Mining Project-Oriented Business Processes

Saimir Bala(✉), Cristina Cabanillas, Jan Mendling,
Andreas Rogge-Solti, and Axel Polleres

Vienna University of Economics and Business, Vienna, Austria
{saimir.bala,cristina.cabanillas,jan.mendling,
andreas.rogge-solti,axel.polleres}@wu.ac.at

Abstract. Large engineering processes need to be monitored in detail regarding when what was done in order to prove compliance with rules and regulations. A typical problem of these processes is the lack of control that a central process engine provides, such that it is difficult to track the actual course of work even if data is stored in version control systems (VCS). In this paper, we address this problem by defining a mining technique that helps to generate models that visualize the work history as GANTT charts. To this end, we formally define the notion of a project-oriented business process and a corresponding mining algorithm. Our evaluation based on a prototypical implementation demonstrates the benefits in comparison to existing process mining approaches for this specific class of processes.

Keywords: Process mining · Projects · Project mining · Version control systems

1 Introduction

Business process management plays an important role for improving the performance and compliance of various types of processes. In practice, many processes are executed with clear guidelines and regulatory rules, but without an explicit centralized control imposed by a process engine. In particular, it is often important to exactly know when which work was done. This is, for instance, the case for complex engineering processes in which different parties are involved. We refer to this class of processes as project-oriented business processes.

Such project-oriented business processes are difficult to control due to the lack of a centralized process engine. However, there are various unstructured pieces of information available to analyze and monitor their progress. One type of data that are often available these processes is event data from version control systems (VCS). While process mining techniques provide a useful perspective on how such event data can be analyzed, they do not produce output that is readily organized according to the project orientation of these processes.

This work has been funded by the Austrian Research Promotion Agency (FFG) under grant 845638 (SHAPE) and the European Union's Seventh Framework Programme under grant 612052 (SERAMIS).

H.R. Motahari-Nezhad et al. (Eds.): BPM 2015, LNCS 9253, pp. 425–440, 2015.
DOI: 10.1007/978-3-319-23063-4_28

In this paper, we define formal concepts for capturing project-oriented processes. These concepts provide the foundation for us to develop an automatic discovery technique which we refer to as *project mining*. The output of our project mining algorithm is organized according to the specific structure typically encountered in project-oriented business processes. With this work, we extend the field of process mining towards the coverage of this specific type of business process.

The paper is structured as follows. Section 2 describes the research problem and summarizes insights from prior research upon which our project mining approach is built. Section 3 defines the preliminaries of our work and presents an algorithm to mine project-oriented business processes. Section 4 describes the implementation of this algorithm and discusses the results from its application to VCS logs from a real-world engineering project. Section 5 highlights the implications of this work before Section 6 concludes.

2 Background

Here, we describe the addressed problem and related work.

2.1 Problem Description

The class of processes that we discuss in this paper are long-term engineering projects. These processes have specific requirements for monitoring. First, they are executed only once according to the specific needs of a particular project, and only partially according to recurring process descriptions. Second, they involve various actors that typically document their work in a semi-structured way using text and tables. Third, work in the project is usually subject to constraints regarding the start and end and the temporal order. Fourth, there is typically no process engine controlling the execution. Fifth, even though these limitations in terms of traceability exist, there are usually strong requirements in terms of tracking when which work was conducted.

In line with these observations, a *project-oriented business process* can be defined as an ad-hoc plan that specifies the tasks to be performed within a limited period of time and with a limited set of resources for achieving a specific goal. Unlike repetitive business processes for which notations such as BPMN [12] or EPC [1] are commonly used, project-oriented business processes may be properly represented with PERT or GANTT models. The concept is illustrated in Fig. 1.

Documentation is required not only explicitly as part of some activities but also to comply with norms and regulations that may require some evidence of the actions being performed in the organization. Documents are usually free of format or contain tables, at best. The unstructuredness of data makes it difficult to monitor processes and check rules on them. A starting point for analysis of project-oriented processes can be data logs that are stored in Software Configuration Management (SCM) systems that help tracking the evolution of data and restore information if needed [19]. However, hundreds of versions of

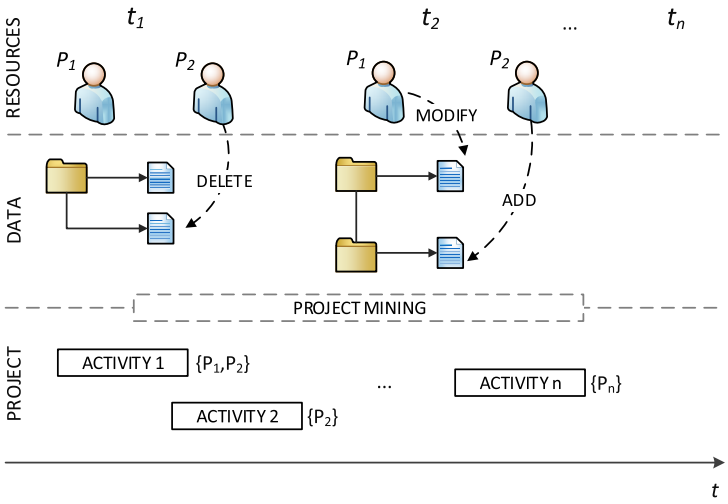

Fig. 1. Problem illustration

thousands of files are common in a single project [20], which makes it impractical to browse this data manually.

