Proceedings of the 10th International Conference on Applied Informatics
Eger, Hungary, January 30–February 1, 2017. pp. 111–119
doi: 10.14794/ICAI.10.2017.111

An open-source solution for automatic bug database creation

Péter Gyimesi

Department of Software Engineering, University of Szeged
pgyimesi@inf.u-szeged.hu

Abstract

In the field of Software Engineering, a recurring theme is that locating errors and trying to fix them in a software package. Several studies on software bugs have used some kind of bug database to assist their work. These databases usually come from publicly available sources, but some researchers have created their own databases. These kinds of studies are important because the more we know about software bugs, the easier it is to prevent or find and fix them. Software developers being human tend to make mistakes. These may arise due to such factors as a tight deadline, poor design, a change in the specification and lack of experience. This is the reason why it is necessary to support this kind of research with bug databases that represent the defects of real software systems well. Depending on the application of the database, it may contain additional information like bug-related test cases, static source code metrics, design patterns and process metrics.

During our previous study we constructed a bug database from Java projects on GitHub. This database contains the faulty source code elements (files and classes) and their static source code metrics. In addition, we used a graph database to compute some process metrics of the buggy source code elements.

In this study, we present our BugHunter tool and we improved it by using the graph database to also locate the buggy source code elements. This way, our tool is capable of locating the defective source code elements at the file and class levels, and now even at the method level. Our tool uses Neo4j - a popular open-source graph database engine - and its query language called cypher. The calculations are carried out by running a cypher query, and it makes our method quite flexible. To analyze the given project's source code, we used the free version of the SourceMeter tool. Also, we made our complete tool-chain publicly available as an open-source project on GitHub. This way, the complete system is freely available for studies on software bugs, especially those concerning bug prediction.

https://github.com/sed-szeged/BugHunterToolchain

Keywords: Bug database, Graph database, Open-source tool

MSC: 68N30

1. Introduction

Managing software bugs is an essential part of software development. It consists of tasks like preventing, finding and fixing bugs. Finding software bugs is usually done by checking the source code manually and looking for the root of the problem based on bug reports. It is a time- and resource-consuming activity, and minimizing the required effort would help to reduce the cost of the development. Unit tests also can help one to detect and localize software faults, but it requires writing test cases in parallel to development and it is also a resource-intensive task.

Another way of assisting bug localization is to characterize the known ones with some appropriate metrics and try to predict which source code elements have the highest probability of containing a bug. Analyzing known bugs requires a source code change history and a bug tracking system. Nowadays, many developers use a versioning system - like Subversion or Git -, hence the source code history is often available. The use of bug tracking systems is also quite common in software development. The bug reports are recorded within these systems and all changes related to the bugs are also tracked, including the source code fixes. If we link these fixes to the source code history, we can locate the source code of the known bugs.

The characterization of these bugs can be carried out for example with static source code metrics or with software process metrics. In a previous study [1], we presented a method for collecting bug information from the history of Java projects on GitHub. The data collected includes the number of bugs in a certain source code element (class or file). This method involves an extra dimension that was never used before, which is called time. It also characterizes these source code elements with static source code metrics and it produces bug databases from the buggy and the fixed state of the source code elements. In the next study [2], we modified our method to build conventional bug databases for 105 selected versions of 15 Java projects on GitHub. Then we used this dataset to assess the bug prediction capabilities of the static source code metrics. We published the databases and the results we obtained were included in an online appendix[1].

In another study [3], we designed a method for building a graph database from the source code change history. We used this graph to compute the most common software process metrics that we found in the literature. These metrics are calculated for files and classes, and even for methods. Continuing this study [4], we assessed the usefulness of the software process metrics in bug prediction. We used bug information of the class and file levels that we published earlier, and we also compared the two sets of predictors (namely, product metrics and process metrics)

[1] `http://www.inf.u-szeged.hu/~ferenc/papers/GitHubBugDataSet/`

against each other. We made our graph tool available on GitHub and we also published the databases and the results in an online appendix[2].

Ranking source code methods based on bug-proneness could be very useful for developers to help them to find bugs more efficiently; however, bug databases that contain bug information at the method level is quite rare. To address this issue, in this article we present our method that we developed and it is now able to compute bug numbers even at the method level. Furthermore, we made our complete toolchain publicly available as an open-source project in the following GitHub repository:
`https://github.com/sed-szeged/BugHunterToolchain`

The remainder of the paper is organized as follows. In Section 2, we summarize some related work on bug localization. Then in Section 3, we present our improved method and the toolchain that we made publicly accessible. Lastly, in Section 4, we draw some pertinent conclusions and suggest some plans for future work.

