Uncovering the Hidden Co-evolution in the Work History of Software Projects

Saimir Bala[1(✉)], Kate Revoredo[2], João Carlos de A.R. Gonçalves[2], Fernanda Baião[2], Jan Mendling[1], and Flavia Santoro[2]

[1] Vienna University of Economics and Business (WU), Vienna, Austria
{saimir.bala,jan.mendling}@wu.ac.at

[2] Federal University of the State of Rio de Janeiro (UNIRIO), Rio de Janeiro, Brazil
{katerevoredo,joao.goncalves,fernanda.baiao,flavia.santoro}@uniriotec.br

Abstract. The monitoring of project-oriented business processes is difficult because their state is fragmented and represented by the progress of different documents and artifacts being worked on. This observation holds in particular for software development projects in which various developers work on different parts of the software concurrently. Prior contributions in this area have proposed a plethora of techniques to analyze and visualize the current state of the software artifact as a product. It is surprising that these techniques are missing to provide insights into what types of work are conducted at different stages of the project and how they are dependent upon another. In this paper, we address this research gap and present a technique for mining the software process including dependencies between artifacts. Our evaluation of various open-source projects demonstrates the applicability of our technique.

Keywords: Artifact co-evolution · Work history dependencies · Project-oriented business processes · Software project mining

1 Introduction

Project-oriented business processes play an important role in various industries like engineering, health care or software development [2]. Such processes are characterized by the fact that work towards a predefined outcome involves complex tasks executed by different parties. Typically, these processes are not supported by a process engine, but their status is fragmented over different documents and artifacts. This is especially the case for software development processes: the expected outcome is the release of a new software version, but the different project members collaborate with tools like version control systems that are only partially aware of the work process.

This work has been partially funded by the Austrian Research Promotion Agency (FFG) under grant 845638 (SHAPE) and the RISE BPM project (H2020 Marie Curie Program, grant 645751). The second author was partially supported by PROAP/-CAPES and fourth and sixth authors by the National Council for Scientific and Technological Development (CNPq), Brazil.

© Springer International Publishing AG 2017
J. Carmona et al. (Eds.): BPM 2017, LNCS 10445, pp. 164–180, 2017.
DOI: 10.1007/978-3-319-65000-5_10

A key challenge for project-oriented business processes like software development is gaining transparency of the overall project status and work history. Literature has recognized that analyzing the evolution of business process artifacts in projects can help obtaining important clues about the project performance in terms of time [5], cost [22] and quality [13]. This is addressed by functionality of version control systems (VCS) to track versions and changes of informational artifacts like source code and configuration files. While prior research has presented various perspectives for analyzing software artifacts, e.g. [3,14,19,23], there is a notable gap on the discovery of dependencies in the work history. For these reasons, project managers often lack insights into side effects of changes in large software processes.

In this paper, we address this research gap by building on partial solutions from the separate fields of mining software repositories and process mining. More specifically, we develop a technique that uncovers non-hierarchical work dependencies which we call *hidden co-evolution*. This technique extracts the labeled work history from VCS repositories and identifies dependencies beyond simple hierarchical containment. In this way, we help the project manager to spot dependencies in the co-evolution of work histories of different information artifacts. Our technique has been implemented and evaluated using data from a diverse set of open source projects.

The paper is structured as follows. Section 2 describes the research problem along with its requirements and summarizes insights from prior research. Section 3 presents our approach in detail. Section 4 shows a prototypical implementation and evaluates its applicability both in a use case scenario and on real world projects from GitHub. Section 5 concludes the paper.

2 Background

This paper follows the Design Science Research (DSR) paradigm [16]. In this section, we describe the research problem in more detail and define requirements for a solution. Against these requirements, we analyze related work.

2.1 Problem Description

In this paper, we focus on a specific class of project-oriented business processes, namely software development processes. These processes share some common characteristics. First, they involve various resources with different roles. In the simplest case, we can distinguish *project managers* and *project participants*. Project managers are responsible for managing the development process and supervising the work of the project participants, who in turn are responsible for specific work tasks. Second, such processes are usually subject to constraints in terms of cost, time and quality, which is mostly associated with the performance of each of the work tasks. Third, the project participants work on a plethora of artifacts, which are logically organized in a hierarchical structure, with complex interdependencies among them. Given these characteristics, it is the goal of the

project manager to organize the software development process in such a way that the work on different files and tasks reflects the complex interdependencies, the constraints and the available participants. Therefore, it is important for the manager to understand the *work history* of the process in order to monitor the progress systematically.

Table 1. An excerpt of a VCS log data

Id	Project Participant	Date	Comment	Diff
1	John	2017-01-31 12:16:30	Create readme file	diff –git a/README.md b/README.md @@ -0,0 +1 @@ +# StoryMiningSoftwareRepositories
2	Mary	2017-02-01 10:13:51	Add a license	diff –git a/README b/README @@ -1,0 +2,3 @@ +The MIT License (MIT) + +Copyright (c) 2015 Mary+
3	Paul	2017-02-02 16:10:22	Updated the requirements.	diff –git a/README.md b/README.md @@ -1,4 +1,5 @@ + # string 1, string 2, string 3 diff –git a/requirements.txt b/requirements.txt @@ -0,0 +1 @@ +The software must solve the problems
4	Paul	2017-02-02 15:00:02	Implement new requirements	diff –git a/model.java b/model.java @@ -1,9 +1,10 @@ +public static methodA(){int newVal=0; @@ -21,10 +23,11 @@ + "1/0",,"0/0", diff –git a/test.java b/test.java @@ -0,0 +1,2 @@ +//test method A +testMethodA()

Software tools like Version Control Systems (VCS) do not provide direct support for monitoring work histories, but they provide a good starting point by continuously collecting event data on successive versions of artifacts. Table 1 shows an excerpt of log data, where the columns, from left to right, indicate the commit identifier, the project participant who committed the changes, the commit date, the comment written by the project participant and the files affected and the change performed[1]. In order to understand the work history and dependencies based upon such data, we identify three major requirements:

R1 (Extract the work history): Discover the process of how artifacts evolve in the project as a *labeled* set of steps. This requirement is difficult because the version changes of a commit in relation to a single file do not directly reveal which type of work has been done. Both commit messages and edit characteristics might inform the labeling.

