git2net - Mining Time-Stamped Co-Editing Networks from Large git Repositories

Christoph Gote
Chair of Systems Design
ETH Zürich
Zurich, Switzerland
cgote@ethz.ch

Ingo Scholtes
Data Analytics Group
Department of Informatics
University of Zurich
Zurich, Switzerland
scholtes@ifi.uzh.ch

Frank Schweitzer
Chair of Systems Design
ETH Zürich
Zurich, Switzerland
fschweitzer@ethz.ch

***Abstract*—Data from software repositories have become an important foundation for the empirical study of software engineering processes. A recurring theme in the repository mining literature is the inference of developer networks capturing e.g. collaboration, coordination, or communication from the commit history of projects. Most of the studied networks are based on the *co-authorship* of software artefacts defined at the level of files, modules, or packages. While this approach has led to insights into the social aspects of software development, it neglects detailed information on code changes and code ownership, e.g. which exact lines of code have been authored by which developers, that is contained in the commit log of software projects.**

Addressing this issue, we introduce `git2net`, a scalable `python` software that facilitates the extraction of fine-grained *co-editing networks* in large `git` repositories. It uses text mining techniques to analyse the detailed history of textual modifications *within* files. This information allows us to construct directed, weighted, and time-stamped networks, where a link signifies that one developer has edited a block of source code originally written by another developer. Our tool is applied in case studies of an Open Source and a commercial software project. We argue that it opens up a massive new source of high-resolution data on human collaboration patterns.

I. Introduction

Software repositories are a rich source of data facilitating empirical studies of software engineering processes. Methods to use meta-data from these repositories have become a common theme in the repository mining literature. Thanks to the availability of massive databases, already simple means allow to query meta-data on the commits of developers [1, 2]. Apart from the evolution of software artefacts, they also contain a wealth of fine-grained information on the human and social aspects of software development teams. Specifically, the commit history of developers allows to construct social networks that proxy collaboration, coordination, or communication structures in software teams. These databases have therefore facilitated data-driven studies of social systems not only in empirical software engineering, but also in areas like computational social science, social network analysis, organisational theory, or management science [3, 4].

The detailed record of file modifications contained in the commit log of, e.g. `git` repositories also enables more advanced network reconstruction techniques. In particular, from the micro-level analysis of textual modifications between subsequent versions of code we can infer *time-stamped, weighted, and directed co-editing relationships*. Such a relationship $(A, B; t, w)$ indicates that at time t developer A modified w characters of code originally written by another developer B. Recent research has shown that such a fine-grained analysis of co-editing networks in large software projects can provide insights that go beyond more coarse-grained definitions [5, 6]. However, a tool to conveniently extract such rich, time-stamped collaboration networks for the large corpus of `git` repositories available, e.g. via public platforms like `gitHub`, is currently missing.

Addressing this gap, we present such a tool that facilitates the scalable extraction of time-stamped co-editing relationships between developers in large software repositories. The contributions of our work are as follows:

- We introduce `git2net`, a python tool that can be used to mine time-stamped co-editing relations between developers from the sequence of file modifications contained in `git` repositories. Building on the repository mining framework `pyDriller` [7], `git2net` can operate both on local and remote repositories. Providing a command-line interface as well as an API, `git2net` can be used as stand-alone tool for standard analysis tasks as well as a framework for the implementation of advanced data mining scripts. Our tool is available as an Open Source project[1].
- Analysing all file modifications contained in the commit log, `git2net` generates a database that captures fine-grained information on co-edited code either at the level of lines or contiguous code regions. Building on text mining techniques, it further analyses the overlap between co-edited code regions using (i) the Levenshtein edit distance [8] and (ii) a text-based entropy measure [9]. These measures facilitate (i) a character-based proxy estimating the effort behind code modifications, and (ii) an entropy-based correction for binary file changes that can have a considerable impact on text-based effort estimation techniques.
- We develop an approach to generate time-stamped collaboration networks based on multiple projections: (i) time-stamped co-editing networks, (ii) time-stamped bipartite networks linking developers to edited files, and (iii) directed

[1] https://github.com/gotec/git2net

acyclic graphs of code edits that allow to infer "paths" of consecutive edits building upon each other. All network projections are implemented in `git2net` and can be directly exported as `HTML` visualisations as well as formats readable by common network analysis tools.

- Thanks to a parallel processing model that utilises modern multi-core architectures, `git2net` supports the analysis of massive software repositories with hundreds of thousands of commits and millions of lines of code. A scalability analysis proves that our parallel implementation yields a linear speed-up compared to a single-threaded implementation, thus facilitating the fine-grained textual analysis even in massive projects with a long history.
- Utilizing `git2net` in a case study on two software projects, we show that the fine-grained textual analysis of file modifications yields considerably different network structures compared to coarse-grained methods that analyse code co-authorship at the level of files or modules. We further demonstrate how our tool can be used to breakdown developer effort into (a) the revision of code authored by the developer him or herself vs. (b) the revision of code written by other team members.

Providing a novel method to mine fine-grained collaboration networks at high temporal resolution from any `git` repository, our work opens new perspective for empirical studies of development processes. It further contributes a simple method to generate data on temporal social networks that are of interest for researchers in computational social science, (social) network analysis and organisational theory.

The remainder of this paper is structured as follows: Section II provides an overview of works addressing the construction of social networks from software repository data. Section III introduces our proposed methodology to extract time-resolved and directed links between developers who subsequently edit each others' code. Section IV presents a case study, in which we apply our tool to `git` repositories from (i) an Open Source Software project, and (ii) a commercial, closed-source project. In section V we draw conclusions from our work and highlight the next steps in our research.

II. Related Work

Given the large body of work using network analysis to study software development processes, we restrict our overview to related works that address the reconstruction of social networks from software repositories. A broader view on applications of graph-based data analysis and modelling techniques in empirical software engineering—including works on (technical) dependency networks that are outside the scope of our work—is, e.g., available in [10, 11, 12].

A number of studies use operational data on software projects to construct graphs or networks where nodes capture developers while links capture social interactions and/or work dependencies between developers. To this end, a first line of works has used data that directly capture communication [13], e.g. via IRC channels [14], E-Mail exchanges [15, 16, 17, 18, 19], mailing lists [20], or communication via issue trackers [21, 22, 23, 24, 25].

While data on direct developer communication facilitate the construction of meaningful social networks, they are often not available, e.g. due to privacy concerns. To address such settings, researchers have developed methods to *infer* or *reconstruct* collaboration networks based on developer actions recorded in code repositories like `CVS`, `SVN`, or `git`. A common approach starts from *code authorship* or *code ownership* networks, which map the relation between a developer and the artefacts (i.e. files, modules, binaries, etc.) that he or she contributed to [26, 27, 28, 29]. The resulting directed bipartite developer-artefact networks [30] can then be projected onto *co-authorship networks*, where undirected links between two developers A and B indicate that A and B have modified at least one common artefact. The authors of [31, 32] have studied co-change based on a large corpus of CVS repositories of Open Source Software projects.

The majority of works mining social networks from software repositories build on this general idea. In [29, 33, 34, 35, 36] a file-based notion of co-authorship is used to construct *co-commit networks*, where a link between two developers signifies that they have committed the same file at least once. The authors of [37] adopt a module-based definition, assuming that two developers are linked in the co-authorship network if they have contributed to at least one common module. Taking a similar approach, Huang and Liu [38] use information on modified file paths in `SourceForge` repositories to infer relations between authors editing the same part of a project. Incorporating the time stamps of commits, Pohl and Diehl [39] used a file-based co-authorship definition to construct *dynamic* developer networks that can be analysed and visualised using methods from dynamic network analysis [40]. The authors of [41] recently developed a similar approach to study the ecosystem of software projects on `gitHub`. To this end, they define project-level co-commit networks, i.e. a projection of commits where two developers are linked if they committed to the same Open Source project. Schweitzer et al. [42] provided a related study, analysing ten years of data from the Open Source project hosting platform `SourceForge`.