2. Related work

There are many studies about mining source code repositories [5, 6, 7]. In one, Williams et al. [5] made a study on using the source code change history to refine the results of a static source code checker. They implemented a function return value checker and collected a list of functions from the change history that were involved in bug fixes where a return value check has been added after the function call. Their method also collects the return checks from the current version of the source code and it ranks the warnings of their function return value checker based on the occurrence rate of these checks. They made two case studies and found a total of 724 warnings, meaning 724 potential bugs in the source code of Apache Web Server and Wine.

Wang et al. [6] studied the bug-proneness of overloaded and overridden methods. They made a tool called Rebug-Detector which is capable of detecting potential bugs based on similar faulty overridden or overloaded methods. By evaluating their tool on Apache Lucene-Java, they detected 21 real bugs and 10 suspected bugs.

In most of these studies, they exploit certain connections between source code changes and bug reports. Wu et al. [7] made a tool called ReLink to recover missing links between fixed bugs and the committed changes. This tool uses natural language processing techniques to restore these links. For our study, we used the linkage obtained from the GitHub data.

There are existing bug databases [8, 9, 10, 11] that were intended for studies on software bugs. D'Ambros et al. [8] made a benchmark for defect prediction which consists of five software systems. This dataset includes bug numbers and various metrics at the class level.

The tera-PROMISE [11] repository is the biggest maintained dataset of software engineering research databases. It contains, among other things, class-level bug

[2] `http://www.inf.u-szeged.hu/~pgyimesi/papers/ActaCybernetica2016/`

databases. iBUGS [9, 10] is a bug localization benchmark made by Dallmeier et al. Their approach semi-automatically extracts bug information at the class level from the source code history. Furthermore, they made a case study using their dynamic bug localization tool to rank the classes based on failing test runs.

There are various ways of ranking the potentially faulty source code elements. Zhou et al. [12] used the textual similarity between the new bug report and the source code to recommend files that are likely to contain the bug.

In contrast to the above studies, we would like to produce bug numbers at the method level, which can be used to rank methods based on bug-proneness.

3. Methodology

In our previous papers [3, 4], we published a graph schema for calculating software process metrics. Now we will briefly describe it together with the minor modifications that we made to calculate bug numbers. The schema is shown in Figure 1 and it only contains the elements used for this current study. It has five types of nodes and a single graph is created for a whole project.

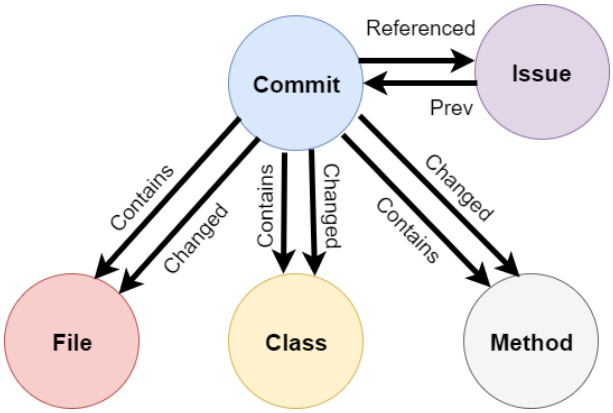

Figure 1: The graph schema

The *Commit* node represents a modification of a project and it has two attributes, namely *hash* and *created*. The former is the Git identifier of the commit, while the latter is the time stamp of the commit's creation. This node has three types of outgoing edges, these being *Contains*, *Changed* and *Referenced*. The first two point to source code elements (files, classes and methods). The *Contains* edges indicate whether a certain source code element is present in the source code after the modification. The *Changed* edges indicate whether a certain source code element was modified during the commit. If a certain modification was made in order to fix a bug, then the appropriate *Commit* node has a *Referenced* edge that points to an *Issue* node.

The *Issue* node represents a bug that was reported to the issue tracking system and it was later fixed. It has two attributes, namely *opened* and *closed*. This first one is the date of the bug report, and the second is the date of the final fix. It has an outgoing edge called *prev* which points to a *Commit* node that the issue was assigned to. For more details, see Section 3.1.

The last three nodes represent the source code elements, namely *File*, *Class* and *Method*. These nodes have two attributes, these being *name* and *filtered*. The *name* is the complete name of the element, including the name of the package, class, method and the type of the return value and parameters. The *filtered* attribute is a flag that indicates whether a certain source code element should be left out from a computation e.g., in the case of test-related elements.

The definitions of additional graph parts which we did not use in this study can be found in our previous paper [4]. We will now proceed to the bug number calculation.

