Software Process Line Discovery

Fabian Rojas Blum
Computer Science
Universidad de Chile
Santiago, Chile
fblum@dcc.uchile.cl

Jocelyn Simmonds
Computer Science
Universidad de Chile
Santiago, Chile
jsimmond@dcc.uchile.cl

María Cecilia Bastarrica
Computer Science
Universidad de Chile
Santiago, Chile
cecilia@dcc.uchile.cl

ABSTRACT
Companies define their software processes for planning and guiding projects. This definition is expensive and the resulting process is not always applied because it needs to be tailored for different kinds of projects. Software process lines (SPrL) take care of this variability by identifying those parts of the process that must be executed in all projects and those that may vary. Nevertheless, specifying SPrL is even more expensive than just specifying a process and therefore more work is lost if the SPrL is not applied after definition. Project logs register all activities that take place as part of software development projects. They contain activities that are part of the process and also unplanned activities, and they can also help detect the absence of planned activities that were never executed. Mining project logs has proven to be a promising approach for discovering the process that is actually applied. However, considering all the possible variations that can be found may result in overly complex processes. Some algorithms filter infrequent traces, but they may discard relevant traces. In this paper we propose the v-algorithm that defines two thresholds that divide traces in three clusters: highly frequent, variable and rare traces; the first cluster is used for building the base process, the second for defining the process variability, and the third is discarded as noise. We applied the v-algorithm to the project log of Mobius, a small Chilean software company. We were able to identify unexpected alternative ways of performing certain activities, as well as identifying an optional activity that was originally specified as mandatory.

Keywords
software process lines, process mining, process discovery, variability, noise in logs

1. INTRODUCTION
Software companies define their software development processes in order to be able to guide, analyze, monitor, measure, and improve them [12]. Formalization also enables tool support, allowing the automation of some activities and thus reducing the overall development effort. However, software projects carried out by a company can be quite varied, e.g., in size, complexity, team size, etc. Therefore, the formalized process must be tailored to each project before it can be used [1]. The set of all the possible software processes that a company follows, i.e., process variants, defines a Software Process Line (SPrL). Even though companies can count on a collection of predefined processes from which to choose from for each project context, model-based tailoring has proven to be an appealing alternative because it obtains the optimal process [5] for each project and the resulting SPrL is easier to evolve [3]. Note however, that to enable automated process tailoring, both the process and its variability have to be formally specified [9][10][14].

Several notations can be used to formalize the SPrL components. The base process (the SPrL common assets) can be modeled using: BPMN1, a general notation for modeling business processes; SPEM2, the OMG standard for modeling software processes; as well as UML Activity Diagrams, and Petri nets, among others. Process variability can be formalized using extensions of process modeling notations, like BPMNt [14], set-theoretic collection operations [24] or vSPEM [10], or using variability modeling notations, like Feature Models [7] and Orthogonal Variability Models (OVDM) [15]. In general, these notations are equally expressive; their use depends on the type of analysis to be carried out, as well as tool availability and required usability [17].

We have worked with seven small and medium enterprises (SMEs) in Chile over the last five years, aiding them in the formalization of their software process: Rhiscom, KI, Anisoft, Mobius, PowerData, DTS and Imagen. The process engineer of only one of these companies managed to formalize the base process by himself. Also the process engineers of three companies were aware of the relevance of SPrL definition and tailoring; however, none of them could specify the process variability unaided. We also realized afterwards that the formalized processes have had poor adoption among developers of these companies [2], even though the process engineers had striven to represent the actual process being applied.

Even if no formal process is followed, companies keep a register of the activities carried out by their employees. In particular, all our partner companies keep project logs in some format. These logs can be mined in order to discover

1 Business Process Model and Notation
2 Software & Systems Process Engineering Metamodel
the process that was applied in practice, although some filtering or ETL (Extraction, Transformation and Loading) tasks may have to be carried out before process mining techniques can be applied. Process discovery is a subfield of process mining that focuses on defining a process using project logs as input [8], combining techniques from data mining and BPM (business process management) [23]. General solutions like the α-algorithm [22] take into account the complete log for process discovery, considering all traces as valid. Unfortunately, this approach can produce overly complex process models for even small logs, that are not easily understood by process engineers or project managers. Other algorithms, like fuzzy miner [6], heuristics miner [25] and genetic miner [11] make use of different parameters to define thresholds that are used to prune infrequent traces and the process model is built using only the most frequent traces. These algorithms yield simpler processes than the α-algorithm, but possibly omit relevant process traces. None of these algorithms considers process variability as something natural that is worth specifying, even though there is a consensus about its relevance. Recent work [4][24] has started addressing this issue for general business processes.