These works have typically constructed *undirected co-authorship networks* based on joint contributions to files, modules, or projects. Such coarse-grained definitions of co-authorship networks introduce a potential issue: They do not distinguish between (i) links between developers that are due to *independent* contributions to the same artefact, and (ii) links that are due to commit sequences where one developer builds upon and/or redacts the particular lines of source code previously authored by another developer. Networks defined based on the latter type of *time-ordered co-editing* of code regions are likely associated with a stronger need for coordination and communication than the mere fact that developers edited the same file or module [43]. So far, few studies have adopted such fine-grained approaches to create developer collaboration networks. Notable exceptions include the function-level co-editing networks constructed by Joblin et al. [5]. The authors

further argue that, using file-based definitions of collaboration networks, network analytic methods fail to identify meaningful communities. The authors of [6] constructed line-based co-editing networks, showing that such an analysis (i) yields insights into the coordination structures of software teams, and (ii) provides new ways to test long-standing hypotheses about cooperative work from social psychology.

While such a fine-grained analysis of the co-editing behaviour of developers has its advantages, it also introduces challenges that have so far limited its adoption. First and foremost, it requires a detailed analysis of file modifications and makes it necessary to identify the original author for every modified line of code affected in each commit. Requiring a potentially large number of `git` operations for every commit being analysed, such an analysis is both complicated to implement as well as time-consuming to perform. Compared to other approaches, which often merely require a suitable query in structured databases like `ghTorrent` [1, 2], a tool that facilitates this task for very large repositories is still missing.

Closing this gap, our work introduces a practical and scalable solution for the construction of fine-grained and time-stamped co-editing networks from `git` repositories. Our work extends the state-of-the-art and facilitates analyses of developer collaboration and coordination in software projects. Providing a new method to construct large, dynamic networks at high temporal resolution we further expect our work to be of interest for the community of researchers developing methods to analyse dynamic (social) networks [40, 44, 45].

III. Mining Co-Editing Relations from Git Repositories

A. From Commit Logs to Co-Edits

We first outline our proposed method to extract co-editing relationships from `git` commits. An overview of the mining procedure, which we will explain in the following, is presented in Algorithm 1.

`git` projects generally consist of multiple files that can be edited by a large number of developers. Sets of changes made by a developer to potentially multiple files are recorded as commits, where each commit is identified by a unique hash. Building on the package `pydriller` [7], we first extract the history of all commits in a repository and record the meta-data (author, time of commit, branch, etc.) for each commit. As the person committing the changes is not necessarily the author of these changes (a different developer can commit code on behalf of the original author), both the committer and author of the changes are considered. Subsequently we analyse the changes made with the commit.

As each commit can contain modifications of multiple files, we analyse each file modification individually to associate every changed text region with its original author. In a first step, select the modifications relevant for the current analysis. To this end, we have implemented a filter allowing to exclude specific files, file types as well as entire directories or sub-directories from the analysis. For all selected modifications, the associated `diff` is analysed, determining which lines have been added or deleted. In addition, we identify the original author of every edited line of code by executing `git blame` on the version of the analysed file before the current commit. By matching the author A of a modification contained in the current commit with time stamp t to all original authors B_i of an edited line i, we obtain time-stamped and directed co-editing relations (A, B_i, t).

Algorithm 1 Simplified mining procedure of `git2net`

```
1: procedure MINE_GIT_REPO(git_repo, output_db)
2:     for all commits in git_repo do
3:         commit_info ← parsed commit data
4:         for all modified files in commit do
5:             deleted_lines, added_lines ← parse diff of modification
6:             blame_info ← git blame on file in parent commit
7:             for each line deleted lines do
8:                 current_author ← modifying author from commit_info
9:                 previous_author ← original author from blame_info
10:                coedits_info ← authors and metadata on changes
11:        output_db ← commit_info, coedits_info
```

For each extracted relation, we record hashes of the original and modifying commit as well as meta-data capturing the location (file name, line number) of the associated co-edit. Naturally, such co-edits can be linked to vastly different development effort, ranging from a change of whitespaces to the complete rewriting of code. To capture to what extent developers edit each others' code, we use a text mining approach to address these differences. We specifically use the Levenshtein edit distance [8], which can be thought of as the minimum number of keystrokes required to transform the prior source code version into the version after the edit. This measure proxies the development effort associated with an edit, where single character changes, line deletions, or the commenting/uncommenting of lines are associated with a minimum effort while the writing of a new line of code is associated with maximum effort. This approach allows us to construct time-stamped and *weighted* co-edit relations $(A, B; t, w)$, where the weight w captures the Levenshtein distance of the associated edit.

An issue that we have encountered during the testing of our method in real-world repositories is associated with the embedding of text-encoded binary objects in source code, e.g. due to the inclusion of `base64`-encoded images in `HTML` or `JavaScript`. Notably, the modification of a single pixel in a text-encoded image, can result in a completely different text encoding. Considering our approach to associate the weight of a co-edit relation with the Levensthein edit distance this can considerably distort our analysis, potentially leading to the issue that binary file modifications dominate the recorded weights. We take an information-theoretic approach to enable the detection (and potential exclusion) of such modifications. In particular, we compute the entropy S of code before and after the change, defined as:

$$S = -\sum_k \mathbf{p}_k \log_2(\mathbf{p}_k) \tag{1}$$

This computation is based on the `utf-8` encoding space with 256 possible symbols. Entries of the vector $\mathbf{p}$ represent a

	code	entropy
a	`for x in 'hello world': print(x)`	3.94
b	`for c in 'hello world': print(c)`	3.94
c	`d = {x[0]:x[1] for x in df['d']}`	3.80
d	`Uatsffm+BC+s7kWKqVpMlrMEWk7nTfK1`	4.41

Fig. 1. Entropy of equal length strings based on discrete `utf-8` (256 possible symbols) probability space. The entropy can take values between 0 and 8 bits. The entropy of `base64` encoded image (**d**) is considerably higher than of typical lines of (`python`) code (**a**–**c**). In practice the effect is amplified as strings of binary encoded images are longer. Small changes within a line have a small or no effect on entropy as can be seen in the entropy difference between **a** and **b**.

symbol's normalised frequency in a given string. Given this definition, the entropy S can take values between 0 and 8 bits. Some examples for this measure are given in Figure 1. The resulting distribution of entropy for all co-edits can be used for a Bayesian classification distinguishing, e.g. binary encoded images or hashes from natural language or source code changes.

In the discussion above, we have considered a purely line-based approach, which treats every modified line of code as a separate entity. However, it is common that developers edit contiguous regions of code, consisting of multiple adjacent lines, with a single modification. As illustrated in Figure 2, `git2net` therefore provides an option to analyse co-edits at the granularity of such contiguous code regions rather than lines. Compared to previous approaches, which have used programming language constructs like functions to identify co-edits at a granularity smaller than files [5], this approach has the advantage that it is agnostic of the programming language. It further allows to analyse co-edit relations in files that do not represent source code, e.g. in text documents.

To explain our approach of identifying edited *blocks* of code, we distinguished between different cases contained in Figure 2: For deleted lines (e.g. line 2 in Fig. 2) a normal co-editing relationship is recorded. As the effort required for deletions can vary both between projects and the type of analysis performed, we mark these cases in the database but do not specify a Levenshtein edit distance. Edits exclusively consisting of

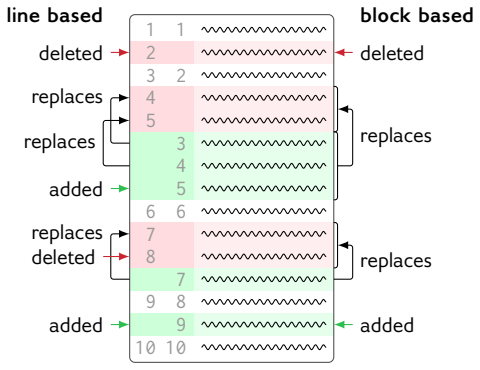

Fig. 2. Identification of replacements using line- and block-based analysis.

