Activity Mining for Discovering Software Process Models

Ekkart Kindler, Vladimir Rubin, Wilhelm Schäfer
Software Engineering Group, University of Paderborn, Germany
[kindler, vroubine, wilhelm]@uni-paderborn.de

Abstract: Today’s enterprises spend much effort in obtaining precise models of their software engineering processes in order to improve the process capability of their organization and to advance one step further in the CMM. The manual design of process models, however, is complicated, time-consuming, error-prone, and the results are rapidly becoming obsolete; capabilities of human beings in detecting discrepancies between the actual process and the process model are rather limited. Consequently, automatic techniques for deriving and updating these process models are becoming more and more important.

In this paper, we present an approach that exploits the user interaction with a document version management system for the automatic derivation of a descriptive process model that faithfully reflects the real process.

1 Introduction

Enterprises can increase their process capability by obtaining faithful models of their ongoing software processes in a timely manner. Therefore, obtaining precise descriptive models of the software engineering processes is an important task in many enterprises. Today, process engineers and managers are solving this problem manually. This is complicated, time-consuming, error-prone, and the results are rapidly becoming obsolete; capabilities of human beings in detecting discrepancies between the actual process and the process model are rather limited. In addition, one usually does not start from scratch when defining an explicit process model; rather existing more or less informally executed processes have to be taken into account.

As a consequence, tool support is needed for deriving the process models. Since a document version management system is an essential part of the software engineering environment nowadays and is usually used even when no precise definition of the process exists, we suggest using it as a source of input information for an automatic approach. In this paper, we present the algorithms and models which constitute our approach to semi-automatically deriving a process model from versioning information, called incremental workflow mining.

This approach supports achieving the third level of the Capability Maturity Model (CMM) [Hum89, PCB'93] automatically once the second level is reached. The Key Process Areas (KPA) [PWG'93] of the second level (repeatable) of CMM include the use of a Software
Configuration Management System (SCM) and tracking the history reports of all the documents. On the third level (defined), an organization must have a well-defined software engineering process. Incremental workflow mining can be used for automatically discovering, formalizing and documenting the organization’s software process from the audit information of a software configuration management system.

The initial step for deriving the software process model is discovering the software process activities. These activities reflect the units of work carried out by employees of the software company. Since we deal with a document version management system, we consider the essential part of an activity to be its input and output documents and the users executing it; this is a document-oriented point of view on the activity. Our algorithm for discovering the activities is called activity mining.

After detecting the sequence and parallel relations on the set of activities, we can discover the structure of the process model, which is formalized as a Petri net. This model can be analyzed and verified using the wide range of techniques and tools available for this formalism. The algorithm for deriving the overall process model is called merging. Later, this Petri net model is transformed into a user-oriented format (UML Activity Diagrams), which is better understandable by software process engineers and managers of a company. As soon as new audit information is available, the algorithms can be executed again. The set of activities and the process model are refined, transformed and shown to the user again. This capability of refining the process model as soon as new log information is available allows us to have an up-to-date model all the time. Consequently, we call our approach incremental workflow mining.

The basic ideas of our approach were presented in our workshop paper [KRS05]; here, we do not deal with such steps as model analysis and model transformation to the user-oriented format, but present the details of the mining and merging algorithms that were implemented in our research prototype.

2 Related Work

The research in the area of software process mining started in the mid 90ties with new approaches to the grammar inference problem proposed by Cook and Wolf in their first articles, which was improved later [CW98]. This research considers events to be tokens and event streams to be sentences generated by some grammar, thus, deals with the grammar inference problem from event logs.

The other work from the software domain is in the area of mining from software repositories [MSR04, MSR05]. Like in our approach, they use SCM systems and especially CVS as a source of input information, but for measuring the project activity, detecting and predicting changes in code, advising newcomers to an open-source project and detecting the social dependencies between the developers [MLRW05, ZW04, CM03].

The first application of “process mining” to the workflow domain was presented by Agrawal et al. in 1998 [AGL98]. This approach models business processes as annotated activity graphs and assumes the absence of cycles in the event log, it is restricted to sequential
patterns only. The approach of Herbst and Karagiannis [HK99] uses machine learning techniques for acquisition and adaptation of workflow models. The seminal work in the area of process mining was presented by van der Aalst et al. [WvdA01, WvdA02]. Within this approach, workflow logs and classes of sound workflow nets are defined. Formal causality relations between events in logs, the $\alpha$-mining algorithm for discovering workflow models, and its improvements are presented. An extension of Aalst’s approach called “Multi-phase Process mining” [vDvdA05] and the approach used in ARIS Process Performance Manager (ARIS PPM) [Sch02] discover the models of process instances from event logs.

In addition to the software process and business process domains, the research concerning discovering the sequential patterns treats similar problems in the area of data mining. The work of Agrawal and Srikant deals with discovering sequential patterns in the databases of customer transactions [AS95].

In comparison to the classical approaches, we do not use event logs and we can not assume that activities are already defined; rather we work on the logs of SCM systems, thus, we have to introduce activity mining. We make use of our document-oriented view on the activities, i.e. the process model is derived from the inputs and outputs of the activities. We suggest coming up with the model very early and refining it as soon as additional log information is available. We call it incremental approach.

## 3 Incremental Workflow Mining

In this section, we describe our incremental workflow mining approach; moreover, we show the role of the activity mining and the merging algorithms in it. Then, we describe the format of the versioning log and our assumptions about the log.

We start with briefly explaining a traditional software engineering environment schema inspired by the works in the area of process-centered software engineering environments (PSEE) and software processes in general [Ost87, CKO92, FH93, Gru02]. Figure 1 gives an overview: The environment consists of an SCM system where the software product (models, documents, source code, etc...) is maintained and practitioners – the users who develop this product and interact with the SCM system. The software product is an instance of the software product structure, which is the informational model containing the software product documents and their relationships. The organizational structure is the organizational model containing the roles, organizational units and resources (practitioners).

