Graal: The Quest for Source Code Knowledge

Valerio Cosentino, Santiago Dueñas, Ahmed Zerouali
Bitergia
Madrid, Spain
{valcos, sduenas, ahmed}@bitergia.com

Gregorio Robles, Jesús M. González-Barahona
Universidad Rey Juan Carlos
Madrid, Spain
{grex, jgb}@gsyc.urjc.es

***Abstract*—Source code analysis tools are designed to analyze code artifacts with different intents, which span from improving the quality and security of the software to easing refactoring and reverse engineering activities. However, most tools do not come with features to periodically schedule their analysis or to be executed on a battery of repositories, and lack support to combine their results with other analysis tools. Thus, researchers and practitioners are often forced to develop ad-hoc scripts to meet their needs. This comes at the risk of obtaining wrong results (because of the lack of testing) and of hindering replication by other research teams. In addition, the resulting scripts are often not meant to be customized nor designed for incrementality, scalability and extensibility. In this paper we present `Graal`, which empowers users with a customizable, scalable and incremental approach to conduct source code analysis and enables relating the obtained results with other software project data. `Graal` leverages on and extends the functionalities of `GrimoireLab`, a strong free software tool developed by Bitergia, a company devoted to offer commercial software development analytics, and part of the CHAOSS project of the Linux Foundation.**

***Index Terms*—Software mining, empirical software engineering, open source, software development, software analytics**

I. Introduction

In recent years, the way of developing software has significantly changed to meet ever-increasing demands in terms of features, security, reliability, cost, and ubiquity. New methodologies, architectures and languages allow to cope with the continuous changes in the product development cycle, the increasing complexity of software systems and the need to accelerate time to market. In this evolving scenario, source code analysis tools play a key role [1]. For instance, they help reducing the risk to introduce in production environments vulnerabilities, bugs and security threats through automated unit tests [2] and continuous integration [3]. Furthermore, they are also used to gather specific knowledge (e.g., code smells, dependency graphs), helpful to support maintenance activities [4] and drive reverse engineering processes [5].

Thanks to the explosion of open source development [6] and its support by emerging social coding platforms [7], more and more source code analysis tools are freely available on the Internet, and attract the interest of researchers willing to mine source code repositories. Nevertheless, these analysis tools are often affected by different limitations, which reduce their usability and force users to set up specific, time-consuming and error-prone work-arounds, hampering in this way also the replication of research studies [8].

On the one hand, most source code analysis tools do not take into account extensibility support, thus combining together the results obtained from different code analysis tools or relating them to other software project data (e.g., bugs) may turn into a challenging task which entails many considerations and technical expertise [9]–[11]. On the other hand, only few tools (e.g., `SonarQube` [12] and `Codacy`[1]) support incrementality (based on the commits history) and scheduling of the analysis as well as the execution on a battery of repositories; the rest require users to deal with the complexities related to build a scalable infrastructure to monitor code repositories and periodically execute the analysis.

In this paper we present `Graal`, a fully open source, customizable, scalable and incremental approach to conduct ad-hoc analysis of source code. It combines and manipulates the output of existing tools and supports cross-cutting analysis on software project data, thus overcoming the aforementioned limitations. `Graal` relies on `Perceval` and `Arthur`, two components of `GrimoireLab`, a popular open source platform for software development analytics, part of the CHAOSS Linux Foundation project.

The remainder of the paper is organized as follows: Section II describes the approach underlying `Graal`, and the functionalities it offers. Section III shows how to install and use `Graal`. Section IV depicts `Arthur` and how to use it with `Graal` to allow scheduled executions. Section V presents how to exploit the proposed approach. Section VI discusses on related work. Finally, Section VII concludes the paper.

II. Overview of Graal

Graal leverages on the Git backend of Perceval and enhances it to set up ad-hoc source code analysis.

A. Perceval

Perceval [13] simplifies the collection of project data by covering more than 20 well-known tools and platforms related to contributing to open source development (e.g., Git, GitHub, StackOverflow). It allows to retrieve collections of homogeneous items (i.e., categories) from multiple sources in an easy

This is a preprint of the paper published in Proceedings of the 18th International Working Conference on Source Code Analysis and Manipulation, SCAM 2018, Madrid, Spain, September 23-24, 2018, DOI 10.1109/SCAM.2018.00021

The work presented in this paper has been funded in part by the European Union's Horizon 2020 Research and Innovation Programme under the Marie Sklodowska-Curie grant agreement No 642954.