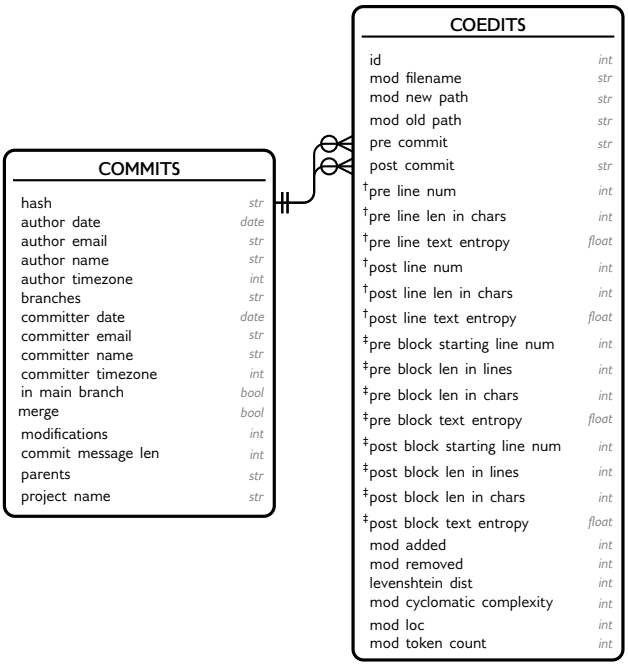

Fig. 3. Relations in the co-editing database. Elements marked † only occur for line based analysis, whereas entries marked ‡ are specific to block based analysis.

added lines are recorded in the database but not considered as co-edits (neither by a line-based nor by a block-based approach) as no previous author exists. The Levenshtein edit distance for pure additions matches the number of characters that were added. For cases where a set D of deleted lines is replaced by a set A of added lines, the line-based approach matches each line $d_i \in D$ with a line $a_i \in A$ for $i \leq \min(|D|, |A|)$. If $|D| < |A|$, a line-based approach would thus treat the excess lines in A as added lines, thus not considering them as a co-edit. This is the case in line 4-5 in Fig. 2. With our block-based approach, we instead identify that a block of lines (lines 4-5) in the original file is replaced by a new block (lines 3-5) in the new file. If $|D| > |A|$, a line-based approach identifies the excess lines in D as deleted lines (see line 7-8 in Fig. 2). Through a block-based analysis we are instead able to identify that a block of lines (lines 7-8) in the original file is replaced by a new block of lines (line 7) in the new file.

While for the line-based approach, all editing statistics such as the Levenshtein edit distance or the entropy are computed on pairs of lines (d_i, a_i), the block based approach considers the set of lines in A as a replacement of the lines contained in D. Consequently all statistics are computed for the pair of code blocks (D, A).

After evaluating each commit, results are written to an `sqlite` database. This allows to pause and resume an analysis at any point in time and helps to prevent data loss from system crashes. The resulting database scheme is shown in Figure 3.

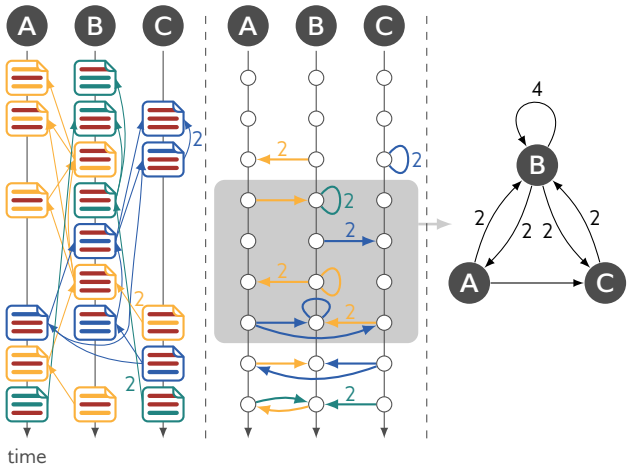

Fig. 4. Process of generating a co-editing network from git commits. To enhance readability, each commits only modifies a single file. Three different colour coded files are considered. Edited lines are shown in red. For all edits, edges to the commit containing the original line are shown on the left hand side. Link weights are determined based on the number of lines changed. A time stamped link between the authors of the modified lines is recorded once the edit takes place (cf. centre figure). The resulting set of time stamped edges can either be analysed itself or aggregated into co-editing networks via a sliding window analysis as shown on the right. Unless indicated otherwise, all edge weights are 1.

B. From Co-Edits to Networks

Given the database of co-editing relationships generated by the approach described above, `git2net` provides procedures to generate three different types of network projections: (i) co-editing networks, (ii) directed acyclic graphs of edit sequences for a given file, and (iii) bipartite networks linking developers to edited files.

The process of generating co-editing networks is illustrated in an example shown in Figure 4. The left column shows three developers (A, B, and C) editing three colour-coded files. Modified lines are shown in red. Edges between files represent the number of overlapping lines, which for illustrative purposes we show instead of the more granular Levenshtein edit distance. Given these edges, we generate a temporal network connecting the developers (cf. Fig. 4, centre for a time-unfolded representation). A link $(A,B;t,w)$ in this network represents a commit by developer A at time t in which w lines originally authored by developer B are modified. By the aggregation of time-stamped links over a (moving) time window we obtain co-editing networks as shown in the right column of Figure 4.

Apart from co-editing networks, `git2net` supports the construction of file-based directed acyclic graphs (DAGs) of commits based on co-editing relationships. Each path in this DAG represent a sequence of consecutive co-editing relationships of developers editing the given file, i.e. a sequence of commits containing file modifications that built upon each other. The nodes in this graph represent commits and edges represent co-editing relationships between the authors of the commits. An example for the construction of such a DAG from a set of five commits containing file modifications is shown in Figure 5. Individual connected components of the DAG represent proxies of knowledge flow for this file. This has been highly valuable in our own research as it immediately allows the extraction of paths from the co-editing relationships. Analysing these paths with the methods provided by the software package `pathpy` [46] allows to trace knowledge flow within specific areas of the development—a topic we identified as highly relevant in discussions with practitioners from software development companies.

To additionally facilitate coarse-grained analyses at the level of file-based coauthorship relations, `git2net` finally supports the construction of bipartite file-developer networks, where directed links $(d,f) \in D \times F$ indicate that a developer $d \in D$ has modified a file $f \in F$.

C. Usage of git2net

`git2net` comes as a `python` package that can be installed via the `python` package manager `pip`. During the installation all dependencies, which consist of the `python` packages `pandas`, `python_Levenshtein`, `pyDriller`, `progressbar2`, and `pathpy`, will be installed automatically. `git2net` runs on all major operating systems and has been tested under Windows, Mac OS X, and Linux. Assuming that the `git` repository that shall be examined has been cloned to a directory `repo`, our tool can be launched by the command

```
./git2net.py mine repo coedits.db
```

where `coedits.db` indicates the `sqlite` database file where the results will be stored. An optional parameter `--exclude` can be used to pass a text file that contains paths of files or directories in the repository tree that shall be excluded from the analysis. In our own analyses of a large commercial software project, this function has proven crucial to exclude directories containing large binary files or external Open Source software dependencies that would considerably distort the analysis. While the analysis of co-edited code uses the line-based approach described above by default, an optional command line switch `--use-blocks` can be used to use the block-based extraction of co-editing relations instead.