In this paper we propose the v-algorithm, a software process discovery approach that explicitly considers variability, thus defining a SPrL instead of a single process model. The v-algorithm takes into account the frequency of the relations between activities found in the log. We define two frequency thresholds, $t_1$ and $t_2$, such that frequently frequent relations ($f > t_1$) define the base software process, relations between the two thresholds ($t_1 \leq f \leq t_2$) define SPrL variability, and infrequent relations ($f < t_2$) are classified as noise and discarded. The initial idea of this algorithm was sketched out in [16]. As our approach is inspired by the α-algorithm, and implemented as a plugin for ProM\textsuperscript{3}, processes in this paper are formalized using Petri nets.

We illustrate the application of our algorithm by discovering the SPrL of Mobius, a small Chilean software company that we have been working with. Before SPrL discovery, the process engineer specified the organizational process, with the help of our team. After SPrL discovery, we compared the SPrL process variants with the originally specified process, realizing that one of these variants is almost identical to the process initially defined by the process engineer. A key finding of our case study was that, in the original process definition, the process engineer did not consider that there were alternative ways of carrying out some activities. Also, one of the activities specified in the original process was almost never carried out in practice, and should thus be considered optional. This last kind of result suggests that either the process is not being correctly followed and developers should be made more aware of the task that they are skipping, so that they perform it in subsequent projects. Another option is that the defined task is actually not required and it should therefore be removed from the defined process. In this way, the v-algorithm serves the purpose of validating the defined process, and identifying its variability whenever it was not previously identified. The process engineer is still in charge of defining how the discovered variability is resolved before process enactment.

The rest of the paper is structured as follows. Section 2 presents and discusses existing approaches for process discovery. Section 3 illustrates the problem with these approaches when applying them to SPrL discovery, using a case study. Section 4 defines the v-algorithm and shows its application to the case study, comparing the discovered process with the one originally specified. Section 5 discusses the results for different threshold values, as well as threats to validity. Section 6 discusses related work. Finally, some conclusions and future work are stated in Sec. 7.

2. PROCESS DISCOVERY

Process discovery [8] tries to automatically build a process model that describes the behavior contained in an event log. Event logs record information about significant events; in the software process domain, the beginning and end of activities are examples of relevant events. Table 1 shows a excerpt of a log taken from real projects developed by Mobius. This small fragment shows two types of traces, with five and nine events, respectively.

This log meets the two basic requirements for process mining [20]: each event within an execution trace must be chronologically ordered and all events must be associated to both an execution trace and an activity. Different kinds of analysis can be applied depending on the data collected. For example, the log in Table 1 also includes the activity’s performer and associated cost, which we are currently not using in our approach. As we focus on process workflow discovery, we just need information about event traces and their frequency. In this example, trace 1 ($a, b, d, e, g$) and trace 2 ($a, b, c, f, b, d, e, g$) from Table 1 appear 17 and 4 times in log $L_1$, respectively; as stated in Table 2.