[1] https://www.codacy.com/, Accessed: May 31st, 2018.

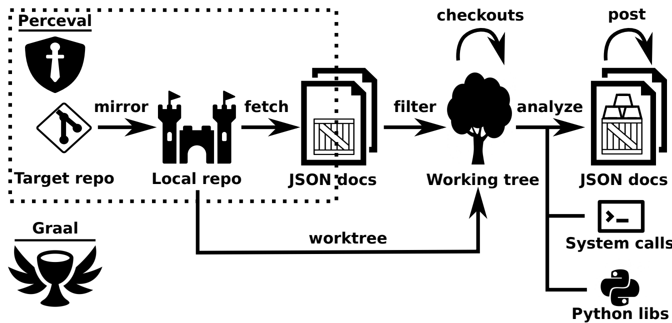

Fig. 1. Overview of Graal.

Listing 1. JSON document excerpt of a Perceval Git item.


```
{"backend_name": "Git",
 "backend_version": "0.10.2",
 "category": "commit",
 "data": {
    "Author": "Santiago Duenas <sduenas@bitergia.com>",
    "AuthorDate": "Tue Aug 18 18:08:27 2015 +0200",
    "Commit": "Santiago Duenas <sduenas@bitergia.com>",
    "CommitDate": "Tue Aug 18 18:08:27 2015 +0200",
    "commit": "dc78c254e464ff334892e0448a23e4cfbfc637a3",
    "files": [{"action": "A", "added": "10",
               "file": ".gitignore", ... }, ...],
    "message": "Initial import", ...
    "parents": [], "refs": []},
"origin": "...github.com/chaoss/grimoirelab-perceval, ...}
```


and consistent way through its backends. For instance, issues and pull requests are the categories extracted by the GitHub backend, while e-mail messages are obtained via one of the supported backends (e.g., Hyperkitty, MBox archives, NNTP and Pipermail). However, **Perceval does not perform source code analysis**, as the Git backend just returns the commit metadata. Perceval is easy to extend, provides incremental support and offers the results in a flexible JSON format. This format has been chosen since it is extremely popular for data interchanging, much more lightweight and easy to manipulate than other formats (e.g., XML).

Listing 1 shows an excerpt of a JSON document produced by Perceval. Each document contains metadata useful for filtering and debugging (e.g., backend version and category) as well as commit-related information, which includes authored and committed dates, the names of the author and committer, the hash and message of the commit plus the references (i.e., tags or branches) where the commit appears. Furthermore, it contains also the commit parents hashes and the list of file paths affected by the commit with the corresponding actions.

B. Extending Percerval's Git backend with Graal

Graal leverages on the incremental functionalities provided by Perceval and enhances the logic to handle Git repositories to process their source code. The overall view of Graal and its connection with Perceval is summarized in Figure 1: the Git backend creates a local mirror of a Git repository (local or remote) and fetches its commits in chronological order. Several parameters are available to control the execution; for instance, *from_date* and *to_date* allow to select commits authored since and before a given date, *branches* allows to fetch commits only from specific branches, and *latest_items* returns only those commits which are new since the last fetch operation.

Graal extends the Git backend by enabling the creation of a working tree (and its pruning), that allows to perform checkout operations which are not possible on a Git mirror. Furthermore, it also includes additional parameters used to drive the analysis to filter in/out files and directories in the repository (*in_paths* and *out_paths*), set the *entrypoint* and define the *details* level of the analysis (useful when analyzing large software projects).

C. Graal at work

Following the *philosophy* of Perceval, the output of the Git backend execution is a list of JSON documents (one per commit). Therefore, Graal intercepts each document, replaces some metadata information (e.g., backend name, category) and enables the user to perform the following steps: (i) filter, (ii) analyze and (iii) post-process, which are described below.