[1] cf. unified diff format https://git-scm.com/docs/git-diff.

R2 (Uncover Work-Related Dependencies): Identify that certain work in one part of the project is connected with work in another part. This requirement is difficult because such dependencies might not only exist between files that reside in the same directory. For example, a change in a source code file might have the side effect of triggering work on a configuration file. We refer to this as *co-evolution* of these files.

R3 (Measure Dependencies): Determine how strong the co-evolution of different artifacts is. This requirement is difficult because measures of *strength* of dependencies and on the *distance* of dependent artifacts have to be devised.

2.2 Related Work

A solution addressing these requirements can partially build upon research in three main areas: *(i)* work on Mining Software Repositories (MSR); *(ii)* Process Mining (PM); *(iii)* and software visualization.

Table 2 shows that these streams of research have mutual strengths, but no contribution covers the full spectrum. In general, methods from MSR have a strength in analyzing dependencies in the structure of the software artifact, but an explicit consideration of the type of work is missing. Contributions in this area focus on the users and the artifacts, mining co-evolution or co-change of project parts [8,24] and network analysis of file dependency graph based on commit distance [1,23,25]. Hidden work dependencies are mentioned as *logical dependencies* [15]. Also techniques for trend analysis [20] and inter-dependencies

Table 2. Requirements addressed by literature and topics covered. Fulfills requirement (✓); Only addresses requirement (⋆)

Main area	Papers	R1	R2	R3	Description
MSR	Zaidman et al. [24]	⋆	✓	✓	Only two labels for processes
	Zimmermann and Nagappan [25]		✓	✓	Only functional dependencies
	Abate et al. [1]		✓	⋆	Only functional dependencies
	D'Ambros et al. [8]		✓	✓	
	Oliva et al. [15]		✓	✓	
	Weicheng et al. [23]		✓	✓	
	Ruohonen et al. [20]		✓	✓	
	Lindberg et al. [13]			⋆	Activity variations
PM	Kindler et al. [12]	✓			
	Goncalves et al. [9]	✓			
	Poncin et al. [17]	✓			
	Beheshti et at. [4]	✓	⋆		
	Mittal and Sureka [14]	⋆			Only bug resolution process
	Bala et al. [2]	⋆		⋆	Unlabelled Gantt chart
Visualization	Voinea and Telea. [21]	⋆	✓		Unlabelled processes
	Ripley et al. [18]	✓		✓	Unlabelled processes
	Greene and Fischer [10]		✓		

between developers [13] are proposed. However, none of these works considers the type of work being done in the process.

In the area of PM, research gives more emphasis to the different tasks of the process. Some works focus on applying process mining for software repositories [2,14,17]. In this context, approaches have been defined that use various queries to extract artifact evolution and resources [4,5]. There is research on identifying the tasks of the process by elicitation from unstructured data of user comments [9]. There are also process mining applications that focus on repetitive steps in software engineering, but not on singular project-oriented processes, such as [12]. All these works only consider the dependencies between work tasks to a limited extend.

There is also work in the area of software visualization. Visualization tools have been proposed in order to allow project managers to have a detailed overview of the software artifact being developed. These tools help to visually inspect artifacts similarities on different levels of granularity [21], observe artifacts evolution or project members contribution [10,18]. In general, they can be characterized as artifact-centric, and largely agnostic to the type of work being done.

In the following, we develop a technique that addresses the three requirements and informs prior research on how to extract work histories and to identify the co-evolution of certain parts of a project-oriented software process.

3 Conceptual Approach

We propose a technique to extract and represent the work history and the dependencies among artifacts of a project-oriented business process. The technique takes as input a VCS log and produces analysis data that describe the evolution of the artifacts, along with metrics about their distance and their similarity in terms of work. The process is depicted in Fig. 1 and consists of three successive steps towards extracting hidden work dependencies from VCS event data. The method works under three main assumptions. First, we assume a *meaningful tree structure*, i.e. the project participants organize the files in a representative hierarchy (e.g., spatially separating documentation from testing into different folders). Second, project participants perform *regular commits* in the VCS. Third, project participants write *descriptive comments* that allow other members to understand the changes.

The first step of the technique is the preprocessing of the VCS log received as input. The main goal of this phase is to generate a set of events and store them into a database. Second, we obtain different views on the stored events. In particular, we are interested in observing *(i)* all the commits that affected the files over time; *(ii)* the amount of change brought by the commits to the files; and *(iii)* the users who issued such commits. The third phase is responsible for considering the different perspectives defined by the project manager and through the generated views extract the necessary knowledge. In the following, we detail the formal concepts and the algorithm of our technique.