2.1 Noiseless Process Discovery

One can extract a process from an event log by analyzing the causality relationships between events. The α-algorithm [21] is an example of such an algorithm. This algorithm takes as input an event log and produces a Workflow-net (WF-net) representing the process extracted from the log. WF-nets [19] are a subset of Petri nets [13], and as such

<table>
<thead>
<tr>
<th>Event Id</th>
<th>Timestamp</th>
<th>Activity</th>
<th>Resource</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001</td>
<td>10-09-2014:08.30</td>
<td>Initial Meeting (a)</td>
<td>John, Peter</td>
<td>2</td>
</tr>
<tr>
<td>2002</td>
<td>11-09-2014:15.35</td>
<td>Requirement Analysis (b)</td>
<td>John, Lucas</td>
<td>4</td>
</tr>
<tr>
<td>2003</td>
<td>13-09-2014:08.00</td>
<td>Req. Elicitation with scenarios (d)</td>
<td>Maria, Lucas</td>
<td>4</td>
</tr>
<tr>
<td>2004</td>
<td>15-09-2014:09.43</td>
<td>Customer Approval (e)</td>
<td>Peter, John</td>
<td>4</td>
</tr>
<tr>
<td>2005</td>
<td>15-09-2014:11.12</td>
<td>Establish Req. Base Line (g)</td>
<td>Javier</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 1: Log $L_1$ - fragment showing two types of traces.

<table>
<thead>
<tr>
<th>Id</th>
<th>Trace</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a, b, d, e, g</td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td>a, b, c, f, b, d, e, g</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2: Log $L_1$ (compressed representation).
Table 3: Ordering relations matrix extracted from $L_1$.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>≣</td>
<td>→</td>
<td>≣</td>
<td>≣</td>
<td>→</td>
<td>→</td>
<td>→</td>
</tr>
<tr>
<td>b</td>
<td>→</td>
<td>≣</td>
<td>→</td>
<td>→</td>
<td>≣</td>
<td>→</td>
<td>→</td>
</tr>
<tr>
<td>c</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
</tr>
<tr>
<td>d</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
</tr>
<tr>
<td>e</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
</tr>
<tr>
<td>f</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
</tr>
<tr>
<td>g</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
</tr>
</tbody>
</table>

they are directed bipartite graphs that consist of two types of nodes (places and transitions), arcs and tokens. WF-nets have one initial input place (i.e., a place without incoming arcs), one final output place (i.e., a place without outgoing arcs), and are strongly connected.

In the context of process modeling, transitions represent process activities, so in the rest of this paper, the terms transition and activity will be used interchangeably. Arcs connect transitions with places and vice-versa, but not nodes of the same type. Both places and arcs are used to express process flow, and more complex constructs (e.g., AND, XOR joins and splits) are used to model concurrency. Tokens are used to represent the current state of execution. Figure 1 shows an example of a WF-net. It has 6 places, 7 transitions, 14 arcs and 1 token.

This WF-net was generated using the four log-based ordering relationships defined by the $\alpha$-algorithm:

1. $a \succ_L b$: iff $a$ directly precedes $b$ in some trace (direct relationship)
2. $a \rightarrow_L b$: iff $a \succ_L b$ and $b \not\in_L a$ (causality)
3. $a \parallel_L b$: iff $a \succ_L b$ and $b \succ_L a$ (possible concurrency)
4. $a \parallel_L b$: iff $a \neq_L b$ and $b \neq_L a$ (no direct relationship)

Only one of these relationships can hold for every pair of activities. Table 3 shows a matrix representation of the ordering relationships present in log $L_1$. This matrix is used to discover workflow patterns, which are then used to create a model of the underlying process. For example, if $a \rightarrow_L b$, $a \rightarrow_L c$ and $b \parallel_L c$ hold, then this suggests that both $b$ and $c$ can be executed in parallel after $a$ (AND-split pattern). Similarly, if $a \rightarrow_L b$, $a \rightarrow_L c$ and $b \parallel_L c$ suggests that after $a$, either $b$ or $c$ can be executed (XOR-split pattern). The WF-net shown in Fig. 1 was generated using these patterns, a complete description of these patterns can be found in [21].

2.2 Noisy Process Discovery

Noise can be defined as infrequent or exceptional traces that appear in the log. For example, according to Table 2, trace 2 might be considered noise because it represents less than 25% of the traces. Another source of noise is incorrect or incompletely logged traces, e.g., the log was analyzed during a run of the process, producing an incompletely logged trace. The $\alpha$-algorithm does not distinguish noise in logs, i.e., it considers the whole set of traces.