1) Filter: The filtering is used to select or discard commits based on the information available in the JSON document and/or via the Graal parameters (e.g., the commits authored by a given user or targeting a given software component). For any selected commit, Graal executes a checkout on the working tree using the commit hash, thus setting the state of the working tree at that given revision. The filtering default built-in behavior consists in selecting all commits.

2) Analyze: The analysis takes the document and the current working tree and enables the user to set up an ad-hoc source code analysis by plugging existing tools through system calls or their Python interfaces, when possible. The results of the analysis are parsed and manipulated by the user and then automatically embedded in the JSON document. In this step, the user can rely on some predefined functionalities of Graal to deal with the repository snapshot (e.g., listing files, creating archives). By default, this step does not perform any analysis, thus the input document is returned as it is.

3) Post-process: In the final step, the inflated JSON document can be optionally processed to alter (e.g., renaming, removing) its attributes, thus granting the user complete control over the output of Graal executions. The built-in behavior of this step keeps all attributes as they are.

D. Graal Backends

Several backends have been developed to assess the genericity of Graal. Those backends leverage on source code analysis tools, where executions are triggered via their Python interfaces (if written in Python) or system calls (useful to exploit tools written in other programming languages). In the current status, the backends mostly target Python code, however other backends can be easily developed to cover other programming languages. The currently available backends are:

- `CoCom` gathers data about code complexity (e.g., cyclomatic complexity, LOC [14]) from projects written in popular programming languages such as: C/C++, Java,

Listing 2. JSON document excerpt of a CoCom item produced by Graal.


```
{"backend_name": "CoCom",
 "backend_version": "0.2.1",
 "category": "code_complexity",
 "data": { ...,
   "AuthorDate": "Mon May 28 10:15:53 2018 +0200",
   "CommitDate": "Tue May 29 11:21:23 2018 +0200",
   "commit": "dc78c254e464ff334892e0448a23e4cfbfc637a3",
   "analysis": [{
      "avg_ccn": 2.42, "avg_loc": 6.27, "avg_tokens": 44.36,
      "blanks": 138, "ccn": 80, "comments": 153,
      "file_path": "perceval/backend.py",
      "loc": 341, "num_funs": 33, "tokens": 1867}, ...],
  "message": "Increase-coverage-pipermail..."},
"origin": "...github.com/chaoss/grimoirelab-perceval, ...}
```


Scala, JavaScript, Ruby and Python. It leverages on Cloc[2] and Lizard[3]; the former is a Linux package used to count blank lines, comment lines and LOC, while the latter is a code complexity analyzer written in Python.

- CoDep extracts package and class dependencies of a Python module and serializes them as JSON structures, composed of edges and nodes, thus easing the bridging with front-end technologies for graph visualizations. It combines PyReverse[4], a reverse engineering tool able to generate UML-like diagrams, plus NetworkX[5], a library to create, manipulate and study complex networks.
- CoQua retrieves code quality insights, such as checks about line-code's length, well-formed variable names, unused imported modules and code clones. It uses PyLint[6], a code, bug and quality checker for Python.
- CoVuln scans the code to identify security vulnerabilities such as potential SQL and Shell injections, hard-coded passwords and weak cryptographic key size. It relies on Bandit[7], a tool designed to find common security issues in Python code.

Once installed, Graal backends can be used as a stand-alone program or Python library. We showcase these two types of executions by fetching code complexity data using the CoCom backend. Listing 2 shows an excerpt of a JSON document produced. As can be seen, the document contains dates, hash and message of the commit (while files, references and parents have been stripped out with the post-process step), metadata and the result of the analysis for the file *perceval/backend.py*. The latter has 341 LOC, 153 commented lines, 138 blank lines and accounts for 89 as total cyclomatic complexity (CCN). It contains 33 methods, which have on average 2.42 of CCN, 6.27 LOC and 44.36 tokens.

III. Installation and Use of Graal

This section describes how to install and use Graal, highlighting its main features.

[2]http://cloc.sourceforge.net/, Accessed: May 31st, 2018.
[3]https://github.com/terryyin/lizard, Accessed: May 31st, 2018.
[4]https://www.logilab.org/project/pyreverse, Accessed: May 31st, 2018.
[5]https://networkx.github.io/, Accessed: May 31st, 2018.
[6]https://www.pylint.org/, Accessed: May 31st, 2018.
[7]https://pypi.org/project/bandit/, Accessed: May 31st, 2018.