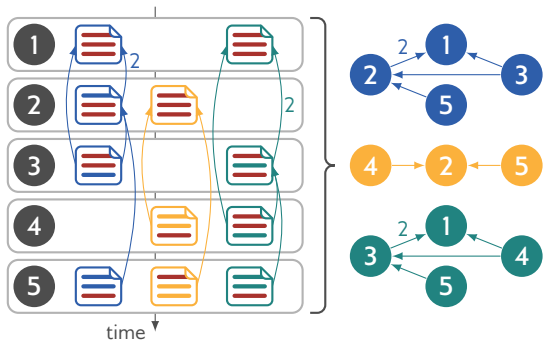

Fig. 5. Process of creating file based directed acyclic co-editing graphs. The left hand side shows a set of commits modifying three colour coded files. For each file a directed acyclic graph is generated linking consecutive commits with overlapping changes.

In addition to the command line interface outlined above, `git2net` provides an API that can be used for the development of custom repository mining scripts. In particular, the API provides methods that allow to extract co-edit relations from individual commits that can be passed as `PyDriller` objects. It can further be used to augment the analysis of edited code blocks by advanced text mining and code analysis techniques. In order to generate network projections based on a database of co-edits, `git2net` can be launched with the command

```
./git2net.py graph [type] coedits.db graph.csv
```

where `type` can be `--coedit`, `--bipartite`, or `--dag`. Depending on the choice, `git2net` generates a projection of the co-editing database in terms of a temporal co-editing network (cf. Fig. 4), a bipartite network linking authors to files, or a directed acyclic co-editing graph (cf. Fig. 5) respectively.

All networks can be exported in a csv-based format that can be read by popular network analysis packages like `igraph` [47], `graphtool`[2], `Gephi` [48], and NetworkX [49]. Time-stamped co-editing networks can further be exported in a format that can be read by the dynamic network analysis and visualisation packages ORA [45] and `pathpy` [46] via the provided API. Moreover, all networks can be exported in terms of dynamic and interactive `d3js` visualisations, which directly run in any `HTML5`-compliant browser.

D. Experimental Evaluation of Scalability

We conclude this section by an experimental evaluation of the scalability of `git2net`. In particular, our tool facilitates the analysis of large repositories thanks to the automatic utilisation of multiple processing cores. By default, `git2net` uses all available processing core, creating multiple child processes that extract co-edits from independent commits in parallel. Through an optional command line switch `--no-parallel`, multi-core processing can be deactivated. An optional command line parameter `--numprocesses N` further allows to limit multi-core processing to at most *N* processing cores. Similarly, the API exposed by `git2net` provides parameters that can be used to control multi-core processing.

In order to evaluate the scalability gains provided by the parallel processing model, we performed an experiment using real-world data. We specifically cloned the `git` repository of the Open Source software `igraph` [50] and used `git2net` to extract line-based co-editing relationships. We then measured the time needed to analyse the full `git` history with close to 6,000 commits and approximately 35,000 file edits over a period of 14 years. We repeated this experiment multiple times, using different numbers of processing cores on a recent 16 core desktop processor[3].

Figure 6 shows the time required to extract all co-editing relationships from the repository of `igraph` (y-axis) plotted against the number of processing threads (x-axis). Up to the number of physical processing cores of the machine (16) we observe an almost perfect linear scaling of processing time, cutting down processing time from close to one hour (single-threaded) to less than 5 minutes. Starting from 16 processing cores we observe deviations from the linear scaling that are likely due to the synchronised writing to the `sqlite` database. This deviation from the linear scaling is naturally intensified as we exceed the number of physical processing cores, additionally utilising logical cores exposed through Intel's implementation of HW-based multi-threading.

[2]https://graph-tool.skewed.de/

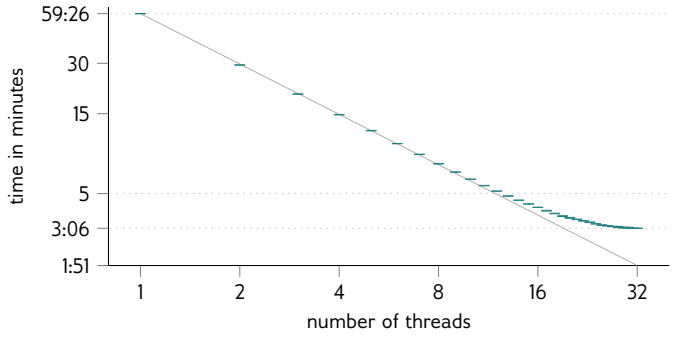

Fig. 6. Time required to analyse the `git` repository of the software package `igraph` [47] for different numbers of parallel processing threads. Both axes are logarithmic. Bars show the mean and standard deviation of three runs. The grey line shows a perfect linear scaling based on the time required by a single-threaded analysis.

IV. Exemplary Co-Editing Analysis of an Open Source and Commercial Project

Having discussed the implementation, usage, and scalability of our tool, we now demonstrate its usefulness through four short exemplary studies of real-world software projects. We apply `git2net` to (i) the `gitHub` repository of the Open Source network analysis software `igraph` [47], and (ii) a large `git` repository of a commercial software project obtained via an industry collaboration with the software company Genua. We specifically demonstrate (A) the construction of different static network projections capturing co-editing, co-authorship, and code-ownership relations, (B) a comparative study of fine-grained co-editing networks vs. coarse-grained co-authorship networks generated at the level of files, (C) the analysis of dynamic co-editing networks by means of temporal network analysis techniques, and (D) a comparison of temporal co-editing patterns between an Open Source and a commercial software project. These case studies should be seen as seeds for future work that demonstrate the usefulness of our approach rather than as conclusive analyses. To support such future studies, the co-editing relationships extracted from the Open Source project `igraph` are available on `zenodo.org` [51].

A. Static Network Projections

To demonstrate our tool, we illustrate the three different network projections introduced in III, using the co-edit information extracted from the public `git` repository of the network analysis package `igraph` [50]. The resulting networks are shown in Figure 7.

[3]Intel® Core™ i9 7960X, 16C/32T, 2.80GHz base, 4.2GHz boost

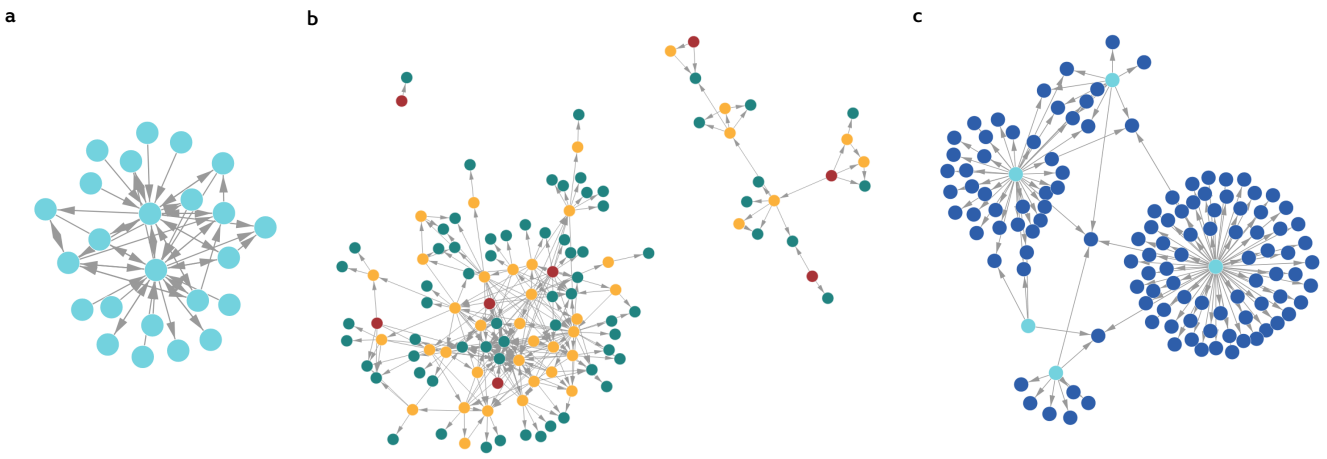

Fig. 7. Three examples for time-aggregated collaboration networks generated by `git2net` based on co-editing relations in `igraph` project: **a** shows a time-aggregated, static, directed network of co-editing relations. Each node represents one developer, while a directed link (A,B) indicates that at some point in the development history developer A edited at least one line of code previously written by developer B. **b** shows a directed acyclic graph of edits of the source code file `flow.c`. Nodes represent commits by developers. Root nodes with in-degree zero are marked in red, leaf nodes with out-degree zero are marked in green, intermediary nodes are marked in red. **c** shows a bipartite network linking developers (lightblue) to the files that they edited (blue).

