35]. Several techniques have been developed with different characteristics such as patter matching techniques [32], machine learning techniques [24], and mixed ones [29] [30].

In [2] the authors introduce the problem of linking the code revisions with the issues reported since the explicit link between the two system is often missing. They propose an approach that heavily relies on manual effort to achieve a high level of reliability. However, this approach cannot scale to large code bases involving thousands of issues and code revisions.

In [35] the authors present an approach to link issues and code revisions based on the analysis of the text provided by

developers and heuristics based on keyword matching such as “bug”, “fix”, etc. Moreover, they improved the approach enhancing the analysis using additional features such as: time frames in which the bugs and issues are reported, bug ownership and identity of the code committer, text similarity between the issue and the change logs, etc. In this way they achieved a 89% precision and 78% recall in linking issues and code revisions.

In [32] the authors analyze the introduction of bugs inside software systems and try to link them to bug reports to characterize the bug introduction process. The identification of the links between the code and the issues is based on pattern matching.

In [24] the authors propose a multi-layered machine learning approach to the problem of linking issues and source code based on the integration of several simpler approaches already available.

In [29] and [30] the authors uses a multi-technique approach to link bugs and issues in large repositories. The techniques used are based on two data mining approaches and the approach based on pattern matching developed in [32].

III. Motivation

As evidenced in the previous section, the analysis, identification, and prediction of bugs are well-studied areas of the software engineering discipline. However, many of the results obtained are hardly replicated in other studies for several reasons including:

- *Limited availability of the datasets:* many studies rely on proprietary information that is not available outside the company in which the study is performed. Therefore, it is impossible for other researchers to replicate the studies and verify the applicability of the developed approaches to different contexts. To cope with this problem, the recent trend for research in this area is to use open source projects to develop models and approaches. However, the structure of open source projects is often different from the structure of corporate projects where there are hierarchical structures and often standards and certification issues.
- *Limited availability of the tools:* many studies rely on ad-hoc tools developed to perform data collection and analysis. Even if such tools are often released to the public as open source software, they were developed as research prototypes and they were not designed to be used or adapted to different contexts. Therefore, in many cases, reusing such tools to perform studies in different environments become nearly impossible and a complete re-development is needed.
- *Limited external validity:* due to the previous limitations, many studies suffer from an unknown external validity due to the practical limitations in replicating the studies in different contexts. The approach adopted in most of the research studies are basically the replication of the studies in almost only open source environments. As state previously, this approach is not enough to guarantee the applicability in corporate settings.
- *Almost no industrial impact:* due to the lack of external validity of the studies, the application of the developed approaches to industrial settings is almost inexistent. Therefore, there are no industrial-quality tools and models that can be actually used in real development settings. At present, almost of the research effort in this area has not found a real impact on the software industry.

We acknowledge all the limitations listed above and we aim at developing tools and approaches that are easily adaptable in different contexts. The availability of customizable tools is a starting point for improving the development of reusable models in the area of bugs analysis and prediction. To do that, we have started developing a data extraction framework that is plug-in based and designed to be extensible. The main data sources we consider are (Figure 1): a) version control systems; b) issue tracking systems. The data are then stored locally to speed-up the analysis through different statistical and machine learning algorithms that are under investigation.

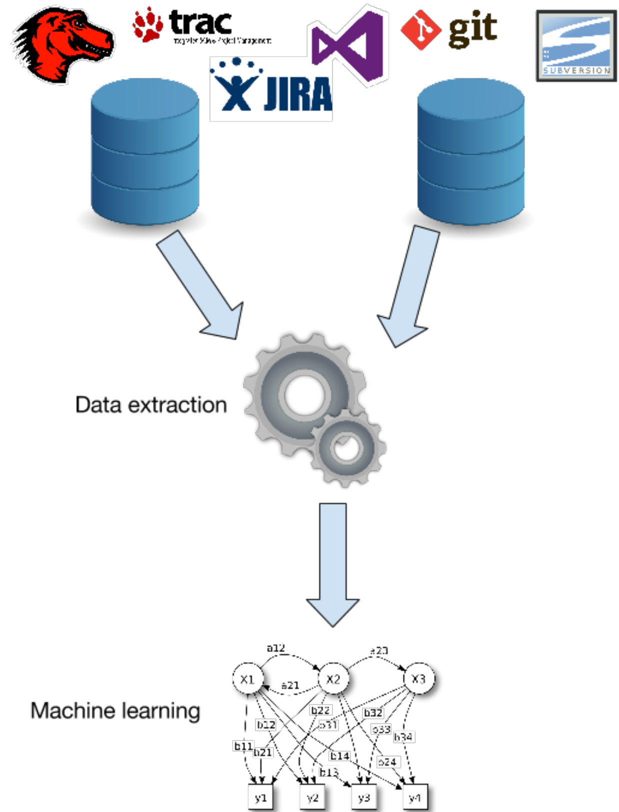

Fig. 1. Overview of the data extraction and analysis architecture.

