Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Architecture of Information Systems Research Group

# Iterative Preprocessing of Event Logs

*Master's Thesis*

P.J. van Heumen, BICT

Supervisors:
dr. ir. B.F. van Dongen
J.C.A.M. Buijs, MSc.
dr. G.H.L. Fletcher

Eindhoven, 14 July, 2011

# Abstract

Information systems, nowadays, play a large role within organizations in supporting business processes. Typically, all events that are handled by the system are recorded. These records of activities handled by the information system are called *event logs*. Ideally, an event log contains *all* events that have occurred, although this is not guaranteed to be the case.

In the area of process mining, we investigate these event logs. There is a certain amount of data available in an event log, such as event name, resource and date and time of execution. We then use algorithms to mine all kinds of information, such as process models, social networks and performance data. However, for these algorithms to function as expected, they need to make certain assumptions. Preprocessing is used to prepare the event log for the conditions specified by the analysis algorithm. Furthermore, different kinds of analysis require different kinds of preprocessing. To discover the main stream process, we require an event log that contains only the most commonly executed cases, while the discovery of the exceptional cases requires an event log that contains only the least commonly executed cases.

In this Master's project we look at ways to improve the support for preprocessing. To achieve this we first we look at scientific literature, a number of available visualizations that can be used for preprocessing, and process mining tool support. We find some issues with the current state of support for preprocessing and improve on this by designing a framework that is capable of providing those features that are lacking up to now.

We design and implement a preprocessing framework that offers support for iterative processing and we modify existing visualizations to take advantage of the features provided by this framework. The implementation of the preprocessing framework shows that preprocessing can be done in a flexible way, allowing the process analyst to focus on the task at hand.

**Keywords**: process analysis, event log, iterative preprocessing, interactive visualization, filtering

# Preface

This Master's thesis is the result of my graduation project at the Architecture of Information Systems research group. The graduation project is part of my Master of Computer Science and Engineering study at Eindhoven University of Technology.

First of all, I would like to thank Boudewijn van Dongen for supervising me during this project. Also, many thanks to Joos Buijs, my second supervisor, for being available whenever I had questions or needed feedback on some crazy idea, and especially for proof reading my thesis as many times as he did. Furthermore, thanks go out to J.C. for being so readily available for discussions and feedback on the project and related matters, and of course for the time he invested in extending his research plug-in (Pattern Abstractions) for this Master's project.

Also, thanks go out to both Rom van Arendonk and Erik Nooijen for the many interesting discussions during the graduation project, to Gino Rombley, a good friend, for proof reading my thesis.

And last, but certainly not least, to my parents for supporting me all those years.

Danny van Heumen,
July 2011

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

This Master's thesis is the result of a Master's project carried out within the *Architecture of Information Systems* research group of the Mathematics and Computer Science department at Eindhoven University of Technology. The *Architecture of Information Systems* research group focus on three main areas: 'process-aware information systems' (PAIS), 'process modeling and analysis', and 'process mining'. This Master's project is executed within the Process Mining context of the research group.

The first section introduces the main concepts of *Process Mining*. Section 1.2 introduces the problem statement. Section 1.3 introduces the project goal for this Master's project. Section 1.4 defines the research scope for the project. Section 1.5 describes the research method followed.

## 1.1 Process Mining

In this section, we introduce the concepts of process mining that are relevant for this project, starting with 'Event Logs', followed by 'Process Mining', 'Preprocessing', 'The Process Mining Framework (ProM)' and the 'Current state of Process Mining'.

### 1.1.1 Event Logs

Information systems, nowadays, play a large role within organizations in supporting business processes. Typically, all events that are handled by the system are recorded. These records of activities handled by the information system are called *event logs*.

Event logs provide a historical record of activities that have been executed by the information system and that have been recorded. Ideally, an event log contains *all* events that have occurred, although this is not guaranteed to be the case. There are a number of reasons why events may not have been recorded, such as failures and whenever logging is disabled. There is also no guarantee that events are recorded in the same order as they occur, hence we are dependent on the data that are mentioned explicitly in the event log. In the area of process mining, we investigate event logs, hence we are inherently bound to the existence of the events in the event log and the accuracy of the data on the recorded events.

As said, we record data of the events in the event log. Basically, any information can be recorded, although the data are typically directly related to the event. Event data that is usually recorded is:

1. The name of the activity of which this event is an occurrence.
2. The current state of the activity (i.e. has the activity been scheduled, started, completed, see [4]).
3. The date and time of execution.

4. The name of the resource that initiated the event.

These 4 data elements provide sufficient data to enable the analysis of the event log. Analysis can focus on the control flow, performance and organizational structure. More information and more details can provide a more accurate analysis, however these 4 data elements already provide the data needed for most analyses. Besides these, there are many other types of data that can be recorded, such as the instance name/number for repeating tasks, costs of execution, the group and/or role of the resource, and data on the process that is used for decision making purposes. All of these data are recorded as part of an event in the event log.

### 1.1.2 Process Mining

In the previous section, we have seen that event logs contain all kinds of data related to activities that have been handled in information systems. Since these activities are related to business processes, the recorded events reflect how business processes are being supported.

*Process Mining* is the area of research that analyzes these event logs in order to gain knowledge about the way the information system handles activities and about the underlying business process. The data in event logs can be analyzed and information can be extracted from them. Among the data that is extracted are process models in various notations, social networks and performance information, although it is in no means limited to these types of information.

As it is nicely stated in the introduction of 'Process Diagnostics', in [3, p. 22]:

> Process mining looks "inside the process" to answer questions like: "What does the actual process look like?", "Are the executed logs conform specification, i.e. follow all cases the specified process model?", "Are there any bottlenecks in the process?", "Who executes what tasks?", "Who typically work together?", etc.
> ...
> Answers on these questions can serve as a handle for organizations to answer two of the main questions: "Are we in control?" and "Does the information system really reflect the state of affairs of the business process?".

Typically there is a goal we are working towards when we are analyzing the event log. With process mining there are 2 main goals. The first is investigative. We have a certain question and try to find as much information as possible that may answer the question. For this goal, we are looking for specific points of interest, a range of time or data and analyze this in order to find an answer to the question or to confirm a suspicion. The second goal is exploratory. We do not have an explicit question, but instead we want to either do a superficial analysis and find out the general idea of the process, or do a in-depth analysis and find out as much as possible about an event log. In either case, we are not looking for anything in particular, instead we look at the complete event log and try to get the general idea of the process and to find anomalies. An anomaly is anything that draws your attention. In many cases it is either the unexpected existence of a pattern, or no pattern where one is expected.

The difficulty in the field of *Process Mining* is in its dependency on the data in the event logs. Event logs often do not contain all the data that we would like to have. To discover the original process model, we would need every possible variation of the process to be available in the event log. Event logs typically do not contain every possible trace through the process, simply because not every possible variation is guaranteed to have happened. Therefore it is almost impossible to reconstruct the original model if we would require all data to be available. Instead, we use algorithms to derive missing information from the data that is available in the event log.

In the previous section, we looked at certain types of data that might be available in an event log. By using one or more of these data elements, we can derive all kinds of information about the execution history of the information system. Using the name of the activity, we can derive which activities subsequently got executed. If we also use the data on the current state of the

activity, we can discover whether it has been finished in one go, or if it has been paused during the execution. If we additionally use date and time data then we can find out how long an activity took to execute. Using resource data we can find out who typically execute this particular activity. These are just a few examples of the information that can be "mined".

To follow up on a certain type of information, we visualize the information in a particular way. To acquire information on the *Control Flow* of the process, we primarily order or aggregate (depending on the type of analysis) on the activity name and state. The resulting information is related to or partitioned in the various activities. The information we can discover, relates to the control flow. We call this the *Control Flow perspective*. Similarly, if we primarily focus on resource data, we assume the *Resource perspective*, since any information that we can discover relates to the resources. The type of information we wish to uncover determines which perspective(s) to use. By looking at execution times in conjunction with activities, we take the *Performance perspective*. Assuming we have previously recorded cost data, we can use this data to find out the costs involved with the execution of certain activities and resources. On the other hand, we can also check the conformance of an event to a preexisting process model. Suppose there is already a process model available. We can evaluate how good the event log fits this process model, thereby evaluating how well the execution of the process in reality fits this preexisting process model.

### 1.1.3 Preprocessing

For the event log analysis algorithms to function correctly, they need to make certain assumptions on the event log that is being processed. Since different algorithms have different methods and goals, these assumptions are specific to each algorithm. Possible assumptions are that the first event that is listed for a certain case, is actually the first event that is executed at the start of the case. The last event listed for a certain case, is actually the last event executed before the case is finished. And that the list of events for a case is actually the complete list of events, i.e. no events are missing, and all events that are listed have actually occurred.

Another purpose for preprocessing is related to the goal of the analysis. Event logs contain all recorded events. There are a number of situations where we do not want to analyze the complete event log. If we are looking into the most common paths through the process then we require an event log that does not contain the exceptional cases. Hence we filter out cases that only occur incidentally. Some event logs, especially recorded over a longer period of time, show signs of change. For example, changes can occur in the way the process is being handled, which resources execute a particular activity, etc. For these reasons, we have to preprocess the event log before analyzing it.

To satisfy the analyst's and the algorithm's requirements, the event log is preprocessed. All events that do not satisfy the requirements are being removed from the event log. For some specific types of analysis, data can also be added to the event log, although we have to be careful not to introduce false truths. The end result is an event log that satisfies all requirements, and is suitable for analysis.

### 1.1.4 The Process Mining Framework (ProM)

*ProM* [14] is the Process Mining Framework that is developed at Eindhoven University of Technology. ProM is a framework that is set up for the purpose of enabling users to easily load any kind of process data (e.g. several types of event logs, process models) and analyze it using any of the available plug-ins.

For researchers and developers, ProM provides the framework to quickly and efficiently develop new plug-ins. The framework provided by ProM offers solutions for common actions such as importing and exporting of data, and cooperating with other plug-ins. As a result, the researchers and developers can focus their attention on the actual implementation of their own research.

The newest version of the framework, ProM 6, has been introduced in September of 2010. With the new version of ProM, a new event log file format has been introduced. The event log file

format, XES (eXtensible Event Stream) [5, 14], is designed to be simple, flexible, extensible and expressive.

### 1.1.5 Current State of Process Mining

Now that the concepts of process mining are explained, we look at the current state of process mining. In the area of *Process Mining*, most of the research goes to solving specific problems. Most of the research conducted is in the area of mining and analyzing models. As a result, papers most often talk about the details of a single algorithm and how to use it.

Analyzing an event log, on the other hand, takes a different set of skills. We have to work with the event log as it is, with all its peculiarities. Depending on our goal (exploration or investigative research) we need to look at the event log in a different way. For exploration, we look at the complete event log. While for investigative research, we look at very specific parts of the event log, i.e. those that are relevant to our question.

A paper on the subject of event log analysis is 'Process Diagnostics: a Method Based on Process Mining' [3]. The method described by the authors consists of 6 steps. First, they extract the event information from the log. Then they familiarize themselves with the event log and preprocess the event log. Subsequently, the event log is analyzed for control flow, performance and role analysis, optionally with an additional preprocessing step. Finally, they prepare the results for presentation to the organization.

The method proposed in 'Process Diagnostics' is a description of a high level approach. Doing process analysis is very much an iterative process. For each analysis step, the event log has to be preprocessed. Preprocessing typically takes multiple iterations. In between iterations, several different visualizations are used to visualize the event log in order to confirm the correct execution of previous preprocessing steps, and to determine possible next preprocessing steps. Furthermore, during the analysis, new questions may arise that also need to be investigated, hence adding new iterations to the analysis process.

## 1.2 Problem description

For preprocessing, there is no "one size fits all" solution. We have seen this in Section 1.1.3. Depending on our goal and the algorithm we require, we have to preprocess the event log in a certain way. Mining and analysis algorithms have strict requirements that have to be met, or at least as closely met as possible, in order to successfully do the mining or analysis. Furthermore, as analyst we also have our requirements. We cannot allow events to be in the event log that are disruptive for the analysis. These events have to be filtered out beforehand.

The analysis of a process log is often described as a linear process, as depicted in Figure 1.1. First the event data is extracted from the data source, such as a database. Then the event log is preprocessed in order to remove inconsistencies. Next, the process is analyzed and finally a report is compiled that describes the findings.

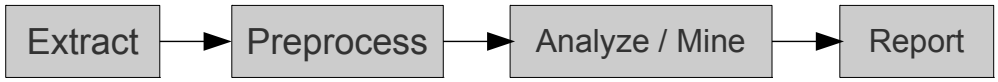


**Figure 1.1:** The classic view on process analysis: a linear process.

However, this is not quite accurate. It turns out that in practice the process analysis is very much an iterative process, depicted in Figure 1.2. Assuming that we need to only analyze one kind of case, the event data has to be extracted only once. During preprocessing we find that it typically takes several iterations to thoroughly prepare the event log for analysis. We filter out events that must be removed, and visualize resulting event log to confirm previous preprocessing steps and to determine possible next preprocessing steps. After we are finished preprocessing the event log, we analyze the event log. We then carefully investigate the result. If the result is not good enough, e.g. a discovered process model is too large and complicated to offer any useful insight into the business process, we have a possibility to go back and further preprocess the event log or retry the analysis with different parameters. If the result raises additional questions, we can add these as new iterations in the analysis process. If, on the other hand, everything is according to wish, we can gather the results and continue with the next analysis iteration.

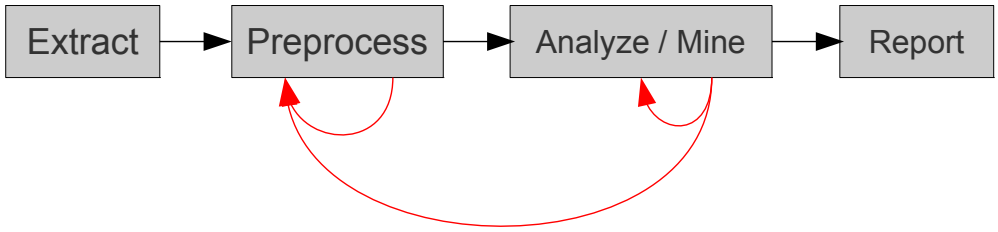


**Figure 1.2:** A more realistic view on process analysis.

## 1.3 Project goal

As described in Section 1.2, there is a difference between the ideal process analysis method, as depicted in Figure 1.1, and what happens in reality when process analysis is applied, as depicted in Figure 1.2. In order to bridge the gap between the ideal and the real-life analysis process, we want to enhance the preprocessing step by introducing support for a more flexible and interactive way of working with these iterations, since they are inherent to the analysis process.

The main goal for this graduation project is **to develop a framework for iterative preprocessing of event logs**.

## 1.4 Research scope

As we have seen, the ideal analysis process consists of 4 steps. First is the extraction step, where we extract event data from a data source. The second is the preprocessing step, which is the primary step in this project. The third is the analysis step, and the fourth is the presentation step.

In this thesis, the 'extraction' step is of no interest to us. We assume that an event log is available in a suitable file format. Similarly, the 'presentation' step is not important for this project either. The 'analysis' step would also be out of the scope of this project, however, it turns out that in practice there is an overlap between preprocessing and analyzing the event log. For this reason, the 'analysis' step is within the scope of the project. So is the 'preprocessing' step, since it is the main topic for this project. Figure 1.3 visualizes the scope of the research project.

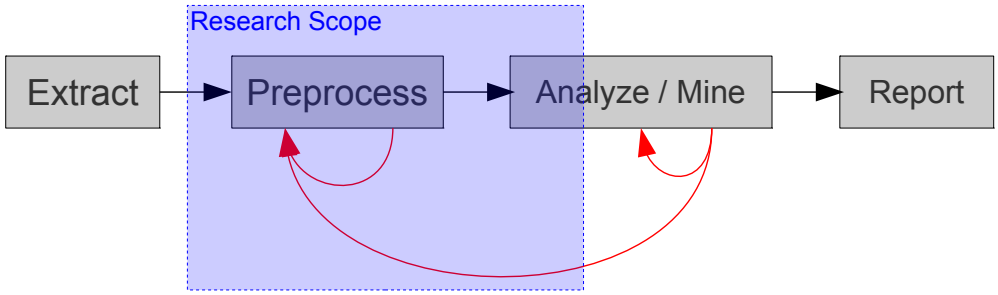


**Figure 1.3:** The research scope of the project: 'preprocessing' and a small part of 'analysis/mining'.

## 1.5 Research method and outline

To reach the goal of this Master's project, we take the following approach:

- **Visualization analysis**
  In Chapter 2, we analyze a number of available visualizations. We evaluate these visualizations by looking at four aspects of visualizations. Afterwards, we evaluate the summaries of the analyses and discuss the issues that have been identified.

- **Requirements**
  In Chapter 3, we derive the requirements for the project. First, we evaluate the issues with current preprocessing support. Then we determine the requirements based on the issues identified in Chapter 2 and the evaluation of the current state on preprocessing in this chapter. Finally, we define requirements for the framework and guidelines for the interactive visualizations.

- **Design**
  In Chapter 4, we design the preprocessing framework to support preprocessing according to the requirements defined in Chapter 3. We also design a number of interactive visualizations to function as reference implementations to demonstrate the features of the preprocessing framework. Using a top-down approach we discuss the plan, the design of the framework, the modifications that are required to make the visualizations function in the preprocessing framework, and the most important decisions related to the design.

- **Implementation**
  In Chapter 5, we discuss the implementation of the preprocessing framework and the reference implementations that are implemented according to the design that was constructed in Chapter 4. We also discuss the most important implementation decisions, which are related to the design decisions discussed in Chapter 4.

- **Conclusions**
  In Chapter 6, we conclude the project.

- **Future work**
  In Chapter 7, we discuss some ideas for future work.

# Chapter 2

# Visualization analysis

In this section, we explore a selection of visualizations available in ProM 6, ProM 5.2 and related literature. These visualizations can be used in event log preprocessing. We summarize the evaluation of each visualization, looking mainly at four properties. We introduce the properties in the next section. Afterwards, in Section 2.14, we draw our conclusions on the current state of the visualizations[1] with regard to preprocessing.

## 2.1 Summaries of the analyzed visualization

In the next sections we summarize the evaluation of each individual plug-in. Our evaluations are focused on the *visualization* properties of a plug-in. The evaluations are based on four aspects discussed below. The visualizations are of a wide variety, ranging from visualization providing basic information, to process discovery, social network discovery and performance analysis.

We discuss the following properties:

- **Visual aspects**: Properties related to information visualization. Visualization can be both in a textual format and a graphical format. We will look at the way information is conveyed. Examples: How clear is the visualization? How is the color usage? Are there unintended side effects to the way the information is visualized?
- **Support for interaction**: The level of support for interactivity within the visualization. Does a visualization provide interactive options, and to what extent? For this property, the rating ranges from having a completely static picture (low rating) to having a fully interacting and reconfigurable visualization (high rating).
- **Performance aspects**: The performance aspects of a visualization. How fast does a visualization load? Are there any noticeable performance issues during visualization? What are the memory requirements?
- **Predictability of results**: How predictable are the results of a visualization? Specifically, how predictable are the results after making modification to the event log.

The purpose of this rating is not to determine which plug-ins are "good" and which are "bad". This rating is not about good or bad, since there are plug-ins that are primarily for analysis of data, while others are primarily for the visualization of data. We use '+' to indicate a high rating, '±' for a medium rating, and '−' for a low rating. The focus is primarily on the suitability for use in preprocessing. Also notice that the ratings of these analyses are *not* meant to be used to compare two visualizations to each other. There are different types of visualizations and these should not be compared directly. The full evaluation of the visualizations is available in Appendix A.

[1] The analyses were performed performed around November of 2010. Some visualizations might have been improved since then. We know, for example, that the performance of the Pattern Abstractions plug-in and the Trace Alignment plug-in have significantly improved since this analysis.

## 2.2 Dotted Chart

A familiar visualization in the world of Process Mining is the Dotted Chart [3, 12]. An example is shown in Figure 2.1. The Dotted Chart is a chart that visualizes all traces and events in a single picture. This enables us to search for patterns, or lack thereof, as a way of exploring or investigating the log.

The Dotted Chart plug-in can visualize the data in a number of different ways, but the core principle is always the same. It visualizes events from the event log as dots in a grid using a variant of time (actual time, relative to the largest trace in the event log, ratio of current trace) on the X-axis and a chosen classification on the Y-axis, such as cases, activity names or resource names. An additional data attribute can be used, which is visualized by the color of each dot in the grid. Typically, the chosen classification is the case identifier and that gives each case its own row on the Y-axis, but it is also possible to select, e.g., the resources or event classes as data values for the Y-axis. The color of the dot typically corresponds to the name of the activity.

The type of information that can be derived from the chart depends on the chosen classification. E.g. if we would choose to display the cases on the Y-axis, events would be partitioned based on the case identifier and this would enable you to compare cases and execution times of events within different cases. On the other hand, if you choose 'Activity Classes' as the classification for the Y-axis, it would show dots for every time an activity has happened. The density of events and whether or not an activity ceases to occur is now information that can be derived.

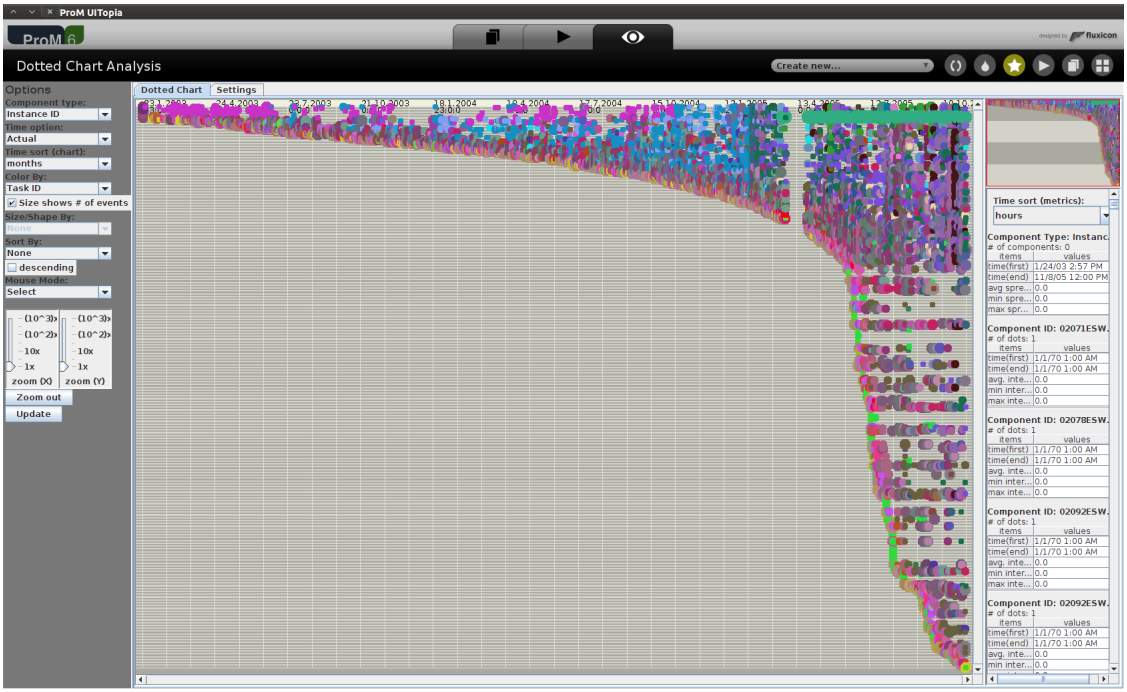


**Figure 2.1:** A screenshot of the Dotted Chart visualization

The Dotted Chart is a powerful visualization. It is a representation of an event log that can visualize a complete event log in a single picture, without losing the ability to visualize individual events. An additional advantage is that the user can inspect a graphical representation of the event log for patterns that are hard to quantify and find with numerical analysis.

The Dotted Chart focuses on traces and events. The perspective it takes depends on the options that are chosen. Note that the perspective for the case of the Dotted Chart does not have a clear meaning, since we can either choose the color of the dots to correspond to each distinct resource, or we can choose resources to be used as the component type. The first option presents resource

information when visualizing the original event log, while the latter restructures the event log such that each distinct resource is a case, while it presents some other value, typically the activity name, as the color of the dots.

The Dotted Chart also has some disadvantages. The first is that the graphical representation is less accurate. This accuracy issue is two-fold: firstly, given the limited number of pixels on a display, it may be that dots get grouped together that actually do not belong to the exact same coordinate. This is a technical limitation due to the granularity of the data that is available. Secondly, the current implementation uses relatively large dots, i.e. larger than a single row or column, that are allowed to overlap even if they belong to different cells of the grid. Note that dots may increase in size, which causes them to overlap even more. These dots are also relatively large, which makes it hard to see the smaller anomalies or patterns that should be visible in the Dotted Chart. Additionally, information is displayed inside a tooltip, but this tooltip can become so large as to fill roughly half of the screen with data about the selected events. This obscures half of the Dotted Chart visualization. To make matters worse, when many events have been selected it will not fit everything on the screen, which makes the tooltip information incomplete.

All of the advantages and disadvantages we just discussed, contribute to the *visual aspects* property of the visualization. Although there are some very attractive properties, the disadvantages are significant too. Overall the *visual aspects* score a medium rating.

The Dotted Chart presents options for zooming, selecting dots, configuration of component axis and time axis, and sorting on a number of criteria. Given that all the relevant options are available, the *support for interaction* is considered high.

The Dotted Chart does not require very much computing power, although the memory requirements are far above the expected requirements for such a visualization. This already proved to be a problem for the 'RWS' log, since the visualization failed due to an insufficient amount of available memory. Considering this issue, the *performance aspects* rating is low.

As we have already seen in the above description of the Dotted Chart, it simply visualizes all events in a visualization configuration. The visualization result is very predictable. Even after modifying the event log, we can make a very good estimate of how the visualization should look. Therefore *predictability of results* is rated high.