3.1. Computing bug numbers

To build a bug database, first of all we have to select a software version that we want to build the database from. An often used approach is to choose a release version. In this study, we process the whole history of a project, thus we handle more than one version. Figure 2 depicts the relationship between the fixed bugs and the selected commits. We assign each bug to a selected version that is the latest one before them. For example, we assign Bug 1 and Bug 2 to Commit A, Bug 3 to Commit B, and we do not assign any bug to Commit C. These assignments are represented in the graph by the *prev* edges. It may happen that a certain bug is reported before a given version but it is fixed later - for example, in the case of Commit B and Bug 2. In such cases, we partially mark the buggy source code elements in this version as well.

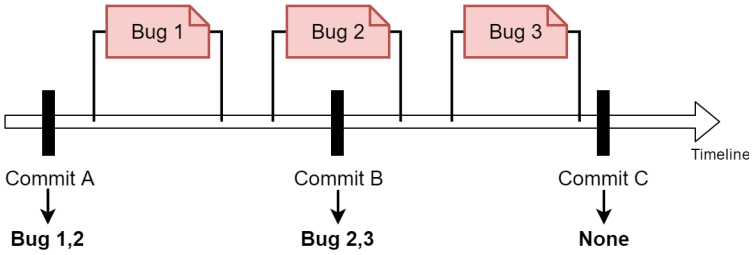

Figure 2: The relationship between bugs and commits

We formulated the calculation into a *cypher* query. This query language is used by Neo4j (see Section 3.2). The query to compute method-level bug numbers is the following:

```
match
  (n:METHOD{filtered:false})<-[:CONTAINS]-(c:COMMIT{hash:'...'}),
```

```
  (n)<-[:CHANGED]-(cc:COMMIT)-[:REFERENCED]->
    (i:ISSUE)-[:PREV*0..1]->(assigned:COMMIT)
where
  c.created < cc.created and
  (assigned = c or i.opened < c.created)
return
  n.name as name, count(DISTINCT i) as ‘Number of Bugs‘
```

In this query, we match for any non-filtered method node n that are present in the selected commit c. We also match for any commit cc that changed the method n and it references an issue i that may or may not have a *prev* edge to the commit *assigned*. In the conditions part of the query, we specify that the commit cc (the modification) had to come later than commit c (the selected version). We also specify that the bug i has to be assigned to commit c or that it was reported before commit c. In the latter case, it implies that issue i was closed after commit c, because there is a fix (commit cc) for this bug after the selected commit. At the end of this query, we return with the name of the source code elements and the associated number of bugs. If we want to compute these values for classes, we can just simply change the type of the node n to *CLASS*. Hence it is a very general way of calculating bug numbers.

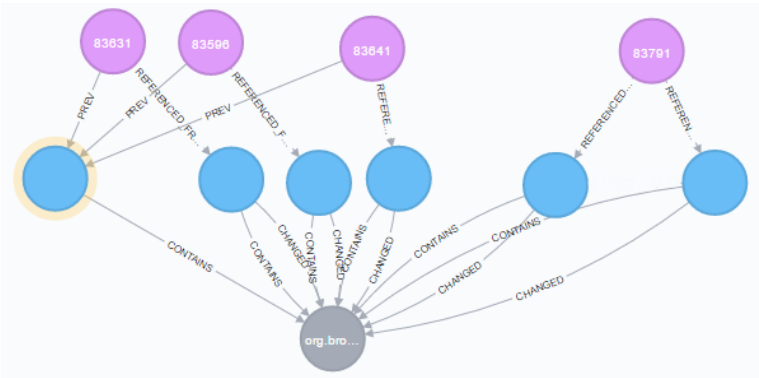

Figure 3: Graph example (purple: Issue, blue: Commit, grey: Method)

Figure 3 shows a typical graph where we can see the matched nodes for a certain method (grey). The blue node on the left highlighted in yellow is the selected commit which contains the method node. The next five blue nodes are bug fixes that are committed later and they changed the method. The first three of these commits reference three issues that are assigned to the selected commit. The last two commits reference an issue that is assigned to another commit, but it was opened before the selected commit. Here, we see that the Number of Bugs for this method in the given version is 4.

In the next section, we will present our tool.

3.2. The open-source toolchain

A schematic overview of the toolchain is shown in Figure 4. The first element on the left represents GitHub, the data source that we utilized. The next element is the preparatory step of the process. It consists of two parts, namely Processing issues and Analyzing source code. In these two steps, we collect every bug report of the chosen project from GitHub's bug tracking system and we analyze every version that is related to any of the bug reports obtained. The source code analysis is carried out by the SourceMeter[3] tool and the bug reports are exported from GitHub by using the GitHub API[4]. More details about this step can be found in our earlier study [1]. Next we build the graph, just like in a previous study [4]. We extended this step to connect the bugs to the selected commits. There we used Neo4j[5] as a graph database engine and it is currently the most popular open-source graph database. It has a query language called *cypher*, as mentioned earlier. After the graph is imported into the Neo4j database, we simply run the queries that we introduced previously. The output of these queries are saved to CSV files. The first line contains the header information, while the rest of the lines are the values returned by the query. Separate outputs are produced for files, classes and methods. These three files form the bug database for the given software version.