Listing 3. How to install/uninstall Graal.

```
# To install, run:
$ git clone https://github.com/valeriocos/graal
$ python3 setup.py build
$ python3 setup.py install
# To uninstall, run:
$ pip3 uninstall graal
```

Listing 4. Using Graal as a program.

```
$ graal cocom https://github.com/chaoss/grimoirelab-perceval
  --git-path /tmp/graal-cocom > /graal-cocom.test
[2018-05-30 18:22:35,643] - Starting the quest for the Graal.
[2018-05-30 18:22:39,958] - Git worktree /tmp/... created!
[2018-05-30 18:22:39,959] - Fetching commits: ...
[2018-05-31 04:51:56,111] - Git worktree /tmp/... deleted!
[2018-05-31 04:51:56,112] - Fetch process completed: ...
[2018-05-31 04:51:56,112] - Quest completed.
```

A. Installation

Graal is being developed and tested mainly on GNU/Linux platforms. Thus it is very likely it will work out of the box on any Linux-like (or Unix-like) platform, upon providing the right version of Python. Listing 3 shows how to install and uninstall Graal on your system. Currently, the only way of installing Graal consists of cloning the GitHub repository hosting the tool[8] and using the setup script, while uninstalling the tool can be easily achieved by relying on the pip management system.

B. Use

Graal can be used in two different ways:

1) Stand-alone Program: Using Graal as stand-alone program does not require much effort, but only some basic knowledge of GNU/Linux shell commands. Listing 4 shows how easy it is to fetch code complexity information from a Git repository. As can be seen, the CoCom backend requires the URL where the repository is located (*https://github.com/chaoss/grimoirelab-perceval*) and the local path where to mirror the repository (*/tmp/graal-cocom*). Then, the JSON documents produced are redirected to the file *graal-cocom.test*. The remaining messages in the listing are prompted to the user during the execution.

Interesting optional arguments are *from-date*, which is inherited from Perceval and allows to fetch commits from a given date, *worktree-path* which sets the path of the working tree, and *details* which enables fine-grained analysis by returning complexity information for methods/functions.

2) Python Library: Graal's functionalities can be embedded in Python scripts. Again, the effort of using Graal is minimum. In this case the user only needs some knowledge of Python scripting. Listing 5 shows how to use Graal in a script. The *graal.backends.core.cocom module* is imported at the beginning of the file, then the *repo_uri* and *repo_dir* variables are set to the URI of the Git repository and the local path

[8]https://github.com/valeriocos/graal

Listing 5. Using Graal as a library.

```
#! /usr/bin/env python3
from graal.backends.core.cocom import CoCom
# URL for the git repo to analyze
repo_uri = 'http://github.com/chaoss/grimoirelab-perceval'
# directory where to mirror the repo
repo_dir = '/tmp/graal-cocom'
# Cocom object initialization
cc = CoCom(uri=repo_url, gitpath=repo_dir)
# fetch all commits
commits = [commit for commit in cc.fetch()]
```

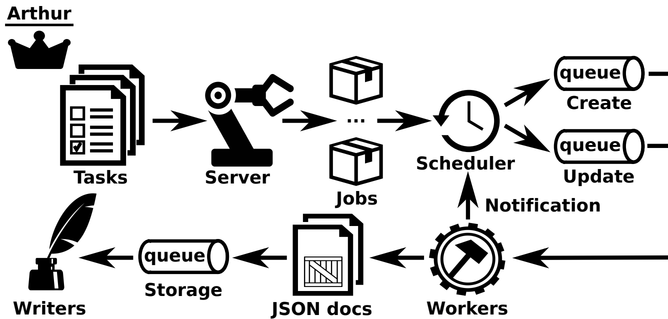

Fig. 2. Overview of Arthur.

where to mirror it. These variables are used to initialize a *CoCom* class object. In the last line of the script, the commits inflated with the result of the analysis are retrieved using the *fetch* method. The *fetch* method inherits its argument from Perceval, thus it optionally accepts two *Datetime* objects to gather only those commits after and before a given date, a list of branches to focus on specific development activities, and a flag to collect the commits available after the last execution.