The evaluation of this visualization is summarized in Table 2.1.

| **Dotted Chart** | |
|---|---|
| *Visual aspects* | $\pm$ |
| *Support for interaction* | $+$ |
| *Performance aspects* | $-$ |
| *Predictability of results* | $+$ |

**Table 2.1:** Ratings of the Dotted Chart visualization.

## 2.3 Log Visualization

The Log Visualizer [14] is the visualization that will open up when you visualize an event log in ProM 6. The Log Visualizer is primarily used to give a first impression of an event log and as such shows some aggregated values such as the total number of traces in the log, the total number of events in the log and an average number of events per trace.

The plug-in is divided into three main sections. The first is the 'Dashboard', which can be seen in Figure 2.2, which shows aggregated information such as the information mentioned above. This is all process-level information that gives a first impression of the process as a whole: time span of the log, number of traces, average number of events per trace, etc.

The second section is the 'Inspector', which can be seen in Figure 2.4, which allows, as the name suggests, for inspection of the event log. It provides a user interface with which we can browse through all the data in the log. We can look at the traces and their attributes, and for each of the traces to their events and their attributes. This is especially useful since the XES data

format, as was already briefly mentioned in Section 1.1.4, is a very flexible data format that can store many types of data. Therefore, it is sometimes useful to look at the raw data, which is more convenient in this form than to look at the raw XML file. Of the 'Inspector' part, we only looked at the event log browser itself, we did not evaluate the 'Explorer' or the 'Log Attributes' parts. The 'Log Attributes' part only shows data specific to the event log file format and does not go into detail in either traces or events, hence it is not of any relevance to preprocessing.

The third section is the 'Summary', which seen in Figure 2.3. It shows all events, starting events and ending events with absolute and relative frequencies of occurrence for all classifiers specified in the log file. The log summary is therefore suitable for determining the start and end events and to get quick impression of what events are contained in the log and in which ratios.

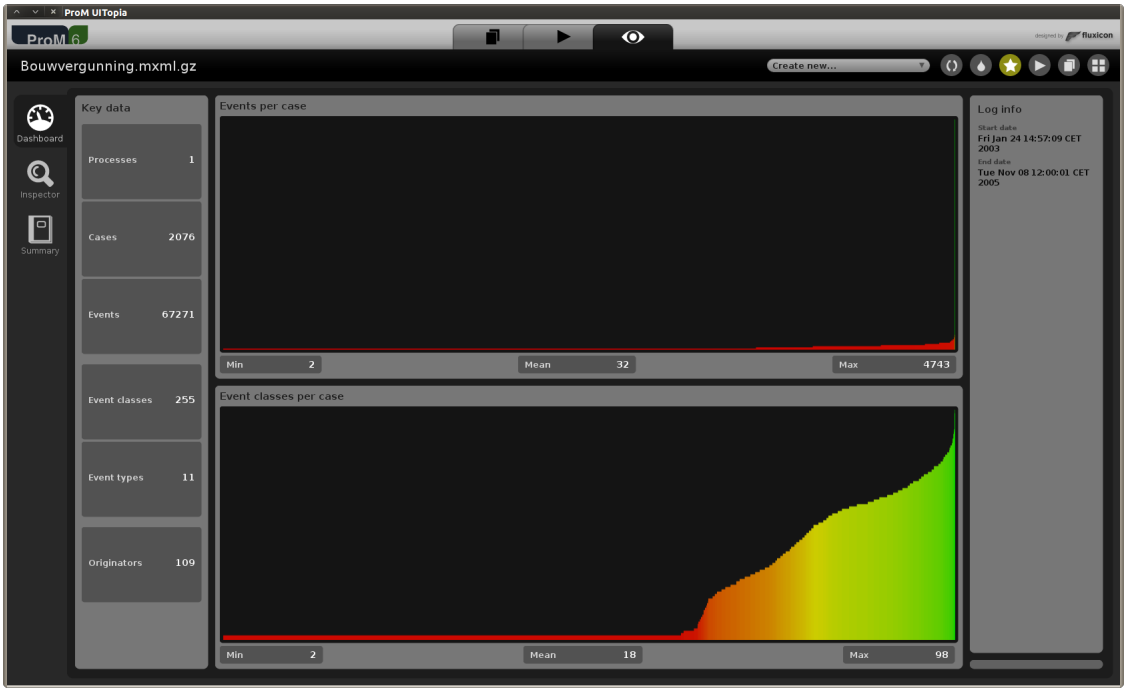


**Figure 2.2:** A screenshot of the Log Visualizer: Dashboard

### 2.3.1 Dashboard & Summary

The Log Visualizer, specifically 'Dashboard' and 'Summary' sections, focus on the process as a whole. Even though the 'Dashboard' shows some charts, the main form of information is textual. Both sections provide information from a general perspective and as such do not go into the details of any type of data or provide interpretations thereof.

The main advantage of the Log Visualizer's sections 'Dashboard' and 'Summary' is the fact that they create a nice overview of the process. One targets process-level information while the other targets the events. It is well suited for a first impression, but not for any in-depth analysis. The disadvantage is that the section 'Summary' is mostly textual representation, which is sometimes not as obvious as a visual representation. Considering this information, the *visual aspects* is rated medium. A lot of information is provided, but it is mostly textual and this can be better supported.

Both plug-ins offer the user only static information. There are no possibilities of interaction or reconfiguration. This is not necessarily bad, however, since there is no interaction, the *support for interaction* property is rated low.

Both plug-ins rely on event log data and basic aggregated data. To compute this data requires little computing power and little memory, hence it performs well with any kind of event log. The property *Performance aspects* therefore is rated high.

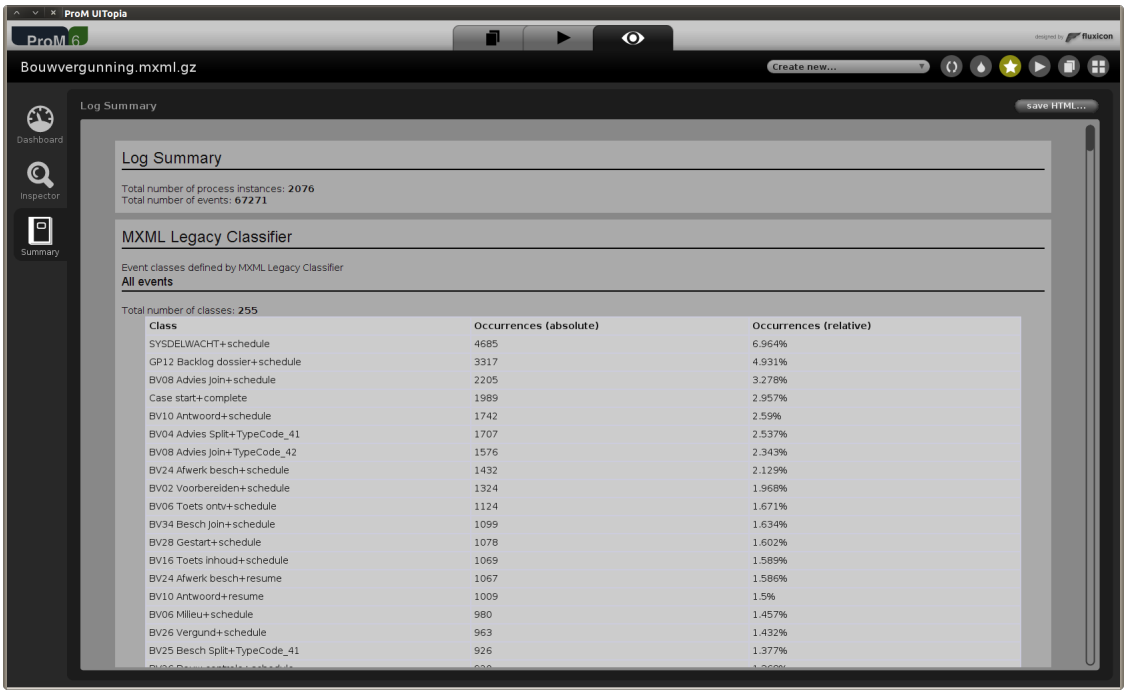


**Figure 2.3:** A screenshot of the Log Visualizer: Summary

These plug-ins only provide event log data and basic aggregated data, therefore they are very predictable. Thus the *predictability of results* is rated high.

The evaluation of the Log Visualizer: Dashboard is summarized in Table 2.2 and the Log Visualizer: Summary is summarized in Table 2.3.

| **Log Visualizer: Dashboard** | |
|---|---|
| *Visual aspects* | ± |
| *Support for interaction* | − |
| *Performance aspects* | + |
| *Predictability of results* | + |

**Table 2.2:** Ratings of the Log Visualizer: Dashboard.

| **Log Visualizer: Summary** | |
|---|---|
| *Visual aspects* | ± |
| *Support for interaction* | − |
| *Performance aspects* | + |
| *Predictability of results* | + |

**Table 2.3:** Ratings of the Log Visualizer: Summary.

### 2.3.2 Inspector

The remaining part of the Log Visualizer is the 'Inspector'. This part focuses specifically on traces and events. The 'Inspector' shows all data that is available, however it does not format or aggregate the data and it does not provide any interpretation either.

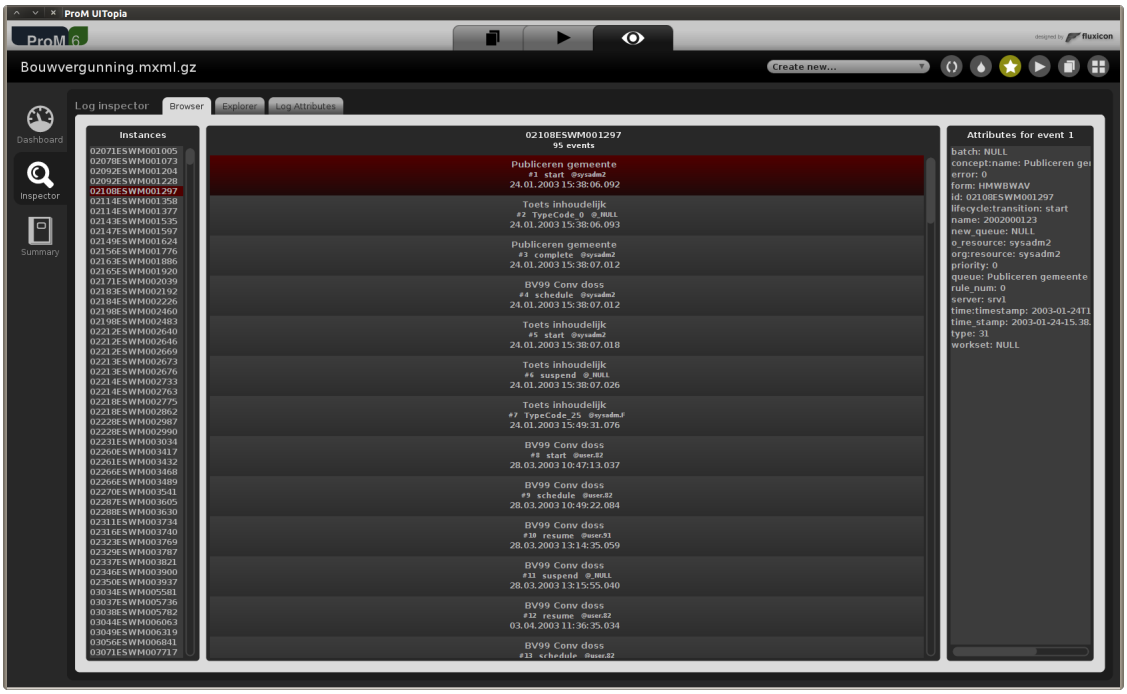


**Figure 2.4:** A screenshot of the Log Visualizer: Inspector

The 'Inspector' is the complete opposite of the 'Dashboard' and 'Summary' sections. It shows all the data in its raw form. There is no formatting or analysis and therefore offers no additional support in our goal towards analyzing and preprocessing the log. Note that currently, it is not possible to view nested attributes of traces and events, since only attributes that are children of a trace or event are visible. Since there are no visual aids, merely raw data output, *visual aspects* is rated low.

It does however offer the possibility to look up data on traces and events whenever we find ourselves with questions during the analysis. Therefore the *support for interaction* is rated medium.

The 'Inspector' part retrieves unmodified, unaggregated data on demand, which is a constant time operation. Therefore the performance of the visualization is very fast. *Performance aspects* is rated high. Similarly, the results are very predictable, hence *predictability of results* is rated high.

The evaluation of the Log Visualizer: Inspector is summarized in Table 2.4.

| **Log Visualizer: Inspector** | |
|---|---|
| *Visual aspects* | − |
| *Support for interaction* | ± |
| *Performance aspects* | + |
| *Predictability of results* | + |

**Table 2.4:** Ratings of the Log Visualizer: Inspector

## 2.4 Logical and Multi-set views

The logical (Figure 2.5) and multi-set (Figure 2.6) views [11] are used to visualize structure of cases. Timing information is not used, so the visualization cannot tell us anything about performance-related questions. However, it does show all the cases and all of the events a case consists of, and since timing information is not incorporated in the visualization, the emphasis is on the case

structure, instead of its execution (e.g. by visualizing cases that take a longer time bigger than cases that take a shorter time).

The Logical view shows subsequent events within a case in execution order. The Multi-set view takes the events out of execution order, and groups them by event classifier (in the reference case, this is the 'event class'). In case of the Multi-set view, the emphasis is on the number of executions of an event class.

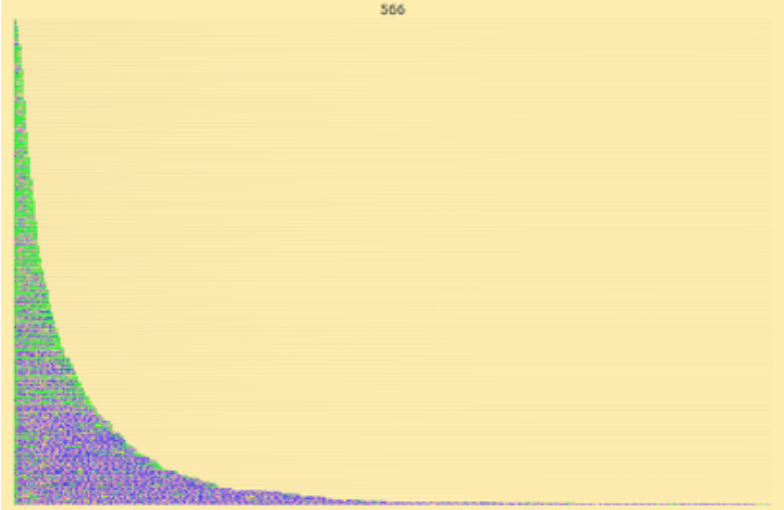


**Figure 2.5:** A picture of the Logical view. ("Sequential patterns of a treatment process", by P. Riemers in [11, p. 51])

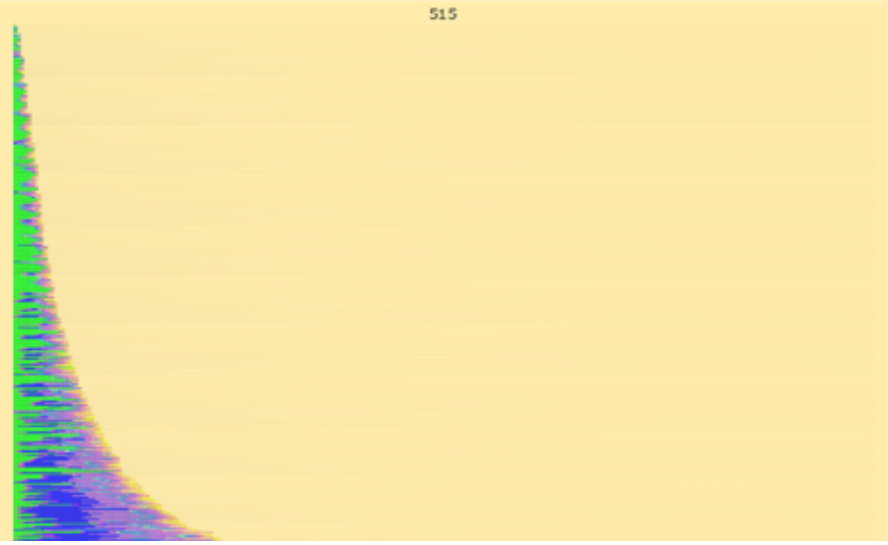


**Figure 2.6:** A picture of the Multi-set view. ("Multi-set patterns for a treatment process", by P. Riemers in [11, p. 52])

The Logical and Multi-set views focus on visualizing individual traces with their events. The visualization takes on a general perspective. There is no intelligent usage of process information, but instead visualizes the trace using a given classifier. In [11], the 'activity name' is used as the classifier.

A typical use case for these visualizations is the detection of patterns within and among the visualized traces. The Logical view can show traces with their events in execution order, allowing us to find traces similar in length and execution patterns. The Multi-set view can show traces with types of events and number of executions, enabling us to find traces that are similar in event classes and number of executions of events per class.

The advantages of the Logical and Multi-set views are that, first, they are in essence very simple views. Second, these visualizations do not depend on the precise semantics of the data (i.e. they do not require any understanding of process mining concepts in order to correctly visualize the information) and as such can visualize any available data type that either is a class or can be classified.

The disadvantage of the Logical and Multi-set views is that they merely visualize data in a certain way, all interpretation has to be done by the user.

All of the before mentioned advantages and disadvantages are related to the *visual aspects* of the visualization. These views are strongly focused on visualizing data, and as such are rated high for *visual aspects*.

The views are not available in ProM, hence we cannot evaluate the *support for interaction* and *performance aspects* properties.

The type of data that is visualized can be acquired in a straight forward way that is easily comprehended, hence its *predictability of results* property is rated high.

The evaluation of the Logical and Multi-Set visualizations are summarized in Table 2.5.

| **Logical and Multi-Set Views** | |
|---|---|
| *Visual aspects* | + |
| *Support for interaction* | |
| *Performance aspects* | |
| *Predictability of results* | + |

**Table 2.5:** Ratings of the Logical and Multi-Set Views.

Note: The Multi-Set view is not yet available as a plug-in in either ProM 5.2 or ProM 6. The Logical view that is similar to the notion as it is described here, is available in the Dotted Chart plug-in, when we select the 'Logical' option for the time axis.

## 2.5 Pattern Abstractions

The Pattern Abstractions plug-in [8,9] focuses on a very specific part of process mining: patterns. More precisely, loops and repeating patterns. The Pattern Abstractions plug-in, as can be seen in Figure 2.7, provides ways to search for these types of patterns. Once we have configured the plug-in and searched for the patterns, we are presented with the options for filtering the patterns based on frequency of occurrence, percentage of instances that match, etc. After that, we can also continue search for abstractions. Related patterns (alphabets) can be grouped together into higher-level activities using abstractions. Because of this, the abstraction can cope with slight variations within the various traces.

The Pattern Abstractions plug-in focuses primarily on the traces and events. It finds loop and repeat patterns in traces of the event log, which can then be used in log preprocessing. Currently the visualization is limited to finding patterns based on 'concept:name' + 'lifecycle:transition' data.

The advantages of the Pattern Abstractions plug-in are that it finds the kind of patterns that are hard to find. The Dotted Chart creates the kind of overview that allows us to see these patterns, but with the Dotted Chart this is like searching for a needle in a haystack and it would have to be found by manual inspection, while the Pattern Abstractions plug-in uncovers these

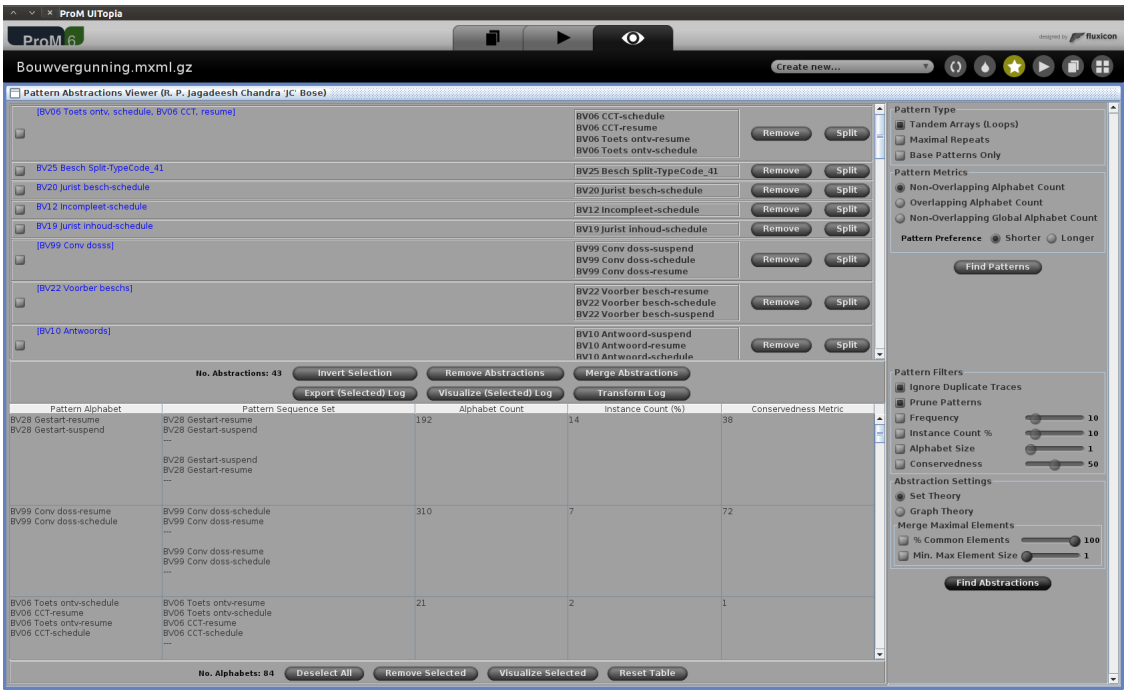


**Figure 2.7:** A screenshot of the Pattern Abstractions plug-in.

patterns automatically. The second advantage, which also has disadvantages, is that the very basic visualization of the pattern gives a good impression of the type of pattern and its extent.

The Pattern Abstractions plug-in has a number of disadvantages. First, there is no control over the event classifier that is used in the search. It may be interesting to search for patterns based on 'resource' or 'lifecycle' data. Second, the user interface is quite complex, using lots of technical terms, hence it is not very accessible to users that do not know the details of the underlying theoretical research. Third, the visualization of the patterns is created for the purpose of observing only the traces that match the pattern. You cannot, however, view the instances of the pattern within the complete event log. So, this visualization is useful to understand the pattern itself, but not to understand the pattern in relation to its context, i.e. the other, often similar, traces. Fourth, there is no interaction in the visualization.

As we have just seen, are the *visual aspects* of the plug-in very limited. This is not completely unexpected given the nature of this plug-in. The *support for interaction* is very strongly present given the multitude of configuration options and the available filter and merge options.

Another disadvantage of the Pattern Abstractions plug-in is the fact that it is computationally fairly heavy. The 'bouwvergunning' log, which has relatively many events, takes 424 seconds for the initial analysis operation. Another event log, which has 14279 traces and 119021 events, takes 178 seconds for the initial analysis. Subsequent analysis steps, there are three in total if you use the Pattern Abstractions completely, take significantly less time. Still, for the 'bouwvergunning' log it takes up to 14 seconds for the second step, and 27 seconds for the third step. Because of complex operations being performed, this plug-in is computationally very heavy, therefore we consider the *performance aspects* rating to be low.

With a sufficient amount of knowledge on the background of this work, the results may be fairly predictable, but for the typical user, this is certainly not the case, hence we consider the *predictability of results* rating to be low.

The evaluation of the Pattern Abstractions visualization is summarized in Table 2.6.

| Pattern Abstractions | |
|---|---|
| *Visual aspects* | − |
| *Support for interaction* | + |
| *Performance aspects* | − |
| *Predictability of results* | − |

**Table 2.6:** Ratings of the Pattern Abstractions visualization.

## 2.6 Preceding and succeeding event classes

A visualization that is proposed by P. Riemers in [11], Figure 2.8, is a visualization that shows a given event class with to the left all event classes that may directly precede the event class, and to the right all event classes that may directly succeed the event class.

To extend on this visualization: we can apply a number of different classifications to the events. P. Riemers looks specifically at the activities that have been executed. But the same approach can be taken to visualize, e.g., which resources precede and succeed a certain resource. In theory, any event classification can be applied, however in practice not every classification makes sense.


**Figure 2.8:** A picture of the visualization showing preceding and succeeding event classes. ("Centered activity with causal relations (1 step)", by P. Riemers in [11, p. 55])

This visualization focuses solely on individual events. The visualization takes on a general perspective. The typical application of this visualization is to find out about preceding and succeeding events, based on a chosen event class. This allows the user to look at a process and inspect an event class and its local environment.

To achieve this, the visualization uses data from events together with a chosen data type on which to classify the events. Since the visualization is general and does not provide any semantics, any data type is applicable. The primary application is to use the 'concept:name' information, i.e. the activity name, but it is just as well possible to use the name of the resource that executed the event as the data type.

The advantages of this visualization are the following. First, it gives a good local overview of the events (corresponding to the selected data type). Second, provided that the preceding and succeeding events are visualized in ratios of occurrence, it gives a good estimate of the chance a particular event class may have preceded or succeeded the selected event class.

Not strictly a disadvantage, but definitely worth mentioning is the fact that this visualization does not give an overview of the complete process, but only a local overview. Therefore, this visualization is not meant to provide a global overview, but to provide behavioral information around the specified event class.

All of the aspects discussed above are part of the *visual aspects* property. This visualization is centered around a graphical representation of data and does so well, hence it is rated high.

The visualization is not implemented in ProM, hence we cannot evaluate the *support for interaction* and *performance aspects* properties.

The data that are visualized are fairly straight forward. Even though you would not be able to directly read the information out of the event log itself, it is straight forward enough that *predictability of results* is rated high.