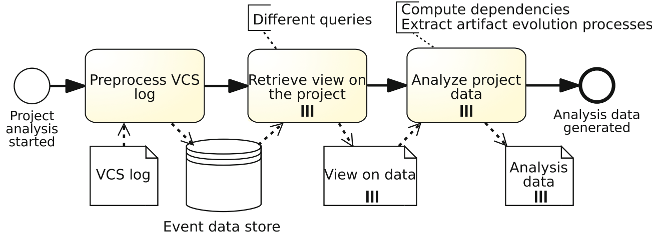


Fig. 1. Approach for generating analysis data from VCS logs

3.1 Preliminaries

As the objective of our technique is to uncover hidden work dependencies, we define the fundamental concepts required to capture them. Work is reflected by *artifacts*, e.g., word documents, spreadsheets, code, etc. Artifacts are leaves in the file tree hierarchy (with directories being special type of non-leaf files). Artifacts evolve over time, while project participants contribute their changes. Each change is an *event* that happens to an artifact in a single point in time. Events can be abstracted into *aggregated events* that allow a coarser grained view on the history. The history of the changes of an artifact over a time interval at a given level of abstraction is referred to as *artifact evolution*. Similar artifact co-evolution establishes a *dependency* between two artifacts.

A software product is subdivided into files and directories. In this work, we consider directories as special type of files which are parents of other files. Formally, let F be the universe of files in a software development project. Files are organized in a file tree. Therefore, each file $f \in F$ has one parent file. The only file without a parent file is the *root* file. We capture this information in the parent relation $Parent : F \times F$. For example, let $f_p \in F$ be the parent of file $f_c \in F$, then $(f_p, f_c) \in Parent$. An *artifact* is a file that is not a parent file, i.e. a file f_a is an artifact if $\forall_{f \in F}(f_a, f) \notin Parent$.

When project participants do a certain amount of work and want to save their current progress, they commit the changes to the VCS. We define changes on artifacts as the *events* of interest on the lowest granularity.

Definition 1 (Event). *Let E be the set of events. An* event *$e \in E$ is a five-tuple (f, ac, ts, k, u), where*

- *$f \in F$ is the affected artifact of the event.*
- *$ac \in AC = \mathbb{N}$ is the amount of change done in the artifact.*
- *$ts \in TS = \mathbb{N}$ represents a unix time stamp marking the time of the event occurrence.*
- *$k \in \Sigma^*$ is a comment in natural language text.*
- *$u \in U$ is the project participant responsible for the change.*

For event $e = (f, ac, ts, k, u)$ we overload f, ac, ts, k and u to be used as accessor functions. For example, f is the function $f : E \to F$ mapping an event to its affected artifact.

In some situations, it can be interesting to have a higher level overview of the changes done to a particular artifact. In this case, an aggregation of events related to this artifact in an interval of time can be performed. The time window for the aggregation, henceforth denoted as tw_{agg}, must be defined, i.e. the size of the time interval. For instance, a time window for aggregation can be a day. Thus, all events occurring for an artifact in the same day will be aggregated. An *aggregated event* is defined as follows:

Definition 2 (Aggregated Event). *An* Aggregated Event *for* tw_{agg} $(AE_{tw_{agg}})$ *is a five-tuple* (f, aac, ats, ak, au), *where*

- $f \in F$ *is the affected artifact in the set of events being aggregated.*
- $aac \in AAC = \mathbb{N}$ *is the aggregate amount of change done in the artifact for* tw_{agg}. *It is calculated by summing the amount of changes done in each of the time aggregated.*
- $ats \in ATS = \mathbb{N}$ *represents an aggregate time of the unix time stamp of the events being aggregated.*
- $ak \in \Sigma^*$ *is the concatenation of the comments presented in the events being aggregated.*
- $au \subseteq U$ *are the project participants responsible for the changes in* tw_{agg} *being aggregated.*

The set of aggregated events for a particular artifact defines how this artifact evolves over time. Considering an interval of analysis, henceforth denoted as *ia*, we define artifact evolution as follows.

Definition 3 (Artifact Evolution). Artifact evolution *is the process describing how the file* f *changed over an interval of time* ia, *i.e., a set of labeled tuples* $A_{evo}(f) = \{(t, a, l) | e \in AE_{ia}, f = f(e), t = ats(e), a = aac, l = ak(e)\}$ *chronologically ordered.*

Note that artifact evolution represents the changes that happened to a file over time. Thus, we can build the time series of a file f as the vectors of changes $\boldsymbol{X_f} = (a_1, ..., a_n)$ in the time window $tw_{agg} = [t_1, t_n]$, with a_i being the sum of the changes of f in of the aggregated intervals t_i of the time window tw_{agg}.

We measure the dependency between two files f_a and f_b in terms of their *degree of co-evolution* as follows.

Definition 4 (Degree of Co-Evolution). *Given two files* f_a *and* f_b, *the* degree of co-evolution $\chi : F \times F \rightarrow [0, 1]$ *is a similarity function of the respective time series.*

In this paper, we fix $\chi(f_a, f_b) = |\sigma(\boldsymbol{X_{f_a}}, \boldsymbol{X_{f_b}})|$, where σ is the correlation function of the two vectors $\boldsymbol{X_{f_a}}$ and $\boldsymbol{X_{f_b}}$.