IV. Scheduling Graal executions with Arthur

This section presents Arthur and how to combine it with Graal, to enable Graal having scalability and support for (cross-cutting) data analysis and visualizations.

A. Original execution of Perceval through Arthur

Originally, Arthur[9] was designed to allow to schedule and run Perceval executions at scale through distributed `Redis` queues[10], and store the obtained results in an `ElasticSearch` database[11], thus giving the possibility to connect the results with analysis and/or visualizations tools.

Figure 2 highlights the overall view of Arthur. At its heart there are two components: the server and one or more instances of workers, in charge of running Perceval executions.

The server waits for HTTP requests, which allow to add, delete or list tasks using REST API commands (i.e., *add*, *remove*, *tasks*). Listing 6 depicts how to send commands to the Arthur server. As can be seen, adding and removing tasks

[9]https://github.com/chaoss/grimoirelab-kingarthur

[10]Redis is an open source in-memory database project implementing a distributed, in-memory key-value store.

[11]ElasticSearch is an open source search engine and schema-free JSON documents storage.

Listing 6. REST API commands to interact with the Arthur server.

```
# Adding tasks
$ curl -H "Content-Type: application/json"
--data @to_add.json http://127.0.0.1:8080/add
# Removing tasks
$ curl -H "Content-Type: application/json"
--data @to_remove.json http://127.0.0.1:8080/remove
# Listing tasks
$ curl http://127.0.0.1:8080/tasks
```

requires specific parameters, sent as JSON data within the request. Adding a task needs a JSON object that contains a task id (useful for deleting and listing operations), the parameters needed to execute a Perceval backend, plus other optional parameters to control the scheduling (i.e., delayed start, maximum number of retries upon failures) and archive the fetched data. Conversely, in order to remove a task, the JSON object must contain the identifier of that given task.

After receiving a task, the server initializes a job with the task parameters, thus enabling a link between the job and the task, and sends the job to the scheduler. The scheduler manages two (in-memory) queues handling first-time jobs and already finished jobs that will be rescheduled. The former are Perceval executions that perform the initial gathering from a data source, while the latter are executions launched in incremental mode (e.g., from a given date, which is by default the date when the previous execution ended). In case of execution failures, the job is rescheduled as many times as defined in the scheduling parameters of the task.

Workers grant Arthur with scalability support. They listen to the queues, pick up jobs and run Perceval backends. Once the latter have finished, workers notify the scheduler with the result of the execution, and in case of success, they send the JSON documents to the server storage queue. Such documents are consumed by writers, which make possible to live-stream data or serialize it to database management systems. In the current implementation, Arthur stores the JSON documents to an ElasticSearch database.

B. Executing Graal through Arthur

Arthur has been extended to allow handling Graal tasks, which inherit from Perceval Git tasks, thus Arthur periodically executes the method *fetch* of a given Graal backend. Optionally, the parameter *latest_items* can be used to run the analysis only on the new commits available after the last execution.

Listing 7 shows two examples of JSON objects to include and delete Graal tasks. As can be seen, adding a task to analyze the code complexity of a repository consists of sending an *add* command to the Arthur server with a JSON object including a task id (*cocom_graal*), the parameters needed to execute an instance of the `CoCom` backend, such as its category (i.e., *code_complexity*), the URI of the target repository and the local path where it will be mirrored (i.e., *uri* and *git_path*). Furthermore, the task defines also the scheduler settings *delay* and *max_retries*, which allow to postpone the scheduling of the corresponding job and set the maximum number of retries

Listing 7. JSON objects to add and remove Graal tasks in Arthur.

```
# Adding tasks
{"tasks": [{
   "task_id": "cocom_graal",
   "backend": "cocom",
   "backend_args": {
     "uri": "https://.../grimoirelab-perceval",
     "git_path": "/tmp/graal-perceval"},
   "archive": {},
   "category": "code_complexity",
   "scheduler": {"delay": 2, "max_retries": 5}, ...]}
# Removing tasks
{"tasks": [{"task_id": "cocom_graal"}, ...]}
```

