

MASTER

Performance analysis of business processes from event logs and given process models

Adriansyah, A.

Award date:
2009

[Link to publication](#)

Disclaimer

This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Departement of Mathematics and Computer Science

Performance Analysis of Business Processes from Event Logs and Given Process Models

Arya Adriansyah

Supervisor: prof.dr.ir. W.M.P van der Aalst

Tutor: dr.ir. Boudewijn van Dongen

Examination Committee: dr.ir. H.M.M. van de Wetering

Eindhoven, August 2009

Dedicated to my parents, sister, and a very special friend

Abstract

The goal of performance analysis of business processes is to gain insights into operational processes, for the purpose of optimizing them. To intuitively show which parts of the process might be improved, performance analysis results can be projected onto process models. This way, bottlenecks can quickly be identified and resolved. Unfortunately, existing approaches to project performance information onto process models are limited to models at relatively low levels of abstraction. Given complex processes with many activities and complex case routings, useful insights are hardly obtained from classical process models. Too many details are shown at once, thus making it impossible to see the global picture.

In this thesis, we investigate an approach to project performance information which is obtained from event logs onto models at any level of abstraction. Based on an analysis of existing process models, we propose a process model which can represent processes intuitively, regardless of their complexity. Given such a model and an event log of a process, we propose an approach to calculate performance information of the process. The obtained information is then projected onto the process model such that performance insights can be obtained intuitively. To evaluate the performance and demonstrate the applicability of our approach, we have implemented it in the ProM framework¹ and tested it using various real-life event logs.

Keywords: business process performance analysis, process model abstraction, performance projection.

¹<http://www.processmining.org>

Executive Summary

Problem Definition

A process model plays a crucial role in business process performance measurement. It does not only provide the necessary information about the way activities are performed, but it can also provide additional insights whenever performance information is projected onto it. Process models with projected performance information can show where exactly bottleneck activities lie, which activities may be affected by them, and which activities cause them.

Exploration of commercial process monitoring tools by Hornix (2007) shows that to obtain such performance representation, a model of the process is required. Constructing a process model is a time-consuming activity. In the AIS group of the Mathematics and Computer Science department of TU/e, the use of event logs to obtain insights into business processes is investigated. Some of the research results are process discovery algorithms which enable process models at different levels of activity abstraction to be constructed from event logs. Figure 1 illustrates several process models of a single process, each with a different level of activity abstraction. Process models with a higher level of abstraction provide less details compared to the models with a lower level of abstraction.

Process model abstraction helps process owners to obtain significant information from a process model based on the detailed model specification. Unfortunately, currently available approaches to measure and visualize performance information are limited to models at a relatively low level of abstraction. In case the process consists of many activities, the model shows too many details such that useful insights are hard to be found.

Research Objective

The described problem leads to the main research objective of this thesis:

Develop an approach to calculate performance information from both a given event log and a given process model at any level of abstraction, and project the information onto the model.

The projected performance information should be informative and intuitive. It has to provide some useful insights into business processes and should be understandable by process owners. The approach needs to be robust, i.e. it produces useful performance information and useful insights regardless of the complexity of the business processes.

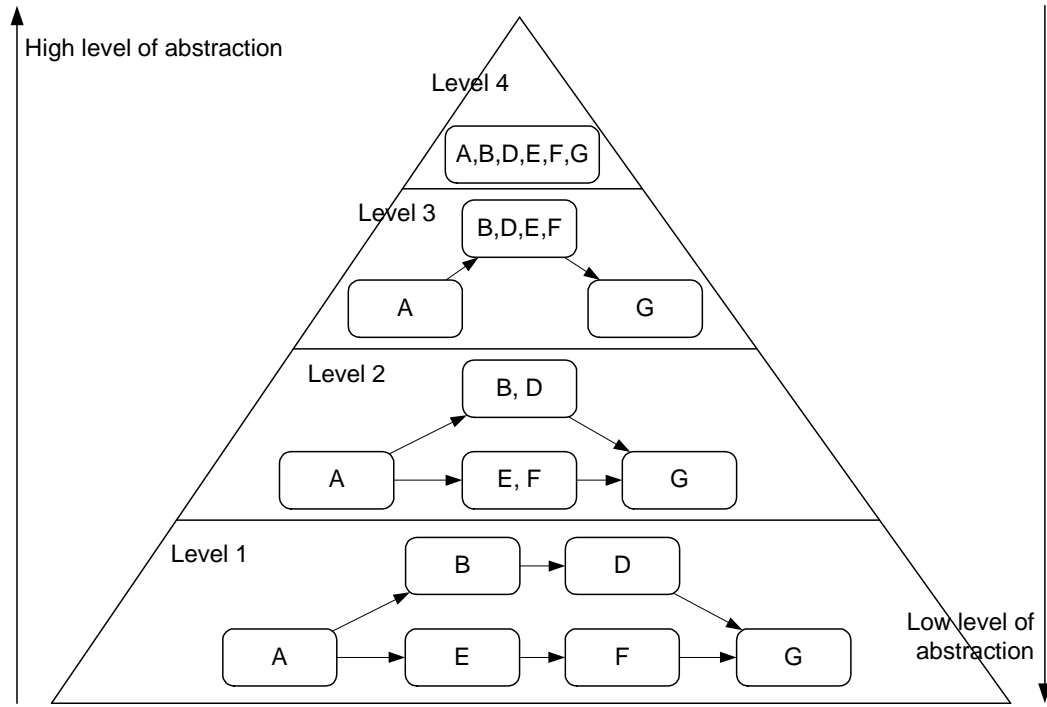


Figure 1: Several process models of a single process, each at a different level of abstraction

Research questions

The research objective resulted in the following main research question:

Given a process model at any level of abstraction and an event log, how can we calculate performance information and project it onto the model?

This main question can be divided into three sub-questions:

1. *What kind of process models can present a process intuitively, regardless of the complexity of the process?*
2. *Given an event log and an instance of the model, how can performance information be calculated?*
3. *How can the information be projected onto the model?*

Research Methodology

To satisfy the research objective and answer the research questions, the following steps are taken:

1. Conduct a literature study on business process performance analysis and process mining. Investigate performance metrics on business processes and identify which metrics can be measured from event logs.
2. Analyze intuitive process models, especially the ones which are potentially suitable to project performance information onto.
3. Develop a conceptual process model such that important performance information can be projected onto it intuitively.

4. Develop an approach to calculate performance information based on the conceptual process model.
5. Develop a process model to project the performance information onto.
6. Implement the approach as plugins in ProM². In this thesis, the new version of ProM (ProM 2008) is used as an implementation platform rather than the latest released version of ProM (ProM 5).
7. Evaluate the implemented methods using both simulated and real life event logs. Analyze insights into processes which are gained from the constructed models.

Results

In this master thesis, several existing process models have been investigated and their strong and weak points have been analyzed. In addition, process models which are currently used in commercial tools have been explored. Based on the analysis results, only process models that can *abstract activities* (hiding activities such that they do not appear in the models) and *aggregate activities* (presenting several activities in a node) are able to *present processes intuitively*, regardless of their complexity. In addition, they need to have relaxed semantics. Thus, *Simple Precedence Diagrams* (SPDs) have been introduced as process models which support these features. Three main approaches to obtain SPDs also have been defined: converting existing process models to SPDs, discovering SPDs from event logs, and creating SPDs manually.

To ensure that a substantial number of performance metrics was taken into account, various useful KPIs for business process analysis were investigated. To measure the KPIs, both academic performance measurement approaches and commercial performance measurement approaches (from several leading commercial tools) were analyzed. Based on these approaches, a log replay approach to calculate performance information based on an event log and an SPD has been proposed to address the second research question.

Finally, to address the last research question, two process models are proposed: *Fuzzy Performance Diagrams* (FPDs) and *Aggregated Activities Performance Diagrams* (AAPDs) (see Figure 2 and Figure 3). The former is basically an SPD with performance information projected onto it and the latter is a model which aggregates activity instances in rectangular elements and shows the performance of each element with respect to one focus element. With both models, performance information can be presented in an intuitive manner.

As a proof of concept, the log replay and all supporting modules have been implemented as ProM plugins. The implemented plugins have been evaluated against various real-life event logs, as well as simulated event logs. Evaluation of the plugins showed that performance information of processes can be obtained easily from the proposed models, regardless of their complexity. Nevertheless, knowledge about the processes and the motivation behind the construction of the process models is essential to interpret the models properly. The implemented plugins also worked

²<http://www.processmining.org>

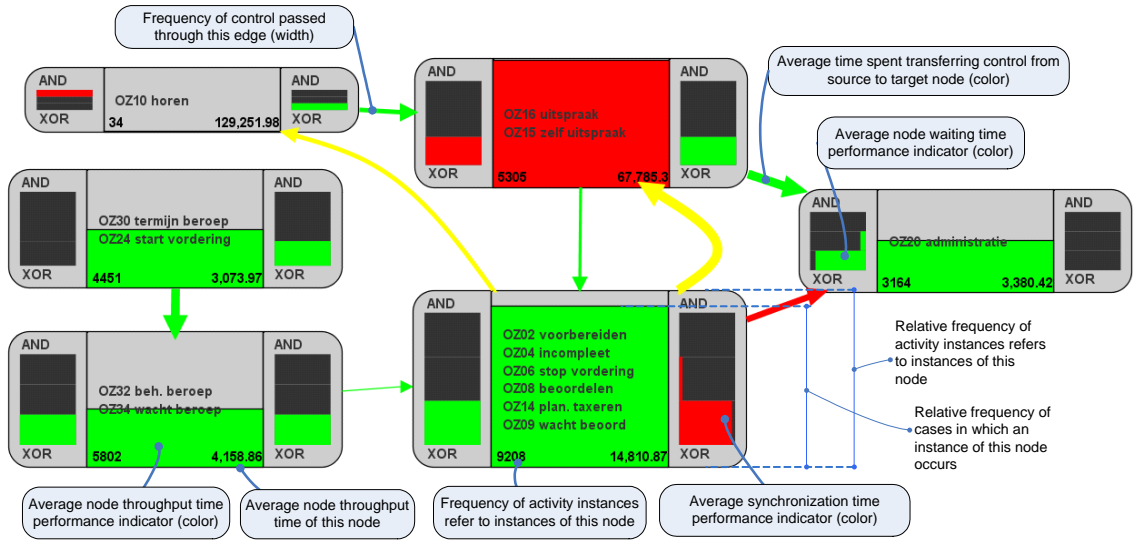


Figure 2: Example of an FPD of a large event log

arguably fast even when dealing with large real life event logs which represent complex processes. Despite its robustness, the log replay approach still leaves room for improvement. Depending on the complexity of the event logs, the approach is sensitive to a look-ahead value and requires a lot of memory. Further investigation is needed to make it more robust.

Conclusion

Complex business processes can only be shown intuitively by process models which support both activity abstraction and aggregation, and have relaxed semantics. The SPD is a model which satisfies these criteria. Therefore, it is able to present even complex processes intuitively. However, the drawback of process models which satisfy these criteria is their inability to describe control flow precisely. Thus, to extract performance information using such models, it is necessary to have a heuristic approach which exploits all available information to determine the control flow of the process. In this thesis, we show that with a simple approach using a look-ahead value and precedence information from SPDs, the control flow of processes can be determined so that performance of the processes can be calculated.

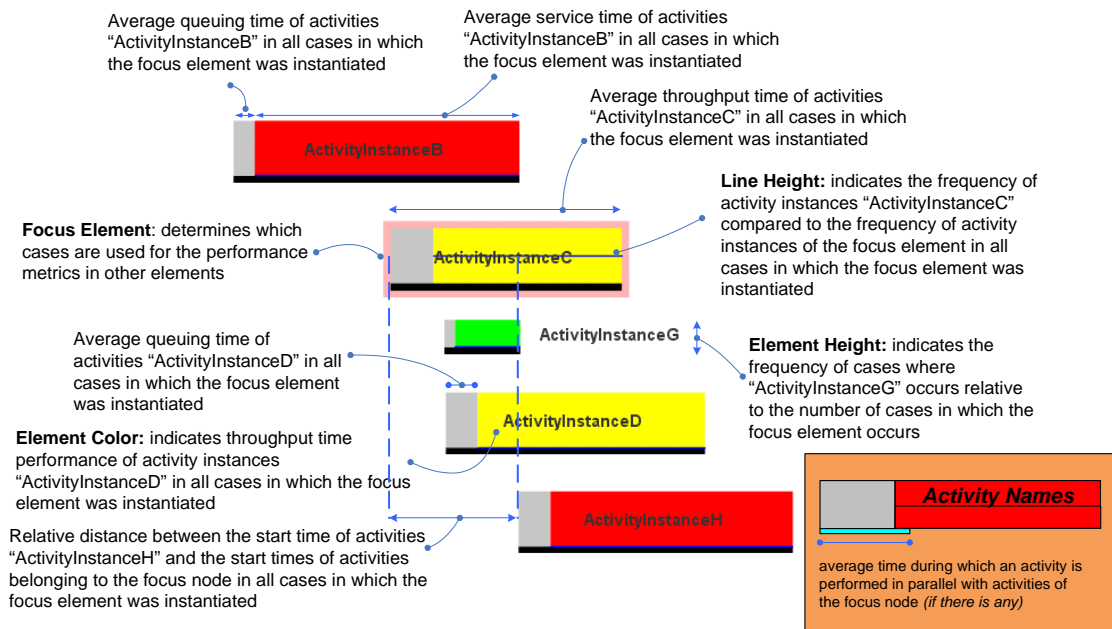


Figure 3: Example of an AAPD of a large event log

Preface

This master thesis is the result of my graduation project which completes my Computer Science and Engineering study at Eindhoven University of Technology. The project is an internal project conducted in the Architecture of Information System Group, Mathematics and Computer Science Department of Eindhoven University of Technology.

The topic of the graduation project suits the courses in the master phase and also suits my personal interest. Therefore, I enjoy working on the project a lot. I find the project is challenging, as I have to think outside of the box and come up with new idea. Throughout the project, I also learned a lot of valuable things.

I would like to thank several people for their support during my graduation project. First, I would like to thank my supervisor Wil van der Aalst for introducing me to such a wonderful project and guiding me throughout the project. I would like to thank for his trust, support, feedbacks, and pleasant cooperation. Second, I would thank my tutor Boudewijn van Dongen for his daily guidance throughout the project. He has become such a great tutor, being more like a close friend than just a tutor who guide me throughout the project. I believe that not many people are willing and capable enough to give constructive feedbacks and ideas as much as them both. I would also thank Huub van de Wetering as one of the members of the examination committee.

Then, I would thank my mother Meydia Darmawan and my father Darmawan Daud for their continuous support, pray, and guidance during both good and bad time. Also thanks to my sister Astrid Hapsari Ningrum who always manage to cheered me up when I was down. Thank you for my special friend, Elva Fitriani, who kept my spirit alive along the way. Huge thanks to my best friends: Merli, Chitra, and Rendy who always have faith in me and helped me gain my mind back during those critical times. Many thanks to Samuel, Goran, Agni, and Jimmy for their moral supports along the way. And of course, many thanks for all of my relatives, friends, and other people that I can not mention in detail for the supports they gave.

Arya Adriansyah
August 2009

Contents

1	Introduction	1
1.1	Thesis Context	1
1.2	Problem Description	3
1.3	Research Objective and Questions	4
1.4	Research Scope	5
1.5	Research Methodology	5
1.6	Outline	5
2	Preliminaries	6
2.1	Business Process Performance Analysis	6
2.2	Process Mining	10
2.2.1	Event Logs	11
2.2.2	Process Discovery	13
2.2.3	The ProM Framework	16
2.3	Performance Analysis through Process Mining	17
2.4	Performance Analysis in Commercial Tools	20
3	Modeling Processes	23
3.1	Intuitive Process Models	23
3.2	Obtaining SPDs	26
3.2.1	Converting Process Models to SPDs	26
3.2.2	Mining SPDs from Event Logs	29
3.2.3	Creating a Hand Made SPD	32
4	Measuring Performance	34
4.1	Overview	34
4.2	Node Sequence Identification	35
4.3	Node Instance Identification	39
4.4	Control Flow Identification	40
4.5	Activity Instances Identification	45
4.6	Key Performance Indicators	46
4.6.1	Case-Level KPIs	48
4.6.2	Process-model-related KPIs	49
4.6.2.1	SPD-Node-related KPIs	49
4.6.2.2	Edge-related KPIs	51
4.6.2.3	Two-nodes analysis	52
4.6.2.4	Aggregated-activities KPIs	52

5	Performance Projection	56
5.1	Fuzzy Performance Diagram (FPD)	56
5.2	Aggregated Activities Performance Diagram (AAPD)	60
6	Implementation	63
6.1	Plugins Overview	63
6.2	SPD Plugins	64
6.3	Performance Measurement Plugins	65
6.3.1	Event Log Replay Plug-in	66
6.3.2	FPD Visualization Plug-in	67
6.3.3	AAPD Visualization Plug-in	69
6.3.4	Global Settings Visualization Plug-in	74
7	Evaluation	76
7.1	Node and Semantics Identification	76
7.2	Real-life Log Analysis	78
7.3	Performance Evaluation	83
8	Conclusion and Recommendation	87
	Bibliography	89
	Appendix	93
A	KPI Formalization	93
A.1	Case-level KPIs	93
A.2	Process-model-related KPIs	95
A.2.1	SPD-Node-related KPIs	95
A.2.2	Edge-related KPIs	96
A.2.3	Two-nodes analysis	97
B	Implementation Design	99
B.1	SPD Plug-in	99
B.2	Performance Measurement	101
B.2.1	Models to Project Performance Information	101
B.2.2	Log Replay Plug-in	103
B.2.3	Performance Information Visualization	105
C	User Manual	109
C.1	SPD Miner Plug-in	109
C.1.1	Introduction	109
C.1.2	Using SPD Miner Plug-in	109
C.2	SPD Visualization Plug-in	110
C.2.1	Introduction	110
C.2.2	How to Use	111
C.2.2.1	Visualize SPD	111
C.2.2.2	Mapping SPD nodes to activities	112
C.3	Event Log Replay Plug-in	113

C.3.1	Introduction	113
C.3.2	How to Use	113
C.4	FPD Visualization	114
C.4.1	Introduction	114
C.4.2	Performance Information	116
C.4.3	How to Use	120
C.5	AAPD Visualization	121
C.5.1	Introduction	121
C.5.2	Performance Information	122
C.5.3	How to Use	123
C.6	Global Setting GUI	124
C.6.1	Introduction	124
C.6.2	How to Use	125
D	Evaluation	127
D.1	Semantics Identification Evaluation	127
D.1.1	Purpose	127
D.1.2	Procedure	127
D.1.3	Result	129
D.2	Multi-level of Abstraction Evaluation	129
D.2.1	Purpose	129
D.2.2	Procedure	129
D.2.3	Result	129