Figure 7a shows a static co-editing network where nodes represent developers. For this initial demonstration we employ a time-aggregated projection, i.e. we use time-stamped co-editing relations $(v,w;t)$ capturing that at time t a developer v edited code originally written by developer w to construct a time-aggregated graph $G(V,E)$ where $(v,w) \in E$ iff $\exists \tau : (v,w;\tau)$. The directionality of links in this projection allows us to distinguish between team members with different roles: Nodes with zero in-degree, i.e. developers with no incoming co-edit relations, have never contributed code that was subsequently revised by other developers. Nodes with zero out-degree, i.e. developers with no outgoing co-edit relations, have never revised code that was originally authored by other developers. Such a maximally simple static projection can thus give a first "birds-eye" view of the collaboration and coordination structures in a software developing team. It highlights pairs of developers who exhibit strong mutual co-editing relations as well as pairs of developers working independently. This analysis can be refined by taking into account the time stamps of co-editing events, which we will do in section IV-C. In section IV-B we further discuss the difference between file-based coauthorship networks considered in prior works and the static projection of a fine-grained line-based definition.

Apart from co-editing relations between developers, in section III we have argued that `git2net` also provides a new perspective on the history of commits modifying a *given* file in the repository. In particular, this information can be used to a construct a directed acyclic graph of commits, where a link (v,w) in the graph indicates that commit w edited a region of source code originally contributed in commit v. Hence, each path from a root node r to a leaf node l in the resulting directed acyclic graph can be interpreted as a time-ordered sequence of commits that transforms code originally introduced in commit r into the "final" version contained in l. We highlight that this projection is different from commonly studied commit graphs, which link each commit to their parent commit independent of whether there is an overlap in the edited code. Fig. 7b illustrates this idea. It shows the directed acyclic graph of commits for the source code file `flow.c` in `igraph` [50]. Root nodes (with in-degree zero) in which the original version of a region of source code was committed are shown in red, while the commits containing the "final" version of code regions (out-degree zero) are highlighted in green. Intermediary nodes (yellow) represent commits that have both (a) edited code originally contributed in a previous commit and (b) contributed new code that is being revised in a subsequent commit. The analysis of such directed acyclic graphs can give insights into the complexity of code edits and their distribution across the team or across time. They further provide a novel abstraction that can be useful for the comparison of software artefacts, development processes, or projects.

In order to make it easy to reproduce file-based definitions of co-authorships used in the literature, `git2net` finally supports the construction of networks linking developers with the files that they have edited. The time-aggregated bipartite network resulting from the file edits made in the year 2016 for the project `igraph` is shown in Fig. 7c. Apart from being a basis for the construction of file-based coauthorship networks, this simple representation can give a coarse-grained view of code ownership and the distribution of contributions across the development team.

B. Co-editing vs. co-authorship networks

As outlined in section II, the analysis of co-authorship networks that capture which developers have contributed to the same files has received significant attention. At the same time, recent works have argued for more fine-grained definitions of collaboration networks, using e.g. function points or code lines [5, 6]. We contribute to this discussion and investigate the differences between a line- and a file-based approach to con-

a

	network	nodes	edges	δ
Open Source	co-authorship	24	67	1.22
	co-editing	24	55	
commercial	co-authorship	84	2551	2.11
	co-editing	84	1211	

b

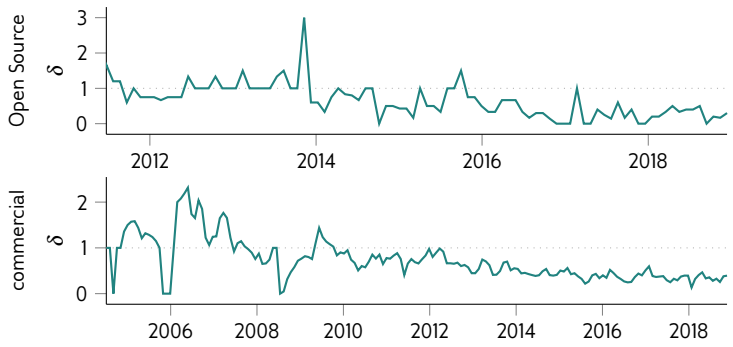

Fig. 8. Comparative analysis of file-based co-authorship vs. line-based co-editing networks. **a** Number of nodes and edges of networks aggregated over the entire project duration. Here, the co-authorship network overcounts relationships as editing the same file does not require a co-editing relationship on a line basis. **b** Proportion of edges in both networks over a moving 90 day window. Here, the co-authorship network frequently does not display links present in the co-editing network, as with co-editing links interactions with developers not contributing code in the present time window can be considered.

struct developer collaboration networks. Our results show that (i) this choice of granularity has considerable influence on the resulting network topologies, (ii) that the resulting differences are project-dependent, and (iii) that the differences between the resulting networks exhibit temporal inhomogeneities.

For our analysis, we first use `git2net` to extract (a) a file-based co-editing network G_f (which for simplicity we call co-authorship network), and (b) a line-based co-editing network G_l for the Open Source project `igraph` as well as for a large commercial software project. For both networks, we compare the time-aggregated projections (constructed as described in IV-A) and the sequence of networks obtained via a rolling window analysis. For each time window (as well as for the time-aggregated network), we then quantitatively assess the difference between G_l and G_f. We first observe that the set of nodes in both networks is necessarily the same. As a maximally simple approach to assess the difference between the two networks, we can thus calculate $\delta := \frac{m_f}{m_l}$, where m_f and m_l are the number of links in the file-based co-authorship network and the line-based co-editing networks, respectively.

Figure 8 shows the result of this analysis. Fig. 8a confirms that the file-based co-authorship network does not resolve where in the file edits take place, leading to a significantly higher number of links compared to the co-editing network in both projects. We expect many of these additional links to be *false positives*, in the sense that despite two developers having made edits to the same file no actual *collaboration* on the same code actually occurred.

Fig. 8b highlights the temporal dimension of these differences. It shows the time-evolving difference between the two network abstractions, using a 90 day moving window. For each window, the difference δ between the two networks is reported. Importantly, we observe time windows where $\delta < 1$, which indicates that the line-based co-editing networks feature additional links over the file-based co-authorship network. This is due to the fact that a file-based (temporal) co-authorship network does not consider commits to files made outside the time window currently analysed. However, our detailed analysis of co-edit relations can nevertheless identify that at time t within the time window developer A has edited code originally authored by developer B in a commit outside the time window. We argue that neglecting this relation introduces the risk of *false negatives*, in the sense that we would omit the need of collaboration or coordination associated with a commit occurring at time t. This subtle but important difference highlights the limitations of a simple file-based extraction of collaboration networks and showcases the advantage of our approach.