In this schema, the Process Engineer (project manager or process engineering department) designs the process model using his experience, existing approaches, like V-model [DW00], RUP [JBR99], etc. The model is instantiated and practitioners follow it during the development of the software product (see Fig. 1).

There are the following problems with this schema:

- The designed model is prescriptive. It does not necessarily reflect the actual way of
how work is done in the company. So, there is a problem of getting a documented and formalized descriptive model of the actual processes for the companies that are already in business.

- Human possibilities in detecting discrepancies between the process model and the actual process are limited.
- The practitioner is not involved in the design of the process model, in spite of the fact that he is the best specialist in the part of the process that he carries out.

### 3.1 Incremental Approach

In contrast to the traditional way of work, in the incremental workflow mining approach, we go the other direction, the steps are shown in Fig. 2:

- First, we take the versioning log of the SCM system and do activity mining. As a result, we get the set of activities and the models of the process instances. This algorithm is defined in Sect. 4.
- Second, we take the set of discovered activities and do merging - derive the overall process model in a system internal formalism (Petri nets). This model can be analyzed and verified. This algorithm is defined in Sect. 5.
- Third, we make the transformation from the system internal model to the external model (UML2.0 Activity Diagrams [OMG03]). This external process model can be
shown to the user. This algorithm is, however, beyond the scope of this paper.

This approach works incrementally, i.e. as soon as new records are added to the versioning log, we refine the set of activities derived in the first step, refine the overall model and make the transformation again. Following this approach, after the process models are discovered, they must be inserted to the Workflow Management System (WfMS), where they are maintained and executed.

Initially, not the whole functionality of a WfMS is used. First, it is utilized only for storing the newly discovered models; after further refinements, when process models become more faithful, the WfMS starts advising the users and controlling their work in the company. In this context, the incremental approach enables gradual process support.

### 3.2 Versioning Log

In this section, we discuss the input information needed for our approach in more detail. SCMs and PDMs provide the audit information in form of logs or reports. This information comprises data about checkouts and checkins (commits) of documents. In this paper, we deal only with checkin information. The reason is that not all SCM systems enforce or support checkouts, checkins to the system are done more accurately – people check in the documents only after having done changes, and after the checkin they become responsible for them to the colleagues. In spite of this fact, checkout information can be rather valuable, especially for improving the models derived from the checkins. The source of such additional information could be also the tools, where the documents were changed.
before the checkin. These kinds of additional information will be regarded in our future work in this area.

An example of a checkin versioning log\(^1\) is shown in Table 1. In order to be independent from a particular SCM system, we define a common log format, which includes the typically available information. The versioning log consists of records (rows). Records are produced after checkins of the documents during different executions of different software processes. For example, in the log in Table 1, we have one process, let’s call it “Design Change”; during this process, some software module has to be designed, code has to be generated and tested and the design has to be reviewed. We must distinguish between the process model and the process instances; our goal is discovering the process model from the versioning log, which contains data about the process instances. So, there are three executions (process instances) of the process (process model) in the log: the first four records belong to the first execution, the second four – to the second execution, and the third four – to the third one. The set of records that belong to one execution of one process is called an execution log.

<table>
<thead>
<tr>
<th>Document</th>
<th>Date</th>
<th>Author</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>design</td>
<td>01.01.05 14:30</td>
<td>de</td>
<td>status: initial</td>
</tr>
<tr>
<td>code</td>
<td>01.01.05 15:00</td>
<td>de</td>
<td>status: generated</td>
</tr>
<tr>
<td>review</td>
<td>05.01.05 10:00</td>
<td>se</td>
<td>status: pending</td>
</tr>
<tr>
<td>testres</td>
<td>07.01.05 11:00</td>
<td>qa</td>
<td>status: initial, type: manual</td>
</tr>
<tr>
<td>design</td>
<td>01.02.05 11:00</td>
<td>de</td>
<td>status: initial</td>
</tr>
<tr>
<td>code</td>
<td>15.02.05 17:00</td>
<td>dev</td>
<td>status: generated</td>
</tr>
<tr>
<td>testres</td>
<td>20.02.05 09:00</td>
<td>dev</td>
<td>status: initial, type: manual</td>
</tr>
<tr>
<td>review</td>
<td>28.02.05 18:45</td>
<td>se</td>
<td>status: pending</td>
</tr>
<tr>
<td>design</td>
<td>01.03.05 11:00</td>
<td>de</td>
<td>status: initial</td>
</tr>
<tr>
<td>review</td>
<td>15.03.05 17:00</td>
<td>se</td>
<td>status: pending</td>
</tr>
<tr>
<td>code</td>
<td>20.03.05 09:00</td>
<td>dev</td>
<td>status: generated</td>
</tr>
<tr>
<td>testres</td>
<td>28.03.05 18:45</td>
<td>dev</td>
<td>status: initial, type: manual</td>
</tr>
</tbody>
</table>

The problem of detecting the log structure is application domain, SCM system and company-specific, we need interaction with the end user to solve it. For the rest of the paper, let us assume that we know the structure of the log, i.e. we know which records belong to which execution log and which execution logs belong to which process.

For the algorithms presented in this paper, we make the following assumptions on the versioning log (the assumptions will be relaxed in future):

- Different execution logs have the same naming conventions. E.g. the document containing the design is called “design” in all the execution logs.

\(^1\)Later, simply called versioning log.
• The document name and the comment together are the unique identifier of the record. So, there is no documents with the same name and the same comment within one execution log.

• There are no two records with the same timestamp in the same execution log.