List of Figures

1	Several process models of a single process, each at a different level of abstraction	v
2	Example of an FPD of a large event log	vii
3	Example of an AAPD of a large event log	viii
1.1	Bar Chart Example	2
1.2	Performance Indicator in an Extended Petri net	2
1.3	Several process models of a single process, each with different level of abstractions	4
2.1	A three dimensional view of a workflow [14, 44]	7
2.2	Transactional model for activities [47]	8
2.3	Performance measures - time dimension	9
2.4	MXML format of a process log [47]	13
2.5	Standard transactional model in this thesis	13
2.6	Example of heuristic model from a case [40]	15
2.7	An excerpt of a Fuzzy model [26]	15
2.8	The new ProM architecture	17
2.9	Performance information in a Petri net process model [16]	18
2.10	Example of a traces of events [25]	19
2.11	Log animation in a Fuzzy model	20
2.12	Example of performance dashboard in the Software AG's WebMethods [11]	21
2.13	Charts and tables to show performance information in the Metastorm BPM [3]	22
2.14	Speedometer to indicate resources workload to handle on-boarding new clients in the Metastorm BPM [3]	22
3.1	Fuzzy model to show sequence pattern from A to B	24
3.2	Petri net to show sequence pattern from A to B	24
3.3	Process model with two possible traces	25
3.4	Fuzzy model of process described in Figure 3.3	25
3.5	An example Petri net (1)	28
3.6	The GPM of the Petri net in Figure 3.5	28
3.7	The SPD of the Petri net in Figure 3.5	28
3.8	An example of a Fuzzy model and a GPM/an SPD which is constructed from the model	29
3.9	An example Petri net (2)	30

3.10	An example of a hand made SPD that is created based on limited information about the process (1)	32
3.11	An example of a hand made SPD that is created based on limited information about the process (2)	32
3.12	An example of a hand made SPD that is created based on limited information about the process (3)	33
3.13	An example of hand made SPD that is created based on limited information about processes (4)	33
4.1	Overview of the approach to measure performance	35
4.2	SPD for example	37
4.3	Sequence of events 1 as case example	38
4.4	Sequence of SPD nodes for the sequence of events in Figure 4.3	38
4.5	Sequence of events 2 as case example	39
4.6	Transformation of the sequence of events in Figure 4.3 during log replay	41
4.7	Control flow identification of node instances in Figure 4.6	42
4.8	Control flow identification of node instances in Example 2	42
4.9	Split semantics identification example	44
4.10	Join semantics identification example	45
4.11	Example of activity instances inside a node instance	46
4.12	Example of constructed activity instances	47
4.13	Different insights into a process from different level of abstraction	47
4.14	Example of two node instances in the same case	54
5.1	Example of an FPD	56
5.2	Example of an FPD node (1)	57
5.3	Example of an FPD node (2)	57
5.4	Control flow indication in FPD edges (See also Figure 5.3)	58
5.5	Activity mapping in an example SPD	59
5.6	Performance and control flow information in an example FPD	59
5.7	Example of an AAPD	61
6.1	The new ProM architecture	64
6.2	SPD Visualization	65
6.3	FPD visualization	67
6.4	Using Two Nodes Performance panel provided by FPD visualization plug-in	68
6.5	AAPD visualization	69
6.6	Example of comparison between AAPD with horizontal distance in linear scale and AAPD with horizontal distance in logarithmic scale	71
6.7	Example of adjustment to AAPD horizontal scaling value	72
6.8	Example of adjustment to AAPD element scaling value	73
6.9	Example of adjustment to horizontal scaling value and element scaling value in AAPD shown in Figure 6.6b	73
6.10	Example of a detailed statistical measurement	74
6.11	Interface to set the values in objects of class <code>GlobalSettingsData</code>	74

7.1	Petri net for evaluation purpose	76
7.2	SPD of Petri net in Figure 7.1	77
7.3	Petri net 1 for evaluation	77
7.4	Petri net 2 for evaluation	78
7.5	Prediction of AND-join semantics of node G which refers to transition G of the Petri net in Figure 7.4 with several look-ahead window values	79
7.6	The constructed FPD of “bezwaar WOZ” log with 18 nodes	80
7.7	The constructed AAPD of “bezwaar WOZ” log with 18 elements	80
7.8	The constructed SPD of “bezwaar WOZ” log with 5 nodes	81
7.9	The constructed FPD of “bezwaar WOZ” from SPD in Figure 7.8	82
7.10	AAPD of “bezwaar WOZ” log with a default settings (all scaling has a zero value)	83
7.11	AAPD of “bezwaar WOZ” log with horizontal scaling adjustment	83
7.12	AAPD of “bezwaar WOZ” log with horizontal scaling, width scaling, and height scaling adjustment	84
7.13	Performance degradation with increasing number of clusters (nodes)	86
B.1	SPD class design	100
B.2	Classes to map nodes in an SPD to activities in an event log	100
B.3	Screenshot of SPDEditorPanel	101
B.4	FPD class design	102
B.5	AAPD class design	103
B.6	Design of classes to perform log replay	103
B.7	Classes to store the result of log replay	105
B.8	FPD visualization	106
B.9	FPD visualization class design	106
B.10	Example of textual information	107
B.11	Example of table information	107
B.12	AAPD visualization	108
B.13	AAPD visualization class design	108
C.1	Using SPD Miner	110
C.2	Dialog which ask for the number of SPD clusters	110
C.3	Progress bar that indicates that the SPD Miner plug-in is processing	110
C.4	SPD Visualization	111
C.5	How to use SPD Visualization plug-in	111
C.6	Error message if there is no event log which is mapped to the selected SPD	111
C.7	The first step of mapping SPD nodes to activity	112
C.8	Popup window after all nodes are mapped	112
C.9	How to use the Event Log Replay Plug-in	113
C.10	Dialog to determine look-ahead value	114
C.11	Dialog to adjust the value of maximum state space in the search of maximum fitting subtraces	114
C.12	Event log replay progress bar	114
C.13	Output objects of log replay plug-in	115
C.14	FPD visualization	115

C.15 The Case-level KPIs panel	116
C.16 Example display of Node-related KPIs panel	117
C.17 Example display of Edge-related KPIs panel	117
C.18 Example display of the Two Nodes Performance panel	119
C.19 How to use the FPD visualization plug-in	120
C.20 Display of boxes to adjust the boundary of performance color	120
C.21 Dialog window to modify the value of node throughput time performance boundary	121
C.22 Example of a selected radio button beside the “ Select source node ” label	121
C.23 AAPD visualization	122
C.24 How to use the AAPD visualization plug-in	123
C.25 The three sliders to adjust AAPD visualization	123
C.26 Boxes to adjust the performance color of AAPD element	124
C.27 Global settings object visualization	124
C.28 Example of a detailed statistical performance table	125
C.29 How to use Global setting visualization plug-in	125
C.30 Popup window after Global settings object is successfully modified	126
D.1 Petri net 1 for evaluation	127
D.2 Petri net 2 for evaluation	128
D.3 Decomposition of each transition in both Figure D.1 and Figure D.2	128
D.4 SPD of the Petri net in Figure D.1	128
D.5 SPD of the Petri net in Figure D.2	129
D.6 SPDs of the Petri net in Figure D.1, each with a different level of activity abstraction	130

List of Tables

6.1	Throughput time table	68
7.1	Traces for evaluation purpose	77
7.2	Node identification evaluation	78
7.3	Metadata of testing event logs	84
7.4	Performance of replay log plug-in (time unit is given in seconds) . . .	85
7.5	Performance of replay log plug-in per case per event (time unit is given in microsecond/ 10^{-6} second)	85

Chapter 1

Introduction

This master thesis is the result of the graduation project for Computer Science and Engineering master study at Eindhoven University of Technology (TU/e). The project is carried within the Architecture of Information Systems (AIS) group of the Mathematics and Computer Science department of TU/e. Throughout this thesis, we investigate the problem of projecting business process performance information onto given process models such that insights can be obtained intuitively, regardless of the complexity of the processes being considered. Within this master project, a solution has been realized using the Process Mining (ProM) framework¹, an open-source framework which is mainly developed by the AIS group to support the implementation of process and log related techniques.

In Section 1.1, the context of this master thesis is explained. Then, the problems which are tackled in this thesis are explained in Section 1.2. Section 1.3 gives the research objectives and the research questions for this thesis. The remaining three sections of this chapter provide the scope of this thesis, the research methodology, and the outline for this master thesis report, respectively.

1.1 Thesis Context

Business process performance analysis has already been an issue for management since the 1980s. In the early 1980s, Total Quality Management (TQM) was introduced as one of the earliest approaches which consider process improvement as an integral part to improve organizations' overall performance and quality. Then, Business Process Re-engineering (BPR) was introduced in the early 1990s to improve organizations' performance by revolutionary process improvements [42]. However, only since the beginning of 2002, significant attention has been paid to Business Process Management (BPM) [31]. Now, it can be argued that BPM is the most important topic on the management agenda [31]. A recent study of Gartner in early 2009 also supports this opinion as BPM has been one of the CIO's top business priorities for the past five years [39].

A business process can be defined as a set of coordinated activities in an organizational and technical environment to realize a business goal [49]. Each business

¹<http://www.processmining.org>

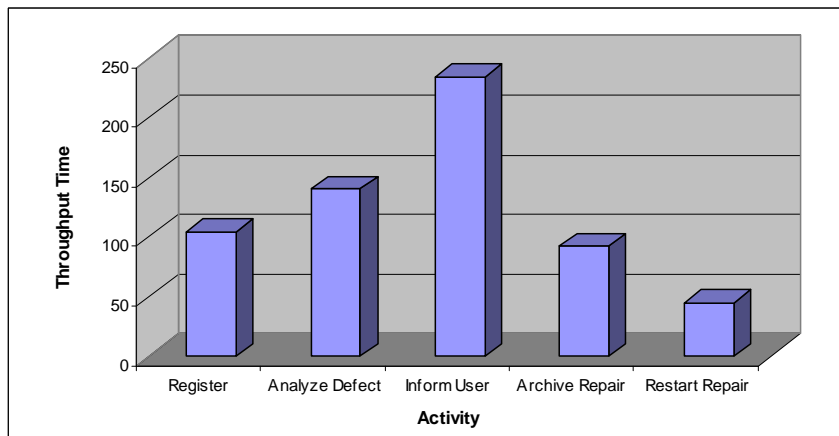


Figure 1.1: Bar Chart Example

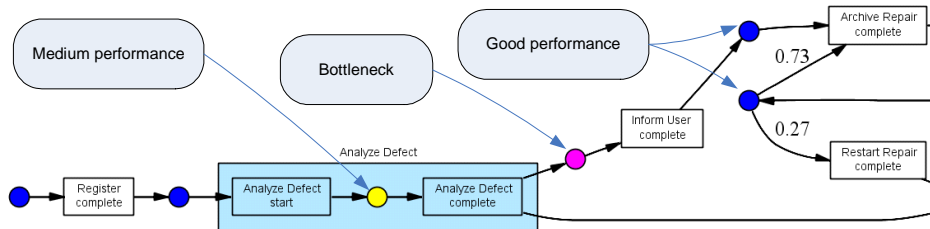


Figure 1.2: Performance Indicator in an Extended Petri net

process can be enacted by a single or multiple organizations. Many case studies show that well-designed business processes lead to improvements and cost saving, while badly designed business processes lead to errors and resource inefficiencies [24, 30, 32, 35].

In this thesis, we limit our scope to *business process performance analysis*. The goal of business process performance analysis is to gain insights into operational processes. With these insights, processes can be improved and optimized. Typically, the performance of a process can be calculated from the process' event log which contains information on which activities have been performed by whom and for which case. The information is presented to the process owners in a simple and intuitive form such that useful insights into the process can be obtained easily. Different presentation forms may provide different insights into the same process. For instance, the bar chart in Figure 1.1 provides insights into activities' throughput time. It is easy to see which activity takes the most time to be finished, and which other activity takes approximately the same time to be finished. If the same information is projected onto a Petri net [28] (see Figure 1.2), another view on the bottleneck activities is achieved, providing new insights into the process. From Figure 1.2, a bottleneck can be easily identified between the "Analyze Defect complete" activity and the "Inform User complete" activity. However, unlike the bar chart in Figure 1.1, no comparison between the activities' throughput times can be obtained from the representation in Figure 1.2.

A process model is an essential element in business process performance measurement. Not only that it provides the necessary information about the way activities

are performed, but it can also provide additional insights whenever performance information is projected onto it. Process models with projected performance information can show where exactly bottleneck activities lie, which activities may be affected by them, and which activities cause them. An exploration of commercial process monitoring tools in [28] showed that in order to obtain such performance presentation for a process, a process model of the process is required. Currently, there are two common approaches to obtain such model. In the first approach, the model is obtained directly from the process owners, e.g. process owners should draw the process model according to their own view of the process. The second approach is to retrieve the model from a Workflow Management System (WFMS).

Compared to the first approach, the second approach has several drawbacks. First, it cannot be used by any organization which does not have any WFMS. Second, as the model is often also an executable specification of the process, there is a one-to-one correspondence between activities in the model and activities that are relevant for performance measurement. Thus, suppose that the process consists of many activities, the model shows all details such that useful insights are hard to be obtained. Finally, an executable process specification may not coincide with the process owners' view on the process. Hence, the process owners may not be able to analyze the performance presentation of the process according to their needs.

1.2 Problem Description

In the AIS group of the Mathematics and Computer Science department, TU/e, research is conducted to use event logs and process mining techniques to obtain insights into business processes. Some of the research results are process discovery algorithms which enable process models at different levels of activity abstraction to be constructed from event logs. Abstraction is generalization that reduces the undesired details in order to retain only information relevant for a particular task [38]. One of the most well-known examples of abstraction can be observed in cartography, where geographical maps visualize landscapes on different scales. While a map of a particular town provides detailed information on houses and side streets, the world map only captures shapes of continents, main river contours, and marks locations of the largest cities [38].

Figure 1.3 illustrates several process models of a single process, each at a different level of activity abstraction. Process models with a higher level of abstraction provide less details compared to the models at the lower level of abstraction. In the figure, the process model with the label “Level 4” has the highest level of activity abstraction as it aggregates all activities within a process in a single node. In the same figure, the process model with the label “Level 1” has the lowest level of activity abstraction as it shows all individual activities.

Process model abstraction helps process owners to obtain significant information from a process model based on the detailed model specification. Unfortunately, the currently available approaches to project performance information onto process models are limited to models at relatively low levels of abstraction. In this thesis, we investigate the approach to project performance information onto models on any level of abstraction.

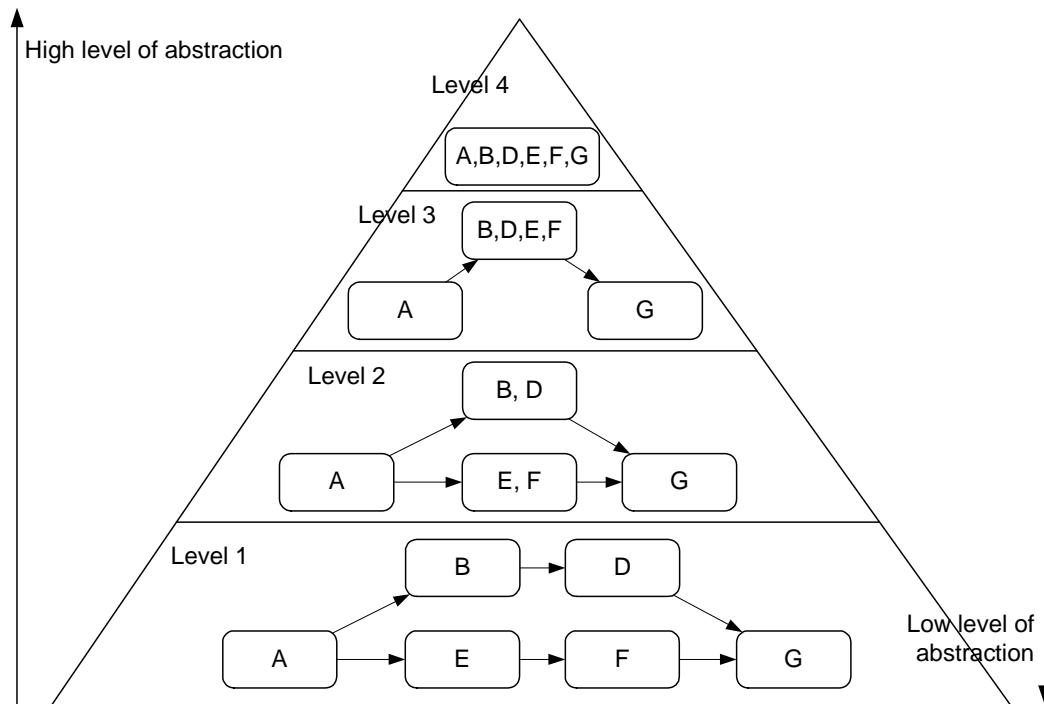


Figure 1.3: Several process models of a single process, each with different level of abstractions

1.3 Research Objective and Questions

The problem described in Section 1.2 resulted in the following research objective:

To develop an approach to calculate performance information from both a given event log and a given process model at any level of abstraction, and project the information onto the model.

The projected performance information should be informative and intuitive. It has to provide some useful insights into business processes and should be understandable by process owners. The approach needs to be robust, i.e. it produces useful performance information and useful insights regardless of the complexity of the business processes.

Research questions

The research objective resulted in the following main research question:

Given a process model at any level of abstraction and an event log, how can we calculate performance information and project it onto the model?

This main question can be divided into three sub-questions:

1. *What kind of process models can present a process intuitively, regardless of the complexity of the process?*
2. *Given an event log and an instance of the model, how can performance information be calculated?*
3. *How can the information be projected onto the model?*

1.4 Research Scope

In this thesis, only solutions that are based on process mining techniques are considered. As process mining techniques rely on the existence of event logs, only performance information that can be retrieved from event logs is within the scope of this thesis.

1.5 Research Methodology

To satisfy the research objective and answer the research questions, the following steps are taken:

1. Conduct a literature study on business process performance analysis and process mining. Investigate performance metrics on business processes and identify which metrics can be measured from event logs.
2. Analyze intuitive process models, especially the ones which are potentially suitable to project performance information onto.
3. Develop a conceptual process model such that important performance information can be projected onto it intuitively.
4. Develop an approach to calculate performance information based on the conceptual process model.
5. Develop a process model to project the performance information onto.
6. Implement the approach in ProM. In this thesis, the new version of ProM (ProM 2008) is used as an implementation platform rather than the latest released version of ProM (ProM 5).
7. Evaluate the implemented methods using both simulated and real life event logs. Analyze insights into processes which are gained from the constructed models.

1.6 Outline

The remainder of this thesis is organized as follows:

In Chapter 2, we provide preliminary knowledge which is used throughout this thesis. The chapter provides a literature overview covering areas such as business process performance analysis, process mining techniques, and performance analysis approaches in the context of process mining.

Chapter 3 provides our analysis result of process models which can intuitively project performance information. Based on the analysis, we provide a conceptual process model in this chapter. An approach to calculate performance information based on event logs and the model is explained in Chapter 4. Chapter 5 provides process models which can be used to project performance information onto. The implementation of the approach in ProM is described in Chapter 6. The evaluation of the implemented approach is given in Chapter 7. Finally, Chapter 8 concludes this master thesis and provides recommendations for future work.