C. Analysis of Temporal Co-Editing Networks

A major advantage of `git2net` is its support for the extraction of *dynamic* co-editing networks with high temporal resolution. To showcase the benefits of such a temporal analysis for the two projects mentioned above, we have used `git2net`'s `python` API to extract a time-stamped co-editing network from the repositories of the two projects mentioned above. We then used the temporal network analysis package `pathpy` [46] to apply a rolling window analysis, which provided us with a time series of network analytic measures. Figure 9 shows the resulting time series for four measures both for the Open Source project `igraph` as well as the commercial software project. The first row gives the number of developers working on the projects in a 365-day sliding window. The number of unique co-editing relations between these developers, shown in the second row, can be used to proxy the amount of collaboration on joint code regions taking place in a project in a given time window. We observe that the number of such collaborations relative to the number of developers is considerably higher for the commercial software project compared to the Open Source project. This finding is further corroborated by the mean out-degree of nodes shown in the third row. This suggests that on average developers in `igraph` edit the code of one to two other developers, while for the commercial software project each developer has to coordinate his or her changes with four to eight other team members. It is a remarkable finding for the commercial software project that both the number of unique directed edges and the mean out-degree decline from 2013 onwards, despite the growing number of developers. This

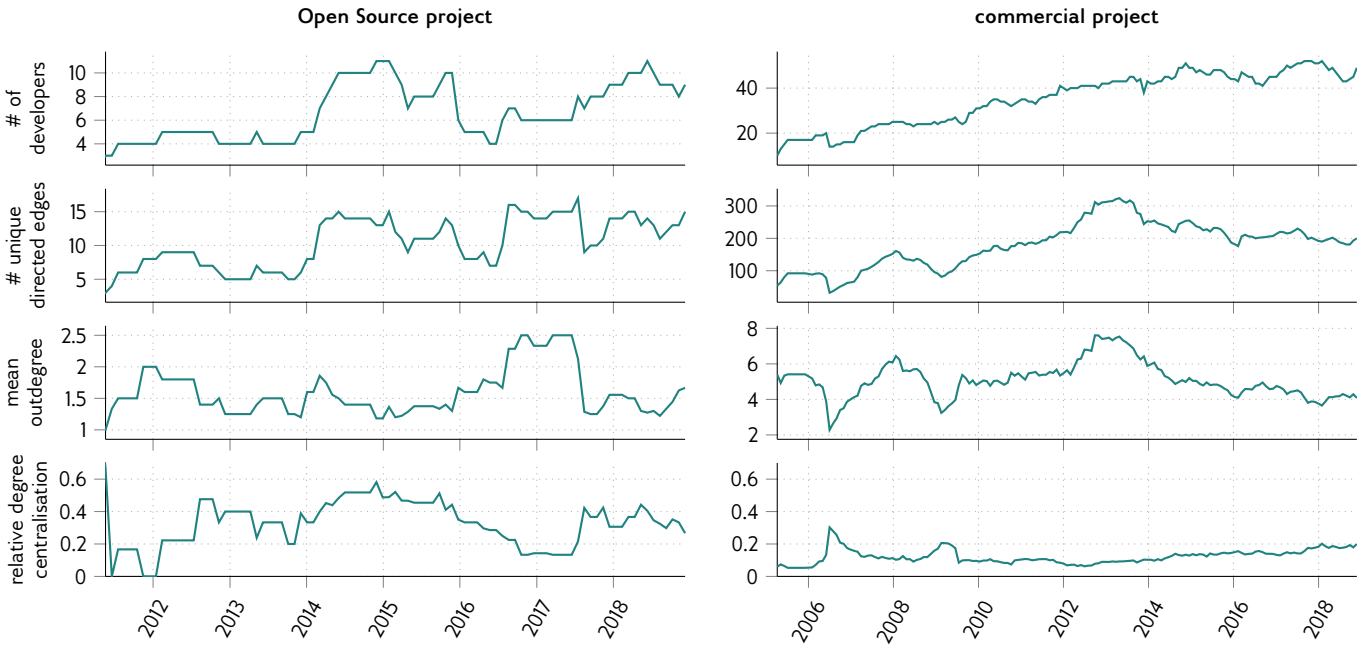

Fig. 9. Time series of different (network-analytic) measures for the time-stamped co-editing networks of an Open Source (left) and commercial software project (right). Results were generated using a rolling window analysis with a window size of 365 days and 30 day increments.

could mark a change in the software development processes and/or the social organisation of teams. While a first feedback from the project managers suggests that this could be related to a change in the adoption of an agile development model, testing this hypothesis requires a separate in-depth study. Finally, in the fourth row in Figure 9 we report the evolution of normalised (total) degree centralisation over time [52]. A minimum value of zero indicates that all nodes in the network have the same degree, while a maximum value of one corresponds to a perfect star network where all nodes except a hub node have degree one. We find that `igraph` exhibits considerably larger degree centralisation than the commercial software project, which is likely related to previous findings of highly skewed distributions of code contributions in Open Source projects [6, 53, 54].

D. Editing of Own vs. Foreign Code

In a final experiment, we showcase how `git2net` can be used to analyse temporal co-editing patterns in software development teams. To this end, we extend our analysis of the mere *topological* dimension of co-editing relations performed in previous sections, to use additional information on the Levenshtein distance associated with these relations. The Levenshtein distance between two source code versions captures the number of characters one has to type to transform one string into another string. It has been used as a proxy for development effort associated with commits [6]. Extending this approach, an interesting aspect of our methodology is that it allows us to distinguish between (i) the cumulative Levenshtein distance of code edits made in a developer's *own* code and (ii) the cumulative Levenshtein distance of edits made in *foreign* code, i.e. code originally written by other developers. This enables us to calculate, for each time window in the commit history of a project, the relative proportion of development effort falling into these two categories.

Figure 10 shows the result of this analysis for the two projects introduced above, where the top-part of the figure reports the total number of (unweighted) co-edit relations, while the bottom part shows the relative proportion of the total Levenshtein distance of own code changes vs. foreign code changes. This analysis highlights considerable project- and time-dependent differences. For the Open Source project `igraph`, during a first phase from 2006 to 2015, the majority of code edits take place in code previously written by the same developer. This indicates a strict notion of code "ownership", where developers rarely touch code written by others. For the commercial software project we observe a completely different dynamics, where for the majority of time windows development effort is dominated by *foreign* code edits. We hypothesise that this finding is likely related to code changes triggered by the specific implementation of the code review process in the commercial software project [55]. While an in-depth study of this claim goes beyond the scope of this tool paper, this finding highlights a specific research question that can be addressed with our tool in future work.

V. Conclusion and Outlook

Over the past two decades, the analysis of co-authorship, co-commit, or co-editing networks in software development teams has experienced huge interest from the empirical software engineering and repository mining community. Exemplary studies have shown that the analysis of such collaboration

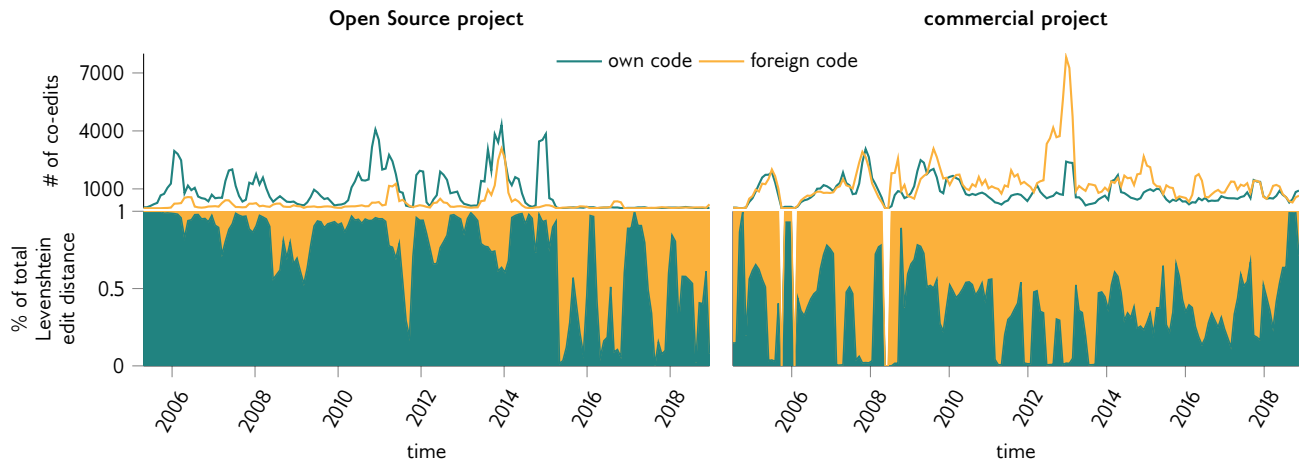

Fig. 10. Editing of own and foreign code for Open Source and commercial project over time. The total number of edited blocks is shown above whereas the bottom figures show proportions of the total Levenshtein edit distance. Results are computed on a 90 day rolling window with 30 day increments.

networks helps to assess the time-evolving social structure of teams [6, 33], predict software defects [34], categorise developer roles [39], identify communities [5], or study knowledge spillover across individuals, teams, and projects [4, 36, 38, 41]. Most of these studies have employed definitions of co-authorship networks which assume that developers are linked if they edited a common file, module, or binary. However, such coarse-grained definitions have been shown to neglect information on the microscopic patterns of collaborations contained in the time-ordered sequence of lines of code edited by developers [5, 6].