The way files are kept in the directory structure establishes an inherent relationship among files being stored close to each other in the hierarchy. For instance, files serving the same purpose are stored close to each other in the file system. Hidden work dependencies are expected to happen between artifacts that are distant in the file structure. We measure this distance as the

length of the shortest route connecting two files in the file tree. We adapt the notion of path from [11] to our file tree. Given a file f, the path to the root node can be obtained by navigating the *Parent* relationship up to the root file. The path p from f_a to the root f_r is the set of parent files encountered along such route. i.e. $p(f_1, f_r) = \{(f_1, ..., f_k, f_{k+1}, ..., f_r)\}$ such that for any k, $(f_{k+1}, f_k) \in Parent$. The length of the path is the cardinality $|p|$ of the set. The shortest path between two files f_a, f_b in a tree passes through the Least Common Ancestor (LCA) [6]. This is equivalent to considering the paths from the single files to the root node $p_a = p(f_a, f_r)$ and $p_b = p(f_b, f_r)$ minus their intersection $I_{p_a,p_b} = \{p(f_a, f_r) \cap p(f_b, f_r)\}$. Thus, we define the *file distance* as the length of the shortest path between two files f_a and f_b as follows.

Definition 5 (File Distance). *The distance $d : F \times F \rightarrow \mathbb{N}$ between two files belonging to the same directory structure is defined as the number of nodes in the minimum path connecting the two files in the project file tree: $d(f_a, f_b) = |p_a| + |p_b| - 2 * (|I_{p_a,p_b}|)$.*

3.2 Hidden Dependencies Discovery Algorithm

We are focused on finding interesting hidden work dependencies. These dependencies are typically reflected by changes that happen to couples of allegedly unrelated files during their evolution. This section details the procedure that implements the technique outlined in Fig. 1.

Algorithm 1 presents the steps required to explicate such hidden dependencies. The procedure `PreprocessLog`($\mathcal{L}$) in line 2 takes as input a VCS log $\mathcal{L}$ structured as in Table 1 and parses out work events at the granularity of line changes. These events are then stored into an event data storage. Events parsed from VCS logs contain rich information about multiple aspects of the work they reflect. In order to represent all these different aspects, we devised the entity-relationship data model. Hence, we are able to store all the information that is possible to obtain after parsing the VCS log. Furthermore, this step allows the user to obtain simple information, such as statistics on the project, already at an early stage of the procedure. The output of the `PreprocessLog`($\mathcal{L}$) step results in the storage of all the events E into a database.

Next, the iterative call of the procedure `RetrieveView`(E, query) in line 3 performs several querying the data storage containing the set E. For example, a possible query can obtain all the comments associated to each change of a specific file. To obtain information on the evolution of files, we query the database for the changes of all the files within a user defined time interval tw_{agg}. In general several time frames can be chosen, each of them producing a *view* V on the data, i.e., a set of aggregated events chronologically sorted within tw_{agg}. For example, users may be interested in artifact-views aggregated by day, by month, etc. Multiple *views* are possible by defining them in the queries parameter. We collect these views into a set $\mathcal{V} = \bigcup_{\mathsf{queries}} V$.

Algorithm 1. Generate project analysis data

Input : A VCS log $\mathcal{L}$
Output: A set of triples $\{(Dist, Stories, D_{co-evo})\}$, artifact evolutions, and dependencies
Data : E event set, $\mathcal{V}$ views set, $AnalysisData = \{(Dist, Stories, D_{co-evo})\}$, degree of co-evolution threshold γ, file distance threshold δ, user defined queries queries

1 $Files \leftarrow \emptyset$, $Stories \leftarrow \emptyset$, $TimeSeries \leftarrow \emptyset$, $AnalysisData \leftarrow \emptyset$, $\mathcal{V} \leftarrow \emptyset$, $A_{evo}(f) \leftarrow \emptyset$;
`/* Preprocess VCS log */`
2 $E \leftarrow$ `PreprocessLog(`$\mathcal{L}$`)`;
`/* Retrieve views on the project */`
3 **for** *i from* 1 **to** |queries| **do** $\mathcal{V} \leftarrow \mathcal{V} \cup$ `RetrieveView(`E, queries[i]`)`;
`/* Analyze project data */`
4 **foreach** *view* $V \in \mathcal{V}$ **do**
5   **foreach** *aggreagated event* $ae \in V$ **do**
6   **foreach** $f = f(ae), t = ats(ae), a = aac(ae), l = aak(ae) \in ae$ **do**
            `/* Construct the artifact evolution set for the file */`
7             $A_{evo}(f) \leftarrow A_{evo}(f) \cup \{(t, a, l)\}$;
            `/* Construct the process using story mining */`
8             $Stories \leftarrow Stories \cup (f,$ `StoryMining(`l`)`$)$;
            `/* Collect files and time series */`
9             $Files \leftarrow Files \cup \{f\}$;
10            $TimeSeries(f) \leftarrow$ construct time series from $A_{evo}(f)$;
11   **end**
12   **end**
13   **foreach** *pair of files* $i, j \in Files$ **do**
        `/* Compute degree of co-evolution */`
14        $coEvoDegree \leftarrow \chi(TimesSeries(i), TimeSeries(j))$;
        `/* Compute file distances */`
15        $distance \leftarrow d(i, j)$;
        `/* Select based on user defined thresholds */`
16   **if** $coEvoDegree > \gamma$ **then** $D_{co-evo} \leftarrow D_{co-evo} \cup \{coEvoDegree\}$;
17   **if** $distance > \delta$ **then** $Dist \leftarrow Dist \cup \{distance\}$;
18   **end**
19    $AnalysisData \leftarrow AnalysisData \cup \{Dist, Stories, D_{co-evo}\}$;
20 **end**
21 **return** $AnalysisData$;