Chapter 2

Preliminaries

This chapter provides preliminary concepts that are used throughout this thesis. Section 2.1 provides an overview of business process performance analysis. Then, Section 2.2 provides an introduction to process mining as an approach to gain insights into processes, including the performance of the processes. A brief introduction to current approaches to measure performance through process mining is given in Section 2.3. Finally, an overview of commercial performance analysis tools is provided in Section 2.4.

2.1 Business Process Performance Analysis

Although the term “performance” is commonly used in literature, there is no clear agreement about what the term actually means. Slightly different definitions of performance are given in [10], [34], and [36]. In this thesis, we adhere to the definition in [36] which defines performance as “the way the organization carries its objectives into effect”. Performance is measured in terms of performance metrics.

Business processes are commonly case-driven, i.e. tasks are executed for specific cases [44]. Examples of cases are insurance claims, customer orders, tax declarations, and mortgages. Case-driven processes are also called workflows and are typically described using three different dimensions: the case dimension, the process dimension, and the resource dimension (see Figure 2.1).

The case dimension signifies that business processes are handled individually and are independent from each other. The process dimension is concerned with the partial ordering of tasks [44]. This dimension determines which tasks need to be executed and how the routing of cases along the tasks is performed. Typical structures which are specified in the dimension include conditional, sequential, parallel and iterative routing of cases. Tasks which need to be executed for a particular case are referred to as *work-items*. An example of a work-item is task “send bill” for case “car order 10 for customer Robert”. Work-items are executed by resources, which is captured by the resource dimension. A resource can be a machine (e.g. a printer, a computer) or a human (e.g. a secretary, a director). A work-item which is executed by a resource is referred to as an *activity*.

Currently, there are many approaches to measure the performance of business processes, each with different metrics which are related to certain workflow dimen-

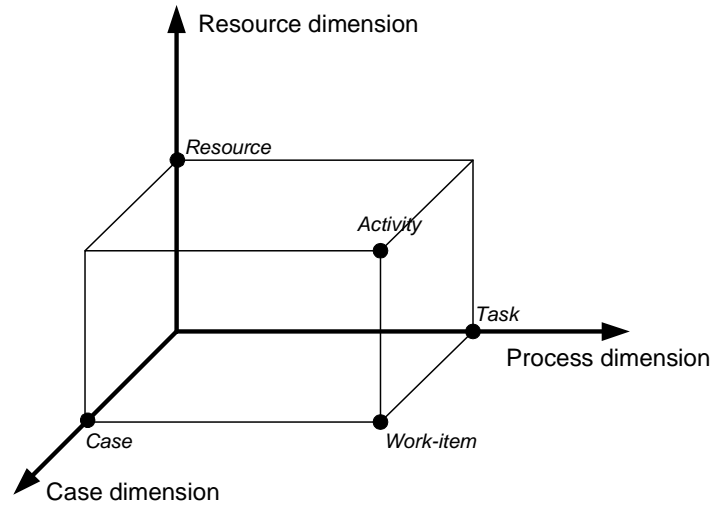


Figure 2.1: A three dimensional view of a workflow [14, 44]

sions. In [29], six performance measurement systems were analyzed in order to derive suitable performance dimensions for measuring the performance of a workflow. As a result, four dimensions of workflow performance metrics were defined: the *time dimension*, the *flexibility dimension*, the *quality dimension*, and the *cost dimension*. Each dimension has its own metrics. In this thesis, we only focus on two dimensions: the time dimension and the flexibility dimension, as both of them can be measured directly from event logs. However, for the sake of completeness, the other dimensions are also explained in the remainder.

The Time dimension

Time is a commonly used performance dimension. It is considered as both a source of competitive advantage and the fundamental measure of performance [13]. Performance metrics in the time dimension can be measured from event logs with time information [29].

Based on [12, 16, 28, 29], we formulate several common workflow-related performance metrics. The first metric in the time dimension is the *case throughput time*, which is defined as the time it takes to handle a case (i.e. process instance). This metric is derived from the case dimension. Other performance metrics in the time dimension are derived from all three dimensions of workflow (i.e. case, process, and resource dimension) as these metrics are based on activities and their states. In a case, activities may go through different states. As an example, the MXML format [22, 46, 47] specifies several states of an activity (see Figure 2.2).

As shown in Figure 2.2, when an activity is created, it is either *scheduled* or skipped automatically (*autoskip*). Scheduling an activity means that the control over the activity is put to a system. The scheduled activity can now be *assigned* to a resource. Assigned activities can later be *reassigned*. All scheduled, assigned, or reassigned activities can be skipped manually (*manualskip*) or be *withdrawn*. Only assigned activities can be *started*. Started activities can be *suspended* and then be *resumed* arbitrarily often. In the end, the activity must be *completed* or

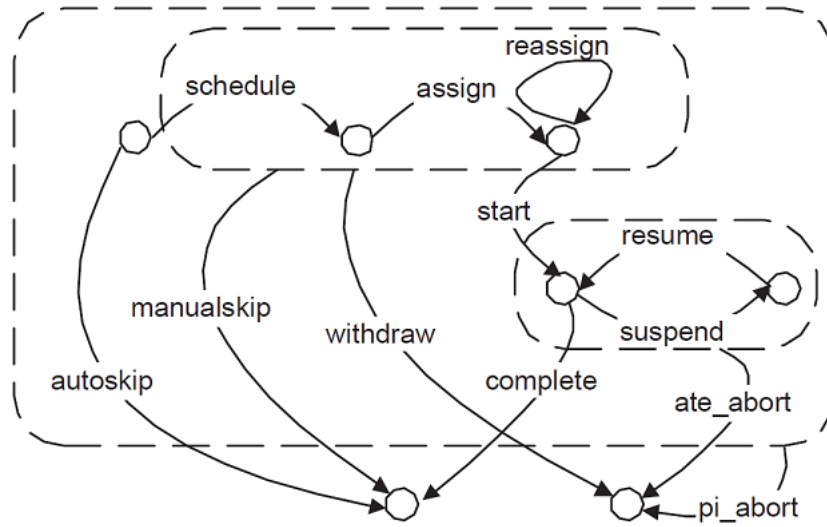


Figure 2.2: Transactional model for activities [47]

aborted (*ate_abort*). In any state during the activity lifecycle, a case can be aborted (*pi_abort*). Note that each state transition is indicated by an event with a certain event type. In Figure 2.2, event types are shown as labels of the arcs connecting a state to another state.

The MXML format may not capture all type of events that can exist in an event log. In fact, there is currently no widely-accepted standard for event types of events in event logs although one has been proposed recently (the Business Process Analytics Format (BPAF) which is proposed by the Workflow Management Coalition (WfMC) [50]). However, the type of events that are provided in the format provides a sufficient basis to measure the most commonly measured performance metrics in the time dimension. These metrics are illustrated in Figure 2.3. Note that Figure 2.3 also shows the *case throughput time* that has been explained earlier.

The first metric that is related to activity is the *activity throughput time*, which is defined as the time between a moment an activity is scheduled and the moment the activity is completed. The throughput time consists of two sub metrics:

1. The *queue time*: the time a scheduled activity spends waiting for a resource to become available.
2. The *service time*: the time that resources spend on doing the activity.

If there are activities in a case which are involved in synchronization relations with other activities, i.e. activities which can only be executed after two or more directly preceding activities are finished, two other performance metrics can be calculated: the *waiting time* and the *synchronization time*. Suppose that there is a set of activities S which need to be executed before an activity X can be executed in the case. Waiting time is defined as the time between the latest moment when all activities in S are finished and the moment the activity X is scheduled. Synchronization time is calculated for each activity $s \in S$ as the time between a moment the latest activity in S is finished and the moment s is finished.

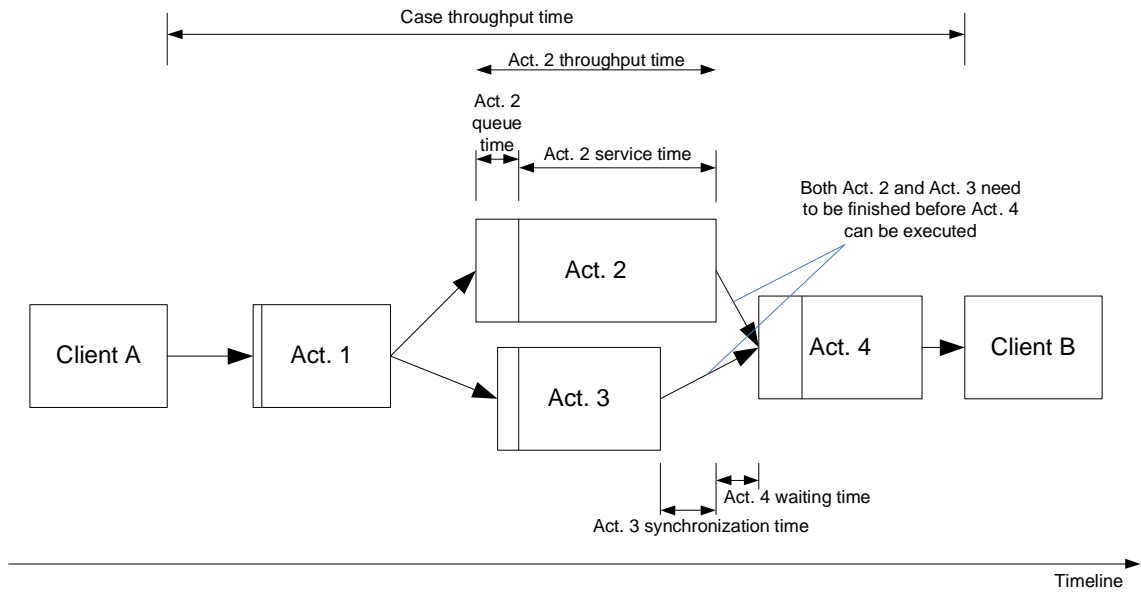


Figure 2.3: Performance measures - time dimension

The Flexibility dimension

In the context of business process execution, flexibility refers to the degree of freedom that users have to make local decisions about how to execute business processes [37]. Several flexibility metrics have been collected in [29] from various resources. The metrics are given as follows:

1. *Mix flexibility*: the ability to process different kinds of cases:
 - (a) For resources: the number of case types a resource can handle.
 - (b) For tasks: the number of case types a task can handle.
 - (c) For workflow: the number of case types that can be handled.
2. *Labor flexibility*: the ability to perform different tasks:
 - (a) For resources: number of executable tasks.
 - (b) For workflow: available resources per task and per case.
3. *Routing flexibility*: the ability to process a case using multiple routes (number of different sequences in the workflow). Note that a process which has looping activities has infinitely many possible sequences.
4. *Volume flexibility*: the ability to handle changing volumes of input (available time per employee).
5. *Process modification flexibility*: the ability to modify the process (number of sub flows in the workflow, complexity, and number of outsourced tasks).

Different aspects of flexibility can be considered for each of the metrics, such as range (the range of variations that can be handled), time (the amount of time required to adapt to change), and cost (the amount of money required to adapt to change).

The Quality dimension

The quality dimension covers subjective performance evaluations of business processes. The quality dimension can be seen from at least two angles: external (quality based on customer) and internal (quality based on worker) [29]. External quality covers both output quality (performance, conformance, and serviceability) and process quality. It cannot be measured directly, as it is influenced and determined by many different factors. Some metrics that can be directly measured for this quality include the number of specialists that work in a case and the number of task per resource. Whether a specific aspect influences the external quality of a process and the degree to which it affects the quality is highly dependent on the type of process [29].

The internal quality is determined by workers' satisfaction which may also depend on their psychological and social factors. Some metrics which can be used to measure internal quality are *skill variety* (number of different tasks and case types), *task identity* (ratio of number of executed tasks and total number of tasks per workflow), and *autonomy* (ratio of number of authorized decisions and total number of decisions). Similar to external quality, internal quality cannot be evaluated as a whole only by performance metrics. How much a metric does reflect the real internal quality of a business process depends on the type of the process.

The Cost dimension

Cost dimension is closely related to the other three dimensions. For instance, long lead times can result in a more costly process; low quality can lead to expensive rework, and low flexibility can also result in a more costly process execution [29]. Several metrics from this dimension include running costs, inventory costs, transport costs, administration costs, and resource usage.

From all performance metrics in all dimensions provided before, no metric is more important than the other. Each organization may have its own priority of performance metrics. For example, a five-star hotel will most likely find the quality of its service to be more important than the costs. In contrast, a small youth hostel will focus more on the costs rather than the quality of service. The metrics that mostly support the mission and strategy of an organization are called Key Performance Indicators (KPIs).

2.2 Process Mining

As shown in our literature study of business process performance analysis in Section 2.1, process models are required in order to calculate several KPIs of business process, especially the KPIs which are related to the time dimension. However, in practice, the models are mostly user-defined and do not support any activity abstraction. In this section, we provide an overview of process mining as an approach to analyze processes based on event logs. With process mining, various types of process models can be discovered from event logs. Section 2.2.1 provides an intro-

duction to event logs as input for process mining. Section 2.2.2 gives an overview of process discovery techniques as part of process mining and the models they extract from event logs. Finally, in Section 2.2.3, a brief explanation of the ProM framework as one of the currently leading process mining tools is given.

2.2.1 Event Logs

Currently, more and more processes are supported by Information Technology (IT) systems. These processes can be very diverse, i.e. from the process of manufacturing microchips to the process of claiming insurance. Most of these IT systems record all events related to the processes they support, leaving footprints of the processes in the form of *event logs*. These footprints provide valuable information which can be further analyzed.

It is common in practice to have dedicated systems to support specific processes. Therefore, it is more likely that there are various types of event logs provided by these systems, each consisting different types of information. For the purpose of process mining, it is important to abstract from all specific event logs implementations. Thus, a set of minimum requirements needs to be introduced such that process mining techniques can be applied to any type of event log, independent of its specific implementation. Based on [22], we identify minimal requirements for an event log to be useful in the context of process mining:

1. Each event refers to a given point in time and should not refer to a period of time. For example, starting to work on some work-item in a workflow system would be an event. Finishing the work-item is another event. The process of working on the work-item itself is not.
2. Each event should refer to one activity only, and activities should be uniquely identifiable.
3. Each event should contain a description of the event type. For example, activity was started or completed. This transactional information allows us to refer to the different events related to the same activity.
4. Each event should refer to a specific case (i.e. process instance). We need to know, for example, for which invoice the payment activity was started.
5. The events within each case are totally ordered, for example by timestamps.

Based on these requirements, we now formalize event logs as follows:

Definition 2.2.1. (Event Logs) An *event log* W is defined as:

$W = (E, ET, A, R, C, t, et, a, r, c)$, where:

E	is a set of events,
ET	is a set of event types,
A	is a set of activities,
R	is a set of resources,
C	is a set of cases,
$t : E \rightarrow \mathbb{R}_0^+$	is a function assigning a timestamp to each event,
$et : E \rightarrow ET$	is a function assigning an event type to each event,
$a : E \rightarrow A$	is a function relating each event to an activity,
$r : E \rightarrow R \cup \{\perp\}$	is a function relating each event to a resource, and
$c : E \rightarrow C$	is a function relating each event to a case.

By $\langle\langle e_0, e_1, e_2, \dots, e_n \rangle\rangle$ we denote a sequence of events in a case such that $\forall_{0 \leq i < j \leq n} c(e_i) = c(e_j) \wedge e_i \neq e_j \wedge t(e_i) < t(e_j)$ and $\forall_{e \in E} c(e) = c(e_0) \Rightarrow e \in \{e_0, \dots, e_n\}$. A set of such sequence in an event log W is denoted by C_W . Note that in this thesis, we assume that no events in the same case have exactly the same timestamps.

A standardized event log format which satisfies these requirements has been proposed in form of the MXML format [22, 46, 47]. The format specifies what kinds of information commonly exist in event logs and how the information should be stored based on XML. The MXML process log format is illustrated as a tree of XML elements in Figure 2.4. As shown in the figure, an event log is represented by the *WorkflowLog* element. As a system may record events from more than one process, the event log contains a *Process* element to store information of each of the recorded processes. We may also use the optional *Data* and *Source* elements, each to store arbitrary textual information and to store information about the system in which the event log originated from, respectively. For each process, information about individual cases (i.e. process instance) is stored in *ProcessInstance* elements. For a case, information about its individual elements is stored in *AuditTrailEntry* elements. In addition, both the *Process* element and the *ProcessInstance* element may have *Data* subelements to store arbitrary textual information.

The children of an *AuditTrailEntry* element store several important types of information about the event which the element refers to. The *WorkflowModelElement* element stores information of the activity which is referred to by the event. The *EventType* element stores information about the type of the event. The *Timestamp* element stores the timestamp of the event, and the *Originator* element stores the resource that executed the event.

Events can be related to anything which happens at a particular time, even if it is not actually useful for any analysis purpose. For instance, starting to work on a “Create report” work-item can be considered an event which is also useful for auditing purpose. Typing a character in a word processor as a part of the work can also be considered as an event, although it may not be useful at all. To date, many systems have defined their own set of event types. The variety of event types creates problems to analyze events consistently across heterogeneous systems. Some systems may only record events with a particular event type (e.g. only start events, complete events), and some others record events with various event types. With the freedom to choose possible event types, we introduce a standard for event types which are used throughout this thesis. The standard is shown in Figure 2.5. Note that the standard is not meant to be complete. It is rather intended to be compact

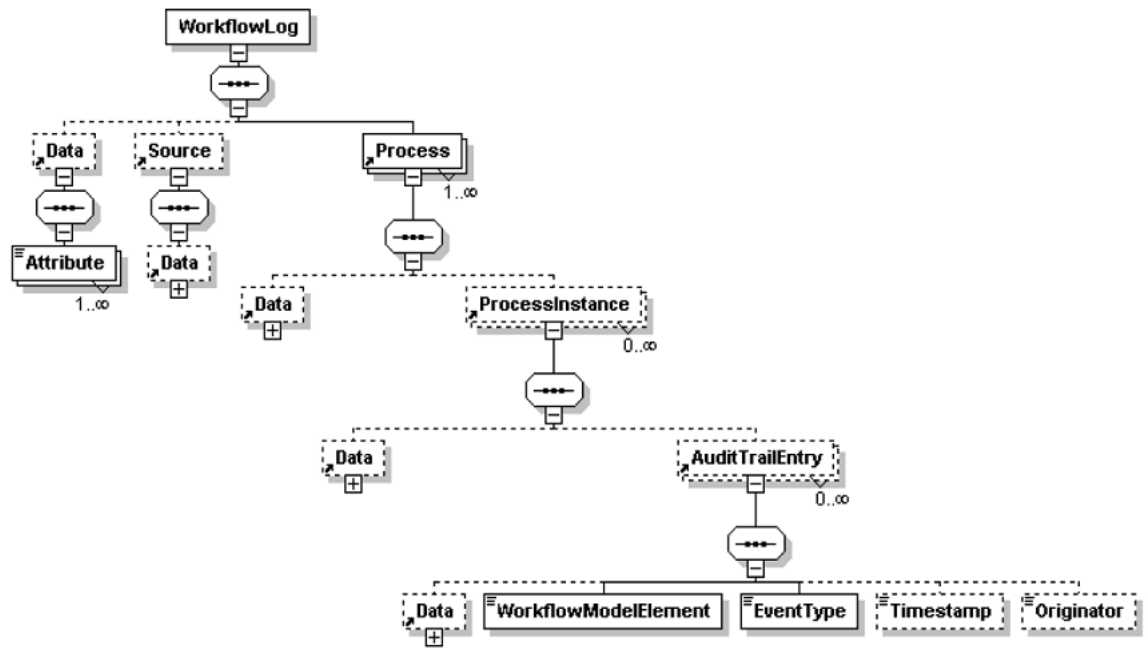


Figure 2.4: MXML format of a process log [47]

and easily understood as a transactional model in this thesis, yet good enough to capture the most commonly event types that exist in real life event logs.