For the rest of the paper, we can ignore the comments and deal only with document names; the timestamps can be converted to the values belonging to the set of natural numbers \( \mathbb{N} \), just to show the order of records. Let us assume, there are \( n \) execution logs and each execution log is given a unique identifier from the set \( C = \{1, \ldots, n\} \); for each log, there is the set \( D \) of document names and the set \( R \) of users, who committed the documents. Hence, a log can be defined the following way:

\[
L \subseteq C \times D \times \mathbb{N} \times R
\]

In our example, we have \( D = \{\text{design, code, review, testres}\} \) and \( R = \{\text{de, se, dev, qa}\} \). Each record of the versioning log is represented as a tuple, e.g. the first row in Table 1 is represented as \((1, \text{design}, 1, \text{de})\). The whole versioning log can be represented as a set of tuples:

\[
L = \{ \\
(1, \text{design}, 1, \text{de}), (1, \text{code}, 2, \text{dev}), (1, \text{review}, 3, \text{se}), (1, \text{testres}, 4, \text{qa}), \\
(2, \text{design}, 1, \text{de}), (2, \text{code}, 2, \text{dev}), (2, \text{testres}, 3, \text{dev}), (2, \text{review}, 4, \text{se}), \\
(3, \text{design}, 1, \text{de}), (3, \text{review}, 2, \text{se}), (3, \text{code}, 3, \text{dev}), (3, \text{testres}, 4, \text{dev})\\
\}
\]

For each tuple, we define the ‘.’ notation, which gives the concrete field value by its name. E.g. for \( r = (1, \text{design}, 1, \text{de}) \), we have \( r.c = 1 \), \( r.d = \text{design} \), \( r.t = 1 \) and \( r.u = \text{de} \).

## 4 Activity Mining

The goal of activity mining within the incremental workflow mining approach is discovering the set of activities from the versioning log and generating the set of process models for all the process executions. In this paper, we define only the algorithm for generating the set of activities. As mentioned earlier, an activity is a logical unit of work defined by the sets of its input and its output documents and by the set of resources that can execute it.

In the activity mining algorithm, we go through the execution logs looking for the outputs of the activities. Every document committed to the system is the output of some activity. E.g. for the log given in Table 1, the set \( D \) (see Sect. 3.2) contains all the outputs of the activities.

Further, for each activity, we also get the set of resources (subset of the set \( R \)) allowed to execute it: this set contains all the resources who committed the output document in some
execution log. E.g. the document “code” was committed by “de” in the first execution log and by “dev” in the second and the third one, thus the set of resources for the activity producing “code” is \{de, dev\}.

The most important problem here is finding the input of each activity, since this information is not given in the logs explicitly. For the time being, we assume that all the documents that were committed to the system always prior to the output of the activity belong to its input. For example, the document “design” was committed prior to the activity producing “code” in all execution logs, thus it is the input of the activity; the document “review” was also committed earlier, but only in the third execution log, thus it does not really belong to the input set.

Now, we define the algorithm more formally. For a fixed execution log with identifier \(c\) and a document \(k\), the set \(B_{c,k}\) contains the documents of this execution log that were committed before \(k\). The set \(B_k\) contains the documents that were committed before document \(k\) in all the execution logs. This set can be defined as the intersection of all sets \(B_{c,k}\) for all \(c\). The set \(U_k\) contains all the users that ever committed the document \(k\). The set \(A\) is exactly the set of activities – the output of the algorithm. Here, we present the algorithm in a formal notation:

Algorithm 1 (Activity Mining)
Let \(L\) be the versioning log, such that the following assumptions are met:

1. \(\forall r_i, r_k \in L : (r_i.c = r_k.c) \land (r_i.t = r_k.t) \Rightarrow r_i = r_k\)
2. \(\forall r_i, r_k \in L : (r_i.c = r_k.c) \land (r_i.d = r_k.d) \Rightarrow r_i = r_k\)

Then we define, for \(c \in C, k \in D\):

1. \(B_{c,k} = \{r_i.d | r_i \in L : (r.d = k) \land (r.c = c) \land (r_i.c = c) \land (r_i.t < r.t)\}\)
2. \(B_k = \bigcap_{c=1}^n B_{c,k}\)
3. \(U_k = \{r.u | r \in L : r.d = k\}\)
4. \(A = \{(B_k, \{k\}, U_k)\}\)

Each activity is a tuple \((I, O, U)\), where \(O\) (output) is a one-element set containing a document \(k\) from the set \(D\); \(I\) (input) and \(U\) (users) are the sets \(B_k\) and \(U_k\) for this document respectively. For our example in Table 1, for the document “code”, the sets \(B_{1,\text{code}}\) and \(B_{2,\text{code}}\) are equal to \{design\}, the set \(B_{3,\text{code}}\) is \{design, review\}, thus the set \(B_{\text{code}}\) is \{design\}; the set \(U_{\text{code}}\) is \{dev, de\}; therefore the activity is

\[
(\{\text{design}\}, \{\text{code}\}, \{\text{dev, de}\})
\]

In general, the set of the activities is a set:

\[
A \subseteq \{(I, O, U) | I \subset D, O \subset D, U \subseteq R\}
\]
4.1 Example

Let us consider the example log from Table 1, which was formalized in (2), to be the input again. To define the beginning and the end of each execution log explicitly, we insert the following records to the versioning log $L$:

$$(1, \text{start}, 0, \{\}), (1, \text{end}, 5, \{\}),$$
$$(2, \text{start}, 0, \{\}), (2, \text{end}, 5, \{\}),$$
$$(3, \text{start}, 0, \{\}), (3, \text{end}, 5, \{\})$$

Then, the activity mining algorithm will create the following sets:

1. $B_{1,\text{start}} = \{\}, B_{2,\text{start}} = \{\}, B_{3,\text{start}} = \{\}$,
   $B_{1,\text{design}} = \{\text{start}\}, B_{2,\text{design}} = \{\text{start}\}, B_{3,\text{design}} = \{\text{start}\}$,
   $B_{1,\text{code}} = \{\text{start}, \text{design}\}$,
   $B_{2,\text{code}} = \{\text{start}, \text{design}\}$,
   $B_{3,\text{code}} = \{\text{start}, \text{design}, \text{review}\}$,
   $B_{1,\text{review}} = \{\text{start}, \text{design}, \text{code}\}$,
   $B_{2,\text{review}} = \{\text{start}, \text{design}, \text{code}, \text{testres}\}$,
   $B_{3,\text{review}} = \{\text{start}, \text{design}\}$,
   $B_{1,\text{testres}} = \{\text{start}, \text{design}, \text{code}, \text{testres}\}$,
   $B_{2,\text{testres}} = \{\text{start}, \text{design}, \text{code}\}$,
   $B_{3,\text{testres}} = \{\text{start}, \text{design}, \text{code}, \text{review}\}$,
   $B_{1,\text{end}} = \{\text{start}, \text{design}, \text{code}, \text{review}, \text{testres}\}$,
   $B_{2,\text{end}} = \{\text{start}, \text{design}, \text{code}, \text{review}, \text{testres}\}$,
   $B_{3,\text{end}} = \{\text{start}, \text{design}, \text{code}, \text{review}, \text{testres}\}$

2. $B_{\text{start}} = \{\}$,
   $B_{\text{design}} = \{\text{start}\}$,
   $B_{\text{code}} = \{\text{start}, \text{design}\}$,
   $B_{\text{review}} = \{\text{start}, \text{design}\}$,
   $B_{\text{testres}} = \{\text{start}, \text{design}, \text{code}\}$,
   $B_{\text{end}} = \{\text{start}, \text{design}, \text{code}, \text{review}, \text{testres}\}$

3. $U_{\text{start}} = \{\}$,
   $U_{\text{design}} = \{\text{de}\}$,
   $U_{\text{code}} = \{\text{de}, \text{dev}\}$,
   $U_{\text{review}} = \{\text{se}\}$,
   $U_{\text{testres}} = \{\text{dev}, \text{qa}\}$,
   $U_{\text{end}} = \{\}$
4. \[ A = \{ \]
\[ (\{\}, \{\text{start}\}, \{\}), \]
\[ (\{\text{start}\}, \{\text{design}\}, \{\text{de}\}), \]
\[ (\{\text{start, design}\}, \{\text{review}\}, \{\text{se}\}), \]
\[ (\{\text{start, design}\}, \{\text{code}\}, \{\text{dev, de}\}), \]
\[ (\{\text{start, design, code}\}, \{\text{testres}\}, \{\text{dev, qa}\}), \]
\[ (\{\text{start, design, code, testres, review}\}, \{\text{end}\}, \{\}) \} \]

The last set is the desired set of activities. E.g for the activity
\[ (\{\text{start, design}\}, \{\text{review}\}, \{\text{se}\}) \]

we know that it is done by \textit{se}, gives the document \textit{review} as a result and needs the \textit{design} as precondition.

For the rest of the paper, we enumerate the activities in this set and remove the first activity, since it is abstract and is used only for introducing the document start, which shows the beginning of the process. Thus, the result of the activity mining algorithm looks the following way:

\[
A = \{ \\
\text{a}_1 = (\{\text{start}\}, \{\text{design}\}, \{\text{de}\}), \\
\text{a}_2 = (\{\text{start, design}\}, \{\text{review}\}, \{\text{se}\}), \\
\text{a}_3 = (\{\text{start, design}\}, \{\text{code}\}, \{\text{dev, de}\}), \\
\text{a}_4 = (\{\text{start, design, code}\}, \{\text{testres}\}, \{\text{dev, qa}\}), \\
\text{a}_5 = (\{\text{start, design, code, testres, review}\}, \{\text{end}\}, \{\}) \\
\}
\]

5 Merging

The goal of merging within the incremental workflow mining approach is generating the software process model using the set of activities. Since the set of activities obtained by activity mining contains the essence of the versioning log needed for deriving the process model, we do not deal with the log anymore.

First, we define three relations on the set of activities, such as sequence, parallel split and parallel join. The \textit{sequence} relation contains the pairs of activities that follow each other, i.e. the input of the second activity must contain the input and the output of the first one. E.g. activities \text{a}_3 and \text{a}_4 as presented on formula 4 are in the sequence relation, since the second one needs the set \{\text{start, design, code}\} as an input and the first produces the document “code” having the set \{\text{start, design}\} as an input. The \textit{parallel split} relation contains the activities which have the same input, but produce different output documents; these activities can be executed concurrently. E.g. \text{a}_2 and \text{a}_3 are in this relation. The last relation is \textit{parallel join}, the activities, that are followed by the activity that needs the input
and the output of both of them as an input, are in this relation. E.g. \(a_2\) and \(a_4\) are in this relation, since the activity \(a_5\) needs “review”, “testres” and \{start, design, code\} as an input.

We define the relations formally. Let \(a_i, a_k \in A\).

**Definition 1 (Sequence)** \(a_i \prec a_k \text{ iff } a_i \neq a_k \text{ and } a_i.I \cup a_i.O = a_k.I\)

**Definition 2 (Parallel split)** \(a_i \parallel a_k \text{ iff } a_i.I = a_k.I \text{ and } a_i.O \neq a_k.O\)

**Definition 3 (Parallel join)** \(a_i \parallel a_k \text{ iff } (a_i.I \cup a_i.O) \not\subseteq (a_k.I \cup a_k.O) \text{ and } (a_k.I \cup a_k.O) \not\subseteq (a_i.I \cup a_i.O) \text{ and } \exists a \in A : ((a_i.I \cup a_i.O) \cup (a_k.I \cup a_k.O) = a.I)\)

For the set of activities and the parallel split relation on this set, we define a set of maximum cliques (maximum subsets) \(\text{maxClique}(\parallel)\), where all the elements of each maximum clique are in a parallel split relation with each other. For example, activities \(a_2\) and \(a_3\) build a maximum clique, since \(a_2 \parallel a_3\) and \(a_3 \parallel a_2\); so, for our example, \(\text{maxClique}(\parallel) = \{\{a_2, a_3\}, \{a_1\}, \{a_4\}, \{a_5\}\}\).