The evaluation of the Preceding and Succeeding Event Classes visualization is summarized in Table 2.7.

| **Preceding and succeeding event classes** | |
|---|---|
| *Visual aspects* | + |
| *Support for interaction* | |
| *Performance aspects* | |
| *Predictability of results* | + |

**Table 2.7:** Ratings of the Preceding and succeeding event classes visualization.

Note: This visualization is not yet available as a plug-in in either ProM 5.2 or ProM 6.

## 2.7 Trace Alignment

Trace Alignment [10], Figure 2.9, is a trace visualization plug-in. Trace Alignment visualizes traces in textual form, but does that in a specific way. It orders traces by similarity and then tries to align the events of each trace that either have equivalent event classes or are options of the same choice construct. By visualizing traces in this way, the emphasis is put on the deviations from the larger whole of the event log.

The Trace Alignment plug-in focuses specifically on traces and the event classes of the events. Trace Alignment currently uses 'concept:name' and 'lifecycle:transition' data to determine the event classes, hence it assumes the control-flow perspective.

A clear advantage of how Trace Alignment visualizes events, is the structured aligned grid approach that is taken. There are no graphical glitches or overlapping events, and because of the alignment the emphasis is put on dissimilar traces, instead of similar traces.

Disadvantages of Trace Alignment are, first, that event logs with a wide variety of traces can result in a big and sparse grid. The size and the sparseness of the grid makes it difficult to do the analysis. Clustering can help to reduce the problem significantly, but especially the sparseness and long traces can cause us to lose the overview over the traces we are analyzing or even the process as a whole.

Second, traces are automatically ordered by similarity to one another, but at the time of visualization, the events may not always be aligned in the most obvious way. Because of this, traces may appear to be more dissimilar than they actually are. We should manually check the alignment before definitively judging the similarity of the traces.

Given the pros and cons discussed, we will rate the *visual aspects* property at medium.

Next is the *support for interaction*. There are a number of advantages. First, the clustering options that are available, enables us to partition the traces in a number of clusters. Clusters are formed based on the similarity of the traces. We can therefore generate more clusters in the case where the log is very large or very diverse, and analyze each cluster of traces separately. Second, the

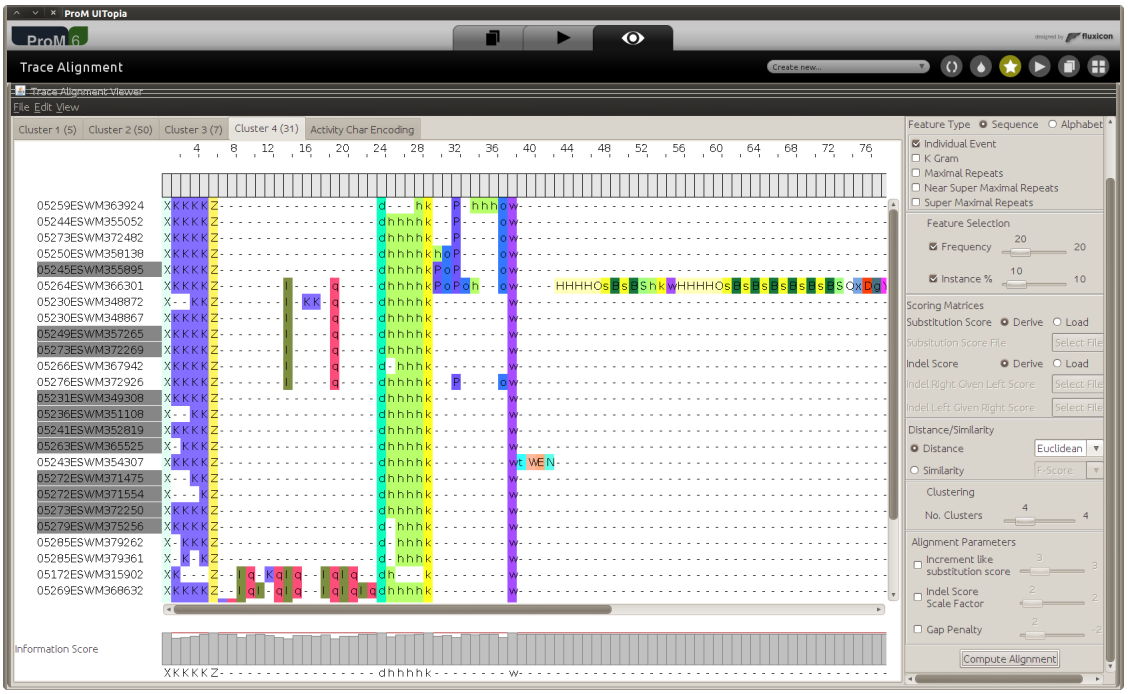


**Figure 2.9:** A screenshot of the Trace Alignment plug-in.

plug-in has its own filtering capabilities. And, third, the plug-in has many configuration options with which it is possible to configure every part of the analysis and subsequent visualization.

A disadvantage of the plug-in is that the plug-in is designed to use 'concept:name' and 'lifecycle:transition' data, and as a result takes the Control Flow perspective. Even though the trace alignment concept is generic enough to be applied to other data types (and hence perspectives) too. We could use, e.g., the 'organizational' data and visualize traces by resource, group or role. The second disadvantage is that there are many options and most have a complicated description. Even though it is a powerful visualization, the technically labeled options make it hard for the typical user to comprehend.

Considering the wealth of options of the Trace Alignment, and that the disadvantages are relatively minor and most do not even limit the plug-in itself, *support for interaction* is rated at high.

There is a significant disadvantage of Trace Alignment: the time it takes to analyze the log. The plug-in is computationally very heavy. It requires an expensive analysis step in which trace similarity is determined, among other things. The 'Bouwvergunning' log, which has a relatively large number of traces, took over 45 minutes to analyze and visualize. This is significantly more than the sub-minute times that most plug-ins require. Therefore, the *performance aspects* is rated at low.

The trace alignment grid is fairly straight forward. The order of traces and the alignment of events is less straight forward. Partly because of the analysis that is required before the visualization, and partly because of the alignment operation. It is hard to predict the first result, but, once a grid is known, it becomes easier to predict subsequent results. For example, predicting the changes in the grid after having executed a filtering operation are feasible if the filtering operation is not too significant. Therefore, we rate *predictability of results* at medium.

The evaluation of the Trace Alignment visualization is summarized in Table 2.8.

| Trace Alignment | |
|---|---|
| *Visual aspects* | ± |
| *Support for interaction* | + |
| *Performance aspects* | − |
| *Predictability of results* | ± |

**Table 2.8:** Ratings of the Trace Alignment visualization.

## 2.8 $\alpha$ Algorithm and the Petri Net Visualizer

A way of mining a process model is the $\alpha$ algorithm [1, 7]. The $\alpha$ algorithm produces a Petri net model as a result of mining the event log as can be seen in Figure 2.10. Because of the strict semantics of the Petri net, it is possible to clearly distinguish the nature of the event log from the resulting Petri net. We can use the produced Petri net as a source of information in which we can look for unexpected structures. Here we look at the combination of a process model mined with the $\alpha$ miner and the visualization of the Petri net model.

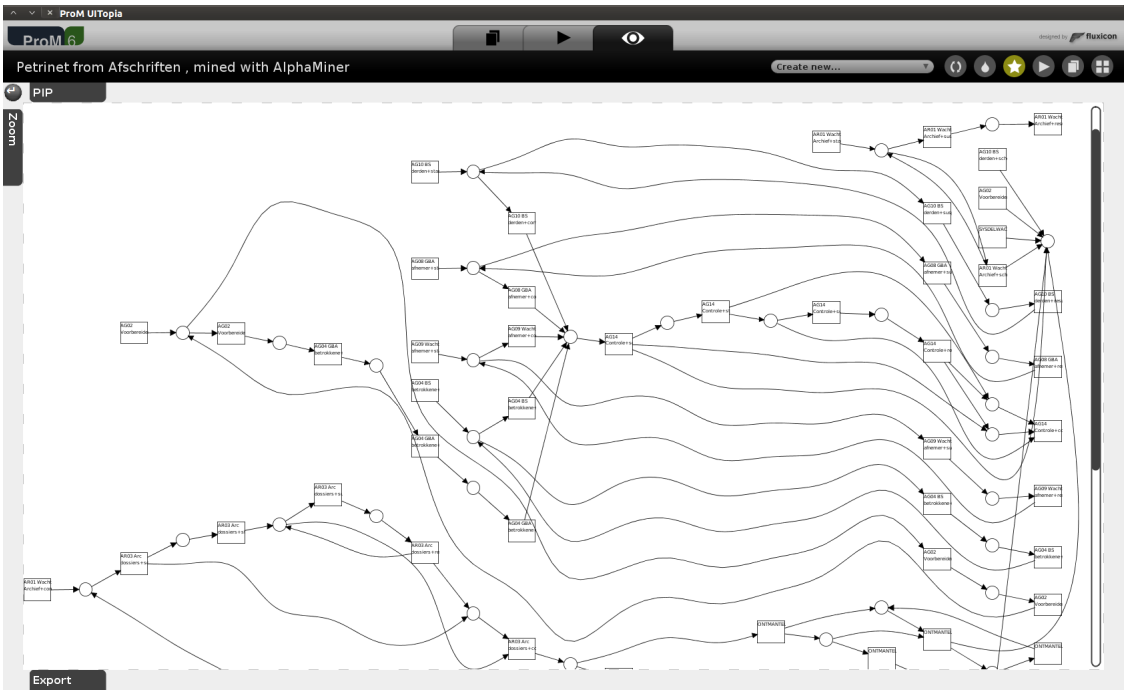


**Figure 2.10:** A screenshot of the Petri net Visualization plug-in

Both the $\alpha$ algorithm and the Petri net visualization focus on the process as a whole and a single perspective: the control-flow perspective. The visualization is minimalistic, it simply shows the process model, i.e. the mined Petri net model. The process model can be used to find out about the control flow of the process of which the event log contains the instances. Given the strict semantics of Petri nets, it is also possible to find anomalies in the process by way of analyzing the Petri net.

The advantage of the Petri net visualizer are that the visualization is very clean, no distractions or useless information is shown. The Petri net models have an unambiguous semantics, which makes it possible to evaluate the produced net in detail.

Now we look at the disadvantages of the Petri net visualizer. First, there is no additional information available, hence it is not easy to investigate any observed anomalies. The second

disadvantage is that the class of the activities often do not fit in the activity symbol and as a result the end of the class name is cut off. As a workaround for this problem, it is also possible to view the full class name in the tooltip whenever you point at the activity. The third disadvantage of Petri net models is that the strict semantics of the Petri net formalism enforce a high level of detail on the model. When attempting to discover the process model for larger processes, this can very easily result in a spaghetti model. These models are so complex and chaotic that it will be difficult to get any insights from looking at the model. Although, it is worth noting that, due to the nature of the $\alpha$ algorithm, it might help with an investigation into exceptional behavior.

Having discussed the pros and cons, we now decide on a rating for the *visual aspects* property for the Petri Net visualization. Even though the visualization itself is fairly clean, the disadvantages are significant enough. Therefore we rate the *visual aspects* at low.

There is little interactivity available. It is possible to move elements. It is also possible to select one or multiple elements of the Petri net model. However, this is only useful, if we want to move the selected elements. The second possibility is that by double-clicking on an element, we can rename an element. Both moving and renaming provide a way of modifying the model to make it slightly more readable, although it does not solve the problem of spaghetti model results. Given that the interactivity is very limited, the *support for interaction* is rated at low.

Another disadvantage is related to the performance. The time required to visualize large Petri nets will take up to minutes. In the case of the 'Bouwvergunning' event log it takes up to 105 seconds. For this time, the *performance aspects* property is rated low.

The $\alpha$ algorithm analyzes the event log in order to come up with a suitable Petri net. Even though the algorithm is not overly complicated, it is complicated enough to make it hard to predict results beforehand. Therefore, *predictability of results* is rated low.

The evaluation of the $\alpha$ Algorithm and Petri Net visualization is summarized in Table 2.9.

| **$\alpha$ Algorithm and the Petri Net visualization** | |
|---|---|
| *Visual aspects* | – |
| *Support for interaction* | – |
| *Performance aspects* | – |
| *Predictability of results* | – |

**Table 2.9:** Ratings of the $\alpha$ algorithm and Petri net visualization.

## 2.9 Heuristics Miner

The Heuristics Miner plug-in [15], Figure 2.11, is a control flow mining plug-in that mines the event log for a process model. The plug-in can cope with noise and low-frequent behavior in an event log. This is important, since real-life event logs tend to have noise. Because of this property of knowing how to deal with noise, the Heuristics Miner can be used for mining the process model from the event log in order to get a first impression of what the process (model) looks like.

The Heuristics Miner focuses on the process as a whole. The perspective of the Heuristics Miner is the Control Flow perspective. It uses the traces and the events within traces to generate a model that represents the control flow through the process.

There are several advantages to the Heuristics Miner. The first is that it can cope with noise and low-frequent behavior, as was already mentioned. The result of the Heuristics Miner is a process model that corresponds to the event log used as input. It can, e.g., be used to learn of the underlying process and to search for unexpected connections in the model which indicate exceptional behavior or, at the very least, previously unknown behavior. The third advantage is that it has a very clean interface, which enables us to focus on the control flow of the process model.

Disadvantages of the Heuristics Miner are the following. There is little information available in the visualization, so whenever we find interesting behavior, we will have to look this up using

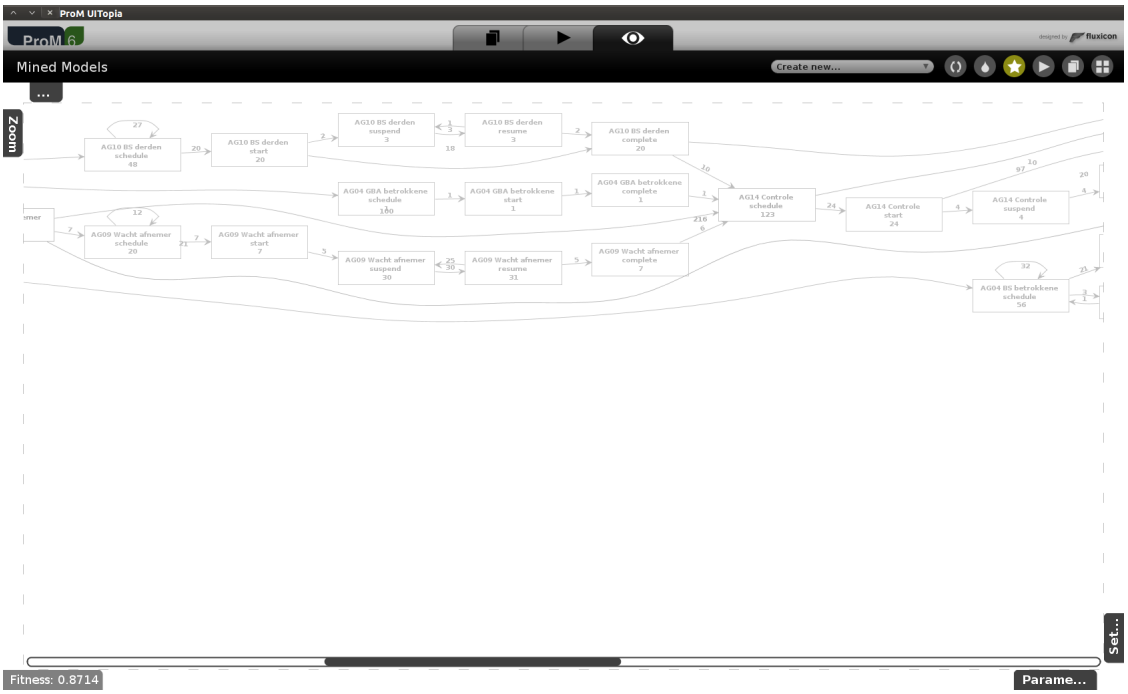


**Figure 2.11:** A screenshot of the Heuristics Miner plug-in.

another plug-in if we want to get the details on the behavior. This holds for both the basic facts of events, and the actual traces flowing through the particular section of the model. A second disadvantage is that the visualization is not too clear on the meaning of the numbers in the model. The last disadvantage is that there is no way to cluster activities, so whenever the process has a sufficiently large number of distinct event classes, the Heuristics Model visualization will become hard to use since it will produce results that are simply too large to be visualized properly.

All of these advantages and disadvantages are related to the *visual aspects* of the visualization. Given that there are still some serious disadvantages, we rate the *visual aspects* property at medium.

As with the $\alpha$ miner, there is support for moving and renaming elements of the process model. The Heuristics miner additionally allows for the reconfiguration of some visualization options. This is important, since it uses a color scale ranging from light gray to black to indicate the frequency of occurrence, and if a model has many different events, the model becomes very hard to read. There is no way to change the mining parameters. For this we need to start a new mining process. There is no way to request additional information on elements in the process model, hence we are only able to get control flow information from the process model. Considering these possibilities and lack thereof, we consider the *support for interaction* to be available but limited, hence it is rated at medium.

A clear advantage of the Heuristics Miner is that it is a fast mining plug-in. With this speed, it is possible to quickly mine a process model. Even the larger event logs that were used to test the mining plug-ins can be mined in a matter of seconds. For this, the *performance aspects* property is rated high.

The Heuristics miner builds up a process model from data all over the event log that is used as input. When we modify the event log only very slightly, the result might seem predictable. However, since the the Heuristics miner also makes use of heuristics for coping with noise, and when an event log is large enough, the amount of information becomes too large to handle ourselves. Therefore we consider the *predictability of results* to be low.

The evaluation of the Heuristics Miner visualization is summarized in Table 2.10.

| Heuristics Miner | |
|---|---|
| *Visual aspects* | ± |
| *Support for interaction* | ± |
| *Performance aspects* | + |
| *Predictability of results* | − |

**Table 2.10:** Ratings of the Heuristics Miner visualization.

## 2.10 Fuzzy Miner

The Fuzzy Miner is a plug-in [6] that mines process models from event logs. Figure 2.12 shows a screenshot of a resulting Fuzzy Model. The mining algorithm of the Fuzzy Miner is most related to the Heuristics Miner. The Fuzzy Miner uses abstraction, aggregation and emphasis to generate a process model that is better understandable and, because of smart abstraction and aggregation approaches, smaller and thus better visualizable. It is also customizable: we can adjust the settings for before mentioned properties in order to fit the model to our own requirements. To get a clear picture of what is available for use in preprocessing a log, we first look at the visualizations that are available. The focus here is on the actual *visualization* of the information, not the algorithm that prepares the data for visualization.

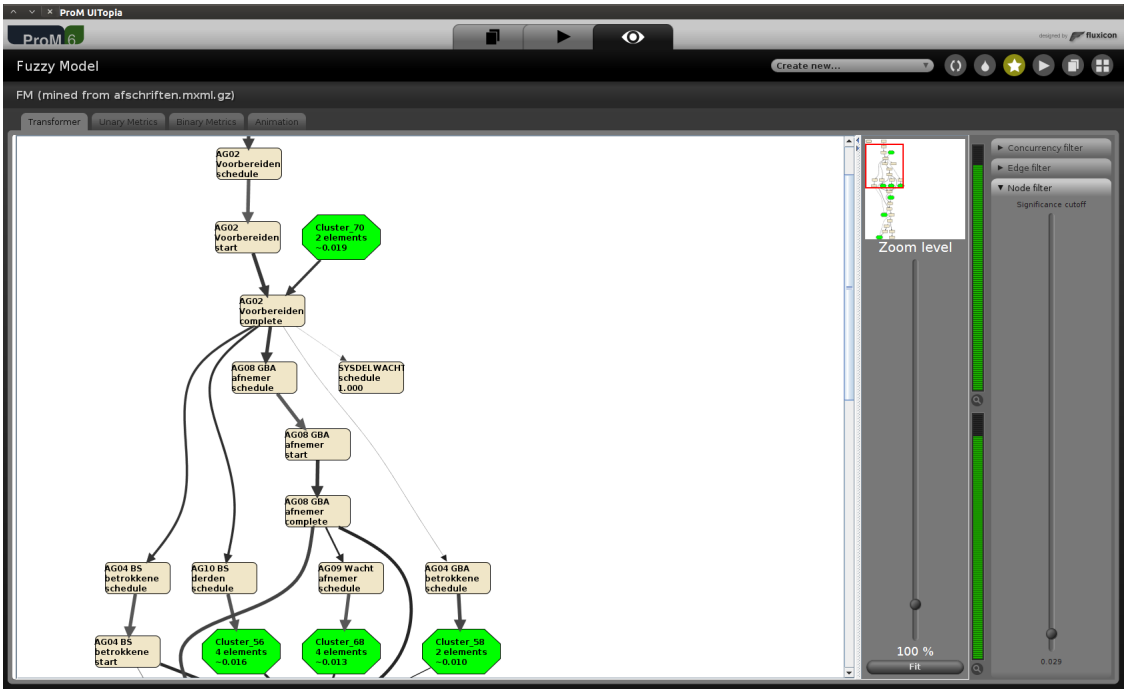


**Figure 2.12:** A screenshot of the Fuzzy Miner plug-in.

The Fuzzy Miner is a process model miner. It focuses primarily on traces and events. Trace information is used to generate a process model and events are used for additional information used for the abstraction and aggregation features of the Fuzzy Miner. The perspective for the Fuzzy Miner is mainly a Control Flow perspective. The accompanying Fuzzy Model Animation provides a way of animating the model in order to give insights into the performance of the process. Although it is worth saying that the Animation only offers a view of the behavior over time. There is no further performance data available.

The advantages of the Fuzzy Model are the support for clustering, which enables users to create a comprehensible overview by clustering events in the cases where the unclustered model is too large to comprehend. And that size and color of arcs provides a notion of importance. The Fuzzy Model Animation shows clearly how the process behaves over time, specifically the workload and the performance by ways of the animation of cases through the process model.

There are two disadvantages. First is the Fuzzy semantics. As we have seen earlier, the Fuzzy Model uses abstractions and other techniques to simplify the model. The result of these techniques is a simpler model, but it also makes the semantics surrounding the nodes and relations more vague. With these semantics, we cannot rely on the process model being absolutely precise, instead it should be used to give a good indication of the process. The second is that there is no additional information available on other data, like resources or performance. This makes it only usable to get a good impression of the process, but does not allow for an investigation that uses more than the control flow data.

As with the other visualizations, the *visual aspects* do not offer more than the typical control flow data. Unlike the other visualizations, the Fuzzy Miner actually copes very nicely with large event logs by offering the clustering options to the user, although these techniques are both a blessing and a curse, given the fuzzy semantics that are a result of those techniques. The animation is a nice addition that clearly shows how the process behaves over time, although it does not offer any additional data. Overall, the *visual aspects* are okay and as such are rated at medium.

As we have already discussed, there are a number of options with which the Fuzzy miner can be configured. These options can interactively control the aggregation and abstraction that the Fuzzy miner uses for its mining operations, as well as being able to inspect a cluster node. Given the control that the user is offered, the *support for interaction* is rated high.

Looking at the performance of the Fuzzy miner, we can see that it is certainly not disappointing. All times are below 30 seconds, even for the larger event logs. When modifying the mining options interactively, it still takes only seconds to adjust the process model. The Fuzzy Model Animation, on the other hand, is more demanding. The animation takes almost 2 minutes to prepare the event log with a large number of events. Taking into consideration both parts of the visualization, we have to rate the *performance aspects* at medium.

Due to the methods used in the Fuzzy miner, such as abstraction and aggregation, the Fuzzy miner becomes a hard-to-predict mining algorithm. The additional tabs in the visualization, 'Unary Metrics' and 'Binary metrics', give the user an insight into the metrics used by the algorithm. However, this is not enough to make the algorithm predictable for the typical user. Therefore *predictability of results* is rated low.

The evaluation of the Fuzzy Miner visualization is summarized in Table 2.11.

| **Fuzzy Miner** | |
|---|---|
| *Visual aspects* | $\pm$ |
| *Support for interaction* | $+$ |
| *Performance aspects* | $\pm$ |
| *Predictability of results* | $-$ |

**Table 2.11:** Ratings of the Fuzzy Miner visualization.

## 2.11 Basic Performance Analysis

The Basic Performance Analysis plug-in is used to do performance analysis on an event log. It shows the key performance indicators such as processing time (task-level) and throughput time (process instance-level) and we can choose from several dimensions: instance, task, resource. Performance information can be visualized in both textual form and graphical form, see Figure 2.13 for an example of the graphical visualization.