Let us see an example inspired by a real scenario of a process to write a project proposal that uses a Version Control System (VCS) to store the data. The project history, and hence, the data produced, starts when people begin to work on the proposal, which involves a description of the project goals and milestones, a division of tasks into work packages, an estimation of cost and resources required, etcetera. This information is spread in the repository over several folders containing different documents, which are later merged into a single file. If the proposal is accepted, the first step is to organize a kickoff meeting and assign specific resources to the work packages. A hierarchical set of folders is then created in the repository in order to store the information generated for each work package. As the project evolves over time, resources contribute by adding, removing or modifying information to the VCS repository. Project evolution is guided by specific norms that impose the execution of predefined steps. For instance, the European norm EN5016 requires a preliminary Reliability, Availability and Maintainability (RAM) analysis to support targets.

Table 1 depicts an excerpt of the log data generated, where the first column (on the left hand side) indicates the commit identifier, the second column indicates the person who committed changes, the third column indicates the commit date, and the fourth column indicates the files affected and the type of action performed among added (A), modified (M) and deleted (D). For the sake of simplicity, the table shows the log data of a specific time period and the actions related to a specific task, namely, *Define example*. That task was assigned to resource X and was supervised by resource Y and, later on, also by resource Z.

Table 1. Excerpt from VCS log data for the referenced time period

CID	Resource	Date	List of changes
1	Y	2014-11-12 11:57:46	A /example A /example/SHAPE/ToyStationExample.docx
...	...	...	...
3	X	2014-11-14 16:34:07	M /example/ToyStation.bpmn M /example/ToyStation.png
4	W	2014-12-15 13:49:11	D /example/Download
5	W	2015-01-08 16:06:41	A /example/Download2
6	X	2015-01-13 11:47:09	M /example/ToyStation_0Loop.bpmn M /example/ToyStation_nLoop.bpmn
7	Z	2015-01-16 16:50:29	A /example/ToyStation_0Loop.pdf A /example/ToyStation-feedbackZ.pdf

Existing frameworks, such as Subversion or Git, allow to access their logs in different ways. However, the covered information is limited to (roughly) that depicted in Table 1. Especially for big projects that are frequently updated over a large period of time, these logs are complex to analyze. Therefore, the problem to address is how to analyze and visualize the information produced in project-oriented business processes such that it can be represented in an understandable and manageable way by project experts and enable, a.o., the automation of mechanisms for compliance checking. The following properties of project-oriented process logs must be taken into account to achieve this goal: (i) VCS repositories consist of a hierarchy of folders and files which are logically organized such that work is grouped in a specific way; (ii) process activities are not registered in VCS log entries. Therefore, such information must be inferred by reasoning on the repository structure and/or the content of the log entries; (iii) the granularity of the events is unknown a priori and it needs to be defined before analyzing the data.

2.2 Related Work

The problem described has been addressed in the literature from different perspectives. The first category of related work tackles the problem by transforming it into a process mining problem. Consequently, approaches have been developed to preprocess VCS data such that process mining techniques can be applied, and hence, a business process can be derived from the log data. In this group, Kindler et al. [9,10] developed an algorithm for extracting software processes that are mapped to Petri Nets. Activities, which are not explicit in the logs, are discovered from their input and output artifacts. However, strong assumptions are made on the filenames as well as on the software process lifecycle. Rubin et al. in [15] addressed the problem of engineering processes that are not well documented and are usually unstructured. They provided a bridge from Kindler et al.'s approach to ProM [5] in order to mine different process perspectives,

such as performance social network analyses. Rubin et al. [16] applied process mining to the touristic industry and obtained user processes from web client logs pursuing the goal of improving the software system by analyzing the underlying process. Poncin et al. [14] developed the FRASR framework for preprocessing software repositories to transform the VCS data to logs that conform to the process mining event log meta model [4] as utilized in ProM [5]. However, these approaches disregard the single-instance nature of project-oriented business processes and treat them as procedures that can be repeated over time.

The second category of related work focuses on the visualization of VCS data for different purposes. Several approaches study the interaction among developers over time from a visualization point of view. For instance, Ogawa and Ma [11] drew storyline pathways to show the story of each developer's contribution. Other approaches analyze and visualize VCS data at file level in order to discover file version evolution. Voinea and Telea [20] introduced an interactive navigation method to surf file version evolution as well as two methods to cluster versions of the same file in an abstraction layer. Wu et al. [22] also visualized the evolutions of entire projects at file level, emphasizing the evolution moments. Finally, several approaches study change prediction with the aim of discovering prediction patterns that can help in the process of software development [23,24]. The approaches mentioned in this category as well as others that apply similar techniques [3,6,8] focus on studying software evolution from different standpoints. However, the goal pursued differs in all cases from our goal in that they are not interested in discovering projects tasks out of the log data, and hence, they lack an explicit notion of work structure that we need to consider for our purpose.