The step in line 4 starts an iteration over the views set $\mathcal{V}$. Here is where we collect the analysis data that are returned by the algorithm. For each of the aggregated artifacts contained in a view V, we retrieve the information necessary to compute the *degree of co-evolution* between pairs of files and their *file distance*. First, we construct the artifact evolution of all the artifacts present in $ae \in V$. Note that an aggregated event $ae \in V$ is a record obtained from a view on the project which is composed, among other attributes (e.g., file, time, amount of

change), by the comment associated to the specific change. Comments describe multiple changes executed on the file, i.e. they describe a *story* of the artifact. Stories associated to each file are collected and the corresponding labels are chronologically ordered. These file stories are then input to the StoryMining technique [9]. Story Mining was designed to receive as input a story freely written by the participants, describing their work in a particular business process. As an output, the *actors* and the process *activities* executed by them are extracted. Our technique is concerned with the stories of the files. Therefore, they are the actors of the story mining, and the resulting business process consists of the steps describing their evolution process. We collect the resulting processes in the step in line 8. The step in line 10 is concerned with the construction of a time series from the set of artifact evolutions A_{evo} computed in line 7. Specifically, this step gathers the values of the changes of each of the artifact f in A_{evo} and records them in $TimeSeries(f)$.

After all the aggregated events *ae* have been explored, the algorithm moves on to computing the metrics (lines 13–18). In this loop, the algorithm iterates through all the pairs of files. For each pair, the *degree of co-evolution* and *artifact-distance* metrics are computed according the Definitions 4 and 5, respectively. These two measures are collected only if their values are above the user defined thresholds γ and δ. After the loop is over, the two measurements and the stories mined with the StoryMiner are stored in $AnalysisData$.

Finally, after iterating over all the user defined views, the algorithm returns the $AnalysisData$ collection which can now be further inspected and analyzed in more detail, as we show next with an example.

3.3 Example

Let us consider the following example of a software development process. It contains 10 files arranged hierarchically as depicted by the file tree in Fig. 2. At the first level of the file tree there is the README.md file which describes the project. The software product in our case is called *running example* and is contained under the f_3 directory. The product consists of an example for software developers who want to organize their projects according to a predefined structure. The project has 21 commits over 10 days.

An excerpt of the VCS log for this project was illustrated in Table 1 above. The project managers are interested in understanding the work process done by project participants in each of the files and whether there is some hidden work dependency. We show how our technique meets the requirements by applying each step to this project and discussing the outcomes.

Let us suppose we have preprocessed our data and have the events set E already stored in a database. Then $\mathcal{V}$ is obtained by querying the data and aggregating them by day. Then, the *parent* relation is $Parent = \{(f_1, f_2), (f_1, f_3), (f_3, f_4), (f_3, f_6), (f_3, f_{12}),\ (f_4, f_5),\ (f_6, f_7), (f_6, f_8), (f_6, f_9),\ (f_9, f_{10}), (f_{10}, f_{11})\}$. Next, we compute the artifact evolution of for each artifact. For example, the artifact evolution of file REAMDE.md (f_2) limited on the information from

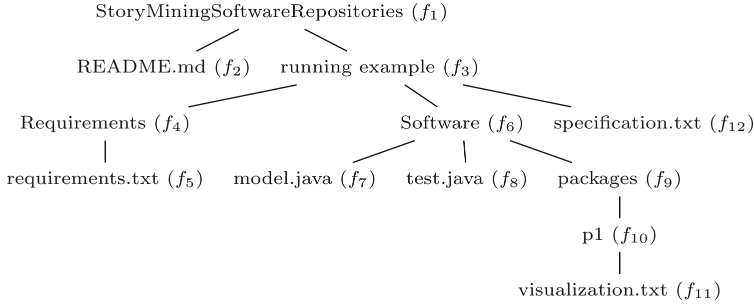


Fig. 2. File tree describing the file structure in our scenario of use.

Fig. 3. Example of business process showing the artifact evolution

Table 1 is A_{evo} = { (2017-01-31, 1, *Create readme file*), (2017-02-01, 3, *Add a license*), (2017-02-02, 1, *Updated the requirements*)}. The resulting process from the story mining algorithm is shown in Fig. 3.

Next, we calculate the metrics. The dependencies are computed in the steps enclosed in lines 13–18 of Algorithm 1. E.g., the artifacts `README.md` (f_2) and `test.java` (f_7) appear in the *TimeSeries* collection as the vectors $\boldsymbol{X_{f_2}} = (1, 3, 1, 0)$ and $\boldsymbol{X_{f_7}} = (0, 0, 0, 2)$. We use the Pearson correlation between the to vectors $\sigma(\boldsymbol{X_{f_2}}, \boldsymbol{X_{f_7}} = -0.66)$ and take its absolute value as degree of co-evolution $\chi = |\sigma|$. Therefore, the *degree of co-evolution* between the considered artifacts is $\chi = 0.66$. The *file distance* is the length of the route from f_2 to f_7, i.e. $d(f_2, f_7) = \{(f_2, f_1), (f_1, f_3), (f_3, f_6), (f_6, f_7)\}$. Therefore, the file distance between `README.md` and `test.java` is $d(f_2, f_7) = 4$.

4 Evaluation

In this section, we show the applicability of our technique to project-oriented business processes and its effectiveness in uncovering work dependencies. With respect to the requirements formulated in Sect. 2, we evaluate against requirements R2 and R3 in Sect. 4.1 and against requirement R1 in Sect. 4.2.