Similarly, for the set of activities and the parallel join relation, we define \(\text{maxClique}(\parallel)\). For our example, \(\text{maxClique}(\parallel) = \{\{a_2, a_4\}, \{a_1\}, \{a_3\}, \{a_5\}\}\). For the rest of the paper, we exclude the subsets with only one element from \(\text{maxClique}\), since they are not necessary for introducing the parallel join pattern to the process model.

Now, after we have got the set of activities and the sequence and parallel relations on them, we present the algorithms for generating the Petri Net from these activities. These algorithms generate the control, the informational and the organizational aspects of the process models.

**Control Aspect** The goal of the control aspect algorithm is generating the Petri Net that represents the control aspect of the process model and, thus, shows the order of execution of the activities (see Fig. 3). This algorithm generates a sound workflow net according to the definition of van der Aalst et al. [vdAvH02].

For each activity, we generate a transition. For example, \(t_{\{\text{design}\}}\) for \(a_1\), \(t_{\{\text{code}\}}\) for \(a_3\). For each activity, which is not followed by parallel activities, we also generate one output place and an arc between them. E.g. for activity \(a_3\), we generate place \(p_{\{\text{code}\}}\) and an arc \((t_{\{\text{code}\}}, p_{\{\text{code}\}})\).

For activity \(a_3\), however, we have to generate two output places, since it is followed by parallel activities \(a_2\) and \(a_3\). Generally, we introduce a **parallel split pattern** [AHKB03] here, which consists of a set of places – one for each parallel branch; the transition has to be connected to all the places. E.g. in our case, we know that activities \(a_2\) and \(a_3\) belong to the maximum clique, therefore, we connect the transition of the activity \(a_1\), say \(t_{\{\text{design}\}}\), to these places. Let’s call these places \(p_{\{\{a_2, a_3\}, a_2\}}\) and \(p_{\{\{a_2, a_3\}, a_3\}}\), where \(\{a_2, a_3\}\) is the clique in the \(\text{maxClique}(\parallel)\) set and \(a_2\) and \(a_3\) are the activities in this clique, and the arcs \((t_{\{\text{design}\}}, p_{\{\{a_2, a_3\}, a_2\}})\) and \((t_{\{\text{design}\}}, p_{\{\{a_2, a_3\}, a_3\}})\) respectively. The places have to be
connected to the transitions of appropriate activities, i.e. \( p((a_2, a_3), a_2) \) has to be connected to the transition of \( a_2 \) i.e. \( t_{\{\text{review}\}} \) and \( p((a_2, a_3), a_3) \) to \( t_{\{\text{code}\}} \) respectively.

Additionally, we have to regard the case, when parallel activities do not have any activity as a predecessor. This can happen only if the input of these activities is equal to \( \text{start} \). In this case, we have to create an additional transition, connect it to the start place \( p_{\text{start}} \) and consider this transition to belong to the predecessor activity. After it, the algorithm works as described earlier.

Next, we have to connect all the activities that belong to the sequence relation, excluding those belonging to the parallel split relation, with each other. E.g. \( a_3 \) and \( a_4 \) are sequential, therefore, we connect the output place \( p_{\{\text{code}\}} \) with the transition \( t_{\{\text{testres}\}} \).

The last step is connecting the parallel join pattern. Here, we have to connect the output places of the activities that belong to the parallel join clique with the transition of the activity that needs the documents produced by all these activities as input. E.g. activities \( a_2 \) and \( a_4 \) belong to one set in \( \text{maxClique}(\{\}) \) relation and \( a_5 \) needs the documents of both of them for input. Thus, we have to connect the output places \( p_{\{\text{review}\}} \) and \( p_{\{\text{testres}\}} \) with the transition \( t_{\{\text{end}\}} \).

Now, we present the algorithm in a formal notation:

**Algorithm 2 (Merging algorithm)** Let \( A \) be the set of activities with sequence, parallel split and parallel join relations on it.
1. \( T_{act} = \{a.O|a \in A\} \)

2. \( T_\# = \begin{cases} \{t_\#\} & \text{if } (C \in maxClique(\#)) \land (a \in C) \land (a.I = \{\text{start}\}) \\ \{\} & \text{otherwise} \end{cases} \)

3. \( T_{control} = T_{act} \cup T_\# \)

4. \( P_{act} = \{a.O|a \in A \land \exists C \in maxClique(\#) : (b \in C) \land (a \prec b)\} \)

5. \( P_\# = \{(C,a)|(C \in maxClique(\#)) \land (a \in C)\} \)

6. \( P_{control} = P_{act} \cup P_\# \cup \{\text{start}\} \)

7. \( E_{start} = \begin{cases} \{(\text{start},a.O)\} & \text{if } (a.O \in T_{act}) \land (a.I = \{\text{start}\}) \land (T_\# = \{\}) \\ \{(\text{start},t_\#)\} & \text{if } T_\# \neq \{\} \end{cases} \)

8. \( E_{act} = \{(a,b)|(a \in T_{act}) \land (b \in P_{act}) \land (a = b)\} \)

9. \( E_d = \{(a.O,b.O)|a,b \in A : (a \prec b) \land (a.O \in P_{act}) \land (b.O \in T_{act})\} \)

10. \( E_{\#in} = \{(a.O,(C,b))|a,b \in A : (a.O \in T_{act}) \land (C \in maxClique(\#)) \land (b \in C) \land (a \prec b)\} \)

11. \( E_{\#instart} = \{(t_\#, (C,a))|a \in A : (C \in maxClique(\#)) \land (a \in C) \land (a.I = \{\text{start}\})\} \)