The Basic Performance Analysis focuses on the process and individual traces, depending

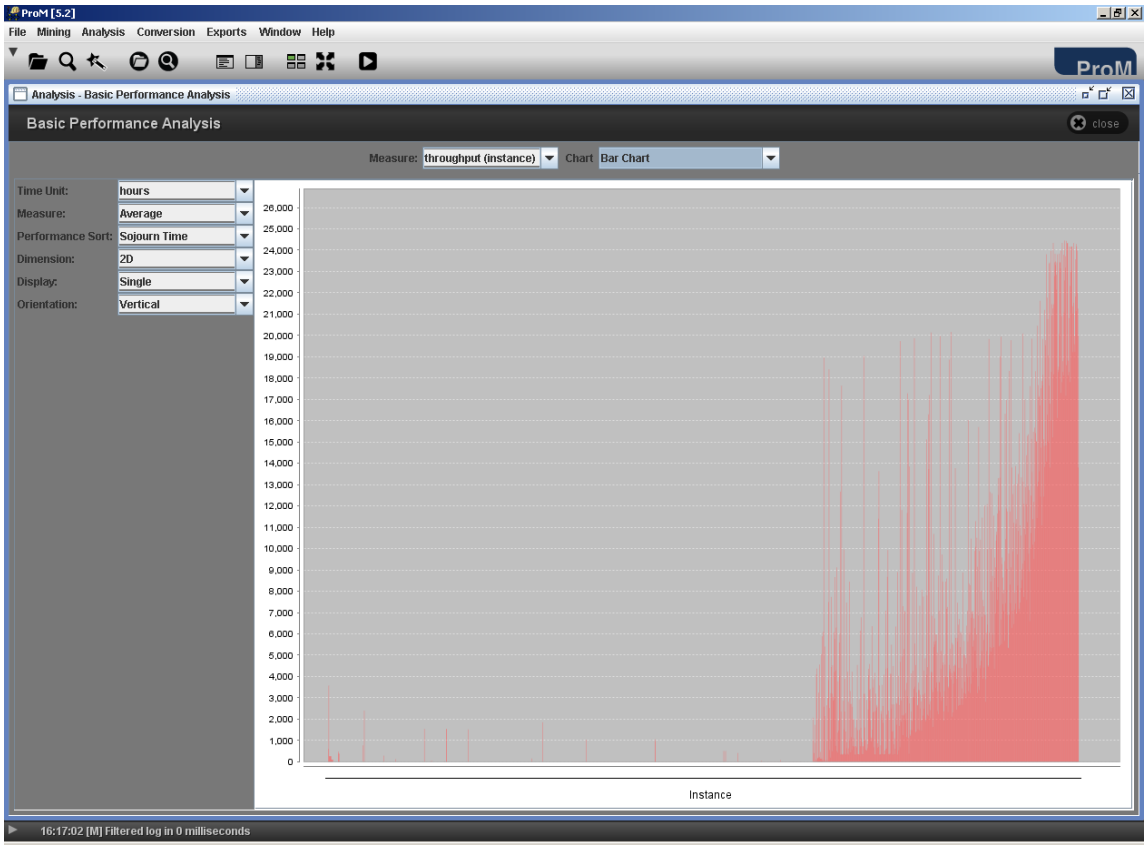


**Figure 2.13:** A screenshot of the Basic Performance Analysis plug-in.

on whether we select 'task' or 'instance'. The perspective for this plug-in is the Performance-perspective. The typical use case for the Basic Performance Analysis is to analyze the log in order to discover the performance information for the log.

In its analysis, the Basic Performance Analysis plug-in uses several types of data. It uses trace-level and event-level data on activities, resources and timing.

The visualization of the Basic Performance Analysis can be modified to visualize the data in a number of different ways, such as bar charts, pie charts, etc. And there is a choice of what data type is used for aggregated values, such as average, maximum, minimum, etc. Apart from the graphical representations, there is also an option to show the data in textual format. In this case, we are presented with a table in which all the information is shown.

Disadvantages of the plug-in are that it only offers the default attributes (instance, task and originator) for visualization. And a typical problem: labels are cut off when they become too long. What remains is often ambiguous. Especially with the instance visualizations, elements on the X-axis so there is not even a single character readable, so its close to impossible to read off anything. The most "accessible" information is the height of the bar, i.e. in case of a bar graph. (Note: When we hover the mouse pointer over the bar itself, it gives a tool tip with the label, so it is still possible to find out what a blank bar represents.)

All of the advantages and disadvantages that have just been discussed are part of the *visual aspects* property. Although the Basic Performance Analysis is a very basic visualization, it does offer an insight into the activities and instances. Given the limitations due to the partial or completely missing labels, the *visual aspects* property is rated medium.

At initialization time of this visualization, there are options that help configure the plug-in. Among them are options such as considering of working hours, weekends and holidays, and which dimensions of data to use. (e.g. activities, resources and cases.) There are also a decent amount of

options available at visualization time, these are all to configure how and what data to visualize. And, although limited, there are zooming capabilities.

All in all, the Basic Performance Analysis gives us a good amount of options that allows us to configure the visualization as we wish. The main limitations are in the area of available data types. Though we have to keep in mind that this is a ProM 5.2 plug-in and ProM 5.2's MXML event log format is not as extensible as the new XES event log format. Considering everything, and keeping in mind that the support for further data types could be easily added, the *support for interaction* is rated high.

Next up is the performance of the plug-in. There is a analysis stage before the visualization is available. In the analysis stage, the required data is gathered. During the visualization, information is shown fairly quickly. Switching e.g. dimensions is not instantaneous, but fast enough for interactive use. The *performance aspects* property is rated medium.

The nature of the information that is shown is basically aggregated data on a number of dimensions. Therefore it is easy to predict what the visualization would present for small changes but even for large changes. The *predictability of results* for this visualization is high.

The evaluation of the Basic Performance Analysis visualization is summarized in Table 2.12.

| **Basic Performance Analysis** | |
|---|---|
| *Visual aspects* | ± |
| *Support for interaction* | + |
| *Performance aspects* | ± |
| *Predictability of results* | + |

**Table 2.12:** Ratings of the Basic Performance Analysis visualization.

Note: The Basic Performance Analysis plug-in is not yet available in ProM 6. This analysis has been done using the Basic Performance Analysis plug-in that is available in ProM 5.2.

## 2.12 Role Hierarchy Miner

The Role Hierarchy Miner, Figure 2.14, is used to discover role hierarchies using resource information that is present in an event log. As an event log typically only records the resource that executed the activity, but nothing about other potential candidates or whether or not work is divided over available resources, this information has to be mined.

The hierarchy is mined using resource information. Resources that perform the same set of activities are grouped. Groups that execute a subset of activities of what another group executes, are connected with an edge to indicate that one groups executes all the activities from another group and more, thus resulting in a hierarchy.

The result is a hierarchy with generalists (i.e. resources who can do almost anything) at the bottom, and specialists (i.e. resources that have a very specific, often small, set of activities) at the top. In addition to the graphical visualization, an 'Originator x Task' matrix is provided. It shows how many times a resource has performed a task.

The Role Hierarchy Miner focuses primarily on the process as a whole, looking at individual events for resource data and visualizing the information from a Resource perspective. The typical use case for the Role Hierarchy Miner is to discover and group resources by the similarity of their work.

An advantage of the Role Hierarchy Miner is the accompanying 'Originator x Task' Matrix. While exploring through the graph that represents the role hierarchy, we can get additional information via the 'Originator x Task' Matrix. There are no serious disdavantages to the Role Hierarchy Miner visualization except for the typical issue with large event logs. When analyzing large event logs, the screen space usage may be insufficient. This also depends on the number of distinct resources and activities, how similar the resources are to one another and how the

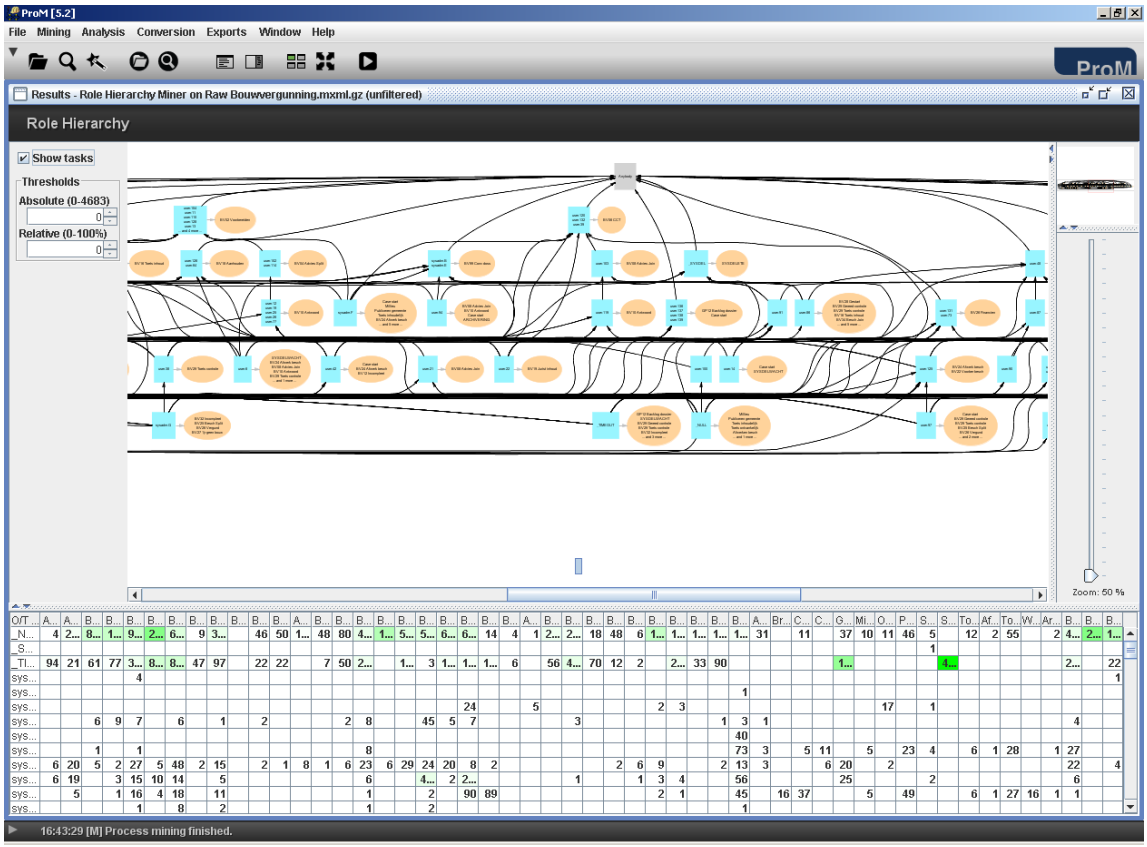


**Figure 2.14:** A screenshot of the Role Hierarchy Miner

threshold levels are configured. In less than ideal cases, we may lose the overview of the graph due to either the amount of distinct nodes or to the number of relations. Whenever there are enough distinct resources and activities, the matrix becomes large enough that the cell labels are not readable anymore. (On a side note: the tooltip information that is provided in the hierarchy graph is useless, since it displays a "node number" instead of the data of the node that we are pointing at.)

The disadvantages for the visualization are mostly due to the size of event logs. The accompanying 'Originator x Task' matrix greatly supports the visualization by showing those pieces of data that cannot be put into the hierarchy visualization itself due to the size of the data. Considering the advantages and disadvantages, we rate the *visual aspects* medium.

The Role Hierarchy Miner has good support for interactivity. By clicking on a node, the matrix changes to provide information on the selected node. There is an option to hide the tasks if desired. And there is a 'threshold' setting (both absolute and relative) that enables us to configure how similar resources must be in order to be grouped. These simple options make the visualization that much more usable, therefore the *support for interaction* is rated high.

The analysis of the event log beforehand and drawing the hierarchy, takes only a number seconds. The adjustments to the visualization that are required when using one of the interactive options also finish within a number of seconds. The performance is not instantaneous, but acceptable for interactive use. The *performance aspects* are rated medium.

The Role Hierarchy miner only considers resources and activities. The concept of the Role Hierarchy miner is easy enough to comprehend, and when the modifications to the event log are small enough, the results of changing the event logs are easily predictable. For larger event log modifications, this might not be the case, since the ratios between resources may change, and the threshold that is used may contribute to making it less predictable. Considering these points, we

rate the *predictability of results* medium.

The evaluation of the Role Hierarchy Miner visualization is summarized in Table 2.13.

| **Role Hierarchy Miner** | |
|---|---|
| *Visual aspects* | ± |
| *Support for interaction* | + |
| *Performance aspects* | ± |
| *Predictability of results* | ± |

**Table 2.13:** Ratings of the Role Hierarchy Miner visualization.

Note: The Role Hierarchy Miner is not yet available in ProM 6. This analysis has been done using the Role Hierarchy Miner that is available in ProM 5.2.

## 2.13 Social Network Miner

Another perspective on event log analysis is social network analysis [2, 13]. The social network miner, Figure 2.15, mines information on resources based on the provided event log. It looks at, among others, the 'handover of work' from one resource to another. (i.e. the way work is handed over from an employee that executed one activity to the employee executing the next activity.)

The Social Network Miner provides 5 types of metrics, which are based on the following concepts:

1. **Handover of work**: Whenever a resource $a$ executes an activity of a case and resource $b$ executes the next activity of the same case, then this suggests that resource $a$ handed over his work on the case to resource $b$. However, this does not hold for activities that are executed in parallel.

2. **Subcontracting**: Whenever a resource $a$ executes 2 activities of a case, and in between these 2 activities, an activity (for the same case) gets executed by resource $b$, then this suggests that resource $a$ subcontracted some of the work to resource $b$.

3. **Reassignment**: Whenever a resource $a$ explicitly (i.e. using the activity transition 'reassign') reassigns an activity to resource $b$. This possibly indicates a delegation of work and thus a hierarchical relation between resources $a$ and $b$.

4. **Similar tasks**: Two resources have a stronger relation when they perform a similar set of activities.

5. **Working together**: Two resources have a stronger relation when they often work on the same case.

These relations between resources reveal how resources work together on a case. Additional metrics, e.g. *Betweenness*, *Centrality*, *In-degree*, and *Out-degree*, further inform us about the way a resource is positioned within the process.

The Social Network Miner looks primarily at the process as a whole with a 'Resource' perspective. The plug-in can be used to find out how resources (e.g. employees) are positioned within the process and how resources cooperate with other resources, other departments, etc. For this purpose, only the 'resource' data is used.

We will now look at the disadvantages of the social network miner. First, there is no information available other than that of resources. Second, the layout is random (and continuously updated). Third, many resources make the visualization hard to read. Fourth, with the increasing number of resources you quickly loose the overview. And lastly, for large event logs the number of relations between nodes in the social network becomes so large, that they seem to lose their usefulness.

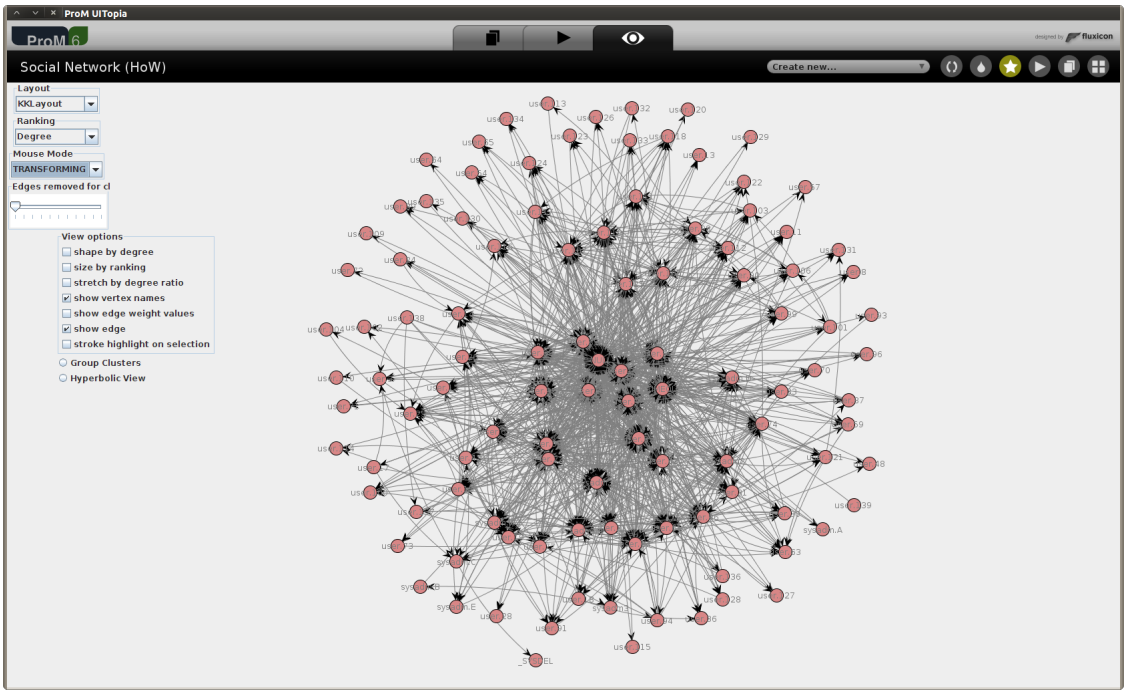


**Figure 2.15:** A screenshot of the Social Network Miner.

The Social Network Miner can, in good circumstances, give a good impression of the organization of resources, however there are quite a few disadvantages. Given these disadvantages, we rate the *visual aspects* at low.

There are a number of options available during the visualization of the social network. The most prominent are the selection of a layout strategy and the method of ranking the nodes in the social network graph. Selection and highlighting options greatly support the social network graph that otherwise becomes very hard to read for large event logs.

Due to the significance of the interactive options in enhancing the visualization, the *support for interaction* is rated high.

Next, we look at the performance aspects of the visualization. Analysis of the event log is done within a number of seconds. For large event logs, this is finished within 10 seconds. The visualization itself is also initialized within less than 10 seconds, with the exception of the 'Similar tasks' metric. (This is presumably due to comparing tasks.) The performance during visualization is not great. The continuously updating layout of the nodes in the graph cripples the responsiveness of the visualization, i.e. the graph, user interface controls and even the ProM UI controls feel sluggish. Updating the visualization upon changing the ranking or layout options, happens quickly.

Overall, the performance seems to be good, but the continuously updating layout significantly decreases the responsiveness. Therefore, the *performance aspects* are rated low.

Next property is the *predictability of results*. The Social Network Miner mines the resource data in order to determine metrics for each of the resources. The metrics are fairly easy to comprehend, so once the initial result is known, modifications to the event log should result in predictable changes to the initial model. Although prediction of the first result is typically hard. The random and continuously updating layout, on the other hand, cripples the predictability and for that matter also the ease with which we can reproduce the exact results. Hence the *predictability of results* is rated medium.

The evaluation of the Social Network Miner visualization is summarized in Table 2.14.

| **Social Network Miner** | |
|---|---|
| *Visual aspects* | − |
| *Support for interaction* | + |
| *Performance aspects* | − |
| *Predictability of results* | ± |

**Table 2.14:** Ratings of the Social Network Miner visualization.

## 2.14 Conclusions

In the previous sections, we looked at some of the available plug-ins in ProM 6 and ProM 5.2, and we looked at some visualizations from other sources that might be interesting to use. Now we summarize the evaluations in a single table, Table 2.15. We look at common preprocessing issues with the visualizations.

| | Dotted Chart | Log Visualizer | Logical & Multi-Set Views | Pattern Abstractions | Preceding & Succeeding Events | Trace Alignment | Petri Net Visualizer | Heuristics Miner | Fuzzy Miner | Basic Performance Analysis | Role Hierarchy | Social Network Miner |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Visual aspects* | ± | − | + | − | + | ± | − | ± | ± | ± | ± | − |
| *Support for interaction* | + | − | | + | | + | − | ± | + | + | + | + |
| *Performance aspects* | − | + | | − | | − | − | + | ± | ± | ± | − |
| *Predictability of results* | + | + | + | − | + | ± | − | − | − | + | ± | ± |

**Table 2.15:** Ratings of all visualizations together in a single table.

Looking at these results, we can find a few interesting and sometimes expected patterns. For instance, the most predictable results come from the most basic of visualizations, while the least predictable results come from the most complex visualizations. Complex in this case means either a plug-in that performs a complex analysis beforehand, or a plug-in that transforms the results to a format that cannot be mapped 1:1 with the source elements from the event log, i.e. events or traces. Another of these observations is that almost every visualization has some degree of interaction.

Apart from the observations we just mentioned, there are a number of larger issues that become apparent during the analyses. These issues surface when we carefully look at the intended use of these visualizations and try to use them to their maximum potential.

1. **Plug-ins are geared towards their primary concern**: Plug-ins tend to only support the primary concern. Process model visualizations show the position of the activity within the process model, however there is no information on, e.g., which traces flow through that part of the model. Another example is the Pattern Abstractions plug-in. It allows you to find repetition patterns, however once these patterns are found, there is only a very limited visualization of these patterns. More analytical plug-ins tend to have a rather poor

visualization and interactivity is geared towards fine-tuning the analysis. Process model discovery plug-ins, however, tend to only provide information visually. Their interaction options are geared towards making slight modifications to the process model.

2. **The provided secondary information is very limited**: An issues common to almost all visualizations, is the limited availability of secondary information. Visualizations easily provide the information directly relevant to their own target perspective. The other perspectives, on the other hand, are forgotten. For instance, a process model miner typically shows the activity names, transitions and their relations. However, no information is provided on the number of occurrences, resources or execution times.

3. **Filtering features are dispersed over a number of plug-ins**: A number of visualizations use preprocessing features internally. A clear example of this is the Trace Alignment algorithm, which does selection based on frequency of occurrence. Another example is the Fuzzy Miner plug-in that uses unary and binary metrics to determine which events to abstract from as a way to cope with noise. These features, although they are clearly a form of preprocessing, are only available to that plug-in.

4. **There are (almost) no plug-ins that provide basic data visualization features**: Specific to the exploratory goal of process analysis, is the ability to view the complete event log in a certain way. Almost all visualizations have very specific information to visualize, such as process models, discovered patterns, or networks. However, we are missing the functionality to simply visualize the raw data in a certain way, such as a bar chart, partitioned using the specified classifier. Or in case of textual presentation, a matrix of data, using user-specified classifications on the X and Y axes.

5. **The use of static event classifier**: ProM 6 and the XES file format support the use of user-defined event classifiers. These classifiers define the way a plug-in should classify events. However, instead of making use of these classifiers, many plug-ins use a hard-coded (and thus unchangeable) event classifier. As a result, the user is limited in its ways of applying the plug-in.

# Chapter 3

# Requirements

In the previous chapter we looked at a number of visualizations that can be used in preprocessing. We looked at four properties as a way to characterize the visualizations and have drawn our conclusions. We identified a number of issues with regard to the usage of these visualizations.

In this chapter, we look at ProM and identify a number of key points in ProM with regard to preprocessing. Next, we look back at the conclusions drawn in Section 2.14 and the issues identified in Section 3.1. We derive a set of requirements for the preprocessing framework, a set of guidelines for implementing interactive visualizations and finally clarify some practical decisions for the project.

## 3.1 Preprocessing in ProM 6

In Chapter 1, we looked at the method for process analysis and preprocessing in literature. In Chapter 2, we looked at visualizations that are available to us in ProM, what information they offer and how they support preprocessing. As a final evaluation step, we inspect the behavior and facilities of ProM 6, the most recent version of the process mining framework. We compare the facilities offered by the ProM framework with the issues identified in the literature and the visualizations that we analyzed before. By performing this comparison, we can determine whether the cause of an issue is rooted in a visualization or the underlying framework. From the results of this comparison we can determine the requirements of the project.

### 3.1.1 Event Log Filtering

Apart from the visualizations that we use during our preprocessing phase, we also require tools with which we can modify the event log once we have found the problems. A notable change in ProM 6, is the way plug-ins are regarded. In ProM 5.2 there were four types of plug-ins: import/export, filtering, analysis, and mining. In ProM 6, the distinction between filtering, analysis and mining has disappeared. As a result of that change, filters are now in the same list as analysis plug-ins.

As new research is done continuously, a lot of new plug-ins are available in ProM 6. However most plug-ins are created for the purpose of analyzing or visualizing data. As we have seen in Section 2.14 issue 3, filters have mostly been implemented ad-hoc in plug-ins that needed them. The most frequently used filters have been ported or reimplemented, but there are significantly less filters available than there were in ProM 5.2.

Thus it seems that there are a few issues at play here. The first is that there are simply less implementations of filters available than there were in ProM 5.2. The second is that there are no separate facilities for filters, which causes a loss of overview of the available filter plug-ins.

### 3.1.2 Iterative preprocessing

Previously, we talked about ProM 5.2 recognizing four types of plug-ins, three of which are now considered equal in ProM 6, i.e. filters, analysis and mining plug-ins. ProM 6 makes the distinction between plug-ins in another way. It recognizes analysis plug-ins and visualization plug-ins. The first type performs the analysis which typically contains the smart algorithms, while the second type visualizes an existing data object. A data object can be imported, such as an event log, or it is a result of an execution of an analysis plug-in. A typical plug-in execution looks like the following, with a graphical depiction in Figure 3.1. First, we select the plug-in to execute and the input data that is required for the execution of the plug-in. Next, we execute the plug-in. Some plug-ins first display a configuration dialog, but this is optional. When the plug-in has finished executing, the result is returned to the framework. The framework then automatically executes the preferred visualization plug-in for the returned result.

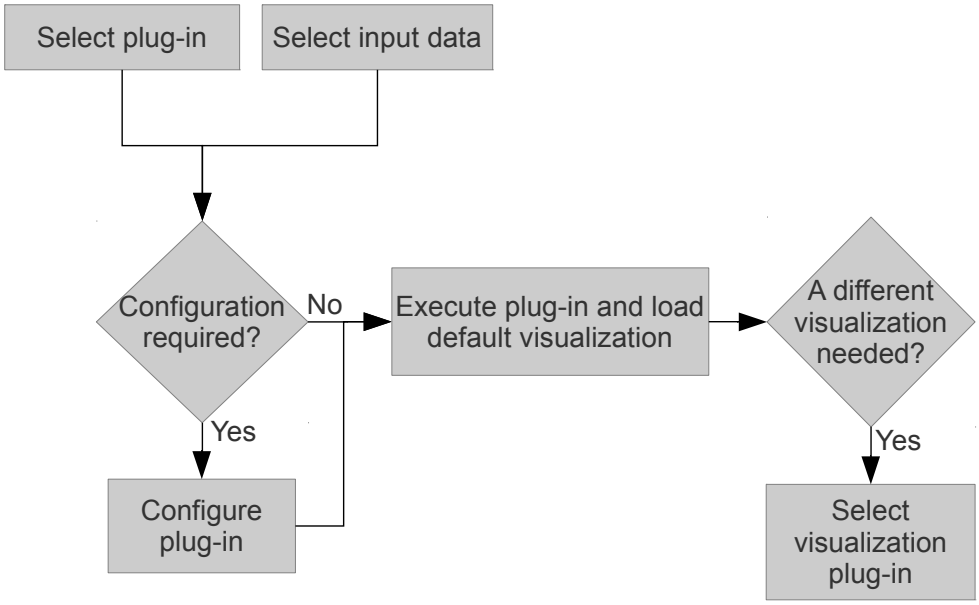


**Figure 3.1:** A graphical depiction of the steps that are required to execute a plug-in and to visualize the result.

As we can see, this is a fairly rigid process. Especially when running multiple plug-ins on the same input data. There is an exception to this process. When we only need to switch to a different visualization of the same data object and no analysis is required, then this can be done by simply selecting the desired visualization. However, this is an exception case that typically does not apply. Now, consider a situation where we need to execute a number of filters, check the results of each step in one or more visualizations, and finally run a process discovery plug-in and visualize its result. This will take quite a number of manual actions. And as such it does not match with the iterative nature of preprocessing, just as the theoretical approach that was explained in Section 1.2. Furthermore, even though the intermediate results are stored and thus available in the framework, switching to a previous result is not so easy. Similar data objects, such as differently filtered versions of the same event log or results of an analysis step, are typically named very similarly.