For practical applications of process discovery, it is essential to consider noise. Some existing algorithms are capable of handling noise through the definition of metrics and thresholds. Heuristic mining [25] builds a model with assigned weights to each activity and a dependency measure between them. This indicates the degree of certainty that there is a dependency relation between two activities. After mining the log, the model is trimmed by setting up a cut-off threshold for the dependencies that should be represented in the model. Genetic Miner [11] is based on mutation and crossover techniques from genetic algorithms. This algorithm tackles noise by defining a set of fitness measures, stopping the process of model generation (“individuals”) once it finds an individual with the predefined fitness configuration, or a previously defined maximum number of generations have been computed, or the model being updated stops changing. Other algorithms have additional goals to process discovery. For example, Fuzzy Miner [6] builds a model that can be visualized at different levels of abstractions. This visualization can be updated based on significance and correlation measures.

All these algorithms discard or hide noise and thus the resulting processes may be simpler. However, they fail to identify the parts of the process that naturally vary according to the particularities of different projects.

3. PROBLEM

Mobius develops integrated software and hardware solutions for Santiago’s public transportation system. Mobius has 20 employees: 8 are directly working in software maintenance and development. Employees perform more than one role in the company, according to the traditional software engineering disciplines (e.g., developer, analyst, tester, etc). Projects typically take from a couple of days for incidents to three or four months for large development projects.

This company started formalizing its development process two years ago, as part of the ADAPTE project\(^4\). A significant amount of effort went into defining the SPPrL: Mobius employees, with the help of the ADAPTE team, specified the organizational process model in ten 3 to 4 hour sessions, and process variability was modeled over a period of five days. Mobius’ general software development process is loosely based on the Rational Unified Process (RUP), and is quite detailed in its definition, with 104 tasks, 10 roles and 44 work products.

We will now show how the algorithms discussed in the previous section discover processes from logs. To this end we are going to use another sample log, $L_2$, which is slightly more complex than $L_1$. This log represents several executions of Mobius’ Development process, which is shown in Fig. 2. Log $L_2$ has 141 traces, of 11 different types, as shown in Table 4. This table shows a compact representation of the complete log, including frequency by trace type. Note that this log in-

\(^4\)http://www.adapte.cl
Figure 2: Mobius’ Development process.

Table 4: Log $L_2$ (compressed representation).

<table>
<thead>
<tr>
<th>Id</th>
<th>Trace</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a, b, c, d, e, f, g, h, j, k</td>
<td>36</td>
</tr>
<tr>
<td>2</td>
<td>a, b, c, d, e, f, g, h, j, k</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>a, b, d, e, f, g, h, j, k</td>
<td>28</td>
</tr>
<tr>
<td>4</td>
<td>a, b, l, d, e, m, g, h, j, k</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>a, b, d, l, e, m, g, h, j, k</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>a, b, c, d, f, g, h, j, k</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>a, b, l, e, m, g, d, c, e, f, g, h, j, k</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>a, b, l, e, m, g, h, i, j, k</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>a, b, d, k</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>a, b, k</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>a, b, d, c, e, k</td>
<td>1</td>
</tr>
</tbody>
</table>

includes two activities that do not appear in Fig. 2: Elaborate Simple Design (l) and Create Functional Tests (m), suggesting that they may either be unplanned activity variants or a part of exceptional executions.

Note that logs generally contain many more traces. However, our intention is to show that even with this relatively small log, the resulting process may be quite complex. Figure 3 shows the process models produced by the $\alpha$-algorithm when different subsets of the log are considered. Since this algorithm does not take noise into account, the data in the Frequency column from Table 4 was not used to generate these processes. As illustrated by Fig. 3(b), even with only 6 types of traces, the output process model is more complex than the one shown in Fig. 3(a), because of possibly concurrent activities.

Comparing figures 3(b) and (c), we notice a great increase in complexity. Five trace types were added to produce the model in Fig. 3(c), these traces represent less than 14% of the log traces. The log tells us that the behavior shown in traces 7, 8 (representing almost 10% of the log traces) is not frequent, but from our experience with developers, they present perfectly valid cases that must be represented in the process model.