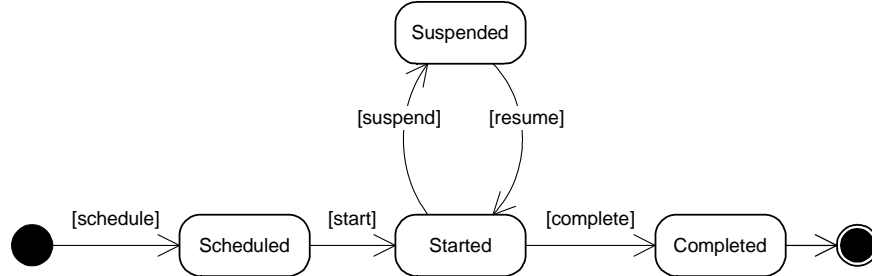


Figure 2.5: Standard transactional model in this thesis

The transactional model shown in Figure 2.5 should be interpreted as follows. When an activity is created, it enters the *scheduled* state. Scheduling an activity means that the control over the activity is put into a system. Next, a resource can *start* working on an activity. Later, an activity can be *suspended* and *resumed* arbitrarily many times. In the end, the activity should be completed. The transition from a state to another state is triggered by an event with a specific event type as indicated by the labels on each arc of the transactional model in Figure 2.5.

2.2.2 Process Discovery

From an event log, a complete process model can be generated using process discovery techniques [15]. Process discovery has been the main focus of early process mining techniques. The purpose of process discovery is to derive information about process models, organizational contexts, and important properties from event

logs. Creating process model manually is a complicated and time-consuming process [6,16]. Many people need to be involved in the creation of the model. Moreover, there are often discrepancies between the actual processes and the processes as perceived by the management [15]. In relation to our purpose of measuring performance of business processes, these discrepancies may lead to misleading performance measurement results as process models are not correctly describing real process executions. Using process discovery techniques, these risks are minimized because process models are derived from reality (by exploiting the availability of event logs).

There are many available process discovery techniques. An example of these techniques is the α algorithm proposed in [18]. The algorithm constructs a process model in the form of a Petri net from an event log based on the causal dependencies between activities. The α algorithm computes which pairs of activities always appeared in sequence, which pairs of activities can appear in any sequence, and which pairs of activities never follow each other directly. This information is used to generate a process model. Unfortunately, the algorithm relies on strong assumptions such as the absence of noise. It also assumes that the log is complete with respect to the “directly follows” relation between activities. If an activity can follow another activity directly, the log should contain an example of this behavior [48], whereas in many real life event logs, completeness and the absence of noise cannot be guaranteed.

To deal with the noise and the completeness issues, other algorithms such as the Heuristic Miner algorithm [48] and the Genetic Miner algorithm [21] have been proposed. The heuristic miner is basically an extension of the α algorithm. It constructs a heuristic process model from an event log. It takes into account the frequencies of precedence relations between activities in order to calculate causal dependencies between nodes. This way, noise can be detected. However, as each node in a heuristic model corresponds to exactly one activity, it may produce an overly complicated process model given an event log of a complex process with many activities. For example, Figure 2.6(i) shows a heuristic model which is discovered from a real-life log with only “complete” events that occurred in 24 cases with 70 activities. Figure 2.6(ii) zooms in a part of the process model in Figure 2.6(i). As can be seen in the figure, the model is so complex that hardly any useful information is obtained from it. The same problems occur when using the Genetic Miner [21]. Although it can handle noise, the model it produces still represents each activity by a node. The one-to-one mapping between a node and an activity makes the models too complicated for processes with many activities.

To reduce the complexity of the generated process models, other process discovery algorithms have been developed which construct less strict process models. An example of such an algorithm is the one used by the Fuzzy Miner [26]. The Fuzzy Miner was developed to solve the problem of having “spaghetti-like” process models considering less-structured event logs. The technique removes some assumptions that must hold for most process discovery algorithms. First, it removes the assumption that every event in an event log has a corresponding logical activity in a process. Activities may go unrecorded and events may not correspond to any activity at all. Second, it removes the assumption that there is a perfect process model which can explain all traces in the event log completely, accurately, and pre-

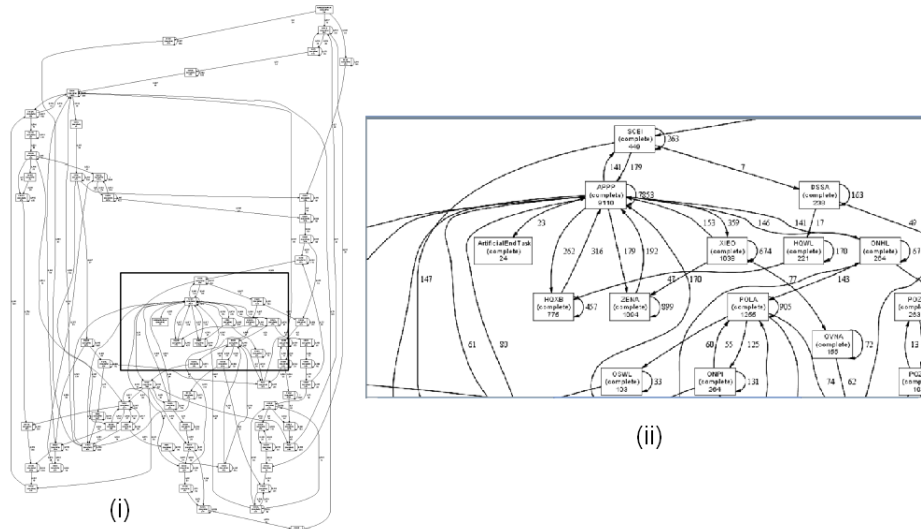


Figure 2.6: Example of heuristic model from a case [40]

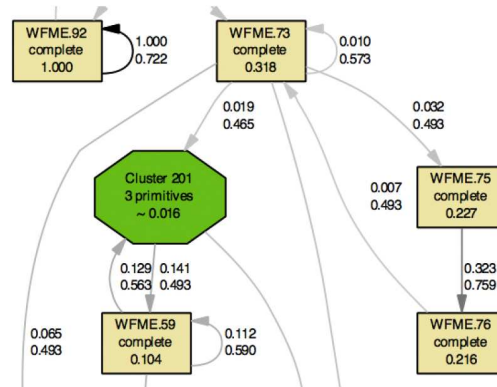


Figure 2.7: An excerpt of a Fuzzy model [26]

cisely without being overly complicated. Process models may just be incomplete, inaccurate, or imprecise.

The Fuzzy Miner generates a Fuzzy model which is able to show processes on different levels of detail, from a high-level view which shows the most significant activities and aggregates coherent, but less significant activities, to a detailed level which shows all activities [26]. A Fuzzy model is a directed graph consisting nodes and edges which connect nodes to other nodes (see Figure 2.7). A node in a Fuzzy model refers to one or more activities. A node that refers to one activity is presented as a rectangle, and a node that refers to more than one activity (cluster node) is presented as a hexagon. Arcs in a Fuzzy model indicate binary precedence relations between two nodes.

To simplify process models, the Fuzzy Miner utilizes two main concepts: aggregation and abstraction. The Fuzzy Miner aggregates coherent clusters of detailed information to provide high-level information. Low level information which is insignificant in a chosen context is abstracted away.

Other than the Fuzzy Miner, another example of a process discovery algorithm

which generates process models with reduced complexity is given in [17]. Process owners determine how generic/specific the constructed process model should be by inserting several parameter values. Although the approach in the end constructs a process model in form of a Petri net, not all activities may be presented as transitions in the model. Some of them may be abstracted away and do not appear in the model at all.

2.2.3 The ProM Framework

Although the usefulness of process mining techniques seems to be apparent, to date, not many tools that support their implementation are available. To our knowledge, the ProM (**P**rocess **M**ining) framework is the only programming framework which supports the implementation of various process mining techniques. The framework provides essential libraries which are needed in order to implement process mining techniques. The latest stable version of the ProM framework is the ProM 5. It has more than 230 plugins (see process mining website¹) and supports various process mining techniques, ranging from process discovery, conformance checking, and their extensions (including performance analysis). Besides process mining, the current version of ProM also supports the implementation of other related subjects such as model conversion and model analysis.

ProM's architecture enables functionalities to be added or to be removed easily by adding or removing plug-ins. A plug-in is basically an implementation of new piece of functionality which conforms to ProM framework standard [47]. New plug-ins can be added to ProM without any need to modify any other parts of the framework, as the framework's core automatically scans for all available plug-ins every time it is started.

In this thesis, we use the newest version of ProM (ProM 2008) which is still under development. This new version of ProM has a slightly different architecture than ProM 5. From our analysis and experience in implementing the plugins in ProM 2008, its architecture can be described as depicted in Figure 2.8. Every component in the figure has one or more elements. A user interacts with the ProM framework using a *User Interface* component. The component is responsible to provide a Graphical User Interface (GUI) to ProM. Elements in the *User Interface* component are generated by *Visualizer* components. To generate such elements, *Visualizers* use objects from the so-called *Object Pool*.

The *Object Pool* component has a crucial role in the ProM framework. This component is a repository for all objects that are either produced or needed by *Plugins*. These objects include Petri nets, their markings, various process model graphs, and event logs. Connections between the objects are also stored in the *Object Pool*. Lists of these objects can be visualized by the *User Interface* component, but only limited to objects which are considered as *Visible Objects*. *Hidden Objects* such as the semantics of Petri nets are also stored in the *Object Pool*, but they are not shown by the *User Interface* component.

The two most interesting components in the ProM framework are the *Plugins*

¹<http://www.processmining.org/>

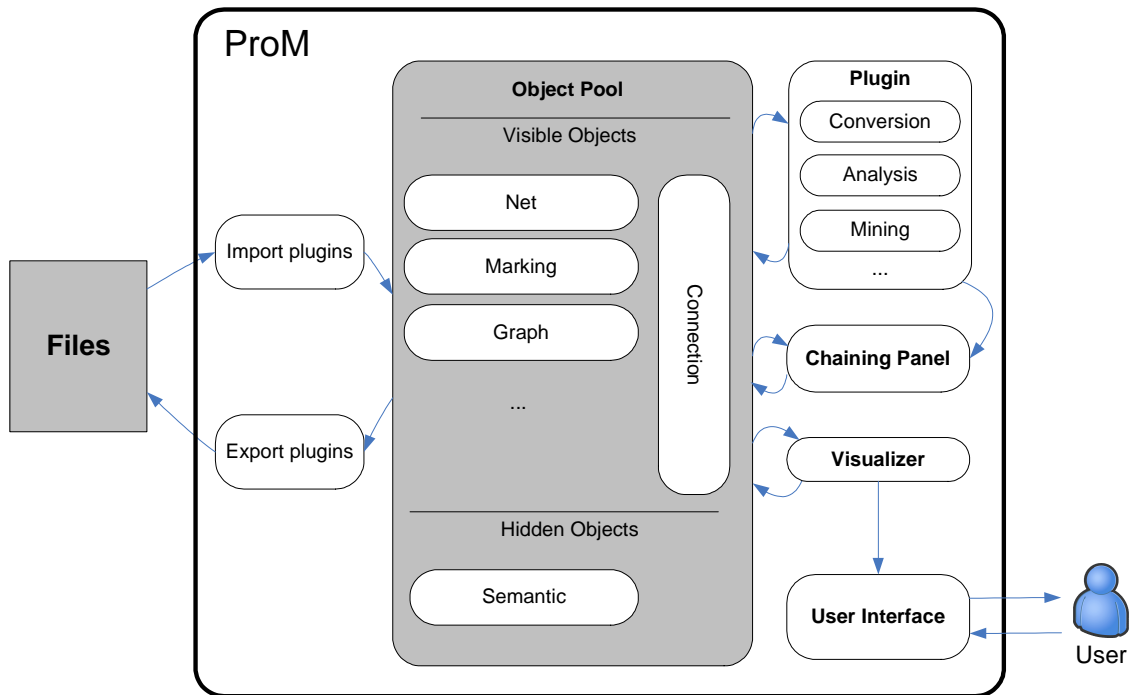


Figure 2.8: The new ProM architecture

and the *Chaining Panel* components. Elements of the *Plugin* component, as indicated by its name, are all ProM plugins which include mining, conversion, analysis, import, and export plugins. Each ProM plug-in has its own interface in the form of input and output parameters. The *Chaining Panel* component enables users to link plugins and execute them in the way they are linked. In the chaining panel, plugins' interfaces can be connected to form a chain of plugins.

ProM can process event logs by first importing them into the *Object Pool* through its import plugins. Only logs which are already imported into *Object Pool* can be accessed by ProM plugins.

2.3 Performance Analysis through Process Mining

Process mining techniques do not only address process model discovery from event logs, but they also address performance analysis of currently running processes. In [16], an approach to obtain performance information through the use of process mining is proposed. The approach projects performance information onto workflow nets (see Figure 2.9). The information is obtained by replaying a timed event log (event log with timestamp information) in a workflow net which is constructed from the log, for example, using the α algorithm. This approach successfully obtains performance information of several time dimension metrics which we described in Section 2.1, such as waiting time and synchronization time. Related to the flexibility dimension, the approach also provides the probability of taking a specific path in the net.

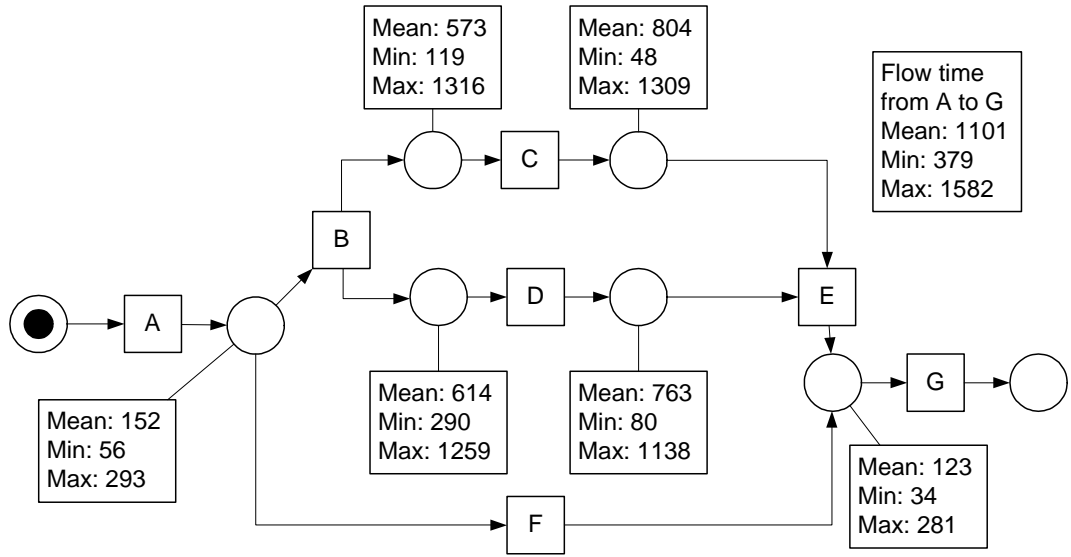


Figure 2.9: Performance information in a Petri net process model [16]

To replay the timed event log in the workflow net, the approach assumes that the start state is known. For each case in the log, a timed workflow trace is constructed. The trace is constructed by mapping events in the event log to transitions in the workflow net. Then, a token with timestamps equal to the first firing transition is placed in the initial place. Then, one by one, each transition in the timed workflow trace fires, thus collecting tokens from its input places and placing them in its output places. This is repeated until the case reaches a final marking. The time spent by tokens in each element of the net serves as a basis for performance calculation.

A drawback of this approach is that it requires the constructed workflow net to fit the log, i.e. each case in the log should be a trace in the net. Hence, given a log of a complex business process (e.g. with many unique traces and exceptional cases), the constructed workflow net is also complex. For the purpose of business process analysis, the complexity of the net prevents further insights to be obtained as such a model is hard to be understood (cf. Figure 2.6).

An extension to the approach of [16] is introduced in [28] by enabling invisible transitions to fire such that the net does not have to fit the log. The extension is based on the log replay approach in [41], although the focus of the replay is measuring conformance between the log and the net and not measuring performance. Even with this extension, additional complexity may still be introduced due to the Petri net language, as this language has limited capabilities for expressing processes with certain control-flow patterns, such as multiple instances, advanced synchronization, and cancellations [45]. In addition, no activity abstraction is introduced in the constructed workflow net, i.e. an activity in the event log refers to exactly one transition in the net.

Another highly related approach to performance analysis through process mining is the animation of cases on Fuzzy models as proposed in [25]. The animation provides a notion of performance of the replayed process. Before cases can be animated, a process model in the form of a Fuzzy model needs to be constructed from an event log as described in [26]. Then, the event log is replayed in the model.

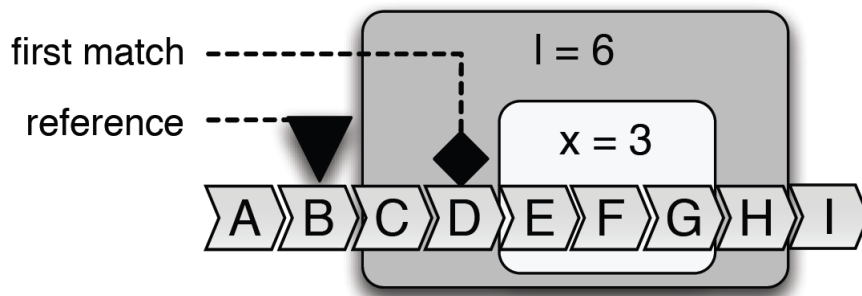


Figure 2.10: Example of a traces of events [25]

As the model has relaxed semantics, the routing of activities needs to be estimated from encountered events in the event logs. For this purpose, a dedicated/relaxed log replay technique was developed for Fuzzy models.

The log replay on a Fuzzy model is based on a heuristic. During log replay, the event currently under inspection in a trace is referred to as the *reference event*. From the reference event, the algorithm looks forward into the trace to find a valid successor, i.e. where the nodes referred to by the reference event and the event in question are connected in the Fuzzy model. The forward scan is limited to a specified *look-ahead window*. Once a valid successor is found, other valid successors are also investigated as long as it is still within both the *look-ahead window* and an *extra-look-ahead window*. The control within a case is routed from the reference event to every valid successor which is found.