Our approach combines ideas from both areas, as we aim at identifying tasks like in the approaches that rely on process mining, but we must cluster the data in an appropriate way, for which techniques developed in the approaches that pursue visualization may be adapted or extended.

3 Mining VCS Event Data

Here, we first formalize the notions encountered in the project mining setting. Then we develop an approach to acquire a hierarchical overview on the project from a repository perspective.

3.1 Preliminaries

Version control systems (VCSs) are used in projects to ensure reliable collaboration. We build our approach on VCS. Typically, the workflow in VCS is that people work on files (e.g., text, source code, spread sheets) and commit them to the central repository. Project participants comment on their commits so that other participants can better understand the nature of the changes to the files.

Let F be the universe of files. 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 $(f_p, f_c) \in Parent$. The transitive closure on the parent files is given by the function $ancestor : F \to 2^F$ that returns the set of files along the path to the root.

When project members did a certain amount of work and want to save their current progress, they commit the changes to the VCS. We define changes on files 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 four-tuple (f, o, ts, k), where*

- *$f \in F$ is the affected file of the event.*
- *$o \in O = \{added, modified, deleted\}$ is the change operation on the file with obvious meaning.*
- *$ts \in TS = \mathbb{N}_0$ represents a unix time stamp marking the time of the event occurrence.*
- *$k \in \Sigma^*$ is a comment in natural language text.*

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

Project participants can commit a number of changes to different files at one step. Therefore, we define the notion of commits as follows.

Definition 2 (Commit). *A commit $\mathcal{C}$ is a set of events sharing the same time stamp and comment, i.e., $\forall e, e' \in \mathcal{C} : ts(e) = ts(e') \wedge k(e) = k(e')$. Additionally, each event in a commit affects different files, i.e., $\forall e, e' \in \mathcal{C} : e \neq e' \to f(e) \neq f(e')$.*

Usually, it is in the hands of project participants, when they decide to commit changes to the VCS. In the extreme case, there could be only a single commit made in a project that adds all files to the repository. Note that this extreme practice would render the use of a VCS obsolete. On the contrary, it is common practice to regularly perform commits in order to securely store work progress and to reduce the chance of conflicts [7,13]. Conflicts occur, when another participant committed changes to a file that is being committed and can cause extra work. Based on these insights, we make the assumption that commits are regularly made during work.

Projects are decomposed into work packages. We assume a hierarchical work package structure of a project, such that a work package can have sub work packages. Further, the amount of work in a single work package need not be done in one single time span, but it can be split into several activities. Activities have a start and end time, and subsequent activities can have idle periods in between. Thus, we define projects as follows.

Definition 3 (Project). *A project P is a tuple $(W, S, A, \alpha, \omega, \beta)$, where*

- *W is the set of work packages in the project.*
- *$S \subseteq W \times W$ is the relation that hierarchically decomposes work packages into a tree structure.*
- *A is the set of activities that are conducted in the work packages.*

- $\alpha : A \rightarrow TS$ *is the function that assigns a start time to activities. Activities are ordered by their start times.*
- $\omega : A \rightarrow TS$ *is the function that assigns an end time to activities.*
- $\beta : A \rightarrow W$ *is the mapping function that maps activities to their corresponding work packages.*

Note that this definition reflects an activity centric view on projects. The definition deliberately omits further dimensions, e.g., costs, resources, risks. The idea is not to capture projects in every detail, but to focus on the work packages of a project to obtain an overview of the work that is being done. We are interested in when work has been started in a work package, and when work packages have been done. This information can be derived from the activities associated to the workpackages. An obvious assumption is that the work package starts with its first activity, and ends when its last activity is completed.

Based on these notions, we can define the task of *project discovery* as reconstructing the project P from a set of low level event data E. In the following, we present an approach to this problem.

3.2 Project Discovery Technique

For project discovery from the VCS commit history, we need to identify activities that are performed, associate the activities to work packages and recreate the work package structure of the project. Our aim is to create a hierarchical model that provides an overview of the project work. Therefore, we have to identify the start and end times of activities and of work packages before we can visualize the project work. The input to the technique is the log that is stored in the VCS. The challenge is that the raw log only records commits on the file system level and information on activity level is missing. However, we can deduce activity information from events based on the following assumptions.

A1: Meaningful file tree structure. The file tree structure in a project represents its work package structure. That is, the knowledge workers organize their work in a file hierarchy that reflects the project structure.

A2: Local changes. Activities in a work package affect only files of the work package folder, or in the corresponding sub-tree in the file tree structure.

A3: Frequent commits. Commits to the VCS are regularly performed, when conducting work in an activity.