Traces with id 9, 10 and 11 (representing less than 4% ) are examples of low-occurring trace types that correspond to a particular execution of the process where something went wrong (e.g., the project went over-budget and an unavoidable refund had to be made) leading to a sudden termination of the project. As this kind of execution traces are undesirable and sporadic, they should not be present in the process model, and should be considered as noise. Since the $\alpha$-algorithm does not take frequencies into account, it can not appropriately deal with this kind of traces.

Moreover, there are additional problems with these models other than complexity. The $\alpha$-algorithm is a one-step algorithm, i.e., the ordering relation is only computed between events that appear together in the log. This means that, by including trace id = 7, activity d now has two input places: one after executing b, and another after executing g. However, in all the traces in Table 4, d occurs before g, so there is no way to add a token to d’s second input place, meaning that the final output place of the model in Fig. 3(c) is unreachable for all the traces used to produce the WF-net, so it is unsound [18].

Figures 4 and 5 show the process models discovered from log $L_2$ when using Genetic Miner and heuristics mining, respectively. The results (Heuristic nets) were transformed using the Convert Heuristics net into Petri net ProM plugin. We do not get an SPrL with these algorithms, just different base process models depending on the threshold values selected. Using a low cut-off threshold leads to the discovery of complex models, but using a high cut-off threshold ignores behavior that could have been relevant. Considering the accelerated growth in complexity that each trace may cause, even picking an acceptable threshold often leads to complex models. For example, heuristics miner calculates a frequency-based metric that is used to indicate how certainly there is a dependency relation between each pair of activities. With these values a so called all-activities-connected heuristic is used to take the ”best” candidates (elements with higher value). After that, some parameters are used to filter the rest of the relations. Dependency is used to accept dependency relations that have a value above a defined threshold. Relative to best accepts relations that have a dependency measure for which the difference with the "best" is lower than a certain threshold. We used the default parameter settings given in ProM for both heuristics miner and genetic miner.
Figure 3: Results of running the α-algorithm on $L_2$, with: (a) trace types 1 and 2, (b) trace types 1 - 6, and (c) all trace types (diagrams generated using ProM, but labels edited for legibility).
4. SPRL DISCOVERY

Certainly, logs contain some behavior that should be discarded, but they also contain infrequent or exceptional behavior that is relevant for the process model. As modeling these last traces directly in the process model causes an increase in model complexity, we propose to model them as variability of the general process model, which is extracted from the frequent and common behavior. For this purpose, we introduce two thresholds ($t_1$ and $t_2$) that split the log into three clusters based on the frequency of the relations between activities found in the log. The first cluster represents common and frequent behavior that should be expressed in the general process model ($f > t_1$). The second represents infrequent but representative behavior that should also be modeled in the process ($t_1 \geq f > t_2$). However, as this would have an impact in complexity, we specify it using a variation model. The last cluster includes incorrect or undesirable behavior that is just omitted; this is actual noise ($f < t_2$).

4.1 The v-algorithm

Our solution, the v-algorithm, makes use of the ordering relations defined by the $\alpha$-algorithm, but takes into account the frequency of these relations. In this section, we illustrate the three steps of our algorithm by showing its application to log $L_2$.

**Step 1:** Given a log $L$, we produce its ordering relations matrix. In this step, we simultaneously allocate a weight to each element of the matrix based on the frequency of causality and possible concurrency relations (see Sec. 2.1, relationships 2 and 3). Table 5 shows the ordering relations matrix extracted from log $L_2$ and Table 6 shows the corresponding weights; we call this second matrix the ordering weight matrix. A particular value of this table is referred to as $W_{a,b}$, where $a$ and $b$ represent the corresponding row and column activities from the table.