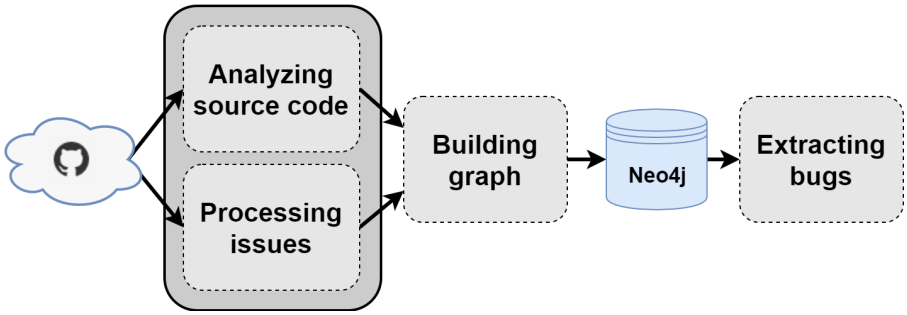

Figure 4: Overview of the toolchain

4. Conclusions and future work

In this study, we presented our improved method and its implementation which is able to compute bug numbers, even at the method level. The computation is carried out in a graph database with queries that are clear and simple. As a database engine, we used Neo4j, which has a lovely visualization interface and it is widely used. Neo4j is an open-source tool and SourceMeter also has a free version. Furthermore, we made the source code available as an open-source project, hence

[3] `https://www.sourcemeter.com/`
[4] `https://developer.github.com/v3/`
[5] `https://neo4j.com/`

the complete system is freely available. Our goal with this tool is to aid studies on software bugs, especially studies on bug prediction at the method level.

In the future, we intend to evaluate the bug prediction capability of method-level process metrics. We also would like to develop the tool further and incorporate it into a continuous integration process.

Acknowledgements. This research was supported by the Hungarian Government and the European Regional Development Fund under the grant number GINOP-2.3.2-15-2016-00037 ("Internet of Living Things"). I would like to express my gratitude to my supervisor Dr. Rudolf Ferenc for his useful comments and remarks, and also David P. Curley for checking this article from a linguistic point of view.

References

[1] Gyimesi, P., Gyimesi, G., Tóth, Z., Ferenc, R., Characterization of source code defects by data mining conducted on GitHub, *In 15th International Conference on Computational Science and Its Applications, ICCSA*, (2015)

[2] Tóth, Z., Gyimesi, P., Ferenc, R., A Public Bug Database of GitHub Projects and Its Application in Bug Prediction, *International Conference on Computational Science and Its Applications*, (2016), 625–638.

[3] Gyimesi, P., An Automatic Way to Calculate Software Process Metrics of GitHub Projects With the Aid of Graph Databases, *The 10th Jubilee Conference of PhD Students in Computer Science*, (2016), 27.

[4] Gyimesi, P., Automatic Calculation of Process Metrics and their Bug Prediction Capabilities, *Acta Cybernetica*, `http://www.inf.u-szeged.hu/~pgyimesi/papers/ActaCybernetica2016`

[5] Williams, C. C., Hollingsworth, J. K., Automatic mining of source code repositories to improve bug finding techniques, *IEEE Transactions on Software Engineering* Vol. 31.6 (2005), 466–480.

[6] Wang, D., Lin, M., Zhang, H., Hu, H., Detect Related Bugs from Source Code Using Bug Information, *Computer Software and Applications Conference (COMPSAC)*, (2010)

[7] Wu, R., Zhang, H., Kim, S., Cheung, S., Relink: recovering links between bugs and changes, *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, (2011), 15–25.

[8] D'Ambros, M., Lanza, M., Robbes R., An Extensive Comparison of Bug Prediction Approaches, *Proceedings of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories)*, (2010), 31–41.

[9] Dallmeier, V., Zimmermann, T., Extraction of bug localization benchmarks from history, *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, (2007), 433–436.

[10] Dallmeier, V., Zimmermann, T., Automatic Extraction of Bug Localization Benchmarks from History, *Universitat des Saarlandes and Saarbrücken and Germany*, (2007)

[11] Menzies, T., Krishna, R., Pryor, D., The Promise Repository of Empirical Software Engineering Data, `http://openscience.us/repo`, *North Carolina State University, Department of Computer Science*, (2016)

[12] Zhou, J., Zhang, H., Lo, D., Where Should the Bugs Be Fixed? More accurate information retrieval-based bug localization based on bug reports, *Software Engineering (ICSE), 2012 34th International Conference on*, (2012)