We implemented our techniques as a prototype[2] and used it on 10 real world software projects with different sizes. The input of our program is a VCS log and the output is a set of analysis data with information about the evolution of the artifacts and their dependencies. We report the results in Table 3. The results are listed in increasing order of project size. The parameters χ and d are the metrics

[2] The source code is available at https://github.com/s41m1r/MiningVCS.

Table 3. Evaluation of real world projects. Respectively the thresholds are: χ^L if $\chi < 0.3$, χ^H if $\chi > 0.7$ low and high degree of co-evolution; d^L if $d \leq 2$, d^H if $d > 2$ respectively low and high distance.

Project	Commits	Files	χ^H	χ^L	(d^L, χ^L)	(d^L, χ^H)	$(d^H, \chi^L,)$	(d^H, χ^H)	$\overline{\lvert p_f \rvert}$	$max(\lvert p_f \rvert)$	$\lvert A_{evo} \rvert$	$\overline{d}$	$max(d)$
mwaligner	21	9	37	7	6	30	1	7	1.11	2	2.40	0.94	3
Biglist	202	15	22	90	31	18	59	4	1.47	3	2.76	1.20	5
camundaRD	11	15	74	26	0	25	26	49	2.18	4	2.05	2.03	7
graphql	256	30	89	357	121	89	236	0	1.40	2	3.18	1.11	4
jgitcookbook	135	89	773	2866	505	289	2361	484	6.93	8	1.33	2.68	14
mysqlpython	749	168	2288	11571	742	591	10829	1697	2.59	7	1.65	2.52	11
gantt	23	228	7006	14343	386	3480	13957	3526	3.30	4	1.71	2.16	7
facebookjavasdk	38	293	16478	26092	2017	16311	24075	167	6.21	8	4.78	5.58	13
caret	864	432	15366	60874	9538	14785	51336	581	3.01	4	3.15	1.60	7
operationcode	1114	1053	84024	444605	2291	5537	442314	78487	4.27	8	2.01	4.85	15

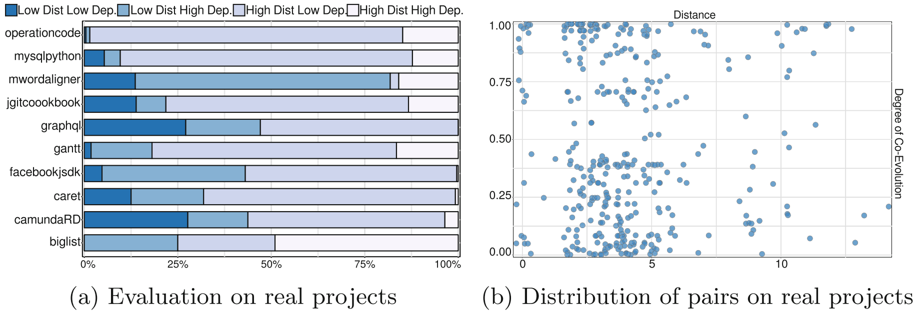
of *degree of co-evolution* and *distance*, respectively. In this example, $\chi > 0.7$ signifies that the co-evolution is high (χ^H) and $\chi < 0.3$ that the co-evolution is low (χ^L). As previously mentioned, this is a user customizable threshold that can be set by the domain expert. Likewise, the distance is considered low (d^L) when $d <= 2$ and high (d^H) when $d > 2$. The parameter $\overline{|p_f|}$ and $max(|p_f|)$ are respectively the average and the maximum lengths of the path to the root (i.e. average tree depth of the files). The column $|A_{evo}|$ shows the average number of activities in the process representing the artifact evolution. Lastly, the columns $\overline{d}$ and $max(d)$ report the average and maximum file distance, respectively. Next, we use these data for a quantitative evaluation of the projects.

4.1 Quantitative Evaluation

Here we address requirements R2 and R3. First, we compute project profiles. These profiles show the distribution of work-related dependencies in a project. Second, we evaluate whether the work on files can be predicted.

Before assessing project profiles, we make the following consideration. Our metrics define four classes: *(i)* low distance low co-evolution; *(ii)* high distance low co-evolution; *(iii)* low distance high co-evolution; *(vi)* high distance high co-evolution. Figure 4b helps clarifying these four classes. In fact, except for values of distance equal to 0, it is possible to see how the density of file pairs is higher when the distance is low. This is a normal situation in project where highly related files are stored closely to each other in the file system. Conversely, the dots on the top right of the plot mark files which are very distant to each other but still highly correlated. These can be, for instance, logical dependencies that can happen because of bad modularization of the project.

Hidden work dependencies belong to the last mentioned case, i.e. files are distant in the file tree but they have similar time series. According to this consideration we computed the project profiles in Fig. 4a. We observe three types of



(a) Evaluation on real projects (b) Distribution of pairs on real projects

Fig. 4. Characterization of the evaluated software projects

processes. First, several projects have hardly any hidden work dependencies. Second, several have a moderate degree between 10% and 20%. Third, the project *Biglist* has a high share of hidden dependencies. This hints at the possibility for better organizing the project according to good modularization best practices. That means, the project can be restructured in a way to reduce the unwanted side-effect the work on one file produces on other files.