To facilitate data-driven studies of developer networks that take advantage of this detailed information, we have introduced `git2net`, a `python` package for the mining of fine-grained and time-stamped collaboration networks from large `git` repositories. Going beyond previous works, we adopt text mining techniques to assess (a) the development effort of an edit in terms of the Levenshtein distance between the version before and after the commit, and (b) the entropy of file modifications, which can be used to filter out changes in text-encoded binary data. Thanks to a parallel processing model our tool exhibits a linear speed up for an increasing number of processing cores. This makes `git2net` suitable to analyse `git` repositories with hundreds of thousands of commits and millions of lines of code.

Apart from a description of our tool, we have reported results of a case study using the repositories of an Open Source and a commercial software project. While the results are rather anecdotal and should thus not be generalised to other projects, this case study is meant to demonstrate that the presented tool simplifies the construction and analysis of dynamic developer collaboration networks and co-editing behaviour. It further showcases scenarios where our tool can be useful and highlights interesting research questions that we will address in future works.

Extending the analysis presented in [5], in section IV-B we report on a small comparative study of a file- vs. line-based construction of co-editing networks. A future systematic study of the differences between these approaches would be important. This should highlight in which case we need fine-grained methods and in which other cases coarse-grained notions of collaboration may be sufficient. Given the large number of studies using coarse-grained definitions of collaboration networks, such a study could make a substantial methodological contribution to the repository mining literature.

The results presented in section IV-C indicate topological differences between co-editing networks that are potentially linked to (a) the difference between Open Source and commercial software projects, and (b) the adoption of an agile development process in the commercial software project. These hypotheses must be tested in a larger corpus of projects that differ in these two dimensions. To support such a study, we recently mined co-editing relationships from the full `git` commit history of Linux[4], comprising more than 800,000 commits over a period of 18 years. Running `git2net` on a machine with 16 processing cores, we were able to complete the extraction of more than 60 million time-stamped co-editing relations in four days.

In section IV-D we further demonstrate that the information extracted by our tool can be used to generate a time-resolved breakdown of developer effort into (a) the revision of code authored by the developer him or herself vs. (b) the revision of code written by other team members. We currently work on a more systematic analysis of this interesting aspect of collaboration in development teams. Specifically, we study how collaboration is related to team size, project types, release schedules, code review processes, or the difference between Open Source and industrial projects.

Finally, a key advantage of our tool is that it provides a simple method to extract fine-grained collaboration networks at high temporal resolution from any `git` repository. Publicly available repositories cover a variety of different collaborative

[4]https://github.com/torvalds/linux

tasks, like software development, manuscript editing, web content management, etc. [56]. Our tool efficiently utilises the large number of such repositories and thus opens up a massive new source of high-resolution data on human collaboration patterns.

The fact that the resulting dynamic collaboration networks can be cross-referenced with project-related information (project success, organisational structures and project culture, developer roles, etc.) is likely to be of value for researchers in computational social science and organisational theory. We further expect the resulting corpus of data to be of considerable interest for the network science and social network analysis community, which have recently moved beyond moving window analyses, developing techniques that incorporate the chronological ordering of interactions in high-resolution time series data [30, 40, 57]. We thus hope that the tool and analyses presented in our work will serve the growing community of interdisciplinary researchers working at the intersection of data science, (social) network analysis, computational social science and empirical software engineering.

ACKNOWLEDGMENT

Ingo Scholtes acknowledges financial support by the Swiss National Science Foundation through grant 176938. We thank Alexander von Gernler as well as all other members of the software company GENUA for allowing us to validate our tool in a large commercial software project. All authors express their thanks to the anonymous reviewers of the manuscript.

TOOL AVAILABILITY, ARCHIVAL, AND REPRODUCIBILITY

The tool presented in this work is available as Open Source software package on `gitHub`[5]. `git2net` is further available via the `python` package index `pypi`, enabling users to simply install and update it via the package management tool `pip`. To support the reproducibility of our work, we have permanently archived the version of our tool that was used to obtain the results presented in this paper on the open-access repository `zenodo.org` [51].

`git2net` comes with unit tests and a comprehensive inline documentation. To support users in developing their first analysis, we further provide access to interactive `jupyter` notebooks, which allow to reproduce our approach.

Since the submission of this paper, the following additional features have been added to the release version of `git2net`:

- Extraction of *line-editing networks*, where nodes represent states of content lines of files, while edges link consecutive versions.
- Detection of copying and moving lines both within and between files for the file-based approach via the `git blame -C` option.
- Extraction of edits rather than co-edits in the `sqlite` database. With this also pure additions are listed in the database allowing users of `git2net` to implement own co-editing measures, e.g. based on the distance (in line numbers) of an addition to other lines.

[5]https://github.com/gotec/git2net

REFERENCES

[1] G. Gousios and D. Spinellis, "Ghtorrent: Github's data from a firehose," in *Mining software repositories (msr), 2012 9th ieee working conference on*. IEEE, 2012, pp. 12–21.

[2] ——, "Mining software engineering data from github," in *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*. IEEE, 2017, pp. 501–502.

[3] K. M. Carley and W. A. Wallace, "Computational organization theory," in *Encyclopedia of Operations Research and Management Science*. Springer, 2001, pp. 126–132.

[4] G. Von Krogh and E. Von Hippel, "The promise of research on open source software," *Management science*, vol. 52, no. 7, pp. 975–983, 2006.

[5] M. Joblin, W. Mauerer, S. Apel, J. Siegmund, and D. Riehle, "From developer networks to verified communities: A fine-grained approach," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 563–573. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818824

[6] I. Scholtes, P. Mavrodiev, and F. Schweitzer, "From aristotle to ringelmann: a large-scale analysis of team productivity and coordination in open source software projects," *Empirical Software Engineering*, vol. 21, no. 2, pp. 642–683, Apr 2016. [Online]. Available: https://doi.org/10.1007/s10664-015-9406-4

[7] D. Spadini, M. Aniche, and A. Bacchelli, *PyDriller: Python Framework for Mining Software Repositories*, 2018.

[8] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8, 1966, pp. 707–710.

[9] C. E. Shannon, "A mathematical theory of communication," *Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

[10] T. Wolf, A. Schröter, D. Damian, L. D. Panjer, and T. H. Nguyen, "Mining task-based social networks to explore collaboration in software teams," *IEEE Software*, vol. 26, no. 1, pp. 58–66, 2009.

[11] T. Xie, S. Thummalapenta, D. Lo, and C. Liu, "Data mining for software engineering," *Computer*, vol. 42, no. 8, 2009.