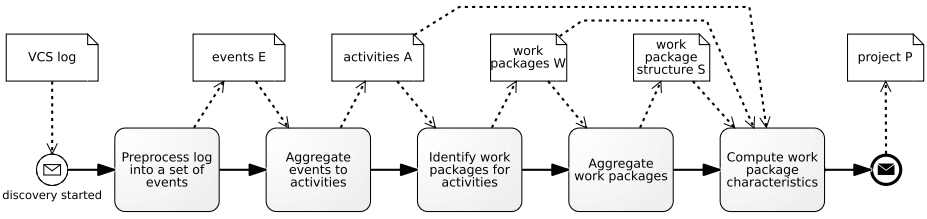

Fig. 2. Project discovery technique overview as BPMN process model.

Note that assumption A1 can be seen as a strong assumption on the file tree structure. Nevertheless, we argue that even if A1 is not entirely met, the aggregation of work information on the file tree hierarchy provides a valuable view on the project. Figure 2 shows the different steps of the technique. We describe each of them in detail.

Step 1: Preprocessing. The first step is to transform raw logs of version control systems (which might be grouped by commits) into a list of events as specified in Definition 1. This step is easily done by replicating the information on commit level to be contained in the events. The output is a set of events E.

Step 2: Aggregating events to activities. Given the set of events E that we gathered from a version control system, the next step is to identify the activities to which the events belong. Note that we do not know the activities of the project in advance, but need to infer them based on the events. Each event affects a single file in the file hierarchy.

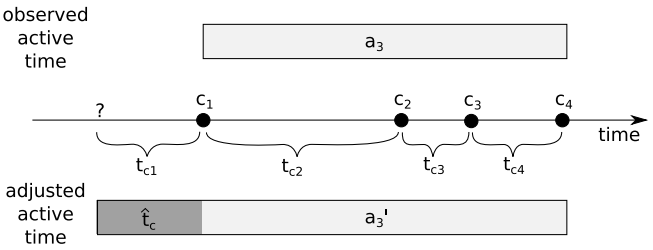

Fig. 3. Adjustment of activity start time α.

Based on assumption A2, we are interested in activities conducted in a work package, that is, we filter for the events that are contained in the given file or its children. For every file f of interest, we select the set of events affecting the file or its children as $E_f = \{e \in E \mid f = f(e) \vee f \in ancestor(f(e))\}$. The task is then to find the activities which emitted the set of events E_f. We rely on assumption A3, which states that during an activity, we expect multiple commits. Assumption A3 allows us to conclude that if we do not observe commits for a longer period of time, there is no activity being performed in the work package.

To this end, we adopt the abstraction technique by Baier et al. [2] and allow the domain expert to formulate rules for aggregating events to activities based on boundary conditions. Assuming that people frequently commit their progress (A3), we can specify a boundary condition based on the temporal distance to previous events. For example, we can specify that a time period of seven days without a commit is a boundary condition. As the result, we obtain the mapping from events to these activities, which we call $\gamma_f : E_f \rightarrow A_f$ in the remainder of the paper. The set of discovered activities identified for the work package based

on given boundary conditions is then $A_f = \{a \mid e \in E_f, \gamma_f(e) = a\}$. We also define the inverse mapping, that is, the mapping from an activity to its events as $\gamma_f^{-1} : A_f \to 2^{E_f}$.

With the events mapped to activities, we need to find the temporal boundaries of the target activities. That is, we define the functions α and ω for each activity. The challenge here is that we do not know when an activity actually started, because the start of the activity is not recorded in the VCS. We can only observe the time of the first commit in that activity, but commits usually mark progress of an already running activity.

To address the challenge of missing start times, we impute the missing start time by prepending the expected active time $\hat{t}_c$ before a commit, as illustrated by Figure 3. This notion assumes that project participants commit their work progress after a certain amount of time. However, we cannot compute $\hat{t}_c$ by looking at the average commit rate in a work package, because this average is based on busy periods and idle periods. We need to factor out the idle periods in the computation of this measure. We know the end time of the activities, as the last commit marks the completion of work. Therefore, each activity a based on given boundary conditions has the associated end time $\omega(a) = \max(\{ts(e) \mid e \in \gamma_f^{-1}(a)\})$. Further, we write the first event's timestamp of an activity as the function $\alpha'(a) = \min(\{ts(e) \mid e \in \gamma_f^{-1}(a)\})$. Then, we define $c : A_f \to \mathbb{N}^+$ as the number of commits in one activity, formally $c(a) = |\{\mathcal{C} \mid e \in \mathcal{C} \wedge \gamma_f(e) = a\}|$.

With this information the expected active time between commits $\hat{t}_c$ is given as follows.

$$\hat{t}_c = \frac{\sum_{a \in A_f} (\omega(a) - \alpha'(a))}{\sum_{a \in A_f} (c(a) - 1)} \tag{1}$$

We assume that there is at least one activity spanning over at least two commits, i.e., $\exists a \in A_f \mid c(a) > 1$. Translated to our boundary condition, this assumption is that there is at least one week in each work package, in which there were at least two commits made. Otherwise, we set $\hat{t}_c$ to 0 for the current file f due to lack of information.