The most prominent issue here, is the rigid execution process. For preprocessing flexibility is required. With the loss of the filtering framework, filters have to be applied one by one, hence they are forced to the rigid plug-in execution process. Furthermore, finding the filters is more cumbersome, since they are mixed with other plug-ins in one big list. This is all completely contradictory to the iterative approach to preprocessing.

### 3.1.3 Limitations of visualization plug-ins

Issue 1 of Section 2.14 explains how the information that is of main concern to the plug-in is typically very well supported and represented. However, issue 2 emphasizes the missing information. Specifically for preprocessing, it is important to have all information readily available, as we need a certain level of understanding of the process in order to make the right decisions on how to proceed with preprocessing. The many available plug-ins provide various types of information. By using several plug-ins, we are able to retrieve much of the information that we require. The visualizations themselves, however, are limited in this matter, since they cannot communicate information outside of the visualization. In ProM, after an analysis step, plug-ins are able to return the results of analysis to ProM for storing and to use in other analysis or visualization plug-ins. However, there is no similar facility for storing and reusing the findings we uncover in visualizations.

The limitation of communicating visualization findings is rooted in the ProM framework. ProM can manage analysis results, but does not provide in facilities for managing visualization findings. Considering the flexibility that preprocessing demands, we consider this to be an issue.

## 3.2 Preprocessing Framework Requirements

As we have seen, there are two main parts to preprocessing: the visualization of data and the modification of the event log. We have discovered a number of findings from theory, the analysis of the various visualizations, and from the implementation of the ProM Framework. Based on this research, the following requirements for the preprocessing framework can be stated:

- **General**
  - *The framework should be usable on a normal size computer screen.*
    The framework itself should be mostly invisible in order to leave room for the interactive visualizations. Screens with a minimum resolution of 1280x1024 should be sufficient to use the framework and its interactive visualizations.
- **Facilitate communication between visualizations**
  - *The framework functions as a communication hub for visualizations.*
    The framework should function as a controlling unit that coordinates communication from and to visualizations.
  - *The framework is able to manage communicated findings.*
    Visualizations communicate their findings to the framework. The framework should manage these communications. At a later point in time, an arbitrary visualization should be able to retrieve communicated information.
  - *Communication is performed in a format that is independent of the visualization.*
    Communication should be performed in a universal "language". It should be independent of the type of visualization that actually communicates the findings. For example, if findings are discovered in a process model, they should also be able to be visualized in a different type of visualization, such as a social network graph.
- **Provide a modification framework**
  - *Provide a central location where all modification plug-ins can be accessed.*
    The framework should provide a central location where all modification plug-ins can be accessed.
  - *The ability to run a number of modification plug-ins consecutively.*
    The user should be able to run multiple modification plug-ins consecutively, since it typically takes multiple steps to preprocess the event log.

  - *Allow the user to define an order of execution.*
    The user should be able to define the order of execution of the modification plug-ins, since the order of execution of these plug-ins may influence results.
  - *The plug-in should be able to export the (modified) event log.*
    Once the user is finished with preprocessing the event log, the resulting event log should be exported back to ProM.

- **Support iterative preprocessing**
  - *The ability to work with a number of visualizations simultaneously.*
    To further support the iterative process, we want to be able to run a number of visualizations simultaneously.
  - *The ability to freely switch from one visualization to another.*
    At the request of the user, the framework should change from one visualization to another.
  - *The possibility to switch between different states of the event log.*
    To further support the iterative process, we want to allow the user to switch to different states of the event log, before and after certain filters are applied.
  - *Support for the integration of interactive visualizations and the modification framework.*
    The interactive communications and the modification framework should be integrated to further support the iterative process.

## 3.3 Visualization Guidelines

Besides the requirements that are defined for the framework, we define a set of developer guidelines for the development of interactive visualizations. Since the framework is developed to be used in a flexible and interactive way, it is important that the visualizations behave appropriately.

There are two rules that must be strictly followed:

1. If a **visualization** needs to **modify data attributes**, you need to **first clone the event log**. The provided event log, which is the original data object, must not be modified.
2. **Traces and events must not be added or deleted by visualizations**.

The preprocessing framework needs to make these assumptions in order to correctly manage the interactive visualizations and the modification framework.

Besides these two rules, there are the following guidelines. The guidelines are not meant as strict requirements, because first and foremost we need visualizations to provide us with the information. If a visualization can provide us with valuable information, but it requires a somewhat longer time to complete, then this can still be reasonable tradeoff. However, we urge the developer of an interactive visualization to keep the following guidelines in mind:

- *The visualization should be usable on a normal size computer screen.*
  Similar to the requirement of the framework, the visualization, too, should be usable on a normal size computer screen.
- *The visualization should be sufficiently responsive.*
  The visualization will be integrated in a larger framework with other visualizations. In order to provide a good user experience it is required that the plug-in is sufficiently responsive for interactive use.
- *CPU/Memory-intensive operations*
  Visualizations should perform computationally heavy operations in a separate thread, to prevent blocking or slowing down the framework or any of the other visualizations that are active within the framework.

- *Handle changing event logs or reload after an event log change has occurred.*
  Because of the integrated modification framework, the contents of an event log may change. Interactive visualizations must be able to handle these changes, either by incorporating the changes in the visualization, or by reloading the data after the modification has been performed.
- *The visualization should provide information, visualize provided information, or both.*
  To make use of the interactive preprocessing framework, the visualization should provide support for communicating information, to visualize information from other plug-ins, or both.

## 3.4 Practical Decisions

Apart from the requirements for the framework and development guidelines for the interactive visualizations, there are also some practical decisions regarding the implementation of the framework.

- *Development platform: ProM 6*
  ProM 6 is the obvious choice as a development platform on which to build the framework. The ProM framework provides a basis with all the required basic functions such as the importing and exporting of events logs. There are also a lot of plug-ins that implement different visualizations, such as the visualizations that have been analyzed in Chapter 2. Furthermore, XES provides a flexible and extensible data structure that is desirable for the implementation of the framework.

# Chapter 4

# Design

In the previous chapter we looked at the requirements for the project. We specified the requirements for the framework and defined guidelines for the development of interactive visualizations.

In this chapter we discuss the design of the framework and interactive visualizations. We first explain the plan, then we go into the design of the framework and the reference implementations of the interactive visualizations. Finally, we discuss some of the design decisions that are at the root of the framework.

## 4.1 The Plan

The requirements that we specified in Chapter 3 consisted of three main requirements: 'Facilitate communication between visualizations', 'Provide a modification framework' and 'Support iterative preprocessing'. Here we explore the idea for the design.

The key point in this research is that the iterative approach to preprocessing should be better supported, as discussed in Section 1.2. Therefore it is a central concept in this idea. To enable effective, iterative work, flexibility is important. We need visualizations to be readily available, when they are required. Furthermore, we want to be able to do filtering and other types of event log modification and quickly and easily get access to the results. Additionally, it would also be a valuable addition to be able to switch back and forth through different versions of the modified event log during our investigation. To realise this type of flexibility we propose the following idea, depicted in Figure 4.1.

We develop a plug-in for the ProM framework: the Preprocessing Framework, that can provide us, i.e. process analysts, with an overview of the event log. This overview is comprised of a selection of *interactive visualizations* that are available in the ProM framework. Each interactive visualization offers a unique view of the event log. This is important, since we require varous kinds of information on the event log to be able to thoroughly investigate and explore. These interactive visualizations would have to be *instantly available*, so we need to run them simultaneously.

We also need to be able to continue the investigation using another interactive visualization without too much interruption. Having multiple of these visualizations readily available helps, however we also need to "*transfer*" our findings from one interactive visualization to the next, if we want to continue our investigation without the interruption of having to look up all of the relevant elements in this other visualization. As stated in Section 3.2, we need to facilitate the *communication of information* from one interactive visualization to the next. For this we introduce the notion of 'tags'. The 'tags' we use here are similar to the notion of tags used on the internet. We 'tag' a set of events that are of particular interest. Afterwards we can easily retrieve the events by calling upon the tag. The tags are managed by the framework. Interactive visualizations can query the framework for information on tags and can retrieve the contents of a tag they need to visualize. Tags are designed to be independent of the visualization, so it is possible to create a tag in one interactive visualization and visualize it in another. Hence it is possible to tag interesting

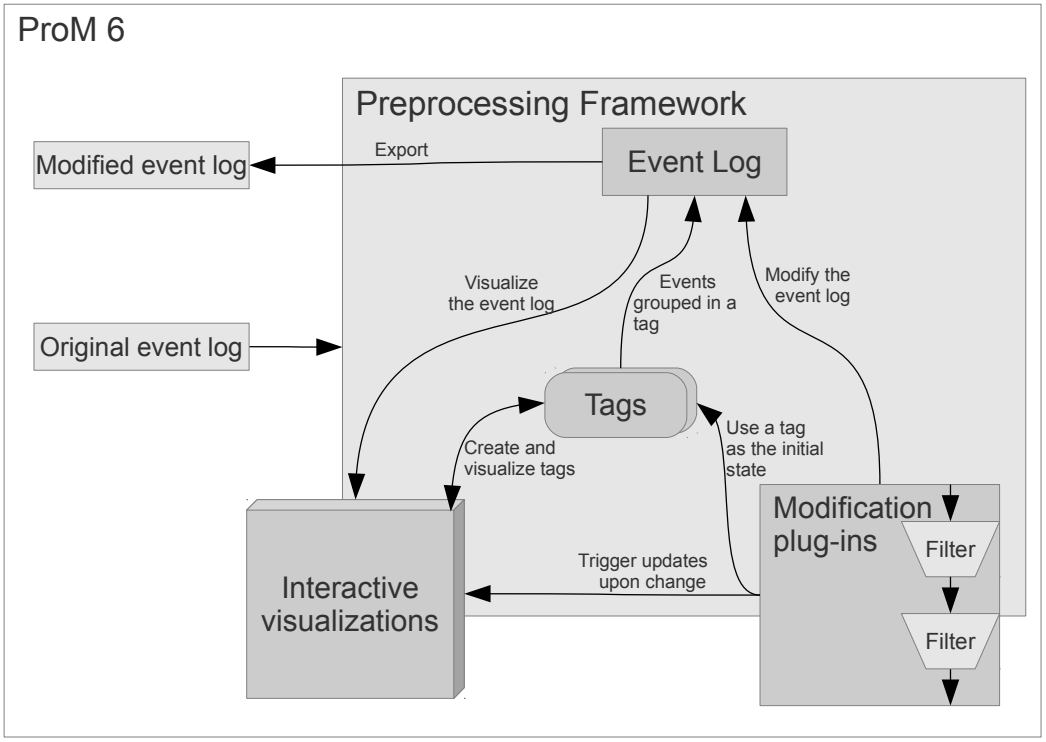


**Figure 4.1:** A high level overview of the Preprocessing Framework cooperating with other components within ProM.

events, switch visualizations and recall the events to continue our research.

Finally, in order to actually do the preprocessing, we need to be able to *modify the event log* based on what we have discovered in our research. Again, we want to do this from within the preprocessing framework, since we are still investigating or exploring the event log. Furthermore, we need to be able to do multiple modifications to the event log, and since the order of execution is relevant to the results, we should be able to specify the order. To facilitate these requirements, we design a *modification framework*. This part manages all modifications to the event log and takes appropriate actions after modifications have been applied. We are able to run multiple modifications by creating a chain of modification plug-ins. The modification plug-ins in the chain are executed consecutively (i.e. the result of the first modification plug-in is used as input for the second, and so on). The final result, as well as intermediate results, can be used in our analysis.

So, now we are able to switch views on the event log by switching interactive visualizations and we are able to switch to different event log states by applying modification plug-ins. We do all this from within the preprocessing framework, hence irrelevant details, such as looking up (i.e. in many cases searching) a particular point of interest in another visualization or adjusting the interactive visualizations to the modified event log, can all be taken care of from within the preprocessing framework itself.

## 4.2 The Preprocessing Framework

The preprocessing framework is designed to accommodate the features described in the section above, Section 4.1. We now discuss a number of properties of the preprocessing framework that together provide all of the required features. First we discuss the Interactive Visualization interfaces. Next we discuss the modification framework.

### 4.2.1 Interactive Visualization interfaces

An important piece of the puzzle are the Interactive Visualization interfaces. Interactive visualizations are a subset of the visualization plug-ins found in ProM. The interactive visualizations are visualizations that additionally implement the *Interactive Visualization* interface, allowing them to be managed by the *Preprocessing Framework*. The interactive visualizations are represented by the 'Interactive Visualizations' element in Figure 4.1.

These interfaces, of which the class structure is shown in Figure 4.2, are defined to facilitate the communication between the interactive visualizations and the managing controller. The interface for the visualizations are designed to be simple and flexible, since we need to support a wide range of visualizations. We want to make it as easy as possible to modify an existing visualization into an interactive visualization. Interactive visualizations are not required to be managed and should function without problems without a manager being present. Therefore it should also be possible to execute any interactive visualization as a stand-alone visualization.

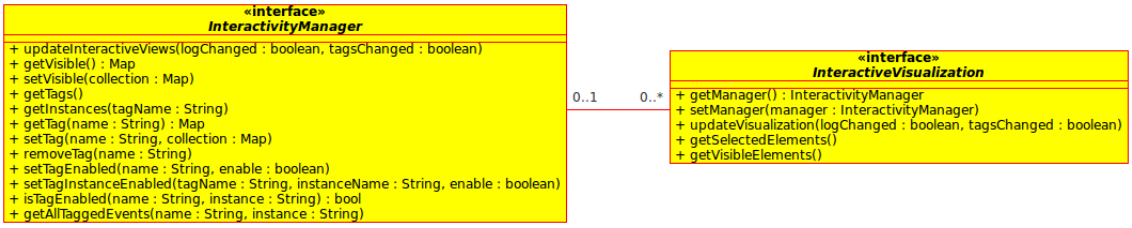


**Figure 4.2:** The interfaces facilitating the interactive visualization features.

#### Interface: Interactive Visualization

The '*Interactive Visualization*' interface is designed for the visualizations that are extended with the interactivity features. They provide a generic way for the controller to query a visualization for its information whenever the user decides to make a tag. This is also shown in Figure 4.1 in the connection 'Create and visualize tags' connecting *Tags* to *Interactive visualizations*. There are two possibilities for tagging. First is to make a tag of a user's selection in the currently active visualization. The second method is to make a tag of everything that is currently visible in the visualization. This is especially useful for visualizations that graphically visualize data, and where the user may have zoomed in on a specific area of interest.

Furthermore, there is a method called 'updateVisualization'. This method is triggered whenever one or more tags have changed or when the event log has changed. Updates triggered by the event log are issued from the modification framework, as shown in Figure 4.1 in the connection 'Trigger updates upon change' between 'Interactive visualizations' and 'Modification plug-ins'. This is to inform the visualization of changes and to offer the visualization a possibility to adjust accordingly.

The other methods in the interfaces are simply a way for the manager to let the visualization know that they are being managed and for the visualization to know how to contact the managing controller.

#### Interface: Interactivity Manager

The 'Interactivity Manager' interface is designed to be implemented by the controller managing the interactive visualizations. This interface is used by visualizations to contact the controller in order to query for tags. The visualization can create and update tags as well as enable tags and ask for the current status (enabled or disabled) for specified tag names.

The most basic feature for a visualization is to ask for all tagged events. It will get a set of all events that are tagged and currently enabled. These can then be shown in the visualization, as is also represented in Figure 4.1 in the connection between *Interactive visualizations* and *Tags*.

A visualization can then determine for itself what is the best way to visualize the tags, such as distinct colors, shapes, alignments, etc. The other methods allow for more specific operations on querying and modifying tags.

The 'updateInteractiveViews' method is used to issue a command to all interactive visualizations to adjust themselves to changes in the environment, i.e. changes to tags and/or their data source: the event log. In Figure 4.1, triggering the updates is handled via this method.

### 4.2.2 Modification framework

The other key requirement for the project is the requirement of a modification framework: a centralized location where plug-ins are available to perform any kind of modification on the event log. The modification framework is easily accessible from within the preprocessing framework and can be used in conjunction with the interactive visualizations. In Figure 4.1, the modification framework is represented by the *Modification plug-ins* element.

The modification framework allows the user to specify any number of modification filters. Since the order of execution of modification plug-ins is relevant, we allow the user to determine the order in which the filters are executed. Furthermore, the framework is based on selecting a state of the event log, rather than executing a number of plug-ins. Instead of receiving an end result after all the modification plug-ins have been executed, we can define a chain of modification plug-ins and every intermediate result can be visualized upon request, as well as the initial data set and of course the final result, without having to remove modification plug-ins.

The modification framework should be flexible enough such that developers are not limited in any way in performing modifications. Furthermore, the modified event log should be sent to each of the visualizations once plug-ins are finished modifying the event log, since the visualizations are expected to adjust to the modified environment, i.e. the modified event log which contains the source data for the visualizations. Figure 4.1 shows that the modification framework triggers the updates for the interactive visualizations.

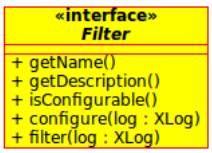


**Figure 4.3:** The 'Filter' interface for the modification plug-ins.

A 'Filter' interface, as shown in Figure 4.3, is defined for modification plug-ins to implement. Using this filter interface, the the modification plug-in is provided with an event log. The first time for the plug-in configuration, but this is optional. Afterwards, when the modifications should be performed, the event log is provided again and this time the modification plug-in is expected to make all the modifications in this event log. This is shown in Figure 4.1 with the connection between the *Modification plug-ins* and the *Event log*.

Finally, when all preprocessing work is done, we can export the event log in its current state. Figure 4.1 shows this. A new event log is generated that represents the current state of the event log in the preprocessing framework and this event log is passed on to the ProM framework where it will become available in the list of data objects, ready for use outside of the preprocessing framework.

## 4.3 Interactive Visualizations

For the framework to actually be useful, we also require visualizations that support the interactive features that the framework offers. We create some reference implementations to demonstrate the capabilities of the framework. These implementations are in Figure 4.1 represented by the

*Interactive visualizations* element. We selected three visualizations that provide us with different types of information we consider to be useful in area of preprocessing. The selection is:

1. *Log Visualizer*
   The Log Visualizer offers a wide range of information, as we found out in Section 2.3. The 'Dashboard' offers the basic information such as averages over the process as a whole. The 'Summary' provides event-level information such as start and end events. Furthermore the inspector enables the user to inspect the event log data in detail, which will certainly be useful once we start investigating our findings in detail.

2. *Dotted Chart*
   The Dotted Chart offers a visual picture of the event log, as we found out in Section 2.2. This visualization gives a good impression of what happened over time in event log and enables the user to look for patterns or absence thereof. This is useful both for investigative research into certain events, and for exploratory research into the event log.

3. *Pattern Abstractions*
   The Pattern Abstractions plug-in does not offer any visual depiction of the event log, as we know from Section 2.5. However, the Pattern Abstractions plug-in is particularly useful for finding the things that are not easily detectable by human observation. Human observation can be used to detect large scale patterns. The Pattern Abstractions plug-in, however, scans the event log for any kind of repetition and provides us with a complete list of patterns. This can be used to detect slight deviations from the normal patterns that would otherwise go unnoticed because of their rare occurrence, i.e. the proverbial needles in the haystack.

This is an initial selection of visualizations, and by no means the only selection. This selection consists mostly of general visualizations, however the Pattern Abstractions plug-in can currently only work with the control flow perspective. There are obviously more candidate visualizations that can be modified to support the interactive features, however, since these implementations are only meant as reference implementations to demonstrate the capabilities of the framework, we limit ourselves to these three visualizations.

Apart from the visualizations themselves, we need the *Preprocessing Framework* to provide a screen area where the visualizations can be shown. In Section 3.2 we already state that the framework itself should be mostly invisible such that the available screen space can be used for visualizations. For this purpose, we use a visualization component that has the ability to hide additional control panels until they are requested, leaving as much room for the visualization as possible. This *ScalableViewPanel*, as it is called, is what we use for the preprocessing framework. The control panels that provide the features of the preprocessing framework are provided via the *ViewInteractionPanel* interface.

### 4.3.1 Log Visualizer

The Log Visualizer [14] is mostly a static visualization. It shows the information typical to the Log Visualizer, but does not provide much interaction. The Log Visualizer is not a type of visualization that is typically used to gain new insights into an event log, however the *Inspector*, which enables us to look into the raw data of an event log, is very suitable for highlighting tagged events. This way, we can easily look up the events of our findings in the event log and see what data is there.

Support for highlighting tagged events in the *Inspector* is easy. As we explained earlier, visualizations communicate by providing or retrieving a set of events. As we are highlighting tagged events, we request all events that are contained in the tag. Since the *Inspector* visualizes the event log as is without analysis or transformation, we only have to look up the events and highlight them in the visualization.

Support for the modification framework is a little less trivial. The *Dashboard* and *Summary* parts contain aggregated data, so that data has to be updated after the event log has changed. We use the new source data to generate the *Dashboard* and *Summary* information. The *Inspector*

is again a trivial case. Since it simply shows the data in the event log, there is no need to update any information. The data is retrieved from the event log upon request, hence the *Inspector* automatically adjusts to the new event log.

### 4.3.2 Pattern Abstractions

The Pattern Abstractions plug-in [8] is the complete opposite of the Log Visualizer. The Pattern Abstractions plug-in is interactive, analyzes data, but does very little visualization. The results of the Pattern Abstractions plug-in are repetition patterns. These repetition patterns, either individual base patterns or an abstraction consisting of a number of related base patterns, could be communicated to other visualizations.

The support for the Interactive Visualization interface has been implemented by the author of the Pattern Abstractions plug-in: *R.P. Jagadeesh Chandra Bose*. Additionally, he suggested (and implemented) the best way to partition the results into tag instances as he obviously is most familiar with the results that the Pattern Abstractions plug-in produces. He also provided valuable input during our discussions on what is the best way to insert a notion of semantics into the set of events that we communicate between visualizations.

As we explained earlier, in the area of communicating information to other visualizations, we have to make a trade-off. The patterns that are discovered by the Pattern Abstractions plug-in have some specific properties. For example a base pattern always exists within a trace, never over multiple traces. A base pattern can occur multiple times, however. Abstractions can contain multiple base patterns. These types of information cannot be communicated to other visualizations, since this information is specific to this particular type of result. By storing each base pattern in its own tag instance, we can keep base patterns separated. This does not provide the full semantics that the original results provide, however it does provide some control over how the information is communicated.

As for the changes that can be made to the event log. Since the nature of the Pattern Abstractions plug-in is to do complex analysis, changes to the event log invalidate the analysis results. Since the type of analysis that the Pattern Abstractions plug-in offers is not always required, we decided to reset the interface after the event log has been modified. If the user requires further analysis, it means that he has to do the initial analysis steps again, as is to be expected since the event log has been modified.

We remarked earlier that the Pattern Abstractions plug-in may require some time to complete its analysis. However, the analysis process is only initiated on demand of the user. Therefore it only takes up computing time when we explicitly instruct the plug-in to analyze the event log. Since it does not start this computation upon initialization it does not directly interfere with the operation of the preprocessing framework, hence it is acceptable given the visualization guidelines. *Furthermore, the author has been working on optimizing the analysis and has managed to significantly increase the performance of the analysis.*

### 4.3.3 Dotted Chart

The Dotted Chart plug-in [12] is a visualization that offers a range of graphical representations of the event log. Primarily, it is meant as a way to visualize cases of the business process over a range of time in a grid. However, there are additional options for using resources, cases and transitions.

The Dotted Chart is a valuable visualization, however the implementation was not very efficient, more specifically the memory requirements were very high, since every combination of visualization options was prebuffered. And it had some issues as we have already seen in Section 2.2. Additionally, the Dotted Chart was originally developed for ProM 5.2 and as such lacks the flexibility that is offered with the data structures that are used in ProM 6. Therefore we decided to reimplement the Dotted Chart to make use of the available features in ProM 6.

We decided to take some of the earlier observations into account in the design of the plug-in. The first addition is related to observation 4 in Section 2.14. This observation points out that there

are hardly any generic components that can be used to quickly visualize raw data. Furthermore, observation 5 points out - and this was also applicable to the original Dotted Chart - that the user is offered only a static set of options to use when drawing the Dotted Chart. Since ProM 6 offers more flexible data structures, we can add an option to let the user choose out of any of the data attributes that are available in the event log which to use for the visualization of the data.

#### Visualizing the dots

The critique that the original Dotted Chart implementation got, were mostly related to the way the dots were visualized. They are big, causing overlap both in horizontal and vertical case, they do not scale, so the overlap can be over dozens of rows or there is no overlap at all, and the size can increase causing even more overlap. Since overlapping the rows (which typically means that the dot overlaps the row of a different case) does not have any significant meaning, so it only obscures information.

The new Dotted Chart implementation uses a number of tricks to make these issues less of a problem. This implementation does not completely solve all of the problems, since this would cause other issues, however it tries the lessen the problems caused by the disadvantages.

1. **Dots scale**: The size of the dot is determined by the size of the grid cell to which the dot belongs. The result is that no dot overlaps with another row or column. Therefore it cannot cause the unintended side effect that it obscures information. However, in a zoomed-out situation, this causes dots to be 1 pixel in size. Which makes them close to invisible. Therefore we make a trade-off with the following two items: 2 and 3.