**Step 2:** Pick values for thresholds $t_1$ and $t_2$ and generate the relations clusters. Note that $t_1 > t_2$, $t_2 \geq 0$ and $t_1 \leq n$, where $n$ is the number of traces of log $L$ being analyzed. These thresholds are used to generate the three clusters mentioned before, based on the values $W_{a,b}$ of the generated ordering weight matrix (OWM). For log $L_2$, thresholds $t_1 = 36$ and $t_2 = 8$ split the relations as follows: the first group ($W_{a,b} > t_1$) contains frequent relations that should be directly represented in the process model and are highlighted in green the OWM; the second group ($t_1 \geq W_{a,b} > t_2$) includes infrequent but desirable relations and are highlighted in blue in the OWM; even though these relations should be represented in the process model, their direct representation introduces complexity. The last group ($W_{a,b} < t_2$) contains relations that should not be represented in the process model since they are considered undesirable behavior or noise; these are highlighted in red in the OWM.

**Step 3:** Generate the process model and identify process variability. Here, the green OWM relations are used to directly model the process that represents the general process model. In this paper, we used the $\alpha$-algorithm to accomplish this task, but any process discovery algorithm could have been used instead. The red OWM relations are discarded. Finally, alternative and optional variation points and variants are identified by analyzing the blue OWM re-
lusions. The resulting SPRL has three types of activities: common, optional and alternative.

Activity $a$ is represented as optional if it participates in blue OWM relations and there is an alternative way to reach its associated activities. The case where $t_1 = 36$, $t_2 = 4$ does not present any examples of optional activities. However, if we changed $t_2$ to 4, then activity $i$ would be in the blue OWM cluster since $W_{b,1} = 7 \geq t_2$, $W_{i,j} = 7 \geq t_2$, but $W_{b,j} = 129 > t_1$ means that activity $j$ can be directly reached from $h$ without executing $i$. Optional activities are drawn above or below the place between the two activities that bypass it.

Activity $a$ is represented as an alternative if it participates in blue OWM relations and it exhibits the same behavior as another activity, i.e., they have the same input and output activities. According to the weights in Table 6, both $b$ and $d$ are possible inputs for activities $c$ and $l$ ($W_{b,c} = 74$, $W_{d,c} = 36$, $W_{b,l} = 25$ and $W_{d,l} = 9$). Additionally, $c$ and $l$ both have the same possible output activities $d$ and $e$ ($W_{c,d} = 36$, $W_{e,c} = 74$, $W_{l,d} = 11$ and $W_{l,e} = 23$). It is easy to see that the relations pertaining to $c$ belong to the green OWM group, whereas those pertaining to $l$ belong to the blue OWM group. As a result, the SPRL includes both activities $c$ and $l$, where $l$ is an alternative to $c$. The same happens with activities $f$ and $m$, where $m$ is identified as an alternative to $f$ for $t_1 = 36$ and $t_2 = 8$.

The resulting SPRL is shown in Fig. 6: activities disconnected from the WF-net that appear above or below a connected activity represent alternative activities, while the disconnected activities that appear above or below a place in the WF-net represent optional activities. Comparing Fig. 6 and Fig. 3(c), we see that the model in Fig. 6 is much more simple, e.g., there are fewer transitions (10 instead of 13), fewer places (13 instead of 15) and fewer arcs (22 instead of 40). Additionally, it stores the information of all the variable activities found in the process model: activity Elaborate Simple Design ($l$) was found to be an alternative to activity Elaborate Design ($c$), that was not originally specified in the process. A similar situation happens with Create Functional Tests ($m$), which is an alternative for Create Unit Tests ($f$). These findings provide information for enriching the original software process specification. The model in Fig. 6 is also more similar to the Development process shown in Fig. 2, it is only missing the backward transitions from the decision nodes, as this behavior was not strongly represented in $L_2$. These transitions can be added by the process engineer if she/he thinks that they are required. For these values of the thresholds ($t_1=36$ and $t_2=8$) we did not identify any optional activities; we will see in the next section that this is not always the case.

5. DISCUSSION

In this section we discuss threshold selection and threats to validity of our case study.

5.1 Varying Thresholds

In the previous section, we showed how the SPRL was built for $t_1 = 36$ and $t_2 = 8$. We chose the value $t_2 = 36$ since the most frequent trace appears 36 times in log $L_2$ (see Table 4). By doing this, we can guarantee that the most frequent execution trace will be represented in the SPRL. The value of $t_2$ was arbitrarily chosen.