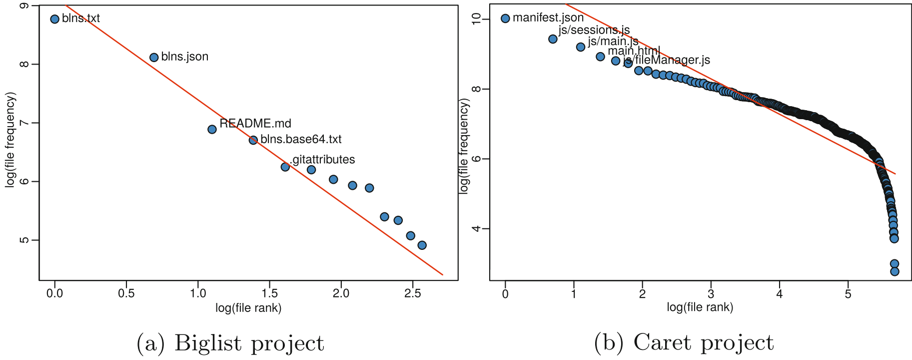
Next, we evaluate whether the work on files can be predicted. Zipf's law is typically used in corpus analysis and states that the *frequency* of usage of any word is inversely proportional to its *rank* in the frequency table. This approach has already been applied to software projects for understanding whether the assignment of developers to tasks in a software project could be predicted [7]. Here, we focus on understanding whether the Zipf's law holds true also for work dependencies within a project.

To this end, we selected one big and one small project from Table 3, namely *Biglist* and *Caret*. Biglist is a small project on a list of strings which are known to cause issues when used as user-input data. Caret is a big project consisting in the development of a sublime text editor for Chrome OS. We collected how frequently were the artifacts worked on to generate a ranking. Figure 5 depicts the corresponding charts and the fitted Zipf distribution. We notice that both projects present a similar distribution of values. This holds also for the other projects analyzed. In particular, Zipf's law is valid for the most frequently changed files. Afterwards, the distribution drops because of files not being worked anymore but still being part of the project.

4.2 Qualitative Evaluation

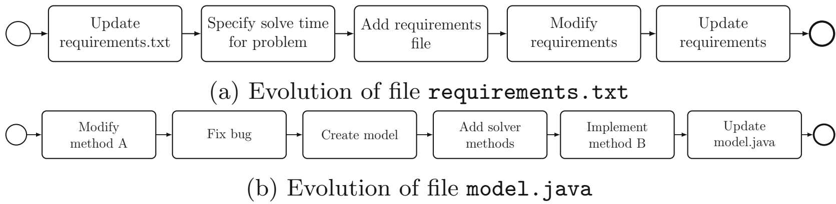
In this section, we address requirement R1 by showing insights on the work history of files that are related. To this end we focused on the project *smsr*, which has 21 commits over a time span of ten days.

Let us consider an example where our technique proves helpful. Our technique finds 6 highly related pairs, as shown in Table 3. We excluded files that have



(a) Biglist project (b) Caret project

Fig. 5. Zipf distribution of the worked files



(a) Evolution of file `requirements.txt`

(b) Evolution of file `model.java`

Fig. 6. Processes of two work-dependent files

a functional dependencies, e.g. interface-class relations, where a change in the interface trivially brings change in the class. Thus, we were able to select the files `smsr/running example/Requirements/requirements.txt` and `smsr/running example/Software/model.java`, having $\chi = 0.7$ and $d = 4$. Moreover, by observing the content we verified that they do not have functional dependencies. Therefore, these two files are work dependent. Figure 6 shows the extracted processes after mining their stories. Interestingly, the two processes do not share any activity because they were never changed together in the same commit.

Our technique can fail under some circumstances. Consider the example above. We know that the files `requirements.txt` and `model.java` are work dependent. Let us now assume that the assumption of *regular commits* in the VCS does not hold. Nevertheless, we know that there is the following work pattern: *at irregular times, one change in the requirements produces 2 changes of work that must be implemented in model in the next day*. In a short time window of 4 days, the time series would be $X_{req} = (1, 0, 1, 0)$, $X_{model} = (0, 2, 0, 2)$ and their correlation is $\sigma(f_{req}, f_{model}) = -1$. Hence, they would score a high degree of co-evolution $\chi = 1$. However, if we double the time window and observe only another pattern the correlation would change. We get $X_{req} = (1, 0, 1, 0, 0, 1, 0, 0)$, $X_{model} = (0, 2, 0, 2, 0, 0, 2, 0)$ which score a $\sigma(f_{req}, f_{model}) = -0.66$, $\chi = 0.66$ and therefore not a high value of correlation.

These results show that our technique helps uncovering work dependencies that are not captured by existing approaches in literature which leverage on social network analysis [23,25]. On the other hand, our technique is currently not yet able to retrieve dependencies with delay. We plan to address this challenge by using moving-average time series models.

5 Conclusion

In this paper, we addressed the problem of uncovering hidden work dependencies from VCS logs. The main goal was to provide project managers with knowledge about the artifacts co-evolution in the project. Three perspectives of analysis were considered, evolution of the artifacts over time, dependencies among them and structural organization of the project.

Our approach works under the assumptions that repositories reflect the hierarchical structure of the project, project participants commit their work regularly during active working times and they provide informative comments for the changes done. The approach was implemented as a prototype. A scenario of use was provided showing how the approach can be applied and providing some discussions. We also evaluated our approach in real-world data from open source projects showing the potential of the approach.

In future work, we will improve our evaluation varying for instance the time window, the dependency threshold and consider a study case with project managers. We plan to investigate other types of dependencies between artifacts. Specifically, we are interested in a semantic analysis of the work performed in both artifacts, considering for instance some similarity measures. We also aim to improve the visualization to consider other knowledge extracted, for instance the type of change performed in the aggregate events could be shown associated to the activities in the artifact process.