2. **Emphasizing dots**: To emphasize the existence of the dot, which could be as small as 1 pixel, we draw a largely transparent, filled circle around the dot. This breaks with the ideal of non-overlapping dots, however the fact that it is largely transparent prevents it from hiding information. This also gives an additional emphasis on dense areas, yet it is still possible to distinguish dots that are close together.

3. **Stacked events**: Stacked events, i.e. events that are on the same grid cell, would obscure each other. We do not draw all events, instead we draw a single dot with a solid circle around the dot to indicate that there is actually more than one event present at that position. Again, this effect does overlap neighboring grid cells, however it does not completely obscure the other grid cells. A disadvantage of this approach is that the different colors are not apparent anymore for other events in the stack.

4. **Positioning based on grid cells**: Dots are positioned in grid cells. Each dot gets its own cell. Therefore it cannot occur that a dot overlaps 99% of the previous dot on the same row. Dots that, given a certain time granularity, would end up on the same position, get stacked. The granularity is determined by the user.

5. **Zooming**: We can zoom with any aspect ratio and zoom level. This is crucial since we need to be able to focus and zoom in on any area that might be of interest. Whatever the aspect ratio of the zoom box is, it gets stretched to the complete, available drawing area of the screen. We can zoom in to any depth, even to a single dot.

6. **The emphasis lessens as the dot grows**: We use a fixed number of pixels between the dot and the filled, transparent circle used for emphasis, and between the dot and the solid circle that indicates multiple events. Therefore, the larger the dot grows (as we zoom in further) the less emphasis there is. Hence we automatically lessen the emphasis when it is not required anymore.

7. **Information on events**: The information that we request for a group of selected events, is not visualized in a tooltip anymore. Instead we use an Interaction Panel to nicely list the information. We can show and hide this interaction panel, so we are always sure that we have all information, and it does not necessarily hide the chart itself.

By applying these techniques, we try to make a more balanced trade-off for visualizing information. We could do with only showing single pixel dots, however these are very hard to find, which makes the analysis next to impossible. We have already experienced the opposite, since the original Dotted Chart implementation uses fairly large dots, and as a result suffers from dots actually obscuring information.

### A generic grid component

The goal of the generic grid component is to be a component that can visualize several types of grid-based visualizations. A property of the Dotted Chart is that it may contain lots of elements, so the grid component must be able to cope with large grids. This is not necessary for every type of grid visualizations.

The grid component has to be both generic enough to handle several types of grid applications and flexible enough to cope with the user's desire for different configurations of the visualization. In order to keep the user in control of the component, we decided to create an abstraction between the source data and the grid component. Instead of a typical approach where the grid component directly reads out the source data and allows itself to be configured in a plethora of different ways, we decided to put the user directly in control of the input data.

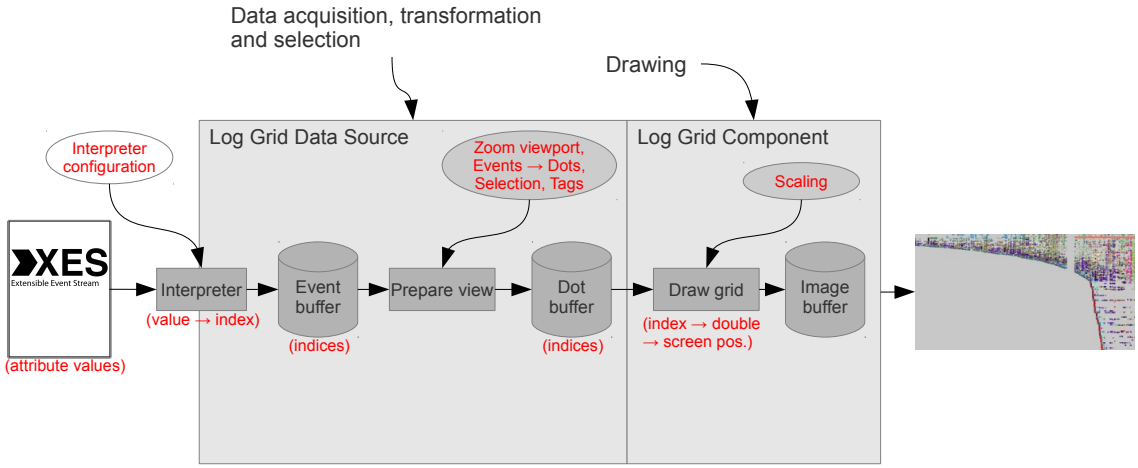


**Figure 4.4:** A graphical representation of the process flow of the generic grid component. (Technical details are colored red.)

Figure 4.4 shows the data flow through the generic grid component in preparation of being drawn on the display. As we described earlier, source data is processed by interpreters and the results are put in an event buffer. This is done in the LogGridDataSource, which is responsible for acquiring, transforming and selecting data. The first buffer contains all the events with X and Y coordinates as provided by the interpreters as well as a color. Then we prepare the events according to the current view configuration, i.e. what should we zoom in to. Events are stacked if they occupy the same grid position, and only events that are actually drawn are buffered. This buffer is filled with dots that are ready to be drawn. We also determine which of the dots are selected and which are tagged.

Then we start drawing the dots on the screen. This is handled by the LogGridComponent. This component converts the indices to screen coordinates and draws the grid on request. As long as the visualization does not change, repaints are performed using a buffered image for efficiency.

### Interpreters

We already shortly talked about the interpreters. The interpreters are used as a way to influence the data at the moment it is being loaded into the grid component for visualization. The class

diagram in Figure 4.5 shows how the generic grid component is built up and how components are interconnected. There are multiple types of interpreters, each for a specific concern of the grid visualization. First is the Coordinate Interpreter, this interpreter is accessed in order to retrieve coordinate information. We can provide two coordinate interpreters to the LogGridDataSource, one for the X-axis and one for the Y-axis. The Dotted Chart typically has time on the X-axis and the event log's cases on the Y-axis, however it is trivial to swap the two axes. The second type of interpreter is the Color Interpreter. This interpreter is used to determine how an event should be colored.

In Figure 4.5 we see the LogGridDataSource and the LogGridComponent. These two classes form the generic grid component, which can also be seen in Figure 4.4. The LogGridDataSource contains the event log, which is the source of event information, and interpreters for determining the X-axis coordinate, Y-axis coordinate and color mapping. The interpreters are also references from within the Interaction Panels. These are the control panels with which the user can configure the Dotted Chart visualization. The LogGridComponent, which is responsible for drawing the grid, implements the Interactive Visualization interface and thus offers the interactivity features. Both the interaction panels and the LogGridComponent implement the interfaces used for the ScalableViewPanel.

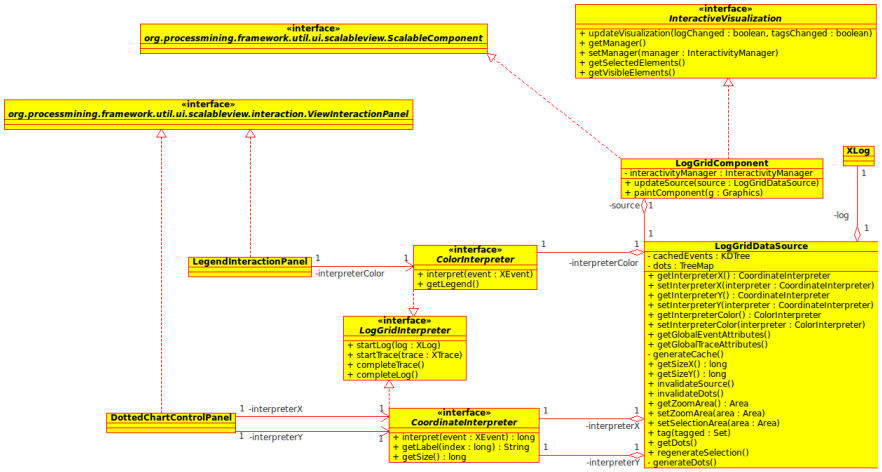


**Figure 4.5:** A class diagram of the new Dotted Chart design showing only the relevant elements: grid data source & component, interpreters, interaction panels, Interactive Visualization interface, Scalable Component & View Interaction Panel interfaces

When the LogGridDataSource is created and the interpreters are provided it is possible for the data source to generate the dots. These dots can then be drawn on the screen. We also want to be able to influence how the grid is drawn. Therefore we can create interaction panels and pass references to the interpreters upon creation. The user can then configure his preferences via the control panel. The control panel passes on the preferences to the interpreters and subsequently requests the grid component to reload its event data. Then the interpreters are able to interpret the data according to the user configured preferences and the grid component consequently draws the new grid.

### Times, cases and attributes

Specifically for the Dotted Chart we design interpreters for the visualization of events in Dotted Chart style. The Dotted Chart visualization consists of a time axis and a case axis. Hence we design a coordinate interpreter that determines the index based on the event's time stamp for one axis. We can choose from four types of time: actual time, relative time, time based on a ratio, and logical time. The time interpreter is exact up to millisecond precision, just as the data is that is

provided by OpenXES. For the other axis, we design a coordinate interpreter that determines the index based on case data. This interpreter can order cases based on the provided Comparator.

For the use of an axis with distinct data attribute values instead of cases, we design the Attribute Interpreter. For this coordinate interpreter we require the data attributes to be registered as a Global Event Attribute (see [5]) since we need a guarantee that every event has this data available.

#### Interaction

The generic grid component supports selecting and zooming operations. To further make the visualization quick and easy to use, we decided to do zooming operations with the left mouse button, while selecting is made possible using the right mouse button. Therefore we do not have to change which operation is currently configured on the mouse buttons. Using an information panel we can access event information of all selected events. And the Dotted Chart control panel enables the user to reconfigure the dotted chart to his own requirements.

The generic grid component also supports the interactive visualization interfaces. The component can return the selected elements upon request, as well as visualize the events contained in an existing tag.

## 4.4 Design Decisions

During the design of the preprocessing framework we have made a number of decisions. These decisions, with arguments and possible alternatives, are now discussed. First up, in Section 4.4.1, is the challenge of identifying visualization elements and find a way to communicate these elements to other, typically completely different, visualizations. The second challenge, in Section 4.4.2, is how to communicate the information without losing all of the semantics. The third challenge, Section 4.4.3, is about what is actually considered changing the event log. The fourth is Section 4.4.4. It is on the decision that only the modification framework is allowed to change the event log. The fifth is Section 4.4.5. It describes how we handle the modifications that are being made to the event log. And finally the sixth section, Section 4.4.6, describes how we handle tags that cannot be completely visualized because the event log has been modified afterwards.

### 4.4.1 Identifying visualization elements

The first problem that we have to tackle is how to communicate findings from one visualization to another. The results of the execution of an analysis plug-in is returned to the framework. However, these results are each in their own format and any plug-in that requires the result would use it in its native format. We do not have this luxury, since visualizations can be completely different and we want them to be able to visualize the findings from any other visualization.

There are a number of possible ways to communicate these findings, however many are not suitable in all situations. We cannot use any data attributes. For one, no specific data attribute is guaranteed to be available in every event log. The XES standard [5] is inconsistent[1] in this sense, but we think it is reasonable to assume that events are not required to have even a single data attribute, hence we cannot rely on this data being available. Additionally, XES does not enforce the existence of any data attribute, so we cannot rely on any data being available. Second is that if we choose one particular data attribute, we are bound to a particular perspective. And third, the chosen data attribute may not be sufficient to identify individual elements in all visualizations. For example, we can identify an activity in a process model by the activity name. This method lets us point precisely at this one element in the process model. However, a social network graph

[1] In the XES standard, at the time of writing: XES version 1.0, revision 1, November $25^{th}$ 2009, the flow schema on page 10 suggests that events require at least one attribute, while the XES XSD requires a minimum of 0 occurrences of attributes.

clusters data differently, and we might as well be pointing to all elements of the social network graph.

We could also make use of the way the event log is partitioned in traces, each trace representing a case. However, this means that we are forced to make very rough selections as a case can contain thousands of events. Furthermore, the partitioning in cases may not always be desirable. For example, if we are looking into employees, an event log partioning scheme based on customers may not be feasible.

The best option for communicating findings is by specifying a *set of events*. For one, we need atomic elements, since we cannot predict how visualization may cluster data, and we need to be able to define any kind of selection. Secondly, events are the key pieces of data that any visualizations recognizes and uses. Third is that events themselves are independent of how the event log is partitioned, hence this is still feasible with alternative case definitions.

### 4.4.2 The problem of tag semantics

Closely related to the problem of identifying visualization elements, is a problem of semantics. For any visualization, there is a meaning to the way the event log is visualized. Similarly, when a selection of events is created within that visualization, be it by the user or within the visualization itself, there is an underlying semantics to this selection of events. However, if we only communicate one big set of events, any semantics is lost.

An ideal solution would be to have full semantics available. Any visualization that knows these semantics would be able to use the data to its full potential, while any other visualization would be able to simply grab the events and visualize them, thus ignoring the underlying semantics. However there is a caveat to this approach. If we fully support semantics, and instead of specifying individual events we define classes, this would require a computation step in order to retrieve the events. We want to limit the amount of computation needed for these kinds of operations in the preprocessing framework in order to ensure as much responsiveness as possible.

The solution in this case is a compromise. Instead of providing full semantics, we will allow individual instances to be defined. Each instance is a set of events. And the set of all instances forms the tag. Now visualizations can communicate a set of instances, instead of a single set of events. The visualization themselves can determine how these instances are used.

In essence, this moves the semantics from code to human interpretation. The plug-in can now express a form of "semantics" by intentionally dividing events over a number of instances. We, however, have to understand the semantics of this division in order to utilize it. Users that are unaware of the semantics can use all instances, the result is a big set of events. On the other hand, users that do understand the semantics, can use a specific selection of instances in order to do more advanced research.

### 4.4.3 What do we consider to be a modification to the event log?

An important item to discuss is what is actually considered to be changing the event log. Conceptually, the XES data format stores the notion of 'an occurrence of an event' that happened for 'a particular case'. Data (attributes) can then be added to the case or event, however the empty case or event, i.e. without data, already expresses that something has happened and is considered valid.

When we are talking about modifying the event log, there are a number of scenarios. We could be talking about adding or deleting a case, adding or deleting an event, and adding, modifying or deleting data from either a case or an event. So, there are several possible operations that we could consider as "changing the event log". These operations can be partitioned in two different classes. First is the class of changes that affect the events themselves: adding and deleting cases and events. The second class changes the data on cases or events. This second class, however, does not change anything about the fact that the event has occurred.

The relevance of data for a case or an event is completely determined by the plug-in that is used to analyze the event log. A social network miner relies heavily on organizational data

and does not care for any other type of data, such as the name of the activity or the time at which the event was executed. While process model discovery algorithm typically does not care for organizational data. Hence the relevance of data attributes is determined by the plug-in. The occurence of events, on the other hand, is relevant for every plug-in, since that is the source of data on which every analysis is based.

Therefore, for this project, we consider the addition or removal of cases or events as changing the event log, while the addition, modification or deletion of data attributes is not. This is closely related to Rule 2 of Section 3.3 'Visualization Guidelines' which refers to this modification of event logs.

### 4.4.4 Only the modification framework changes the event log.

The preprocessing framework brings together both the visualization and the modification of event logs. However, many visualizations, among them also the Dotted Chart and the Pattern Abstractions visualizations, require that the event log does not change during the visualization. Visualizations typically perform analysis steps prior to visualization or buffer data in order to speed up the visualization. Therefore, changing the source data will lead to an inconsistent state amongst the interactive visualizations.

The modification framework, however, may perform modifications to the event log. The modification framework forces an update from all visualizations after the modifications have been performed. This creates a controlled situation where the inconsistent state is immediately resolved by allowing the interactive visualizations to adjust to the latest changes before further action is possible.

### 4.4.5 Handling the modifications that are being made to the event log.

Another important design decision that is necessary for the goal of iterative preprocessing is about the modification of event logs. As we discussed in Section 4.1, the idea is to simultaneously load multiple visualizations that can then be used, as well as facilitate a framework that can modify the event log. For this idea to work we need to be able to communicate the changes made to the event log to the visualizations and they have to adjust the visualization to the new data.

Visualizations can be partitioned in three groups:

1. **Visualizations that visualize individual events**: These visualizations display individual events. When these visualizations need to adjust to modifications to the event log, they only need to leave out or add one additional visual element. (See Section 4.4.3 for a definition of what exactly we consider a modification to the event log.) An example of such a visualization is the Dotted Chart.

2. **Visualizations that aggregate data and visualize these aggregates**: These visualizations display elements that represent an aggregation of events. These visualizations can still adjust to the changes, however they need to "adjust" the aggregations in order to do so. An example of such a visualization is the Log Visualizer's Summary.

3. **Visualizations that perform complex computations**: The last group of visualizations is the group that does complex computations on data. These computations may not be easy to adjust. As a result, in many cases to adjust to changes in the event log, the whole analysis has to be redone. An example of this is the Fuzzy Miner.

The first two groups should be able to adjust to changes in the event log fairly easily, the third group however, will not. Unfortunately, the third group is an important group. These are typically the more complex visualizations, among them the process model discovery algorithms.

There are some conceptual issues with the challenge of handling modified event logs. We need to, in some way, catch modifications that are being made and communicate them to the

visualizations. Furthermore, we need *all visualizations* to adjust to the modifications, since we otherwise end up with an inconsistent view on the event log. Additionally, there are some practical issues. Visualizations are known to clone the event log and possibly make changes. Actually adding or deleting traces or events is not allowed. However, there is still a possibility of attributes being added, modified or deleted. Furthermore, given the choice of event identifier, we need to somehow cope with the modifications thare are being made to the event log and how this affects the indices of traces and events.

Given the types of visualizations and the issues described above, we need to look for an appropriate method for handling modifications and updating the visualizations. First we look at a number of options. Afterwards we select the best option given the requirements.

1. **Reinitialize everything after a modifications have been made**: Every time the event log is modified, reinitialize the visualizations with the current version of the event log. There is no adjustment necessary, since we reinitialize all visualizations using the modified event log as input.

2. **Update event log by passing down a modified event log after modifications have been made**: Initialize the visualizations with the pristine event log. Whenever the event log is modified, pass on the modified event log to each visualizations for them to adjust.

3. **Keep using the initial event log, separateply communicate and process modifications**: Initialize the visualizations with the pristine event log. Whenever the event log is modified, communicate the changes in some representation to each visualization allowing them to adjust to the event log.

4. **Initialize visualizations with a self-modifying event log**: Instead of having to communicate changes to the event log, a self-modifying event log is passed down to the visualizations. Whenever the event log is modified, the visualizations only require a trigger that allows them to adjust to the changes that have occurred to the event log.

Now that we have a list of possible approaches to handling modifications, we look at the advantages and disadvantages. First are options 1 and 2. Reinitializing everything will work, however this also takes the longest time, since we also reload things like the user interface which may not require reloading. Option 2, on the other hand, can selectively reload those things related to the event log. However, this does not solve the problem with the event identifiers as they still rely on the pristine event log. Modifying the event log still renders the event identifiers unusable. So we require a separate solution for that problem. And visualizations that clone the event log and add data in the form of attributes will lose that data.

Option 3 does not have the problem that data will be lost, since the changes are communicated separately. However, these changes have to be parsed and incorporated in their version of the event log. Hence every visualization has to parse these changes. Additionally, if the changes are known, they can be used to convert the event identifiers, so this does not render the event identifiers unusable.

Option 4 is a sort of hybrid of options 2 and 3. Modifications do not need to be communicated. Instead of communicating the changes, we only trigger the visualizations, asking them to adjust to the modified event log. The modifications performed on the event log automatically also occur on the event logs that are used within the visualizations. From the visualization's point of view it seems as if the event log modifies itself. However, it is still possible to preserve changed data attributes. Furthermore, with the internal knowledge available within the self-modifying event log it is possible to convert the event identifiers. There is a possibility to compile a list of changes (delta), however this delta would always be against the pristine event log, instead of against the previous version of the event log.

Our solution is based on option 4, since it offers a flexible approach and at the same time keeps the additional work that the visualization has to do at a minimum. The implemented solution currently does not generate the delta, therefore it does not take full advantage of the possibility for the first two types of visualizations to efficiently adapt, however these types of visualizations

are typically lighter to compute. This is the trade-off for the other obvious advantages that we get with option 4. The details of our implementation of this solution are discussed in the next chapter, Chapter 5 sections 5.2.2 and 5.2.3.

### 4.4.6 Incomplete tags after modifying the event log

An unintended side effect of modifying the event log is the fact that events or even complete traces may get to be removed from the event log. Tags that have been created before the event log got modified, do still contain those events. However, when an attempt is made to visualize this particular tag and the framework has to convert the tag, or to be more precise the contents of the tag, to the current state of the event log, we will encounter some events that cannot be converted to the current state. The reason being that these events have been removed from the event log in its current state.

The question whether or not this really is an issue, strongly relates to an earlier decision we made in Section 4.4.2 on the semantics of a tag. We then decided to leave the semantic interpretation to the process analyst. We reason similarly here. Given that we do not have full semantic information on the contents of the tag, we cannot know whether or not it poses a problem if we leave out events. For example, if we previously tagged all occurrences of a particular pattern, when we filter out some event that is contained in the pattern, the pattern will be broken. Therefore the tag has become useless, since it does not offer correct information anymore. However, we are unable to determine that without access to the semantics. We, again, leave the decision on whether or not the tag still contains valuable information to the process analyst. We convert the tag in order to match the current state of the event log, therefore we are able to visualize the tag. The user then has to determine for himself whether or not the tag still has value.

## 4.5 Conclusion

In Section 4.1 we have explained our plan for a *preprocessing framework*. The framework itself houses two main features. The first is to support multiple simultaneous visualizations, facilitate in communication between these visualizations and manage the visualizations. The second feature is related to the modification of the event log. The modification framework provides a central location for modification plug-ins, controls the modifications and the different states of the event log. Using this framework we move the focus from executing a single modification on an event log, to selecting a particular state of the event log that is needed. The design of these two features is discussed in Section 4.2.

By using the preprocessing framework, we allow for far more flexibility than was possible without the framework. This flexibility is what opens up the possibility to iterative preprocessing. The framework also manages the state of both the event log and the visualizations, allowing us to focus on process analysis instead of the management of the data objects and visualizations.

In Section 4.3 we discussed the visualizations that we are modifying and to what extent they need to be modified. The Log Visualizer plug-in, in Section 4.3.1, and the Pattern Abstractions plug-in, in Section 4.3.2, only need to be extended such that they implement the *Interactive Visualization* interface. The Dotted Chart plug-in, in Section 4.3.3, is redesigned completely to, among other reasons, make full use of the features that are newly available in ProM 6.

Finally, we discussed the most important design decisions in Section 4.4. These decisions are related to how the underlying design of the preprocessing framework functions in order to facilitate its features. Decisions are made for the identification of visual elements in Section 4.4.1, the problem of tag semantics in Section 4.4.2, what we consider modifying the event log in Section 4.4.3, who is allowed to modify the event log in Section 4.4.4, how to handle event log modifications in Section 4.4.5, and finally what to do with incomplete tags in Section 4.4.6.

# Chapter 5

# Implementation

Now that the ideas and the design of the framework have been discussed in Chapter 4, we move on to the implementation. First we summarize in Section 5.1 what exactly is implemented. We make a number of decisions that are required for the implementation. We discuss these decisions in Section 5.2. Afterwards, in Section 5.3, we explain how to use the *generic grid component* and how it is used for the Dotted Chart implementation. We then discuss a number of limitations of the current implementations in Section 5.4. Finally, we draw our conclusions for this chapter in Section 5.5.

## 5.1 Implementing the designed preprocessing framework

As we have seen in Section 3.4, the ProM 6 Framework is the most suitable development platform for an implementation of the design. ProM 6 makes use of the OpenXES library in order to support the XES event log file format and also uses these data structures for the representation of the event log in memory during the execution of the program.

The OpenXES library, which is available at `http://www.openxes.org/`, offers data structures for event logs and every element thereof. Since there is no bound to the number of data attributes per event/trace/log, storing a complete event log in memory will consume too much of the available system memory for large event logs. We expect the preprocessing framework to work on larger event logs, therefore we limit the memory demands by using event identifiers. These event identifiers represents events. More on these event identifiers is discussed in the next section.

The implementation consists of two parts. First is the development of the preprocessing framework, including the support for interactive visualizations and the modification framework. This plug-in is implemented from scratch. The second part is modifying the visualizations to support the Interactive Visualization interface. The *Log Visualizer* plug-in is modified to implement the *Interactive Visualization* interface. The Pattern Abstractions plug-in is modified by the original author, *R.P. Jagadeesh Chandra Bose*. He implemented the Interactive Visualization interface. The *Dotted Chart*, we concluded in Section 4.3.3, is not suitable to be modified. This is due to the way it is implemented, some existing issues both discovered during the analysis (see Section 2.2) and known beforehand, the fact that it is ported from ProM 5.2 and the newly available features in ProM 6. Hence we create a new Dotted Chart implementation from scratch, based on a newly created *generic grid component*. This generic grid component is designed to be a foundation for grid-based visualizations.

The implementations are ProM 6 plug-ins and as such are available via the ProM Package Manager. The following packages are available:

- **LogOverview**: The preprocessing framework.
- **InteractiveVisualization**: The interactive visualization interfaces.

- **DottedChart2**: The newly implemented Dotted Chart based on the generic grid component. The generic grid component is also contained in this package.
- **LogDialog**: The Log Visualizer plug-in.
- **PatternAbstractions**: The Pattern Abstractions plug-in.

Within the implementations described above we use the OpenXES library and two third-party libraries.