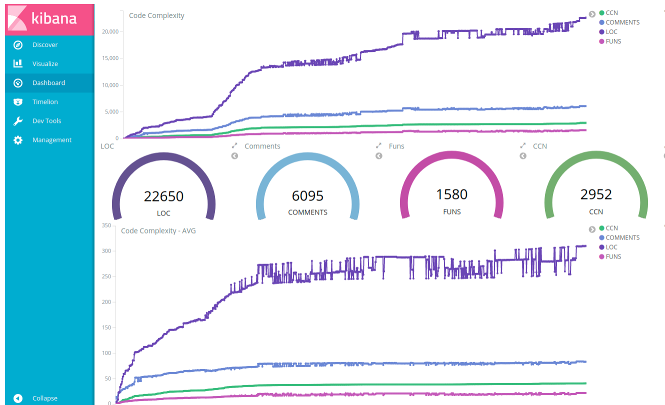

Fig. 3. Kibana dashboard with data gathered with Graal. The top chart shows the evolution of LOC, comment lines, CCN and functions in the Perceval repository, while the widgets in the middle summarize their current values, and the bottom chart presents the evolution of their averages.

upon job failures before raising an exception[12]. Deleting the *cocom_graal* task requires less effort, it suffices to send a *remove* command to the Arthur server that includes a JSON object with the target task.

V. Exploitation

The JSON documents obtained by Graal and persisted to ElasticSearch can be visualized by means of `Kibana` dashboards[13] or exploited using common libraries for data analytics like `Pandas` [15] and `R` [16], thus enabling the proposed approach with support for data visualization and analysis. Furthermore, the Graal documents can be easily combined with the Perceval ones (storing both to the database), thus enabling cross-cutting analysis on software project data.

We showcase some possible visualizations and analysis on the data collected from the Perceval[14] GitHub repository, which include more more than 1195 commits, 388 issues and 250 pull requests in the last 4 years.

A. Visualization

True interactive dashboards can be easily created by relying on Kibana. Figure 3 shows a dashboard composed of two

[12]Note that archive parameters are not set, since in the current implementation the archiving of data obtained by Graal backends is not supported.

[13]`Kibana` is an open source tool enabling visual exploration and real-time analysis of data in ElasticSearch.

[14]https://github.com/chaoss/grimoirelab-perceval

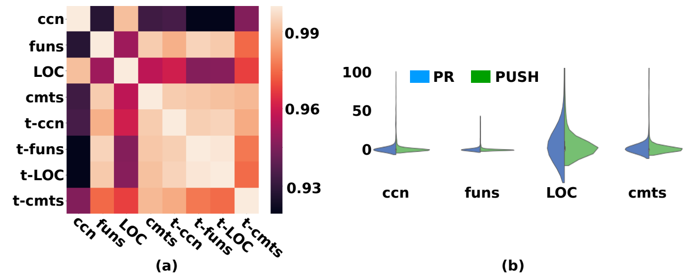

Fig. 4. Analysis with `Pandas` and `Python Notebooks` of the Perceval repository. (a) shows the correlation between the complexity of test and no-test code, (b) compares code complexity between pull requests and direct pushes.

charts and some widgets that use the code complexity data, such as LOC, comment lines, number of functions and CCN, obtained via the `CoCom` backend.

The top chart of Figure 3 shows the evolution of LOC, comment lines, number of functions and CCN across the commit history of the Perceval repository. As can be seen, the evolution of LOC in Perceval witnessed a spike around the first year of the project due to the addition of new backends. A similar spike occurred also on comment lines and CCN, but to a lesser extent. Conversely, the number of functions has increased smoothly over time. The widgets in the middle of the dashboard summarize the current values of code complexity in Perceval. Thus, the latest snapshot of Perceval (i.e., May 16th, 2018) contains 22,650 Python LOC, 6,095 comment lines, 1,580 functions and 2,952 as total of CCN.

Finally, the bottom chart of Figure 3 presents the evolution of the average of code complexity. As can be seen, except for an initial fast growth, LOC, comment lines, CCN and number of functions have grown slowly. Interesting enough is the shape of the LOC evolution: the high and low peaks represent the addition of new features (e.g., new backends or enhancements of existing ones) and refactorings, thus pointing out that the project has been constantly active. In the latest snapshot, every Python file has on average 310 LOC, 83 comment lines, 22 functions and 40 of CCN.