We now show how changes to the thresholds affects the discovered SPRL. Let us now consider $t_1 = 36$ as before, but now $t_2 = 12$, i.e., more relations may fall below $t_2$ and will thus be discarded as noise. Table 7 shows the same ordering weight matrix as Table 6, but now colored according to the new thresholds. Since $c$ and $l$ no longer share input and output activities, as $W_{d,c}=9$ and $W_{l,d}=11$ are considered noise, activity $l$ is no longer modeled as an alternative to $c$. Figure 7 shows the resulting SPRL. We can see that there is only one pair of alternative activities: $f$ and $m$; $c$ is now mandatory, and it does not have alternatives.

Let us now consider a lower value of $t_2$: 4. Table 8 shows how transitions are classified. Only $W_{b,k}$, $W_{d,k}$ and $W_{e,k}$ fall below the threshold and are discarded. The resulting SPRL, shown in Fig. 8, keeps the same alternative transitions as the first example, $c$ and $l$, and $f$ and $m$, and it also identified an optional transition: $i$, that is shown as a transition above the place between $b$ and $j$.

Of the three discovered SPRLs, Fig. 8 is the most representative of the behavior found in log $L_2$, as it not only includes alternative activities that had not been identified by the process engineer, but also identifies activity $i$ as optional in some cases. All this, without adding the complexity to the process model that we have seen with existing process discovery algorithms. The process engineer must now define the context rules that specify how SPRL variability is resolved, so that the SPRL can be tailored to individual projects.

Note that in this case, Mobius developers followed a similar process during development, so the base process is more or less similar in the discovered SPRLs. This means that we do not observe any changes in the SPRL when changing the value of $t_1$ to nearby values. However, this is not true if there is a greater variety of traces. In that case, choosing good thresholds for the $v$-algorithm will be harder because it may not be clear how to differentiate the clusters.
of relationships. We will continue studying industrial cases in order to determine possible relationships that may help identify candidate threshold values for logs.

5.2 Threats to Validity

**Internal Validity.** We have applied our approach at companies that either: a) have a formally defined process, or b) follow a rigorous process, even though it has not yet been formalized. In both cases, there are logs that can be mined by our algorithm. However, project logs of companies that do not have such conditions will probably be more disorganized, and thus the results of applying the v-algorithm may not be clear or useful.

**External Validity.** Clearly, this approach can only be applied to companies that keep project logs. Also, at least for the moment, the v-algorithm only applies to logs complying with the XES standard. We have converted the original Mobius logs to this format, since not only this company, but all our industrial partners, use different tools and formats to log their projects.

**Construct Validity.** The v-algorithm claims to identify variable activities. However, the output of our algorithm only makes sense if the same activity is registered with the same identifier throughout the log. In the case of infrequent or exceptional behavior, this may not be the case as some log information is entered manually as free-form text. In this case, we would need a preprocessing step that identifies these activities, so that the process engineer can standardize activity names before applying the v-algorithm.

**Reliability.** As we have discussed in previous subsection, choosing the appropriate threshold values is essential for obtaining a SPrL with the appropriate abstraction level. For the presented case study, we choose $t_1$ and $t_2$ so that the discovered SPrL could be compared to the one specified by the process engineer. We do not yet have conclusive guidelines for picking these values, but our intuition is that the lower bound for $t_1$ is the frequency of the most frequent trace in the log.

6. RELATED WORK

Weidlich et al. [24] present an approach for managing process variability. They provide a set algebra for behavioral profiles, i.e., abstractions for reasoning about order constraints. This is particularly useful for decoupled variants that may give different names to similar activities where similarities are identified mainly for their behavior. In the case of software project logs, the data is stored altogether registering traces of activities of the same process, and therefore the situation is simpler.

The work of Buijs et al. [4] is the closest to ours in spirit, as the authors propose and compare four approaches for extracting configurable process models from event logs. A configurable process model is a representation of a business process, where activities can be hidden and paths may be blocked during configuration. The first two approaches proposed by the authors create a configurable process model by merging individually discovered process models, whereas the second two approaches focus on discovering a single process model and its possible configurations (intuitively similar to our approach). In all approaches, the result is a process model represented as a tree, where internal nodes document control flow and leaf nodes represent activities. Variability is modeled using node labels that guide tree traversal, a configured process model is created by exploring a tree branch.

