GitcProc: A Tool for Processing and Classifying GitHub Commits

Casey Casalnuovo
Yagnik Suchak
University of California, Davis, USA
{ccasal,yvsuchak}@ucdavis.edu

Baishakhi Ray
University of Virginia, USA
rayb@virginia.edu

Cindy Rubio-González
University of California, Davis, USA
crubio@ucdavis.edu

ABSTRACT

Sites such as GitHub have created a vast collection of software artifacts that researchers interested in understanding and improving software systems can use. Current tools for processing such GitHub data tend to target project metadata and avoid source code processing, or process source code in a manner that requires significant effort for each language supported. This paper presents GitcProc, a lightweight tool based on regular expressions and source code blocks, which downloads projects and extracts their project history, including fine-grained source code information and development time bug fixes. GitcProc can track changes to both single-line and block source code structures and associate these changes to the surrounding function context with minimal set up required from users. We demonstrate GitcProc's ability to capture changes in multiple languages by evaluating it on C, C++, Java, and Python projects, and show it finds bug fixes and the context of source code changes effectively with few false positives.

CCS CONCEPTS

•Information systems →Data mining; Information extraction; Summarization; *Retrieval efficiency;* **•Software and its engineering →Software maintenance tools;**

KEYWORDS

Git Mining Tool, Information Extraction, Language Independence

ACM Reference format:
Casey Casalnuovo, Yagnik Suchak, Baishakhi Ray, and Cindy Rubio-González. 2017. GitcProc: A Tool for Processing and Classifying GitHub Commits. In *Proceedings of 26th ACM SIGSOFT International Symposium on Software Testing and Analysis , Santa Barbara, CA, USA, July 2017 (ISSTA'17-DEMOS),* 4 pages.
DOI: 10.1145/3092703.3098230

1 INTRODUCTION

Software development and maintance involves changing programs to add new features, fix defects, and improve performance. Such changes take place in many forms: adding new methods, implementing different programming constructs like concurrency and exception handling, updating API functions, etc. Managers, developers, and researchers are often interested in analyzing such evolutionary data to understand the characteristics of software development and maintenance. For example, a manager might be interested to know *"How does programming language choice affect software quality?"*. Similarly, a developer might wonder *"If I use many asserts in my code, does it help reduce bugs?"*. Finally, researchers might be curious *"How does API evolution impact code quality?"*, or *"What are the characteristics of different real-world bugs — e.g., concurrency and performance bugs?"*.

This paper presents GitcProc, a tool that tracks changes to code structures in git projects to facilitate answering project evolution questions. Given a project's git repository URL as input, GitcProc (i) retrieves global statistics of the project's evolution consisting of number of commits, commit dates, number of associated authors, and which commits are involved in bug-fixes, and (ii) tracks the changes of particular program element(s) of interest over time. For example, GitcProc can measure how many changes have taken place involving the *synchronization* mechanism, a language primitive of Java to avoid race conditions. GitcProc is also able to find out the location of the changes (*i.e.*, project version, file, and method), the commit at which such changes are made, and the associated developers. From the input git URL, GitcProc retrieves the entire evolutionary history of the project in the form of a commit log, which is parsed in a language independent way.

The main challenge addressed by GitcProc is the processing and classification of commit diffs. We have designed and implemented GitcProc to be lightweight, easy to use, extensible, and efficient. Our approach for processing commits is based on regular expression matching, and scope tracking. GitcProc processes commit diff code fragments; it does not require to compile or process the entire project's source code. GitcProc is easy to use and does not require the user to learn a new domain specific language (DSL). We demonstrate the extensability of GitcProc by implementing support for four languages: C, C++, Java, and Python. Finally, we present three examples that show the variety of scenarios in which GitcProc can be used, and its effectiveness at identifying relevant commits in an efficient manner.

The rest of the paper is organized as follows. Section 2 presents a motivating example, and Section 3 describes GitcProc. Section 4 shows our experimental evaluation, Section 5 decribes related work, and Section 6 concludes.

2 MOTIVATING EXAMPLE

This section describes the features of GitcProc using the example in Table 1. Assume a user wants to analyze evolution of a project `androidannotations` using GitcProc. First, our tool requires three set of inputs: (a) GitHub project URL `excilys/androidannotations`, (b) some configuration options like language (*e.g.,* Java) (see Section 3 for details), and (c) a set of terms that the user wants to search, say "synchronized" and "UncaughtExceptionHandler". Since "synchronized" can be applied to either method

ISSTA'17-DEMOS, Santa Barbara, CA, USA

978-1-4503-5076-1/17/07…$15.00
DOI: 10.1145/3092703.3098230

Table 1: Bug fix commit for *synchronized* block and *UncaughtExceptionHandler* API function.