B. Data analysis.

`Pandas` [15] is one of the most common libraries used in data analytics with Python. It can be very useful when dealing with the output of Graal and Perceval. `Jupyter Python Notebooks` can be used in combination with Pandas for early prototyping of data visualization [17]. Since the process is completely transparent (i.e., Python Notebooks show the results and the code needed to produce them), researchers and practitioners can examine the output and perform changes fast.

Figure 4 shows the result of two analysis using Python Notebooks. The analysis in Figure 4(a) relies on the code complexity data obtained via the `CoCom` backend and studies the correlation between the test and no-test code in Perceval. As can be seen, test and no test code are highly correlated, however the relations between the number of functions and test code (i.e., *t-ccn*, *t-funs*, *t-LOC*, and *t-cmnts*) is stronger than the relation between the *CCN* and test code. The analysis

in Figure 4(b) combines code complexity data with pull requests and commit information to identify differences between code submitted via pull requests and direct pushes. We have found that commits from pull requests have more LOC and comments (i.e., statistically significant effect size) than other commits.

VI. Related work

There exist several approaches similar to `Graal`. They can be divided into two main categories: i) frameworks that define database schemas to store project data and query it, and ii) platforms for downloading open source projects, analyzing their data (e.g., code, issues, revisions) and building datasets.

The first category includes tools like Gitana [18], CVSAnalY [19], Churrasco [20], Moose [21], SPO [22], MetricMiner [23], SonarQube [12], QWALKEKO [24] and Codacy. However, when compared to Graal, the project data they cover is limited [19], [21]–[24] (or do not include code information [18]) and they do not provide an easy way to extract their data [12]. Furthermore, these tools do not rely on schema-less data structures as Graal does, thus hampering their extensions with new project data by non-technical users. For instance, Moose and Churrasco use UML to handle the data, while Gitana and MetricMiner leverage on relational schemas.

The second category includes popular platforms and infrastructures such as BOA [25], Black Duck[15], Candoia [26], Google BigQuery [27], GHTorrent [28], OSSMeter [29] and Software Heritage [30]. They enable users to benefit (e.g., querying, building datasets) from the data they host, such as source code [30], activities (e.g., issues, pull requests, commits) on forges [28] and code metrics [25], [26], [29]. However, unlike Graal, they are not meant for customization (e.g., adding new code metrics or project data) and they cover far less data sources than the proposed approach.

VII. Conclusions

In this paper we have presented `Graal`, a fully open source, customizable, scalable and incremental approach to conduct ad-hoc analysis of source code by leveraging on existing tools, which can be easily integrated via system calls or their Python interfaces. Graal is built on top of `Perceval` and `Arthur`, two components of `GrimoireLab`. The former is used to grant the approach with incremental functionalities and enhanced to process source code in Git repositories; the latter has been extended to enable Graal with scalability and support for (cross-cutting) data analysis and visualizations.

As further work, we would like to have a deeper integration of Graal with GrimoireLab, thus providing specific dashboards through the platform. Moreover, at the tool level, we would like to speed up the time needed to perform the initial analysis by parallelizing it on multiple Git working trees. We are also interested in combining Graal and Perceval output to better understand the effects of open source development applied to proprietary software products (e.g., `Inner Source` [31]).

[15] https://www.openhub.net/, Accessed: May 31st, 2018

VIII. Acknowledgment

This research has received funding from SENECA, an European Union's H2020 Marie Sklodowska-Curie grant, agreement No 64295.