As an example, see Figure 2.10. The figure shows a trace of events. Each event corresponds to a node in the Fuzzy model which has the same label. Suppose that the reference event is **B**, and the successors of node B in the Fuzzy model are the nodes D and node I. The look-ahead window is set to 6, and the extra-look-ahead window is set to 3. The algorithm iterates through the traces and finds event **D** as the first event which refers to one of the successors of node B (event **D** refers to node D in the Fuzzy model). As the first successor is found, the future events within the extra-look-ahead window (until event **G**) are then checked for other successors of node B. As shown in the figure, no event within the extra-look-ahead window refers to any successor of node B. As there is still an event within the look-ahead window, the algorithm continues to inspect event **H**. From the iteration, only node D is encountered as a successor. Therefore, from node B, the inspected case is routed only to node D (ignoring node I which is located outside of both the look-ahead-window and the extra-look-ahead window).

Case animation in a Fuzzy model provides an indication of both performance and control flow. For example, see the snapshot of an animation based on a Fuzzy model in Figure 2.11. The routing of control within a case is clearly seen as the white dots flowing from one node to another through the arcs between the nodes. The arc width indicates how often a case is routed over the arc during animation. From the figure, we can observe bottleneck activities from arcs which are filled by white dots. Arcs with a lot of white dots indicate that there are many cases queuing to be routed through the arcs. Thus, it may indicate that the target nodes of the

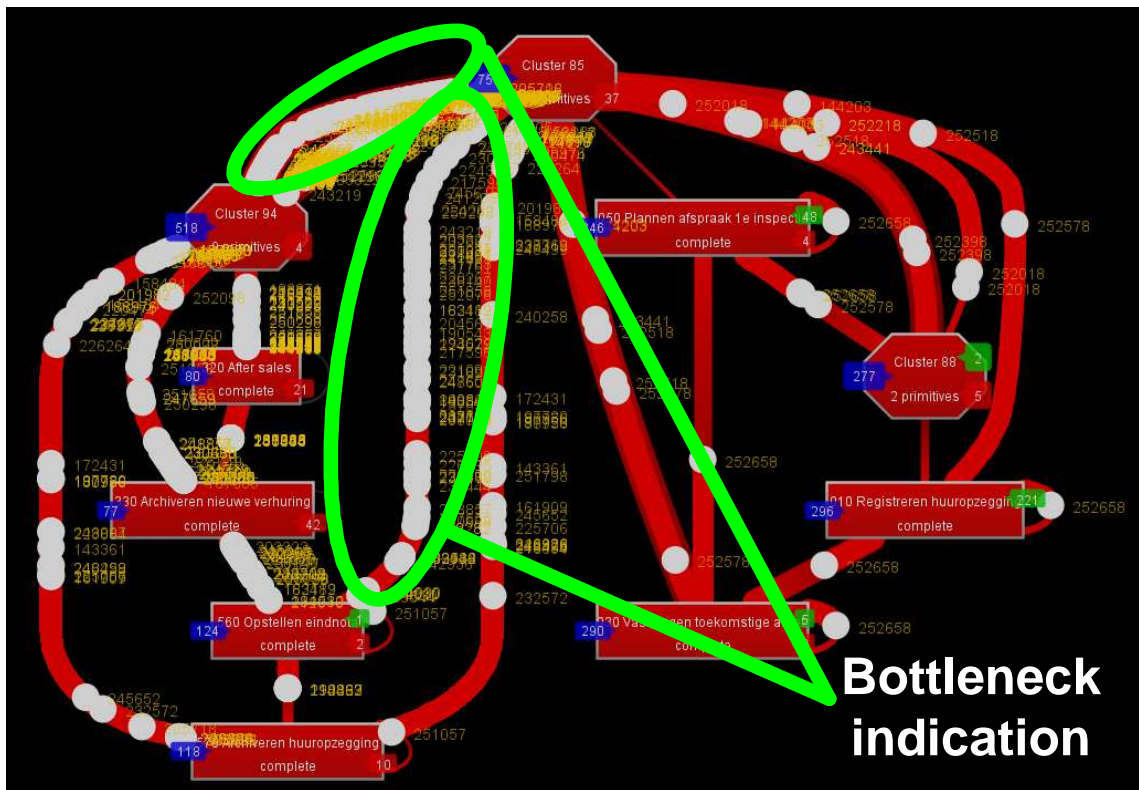


Figure 2.11: Log animation in a Fuzzy model

arcs require a long time to be finished. Unfortunately, other than an indication of bottlenecks and node frequency measurements, the currently proposed animation technique does not produce any explicit performance information.

2.4 Performance Analysis in Commercial Tools

To investigate how the performance metrics in Section 2.1 are measured and how they are presented in practice, we have investigated several leading commercial Business Performance Analysis (BPA) tools available in today's market. The tools include Fujitsu Interstage [1], Lombardi Teamworks 7 [2], Pegasystems' SmartBPM Suite [5], Software AG's WebMethods [8], Savvion's BusinessManager [7], Metastorm BPM, Oracle's Business Process Analysis (BPA) Suite [4], and IBM's Websphere Business Monitor [9]. All of the tools are among the market leaders in the area of BPA [27]. Since the full versions of these tools were only available commercially, the investigation was conducted mostly through each product's documentation.

Based on our investigation, most BPA tools measure various KPIs of a process in a real time fashion. In some tools, KPIs can even be customized. Most KPIs that are measured are categorized as time dimension KPIs, such as throughput times of activities or lead times of cases. Important KPIs are often presented in a single main panel which is named the *dashboard*. An example of which is shown in Figure 2.12). On the dashboard, a user can see various KPIs, each with its own form and insights into the currently running process. Thus, in a single dashboard, elaborated

insights into process are presented.

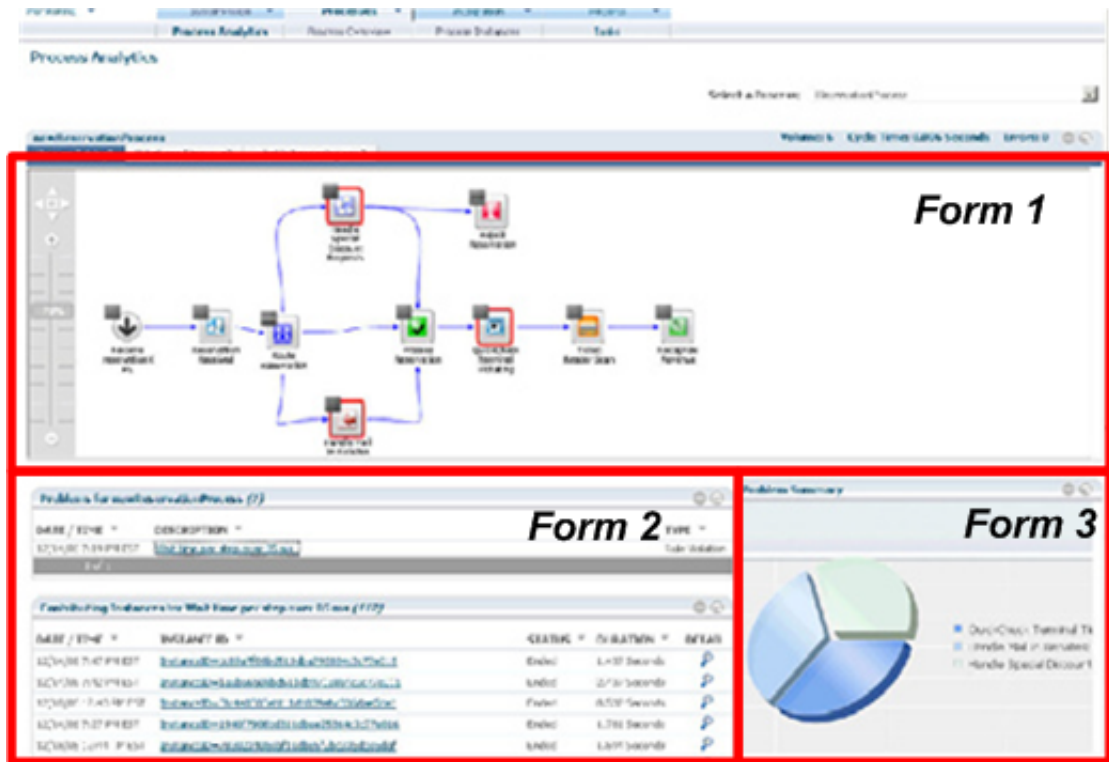


Figure 2.12: Example of performance dashboard in the Software AG's WebMethods [11]

In most of the investigated tools, the values of the calculated KPIs are shown in the form of graphs and tables (see for example Figure 2.13). In its simplest form, a graph only plots the current value of a KPI compared against a threshold value. For example, in Figure 2.14, the workload of resources which handle on-boarding new clients is presented as a “speedometer”. In the example, the workload is still at an acceptable level, i.e. it does not point into the red arc.

A more complex type of graph is a process model with projected performance information (see Figure 2.12, Form 1). Typically, activities which have a throughput time below/above a certain threshold value are highlighted in the model. In Figure 2.12, these activities are bordered by a thin red line. Each tool may support different process models, and some tools even introduce their own process modeling language. From all modeling languages that are supported by the tools, only some enable multiple activities to be represented by a single node in a process model (e.g. in BPMN, activities in a process model with a lower hierarchy can be represented by a single node in another process model with a higher hierarchy).

Most tools require user-defined process models to project performance information onto. Typically, these models are also used as an execution model, i.e. each node in the models refers to a single activity in a process. Hence, most of the models provide all details of their processes without any abstractions. Some tools provide business process simulation features so that process designers can get estimated performance values for their new process models before the models are actually enacted.

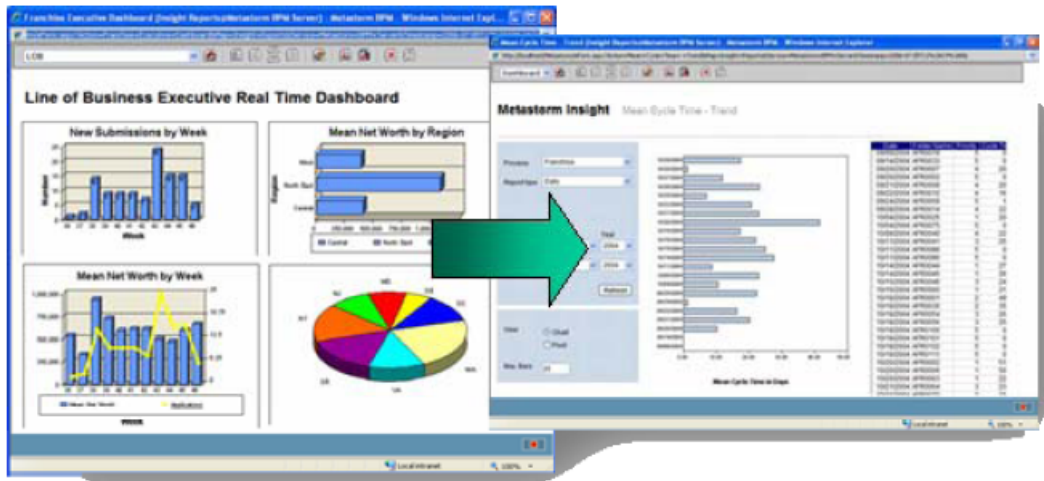


Figure 2.13: Charts and tables to show performance information in the Metastorm BPM [3]



Figure 2.14: Speedometer to indicate resources workload to handle on-boarding new clients in the Metastorm BPM [3]

After process models are committed, performance is measured based on real-time events which are captured by the enactment systems.

In this chapter, we provided several important performance metrics which can be measured from event logs and how they can be calculated. We have also investigated several process models which can represent processes intuitively. Furthermore, we investigated several approaches to measure the performance of a business process given a process model. Although there are some algorithms which produce process models with activity abstractions, none of the approaches described in literatures and supported by tools are able to extract performance information from event logs.

Therefore, in Chapter 3 of this thesis, we introduce an intuitive process modeling language that allows for arbitrary abstractions of a process. Furthermore, in Chapter 4, we present an algorithm to replay a log in the model to obtain performance information. How to project this performance information on the model is shown in Chapter 5, while an implementation of all of the approaches is presented in Chapter 6.

Chapter 3

Modeling Processes

Based on our investigation in Section 2.2.2, currently there are already several process discovery techniques which construct intuitive process models from event logs, even if the logs describe complex processes. In this chapter, we analyze the intuitive process models introduced in Section 2.2.2 in more detail. To solve the issues with these models, we introduce a conceptual process model called Simple Precedence Diagram (SPD) in Section 3.1. In Section 3.2, we explain how to obtain SPDs.

3.1 Intuitive Process Models

Each process discovery technique described in Section 2.2.2 constructs a different type of process model. However, only two of the techniques construct process models that support activity abstraction such that even complex processes can be presented in a simple and intuitive way.

The first technique is the Fuzzy Miner which constructs Fuzzy models. Simplification of a complex process in a Fuzzy model is performed by abstracting away some activities from the model and aggregating several other activities in cluster nodes. Given a Fuzzy model and an event log which is represented by the model, not all activities in the log need to be presented as nodes in the model. Activities which are not presented in the model are said to be *abstracted*. Activity aggregation in Fuzzy models is achieved using cluster nodes. A cluster node in a Fuzzy model has a one-to-many relation with activities in an event log. As there are multiple activities referring to the same cluster node in a Fuzzy model, we can say that the cluster nodes of Fuzzy model *aggregate* activities.

Fuzzy models also have relaxed semantics. The arcs in a Fuzzy model represent precedence relations between activities. Suppose that there is an arc from node A to node B in a Fuzzy model, then an observation of an activity which is referred to by node A *may be followed* by another observation of an activity which is referred to by node B. With such relaxed semantics, the arcs of a Fuzzy model do not constrain any possible case routing in the process that the model represents. If there is no arc from node A to node B in the fuzzy model, the observation of an activity which is referred to by node A followed by the observation of an activity which is referred to by node B does not violate semantics of the fuzzy model. In addition, to provide additional insights into the process' bottlenecks during log replay (as discussed in

Section 2.3), the color and the size of the model's elements are utilized (e.g. arcs which are frequently used to route cases are depicted wider). Therefore, no extra elements are introduced in the model to provide insights into the process.

The second technique which construct process model with activity abstraction is the one which is described in [17]. This technique constructs Petri nets which may abstract some activities away, depending on the values of the input parameters (e.g. some activities in the event log may not be presented as transitions in the constructed Petri nets). In 2.3, the abstraction process need to be guided by the users.

A Petri net consists of places, transitions, and arcs which connect places to transitions and the other way around. A transition in a Petri net has one-to-one relation to an activity in an event log. Petri nets have strictly defined semantics. A transition in a Petri net can only fire if all places which are connected to the transition by any of its incoming arcs (e.g. all input places of the transition) contain at least one token. When the transition fires, a token from each of the input place is removed, and a token is placed to each place which is connected to the transition by any of the transition's outgoing arc (e.g. all output places of the transition). With the strict semantics, Petri nets can describe possible case routing in processes more precise than Fuzzy models. However, the advantage comes at the expense of visualization complexity. A Petri net requires more elements than a Fuzzy model to describe a process. For example, a process consisting of activity A followed by activity B can be described by a Fuzzy model using only two nodes and an arc (3 elements in total as shown in Figure 3.1). If the same sequence is represented by a Petri net, two transitions, two arcs, and a place are needed (5 elements in total as shown in Figure 3.2).

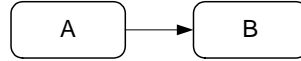


Figure 3.1: Fuzzy model to show sequence pattern from A to B

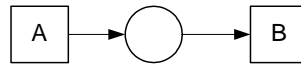


Figure 3.2: Petri net to show sequence pattern from A to B

Without any extensions, Fuzzy models can only represent an activity as a node. In some other models, such as Petri nets, this limitation does not exist. For example, the Petri net in Figure 3.3 has two transitions which refer to activity D. The net gives us an insight into the only two possible traces: **A-B-D-E-G** and **A-C-D-F-G**. If the same process is represented in the form of a Fuzzy model, the model would look similar to the one given in Figure 3.4. As an activity in an event log may only refer to one node in a Fuzzy model, there can only be one node **D** in the model. From the Fuzzy model in Figure 3.4, other traces such as **A-B-D-F-G** and **A-C-D-E-G** are also valid.

Based on our analysis of fuzzy models and Petri nets, we conclude that to visualize complex processes conveniently, it is important to have a process model which

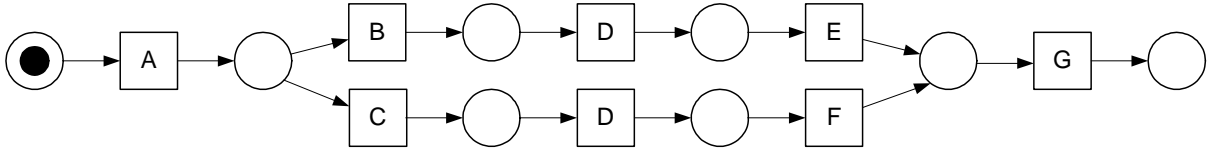


Figure 3.3: Process model with two possible traces

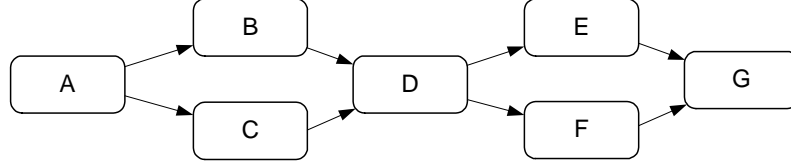


Figure 3.4: Fuzzy model of process described in Figure 3.3

supports both activity abstraction and activity aggregation. By activity abstraction, given the process model and an event log which is represented by the model, not all activities in the log may be presented as nodes in the model. Activity aggregation means that a node in the model may represent more than one activity, and an activity may refer to more than one node. These requirements can be satisfied by a process model with a many-to-many relation between nodes in the model and activities in the process it represents. Furthermore, it is also important that arcs between nodes do not constrain possible case routings such that given a complex process, arcs can be removed from the model as needed to decrease the complexity of the model without constraining the possible routing. This implies that the model needs to have relaxed semantics.

Therefore, we define a process model on a very high level which is named *Simple Precedence Diagram* (SPD). SPD is a process model which combines the advantages of Fuzzy models and Petri nets to represent processes intuitively using activity abstraction and aggregation. It generalizes process models which can be constructed by process discovery techniques (e.g. Petri nets and Fuzzy models). An SPD should be seen as a conceptual process model of a process in the form of a directed graph consisting of nodes and edges.

Nodes in an SPD represent activities in a loose way. A node in an SPD represents a set of activities which are performed within a continuous period in the process. An SPD node may represent a set of activities that do not have precedence relations between each other, e.g. activities that are executed in parallel, or activities that are randomly executed without clear precedence relations. An SPD node may also represent a set of activities that are always started one after another but intersects during some periode of time. A node in an SPD has a many-to-many relation with activities in an event log. Each node in an SPD refers to one or more activities in an event log, while an activity in the log refers to zero or more nodes in the SPD. Edges in SPDs define a notion of control flow between SPD nodes with relaxed semantics. An edge between node α and node β in an SPD means that an occurrence of an instance of the node α *may be followed* by an occurrence of an instance of the node β .