Sample Input:	
Project's Name:	excilys/androidannotations
Search Keyword:	"synchronized", included, block "UncaughtExceptionHandler", included, single
Configuration:	Languages: Java
Sample Output:	**Change Summary**
Project:	androidannotations
Commit ID:	ed8e0318f6d767882c24c563bf9d0db02ccbc51f
Author & Date:	Joan Zapata, 2013-10-17
Bugfix Commit?	Yes
Commit Msg:	Fixed #646 sending background exception to the global UncaughtExceptionHandler

Function Patch:

```
+ @Test
+ public void propagateExceptionToGlobalExceptionHandler(){
+  // Prepare lock on which we'll wait for the
+  // background exception handler to catch the exception
+  final Object LOCK = new Object();
+  Thread.setDefault...(new Thread.UncaughtExceptionHandler() {
+   @Override
+   public void uncaughtException(...) {
+     synchronized (LOCK) {
+      propagated... = true;
+      LOCK.notify();
+    }
+   }
+  });
```

Sample Output:	**Patch Summary per Function**
File:	test15/ThreadActivityTest.java
Test File?	Yes
Function Name:	propagateExceptionToGlobalExceptionHandler
# Lines:	added = 14, deleted = 0
# Keywords:	synchronized added = 4, deleted = 0 UncaughtExceptionHandler added = 1, deleted = 0

declarations or block statements, the user may want to know how many lines are added or deleted inside "synchronized" blocks. In contrast, the user may want to search for "UncaughtExceptionHandler" as single-line keyword. While searching for a keyword, GitcProc provides both single-line and block options.

With the GitHub URL as input, GitcProc downloads the whole repository in the local machine using the command `git clone`. It then retrieves the entire evolutionary history using `git log`; the log contains details of each commit made to the project to date including the commit message, commit related metadata, and the associated program patch (see Table 1). Next, GitcProc parses the retrieved log to output a summary of the changes at two granularities: (i) an overall summary of the changes per project, and (ii) a detailed summary of each modified function/method.

Change Summary. GitcProc processes the metadata from the log file and retrieves commit specific information: the identifying SHA, commit metadata like author information, author date, etc., as shown in Table 1. It also uses a keyword search to determine if a commit is related to bug fixing. GitcProc searches for bug-fix related words such as *error*, *bug*, *defect*, and *fix* within the commit message. For example, since our commit message (Fixed #646 ...) contains the term "fix", GitcProc classifies it as a bug-fix commit.

Patch Summary per Function. Each commit can change multiple files and subsequently multiple functions. GitcProc processes the diff patch associated with each commit and outputs statistics for each modified function/method. The tool outputs the file name, method name, and number of lines that are added and deleted during the method change. Further, GitcProc checks whether the changed method contains the input keywords. If found, it also outputs statistics relevant to them. For example, the sample patch

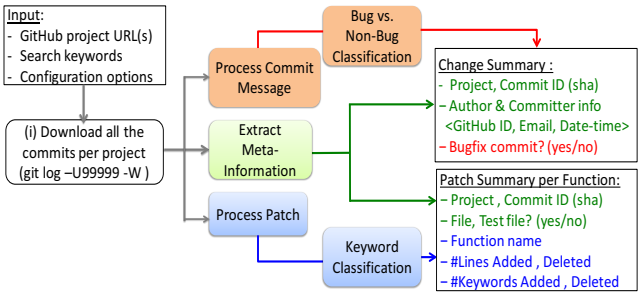

Figure 1: GitcProc **Workflow**

contains both the single-line and block keywords. "Synchronized" being a block keyword, 4 lines have changed under the block. Thus, our tool reports 4 additions and 0 deletions. Since "UncaughtExceptionHandler" is marked as a single-line keyword, our tool reports only one line of addition. GitcProc also checks whether the modification is done in a test file by checking whether the "test" keyword is present in the file path and name. In our example, the tool marks the file `ThreadActivityTest.java` as a test file.

Users can have GitcProc output the extracted information in CSV format, or in a Postgres database. Users can then query the information with SQL or load the CSV files into a variety of frameworks to generate further metrics of interest.