References

[1] M. Harman, "Why source code analysis and manipulation will always be important," in *SCAM*, 2010, pp. 7–19.
[2] R. Ramler, M. Moser, and J. Pichler, "Automated static analysis of unit test code," in *SANER*, vol. 2, 2016, pp. 25–28.
[3] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in github," in *FSE*, 2015, pp. 805–816.
[4] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *TSE*, vol. 39, no. 8, pp. 1144–1156, 2013.
[5] V. Cosentino, J. Cabot, P. Albert, P. Bauquel, and J. Perronnet, "A model driven reverse engineering framework for extracting business rules out of a Java application," in *RuleML*, 2012, pp. 17–31.
[6] S. Weber, *The success of open source*. Harvard University Press, 2004.
[7] V. Cosentino, J. L. Cánovas-Izquierdo, and J. Cabot, "A systematic mapping study of software development with github," *IEEE Access*, vol. 5, pp. 7173–7192, 2017.
[8] J. M. González-Barahona and G. Robles, "On the reproducibility of empirical software engineering studies based on data retrieved from development repositories," *Empir. Soft. Eng.*, vol. 17, no. 1-2, pp. 75–89, 2012.
[9] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *ICSM*, 2003, pp. 23–32.
[10] G. Robles, J. M. González-Barahona, and R. A. Ghosh, "Gluetheos: Automating the retrieval and analysis of data from publicly available software repositories," in *MSR*, 2004, pp. 28–31.
[11] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "Relink: recovering links between bugs and changes," in *FSE*, 2011, pp. 15–25.
[12] G. Campbell and P. P. Papapetrou, *SonarQube in action*. Manning Publications Co., 2013.
[13] S. Dueñas, V. Cosentino, G. Robles, and J. M. Gonzalez-Barahona, "Perceval: Software project data at your will," in *ICSE*, 2018, pp. 1–4.
[14] K. Magel, R. M. Kluczny, W. A. Harrison, and A. R. Dekock, "Applying software complexity metrics to program maintenance," *Computer*, 1982.
[15] W. McKinney *et al.*, "Data structures for statistical computing in python," in *SciPy*, vol. 445, 2010, pp. 51–56.
[16] R. Core Team, "R language definition," 2000.
[17] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay *et al.*, "Jupyter notebooks-a publishing format for reproducible computational workflows." in *ELPUB*, 2016, pp. 87–90.
[18] V. Cosentino, J. L. Cánovas-Izquierdo, and J. Cabot, "Gitana: A software project inspector," *Sci. Comput. Program.*, vol. 153, pp. 30–33, 2018.
[19] G. Robles, S. Koch, and J. M. González-Barahona, "Remote analysis and measurement of libre software systems by means of the CVSAnalY tool," in *RAMSS*, 2004, pp. 51–56.
[20] M. D'Ambros and M. Lanza, "A flexible framework to support collaborative software evolution analysis," in *CSMR*, 2008, pp. 3–12.
[21] S. Ducasse, T. Gîrba, and O. Nierstrasz, "Moose: An agile reengineering environment," in *ESEC/FSE*, 2005, pp. 99–102.
[22] M. Lungu, M. Lanza, T. Gîrba, and R. Robbes, "The small project observatory: Visualizing software ecosystems," *Sci. Comput. Program.*, vol. 75, no. 4, pp. 264–275, 2010.
[23] F. Z. Sokol, M. F. Aniche, and M. A. Gerosa, "Metricminer: Supporting researchers in mining software repositories," in *SCAM*, 2013, pp. 142–146.
[24] R. Stevens and C. De Roover, "Querying the history of software projects using QWALKEKO," in *ICSME*, 2014, pp. 585–588.
[25] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: Ultra-large-scale software repository and source-code mining," *ACM T. Softw. Eng. Meth.*, vol. 25, no. 1, p. 7, 2015.
[26] N. M. Tiwari, G. Upadhyaya, H. A. Nguyen, and H. Rajan, "Candoia: a platform for building and sharing mining software repositories tools as apps," in *MSR*, 2017, pp. 53–63.

[27] J. Tigani and S. Naidu, *Google BigQuery Analytics*. John Wiley & Sons, 2014.

[28] G. Gousios and D. Spinellis, "GHTorrent: GitHub's data from a firehose," in *MSR*, 2012, pp. 12–21.

[29] D. Di Ruscio, D. S. Kolovos, I. Korkontzelos, N. Matragkas, and J. J. Vinju, "OSSMETER: A software measurement platform for automatically analysing Open Source software projects," in *FSE*, 2015, pp. 970–973.

[30] R. Di Cosmo and S. Zacchiroli, "Software Heritage: Why and how to preserve software source code," in *iPRES*, 2017.

[31] K.-J. Stol and B. Fitzgerald, "Inner source–adopting open source development practices in organizations: a tutorial," *IEEE Soft.*, vol. 32, no. 4, pp. 60–67, 2015.