Given the expected active time between commits $\hat{t}_c$, we can finally adjust the start time of each activity. Therefore, we set the associated start time for each activity as $\alpha(a) = \alpha'(a) - \hat{t}_c$. That is, we subtract the expected active time from the first commit's timestamp.

We apply Step 2 to all files f in the file tree to get A_f. For the remainder of this paper, we define the function $\psi : A \to F$ that contains the mapping information of the discovered activities to their originating files. Finally, we set the activities A in the project to be the union of the activity sets per file $\bigcup_{f \in F} A_f$.

Steps 3 and 4: Mapping activities to work packages and aggregating. Once activities have been identified, we want to climb to the next abstraction layer: the work packages. Assumption A1 allows us to specify a one-to-one mapping $\kappa : F \to W$ between files in the file tree structure and work packages. More

precisely, we construct the set of work packages W isomorphic to the set of files F, such that the *Parent* relation is preserved in the work package structure S relationship.

The mapping β of activities to work packages is simply $\beta(a) = \kappa(\psi(a))$. That is, the corresponding work package of the activity that was discovered for a file. In this way, we provide an activity based view on work packages, and we can aggregate on each level in the file system to see active periods of the corresponding hierarchy level.

Step 5: Computing work package characteristics. In this final step, we compute measures of interest for the discovered work packages. First, we obtain the temporal boundaries of a work package by the functions α and ω of the associated activities.

Let $\beta^{-1} : W \to 2^A$ be the inverse of the mapping function β of the project. The start and end time of a work package (α_W and ω_W) are functions from work packages to timestamps. The start time is defined as $\alpha_W(w) = \min(\{\alpha(a) \mid a \in \beta^{-1}(w)\})$, and the end time function of work packages ω_W is analogously defined using the maximum of the end times $\omega(a)$ of the activities. We call the duration of a work package τ that is the difference between ω_W and α_W.

Moreover, we are interested in the ratio of active working periods (i.e., the time spans of activities) to the total work package duration. This quantity helps to estimate the average work intensity in a work package.

Definition 4 (Coverage). *The coverage χ of work packages by activities is a function $\chi : W \to [0, 1]$ and is defined as follows.*

$$\chi(w) = \frac{\sum_{a \in \beta^{-1}(w)} (\omega(a) - \alpha(a))}{\tau(w)} \tag{2}$$

With this final step, we lifted the information hidden in low level events to a high-level Gantt chart perspective, with which project managers are familiar. In the following, we compare our technique to existing process mining approaches.

4 Evaluation

In this section we evaluate our solution to the project mining problem, and show results for the example presented in Section 2.

4.1 Experimental Setup

We evaluate our technique by a visual perspective and by comparison to possible different approaches. To this end we implemented our technique as a prototype. We used JAVA as a programming language to code the logic of our technique. For the visualization part we made use of custom SWT widgets provided by the

Nebula Project[1]. Our program can deal with logs from Subversion (SVN) [13] and Git[17], but it can be extended to other version control systems by providing an implementation of the preprocessing step discussed in Section 3.2. We ran the software in an Intel®Core ™ i5-4570 CPU @ 3.20 GHz x 4 machine with 15.6 GiB of RAM and Linux kernel 3.13.0-46-generic 64-bit version.

4.2 Input Data Description

We tested our prototype with real-world log data taken from the SHAPE project. Logs were exported from the SVN and Git repositories of different projects. They come from the railway domain and describe engineering processes. Documentation stored in the repositories consists of manually produced text files, diagrams, and files coming from proprietary tools that are typically used in the domain.

We will display results for the SVN log that describes the process oriented project for SHAPE. Data span over one year, going from January 2014 to January 2015. This time window covers the phases of project definition and planning, and a part of the project execution. In the first phase, feasibility of the project was studied and budget, schedule and resources were determined. Proposal submission marked the end of this phase. The second phase started with a kickoff meeting in October 2014 and is still ongoing.

The total number of participants who actively contributed to the work packages stored in the SVN repository was 8 people in the beginning, with new resources joining the project after the kickoff date. The total number of files and directories counts up to 156 objects and 226 overall commit events. The total number of extracted change events after preprocessing (i.e. atomic changes on all the files) was 453.

The last part of the log data contains the task *Define example*, introduced in Section 2.1. For our showcase we assume that this task is contained in a work package named *example*.

4.3 Output Data

To monitor the project execution, we visualize the work progress that was done for each work package. Monitoring is performed by managers who want to have an overview on the project (which work packages are done, when and for how long, and where idleness or congestion occurs). Gantt charts offer a graphical representation for displaying schedules and jobs that were done on the various work packages [21] in a way that can easily be communicated to managers.

Figure 4 is a screenshot of how our tool presents the data. The tree structure on the left represents the *Parent* relation in the file tree. Events belonging to the same commit have the same color. On the top part of the chart we can see the result of merging events to activities with our aggregation method. Here we have merged the events of the example scenario on their highest abstraction level. The chart shows the three main activities and the idle times between them.