References

1. Abate, P., Cosmo, R.D., Boender, J., Zacchiroli, S.: Strong dependencies between software components. In: 3rd International Symposium on Empirical Software Engineering and Measurement ESEM, pp. 89–99 (2009)
2. Bala, S., Cabanillas, C., Mendling, J., Rogge-Solti, A., Polleres, A.: Mining project-oriented business processes. In: Motahari-Nezhad, H.R., Recker, J., Weidlich, M. (eds.) BPM 2015. LNCS, vol. 9253, pp. 425–440. Springer, Cham (2015). doi:10.1007/978-3-319-23063-4_28
3. Bani-Salameh, H., Ahmad, A., Aljammal, A.: Software evolution visualization techniques and methods - a systematic review. In: 2016 7th International Conference on Computer Science and Information Technology (CSIT), pp. 1–6 (2016)
4. Beheshti, S.-M.-R., Benatallah, B., Motahari-Nezhad, H.R.: Enabling the analysis of cross-cutting aspects in ad-hoc processes. In: Salinesi, C., Norrie, M.C., Pastor, Ó. (eds.) CAiSE 2013. LNCS, vol. 7908, pp. 51–67. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38709-8_4

5. Beheshti, S.-M.-R., Benatallah, B., Sakr, S., Grigori, D., Motahari-Nezhad, H.R., Barukh, M.C., Gater, A., Ryu, S.H.: Process Analytics - Concepts and Techniques for Querying and Analyzing Process Data. Springer, Cham (2016)
6. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 88–94. Springer, Heidelberg (2000). doi:10.1007/10719839_9
7. Canfora, G., Cerulo, L.: Supporting change request assignment in open source development. In: Proceedings 2006 ACM Symposium on Applied Computing - SAC 2006, p. 1767, April 2016
8. D'Ambros, M., Lanza, M., Lungu, M.: Visualizing co-change information with the evolution radar. IEEE Trans. Softw. Eng. **35**(5), 720–735 (2009)
9. Gonçalves, J., Santoro, F.M., Baião, F.A.: Let me tell you a story - on how to build process models. J. Univers. Comput. Sci. **17**(2), 276–295 (2011)
10. Greene, G.J., Fischer, B.: Interactive tag cloud visualization of software version control repositories. In: 3rd Working Conference on Software Visualization, pp. 56–65 (2015)
11. Gubichev, A., Bedathur, S., Seufert, S., Weikum, G.: Fast and accurate estimation of shortest paths in large graphs. In: 19th ACM International Conference on Information and Knowledge Management, p. 499 (2010)
12. Kindler, E., Rubin, V., Schäfer, W.: Activity mining for discovering software process models. Softw. Eng. **79**, 175–180 (2006)
13. Lindberg, A., Berente, N., Gaskin, J., Lyytinen, K.: Coordinating interdependencies in online communities: a study of an open source software project. Inf. Syst. Res. **27**(4), 751–772 (2016)
14. Mittal, M., Sureka, A.: Process mining software repositories from student projects in an undergraduate software engineering course. In: ISCE Companion, pp. 344–353 (2014)
15. Oliva, G.A., Santana, F.W., Gerosa, M.A., de Souza, C.R.: Towards a classification of logical dependencies origins. In: Proceedings 12th International Workshop 7th Annual ERCIM Workshop on Principles of Software Evolution - IWPSE-EVOL 2011, p. 31 (2011)
16. Peffers, K.E.N., Tuunanen, T., Rothenberger, M., Chatterjee, S.: A design science research methodology for information systems research. J. Manag. Inf. Syst. **24**, 45–77 (2007)
17. Poncin, W., Serebrenik, A., Brand, M.V.D.: Process mining software repositories. In: 2011 15th European Conference Software Maintenance Reengineering, pp. 5–14 (2011)
18. Ripley, R.M., Sarma, A., Van Der Hoek, A.: A visualization for software project awareness and evolution. Visualization 2007 - Proceedings 4th IEEE International Workshop on Visualizing Software for Understanding Analysis, pp. 137–144 (2007)
19. Robles, G., González-Barahona, J.M., Cervigón, C., Capiluppi, A., Izquierdo-Cortázar, D.: Estimating development effort in free/open source software projects by mining software repositories: a case study of openstack. In: 11th Working Conference on Mining Software Repositories, pp. 222–231 (2014)
20. Ruohonen, J., Hyrynsalmi, S., Leppänen, V.: Time series trends in software evolution. J. Soft. Evol. Process **27**(12), 990–1015 (2015)
21. Voinea, L., Telea, A.: CVSgrab: mining the history of large software projects. In: Eurographics/EuroVisualization, pp. 187–194 (2006)
22. Voinea, L., Telea, A.: Visual data mining and analysis of software repositories. Comput. Graph. **31**, 410–428 (2007)

23. Weicheng, Y., Beijun, S., Ben, X.: Mining GitHub: why commit stops - exploring the relationship between developer's commit pattern and file version evolution. In: 20th Asia-Pacific Software Engineering Conference, pp. 165–169 (2013)
24. Zaidman, A., Van Rompaey, B., Demeyer, S., Van Deursen, A.: Mining software repositories to study co-evolution of production & test code. In: 1st International Conference on Software Testing, Verification and Validation, pp. 220–229 (2008)
25. Zimmermann, T., Nagappan, N.: Predicting defects using network analysis on dependency graphs. In: 13th International Conference on Software Engineering, p. 531 (2008)