- **OpenXES**
  by Eindhoven, University of Technology,
  `http://www.openxes.org/`,
  and is released under the *GNU Lesser General Public License*.
  This library is used throughout the project for the in-memory representation of the event log.
- **KDTree**
  by Simon D. Levy,
  `http://home.wlu.edu/~levys/software/kd/`,
  and is released under the *GNU Lesser General Public License*.
  This library is used within the generic grid component for efficiently selecting relevant events whenever a user is zoomed in on a particular part of the grid. This saves us the time of having to process all events when a large part is not being displayed.
- **Apache Commons: Collections**
  by The Apache Software Foundation,
  `http://commons.apache.org/collections/`,
  and is released under the *Apache Software License, Version 2.0*.
  We use the TreeList list implementation for its ability to efficiently handle insertions and deletions, as well as random access to elements.

## 5.2 Implementation decisions

Now we discuss a number of implementation decisions that were made in the interest of scalability and flexibility. First we discuss our choice of event identifiers that we used to represent events during communication between visualizations. Next we discuss the implementation details of how we accomplish communicating the event log after it has been modified using the modification framework. Finally we discuss how we manage to get event identifiers to work for an event log that changes.

### 5.2.1 Custom event identifiers: compact, derivable & direct access

First off is a design decision that is at the roots of the project. As we discussed in Section 4.4.1 we need a way to communicate what elements are selected in a visualization. We decided upon the use of the 'event' as the unit for communication. However, we cannot use the original event instances to use in our communiqués, since these events can be very large and there is no upper limit to the number of events that may be communicated either. Therefore we use event identifiers to represent the events.

There are two requirements for these identifiers. First, they should be a very compact reference to the original event, since we may need many and we do not want to spend too much memory only to store event identifiers. Second is that we should be able to, given an event identifier, quickly access the corresponding original event. This is obvious, since we are using these identifiers as a replacement for events during communication.

Currently, the ProM Framework does not officially offer anything that can function as an event identifier. Under the surface, inside the OpenXES library, there is an XID data type that is used

as an event identifier, however this identifier does not offer direct access to the corresponding event or even to the trace that contains the event. Therefore it is not a very suitable candidate. Also, this identifier was only meant for use inside the OpenXES library.

Aside from the XID, there are no identifiers available in either XES or the ProM Framework, so any viable solution should also be easily derivable from the event log, since we cannot directly ask for the identifier.

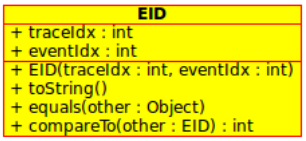


**Figure 5.1:** The EID (Event ID) class.

We finally decided upon using a combination of the index of the event in the containing trace and the index of the containing trace in the event log. Figure 5.1 shows a UML representation of the class. This combination uses only two integer values, thus has only a small footprint ($2 \times 4$ bytes $= 8$ bytes per event). Furthermore, since plug-ins have a habit of iterating through the event log during their analysis, both the index of the event in the trace, and the index of the trace in the event log are easily accessible. This makes it a very portable solution, which is important since we expect (external) visualizations to provide us this information.

The EID class implements the 'Comparable' interface. It compares respectively by trace index and event index in ascending order. This allows the EID to be used in sorted sets, hence we can quickly and efficiently handle large sets of event identifiers.

As we said earlier, there are no identifiers available. Therefore, developers of interactive visualizations have to create these event identifiers themselves in order to communicate the events for a tag. The downside to using the EID event identifier is that it is possible to create incorrect event identifiers. For example, it is possible to use integers that lie outside of the range of traces or events, such as negative values or values above the number of traces in the log or above the number of events in the trace. Furthermore, it is also good to realize that the scope of this approach is limited to a single event log. This is not a problem for this project, since we always work with one event log at a time (i.e. not counting modified versions of the original event log).

### 5.2.2 Communicating event log modifications

A second fundamental problem in this project is the problem of how to handle modifications to the event log. The design-level aspects are discussed in Section 4.4.5. We chose option 4 as the solution for this problem. This solution uses a self-modifying event log. Now we dive into the implementation of the solution.

As we have seen earlier, the ProM framework uses the XES file format for storing event logs. OpenXES implements the XES file format in Java and offers XES data structures for use in programs. ProM internally uses the XES data structures for event logs and so do the visualizations that use event logs as input.

Our goal is to implement a self-modifying event log that we can use to bring changes to the interactive visualizations. OpenXES contains the XLog and XTrace interfaces. We implement these interfaces to create our own implementations of XLog and XTrace. These implementations wrap around the original XLog and XTrace instances. Internally we create an abstraction layer that separates the modifications we make from the original event log.

There are a number of important properties for these implementations. These properties allow them to replace the original objects and function the way we want them to.
Properties of the wrappers are:

- **Separation of modification operations**: Implementations of addition and deletion methods, including related methods, allow us to keep a record of modifications that have been

made to the event log. Or to be more precise, in its current implementation we keep a record of the data that is still available, i.e. not yet deleted.

- **Equality**: The wrappers implement list equality, just as the original XLog and XTrace instances. The unmodified wrapped instances are equal to their original counterpart.

- **Clonability**: The wrappers are clonable. They implement a very specific clone requirement: the original instances are cloned, so any changes to attributes are isolated in the cloned instances. However, the record of deletions of elements, i.e. traces in case of XLog and events in case of XTrace, are shared by all clones of the wrapped implementation. This is how the event log becomes self-modifying. (*At this time we have not yet implemented adding new events, however this should be quite easy to implement.*)

- **Convertability**: The separation of the original log and the modifications that are being performed enables us to know about the indices both in the original state of the event log, and in the modified state. Therefore, we can convert event identifiers from one state to the other.

- **Generate event log**: To finalize the modifications that have been performed on the event log, we can generate a new event log, i.e. a normal event log, from the event log in its current (modified) form.

The *separation of modification operations* feature provides a way of controlling modifications. The way it is implemented also offers us insight in both the indices in the original event log and the indices in the modified state of the event log. Hence there is a way to convert event identifiers from one state to another. Additionally, it offers lots of flexibility, since we do not actually remove anything from the event log. We can literally play around with the event log and at any moment reset it to its pristine state. Which is what we need for iterative preprocessing. Furthermore, the *cloneability* feature enables us to keep event log objects of several visualizations in sync. The *equality* feature allows us to replace the original instances with wrapped instances without breaking functionality. Finally, we can *generate* an new event log based on any state of modification and export it to the ProM Framework.

### 5.2.3 How to make event identifiers work in a changing event log

In Section 4.4.1 we decided that the best way of communicating information is in the form of events. In Section 5.2.1, we decided upon the type of identifiers to use in order to express what events we are referring to. The solution for this turned out to be the use of the combination of the index of the event in the containing trace and the index of the containing trace in the event log.

Now, when we start to modify the event log, indices start to change. This renders (some of) the event identifiers incorrect and useless. So we need a way to either correct this or to adjust to these changes. As we have seen in the implementation of the XES data structure wrappers, Section 5.2.2, we have access to the indices in both the original event log and the modified event log in its current state. Therefore we can convert the indices to these two states.

The original event log is a static state, it does not change. The modified event log in its current state is dynamic. In order to make communicating tags work as it is supposed to, we store the events inside tags in the form where they use the original indices. To visualize a tag, we simply convert the event identifiers to the current state of the modified event log and pass them on to the visualizations. Similarly, whenever we request events from a visualization in order to create a tag, we know that the events we receive will be in the form of the current state of the modified event log. We convert them to event identifiers for the original event log and then store them.

## 5.3 Using the generic grid component

First we explain the process of reading the data and how the interpreters are used in this process. Afterwards we give a small example of how we apply the generic grid component in the case of the Dotted Chart.

Algorithm 1 shows the execution flow of the LogGridDataSource when the event log is being read. The algorithm concisely shows when exactly each method is being called. At the start of the parsing process 'startLog' is called, and at the start of each trace 'startTrace' is called. When a trace is finished we call 'completeTrace' and when we reach the end of the event log we call 'completeLog'. Notice that the 'getSize' methods are called only at the very last moment. Interpreters are only expected to know the final size at the very last moment, hence up to that moment the size may still vary or is allowed to be unknown.

```
input: an event log log of type XLog
These interpreters are already known within the instance, since they are provided at the
construction of the LogGridDataSource.
input: interpreters = {interpreterX, interpreterY, interpreterColor}

First inform all interpreters that we start parsing a new event log.
foreach interpreter ∈ interpreters do interpreter.startLog (log)

foreach trace ∈ log do
    Inform all interpreters that we start parsing a new trace.
    foreach interpreter ∈ interpreters do interpreter.startTrace (trace)

    For each event in the trace interpret the event and return a value.
    foreach event ∈ trace do
        interpreterX.interpret (event)
        interpreterY.interpret (event)
        interpreterColor.interpret (event)
    end

    Then inform all interpreters that we completed parsing this trace.
    foreach interpreter ∈ interpreters do interpreter.completeTrace ()
end

Inform all interpreters that we completed parsing the event log.
foreach interpreter ∈ interpreters do interpreter.completeLog ()

Finally, get the sizes of the X and Y axes from the coordinate interpreters.
interpreterX.getSize ()
interpreterY.getSize ()
```

**Algorithm 1**: The execution flow of the grid's data source parser.

The next example, Listing 5.1 shows how the generic grid component and interpreters are used to combine into the visualization component for the Dotted Chart visualization. First we create the individual interpreters. Then we create the generic grid component passing on respectively the event log, the X-axis interpreter, the Y-axis interpreter and the color interpreter in the constructor. Finally, to give the user some control over the data, we create the control panels and pass on a reference to the interpreters during their creation.

Now we can directly influence how the data is read from within the control panel. For example, using the '*setType*' method we can define which type of time we wish to see in the visualization and with the method '*setTimeUnit*' we can define the size of a single time unit, such as '*3600000*' for a time unit of 3600000 milliseconds (1 hour).

Now, when we visualize the component for the first time, it reads the data source and visualizes the dots. Whenever options have been changed, we can force the component to update the source data by invalidating the source with the method '*invalidateSource*'.

```
public static JComponent doAnalysis(UIPluginContext context, XLog log, XLogInfo
    summary) {

  //Create interpreters for the event data used in the grid visualization.
  DottedChartTimeInterpreter timeinter = new DottedChartTimeInterpreter(summary);
  DottedChartCaseInterpreter caseinter = new DottedChartCaseInterpreter();
  ColorMapInterpreter colorinter = new ColorMapInterpreter("concept:name");

  //Create Log Grid components.
  LogGridDataSource datasource = new LogGridDataSource(log, timeinter, caseinter,
      colorinter);
  LogGridComponent gridcomponent = new LogGridComponent(datasource);

  //Create the Scalable View Panel and its interaction panels.
  ScalableViewPanel viewPanel = new ScalableViewPanel(gridcomponent);
  viewPanel.addViewInteractionPanel(new DottedChartControlPanel(datasource,
      timeinter, caseinter, colorinter), SwingConstants.WEST);
  viewPanel.addViewInteractionPanel(new LegendInteractionPanel(colorinter),
      SwingConstants.WEST);
  viewPanel.addViewInteractionPanel(new InformationInteractionPanel(log),
      SwingConstants.WEST);

  return viewPanel;
}
```

Listing 5.1: An example of how the generic grid component is used to create the Dotted Chart visualization.

## 5.4 Limitations of the implementation

There are a few limitations with the current implementation of the preprocessing framework. We discuss three of these limitations and shortly explain where the problem originates. Where possible we also describe the measures taken to work around these issues.

### 5.4.1 Sorting the Event log

An unintended side effect of restricting modifications to the event log as described in Section 5.2.2, is that the sorting of the event log and individual traces is not allowed anymore. At first glance, this seems like an obvious effect of restricting modifications to the event log. However, some visualizations may rely on sorting the event data, before it is being analyzed. They make assumptions on the order in which the data is being processed and therefore we block these visualizations from functioning correctly. Furthermore, these visualizations typically do not consider reordering events in the event log as changing the event log.

However, this functionality is now blocked. By sorting we do not change the events themselves, and the same events are still contained in a trace, however the order of elements is also considered relevant in a list, therefore the event log is considered changed when traces and/or events are rearranged.

A solution for this issue is proposed in the section on future work. We propose an extension to this separation between visualizing the event log and modifying the event log, in which we allow visualizations to propose modification plug-ins to the modification framework. Section 7.2.3 explains this proposal in further detail.

### 5.4.2 Performance of the XES data structures

This project is founded on the XES data structures that are provided in the OpenXES library. The performance of operations such as iterating through the contents of the event log are largely dependent on the performance of the XES data structures. For example, the parsing process described in Algorithm 1 was profiled using a java profiler and it turns out that over 60% of the time

used to regenerate the event buffer is spent within operations of the XES data types. Therefore, we expect that any optimizations to these data types can result in significant performance gains for these and any other implementations that make (heavy) use of these data types.

### 5.4.3 Performance of the wrapped filter structures

As we explained in Section 5.2.2, we use an intermediate list to catch modification that are made to the event log and record these changes. However, there is a performance penalty involved with this abstraction. The performance penalty for this is most clearly visible when a set of events is converted from original event log to the current state of the event log, since we have to look up the event indices in the current state of the event log that correspond to event indices in the original event log.

We manage to contain the problem by buffering events once they are converted to the current state of the event log. Future requests can therefore be accelerated, since the conversion is not required anymore. However, these buffers are only valid as long as the event log is not modified. After modifications have been made, the caches are being cleared since they are out of date.

We, however, still consider the wrapped XES data structures to be the best approach to solving the challenge of communicating changes. This approach adds little complexity to existing visualizations that have to be modified, it is applicable to all types of visualizations and the underlying structure offers a solution for the problem of handling event identifiers in a changing event log (this was discussed in Section 5.2.3). Furthermore, we can be absolutely sure that visualizations are in complete synchronicity with regard to the source event data. Section 4.4.5 explains the chosen solution for communicating event log modifications.

## 5.5 Conclusions

In Section 5.1, we discussed the implementation of the preprocessing framework and the modification of the interactive visualizations. We discussed in detail what exactly has been (re)implemented, what has been modified and by whom. We also explain where the implementations can be found. Next, in Section 5.2, we discussed some of the implementation details of this project. The first detail, that was discussed in Section 5.2.1, is the custom event identifier that is used within this project. The second detail is the communication of event log changes. This is discussed in Section 5.2.2. And the third detail, in Section 5.2.3, is how to work with these event identifiers in a changing environment. In Section 5.3, we discussed how the generic grid component can be used for various grid visualizations, and an example is given of how we use the generic grid component to implement the Dotted Chart. We also discussed the execution flow for reading an event log into the generic grid component. Afterwards, in Section 5.4, we discussed some of the limitations that exist in the current implementation of the preprocessing framework.

# Chapter 6

# Conclusions

In this project we improved support for preprocessing event logs.

In Chapter 1 we introduced the concepts of process mining and preprocessing. We studied some references on process analysis and preprocessing in literature. We discovered there that process analysis and more specifically on preprocessing is largely considered to be a linear process. The goals that are described in the literature are identical, however the method lacks the *flexibility* that is often required in a practical setting. It turns out that in practice it is very much an *iterative process*, as opposed to linear.

Then we continued with the investigation of available visualizations that can be used for preprocessing. In Chapter 2 we investigated a number of visualizations and observed their behavior, performance and features and compiled a rating consisting of 4 properties: *visual aspects*, *support for interaction*, *performance aspects* and *predictability of results*. We then compared the ratings of all the visualizations in order to get an idea of how different types of visualizations relate to one another. We looked at differences and similarities, and observed some *issues related to the visualizations and/or preprocessing*.

In the next chapter, Chapter 3, we looked at preprocessing support in the ProM Framework and related these observations to the earlier work on preprocessing in the literature and properties of the available visualizations. From this we concluded that, besides the literature, also the tools support preprocessing mostly as a linear process. We also concluded that preprocessing as well as process analysis could be improved by *extending visualization capabilities*. Furthermore, preprocessing could be improved by *interactively modifying the event log*. Using these results as well as results from the previous chapters, we defined a set of requirements for a preprocessing framework and interactive visualizations.

In Chapter 4, we designed a plan to *introduce iterative preprocessing* to the ProM Framework in the form of a plug-in that provides a **Preprocessing Framework**. We looked at *two important featuresets* that should be available in this preprocessing framework: **Interactive Visualizations** that can communicate visualization elements, and a **Modification Framework** that manages modifications that are made to the event log. Then we selected three visualizations that each provide a unique view of the event log. These plug-ins are selected as **reference implementations** for the preprocessing framework. The information provided by the interactive visualizations can then be used in preprocessing. We determined how they could be modified such that they can be plugged into the preprocessing framework. Of one plug-in, *the Dotted Chart*, we determined that, in its current state, it is not suitable to be integrated. Therefore we designed a plan to *reimplement and improve* this visualization and integrate it into the preprocessing framework.

Finally, in Chapter 5 we implemented the **Preprocessing Framework** according to the design. We also modified the chosen **visualizations** such that they support the features of the preprocessing framework. And we *reimplemented the Dotted Chart* visualization according to the design.

The preprocessing framework and the interactive visualizations, open up new possibilities in the area of preprocessing. Before, we needed to manually look up points of interest in every visu-

alization separately, now we can simply tag this point of interest in one visualization and recall the tag in another visualization. We can now perform modifications to the event log and visualizations are able to adjust automatically. In short, the preprocessing framework takes preprocessing to a new level by taking care of the visualizations, so we only have to think of what we want to see. And it takes care of modifying the event log, such that we only have to think of how we wish to change the event log.

# Chapter 7

# Future Work

Aside from the work that we have done in this project, there are some ideas for improving on the preprocessing framework and extending the underlying concepts. We have categorized these ideas in three categories:

1. **Improving usability**: Here we explore some of the smaller improvements that should help in perfecting the preprocessing framework.
2. **Extending on the concept**: Here we explore some of the ideas that extend on the current concept of the preprocessing framework. These ideas are more comprehensive.
3. **Further supporting the analyst**: Aside from the preprocessing framework, there are some other ideas that might improve on the task of analyzing the event log.

## 7.1 Improving usability

### 7.1.1 Enabling/disabling interactive visualizations

The framework is greatly supported by more visualizations, however simultaneously loading all of these visualizations can result in high memory requirements. To ease off on the memory requirements, we should be able to enable and disable interactive visualizations at will. Additionally, we should be able to define a user preference, such that we can define a default set of interactive visualizations that will be loaded at the start. Other interactive visualization can be enabled at runtime. This would also solve potential problems with CPU-intensive or high memory demand visualizations and large event logs.

Furthermore, the current preprocessing framework is still a rough implementation. Progress indication would improve the user friendliness at those moments when the event log is modified, the interactive visualizations are updated or the event identifiers are converted. Although it would most likely only be necessary for the larger operations.

Also, when switching event log states, the new state is currently computed from the initial state of the event log. If we keep a record of intermediate states, we can speed up switching event log states. This is especially advantageous if we frequently switch event log states during preprocessing.

### 7.1.2 ProM metadata

The ProM Framework offers facilities to store metadata on provided objects, i.e. the imported data and plug-in execution results. We should link the preprocessing result to the source event log. Furthermore, we could keep a record of the executed modification plug-ins and their order of execution. With this information we should be able to keep a historical record on the types of modifications that have been performed on the event log during its preprocessing phase. We

could also use the same information to load an event log including its history in the preprocessing framework. These capabilities help with reproducing earlier results and understand the steps taken to come to these results.

## 7.2 Extending on the concept

### 7.2.1 Extend the FilterLog and FilterTrace wrappers with support for adding (new) events

The FilterLog and FilterTrace implementations in their current state cannot handle the addition of events to the (wrapped) event log. It should be fairly easy to add support for adding events and traces. With this support we can not only filter the event log, but also add missing events.

### 7.2.2 Modification mode for FilterLog and FilterTrace wrappers

To better control the modification of the event log and traces, we should add a possibility to change modes. A read-only mode, the mode that the instances are normally running under, would prevent modifications from being made. Whenever the modification framework needs to perform any modifications, we switch to a mode where writing is allowed. The modification framework can then perform its required modifications and afterwards switch back to read-only mode. Upon switching back to read-only mode, we could automatically check the state of the event log and remove, for example, any emptry traces in order to conform to the XES Standard. This would ensure that the event log is always in a correct state after modifications have been performed.

### 7.2.3 Extend the modification framework with support for cooperating with interactive visualizations

One of the topics of an earlier ProM meeting was about the impression that visualizations seem to move from a "read-only visualization" towards an "editor". The concern is that, similar to the state in ProM 5.2, plug-ins will do all kinds of operations. The result is that the borders begin to fade and every plug-in, analysis plug-in or visualization, becomes an "editor".

A possible solution for this issue is to adopt the approach we take with the preprocessing framework. We make a clear distinction between *visualization* and *modification*. Of course, we cannot simply ban modifications entirely, since these are an essential part of preprocessing and the process analysis itself. Therefore, we define an interface for visualizations. Visualizations can then "propose" a modification plug-in to the modification framework. The modification framework can insert this modification plug-in at the correct position, i.e. we cannot simply add the plug-in to the end if the event log is currently in some intermediate state, and perform the requested modification.

This would be an approach that separates the concern of visualization from the concern of modification, allowing the framework to better control the state of the event log. This does not mean, however, that the visualization loses all control. A visualization should, for example, be able to query the framework for the modification plug-ins that have been applied, in order to ensure that the input into the visualization is acceptable.

### 7.2.4 Event Pool, different case definitions & cross-event log preprocessing and process analysis

During the analysis of the Dotted Chart visualization, we remarked that there are two options to 'change the perspective' of the visualization. One option would actually change the case definition of the event log. We see a possibility for making the existing plug-ins more flexible and more powerful by allowing us to easily change the case definition.

In the current process, we choose a case definition as early as at the time we extract the event log data from the data source, such as a database. The chosen case definition, typically based on the notion of a 'customer' or 'job', is used throughout the entire process analysis process. However, we might be interested in more than just the information on a 'customer' or 'job'. We can, for example, take an interest in the employees of process. An attempt has already been made towards these alternative case definitions, since we are able to select component types other than 'cases' in the Dotted Chart.

An approach that we could take to provide a more dynamic method to case definitions is to first collect events in an 'event pool'. Here we already extract the events, however we do not yet partition them into cases according to *one* case defintion. There are some challenges related to this approach, since an event can sometimes have more than one occurrence in an event log, depending on the choice of case definition. However, we suspect there is always a possibility to extract an atomic event that we only need to use once.

If this event pool is realised and events in the event pool can be uniquely identified, we can extend the features of the preprocessing framework to function across multiple event logs as long as they contain events from a common event pool. When this is realised we can not only switch between different views and perspectives of the event log, but we can even switch between various case definitions of the event log.

## 7.3 Further supporting the analyst

### 7.3.1 Other candidates suitable as interactive visualizations

#### Fuzzy Miner

The Fuzzy Miner is analyzed in Section 2.10. This plug-in is able to cope with large event logs. The Fuzzy Miner is especially useful in the preprocessing framework, since real life event logs typically require most preprocessing and thus benefit the most from the Fuzzy Miner's features. The Fuzzy Miner can discover and visualize the process flow and allows for flexibility in its visualization such the process analyst can learn to understand the underlying process.

#### Social Network Miner

The Social Network Miner, Section 2.13 is another potential candidate for the preprocessing framework. The Social Network Miner mines and visualizes the organizational perspective of the event log. With the focus on people, it helps to visualize the organizational aspects of the underlying process.

#### Basic visualizations

One of the observations that was made in the chapter on Visualization Analysis, Chapter 2, was that there are hardly any visualizations that can be used to visualize basic, raw event log data. The Dotted Chart is one example of such a visualization. It does not aggregate or transform this data, instead it simply visualizes it in a grid. A first attempt has been made to provide functionality for basic visualizations. The *generic grid visualization* is a component that should, at least in theory, be able to be used for various types of grid visualizations. Several other types of such visualizations can be thought of, such as a Matrix visualization that can visualize two classes of data in a (textual) matrix, similar to how the generic grid does this in a graphical way. Another example of such a basic visualization is a plug-in that can provide various kinds of charts.

### 7.3.2 Improving the generic grid component

In this project we developed a generic grid component that can be used for grid-based visualizations. The generic grid component is used for the first time in the Dotted Chart. This

implementation uses mostly common functionality such as the support for zooming into any area and to select grid elements.

To make this generic grid component more flexible, we need to add support for common event handlers such as an event handler that responds whenever the mouse moves/enters/exits to another grid cell, and an action listener that informs the listeners whenever the user clicks in a certain grid cell.

These event handlers open up possibilities for "personalizing" the grid component for more specific functions that might be required by future grid visualizations.

Furthermore, it might be interesting to extend the CoordinateInterpreter interface with a way for coordinate interpreters to express their desired minimum grid cell size. The Dotted Chart is a visualization that depends on the ability to infinitely scale in order to always visualize itself completely. However, a visualization such as, for example, the Trace Alignment plug-in requires the opposite behavior, i.e. it needs to have a minimum grid cell size enforced. Hence this should also be possible in the generic grid component.