[1] https://www.eclipse.org/nebula/

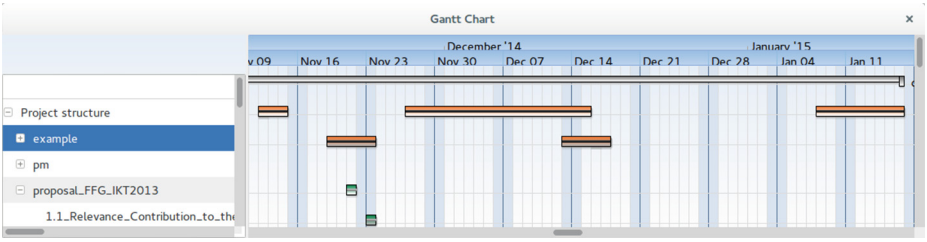

Fig. 4. Data representation from our tool. Atomic events are drawn as dot with a minimal duration and different color per commit.

On the other hand, in correspondence to expanded directories we show only their status before the aggregation. That is, every time a directory is fully expanded we apply a disaggregation into the corresponding activities. In this way, we can also show the finest granularity of work, i.e. the atomic events.

4.4 Project Analysis

Next, we apply our algorithm to the example case from Table 1 and check how it helps to identify work packages. The data is aggregated according to our threshold of seven days. We can observe three groups of events being temporally close to each other according to our threshold. That is, we expect the event data to be grouped into three activities.

The second step of our algorithm takes care of adjusting the starting time of the activities. Furthermore, we vertically order the events and activities in the Gantt chart according to the directory structure to show the mapping from the objects on the Gantt chart to each work package in the tree structure. The last step, computing work package characteristics is done automatically when we collapse a node of a tree.

Figure 5 shows a comparison of the case when we do not implement the activity adjustment to the case which adjusts it. In the upper part, activity

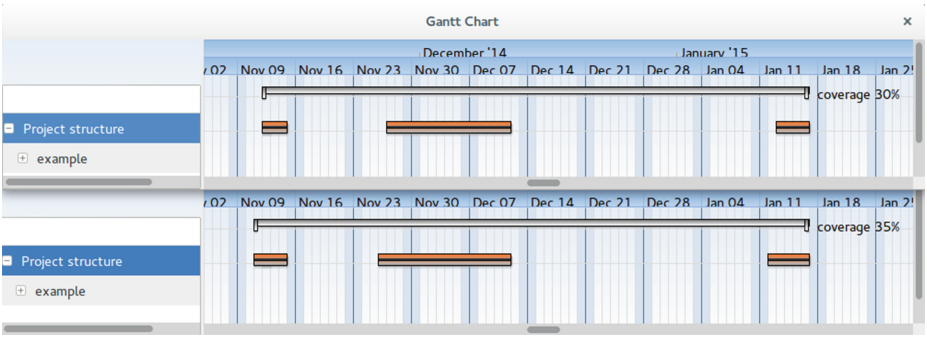

Fig. 5. Before and after the prepending the expected time before commit. Coverage factor increases when we adjust the starting times of the activities.

boundaries are based only on the first commit time that we see in the data. In the lower part, we observe that the start times were adjusted by approximately one day. The tool automatically adjusted the start time of the activities. As a consequence, the coverage factor increases because we expect that there was more work than what we observe by only considering the first commit time.

4.5 Coverage Tests on Available Open Projects

Finally, we apply our approach on different input data from open source projects. We are interested in exploring how the coverage factor varies in different existing projects. Hence, we take the work package w as our controlled variable and set it to the highest level of aggregation. Then, we analyze each project of the data set and observe the dependent variable $\chi(w)$. Another variable of interest is the $\hat{t}_c$ since it gives an idea of the average work speed (commit frequency) during active times.

Table 2. Coverage results for different open source projects

Log File name	**Duration** Days	**Idle periods** Number	**Files** Number	**Commits** Number	$\hat{t}_c$ Hours	$\chi(w)$ %
MiningCVS	24	0	89	63	9	100
Whitehall	1279	6	6539	15566	2	95
Petitions	834	17	1562	914	13	59
Study	624	13	7501	736	11	58
The Guardian	1667	59	12889	621	30	44
Book	414	15	154	592	5	32
Papers	1859	55	1791	649	20	30
Requirements	771	22	505	231	17	21
Yelp	206	6	24	54	20	20
Adobe	1076	13	356	237	24	15

The data we used stems from the following projects. *MiningVCS* is our tool. It consists of daily commits and was developed over 24 days. *Whitehall* is the code name for the Inside Government project, which aims to bring Government departments online in a consistent and user-friendly manner. *Petitions* is a Drupal 7 code base used to build an application on "We The People", the platform to create and sign petitions of the White House. *Study* is an SVN log about Healthcare domain, taken from SHAPE. *The guardian* is the log data from the Git repository of the well-known British national daily newspaper. *Book* is the log data that describes the writing of the book Crypto 101 by Laurens Van Houtven, taken from Git. *Papers* is taken from SHAPE project for building a paper archive. *Requirements* log data is taken from the the Git repository of OpenETCS and belongs to the railway domain. *Yelp* is the main Github page of Yelp were they showcase all their projects. *Adobe* is the Adobe Github Homepage v2.0, which is a central hub for Adobe Open sources projects.