Using the above outlined system, we have started to investigate a specific subset of bugs: the ones related to the concurrency aspects of the code.

Concurrency bugs are hard to detect (being a specific case of the heisenbugs class [4]) and fixing them is usually more

expensive than other classes of errors. In the last few years several approaches have been developed to limit the impact of concurrency-related bugs, all with limited success. Many authors agree that solutions integrating different approaches are probably the most promising [30]. But all solutions, whether based on prediction [37], static code analysis [7] [11] [1] [34] [22], dynamic code analysis [6] [13] [21] [10], manual characterization [8] [18] or pattern matching [17] need erroneous code to train their models, refine their patterns and improve their taxonomies. Generally speaking we can say that improving our knowledge on what goes wrong with concurrency management can lead to better code ("know your enemy" as per Sun Tzu in The Art of War).

However, concurrency bugs are (relatively) rare. We performed an analysis on the Apache HTTPD project [1] to assess the frequency of these bugs with respect to all other categories. Starting from 3,052 linked bugs [2] (i.e., bugs for which a reported issue can be liked to a code revision fixing it, more details on the procedure we used can be found later in the paper) we sampled 385 of them and checked manually whether they were concurrency-related. The result of this analysis shows that 4.16% of all reported bugs are expected to be concurrency-related with a 95% confidence interval and 5% error margin.

We also used the HTTPD code base to assess the effort needed to solve these bugs. Several different metrics can be used to characterize the effort, we focused our analysis on two parameters that can be extracted with software repositories mining techniques:

1) Number of people involved in its definition and fixing (counted as number of people participating in the discussion of the bug in the bug tracking system).
2) Number of comments per bug in the bug tracking system.

We extracted these values from a sample composed by 103 concurrency-related and 821 non concurrency-related bugs (the classification has been performed manually). The results we obtained are summarized in Table I.

TABLE I
SOME PROCESS METRICS FOR CONCURRENCY-RELATED AND NON CONCURRENCY-RELATED BUGS.

	Average persons	**Average comments**
Concurrency	4.77	11.08
Non-concurrency	3.76	6.78

On average concurrency bugs involve 4.77 persons while other bugs involve 3.76 persons; 11.08 comments on average appear on the bug tracking system for concurrency-related bugs while 6.78 comments appear for other bugs.

This is just a very basic analysis that does not provide a comprehensive overview of the characterization of the concurrency and non concurrency related bugs but it is useful to understand that the two kinds of bugs are deeply different from several point of view.

Given this context a method that is able to reliably extract code with concurrency-related problems can be a valuable asset.

Existing methods based uniquely on (static) code analysis are available and are known to be effective [14] but are often limited to specific programming languages and tend to identify only specific error patterns, so we decided to define an approach based on software repositories mining.

We initially tried to identify such method by characterizing a bug based on its description and the associated discussion in the bug tracking system. By using pattern matching techniques we selected 948 relevant issues. A manual check with a sample of 543 of these issues (giving a 95% confidence interval and 5% error margin) showed a very high recall (97.09%) but a limited precision: 18.73%. A study of the false positives showed what words such as "lock", "concurrent", and "atomic" are often used to refer to aspects of the development process rather than to aspects of the code to be fixed. While more sophisticated text analysis methods can probably improve this approach, it was quite obvious that high precision was out of reach, this led to the decision to design an approach based on both issue reports and code analysis.

IV. OUR APPROACH

Our approach is based on an analysis that take advantage of information coming from two different sources: code analysis and issue reports. Code analysis is used to identify concurrent code based on the primitives used inside the code that are concurrency-related. Issue reports are used to identity erroneous code from the description of the issue in the issue tracking system.

We follow the intuition that the modification to the code that leads to closing an issue as *fixed* is a bug fix. This implies then the version of the code before this modification contains a bug. To take advantage of that, we need to tie together issue reports and code revisions. This is a problem that has been originally posed by [2] and is subject of some more recent research work [35] as discussed in Section II.