In the case of software project logs, the lack of separation between common and variable makes these models proposed in [4] harder to use for enactment, especially since looping behavior leads to node replications (as the output models
are trees). Most importantly, the idea of alternative activities, a concept which is explicitly present in the software process domain (as there are different ways of carrying out a same activity), is not present in the authors’ definition of configuration.

7. SUMMARY AND FUTURE WORK

Companies that specify a SPrL have at their disposal a rigorously defined process, which can also be adapted to suit the needs of individual projects in a structured manner. During SPrL specification, the process engineer must channelize his/her knowledge about the company process, as well as past experience in executing software development projects. This is a labor-intensive task with a high learning curve, and there is no guarantee that the process modeled by the process engineer actually represents the process carried out in practice. This is detrimental to SPI initiatives, as it is unclear if the discrepancies between the defined and applied process are due to problems with employees (e.g., lack of familiarity with the process), or if the process was incorrectly specified.

If a company registers project events in logs, e.g., the beginning and end of project activities, then this problem can be remedied by applying process discovery algorithms. Noiseless algorithms, like $\alpha$-algorithm, consider all events as valid events. As a result, process models produced by this type of algorithm become more and more complex as different types of traces are considered in the analysis, so these models are rarely useful for project enactment. Introducing a cut-off threshold to separate frequent behavior from noise does not necessarily lead to better models: picking a low threshold means that more exceptional behavior is added to the process model, increasing its complexity, but picking a higher threshold may leave out exceptional but infrequent behavior.

To address these problems, we propose partitioning the log relations into three groups, by defining two frequency thresholds ($\nu$-algorithm). This allows us to explicitly separate frequent and infrequent behavior, as well as noise. The first group of relations is used to build the general process model, the second group is used to identify process variability, and the third group is ignored. The output of our approach is a SPrL represented as a Petri net, which includes information about activity variability. An advantage of this approach is that it allows us to quickly setup SPrLs at companies that have no formally defined processes, as long as they log process events.

We have applied our approach to the logs of one of our industrial partners, Mobius. Here, the process engineer had already defined company’s general process, taking over 40 hours to do so with the help of our team. In this paper, we applied the $\nu$-algorithm to the Development process, a subset of the general process. In order to examine the effect of the chosen threshold values, we generated SPrLs for different thresholds pairs. The base processes of the resulting SPrLs were all less complex than the models discovered using existing process discovery algorithms, and more similar to the process that had been hand-crafted by the process engineer. We also identified alternative and optional activities that had not been taken into account by the process engineer during the initial process model specification.

**Future Work.** The results of the Mobius case study are quite promising; however, we need to continue our experiments with additional datasets. This includes comparing SPrLs created with our and other approaches with those hand-crafted by process engineers for the same company, as well as determining empirical estimates for the appropriate threshold values. Also, the process models in the discovered SPrLs are not complete: since we used the $\alpha$-algorithm to generate these models, we only compute relations between events that appear together in the log. This makes it hard to discover looping behavior, especially this type of behavior appears infrequently in the log. Also, the process engineer must examine the variability identified in the SPrL to determine how variability must be resolved for different project contexts. For example, if an activity is identified as optional, in which types of projects should this activity be carried out? Finally, the process engineer must also examine differences between the defined and the applied process in order to understand if employees are carrying out the specified project correctly; this can be done by checking for log conformance.

We have currently limited ourselves to two types of variability (optionality and alternatives) as these appear in the variability models created by the process engineers that we are currently working with. However, models using only these two types of variability may be underspecified, allowing more configurations than those that the process engineer intended, just because they are not familiar with all the concepts of variation models, like constraints. For example, we can deduce that there is a “requires” constraint between $f$ and $c$ in our case study ($f$ requires $c$), but since this constraint is not included in the SPrL, more behaviors than those seen in the log are allowed. We are currently exploring how constraints like “requires” and “excludes” can be discovered automatically from logs.

8. REFERENCES