The formal definition of SPDs is given as follows:

Definition 3.1.1. (Simple Precedence Diagram)

Let $W = (E, ET, A, R, C, t, et, a, r, c)$ be an event log. We say that $S = (W, N, L, l_a, l_n)$ is a corresponding *Simple Precedence Diagram* of the event log W , where¹

- N is a set of nodes,
- $L \subseteq N \times N$ is a set of edges linking the nodes,
- $l_a : A \rightarrow \mathcal{P}(N) \setminus \{\emptyset\}$ is a function relating an activity to a set of nodes, and
- $l_n : N \rightarrow \mathcal{P}(A) \setminus \{\emptyset\}$ is a function relating a node to a set of activities. Let $n \in N, l_n(n) = \{a \in A \mid n \in l_a(a)\}$.

Note that for convenience, we assume that all abstracted activities and all events which refers to them are removed from event logs. Therefore, although SPDs support activity abstraction (an activity may not appear in any SPD nodes), in Definition 3.1.1 activities in event logs must refer to at least one SPD node. The assumption also holds for all remaining definitions in Chapter 3 and Chapter 4.

3.2 Obtaining SPDs

To obtain an SPD of a process, we propose three approaches. The first approach is to convert a process model that represents the process to the SPD. An explanation of the approach is given in Section 3.2.1. The second approach is by constructing the SPD directly from event log using process discovery algorithms. In Section 3.2.2, we present such an algorithm. The third approach is by simply creating a hand made SPD for the process and map each node of the SPD to activities in a log. A short explanation about the third approach is given in Section 3.2.3.

3.2.1 Converting Process Models to SPDs

SPDs generalize process models which consist of nodes and directed arcs. However, SPDs are not generic enough to generalize all process models, because all nodes in SPDs are mapped to activities. Some process models have types of nodes which cannot be mapped to any activity (e.g. places in Petri nets, events and connectors in EPCs). Therefore, we define a Generic Process Model (GPM) as a generalization of all process models with nodes and directed arcs as their elements. GPMs and SPDs are similar, except that some nodes in a GPM may refer to no activity at all in an event log, while every node in an SPD must refer to at least one activity in the event log.

The formal definition of GPM is given as follows:

Definition 3.2.1. (Generic Process Model)

Let $W = (E, ET, A, R, C, t, et, a, r, c)$ be an event log. A corresponding *Generic Process Model* of the event log W is defined as $GP = (W, GN, GE, g_a, g_n)$, where:

¹with $\mathcal{P}(N)$, we denote the powerset of N

GN	is a set of nodes,
$GE \subseteq GN \times GN$	is a set of edges linking the nodes,
$g_a : A \rightarrow \mathcal{P}(GN) \setminus \{\emptyset\}$	is a function relating an activity to a set of nodes, and
$g_n : GN \rightarrow \mathcal{P}(A)$	is a function relating a node to a set of activities.
Let $n \in GN, g_n(n) = \{a \in A \mid n \in g_a(a)\}$.	

We argue that every process model which is represented by nodes and directed arcs can be converted to a corresponding GPM. After the GPM is obtained, an SPD can be obtained by converting the GPM to the corresponding SPD. To explain our approach to convert GPMs to SPDs, we need to define unmapped path in a GPM. An unmapped path between two GPM nodes is a path such that none of the nodes in between the origin and the destination is mapped to any activity. The formal definition of unmapped path is given as follows:

Definition 3.2.2. (Unmapped Path)

Let $GP = (W, GN, GE, g_a, g_n)$ be a GPM. Let $n, n' \in GN$. We say that there is an *Unmapped Path* from n to n' iff there exists a sequence of nodes $\langle n_0, n_1, \dots, n_k \rangle$ such that $n_0 = n, n_k = n', k > 0$ and $\forall_{0 < i \leq k} (n_{i-1}, n_i) \in GE$ and $\forall_{0 < i < k} g_n(n_i) = \emptyset$. By $n \rightsquigarrow n'$, we denote that such a path exists.

The basic idea to convert a GPM to an SPD is to create a node in the SPD for each node in the GPM which refers to an activity or a set of activities. Each SPD node should refer to the same activity or set of activities as the GPM node it is created from. Then, for each pair of GPM nodes (n, n') in which both n and n' refer to an activity or a set of activities, a directed arc is created in the SPD if there exists an unmapped path between them. This approach can be formalized as follows:

Definition 3.2.3. (SPD of a GPM)

Let $W = (E, ET, A, R, C, t, et, a, r, c)$ be an event log and $GP = (W, GN, GE, g_a, g_n)$ be a GPM of the event log W . We define an SPD $S = (W, N, L, l_a, l_n)$ as the SPD of GP , where

- $N \stackrel{def}{=} \{n \in GN \mid g_n(n) \neq \emptyset\}$,
- $L \stackrel{def}{=} \{(n, n') \in N \times N \mid n \rightsquigarrow n'\}$,
- $\forall_{a \in A} l_a(a) \stackrel{def}{=} g_a(a)$, and
- $\forall_{n \in N} l_n(n) \stackrel{def}{=} g_n(n)$.

Both Definition 3.2.1 and Definition 3.2.3 help us to construct SPDs of specific process models. Suppose that we want to construct an SPD of a Petri net. Petri nets are basically directed graphs consisting nodes (places and transitions) and directed arcs between the nodes. Therefore, using Definition 3.2.1, we can define the GPM of the Petri net as given in Definition 3.2.4. Then, using Definition 3.2.3, the constructed GPM can be converted to an SPD.

Definition 3.2.4. (GPM of a Petri Net)

Let $W = (E, ET, A, R, C, t, et, a, r, c)$ be an event log, $PN = (P, T, F, l)$ be a Petri net where P is a set of places, T is a set of transitions, F is a set of directed edges between places and transitions, $l : T \rightarrow A$ is a function relating a transition to an

activity, and $l_2 : A \rightarrow \mathcal{P}(T)$ is a function relating an activity to a set of transitions. We say that $GP = (W, GN, GE, g_a, g_n)$ is a *GPM* of PN where

- $GN \stackrel{def}{=} P \cup T$,
- $GE \stackrel{def}{=} F$,
- $g_a : A \rightarrow \mathcal{P}(GN)$ where $\forall_{a \in A} g_a(a) \stackrel{def}{=} l_2(a)$, and
- $g_n : GN \rightarrow \mathcal{P}(A)$ where $\forall_{p \in P} g_n(p) \stackrel{def}{=} \emptyset$ and $\forall_{t' \in T} g_n(t') \stackrel{def}{=} \{l(t')\}$.

As an example, suppose that we want to convert the Petri net in Figure 3.5 to an SPD. Using Definition 3.2.4, we convert the Petri net to a GPM which is illustrated in Figure 3.6. Then, using Definition 3.2.3, each node referring to one or more activities is kept, and each unmapped path between these nodes is translated into an arc. The resulting SPD is shown in Figure 3.7.

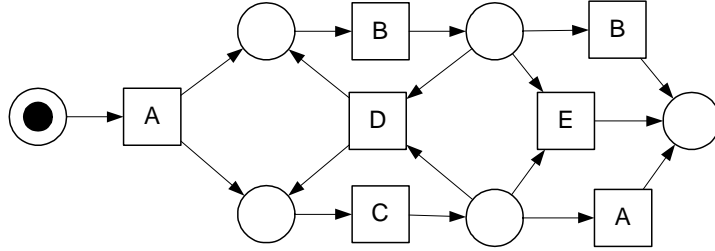


Figure 3.5: An example Petri net (1)

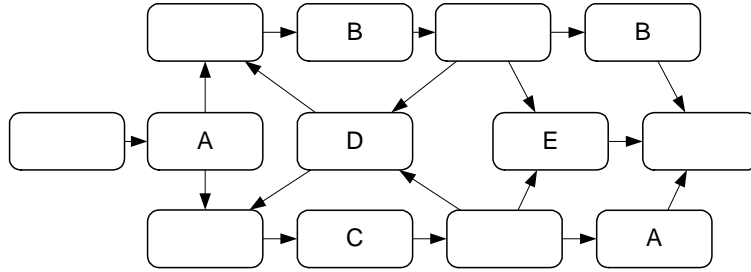


Figure 3.6: The GPM of the Petri net in Figure 3.5

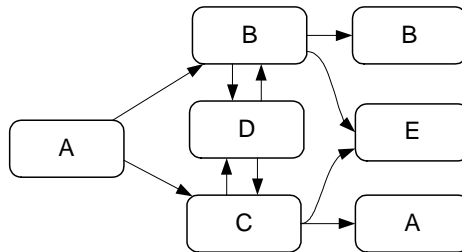


Figure 3.7: The SPD of the Petri net in Figure 3.5

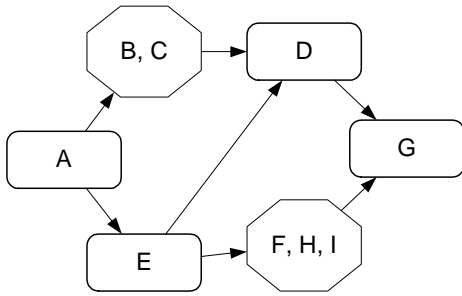
Using GPMs, we can also convert Fuzzy models to SPDs. As every nodes in a Fuzzy model must refer to either an activity or a set of activities, conversion from Fuzzy models to GPMs is straightforward. A GPM is a Fuzzy model that does not

distinguish cluster nodes and ordinary nodes. As an example, the GPM of the Fuzzy model in Figure 3.8a is shown in Figure 3.8b. Notice that the GPM in Figure 3.8b is also an SPD of itself.

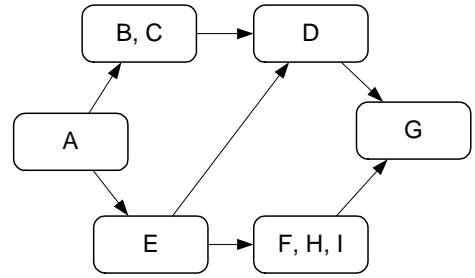
Definition 3.2.5. (GPM of a Fuzzy Model)

Let $W = (E, ET, A, R, C, t, et, a, r, c)$ be an event log, $FM = (W, FN, FE, f_a, f_n)$ be a Fuzzy model of the event log where FN is a set of nodes, FE is a set of directed edges between the nodes, $f_a : A \rightarrow FN$ is a function relating each activity to its node in the Fuzzy model, and $f_n : FN \rightarrow \mathcal{P}(A)$ is a function relating a node in the Fuzzy model to a set of activities where let $n \in FN$, $f_n(n) = \{a \in A \mid n \in f_a(a)\}$. We say that $GP = (W, GN, GE, g_a, g_n)$ is a *GPM* of FM where

- $GN \stackrel{def}{=} FN$,
- $GE \stackrel{def}{=} FE$,
- $\forall_{a \in A} g_a(a) \stackrel{def}{=} \{f_a(a)\}$, and
- $\forall_{n' \in GN} g_n(n') \stackrel{def}{=} f_n(n')$.



(a) An example Fuzzy model



(b) A GPM/an SPD of the Fuzzy model

Figure 3.8: An example of a Fuzzy model and a GPM/an SPD which is constructed from the model

The loosely defined semantics of both nodes and arcs in SPDs may lead to ambiguity. For instance, using our conversion approach, the Petri net in Figure 3.9 is also converted to the SPD in Figure 3.7. Both the net in Figure 3.9 and the net in Figure 3.5 are converted to the same SPD, although from a behavioral point of view they are very different. SPDs are intended to be sketch of process models rather than precise process models. The idea of having a conceptual process model is that no matter what kind of technique is used to construct a process model from an event log, the model can always be converted to an SPD.

3.2.2 Mining SPDs from Event Logs

Other than constructing an SPD based on other process model, an SPD can also be constructed directly from an event log. For this purpose, we can use a straightforward, fuzzy clustering algorithm [23]. The goal of clustering algorithms is to divide observations over a number of subsets (or clusters), such that the observations in each of these clusters are similar in some sense, i.e. often precede each other. The idea behind the clustering algorithm we use follows this concept in a very simple

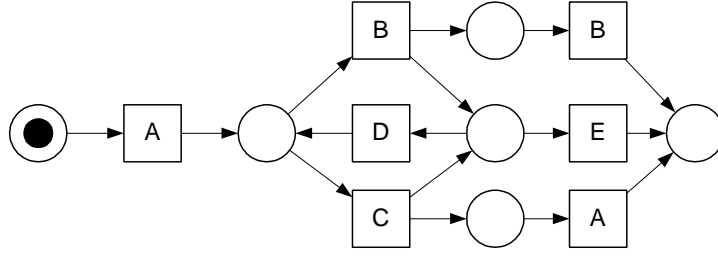


Figure 3.9: An example Petri net (2)

way. First, we define a similarity metric on activities. Then, we choose a number of clusters and use a Fuzzy k-Medoids algorithm to create clusters of activities that maximize the similarity values in each cluster. Note that an activity may appear in more than one cluster.

Such a Fuzzy k-Medoid algorithm requires two metrics, namely (1) a measure for the (dis)similarity of objects (activities in our case) and (2) a measure for the probability that an object belongs to a cluster of which another object is the medoid. We define both metrics based on direct succession of events.

Definition 3.2.6. (Event Succession)

Let $W = (E, ET, A, R, C, t, et, a, r, c)$ be an event log. We define $>_W: A \times A \rightarrow \mathbb{N}$ as a function counting how often events from two activities directly succeed each other in all cases, i.e. for $a_1, a_2 \in A$, we say that $>_W(a_1, a_2) = \#_{e_1, e_2 \in E}(t(e_1) < t(e_2) \wedge a(e_1) = a_1 \wedge a(e_2) = a_2 \wedge c(e_1) = c(e_2) \wedge \nexists e_3 \in E(c(e_3) = c(e_1) \wedge t(e_1) < t(e_3) < t(e_2)))$. We use the notation $a_1 >_W a_2$ to denote $>_W(a_1, a_2) > 0$.

The similarity between activities is defined by looking at how often events relating to these activities follow each other directly in the log. If events relating to these activities follow each other more often, their similarity increases. Note that if two activities a_1, a_2 are different, their similarity is never equal to 1.

Definition 3.2.7. (Activity Similarity)

Let $W = (E, ET, A, R, C, t, et, a, r, c)$ be an event log. We define the similarity $\sigma: A \times A \rightarrow (0, 1]$ between two activities $a_1, a_2 \in A$, such that if $a_1 = a_2$ then $\sigma(a_1, a_2) = 1$, otherwise $\sigma(a_1, a_2) = \frac{>_W(a_1, a_2) + >_W(a_2, a_1) + 1}{2 + 2 \cdot \max_{a_3, a_4 \in A} (>_W(a_3, a_4))}$.

Note that $\max_{a_3, a_4 \in A} (>_W(a_3, a_4))$ is the maximum value of $>_W$ from all pair of activities in the event log. With this function, we are able to obtain value between 0 and 1 for every pair of activities even in extreme conditions such that no activities ever precede others (e.g. all cases only consists of one event). In that case, the similarity between two activities would be 50%.

As stated before, we also need a measure for the probability that an activity belongs to a cluster of which another activity is the medoid. For this purpose, we use the FCM membership model from [19].

Definition 3.2.8. (Cluster Membership Probability)

Let $W = (E, ET, A, R, C, t, et, a, r, c)$ be an event log. Furthermore, let $A^k \subseteq A$ with $|A^k| = k$ be a set of medoids, each being the medoid of a cluster. For all $a_1 \in A^k$ and $a_2 \in A$ we define the probability $u(a_1, a_2)$ to denote the probability

that a_2 belongs to the cluster of which a_1 is the medoid, i.e. $u : A^k \times A \rightarrow [0, 1]$, where $u(a_1, a_2) = \frac{\sigma(a_1, a_2)^{\frac{1}{m-1}}}{\sum_{a_3 \in A^k} \sigma(a_3, a_2)^{\frac{1}{m-1}}}$. Note that $m \in [1, \infty)$ here denotes the so-called “fuzzifier” which can be fixed at a certain value, e.g. suppose that $m = 2$, $u(a_1, a_2) = \frac{\sigma(a_1, a_2)}{\sum_{a_3 \in A^k} \sigma(a_3, a_2)}$.

Using the cluster membership and the similarity functions, we can introduce the fuzzy k-Medoid algorithm.

Definition 3.2.9. (Fuzzy k-Medoid Algorithm)

Let $W = (E, ET, A, R, C, t, et, a, r, c)$ be an event log. Furthermore, let $0 < k \leq |A|$ be the desired number of clusters. We search a set of medoids $A^k \subseteq A$ with $|A^k| = k$, such that this set minimizes $\sum_{a \in A} \sum_{a^k \in A^k} \left(\frac{u(a^k, a)^m}{\sigma(a, a^k)} \right)$.

Finally, after the medoids have been found, we need to construct an SPD. Obviously, the found clusters correspond to the nodes in the SPD model, thereby also providing the mapping between activities in the log and nodes in the model. The edges however are again constructed using the succession relation defined earlier.

Definition 3.2.10. (SPD Mining Algorithm)

Let $W = (E, ET, A, R, C, t, et, a, r, c)$ be an event log and let $A^k \subseteq A$ be a set of medoids. We define the mined SPD model from the event log L as $S = (W, N, L, l_a, l_n)$ such that ²

- $N = A^k$, i.e. the nodes of the SPD model are identified by the cluster medoids,
- $L = \{(a_1^k, a_2^k) \in A^k \times A^k \mid \exists_{a_1 \in l_n(a_1^k) \setminus l_n(a_2^k)} \exists_{a_2 \in l_n(a_2^k) \setminus l_n(a_1^k)} a_1 >_W a_2\}$.
- $l_a : A \rightarrow \mathcal{P}(A^k)$, such that $l_a(a) = \{a^k \in A^k \mid u(a^k, a) \approx \max_{a_1^k \in A^k} (u(a_1^k, a))\}$, and
- $l_n : A^k \rightarrow \mathcal{P}(A)$, such that $l_n(a^k) = \{a \in A \mid a \in l_a(a^k)\}$.

According to Definition 3.2.10, an activity refers to a node (and vice versa) if the probability that the activity belongs to the cluster represented by the node is approximately the same as the maximum probability over all clusters. This implies that each medoid belongs to its own cluster. Furthermore, all other activities belong to at least one cluster, namely the one for which the function u is maximal. An activity can belong to multiple clusters. Note that we do not use equality of probabilities, as this would require the number of direct successions in the log to be the same for multiple pairs of activities and this is rarely the case in practice.

After nodes are constructed, the only thing left is to construct edges of the connected SPD. The edges are determined using the direct succession relation. Basically, two nodes are connected if there is an activity referred to by the first node that is not referred to by the second node that is at least once directly succeeded by an activity referred to by the second node, but not by the first.

²with $a \approx b$, we denote the value of a is approximately the same as the value of b

3.2.3 Creating a Hand Made SPD

SPD is a process model that can present processes intuitively, regardless of their complexity. In situations where process owners already have intuitions about their processes execution, they can create their own SPDs. An advantage of using SPDs to describe processes over other process models is that SPDs can describe processes in any level of abstraction. Hence, process owners can adjust the level of abstraction of their hand made process models according to both their need and knowledge about the processes.