12. \( E_{\#out} = \{(b,O, (C,a))|a \in A : (C \in maxClique(\#)) \land (a \in C) \land (a.O \in P_{act}) \land (b.O \in T_{act}) \land (\bigcup_{C \in C} (a.I \cup c.O) = b.I)\} \)

13. \( E_\# = \{(a.O,b.O)|a,b \in A : (C \in maxClique(\#)) \land (a \in C) \land (a.O \in P_{act}) \land (b.O \in T_{act}) \land (\bigcup_{C \in C} (a.I \cup c.O) = b.I)\} \)

14. \( E_{control} = E_{start} \cup E_{act} \cup E_d \cup E_{\#in} \cup E_{\#instart} \cup E_{\#out} \cup E_\# \)

15. \( N_{control} = (P_{control}, T_{control}, E_{control}) \)

Altogether, the Petri Net \( N_{control} \) consists of the set of places \( P_{control} \), the set of transitions \( T_{control} \) and the set of arcs \( E_{control} \). The set of places consists of sets \( P_{act} \) and \( P_\# \) and an initial place. The first set contains the places, which are needed for connecting the activities with each other. The second set contains the places which belong to the parallel split pattern.

The set of transitions consists of two sets: \( T_{act} \) and \( T_\# \). The first set represents the activities themselves. The second set contains a transition that is needed to model the parallel split pattern after the initial place.

The set of arcs consists of seven sets. The set \( E_{start} \) either contains an arc connecting the initial place with the first transition or contains an arc connecting the initial place with the transition, which belongs to the parallel split pattern if it starts from the initial place. The set \( E_{act} \) contains the arcs connecting the \( T_{act} \) transitions with their output places. The set \( E_d \) connects the activities which belong to the sequence relation, excluding those belonging to the parallel split. The sets \( E_{\#in} \) and \( E_{\#out} \) connect the transitions and places belonging to the parallel split pattern to the rest of the net. The set \( E_\# \) represents the parallel join pattern and connects parallel branches.
Informational Aspect  Next, we present the algorithm for generating the informational aspect of the process model. In this algorithm, we create one place for each document in our document set \( D \). Since each activity produces a document as an output, we can connect the corresponding transitions to the document places. E.g. activity \( a_1 \) produces document \( \text{design} \) as an output, thus, we create a place \( p_{\text{design}} \) and connect the transition to this place. Therefore, after firing this transition, the place of the output document will get a token. In a formal notation, this algorithm looks following:

1. \( P_{\text{inf}} = D \cup \{\text{end}\} \)
2. \( E_{\text{inf}} = \{(a, b) | (a \in T_{\text{act}}) \land (b \in P_{\text{inf}}) \land (b \in a)\} \)
3. \( N_{\text{control}_{\text{inf}}} = (P_{\text{control}} \cup P_{\text{inf}}, T_{\text{control}}, E_{\text{control}} \cup E_{\text{inf}}) \)

Organizational Aspect  In the organizational aspect algorithm, we create one place for each user in our user set \( R \) and put a token to each of these places to show that the user is available. For example, for the user \( de \), we create a place \( p_{de} \). For each activity, for each user that can execute the activity, we create a transition, which fires when the activity is assigned to the user. For example, since the user \( de \) can execute \( a_1 \), we create a transition \( t(a_1, de) \). The user place \( p_{de} \) is connected to the transition \( t(a_1, de) \). For each activity, we create a place, which is marked initially and looses the token as soon as assignment is done. For example, for activity \( a_1 \), we create a place \( p_{a_1\{de\}} \). This place is connected to the transition \( t(a_1, de) \), and, thus, looses the token after firing the transition. For each activity, we create a place, which gets the token after firing the assignment transition and shows that the resources are available for the activity. For example, for activity \( a_1 \), we create a place \( p_{a_1\{de\}} \) and connect the \( t(a_1, de) \) transition to this place and this place to the activity’s transition \( t(\text{design}) \). If an activity can be executed by one of several users, we use the defined above elements to model the exclusive choice between these users. In a formal notation, this algorithm looks following:

1. \( P_{\text{org}_{\text{act}}} = \{a.U | a \in A\} \)
2. \( P_{\text{user}} = R \cup \{\text{end}\} \)
3. \( P_{\text{assign}} = \{a.U | a \in A\} \)
4. \( P_{\text{org}} = P_{\text{org}_{\text{act}}} \cup P_{\text{user}} \cup P_{\text{assign}} \)
5. \( T_{\text{assign}} = \{(a, u) | a \in A, u \in a.U\} \)
6. \( E_{\text{org}_{\text{act}}} = \{(a.U, a.O) | a \in A : (a.U \in P_{\text{org}_{\text{act}}} \land (a.O \in T_{\text{act}}))\} \)
7. \( E_{\text{org}_{\text{assign}}} = \{(u, (a, u)), ((a, u), u) | a \in A : (u \in P_{\text{user}}) \land ((a, u) \in T_{\text{assign}}))\} \)
8. \( E_{\text{assign}} = \{(a.U, (a, u)) | a \in A : (a.U \in P_{\text{assign}}) \land ((a, u) \in T_{\text{assign}}))\} \)
9. \( E_{\text{assign}_{\text{org}_{\text{act}}}} = \{(a, u), a.U) | a \in A : ((a, u) \in T_{\text{assign}}) \land (a.U \in P_{\text{org}_{\text{act}}})\} \)
10. \( E_{\text{org}} = E_{\text{org}_{\text{act}}} \cup E_{\text{org}_{\text{assign}}} \cup E_{\text{assign}} \cup E_{\text{assign}_{\text{org}_{\text{act}}}} \)
11. \( N_{\text{control-org}} = (P_{\text{control}} \cup P_{\text{org}}, T_{\text{control}} \cup T_{\text{assign}}, E_{\text{control}} \cup E_{\text{org}}) \)