Table 2 shows our experiments on the above-mentioned logs and the corresponding coverage factors. Projects that score a high coverage factor are characterized by continuous work. This can be further seen by looking at their average idle times $\overline{t}_{Idle}$. Let n_c be the number of commits per work package. We compute the average idle time as follows.

$$\overline{t}_{Idle} = \frac{\tau - n_c \cdot \hat{t}_c}{n} \ , \ n > 0 \tag{3}$$

where n is the number of idle times in the work package. If $n = 0$, then we trivially assign $\overline{t}_{Idle} = 0$, because there were no break periods over time.

Applying the formula to the above projects, we can observe how projects with a higher coverage factor have actually low values of $\overline{t}_{Idle}$. For instance, *Whitehall* scores a $\overline{t}_{Idle}$ of 11 days, whereas *Adobe* scores a $\overline{t}_{Idle}$ of 36 days. This supports the usage of the coverage factor χ as an indicator for work package time utilization.

5 Discussion

In this section we compare our method to other alternatives for mining data out of logs and interpret our results.

Well known tools that are used in academia and practice include ProM[5] and Disco[2]. Both tools require input data to be in the XES [18] format. Thus, we convert our data from the *Define example* case into XES. To show events per objects of the project structure, we choose the file path as the *caseId*. To flatten the logs we extract all the file paths and build a mapping from each file to the set of changes done to it.

Figure 6 depicts the results of the Dotted chart plugin of ProM applied to our log data. Also here, we observe different changes of each file of the repository. While the files and their corresponding events are shown, the plugin does not allow to rearrange the data in order to understand the file structure, nor does it allow to perform any kind of aggregation or connection between data, to observe them from a higher level perspective.

Figure 7 shows the results from mining our log data with the Disco tool. Here we can see a plot that displays the events that happen over time. The plot has some peeks in correspondence to active times of the *example* work package. They can be grouped in three clusters: an initial cluster with a few amount work, an intermediate cluster with the most significant part of the work, and a final cluster that again is not very active. In this way, clusters can be associated to activities. As a drawback, when the number of work packages and activities increase, the number of peeks grows and generate identifying clusters of activities by look at active (or idle) times becomes unworkable.

Our approach to mining the work progress of project-oriented business processes complements these techniques with metrics and a corresponding visualization that is informative to managers.

[2] http://fluxicon.com/disco/

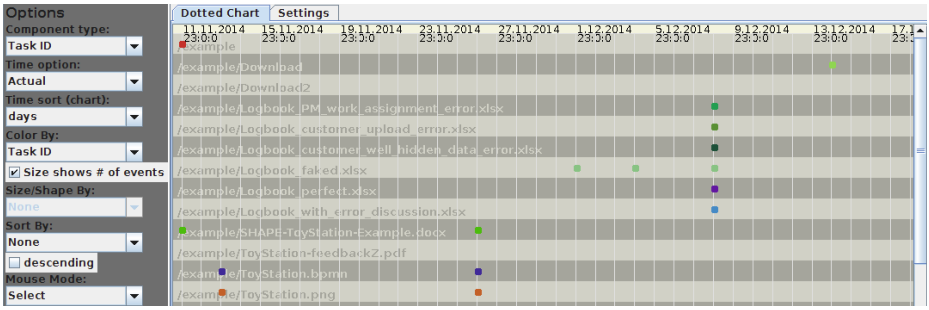

Fig. 6. Dotted chart from ProM

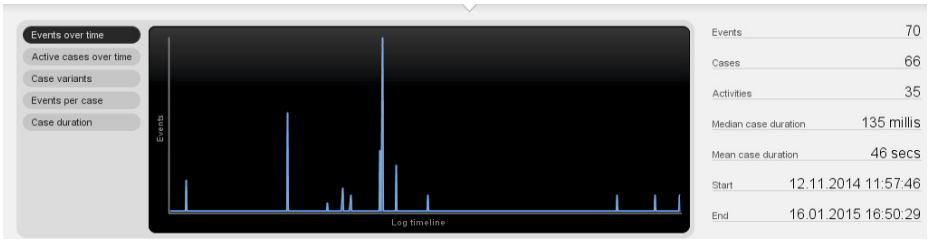

Fig. 7. Chart from Disco plotting the events over time.

6 Conclusion

In this paper we addressed the problem of mining and visualizing project-oriented business processes in a way that is informative to managers. We define an approach that takes VCS logs as input to generate Gantt charts. Our algorithm works under the assumptions that repositories reflect the hierarchical structure of the project, each work package is contained in a corresponding directory and project members commit their work regularly during active working times. The approach was implemented as a prototype and evaluated based on real-world data from open source projects.