3 TOOL ARCHITECTURE

GitcProc is implemented in Python (5000 SLOC) and Java (300 SLOC), and is available under BSD license [1] [2]. Figure 1 shows GitcProc's inputs: (1) a file with a list of GitHub repositories, (2) a list of keywords, and (3) configuration options. The search configuration option includes exact and approximate match. For exact match, a keyword is only considered matched if it is surrounded by non-alphanumeric characters, while the approximate match should be contained in the line. The exact match is useful for matching code block structures such as *synchronized*, *for*, etc. or specific function calls. The approximate match is useful where projects use different functions for similar functionality, such as *assert*, *ASSERTEQUAL*, *mutex_assert*, etc.—all indicate assertions.

The configuration options also include two flags. The first flag is either *included* or *excluded*. *Included* is the standard case, and signifies this keyword as one to be tracked. The flag *excluded* excludes keywords that overlap with approximate match keywords. For instance, we would use *excluded* with *assert.h* to ignore assert import statements while searching for assertion calls. The second flag is either *single* or *block*, denoting if the keyword is single-line or a block structure. Finally, GitcProc takes a .ini file that specifies database connection information (if any), languages to parse, and optional flags for debugging and logging.

GitcProc outputs a line for each method change in every commit. Figure 1 gives an example of the information contained in the output. These rows are identified by a 4-tuple of project, commit ID (SHA), file and function name, but are not necessarily unique as functions may share a name as a result of overloading. Each row records the additions and deletions to the function, all keywords

[1] GitHub repository: https://github.com/caseycas/gitcproc
[2] Short demo: https://youtu.be/5sOUoMHuP9s

found in the function, and whether the file is a test file. A separate file records information such as commit date and author, along with whether the commit is bug-fixing or not.

3.1 Log Retrieval and Processing

Given a list of GitHub projects, GitcProc retrieves the full history of all non-merge commits along with their commit logs and associated patches. The patches contain the complete file with lines added and deleted for a specific patch, highlighted by +/- at starting of the line, respectively. In particular, we use the command *git log –date=short –no-merges -U99999 –function-context –**extensions***, where **extensions** is the set of extensions for the user specified language of interest. The options -U99999 and –function-context download commit patches and provide sufficient context to parse function names in which changes occurred. Next, we process the retrieved logs, which consists of the following three steps:

(1) Extract Commit Meta Information. For each commit, we extract its ID, author information, and date.

(2) Text Processing. GitcProc can infer whether a given commit message is bug related or not. This feature is useful to study development time bug-fixing activities, since these are not always recorded in an issue database. GitcProc analyzes the commit messages associated with each commit for the entire project evolution, looking for error related keywords. First, we convert each commit message to a bag-of-words and then stem them using standard NLP techniques. Similar to [7], GitcProc marks a commit as a *bug-fix* if the corresponding stemmed bag-of-words contains at least one of the error keywords: 'error', 'bug', 'fix', 'issue', 'mistake', 'incorrect', 'fault', 'defect', 'flaw', and 'type'.

(3) Patch Processing. We provide a general parser for code patches included in GitHub commit logs. The parser tracks change to the function level, providing information on the number of added and deleted lines for each function. If any of the code changes in a given function include the keywords of interest, then these changes are associated with that function's context. The tool allows two types of keywords: *single-line* and *block*. The single-line keywords are code structures that consist only of one line, such as a function call. Block keywords are code structures that have an associated scope, including conditional and loop statements, try/catch blocks, synchronized blocks, etc. Changes inside a block's scope are associated with that block.

We use regular expressions and stacks of open scopes to track when functions and code blocks begin and end. There are two stacks, one for the old and new versions of the source code, which maintain a record of three types of changes: functions, keyword blocks, and other. Whenever the scope of a new source block is opened (identified by brackets in C or Java, or indents in Python), a block is added to the stack. All further changes are associated with these open scopes until the corresponding closing symbol is found, and popped from the stack. The type of change put on the stack is decided by regex matching. Keyword blocks for matches to user-defined keyword, function blocks for matches to the language specific function regexes when no functions are currently on the stack, and all other source blocks are classified as other.

3.2 Usability and Generality

One of our major goals is ease of use. Using GitcProc requires Java 7 and Python 2.7, along with the Python libraries PyYaml, nltk, psycopg2, and git. As described earlier, running the tool requires providing the list of GitHub repositories, search terms of interest, and configuration options.