So, the Petri Net that contains all three aspects is defined the following way:

\[
N = (P_{\text{control}} \cup P_{\text{inf}} \cup P_{\text{org}}, T_{\text{control}} \cup T_{\text{assign}}, E_{\text{control}} \cup E_{\text{inf}} \cup E_{\text{org}}) \quad (5)
\]

Strictly speaking, by now we have defined only the structure of the Petri Net; to define the Place/Transition system, we have to enrich it with an initial marking and a weight function. The Petri Net has the following initial marking: The initial place \( P_{\text{start}} \) in the control aspect has a token to show the start of the process. In the organizational aspect, places \( P_{\text{user}} \) have tokens to show the users’ availability; places \( P_{\text{assign}} \) are marked to show that user can be assigned some activity. In a formal notation, initial marking of the Petri Net is the following set (in general, a multiset):

\[
M = P_{\text{start}} \cup P_{\text{user}} \cup P_{\text{assign}} \quad (6)
\]

As a result, we get a Place/Transition system

\[
\Sigma = (N, M, W) \quad (7)
\]

where \( N \) is a net, \( M \) is an initial marking and \( W : (E_{\text{control}} \cup E_{\text{inf}} \cup E_{\text{org}}) \to \{1\} \) is a weight function.

Therefore, for our example in Table 1, after executing the algorithms, we get the Petri Net presented in Fig. 3.

5.1 Example

In this section, we present an example of the merging algorithm. We take the set \( A \) from section 4 as an input.

According to the definitions in Sect. 5, we have the following relations on the set \( A \):

\[
a_1 \preceq a_2, a_1 \preceq a_3, a_3 \preceq a_4,
\]

\[
a_2 \gtrless a_3, a_2 \lessgtr a_4
\]

Additionally, we build the following sets using the relations:

\[
\text{maxCliques}(\gtrless) = \{\{a_2, a_3\}\}
\]

\[
\text{maxCliques}(\lessgtr) = \{\{a_2, a_4\}\}
\]

In the example, we add a prefix “t” to the elements of the transition sets and “p” – to the elements of the place sets, because it makes easier to distinguish the elements of these sets. The merging algorithm forms the following sets:
Thus, having the sets of places \( P \), transitions \( T \), and arcs \( E \), we can build the Petri Net, see Fig. 4. Since we have one input place \( p_{\text{start}} \) and one output place \( p_{\text{end}} \), such that \( \bullet p_{\text{start}} = \{ \} \) and \( p_{\text{end}} \bullet = \{ \} \), we can say that we are building the workflow net [Aal98].

The informational aspect algorithm forms the following sets:

1. \( P_{\text{inf}} = \{ p_{\text{design}}, p_{\text{code}}, p_{\text{preview}}, p_{\text{testres}}, p_{\text{end}} \} \)
2. \( E_{inf} = \{ \langle t_{design}, p_{design} \rangle, \langle t_{code}, p_{code} \rangle, \langle t_{review}, p_{review} \rangle, \langle t_{testres}, p_{testres} \rangle, \langle t_{end}, p_{end} \rangle \} \)

3. \( N_{control, inf} = (P_{control} \cup P_{inf}, T_{control}, E_{control} \cup E_{inf}) \)

Thus, control and informational aspects together form the following Petri Net \( N_{control, inf} \), see Fig. 5.

The organizational aspect algorithm forms the following sets:

1. \( P_{org, act} = \{ p_{de}, p_{se}, p_{dev, de}, p_{dev, qa}, p_{end} \} \)

2. \( P_{user} = \{ p_{de}, p_{se}, p_{dev}, p_{qa}, p_{end} \} \)

3. \( P_{assign} = \{ p_{a_{de}}, p_{a_{se}}, p_{a_{dev, de}}, p_{a_{dev, qa}}, p_{a_{end}} \} \)

4. \( P_{org} = \{ \)
   \( p_{de}, p_{se}, p_{dev, de}, p_{dev, qa}, p_{end}, p_{de}, p_{se}, p_{dev}, p_{qa}, p_{end}, p_{a_{de}}, p_{a_{se}}, p_{a_{dev, de}}, p_{a_{dev, qa}}, p_{a_{end}} \} \)
5. \( T_{\text{assign}} = \{ t_{(a_1, \text{de})}, t_{(a_2, \text{sec})}, t_{(a_3, \text{dev})}, t_{(a_4, \text{dev})}, t_{(a_4, \text{qa})}, t_{(a_5, \text{end})} \} \)

6. \( E_{\text{org,act}} = \{ \)
\( \quad (p_{\{\text{de}\}}, t_{\{\text{design}\}}), (p_{\{\text{sec}\}}, t_{\{\text{review}\}}), (p_{\{\text{dev,de}\}}, t_{\{\text{code}\}}) \)
\( \quad (p_{\{\text{dev,qa}\}}, t_{\{\text{testres}\}}), (p_{\{\text{end}\}}, t_{\{\text{end}\}}) \} \)

7. \( E_{\text{org,assign}} = \{ \)
\( \quad (p_{\text{dev}}, t_{(a_1, \text{de})}), (t_{(a_1, \text{de})}, p_{\text{dev}}), \)
\( \quad (p_{\text{sec}}, t_{(a_2, \text{sec})}), (t_{(a_2, \text{sec})}, p_{\text{sec}}), \)
\( \quad (p_{\text{dev}}, t_{(a_3, \text{dev})}), (t_{(a_3, \text{dev})}, p_{\text{dev}}), \)
\( \quad (p_{\text{sec}}, t_{(a_3, \text{de})}), (t_{(a_3, \text{de})}, p_{\text{sec}}), \)
\( \quad (p_{\text{dev}}, t_{(a_4, \text{dev})}), (t_{(a_4, \text{dev})}, p_{\text{dev}}), \)
\( \quad (p_{\text{qa}}, t_{(a_4, \text{qa})}), (t_{(a_4, \text{qa})}, p_{\text{qa}}), \)
\( \quad (p_{\text{end}}, t_{(a_5, \text{end})}), (t_{(a_5, \text{end})}, p_{\text{end}}) \} \)