We operated in a similar way finding links from issue reports (specifically, from the discussion associated to the report) to revisions and from commit comments to bug reports. As an example, consider the following excerpt of an issue report taken from the Bugzilla[2] bug tracking system as deployed for the Apache HTTP Server project[3]:

[1]https://httpd.apache.org/

[2]http://www.bugzilla.org

[3]https://bz.apache.org/bugzilla/

Bug 43359:
trunk EventMPM's graceful restart/stop are not graceful
Description:
I tested rev 574843 Event MPM.
While downloading a file, a graceful restart/stop kill a connection immediately.
I tested rev 327944 (before async write was merged) too, and no problem.
Discussion:
...
Comment 12:
Committed in r1137182, thanks for the patch

Since bug 43359 is marked as closed, we can assume that revision 1137182 contains a bug fix. Therefore, the previous revision contains a bug.

Now consider the following revision commit taken from the SVN[4] version control system as deployed for the very same Apache HTTP Server project [5]:

Revision 574462
Don't call into get_worker while holding the timeout_list mutex.
PR: 42031

From this text we can infer that revision 574462 has been created to fix issue 42031, so we can assume that the previous revision contained erroneous code (which is obviously not the case for all revisions, since some of them are created to introduce new features or to modify existing, logically correct, ones).

Notice that, while developers should keep track of the links between issues and revisions, several links are missing: many issues are closed with no indication of the revision in which the fix is introduced and many commit logs do not contain references to the issue they fix. This implies not only that many existing links cannot be retrieved but also that several of the retrieved one are unidirectional (i.e., we either have an issue referencing a revision or a revision referencing an issue but we do not have both); it should also be considered that a revision can fix several bugs and that the same bugs can be fixed in several revisions (even if this last case is relatively rare).

Moreover, the format used to refer to issue or revisions is not standardize and usually changes between developers and may have very different styles such as:

- "fix for bug 12345"
- "PR12345"
- "PR: 12345"
- "closes issue 12345"
- "fixed in 54321"
- "fixes I54321"
- etc.

[4]https://subversion.apache.org/
[5]https://svn.apache.org/

This last problem can be faced using some text analysis heuristics, existing literature presents approaches based on relatively simple pattern matching [32] or to more advanced multi-layer techniques adopting machine learning [24] as described in Section II.

Since our aim is to develop a method with a high precision, we decided to adopt a strategy based on pattern matching that is known to achieve high precision while sacrificing a bit of recall [32].

We tested our approach on the HTTP Server and Portable Runtime (APR) Apache projects. By analyzing 35,087 issues (having status "fixed") and 51,995 revisions from 2002 to 2015, we have been able to find 4,491 links between the two groups (this figure is consistent with those found in literature).

We then retrieved the code that was changed for all the revisions and analyzed it to infer if it was related to concurrency management. We also factored out of our analysis links associated to large code changes because we realized that this was usually a wrong categorization of the related issue. Even if these cases are described as bug reports, they are actually a new feature request, therefore out of the scope of our analysis.

To perform this analysis we adopted a pragmatic approach: to manage concurrency, the project considered uses a set of functions defined in APR, we marked as concurrency-related all code containing calls to these functions. This resulted in a set of 15 code changes, and thus to 15 code fragments (those changed by the revision) potentially containing concurrency-related bugs. A manual verification of this code fragments showed that this was indeed the case for 13 of them, leading to an overall precision for our method of 87%.

While we did not perform an exhaustive analysis to access the recall for the method (that, for the aforementioned considerations, is not as important as precision in our scope) we can observe that, as previously shown in our analysis of a sample of the issues database, we should expect 4.16% (with a 95% confidence interval and 5% error margin) of the issues to be related to concurrency problems. Given that we applied our method to a population of 35,087 issues and identified 15 of them as concurrency related we can expect its recall to be about 1/10th.

V. Discussion

Our method is able to extract concurrency bugs in large codebases with a high level of precision. However, given the peculiar characteristics of concurrent code, high precision and high recall are probably not achievable in this context.

This is a first attempt, several aspects of the proposed approach can be improved: linking algorithm, code analysis, etc. The fact that we obtained promising results by using a simple approach confirms that we are heading in the right direction.