Currently, GitcProc is able to parse C++, C, Java, and Python diff logs. GitcProc is designed to be extensible to other languages as long they have some notation of scope and code blocks. Extending the tool to languages that express scope either through indentation (Python) or brackets (C/C++/Java) requires creating only a new *LanguageSwitcher* class, which contains a set of regular expressions that describe how the language represents functions, strings, comments, and optionally classes, along with a few preprocessing functions. This class is about at most 100 SLOC. It is possible to extend GitcProc to languages with other representations of scope, though this requires more effort than adding just the regex defining class described above.

3.3 Limitations

As with any regex based tool, our log parser will not work in all cases. Robustly identifying functions in C/C++ is particularly difficult, and our tool misses some cases. We do not currently support finding constructors and destructors in files outside of their class declaration or functions declared in K&R C style. More generally, we do not yet handle very complex scope changes in C/C++/Java with more than 2 changes in scope in a line, such as: `int foo(int x) if (x < 0) return 0; else return 1;`

In our experience, such examples are uncommon in real project code. In general, our parser assumes each version of the code is compilable. However, this is not always true for extracted code. In these and other cases where our tool is unable to parse a line or chunk correctly, it produces a warning flag or error message in the output for that section.

4 EXPERIMENTAL EVALUATION

We use GitcProc to process and classify a total of 153,364 commit logs from projects written in C, C++, Java, and Python (see Table 2). First, we evaluate the effectiveness of GitcProc in identifying bug-fix commits based solely on commit messages. Second, we consider log classification based on specific search keywords.

Bug Classification. We used GitcProc to process a total of 129,316 commit logs from 10 C++/C projects: git, mysql, folly, subvim, xbmc, v8, reds, Torque3d, bitcoin, and mongo. Our tool found that 49,161 are bug-fix related, while the remaining logs are non-bug-fix related. We randomly sampled 100 bug-fix related commit logs, spread evenly among the projects, except for the project subvim, which had very few commits. Manual inspection consisted of looking at the code patch along with any available issue tracker and source code data to determine whether the commit was actually bug fixing or not. We found 4 false positives: in two cases the developer used the word 'fixed' to refer to length and in two cases to refer to improvements made to error handling, not bug fixes. Additionally, there was one other case where we could not determine whether the commit was intended to be a bug fix.

Keyword-Based Log Classification. Here we present three case studies to evaluate GitcProc's precision in identifying the context and changes associated with search keywords : (1) single-line keyword `assert` in C/C++ projects, (2) single-line keyword `asarray` (a NumPy API function) in Python projects, and (3) block keyword `synchronized` in Java projects.

Table 2: Total number of projects and commits per language. The column 'Changed Functions' reports the number of changed functions containing the desired search keywords followed by the total number of changed functions. In the evaluation, we sample from the # of functions in bold.

Language	Total Projects	Total Commits	Commits w/Keywords	Changed Functions
C/C++	10	129,316	20,918	**210,293**/2,118,761
Python	8	7,768	102	**142**/44,279
Java	11	16,280	890	**1,378**/36,002
Total	29	153,364	21,910	**211,813**/2,199,042

(i) Asserts in C/C++. We evaluate GitcProc on the same 10 C/C++ projects, all of which make use of assertions. We select as 'keywords of interest' any approximate matching of the word `assert` as well as exact matches for the function names `ut_a` and `ut_ad`, which are known to be `assert` functions in the mysql project. We run GitcProc to find the commits in which modified lines of a function include the above `assert` related keywords.

Table 2 reports the number of changes at the commit and function level for both keyword containing and all source code changes. Of the 129,316 commits that modify C or C++ files, GitcProc finds that 20,918 contain the keywords of interest. For functions, there were 2,118,761 changed in total, but only 210,293 contain `assert` keywords. We evaluate the precision of our tool by randomly sampling 100 of these functions.

We mark a sample as true positive if we verify that the changed function exists in the commit, and that we correctly count both the lines where assertions were added/deleted along with the changes to all lines. We found that 98 out of 100 samples are correctly classified and reported. The two remaining samples were false positives. One was correctly classified, but the reported counts were wrong. The other did not include `assert` keywords.