For example, suppose that a process consists of six activities labelled A, B, C, D, E, and F. A process owner knows that the process always starts with activity A and ends with activity F, but he does not know the precedence relations between activities B, C, D, and E. In this case, the process owner can draw an SPD consisting of 3 nodes as shown in Figure 3.10. One node represents activity A, another node represents activity F, and the last node represents the rest of the activities. The owner does not need to know the precedence relation between B, C, D, and E. As long as he knows that activity B, C, D, and E are executed within a continuous time period between the executions of activity A and activity F in the process, the activities can be represented as a single SPD node *B,C,D,E*. Note that the process can also be described as the SPD shown in Figure 3.11 where activity B, C, D, and E can be executed in an arbitrary order. In this case, the decision to choose which process model is the best to describe the process is left to the process owner.

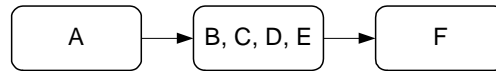


Figure 3.10: An example of a hand made SPD that is created based on limited information about the process (1)

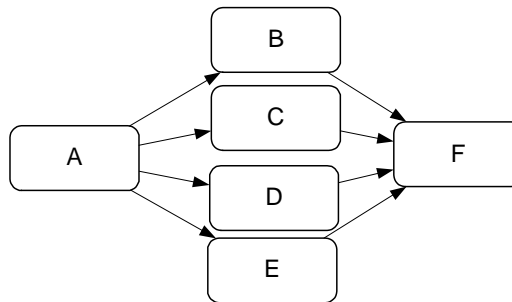


Figure 3.11: An example of a hand made SPD that is created based on limited information about the process (2)

Using the same example, suppose that the process owner has additional information that activity C is always started after activity B is started in the process, but before it is finished. Thus, activities B and C are performed within a relatively short continuous period in the process. Therefore, these activities are better to be represented as a single node, separated from activities D and E. As the precedence between activity D, and activity E are unknown, it is safe to assume that they can

be executed in an arbitrary order. This process can be represented by an SPD as shown in Figure 3.12. The SPD in Figure 3.12 provides arguably better insights into the process than the SPD in Figure 3.10 and Figure 3.11, as it provides an intuition that activities B and C are related to each other in some sense closer than other relations which can be formed by a combination of activities B, C, D, and E.

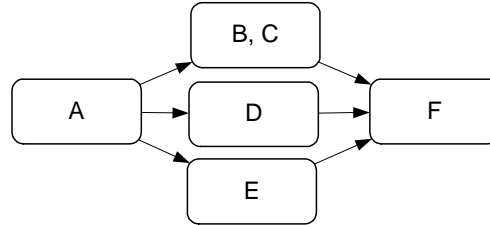


Figure 3.12: An example of a hand made SPD that is created based on limited information about the process (3)

However, with such a relaxed way to represent activities, SPD nodes can be easily misinterpreted. For instance, Figure 3.13 shows another hand made SPD of our previous process example. In this figure, the node D, E is created to indicate that activity D and activity E are performed without any precedence order. However, based on the way activities B and C are presented as node B, C in Figure 3.12, node D, E may also give a false indication that activity E is always started during execution of activity D. Therefore, we would like to emphasize that to interpret SPDs correctly, knowledge about the process under consideration and reasoning behind the construction of nodes and arcs is required.

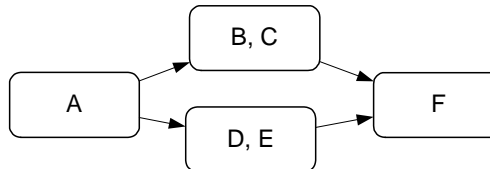


Figure 3.13: An example of hand made SPD that is created based on limited information about processes (4)

In this chapter, we provided SPDs as high level process models which can present a process intuitively, regardless of the process' complexity. We also provided several approaches to obtain SPDs of processes: by converting process models to SPDs, by constructing SPDs from event logs, or by creating hand made SPDs. In the next chapter, we use both an SPD and an event log to obtain performance information of a business process which is represented by the log.

Chapter 4

Measuring Performance

In Chapter 3, we presented SPDs as conceptual process models which can represent even complex processes intuitively. In this chapter, we explain an approach to measure performance of a business process based on a given SPD and the process' event log. Section 4.1 provides an overview of how the performance information is extracted. Sections 4.2 to 4.5 provide step-by-step explanation of how to extract the information. In Section 4.6, a complete list and explanation of the Key Performance Indicators (KPIs) that can be obtained from the information extraction is provided.

4.1 Overview

In order to obtain performance information from an event log, we replay the log on a given model. We assume the model to be an SPD, but we ensure robustness of our approach by not making assumptions on the structure of the process. Our log replay is influenced by both the case animations of Fuzzy models [25] and the replay of event logs in Petri nets [28, 41]. An overview of our log replay is given in Figure 4.1. As shown in the figure, in order to perform the replay, both an SPD and an event log are given. Each case in the log is treated independently. Therefore, we only show a sequence of events of a single case in the figure. Each node of the SPD should be mapped to an activity or a set of activities in the log. In the figure, each node is labelled by a greek alphabet (α, β, γ , and δ) and the activities each node refers to are given in brackets (e.g. node α refers to both activity A and B).

The first step of the log replay is to determine for each event in the sequence, which SPD node it refers to. This step is needed because there can be more than one node which is referred to by an activity. The result of the first step is an SPD node sequence for each sequence in the log. An illustration of the first step is given in Figure 4.1 where a sequence of SPD nodes $\langle \alpha, \alpha, \dots, \delta \rangle$ is derived from a sequence of events. Each event is represented by the name of activity which is referred to by the event and its event type (e.g. A(start) represents an event e in an event log W where $et(e) = start$ and $a(e) = A$). Details of how to obtain node sequences from sequences of events in the event log is given in Section 4.2.

The second step of the log replay is to identify to which node instance a node in the node sequence refers to. In order to do this, we define a *look-ahead* value as a user-defined value which determines how many nodes ahead of the currently

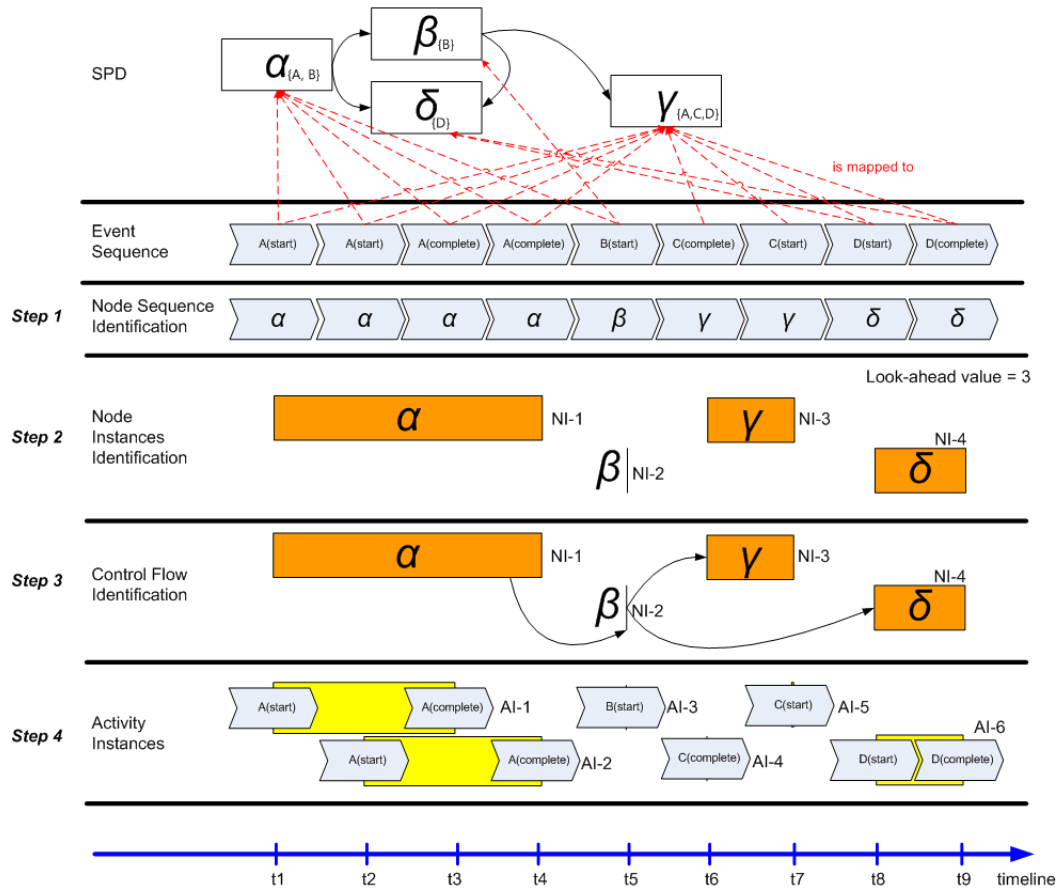


Figure 4.1: Overview of the approach to measure performance

inspected node are considered to determine the node instances. This look-ahead is similar to the look-ahead of cases animation in [25]. The details of the second step are given in Section 4.3.

After the node instances are known, the third step is to identify case routing between the instances. The identified case routing and node instances are the basis of performance calculation. Details of the third step are given in Section 4.4. In the fourth step, activity instances are identified. This step can actually be performed in conjunction with the third step. The identified activity instances also become the basis of our performance calculation. Further explanation of the activity instance identification is given in Section 4.5. Note that all steps in Sections 4.2 to 4.5 are necessary to perform a performance measurement of the KPIs given in Section 4.6.

4.2 Node Sequence Identification

Given an SPD and a sequence of events which represents a case in an event log, we need to relate each event to an instance of an SPD node. To construct a sequence of SPD nodes from the event sequence, we assume that the execution of activities conforms to activities' precedence relation which is indicated by the SPD. The sequence of SPD nodes is identified by decomposing the sequence of events to *maximum fitting subtraces* and replace each subtrace by its corresponding sequence

of SPD nodes.

Before we define the maximum fitting substraces, we need to define a notion of a *fitting subtrace*. A fitting subtrace is a sequence of events for which a mapping from each event in the sequence to a possible SPD node can be constructed such that given an event e in the sequence and a set of SPD nodes S consisting all nodes where at least one of e 's predecessors in the sequence is mapped to, e is mapped to either an element of S or a successor of any element of S . Fitting subtrace can be formalized as follows:

Definition 4.2.1. (Fitting Subtrace)

Let $W = (E, ET, A, R, C, t, et, a, r, c)$ be an event log, C_W be a set of event sequences that represents cases in W , and $S = (W, N, L, l_a, l_n)$ be an SPD of the event log. A *Fitting Subtrace* f_s^l of a sequence of events $\langle\langle e_0, \dots, e_n \rangle\rangle \in C_W$ is a subsequence $\langle e_s, \dots, e_{s+l-1} \rangle$, such that there exists a sequence of SPD nodes $\langle n_0, \dots, n_{l-1} \rangle$ where

- $\forall_{0 \leq i < l} n_i \in N \wedge a(e_{s+i}) \in l_n(n_i)$, and
- $\forall_{0 \leq i \leq j < l} (n_i = n_j) \vee (\exists_{i \leq k < j} (n_k, n_j) \in L)$

Given a sequence of events of a case, we argue that we can always decompose the sequence to fitting substraces. A fitting decomposition of a sequence is a decomposition of the sequence to sub sequences where each sub sequence is a fitting subtrace. A fitting decomposition can be formalized as follows:

Definition 4.2.2. (Fitting Decomposition)

Let $W = (E, ET, A, R, C, t, et, a, r, c)$ be an event log, C_W be a set of event sequences that represents cases in W , and $\langle\langle e_0, \dots, e_n \rangle\rangle \in C_W$ be a sequence of events that represents a case in W . $\langle l_0, \dots, l_m \rangle$ is a *Fitting Decomposition* of the sequence iff

- $\sum_{i=0}^m l_i = n + 1$, and
- $\forall_{0 \leq i \leq m} l_i > 0$, and
- $\forall_{0 \leq i < m} f_{\sum_{j=0}^{i-1} l_j}^{l_i}$ is a fitting subtrace of $\langle\langle e_0, \dots, e_n \rangle\rangle$.

Let C_W be a set of event sequences that represents cases in an event log W , for every sequence of events $\langle\langle e_0, \dots, e_{n-1} \rangle\rangle \in C_W$, we argue that such fitting decomposition exists. The proof is straightforward. Every sequence can be decomposed into subsequences, each consisting of only one event. A sequence which only consists of one event is a fitting subtrace. Thus, every sequence can be decomposed to fitting substraces.

Proposition 4.2.1 (Event sequences can be decomposed)

Let $W = (E, ET, A, R, C, t, et, a, r, c)$ be an event log, C_W be a set of event sequences that represents cases in W , and $\langle\langle e_0, \dots, e_k \rangle\rangle \in C_W$ be a sequence of events that represents a case in W . A fitting decomposition of the form $\langle l_0, \dots, l_m \rangle$ exists.

Proof: Since for all $0 \leq i \leq k$ holds that there exists a sequence $\langle n \rangle$ with $n \in l_a(a(e_i))$, which implies $a(e_i) \in l_n(n)$ (Definition 3.1.1), we know that $f_i^1 = \langle e_i \rangle$ is a fitting subtrace. Therefore, $\langle l_0, \dots, l_m \rangle$ is a fitting decomposition, where for all $0 \leq j \leq m = k, l_j = 1$. \square

Based on the definition of a fitting subtrace, we define a *Maximum Fitting Subtrace* as a fitting subtrace which is not a sub sequence of another fitting subtrace. This definition can be formalized as follows:

Definition 4.2.3. (Maximum Fitting Subtrace)

Let $W = (E, ET, A, R, C, t, et, a, r, c)$ be an event log, C_W be a set of event sequences that represents cases in W , and $f_s^l = \langle e_s, \dots, e_{s+l-1} \rangle$ be a fitting subtrace of a sequence of events $\langle\langle e_0, \dots, e_n \rangle\rangle \in C_W$. A *Maximum Fitting Subtrace* is a fitting subtrace f_s^l such that there is no other fitting subtrace $f_s^{l'}$ where $l' > l$.

By decomposing sequence of events $\langle\langle e_0, \dots, e_{n-1} \rangle\rangle \in C_W$ to maximum fitting subtraces, we obtain a mapping function $m : E \rightarrow N$ which maps each event in the sequence to exactly one node in SPD. However, note that not all fitting subtraces can only be mapped to exactly one sequence of SPD nodes, i.e. some fitting subtraces can be mapped to more than one sequence of SPD nodes. Hence, the selection of which sequence of SPD nodes is a mapping of a fitting subtrace is left to the implementation. In the current implementation, sequence of SPD nodes with higher number of unique SPD nodes are prioritized over sequences of SPD nodes with the lower number. In addition, the identification for maximum fitting subtraces are started from the first event in the sequence (e_0). After the first maximum fitting subtrace is identified, the next maximum fitting subtrace is identified from the next event of the sequence which is not a part of the previously identified maximum fitting subtrace. The identification is repeated until all maximum fitting subtraces are identified.

As an example, see the SPD in Figure 4.2 and the sequence of events in a case in Figure 4.3. Each event in Figure 4.3 is labelled by an activity it refers to. Using Definition 4.2.1, we obtain three maximum fitting subtraces from the sequence:

- $\langle A, B, C, D \rangle$, which can be mapped to either $\langle \alpha, \beta, \epsilon, \theta \rangle$ or $\langle \alpha, \beta, \epsilon, \delta \rangle$
- $\langle E \rangle$, which is mapped to $\langle \omega \rangle$
- $\langle F, F, G \rangle$, which is mapped to $\langle \epsilon, \epsilon, \epsilon \rangle$.

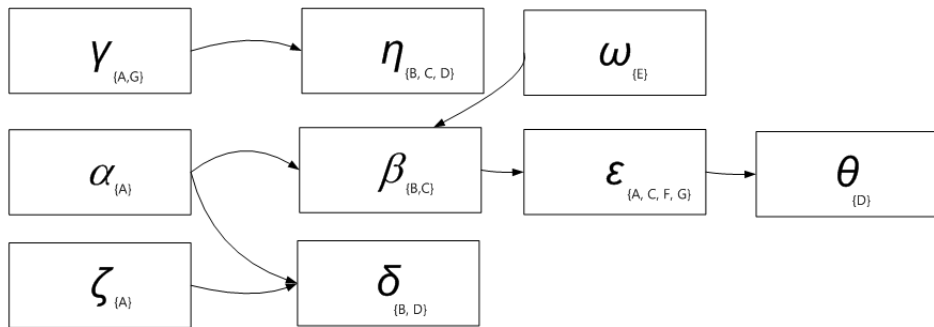


Figure 4.2: SPD for example

Both events A and B belong to the same fitting subtrace. Although there is no SPD node which refers to by both activity A and B , there exists a pair of predecessor and successor nodes connected by an arc, in which the predecessor refers to activity A and the successor refers to activity B . An example of such a pair is (α, β) , where

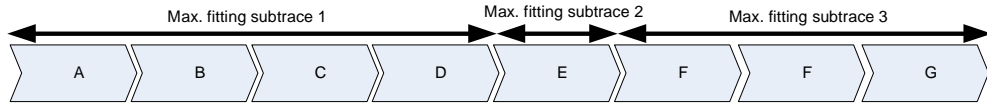


Figure 4.3: Sequence of events 1 as case example

α refers to activity A and β refers to activity B and there is an arc from node α to node β .

Event E is not a part of the first maximum fitting subtrace. According to Figure 4.2, the activity only refers to node ω . Node ω does not have any predecessor nodes and only refers to activity E . Thus, according to Definition 4.2.1, event E must be placed in a separate fitting subtrace.

According to Definition 4.2.1, there exists a sequence of SPD nodes for every fitting subtrace. According to Definition 4.2.3, maximum fitting subtraces are also fitting subtraces. For the first maximum subtrace, there are several possible sequences of nodes, such as: $\langle \alpha, \beta, \epsilon, \theta \rangle$, $\langle \alpha, \beta, \epsilon, \delta \rangle$, $\langle \alpha, \beta, \beta, \delta \rangle$, and $\langle \gamma, \eta, \eta, \eta \rangle$. In this thesis, we choose the sequences with maximum number of unique SPD nodes. Thus, our sequence candidates for the first maximum subtrace is filtered to only $\langle \alpha, \beta, \epsilon, \theta \rangle$ and $\langle \alpha, \beta, \epsilon, \delta \rangle$, as both of them have the highest number of unique nodes (4 nodes). Any of this candidates can be selected to represent the first maximum subtrace.

The second subtrace only consists of one event which refers to activity E . As node ω is the only one which is referred to by activity A , the event can only be mapped to node ω . Thus, the sequence of SPD nodes of the second event subtrace is $\langle \omega \rangle$. Using the same approach, we identify that the third subtrace corresponds to sequence $\langle \epsilon, \epsilon, \epsilon \rangle$.