# Bibliography


[1] W. M. P. van der Aalst, A. J. M. M. Weijter, and L. Maruster. Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 16:2004, 2003. 19

[2] Wil M. P. van der Aalst, Hajo A. Reijers, and Minseok Song. Discovering social networks from event logs. *Computer Supported Cooperative Work (CSCW)*, 14:549–593, 2005. 10.1007/s10606-005-9005-9. 27

[3] Melike Bozkaya, Joost Gabriels, and Jan Martijn van der Werf. Process diagnostics: A method based on process mining. *Information, Process, and Knowledge Management, International Conference on*, 0:22–27, 2009. 2, 4, 8

[4] B. F. van Dongen. A meta model for process mining data. In *Proceedings of the CAiSE WORKSHOPS*, pages 309–320, 2005. 1, 67

[5] Christian W. Günther. XES standard definition, revision 1, 2009. 4, 46

[6] Christian W. Günther and Wil M. P. van der Aalst. Fuzzy mining - adaptive process simplification based on multi-perspective metrics. In *BPM*, pages 328–343, 2007. 22

[7] A. K. A. de Medeiros, B. F. van Dongen, W. M. P. van der Aalst, and A. J. M. M. Weijters. Process mining: Extending the $\alpha$-algorithm to mine short loops. In *Eindhoven University of Technology, Eindhoven*, 2004. 19

[8] Jagadeesh Chandra Bose R. P. and W. M. P. van der Aalst. Abstractions in process mining: A taxonomy of patterns. In *BPM*, volume 5701 of *Lecture Notes in Computer Science*, pages 159–175, 2009. 14, 42

[9] Jagadeesh Chandra Bose R. P. and W. M. P. van der Aalst. Trace clustering based on conserved patterns: Towards achieving better process models. In *Business Process Management Workshops*, volume 43 of *Lecture Notes in Business Information Processing*, pages 170–181, 2009. 14

[10] Jagadeesh Chandra Bose R. P. and W. M. P. van der Aalst. Trace alignment in process mining: Opportunities for process diagnostics. In *BPM*, volume 6336 of *Lecture Notes in Computer Science*, pages 227–242, 2010. 17

[11] P. Riemers. Process improvement in healthcare: A data-based method using a combination of process mining and visual analytics. Master's thesis, Eindhoven University of Technology, 2009. xi, 12, 13, 16, 79, 80, 81

[12] M. S. Song and W. M. P. van der Aalst. Supporting process mining by showing events at a glance. *Information Technologies and Systems (WITS'07), Annual Workshop on*, 17, 2007. 8, 42

[13] M. S. Song and W. M. P. van der Aalst. Towards comprehensive support for organizational mining. *Decision Support Systems*, 46(1):300–317, 2008. 27

[14] H. M. W. Verbeek, J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst. XES, XESame, and ProM 6. *Information Systems Evolution*, 72:6075, 2011. 3, 4, 9, 41

[15] A. J. M. M. Weijters and A. K. A. de Medeiros. Process mining with the heuristicsminer algorithm, 2006. 20

# Appendix A

# Visualizer Analyses

## A.1 Analysis of visualizations

To get a clear picture of what tools are available for use in preprocessing an event log, we first look at the available plug-ins and visualizations. The focus here is on the actual *visualization* of the information and its usability. We do not look at the analysis or the data prepared prior to the visualization. We do, however, take performance into account. We also look at some of the mining algorithms whose process model visualization can support us with a first impression of the event log that we are preprocessing.

The target for this analysis is to find what information is shown in each visualization and how it is visualized. We do this to get an idea of what information is available to us as a source of information to base our preprocessing decisions on.

The visualizations have been analyzed on the following properties:

1. **Primary Focus**: The level of detail of the data on which the visualization focuses. For example, aggregated data like averages that may nicely describe a process as a whole, do not provide any event-specific information. Available levels are:
   - *Process*: Looking at the log on the level of a process in its entirety.
   - *Traces*: Looking at the log on the level of individual traces.
   - *Events*: Looking at the log on the level of individual events.
2. **Perspective**: The perspective from which a plug-in visualizes the event log. There is a close correspondence between the visualized perspective and from which XES extension data is used for the visualization of the information.

   Available perspectives are:
   - *General*: General information about the process, like 'number of occurrences', both absolute and relative, 'number of event classes', etc. Note that the 'General' perspective will mainly look at core XES information, i.e. existence and numbers of events and traces. Other data attributes are (largely) ignored.
   - *Control Flow*: Visualization of the control flow of the process. Depending on the details required for the visualization, a plug-in can choose to only make use of the 'concept' extension. It would mean that only the classes of events are of importance. For an additional level of detail, a plug-in can use the 'lifecycle' extension, which also indicates in which activity lifecycle state an event is: scheduled, started, completed, or another state.[1] Plug-ins that focus on control flow are largely dependent on the 'concept' and 'lifecycle' extensions for their data.

[1] See [4] for more information on the event lifecycle.

- *Performance*: The visualization of performance information of the log. Using the time data from the log, we can analyze the execution of cases and use the time data to analyze the duration of tasks. We can look at information like the average execution time of a case, or an event class, and find outliers. These plug-ins rely heavily on data from the 'time' extension.
- *Resource*: For the analysis and visualization of organizational information stored in the log. Using organizational information, such as the resource, role or group that initiates an event, we can derive how the process could have been organized and which resources might have worked together. The plug-ins use data from the 'organizational' extension.

3. **Use Case**: A typical use case for this visualization.

4. **Data types**: Types of data that are used in the visualization. Examples of these data types are: traces, events, extensions such as 'concept' and 'organizational', classifiers, etc.

5. **Advantages**: Advantages of this particular visualization.

6. **Disadvantages**: Disadvantages of this particular visualization.

7. **Information discovery**: Concrete types of information that can be discovered using this visualization.

8. **Initial loading times**: The time it takes to load the visualization for 3 different logs. These times are used as a benchmark to compare visualization loading times. Most of the visualizations provide 2 numbers. The first number is the analysis time beforehand, the second is the actual time necessary to load the visualization.

   The times have been measured by hand, since we are looking for the waiting time for users. Keep in mind that these measurements are not meant as benchmarks for the different plug-ins, but rather as a very rough estimate of the differences in analysis times. We are not interested in millisecond differences, but rather in whether a plug-in takes less than a second, a few seconds, a few minutes, or hours to analyze/visualize.

   The tests have been performed on a notebook computer with a Core 2 T7200 (2.0 GHz) CPU and 1 GB of memory dedicated to the JVM during execution. The operating system is Ubuntu Linux and we have used Sun Java Runtime Environment 6. The ProM 5.2 plug-ins were tested on Windows XP, since there were issues with user interface on the Ubuntu Linux operating system.

   We have used 3 event logs to test the initial loading times on. The vent log called 'repair example' is an event log that has been available for quite a while. It is generated using a simulation and therefore is really consistent and complete. Given that this log is quite small, every plug-in should be able to handle this event log without a problem. The event log consists of 1104 cases and 11855 events. The second log, 'bouwvergunning', is a real-life event log from a municipality. This log is quite a bit larger than the repair example, with 2076 cases and 67271 events. And the third log is 'rws', which is another real-life event log. This log contains 14279 cases and 119021 events. The 'rws' event log is quite large and makes for a good test case for the feasibility of a visualization.

9. **Expected limitations**: Limitations that can be expected with the use of this visualization, especially with regard to the interactivity of the visualization.

The data of the visualization analyses have been written in a brief and direct style, and are only meant as an intermediate format in the evaluation process. Also notice that these evaluations are **not** comparisons of one visualization to another.

## A.2 Log Visualizer

### A.2.1 Dashboard

1. **Source**: ProM 6
2. **Primary Focus**: Process
3. **Perspective**: General
4. **Use Case**: Totals and aggregated information on the process level.
5. **Data types**: traces, events, concept, lifecycle, resource, time
6. **Advantages**:
   - Good for size estimation.
   - Useful for first impression of the event log.
7. **Disadvantages**:
   - Only useful for first impression: data too aggregated for any in-depth analysis.
8. **Information discovery**:
   (a) numbers for a whole log: totals, minima, maxima, mean, time span
9. **Interactivity**: No interactivity available, only displays information.
10. **Initial loading times**: The 'Log Visualizer' plug-in does not require analysis. Loading times are only for generating the visualization.
    - **Repair example**: 3 seconds
    - **Bouwvergunning**: 10 seconds
    - **RWS**: 7 seconds
11. **Expected limitations**: None.

### A.2.2 Inspector

1. **Source**: ProM 6
2. **Primary Focus**: Process, traces, events
3. **Perspective**: General
4. **Use Case**: User interface for reading through the raw log data.
5. **Data types**: traces, events, concept, lifecycle, resource, time
6. **Advantages**:
   - Ability to look up (almost) any information in the log.
7. **Disadvantages**:
   - No overview of the data.
   - No hints to interesting data.
   - Cannot show nested attributes in traces and events.
8. **Information discovery**:

(a) Raw data.
(b) Frequency of occurrence (in the Explorer).
(c) Length of traces.

9. **Interactivity**:
   (a) Select trace to view information about: corresponding events, trace attributes.
   (b) Select event to view information about: event attributes.
10. **Initial loading times**: The 'Log Visualizer' plug-in does not require analysis. Loading times are only for generating the visualization.
    - **Repair example**: 3 seconds
    - **Bouwvergunning**: 10 seconds
    - **RWS**: 7 seconds
11. **Expected limitations**: None.

### A.2.3 Summary

1. **Source**: ProM 6
2. **Primary Focus**: Process
3. **Perspective**: General
4. **Use Case**: Textual, aggregated overview of the log:
   (a) for each of the defined classifiers: (*the classifier defines a unique event class*)
      - occurring element.
      - start element.
      - end element.
5. **Data types**: events, classifiers, concept, lifecycle, resource
6. **Advantages**:
   - Summary of the process with focus on occurring events. (absolute + relative numbers)
7. **Disadvantages**:
   - Lots of text, requires interpretation.
   - The anomalies that can be detected in the summary are not always obvious.
8. **Information discovery**:
   (a) Event occurrences.
   (b) (Potential) start/end events.
   (c) Resource occurrences.
   (d) Resources executing start/end elements.
   (e) Distribution of elements.
9. **Interactivity**: None.
10. **Initial loading times**: The 'Log Visualizer' plug-in does not require analysis. Loading times are only for generating the visualization.

- **Repair example**: 3 seconds
- **Bouwvergunning**: 10 seconds
- **RWS**: 7 seconds

11. **Expected limitations**: None.

## A.3 Dotted Chart

1. **Source**: ProM 6
2. **Primary Focus**: Traces, events.
3. **Perspective**: Control flow + resource + performance
4. **Use Case**: Plotting of (all) events on the screen as a raw visual overview. Used for spotting exceptions, anomalies, patterns (existing or broken).
5. **Data types**: Combinations of several types of information:
   - traces with events, time
   - event classes with events, time
   - resources with events, time
   - lifecycle with events, time

   Secondary: concept, lifecycle, resource, time
6. **Advantages**:
   - Graphical overview: easier to spot patterns (as opposed to textual overview).
   - Ability to find patterns that are hard to quantify and thus hard to find with numerical analysis.
7. **Disadvantages**:
   - Graphical overview is less accurate.
   - Current implementation has overlapping dots.
   - Dot may increase in size, hence increasing the overlapping dot problem.
   - Only large anomalies visible. (Either large time span or patterns that create a large shape.)
   - Tool tips are unsuitable as the way to display the amount of information that needs to be shown.
   - Tool tips show additional data from the event log, but the types of data shown are hard coded, so it does not take into account new XES extensions.
   - Sidebar shows information that is not used very frequently.
8. **Information discovery**:
   (a) Vertical repetitions (close together on the x-axis which represents time)
   (b) Horizontal repetitions
   (c) Gaps
   (d) Increase/decrease in case arrivals.
   (e) Instantiation of event classes.

(f) Steady repetition of series of events.

(g) Unexpected or sudden changes of property. (e.g. change of color)

9. **Interactivity**:

   (a) Selection

   (b) Zooming

   (c) Dragging

10. **Initial loading times**:

    - Original Dotted Chart:
      - **Repair example**: 7 seconds, 2 seconds.
      - **Bouwvergunning**: 60 seconds, 5 seconds.
      - **RWS**: out-of-memory
    - Original Dotted Chart (tweaked):
      - **Repair example**: 2 seconds, 4 seconds.
      - **Bouwvergunning**: 4 seconds, 10 seconds.
      - **RWS**: 5 seconds, 37 seconds.

11. **Expected limitations**:

    - Number of dots (often representing events) slows down the visualization.
    - High memory usage.
    - Overlapping dots impact precision of the visualization.

## A.4 Heuristics Miner

1. **Source**: ProM 6
2. **Primary Focus**: Process, traces.
3. **Perspective**: Control flow.
4. **Use Case**: Discover a process model using traces from the event log.
5. **Data types**: traces, events, concept, lifecycle.
6. **Advantages**:

   - Handles noise.
   - Gives a good (first) impression.
   - Helps with the identification of exceptions.
   - Clean interface, large area for graph.

7. **Disadvantages**:

   - (Relatively) little information about the model itself.
   - No information on which traces flow through a task or over an edge.
   - Numbers near the edge are vague and do not sum up as expected at first sight.

8. **Information discovery**:

   (a) Flow of cases through the mined model.

(b) Frequency of occurrence of an event.

(c) Distribution of cases over the model. (E.g. do 90% of the case follow a strict path, or is the amount evenly divided over a number of paths.)

9. **Interactivity**:

   (a) Select/move vertices.

   (b) Select/move edges.

10. **Initial loading times**:

    - **Repair example**: 4 seconds, 1 second.
    - **Bouwvergunning**: 10 seconds, 11 seconds.
    - **RWS**: 8 seconds, 1 second.

11. **Expected limitations**:

    (a) Since there is no support for clustering events or abstracting from events, the mined models might become too large to maintain usability for event logs that contain many events and/or contain many different cases.

## A.5 Trace Alignment

1. **Source**: ProM 6
2. **Primary Focus**: traces, event classes
3. **Perspective**: Control flow
4. **Use Case**: Creates a general picture of all traces by aligning their events in a grid.
5. **Data types**: traces, events, concept, lifecycle
6. **Advantages**:

   - Structured visualization: events do not get obscured.
   - Clustering (optional clustering options)
   - Has its own filtering capabilities.

7. **Disadvantages**:

   - Many (too) technical options for a user who is not familiar with the plug-in.
   - Has its own filtering capabilities.
   - Clustering is on by default. (i.e. an initial setting of 1 big cluster would be more interesting such that we can first familiarize ourselves with the event log, before splitting it up.)
   - Positioning of events. (Sometimes traces seem more different than they are in reality because events are misaligned.)
   - Computationally very heavy.
   - Event logs with a large variety in traces can result in a very big and sparse grid, thus making it difficult to perform analysis without having generated several clusters.

8. **Information discovery**:

   (a) Suspect parallel behavior.

(b) Repetitions/loops.

(c) "Common positions in the grid": patterns can be found whenever a large part of the traces executes the same activities at the same time. Pattern breaks are visualized as "holes" in the grid.

(d) Discover exception behavior by looking for columns with only very few events and columns missing only very few events.

9. **Interactivity**:

   (a) Sorting.

   (b) Filtering.

   (c) Clustering.

   (d) Trace editing.

   (e) Delete all-gap columns.

10. **Initial loading times**: The way the Trace Alignment plug-in works, it does not do analysis before visualization. The visualization starts with a user interface and does several computation steps upon selection of these options. Hence there is only one time recorded per event log.

    - **Repair example**: 4 seconds.
    - **Bouwvergunning**: 2880 seconds (+/- 500).
    - **RWS**: 47 seconds.

11. **Expected limitations**: Depending on the complexity of the event log, especially when there are many long traces, one could lose the overview of the event log, since the trace cannot be completely shown on the screen.

## A.6 Fuzzy Miner

1. **Source**: ProM 6
2. **Primary Focus**: Process, traces, events
3. **Perspective**: Control flow, performance
4. **Use Case**: Generate a fuzzy model of the process.
5. **Data types**: traces, events, concept, lifecycle, time
6. **Advantages**:

   - Fuzzy Model:
     (a) Support for clustering. (Enables the user to create a comprehensible overview by clustering events in the cases where the unclustered model is too large to comprehend.)
     (b) Size + color of arcs provide clear notion of ratio of occurrence.
   - Fuzzy Model Animation:
     (a) Good view of workload/performance over time.

7. **Disadvantages**:

   - No additional information available.
   - No information on other data, like resources.

8. **Information discovery**:
   (a) The process model according to the event log (except for changes where abstraction and aggregation is concerned).
9. **Interactivity**:
   (a) Moving elements.
10. **Initial loading times**: An additional time ($3^{rd}$ number) has been measured for generating the Fuzzy Model Animation.
    - **Repair example**: 4 seconds, 1 second, 3 seconds.
    - **Bouwvergunning**: 28 seconds, 12 seconds, 16 seconds.
    - **RWS**: 23 seconds, 6 seconds, 112 seconds.
11. **Expected limitations**:
    (a) Large number of events stresses the memory usage.

## A.7 Petri Net Visualizer

1. **Source**: ProM 6
2. **Primary Focus**: Process
3. **Perspective**: Control flow
4. **Use Case**: Petri net model represents the control flow of a process. Possibility to derive facts and anomalies from the way the model is constructed and the suggested flow directions.
5. **Data types**: traces, events, concept, lifecycle
6. **Advantages**:
   - Clean visualization.
   - Control flow with clear semantics. (Further analysis possible.)
7. **Disadvantages**:
   - No additional information available.
   - The strict semantics enforce a level of detail that makes the model too complex.
8. **Information discovery**:
   (a) Preceding and succeeding events and possible splits and joins.
9. **Interactivity**:
   (a) Select elements.
   (b) Move elements.
10. **Initial loading times**:
    - **Repair example**: 2 seconds, 3 seconds
    - **Bouwvergunning**: 3 seconds, 105 seconds
    - **RWS**: 2 seconds, 16 seconds
11. **Expected limitations**:
    (a) Complex workflows represented in Petri net format grow enormously in complexity because of the low level of the Petri net formalism.

## A.8 Social Network Miner

1. **Source**: ProM 6
2. **Primary Focus**: Process
3. **Perspective**: Resource
4. **Use Case**: Displays a social network of connected resources. The edges represent the relation of the chosen type: handover of work, reassignment, subcontracting, similar tasks, working together.
5. **Data types**: traces, events, resource
6. **Advantages**:
   - Support for highlighting the nodes adjacent to the selected node.
7. **Disadvantages**:
   - No other information (than resource) available.
   - Layout is random.
   - Many resources make the visualization hard to visualize and/or interpret.
   - Whenever the number of nodes increases, you quickly loose overview.
   - When there are sufficiently many relations, they use their usefulness because they are indistinguishable from each other.
8. **Information discovery**:
   (a) Relations between resources.
9. **Interactivity**:
   (a) Selecting.
   (b) Highlighting.
   (c) Zooming.
10. **Initial loading times**: *Handover of Work*:
    - **Repair example**: 3 seconds, 1 second
    - **Bouwvergunning**: 8 seconds, 5 seconds
    - **RWS**: 7 seconds, 4 seconds

    *Reassignment*:
    - **Repair example**: 2 seconds, 1 second
    - **Bouwvergunning**: 6 seconds, 2 seconds
    - **RWS**: 7 seconds, 2 seconds

    *Subcontracting*:
    - **Repair example**: 2 seconds, 1 second
    - **Bouwvergunning**: 9 seconds, 2 seconds
    - **RWS**: 8 seconds, 2 seconds

    *Similar tasks*:

- **Repair example**: 3 seconds, 2 seconds
- **Bouwvergunning**: 6 seconds, 8 seconds
- **RWS**: 9 seconds, 87 seconds

*Working together*:

- **Repair example**: 1 second, 3 seconds
- **Bouwvergunning**: 7 seconds, 6 seconds
- **RWS**: 7 seconds, 9 seconds

11. **Expected limitations**: Whenever there are many nodes in the visualization, the real-time layout adjustment slows down the whole application so much that it is virtually unresponsive.

## A.9 Pattern Abstractions

1. **Source**: ProM 6
2. **Primary Focus**: Traces, events
3. **Perspective**: Control flow
4. **Use Case**: Find loop patterns and repeating patterns in event logs.
5. **Data types**: traces, events, concept, lifecycle
6. **Advantages**:
   - Find patterns, i.e. loops and repeats, that may normally not be found because of the vast amount of data and limitations of visualizations.
   - Given the simple visualization, the severity of the patterns is clearly visible.
7. **Disadvantages**:
   - No control over the used event classifier.
     - It might be interesting to search using other classifiers. (E.g. based on organizational data or lifecycle transitions.)
   - Complex UI, technical terms.
   - The embedded pattern visualization is not suitable for viewing the pattern in its context, i.e. the event log.
   - Traces sometimes contain noise from events that were executed in parallel.
   - No interaction in the embedded pattern visualization.
8. **Information discovery**:
   (a) Loop patterns
   (b) Maximal repeat patterns
9. **Interactivity**:
   (a) Select/control the search options.
   (b) Transforming the log (merging found patterns and exporting the transformed log)
10. **Initial loading times**: This plug-in does everything within the visualization phase. The 3 timings are measurements for the three subsequent steps executed in the plug-in:

- **Repair example**: 6 seconds, 2 seconds, 1 second
- **Bouwvergunning**: 424 seconds, 14 seconds, 27 seconds
- **RWS**: 178 seconds, 2 seconds, 1 second

11. **Expected limitations**: Finding patterns can take a long time for large logs.

## A.10 Role Hierarchy Miner

1. **Source**: ProM 5.2
2. **Primary Focus**: Process, events
3. **Perspective**: Resource
4. **Use Case**: Find out, with a given abstraction level, what groups of resources do similar work, thus have the same role, for a certain set of tasks.
5. **Data types**: events, classifiers (either concept or concept + lifecycle), resource
6. **Advantages**:
   - The O/T matrix gives a good global indication of how work is divided over the available resources or for the selected activity.
7. **Disadvantages**: None.
8. **Information discovery**:
   (a) Role hierarchy of users. Discover which users execute the same tasks and how many times have been recorded per tasks.
   (b) Distinction between Specialist and Generalist by the way the nodes are interconnected.
9. **Interactivity**:
   (a) Zooming.
   (b) Selection. (get information on the number of times a user executes a set of events.)
   (c) Configuration of the threshold.
   (d) Showing/hiding tasks from hierarchy graph.
10. **Initial loading times**:
    - **Repair example**: 3 seconds
    - **Bouwvergunning**: 11 seconds
    - **RWS**: 3 seconds
11. **Expected limitations**: Whenever there are many nodes and relatively little hierarchy, the diagram becomes completely unreadable, since you cannot at the same time read the texts and follow the connections.
12. **Note**: This plug-in is not available in ProM 6.

## A.11 Basic Performance Analysis

1. **Source**: ProM 5.2
2. **Primary Focus**: Process, trace
3. **Perspective**: Performance
4. **Use Case**: Visualization of performance for tasks. (task processing time, instance throughput time)
5. **Data types**: traces, events, concept, lifecycle, resource, time
6. **Advantages**:
   - Lots of options with regard to the visualization of the data.
7. **Disadvantages**:
   - Static implementation: Only possible to select tasks and resources, no other data types.
   - Chart labels are cut off when they become too long. The remainder that is used as the label can become ambiguous.
8. **Information discovery**:
   (a) Waiting/processing times for events/tasks/resources. (Basically for classifiers.)
9. **Interactivity**:
   (a) Zooming. (on Y-axis)
10. **Initial loading times**: For this plug-in, there is only a single analysis step. This is the step that has been timed.
    - **Repair example**: 3 seconds
    - **Bouwvergunning**: 52 seconds
    - **RWS**: out-of-memory
11. **Expected limitations**:
12. **Note**:
    - This plug-in is not available in ProM 6.
    - This type of functionality should be implemented on top of a generic charting plug-in. The types of information that are basically presets of type of chart + data class.

## A.12 Logical and Multi-Set Views

1. **Source**: Based on the visualization used in [11] pages 30, 31, and 35.
2. **Primary Focus**: Traces, events.
3. **Perspective**: General
4. **Use Case**: Detecting patterns in logical view of traces and length of trace representation.
5. **Data types**: traces, events, concept, lifecycle, resource, time
6. **Advantages**:

- Simple view.
- Semantically independent. (Can process any data class.)
- With centered event, a new way of aligning.

7. **Disadvantages**:
   - No semanic support: the user has to correctly interpret the provided visualization.
8. **Information discovery**:
   (a) Traces: similar traces.
   (b) Events: depends on the chosen data and options, identifiers could be:
      - length (of trace/event)
      - patterns (subsequent events/event classes)
      - recurring group of events
      - missing expected patterns
9. **Interactivity**:
   (a) Selecting events.
   (b) Setting/unsetting event as 'centered' event.
   (c) Sort on length, data, events before/after selected event.
   (d) Zooming.
10. **Initial loading times**: Not applicable. (Visualization has not yet been implemented.)
11. **Expected limitations**: Not applicable.
12. **Notes**:
    - No implementation available yet.
    - 'Logical sequence' and 'Multi-Set' variations are possible.
    - W/ or w/o centered event.
    - W/ or w/o grouping similar traces.
    - The 'Logical' view is similar to the Trace Alignment plug-in, except for the way the elements are aligned. The resulting visualization is significantly different.

## A.13 Event classes with preceding and succeeding event classes

1. **Source**: Based on the visualization used in [11] page 34.
2. **Primary Focus**: Events (or some other individual element)
3. **Perspective**: General
4. **Use Case**: Finding most common subsequent events relative to the selected event.
5. **Data types**: events, concept, lifecycle, resource, time
6. **Advantages**:
   - Good local overview for given event.
   - Visual representation of ratios of preceding and succeeding elements.

7. **Disadvantages**:
   - Does not provide a clear overview.
8. **Information discovery**:
   (a) Preceding/succeeding classes with ratio.
9. **Interactivity**:
   (a) Select an element (centered event, preceding/succeeding elements)
   (b) Select an event class to center.
   (c) Limit the number of visible cases. (E.g. do not show elements ¡ 10% or group them.)
10. **Initial loading times**: Not applicable.
11. **Expected limitations**: Not applicable.
12. **Notes**:
    - No implementation available yet.
    - In case of data other than event class, it may be necessary to do aggregation before it is possible to show a useful diagram.

## A.14 Standard charts

1. **Source**: Derived from some of the charts presented in [11], chapter 5: 'Process analysis'.
2. **Primary Focus**: Data (per trace or event)
3. **Perspective**: General
4. **Use Case**: Show available data in graphical format (useful for visual analysis).
5. **Data types**: traces, events, concept, lifecycle, resource, time
6. **Advantages**:
   - Graphical representation of numerical information.
   - Applicable on any data class.
7. **Disadvantages**:
   - No semantic support: the user has to correctly interpret the given chart.
   - No knowledge of available data.
   - No information on usefulness of semantics of data.
8. **Information discovery**: None. (No definitive answer possible, depends on the type of data that is visualized.)
9. **Interactivity**:
   (a) None.
10. **Initial loading times**: Not applicable.
11. **Expected limitations**: Not applicable.
12. **Notes**:
    - No implementation available yet.