VI. Threats to validity

The design and the execution of this study have have been performed under several assumptions and there are several

limitations that can limit the validity of the study and the related analysis. In particular, we need to consider the following aspects:

- This is a preliminary study only based on the Apache HTTP Server. Even if we already have investigated some other projects to cross-check the validity of the analysis performed, the investigation of such additional data sets is not complete and could not be reported in this study. In any case, different project may present different properties of the concurrency bugs.
- We are dealing with small numbers so the statistical significance of some of our analyses can be limited
- There could be some mistakes in the manual identification of the concurrency bugs. However, to mitigate the risk, the manual check was performed by at least two people independently.
- Having to deal with a small number of instances and small recall figures (which implies several potential false negatives) can result in a bias that could effect classification algorithms as discussed in [16]. Further study in needed to determine is this problem can effect also our specific domain. Moreover: analytic and pattern detection approaches to characterize concurrent bugs are not exposed to the same problems of machine learning approaches.
- The software analyzer we have developed may also include some bugs that prevent the identification of some relevant bugs.

VII. Conclusions and future work

The paper has presented a first analysis of the characterization of concurrency-related bugs compared to non concurrency-related ones. There are several aspects of the developed approach that can be improved but the first set of results are already useful for the implementation of a basic analysis tool. Such tool is based on a plug-in architecture that can be easily extended to support different issue tracking systems and different version control systems. The analysis phase can also be automated and customized but it requires more investigation to completely automate the process.

Future work will include the integration of the source code analysis with the analysis of the linked issues to improve the level of reliability of the proposed approach, especially the recall. Moreover, a more advance analysis of the source code will be implemented using the code metrics proposed in [5]. Finally, based on such description models, a prediction model able to help developers in the identification of not already detected bugs will be developed. From the point of view of the supporting tools, additional plug-ins to support different systems will be developed. Finally, the analysis will be automated as well.

Acknowledgment

The research presented in this paper has been partially funded by the ARTEMIS project EMC2 (621429).

References

[1] C. Artho, K. Havelund, K., "Applying Jlint to space exploration software.", Verification, Model Checking, and Abstract Interpretation (pp. 297-308). Springer Berlin Heidelberg, 2004.

[2] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, A. Bernstein, "The missing links: bugs and bug-fix commits." 18th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE), 2010.

[3] C. Boyapati, R. Lee, M. Rinard, "Ownership types for safe programming: Preventing data races and deadlocks", 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '02), 2002.

[4] G. Carrozza, D. Cotroneo, R. Natella, R. Pietrantuono, S. Russo, "Analysis and Prediction of Mandelbugs in an Industrial Software System", IEEE 6th International Conference on Software Testing, Verification and Validation (ICST 2013), 2013.

[5] P. Ciancarini, F. Poggi, D. Rossi, A. Sillitti, "Improving bug predictions in multi-core cyber-physical systems", 4th International Conference on Software Engineering for Defense Applications (SEDA 2015), Rome, Italy, 26 -. 27 May 2015.

[6] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, S. Ur, "Multithreaded Java program test generation.", IBM systems journal, 41(1), 2002.

[7] D. Engler, K. Ashcraft, "RacerX: effective, static detection of race conditions and deadlocks", 19th ACM symposium on Operating systems principles (SOSP '03), 2003.

[8] E. Farchi, Y. Nir, S. Ur, "Concurrent bug patterns and how to test them", Parallel and Distributed Processing Symposium, 2003.

[9] C. Flanagan, S. N. Freund, "Atomizer: a dynamic atomicity checker for multithreaded programs", 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '04), 2004.

[10] K. Havelund, T. Pressburger, "Model checking java programs using java pathfinder.", International Journal on Software Tools for Technology Transfer, 2(4), 2000.

[11] D. Hovemeyer, W. Pugh, "Finding bugs is easy.", ACM Sigplan Notices, 2004.

[12] A. Jermakovics, A. Sillitti, G. Succi, "Mining and Visualizing Developer Networks from Version Control Systems", 4th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE 2011) at ICSE 2011, Honolulu, HI, USA, 21 May 2011.

[13] P. Joshi, M. Naik, C.S. Park, K. Sen, "CalFuzzer: An extensible active testing framework for concurrent programs.", Computer Aided Verification ,Springer Berlin Heidelberg, 2009.

[14] D. Kester, M. Mwebesa, J.S. Bradbury, "How Good is Static Analysis at Finding Concurrency Bugs?,", 10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM), 2010.