In future work, we aim to extract further details of the VCS logs in order to calculate metrics that approximate the work effort. We plan to investigate on how the project mining approach is affected by project characteristics. Furthermore, we want to utilize statistical methods to better estimate the boundaries of the activities and work packages. Finally, we have already incorporated feedback from managers and plan to extend these to full user studies.

References

1. van der Aalst, W.: Formalization and verification of event-driven process chains. Information and Software Technology **41**(10), 639–650 (1999)
2. Baier, T., Mendling, J., Weske, M.: Bridging abstraction layers in process mining. Information Systems **46**, 123–139 (2014)
3. D'Ambros, M., Lanza, M.: A flexible framework to support collaborative software evolution analysis. In: Software Maintenance and Reengineering, pp. 3–12 (2008)

4. van Dongen, B.F., Van der Aalst, W.M.: A Meta Model for Process Mining Data. EMOI-INTEROP **160**, 30 (2005)
5. van Dongen, B.F., de Medeiros, A.K.A., Verbeek, H.M.W.E., Weijters, A.J.M.M.T., van der Aalst, W.M.P.: The ProM framework: a new era in process mining tool support. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 444–454. Springer, Heidelberg (2005)
6. Feldt, R., Staron, M., Hult, E., Liljegren, T.: Supporting software decision meetings: Heatmaps for visualising test and code measurements. In: 39th Conf. on Software Engineering and Advanced Applications, pp. 62–69. IEEE (2013)
7. Hou, Q., Ma, Y., Chen, J., Xu, Y.: An empirical study on inter-commit times in SVN. In: Int. Conf. on Software Eng. and Knowledge Eng., pp. 132–137 (2014)
8. Kagdi, H., Yusuf, S., Maletic, J.I.: Mining sequences of changed-files from version histories. In: Workshop on Mining Software Repositories, pp. 47–53. ACM (2006)
9. Kindler, E., Rubin, V., Schäfer, W.: Activity Mining for Discovering Software Process Models. Software Engineering **79**, 175–180 (2006)
10. Kindler, E., Rubin, V., Schäfer, W.: Incremental workflow mining based on document versioning information. In: Li, M., Boehm, B., Osterweil, L.J. (eds.) SPW 2005. LNCS, vol. 3840, pp. 287–301. Springer, Heidelberg (2006)
11. Ogawa, M., Ma, K.L.: Software evolution storylines. In: Proceedings of the 5th International Symposium on Software Visualization, pp. 35–42. ACM (2010)
12. OMG: BPMN 2.0. Recommendation, OMG (2011)
13. Pilato, C.M., Collins-Sussman, B., Fitzpatrick, B.W.: Version control with subversion. O'Reilly Media, Inc. (2008)
14. Poncin, W., Serebrenik, A., van den Brand, M.: Process mining software repositories. In: 2011 15th European Conference on Software Maintenance and Reengineering (CSMR), pp. 5–14. IEEE (2011)
15. Rubin, V., Günther, C.W., van der Aalst, W.M.P., Kindler, E., van Dongen, B.F., Schäfer, W.: Process mining framework for software processes. In: Wang, Q., Pfahl, D., Raffo, D.M. (eds.) ICSP 2007. LNCS, vol. 4470, pp. 169–181. Springer, Heidelberg (2007)
16. Rubin, V., Lomazova, I., van der Aalst, W.M.: Agile development with software process mining. In: Int. Conf. on Softw. and System Process, pp. 70–74 (2014)
17. Torvalds, L., Hamano, J.: Git: Fast version control system (2010). http://git-scm.com
18. Verbeek, H.M.W., Buijs, J.C.A.M., van Dongen, B.F., van der Aalst, W.M.P.: XES, XESame, and ProM 6. In: Soffer, P., Proper, E. (eds.) CAiSE Forum 2010. LNBIP, vol. 72, pp. 60–75. Springer, Heidelberg (2011)
19. Voinea, L., Telea, A.: An open framework for CVS repository querying, analysis and visualization. In: International Workshop on Mining Software Repositories (MSR 2006), pp. 33–39. ACM (2006)
20. Voinea, L., Telea, A.: Multiscale and multivariate visualizations of software evolution. In: Symposium on Software Visualization, pp. 115–124. ACM (2006)
21. Wilson, J.M.: Gantt charts: A centenary appreciation. European Journal of Operational Research **149**(2), 430–437 (2003)
22. Wu, J., Spitzer, C., Hassan, A., Holt, R.: Evolution spectrographs: visualizing punctuated change in software evolution. In: Workshop on Principles of Software Evolution, pp. 57–66, September 2004
23. Ying, A., Murphy, G., Ng, R., Chu-Carroll, M.: Predicting Source Code Changes by Mining Change History. IEEE Trans. Softw. Eng. **30**(9), 574–586 (2004)
24. Zimmermann, T., Weisgerber, P., Diehl, S., Zeller, A.: Mining version histories to guide software changes. In: Int. Conf. Software Engineering, pp. 563–572 (2004)