8. \( E_{\text{assign}} = \{ \)
\( \quad (p_{\{\text{de}\}}, t_{(a_1, \text{de})}), \)
\( \quad (p_{\{\text{sec}\}}, t_{(a_2, \text{sec})}), \)
\( \quad (p_{\{\text{dev,de}\}}, t_{(a_3, \text{dev})}), (p_{\{\text{dev,de}\}}, t_{(a_3, \text{de})}), \)
\( \quad (p_{\{\text{dev,qa}\}}, t_{(a_4, \text{dev})}), (p_{\{\text{dev,qa}\}}, t_{(a_4, \text{qa})}), \)
\( \quad (p_{\{\text{end}\}}, t_{(a_5, \text{end})}) \} \)

9. \( E_{\text{assign,org,act}} = \{ \)
\( \quad t_{(a_1, \text{de})}, p_{\{\text{de}\}}), \)
\( \quad t_{(a_2, \text{sec})}, p_{\{\text{sec}\}}), \)
\( \quad t_{(a_3, \text{dev})}, p_{\{\text{dev,de}\}}), (t_{(a_3, \text{de})}, p_{\{\text{dev,de}\}}), \)
\( \quad t_{(a_4, \text{dev})}, p_{\{\text{dev,qa}\}}), (t_{(a_4, \text{qa})}, p_{\{\text{dev,qa}\}}), \)
\( \quad t_{(a_5, \text{end})}, p_{\{\text{end}\}}) \} \}

10. \( E_{\text{org}} = E_{\text{org,act}} \cup E_{\text{org,assign}} \cup E_{\text{assign}} \cup E_{\text{assign,org,act}} \)

11. \( N_{\text{control,org}} = (P_{\text{control}} \cup P_{\text{org}}, T_{\text{control}} \cup T_{\text{assign}}; E_{\text{control}} \cup E_{\text{org}}) \)

In the example, we added prefix “p_\_” to the \( P_{\text{assign}} \) places to distinguish them from the \( P_{\text{org,act}} \) places, which are named with standard prefix “p”. Thus, control and organizational aspects together form the following Petri Net \( N_{\text{control,org}} \), see Fig. 6.

Thus, combining all the aspects, we get the Petri Net

\[ \Sigma = \left( \left( P_{\text{control}} \cup P_{\text{inf}} \cup P_{\text{org}}, T_{\text{control}} \cup T_{\text{assign}}, E_{\text{control}} \cup E_{\text{inf}} \cup E_{\text{org}} \right), M, W \right) \]

where, M, the initial marking, is following:

\[ M = \{ p_{\{\text{start}\}}, p_{\text{dev}}, p_{\text{sec}}, p_{\text{dev}}, p_{\text{qa}}, p_{\text{end}}, p_{\{\text{dev,de}\}}, p_{\{\text{dev,qa}\}}, p_{\{\text{sec}\}} \} \]

and the weight function is \( W : (E_{\text{control}} \cup E_{\text{inf}} \cup E_{\text{org}}) \to \{1\} \). This Petri Net was shown in Fig. 3.
6 Conclusion and Future Work

The activity mining and merging algorithms were implemented in SWI Prolog [Wie03] as a part of the incremental workflow mining research prototype. The declarative semantics of Prolog was considered to be especially valuable for implementing such kind of algorithms, since the program could be formulated as a set of first-order logic clauses defining relations on the set of activities. A special clause generates the resulting Petri net in a file following the syntax of the DOT language, which belongs to the GraphViz set of graph visualization tools [GN00]. From this dot file, the “.jpeg” file containing the image of the Petri Net is generated. An example of the automatically generated Petri net, which contains the control and the informational aspects, is given in Fig. 7.
In our approach, we do mining from different perspectives, we use data on such aspects of business process modeling as informational, organizational and control. Since Software Configuration Management systems do not immediately support the concept of activities, we must derive this information from the logs of the SCM system. Then, we discover the overall process model from this set of activities. Our approach has the following benefits:

- It does not need information on the existing activities; rather, we can mine this information from the versioning logs.
- After deriving the set of activities from the versioning log, we do not deal with logs any more. This set contains already sufficient information for discovering the overall process model using our merging algorithm.
- Our approach works incrementally and it is efficient, since all the information on the dependencies is captured in the set of activities. Therefore, we do not need to go back to the logs when mining the process.
- As a result, the algorithms produce a multi-perspective process model, which reflects the real process carried out in a company. This model can be later visualized, analyzed, verified and optimized by process engineers and managers of the company.

Consequently, the approach helps gradually improving the management of the software processes of a company along with incrementally improving the process models.

In this paper, we presented algorithms which work under rather restrictive assumptions about the versioning log (see Sect. 3.2). The algorithms were implemented and tested following these assumptions. In the future, we will improve the algorithms by means of relaxing these assumptions: several documents can be committed at the same time, there can be several commits of the same document within one execution log. We have to deal with loops and choices. Additionally, we must identify the types of documents, the roles of users and we must generate meaningful names for the activities. This can be achieved by exploiting checkout information of SCM systems or information of the additional tools and by interacting with the user.

For discovering not only software processes, but system engineering processes in general, we shall deal with similar audit information that can be obtained from Product Data Management (PDM) systems or other types of configuration and change management systems. Altogether, our approach supports mining process models from ad-hoc executions without having predefined information on them. This way, our approach supports increasing the repeatability of the processes; in terms of workflow terminology, the ad-hoc processes become administrative. In terms of the CMM, this means automatic support for increasing the maturity level of some enterprise from repeatable to defined.
References