[15] S. Kim, T. Zimmermann, E. J. Whitehead, A. Zeller, "Predicting Faults from Cached History", 29th international conference on Software Engineering (ICSE '07), 2007

[16] S. Kim, H. Zhang, R. Wu, L. Gong, "Dealing with noise in defect prediction", 33rd International Conference on Software Engineering (ICSE), 2011

[17] D. Lo, H. Cheng, J. Han, S.C. Khoo, C. Sun, C., "Classification of software behaviors for failure detection: a discriminative pattern mining approach.", 15th ACM SIGKDD international conference on Knowledge discovery and data mining, 2009.

[18] S. Lu, S. Park, E. Seo, Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics", ACM Sigplan Notices, vol. 43, pp. 329–339, 2008.

[19] A. H. Moin, M. Khansari, "Bug Localization Using Revision Log Analysis and Open Bug Repository Text Categorization", 6th International IFIP WG 2.13 Conference on Open Source Systems, OSS 2010, Notre Dame, IN, USA, May 30 – June 2, 2010.

[20] R. Moser, W. Pedrycz, A. Sillitti, G. Succi, "A model to identify refactoring effort during maintenance by mining source code repositories", 9th International Conference on Product Focused Software Process Improvement (PROFES 2008), Frascati (Rome), Italy, 23 - 25 June 2008.

[21] M. Musuvathi, M., "Systematic concurrency testing using CHESS.", Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging, 2008.

[22] M. Naik, C.S. Park, K. Sen, D. Gay, "Effective static deadlock detection.", 31st International Conference on Software Engineering, 2009.

[23] R. H. B. Netzer, B. P. Miller, "Improving the accuracy of data race detection", 3rd ACM SIGPLAN symposium on Principles and practice of parallel programming (PPOPP '91), 1991.

[24] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, T. N. Nguyen, “Multi-layered approach for recovering links between bug reports and fixes.”, 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE), 2012.

[25] T. J. Ostrand, E. J. Weyuker, R. M. Bell, “Predicting the location and number of faults in large software systems”, IEEE Transactions on Software Engineering, Vol. 31, No.4, pp. 340 – 355, April 2005.

[26] W. F. Pan, B. Li, Y. T. Ma, Y. Y. Qin, X. Y. Zhou, “Measuring structural quality of object-oriented softwares via bug propagation analysis on weighted software networks”, Journal of Computer Science and Technology, vol. 25, no. 6, pp. 1202–1213, 2010.

[27] S. Rao, A. Kak, “Retrieval from software libraries for bug localization: a comparative study of generic and composite text models”, 8th Working Conference on Mining Software Repositories (MSR ’11), 2011.

[28] T. Remencius, A. Sillitti, G. Succi, “Assessment of software developed by a third-party: A case study and comparison”, Information Sciences, Elsevier, Vol. 328, pp. 237 – 249, January 2016.

[29] B. A. Romo, A. Capiluppi, T. Hall, “Filling the Gaps of Development Logs and Bug Issue Data”, The International Symposium on Open Collaboration (OpenSym ’14), 2014.

[30] B. A. Romo, A. Capiluppi, “Towards an automation of the traceability of bugs from development logs: a study based on open source software”, 19th International Conference on Evaluation and Assessment in Software Engineering (EASE ’15), 2015.

[31] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T. Anderson, “Eraser: A dynamic data race detector for multithreaded programs”, ACM Transactions on Computer Systems, 15(4), 1997.

[32] J. Sliwerski, T. Zimmermann, A. Zeller, “When do changes induce fixes?”, ACM SIGSOFT Software Engineering Notes (Vol. 30, No. 4, pp. 1-5). 2005

[33] L. Voineaand, A.Telea, “How do changes in buggy Mozilla files propagate?”, ACM symposium on Software visualization, 2006.

[34] J.W. Voung, R. Jhala, S. Lerner, “RELAY: static race detection on millions of lines of code.”, 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, 2007.

[35] R. Wu, H. Zhang, S. Kim, S.C. Cheung, “ReLink: Recovering Links between Bugs and Changes”, 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE ’11), 2011.

[36] Y. Yu, T. Rodeheffer, W. Chen, “RaceTrack: efficient detection of data race conditions via adaptive tracking”, 20th ACM symposium on Operating systems principles (SOSP ’05), 2005.

[37] B. Zhou, I. Neamtiu, R. Gupta, “Predicting concurrency bugs: how many, what kind and where are they?”, 19th International Conference on Evaluation and Assessment in Software Engineering (EASE ’15), 2015.