[12] M. Cataldo, I. Scholtes, and G. Valetto, "A complex networks perspective on collaborative software engineering," *Advances in Complex Systems*, vol. 17, no. 7-8, 2014. [Online]. Available: http://dx.doi.org/10.1142/S0219525914300011

[13] M. M. Geipel, K. Press, and F. Schweitzer, "Communication in innovation communities: An analysis of 100 open source software projects," *ACS - Advances in Complex Systems*, vol. 17, no. 07n08, p. 1550006, 2014. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S021952591550006X

[14] M. Cataldo and J. D. Herbsleb, "Communication networks in geographically distributed software development," in *Proceedings of the 2008 ACM conference on Computer supported cooperative work*. ACM, 2008, pp. 579–588.

[15] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," in *Proceedings of the 2006 international workshop on Mining software repositories*. ACM, 2006, pp. 137–143.

[16] T. Wolf, A. Schroter, D. Damian, and T. Nguyen, "Predicting build failures using social network analysis on developer communication," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070503

[17] A. Bacchelli, M. Lanza, and M. D'Ambros, "Miler: A toolset for exploring email data," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 1025–1027.

[18] Q. Hong, S. Kim, S. C. Cheung, and C. Bird, "Understanding a developer social network and its evolution," 2011.

[19] Q. Xuan and V. Filkov, "Building it together: Synchronous development in oss," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 222–233. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568238

[20] A. Guzzi, A. Bacchelli, M. Lanza, M. Pinzger, and A. v. Deursen, "Communication in open source software development mailing lists," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 277–286.

[21] Y. Long and K. Siau, "Social network structures in open source software

development teams," *Journal of Database Management (JDM)*, vol. 18, no. 2, pp. 25–40, 2007.

[22] J. Howison, K. Inoue, and K. Crowston, "Social dynamics of free and open source team communications," in *IFIP International Conference on Open Source Systems*. Springer, 2006, pp. 319–330.

[23] A. Sureka, A. Goyal, and A. Rastogi, "Using social network analysis for mining collaboration data in a defect tracking system for risk and vulnerability analysis," in *Proceedings of the 4th india software engineering conference*. ACM, 2011, pp. 195–204.

[24] M. S. Zanetti, I. Scholtes, C. J. Tessone, and F. Schweitzer, "Categorizing bugs with social networks: a case study on four open source software communities," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds. IEEE / ACM, 2013, pp. 1032–1041, http://dl.acm.org/citation.cfm?id=2486930. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486930

[25] ——, "The rise and fall of a central contributor: Dynamics of social organization and performance in the gentoo community," in *CHASE/ICSE '13 Proceedings of the 6th International Workshop on Cooperative and Human Aspects of Software Engineering*, 2013, pp. 49–56. [Online]. Available: http://dx.doi.org/10.1109/CHASE.2013.6614731

[26] T. Fritz, G. C. Murphy, and E. Hill, "Does a programmer's activity indicate knowledge of code?" in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 341–350. [Online]. Available: http://doi.acm.org/10.1145/1287624.1287673

[27] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code!: Examining the effects of ownership on software quality," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 4–14. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025119

[28] M. Greiler, K. Herzig, and J. Czerwonka, "Code ownership and software quality: A replication study," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, May 2015, pp. 2–12.

[29] A. C. MacLean and C. D. Knutson, "Apache commits: social network dataset," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 135–138.

[30] M. Newman, *Networks*. Oxford university press, 2018.

[31] M. M. Geipel and F. Schweitzer, "Software change dynamics: evidence from 35 java projects," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2009, pp. 269–272.

[32] M. M. Geipel, "Modularity, dependence and change," *Advances in Complex Systems*, vol. 15, no. 06, p. 1250083, 2012.

[33] G. Madey, V. Freeh, and R. Tynan, "The open source software development phenomenon: An analysis based on social network theory," *AMCIS 2002 Proceedings*, p. 247, 2002.

[34] A. Meneely, L. Williams, W. Snipes, and J. Osborne, "Predicting failures with developer networks and social network analysis," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '08/FSE-16. New York, NY, USA: ACM, 2008, pp. 13–23. [Online]. Available: http://doi.acm.org/10.1145/1453101.1453106

[35] M. Ogawa and K.-L. Ma, "Software evolution storylines," in *Proceedings of the 5th international symposium on Software visualization*. ACM, 2010, pp. 35–42.

[36] V. S. Vijayaraghavan, P.-A. Noël, Z. Maoz, and R. M. D'Souza, "Quantifying dynamical spillover in co-evolving multiplex networks," *Scientific reports*, vol. 5, p. 15142, 2015.

[37] L. Lopez-Fernandez, G. Robles, J. M. Gonzalez-Barahona *et al.*, "Applying social network analysis to the information in cvs repositories," in *International Workshop on Mining Software Repositories*. IET, 2004, pp. 101–105.

[38] S.-K. Huang and K.-m. Liu, "Mining version histories to verify the learning process of legitimate peripheral participants," in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, ser. MSR '05. New York, NY, USA: ACM, 2005, pp. 1–5. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083158

[39] M. Pohl and S. Diehl, "What dynamic network metrics can tell us about developer roles," in *Proceedings of the 2008 international workshop on Cooperative and human aspects of software engineering*. ACM, 2008, pp. 81–84.

[40] P. Holme, "Modern temporal network theory: a colloquium," *The European Physical Journal B*, vol. 88, no. 9, p. 234, Sep 2015. [Online]. Available: https://doi.org/10.1140/epjb/e2015-60657-4

[41] E. Cohen and M. P. Consens, "Large-scale analysis of the co-commit patterns of the active developers in github's top repositories," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, May 2018, pp. 426–436.

[42] F. Schweitzer, V. Nanumyan, C. J. Tessone, and X. Xia, "How do oss projects change in number and size? a large-scale analysis to test a model of project growth," *ACS - Advances in Complex Systems*, vol. 17, no. 07n08, p. 1550008, 2014. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S0219525915500083

[43] M. Cataldo, P. A. Wagstrom, J. D. Herbsleb, and K. M. Carley, "Identification of coordination requirements: implications for the design of collaboration and awareness tools," in *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*. ACM, 2006, pp. 353–362.

[44] T. Y. Berger-Wolf and J. Saia, "A framework for analysis of dynamic social networks," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2006, pp. 523–528.

[45] K. M. Carley and J. Pfeffer, "Dynamic network analysis (dna) and ora," *Advances in Design for Cross-Cultural Activities Part I*, pp. 265–274, 2012.

[46] Ingo Scholtes, "Software Package pathpy," http://pathpy.net, 2017, [Online].

[47] G. Csardi and T. Nepusz, "The igraph software package for complex network research," *InterJournal*, vol. Complex Systems, p. 1695, 2006. [Online]. Available: http://igraph.sf.net

[48] M. Bastian, S. Heymann, M. Jacomy *et al.*, "Gephi: an open source software for exploring and manipulating networks." *Icwsm*, vol. 8, no. 2009, pp. 361–362, 2009.

[49] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2008.

[50] G. Csardi and T. Nepusz, "The igraph software package for complex network research," *InterJournal*, vol. Complex Systems, p. 1695, 2006. [Online]. Available: http://igraph.org

[51] C. Gote, I. Scholtes, and F. Schweitzer, "git2net - An Open Source Package to Mine Time- Stamped Collaboration Networks from Large git Repositories," May 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2587483

[52] L. C. Freeman, "Centrality in social networks conceptual clarification," *Social networks*, vol. 1, no. 3, pp. 215–239, 1978.

[53] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and mozilla," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 3, pp. 309–346, 2002.

[54] Z. Lin and J. Whitehead, "Why power laws? an explanation from fine-grained code changes," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, May 2015, pp. 68–75.

[55] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?" in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 202–211. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597082

[56] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "An in-depth study of the promises and perils of mining github," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2035–2071, 2016.

[57] I. Scholtes, "When is a network a network?: Multi-order graphical model selection in pathways and temporal networks," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '17. New York, NY, USA: ACM, 2017, pp. 1037–1046. [Online]. Available: http://doi.acm.org/10.1145/3097983.3098145