(ii) Asarray API function in Python. *asarray* is a NumPy API function that we search for in Python projects. We chose eight Python projects that use NumPy libraries: bottleneck, crab, distarray, minpy, matplotlib, numba, NumPy, and spartan. For the first four projects we process all the commits, for the others we process commits from 1st January 2015 till the current date; Table 2 shows the summary. Out of 7,768 commits studied, only 102 contained the *asarray* function, as found using approximate single keyword search. Out of 44,279 changed methods, only 142 of them contained *asarray*. We also computed the accuracy of keyword search using the same method as described above and examined 20 commits selected randomly; 19 were correctly classified. One false positive came from project minpy—although the commit contains one *asarray* addition, the corresponding method name was not correct.

(iii) Synchronized in Java. We use GitcProc to identify commits that modify code within `synchronized` blocks. We consider 11 Java projects including android, atmosphere, clojure, CraftBukkit, dagger, and elasticsearch. We process and classify a total of 16,280 commits, from which 890 are reported to be `synchronized` related. As with previous case studies, we identify changed functions that include the `synchronized` keyword. We found that 1,378 out of 36,002 changed functions include this block keyword. We randomly selected 50 samples, of which 48 were correctly classified. Two were incorrect: (1) the commit modifies a function called `synchronized`, and not a `synchronized` block, and (2) the number of modified lines is incorrect.

Performance. We ran our tool on a 2.67GHz server with 100GB RAM. The median throughout (lines parsed per second) was 13,757, while median time was 33 secs for Java, 18 secs for Python, and 38 mins for C/C++, which had larger projects.

5 RELATED WORK

GHTorrent[5] and GitHub Archive [1] are designed for large scale language independent repository mining, primarily focusing on GitHub metadata. Other frameworks for mining software artifacts exist, including tools to scalably build web-crawlers to extract diverse artifacts [11], search engines similar to GHTorrent but not restricted to GitHub [2], or proposals for fact extraction based approaches [6]. Other tools include Repograms [8] to visualize high-level commit information, and Chronos [9] to visualize changes over time to specific regions of code.

BOA [4] is a domain-specific language and infrastructure to extract and analyze data from repositories. It supports queries on metadata of all Github projects, and source code AST-based analysis of Java files only. *Gitana* [3] converts metadata and source code information into a database format. *MetricMiner* [10] is a cloud-based web application that processes commit logs and creates code metrics, currently implemented only for Java. We propose a lightweight source code analysis that sits between metadata focused tools like GHTorrent and tools that offer code analysis but require learning DSLs or are difficult to extend to other languages.

6 SUMMARY

We presented GitcProc, a tool to extract information from git log metadata and source code diffs. Its lightweight regex-based approach makes it extensible to other languages, which we demonstrated by tracking diverse code structures in C, C++, Java, and Python. GitcProc allows researchers to generate data locally, while requiring only a list of project names, keywords of interest, and configuration input/output options. We hope the extensibility and low set up costs of GitcProc will assist in fostering reproducibility and ease of extraction in future software studies.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. This work was supported by NSF grant CCF-1464439, and a CAMPOS Fellowship.

REFERENCES

[1] GitHub Archive. https://www.githubarchive.org/.
[2] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillère. Orion: A Software Project Search Engine with Integrated Diverse Software Artifacts. In *ICECCS'13*.
[3] V. Cosentino, J. L. C. Izquierdo, and J. Cabot. Gitana: A SQL-Based Git Repository Inspector. In *ER'15*.
[4] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: a Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *ICSE'13*.
[5] G. Gousios. The GHTorrent Dataset and Tool Suite. In *MSR'13*.
[6] P. Makedonski, F. Sudau, and J. Grabowski. Towards a Model-based Software Mining Infrastructure. *SIGSOFT SEN'15*.
[7] A. Mockus and L. G. Votta. Identifying Reasons for Software Changes using Historic Databases. In *ICSM'00*.
[8] D. Rozenberg, I. Beschastnikh, F. Kosmale, V. Poser, H. Becker, M. Palyart, and G. C. Murphy. Comparing Repositories Visually with Repograms. In *MSR'16*.
[9] F. Servant and J. A. Jones. Chronos: Visualizing slices of source-code history. In *VISSOFT'13*.
[10] F. Z. Sokol, M. F. Aniche, and M. A. Gerosa. MetricMiner: Supporting Researchers in Mining Software Repositories. In *SCAM'13*.
[11] L. Zhang, Y. Zou, and B. Xie. A Scalable Crawler Framework for FLOSS Data. In *Internetware'13*.