By merging all identified sequences of SPD nodes ($\langle \alpha, \beta, \epsilon, \theta \rangle$ or $\langle \alpha, \beta, \epsilon, \delta \rangle$, $\langle \omega \rangle$, and $\langle \epsilon, \epsilon, \epsilon \rangle$), we obtain two sequences of SPD nodes for the sequence of events in Figure 4.3, namely: $\langle \alpha, \beta, \epsilon, \theta, \omega, \epsilon, \epsilon, \epsilon \rangle$ and $\langle \alpha, \beta, \epsilon, \delta, \omega, \epsilon, \epsilon, \epsilon \rangle$. Figure 4.4 illustrates both possible sequence of nodes for sequence of events in Figure 4.3.

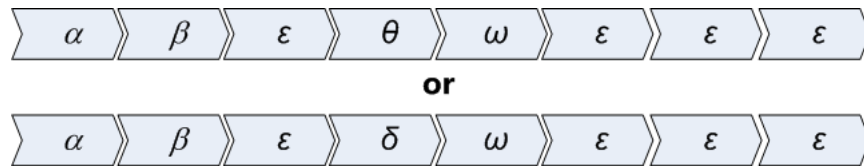


Figure 4.4: Sequence of SPD nodes for the sequence of events in Figure 4.3

In an extreme case, the number of maximum subtraces can be equal to the number of events. See the sequence in Figure 4.5 as an example. There is no two consecutive events e, e' in this sequence which either refer to the same node or refer to two different nodes which are connected by an arc. Thus, each event in the sequence is a maximum fitting subtrace. All maximum subtraces of the sequence are $\langle D \rangle$, $\langle A \rangle$, $\langle E \rangle$, $\langle D \rangle$, $\langle C \rangle$, $\langle E \rangle$, and $\langle A \rangle$. In cases where the maximum fitting subtrace only consists of one event, we can choose any SPD node that the event refers to. Three of the possible sequences of SPD nodes which can represent the sequence of events in Figure 4.5 are $\langle \delta, \zeta, \omega, \delta, \beta, \omega, \epsilon \rangle$, $\langle \delta, \alpha, \omega, \theta, \eta, \omega, \gamma \rangle$, and $\langle \theta, \alpha, \omega, \delta, \eta, \omega, \zeta \rangle$.

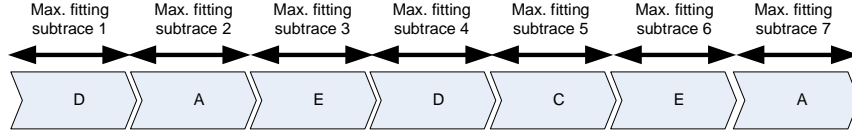


Figure 4.5: Sequence of events 2 as case example

4.3 Node Instance Identification

After sequences of SPD nodes are obtained, the next step of our log replay is to identify SPD node instances. An SPD node may refer to more than one activity. Thus, performance calculations based on SPD node instances may provide insights into a process' performance on a higher level of abstraction than calculations which are based on activity instances.

To define SPD node instances formally, we introduce the relation \models as relation between classes of events such that each of the class represents an instance of an SPD node. The relation is influenced by the case animation approach in [25]. Let W be an event log, C_W be a set of event sequences that represents cases in the log, $\langle\langle e_0, \dots, e_k \rangle\rangle \in C_W$ be a sequence of events in the event log, and $S = (W, N, L, l_a, l_n)$ be an SPD of the log. Two events e, e' in the sequence $\langle\langle e_0, \dots, e_k \rangle\rangle \in C_W$, where e occurs earlier than e' , are related with respect to their node instance if they both refers to the same node $n \in N$ and the index of event e' is at most l_a more than the index of event e in the sequence. If there is any other event e'' in the sequence that has an index between the two events and can be mapped to any successor of node n (we refer to the successor node as n'), there should be another event between event e and event e'' in the sequence that can be mapped to predecessor of node n' . l_a is named as *look-ahead value* and a sub sequence that is formed from l_a direct successors of event e in the sequence is named the *look-ahead window*.

The relation is formalized as follows:

Definition 4.3.1. (SPD Node Instance Relation) Let

- $W = (E, ET, A, R, C, t, et, a, r, c)$ be an event log,
- C_W be a set of sequences of events in event log W that represents cases,
- $S = (W, N, L, l_a, l_n)$ be an SPD of the event log W ,
- LA be a look-ahead value ($LA \in \mathbb{N}_1$),
- $tr = \langle\langle e_0, \dots, e_k \rangle\rangle, tr \in C_W$ be a sequence of events in event log W , and
- $m : E \rightarrow N$ be a function mapping an event to an SPD node that is obtained from decomposing tr to maximum fitting subtraces.

We define an relation \models on the events in tr , such that given $i, j \in \mathbb{N}_1, 0 \leq i \leq j \leq \min(k, i + LA), e_i \models e_j$ iff¹

- $i = j$, or
- $\forall_{i < h < j} [(m(e_i), m(e_h)) \in L \Rightarrow (m(e_h) \neq m(e_i) \wedge \exists_{i < l < h} (m(e_l), m(e_h)) \in L)] \wedge (m(e_i) = m(e_j))$

¹with $\min : \mathbb{Z} \times \mathbb{Z}$, we denote a function that returns the minimum value between two values

Based on the relation, we define a node instance of node $n \in N$ in a sequence of event $tr \in C_W$ as a sequence of unique events, ordered based on their timestamps consisting all events in tr that are node instance equivalent. Formally, a node instance is defined as follows:

Definition 4.3.2. (SPD Node Instance) Let

- $W = (E, ET, A, R, C, t, et, a, r, c)$ be an event log,
- C_W be a set of sequences of events in event log W that represents cases,
- $S = (W, N, L, l_a, l_n)$ be an SPD of the event log W , and
- $tr = \langle\langle e_0, \dots, e_k \rangle\rangle, tr \in C_W$ be a sequence of events in event log W .

We define a *Node Instance* of node $n \in N$ in sequence of events tr as a sequence of events $ni_n^{tr} = \langle e_{i_0}, \dots, e_{i_j} \rangle$ where

- $\forall_{0 \leq l \leq j} 0 \leq i_l \leq k$,
- $\forall_{0 < l \leq j} e_{i_{l-1}} \models e_{i_l}$,
- $\forall_{0 \leq l < p \leq j} i_l \neq i_p \wedge t(e_{i_l}) < t(e_{i_p})$, and
- $\forall_{0 \leq l \leq k}, (\exists_{0 \leq h \leq j} e_{i_h} \models e_l \vee e_l \models e_{i_h}) \Rightarrow \exists_{0 \leq g \leq j} i_g = l$.

Furthermore, we define NI_n^{tr} to be the set of node instances of node $n \in N$ in tr .

As an example, consider our previous example in Section 4.3. Suppose that the sequence of SPD nodes $\langle n_0, \dots, n_7 \rangle = \langle \alpha, \beta, \epsilon, \theta, \omega, \epsilon, \epsilon, \epsilon \rangle$ is selected to represent the sequence of events $\langle\langle e_0, \dots, e_7 \rangle\rangle$ in Figure 4.3. With a look-ahead value equal to 3, we obtain these node instances: $\langle e_0 \rangle, \langle e_1 \rangle, \langle e_2 \rangle, \langle e_3 \rangle, \langle e_4 \rangle, \langle e_5, e_6, e_7 \rangle$. Notice that although e_5 refers to the same node as e_2 and is within the look-ahead window of $e_2 (2 + 3 \leq 5)$, it is not a part of node instance $\langle e_2 \rangle$ because of the existence of e_3 . e_3 refers to a successor of $m(e_2)$, lies between e_2 and e_5 in the sequence of events, and is not preceded by any events which refer to any of the predecessors of $m(e_3)$. e_5, e_6 , and e_7 are grouped in a node instance as all of them refers to the same node and satisfies the Definition 4.3.1.

As another example, one of the possible results of node sequence identification of the sequence of events in Figure 4.5 is $\langle n_0, \dots, n_6 \rangle = \langle \delta, \zeta, \omega, \delta, \beta, \omega, \epsilon \rangle$. With this node sequence and look-ahead value equal to 3, we obtain these node instances: $\langle e_0, e_3 \rangle, \langle e_1 \rangle, \langle e_2 \rangle, \langle e_4 \rangle, \langle e_5 \rangle, \langle e_6 \rangle$. e_0 and e_3 are grouped as one node instance because both of them refer to node δ which does not have any successors and event e_3 is within the look-ahead window of $e_0 (0 + 3 \leq 3)$. Although both e_2 and e_5 refer to the node ω , they are not grouped as one node instance because there is e_4 whose SPD node refers to one of the successors of node ω .

Figure 4.6 illustrates the transformation from the sequence of events in Figure 4.3 to a sequence of SPD nodes and later to SPD node instances. As shown in Figure 4.6, node instances may have a notion of throughput time (see Section 2.1). This notion is used as one of our performance metrics which is further explained in Section 4.6.

4.4 Control Flow Identification

The third step of our log replay is performed to identify the control flow between the identified node instances. Control flow indicates the moment a process' control

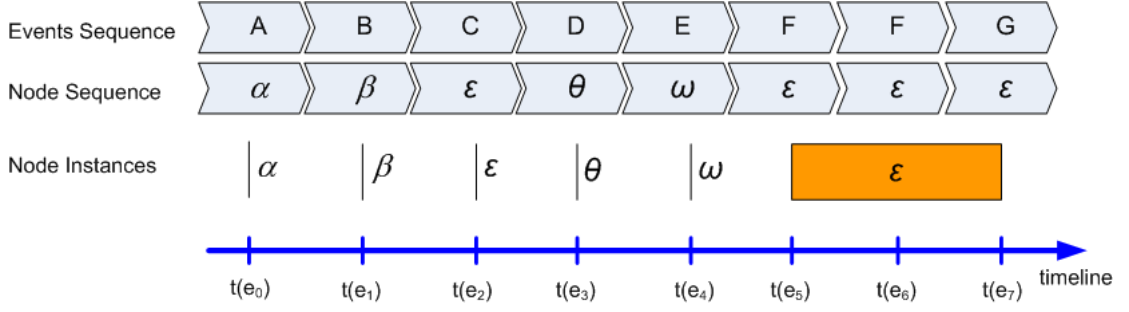


Figure 4.6: Transformation of the sequence of events in Figure 4.3 during log replay

is passed from an event in a node instance to another event(s) in another node instance(s). Control flow is passed from the last event which forms an instance of node n to each closest future events within its look-ahead window that is mapped to any successor of n such that a node instance only receive process control from another node instance exactly once.

Control flow identification can be formalized as follows:

Definition 4.4.1. (Control Flow Identification) Let

- $W = (E, ET, A, R, C, t, et, a, r, c)$ be an event log,
- C_W be a set of sequences of events in event log W that represents cases,
- $S = (W, N, L, l_a, l_n)$ be an SPD of the event log W ,
- LA be a look-ahead value ($LA \in \mathbb{N}_1$),
- $tr = \langle e_0, \dots, e_k \rangle$, $tr \in C_W$ be a sequence of events in event log W ,
- $m : E \rightarrow N$ be a function mapping an event to an SPD node that is obtained from decomposing tr to maximum fitting substraces, and
- i, j be integer values, $0 \leq i < j \leq \min(k, i + LA)$.

We say that e_i passes control to e_j , denoted by $e_i \succ_c e_j$ iff $(m(e_i), m(e_j)) \in L \wedge \nexists_{i < h \leq k} e_i \models e_h \wedge \nexists_{i < l < j} e_l \models e_j$.

Based on Definition 4.4.1, the flow of process control between events in node instances in Figure 4.6 can be illustrated in Figure 4.7. Edges between events in different node instances indicate control passing between source node instances to destination node instances. Recall that the look-ahead value for this example is equal to 3 as explained in Section 4.3. Thus, the look-ahead window for the event at $t(e_0)$ is $\langle e_{0+1}, e_{0+2}, e_{0+3} \rangle = \langle B, C, D \rangle$. Based on SPD in Figure 4.2, the only successor of node α is node β . In the look-ahead window, only event B is mapped to node β . Therefore, at time $t(e_0)$, the last event of node instance α only passes control to the event B at $t(e_1)$, which is also an element of node instance β . The other control flows in Figure 4.7 are identified using the similar approach.

As illustrated in Figure 4.7, not all process controls are gained from other node instances and some node instances may not pass control to any other node instances. Node instance θ in the figure receives a control from instance ϵ , but it does not distribute the control to any other node instance. Both the only instance of node ω and the the second instance of node ϵ do not receive any control from other

node instance. These two instances are assumed to gain process controls as long as they exist in the process and lose the control without ever passing it to any other activity instances. This relaxed notion of process control makes our approach robust enough to handle unstructured processes or logs that do not match the model under consideration.

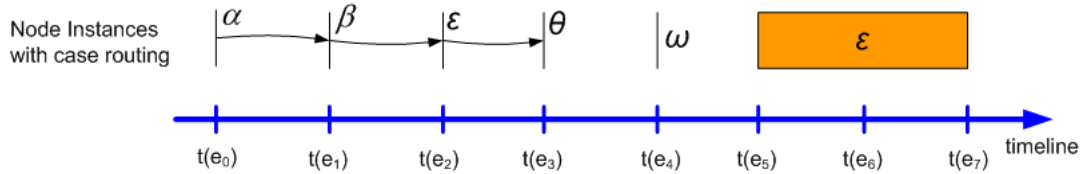


Figure 4.7: Control flow identification of node instances in Figure 4.6

As another example, consider the node instances $\langle e_0, e_3 \rangle, \langle e_1 \rangle, \langle e_2 \rangle, \langle e_4 \rangle, \langle e_5 \rangle, \langle e_6 \rangle$ which are gained from the trace of events in Figure 4.5. Suppose that the trace of events corresponds to a trace of SPD nodes $\langle n_0, \dots, n_6 \rangle = \langle \delta, \zeta, \omega, \delta, \beta, \omega, \epsilon \rangle$. Using Definition 4.4.1, we can identify the control flow between the node instances as illustrated in Figure 4.8. An interesting remark on the figure is that by the definition, node instance ζ passes control to the last event of node instance δ . Process control is passed between events, while a node instance may be constructed from several events. Thus, the controls are always passed from the end of node instances, but it may be passed to any location (start, middle, or end) of other node instances.

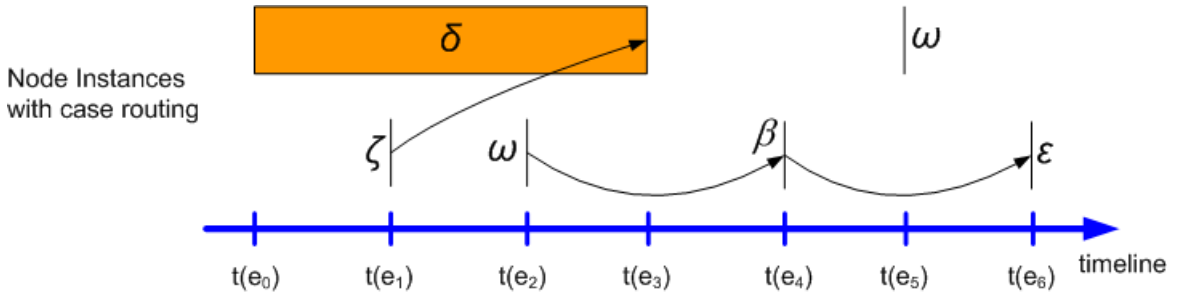


Figure 4.8: Control flow identification of node instances in Example 2

Control flow identification is an important step to determine both join and split semantics of SPD nodes. Split semantics of a node n can be determined by calculating how many successor nodes, represented by their instances, are given process control by an instance of node n . If all successor nodes of n are given process control, then node n has AND-split semantics. If only some of them are given process control, then the node n has OR-split semantics. If only one of them is given process control, then the node n has XOR-semantics.

Our formal approach to identify the type of split semantics of a given SPD node is given as follows:

Definition 4.4.2. (Split Semantics Identification) Let

- $W = (E, ET, A, R, C, t, et, a, r, c)$ be an event log,
- C_W be a set of sequences of events in event log W that represents cases,

- $S = (W, N, L, l_a, l_n)$ be an SPD of the event log W ,
- LA be a look-ahead value ($LA \in \mathbb{N}_1$),
- $tr = \langle \langle e_0, \dots, e_k \rangle \rangle, tr \in C_W$ be a sequence of events in event log W , and
- $m : E \rightarrow N$ be a function mapping an event to an SPD node that is obtained from decomposing tr to maximum fitting substraces, and
- i be an integer, $0 \leq i < k$.

We say that $m(e_i)$ has *split semantics* at time $t(e_i)$ iff $\exists_{i < j \leq \min(k, i+LA)} e_i \succ_c e_j$ and $\nexists_{i < l \leq k} e_i \models e_l$.

The type of the split semantics, given $S_{succ} = \{n_{succ} \in N \mid (m(e_i), n_{succ}) \in L\}$ as the set of successors node of node $m(e_i)$ is²

- AND-split, iff $\forall_{n_{succ} \in S_{succ}}, \exists_{i < j < \min(k, i+LA)} m(e_j) = n_{succ} \wedge e_i \succ_c e_j$
- OR-split, iff AND-split criteria does not hold, and $\exists_{S'_{succ} \subset S_{succ}} |S'_{succ}| > 1 \wedge \forall_{n_{succ} \in S'_{succ}}, \exists_{i < j < \min(k, i+LA)} m(e_j) = n_{succ} \wedge e_i \succ_c e_j$
- XOR-split, iff OR-split criteria does not hold, $\exists_{n_{succ} \in S_{succ}, i < j < \min(k, i+LA)} m(e_j) = n_{succ} \wedge e_i \succ_c e_j$

As an example, see the SPD, the node sequence, the node instances (given the value of look-ahead equal to 3), and the control flow in Figure 4.9. At $t(e_0)$, node instance β passes control to both node instance α and node instance ϵ . From the SPD, both node α and node ϵ are the only successors of node α . Thus, based on Definition 4.4.2, node β has AND-split semantics at time $t(e_0)$. Using the same definition, node γ has XOR-split semantics at time $t(e_3)$. As shown in the SPD, node γ has two successors node: node α and node ϵ . In the node sequence, only node ϵ is within the look-ahead window of e_3 . At time $t(e_3)$, node γ only passes control to one successor out of two possible successors. Therefore, the node has XOR-split semantics at time $t(e_3)$.

Using a similar approach, we can also identify the join semantics of a node at a particular time. Join semantics of a node n is determined by calculating how many predecessor nodes of n , each represented by its instance, gave process control to an event which is mapped to node n . If the event accept process control from all predecessor nodes of n , then the node n has AND-join semantics. If the event accept process control only from some predecessor nodes of n , then the node n has OR-join semantics. If the event accept process control only from one predecessor node of n , then the node n has XOR-join semantics.

The approach to identify join semantics can be formalized as follows:

Definition 4.4.3. (Join Semantics Identification) Let

- $W = (E, ET, A, R, C, t, et, a, r, c)$ be an event log,
- C_W be a set of sequences of events in event log W that represents cases,
- $S = (W, N, L, l_a, l_n)$ be an SPD of the event log W ,
- LA be a look-ahead value ($LA \in \mathbb{N}_1$),

²with $|S|$, we denote a function that returns the number of elements in set S

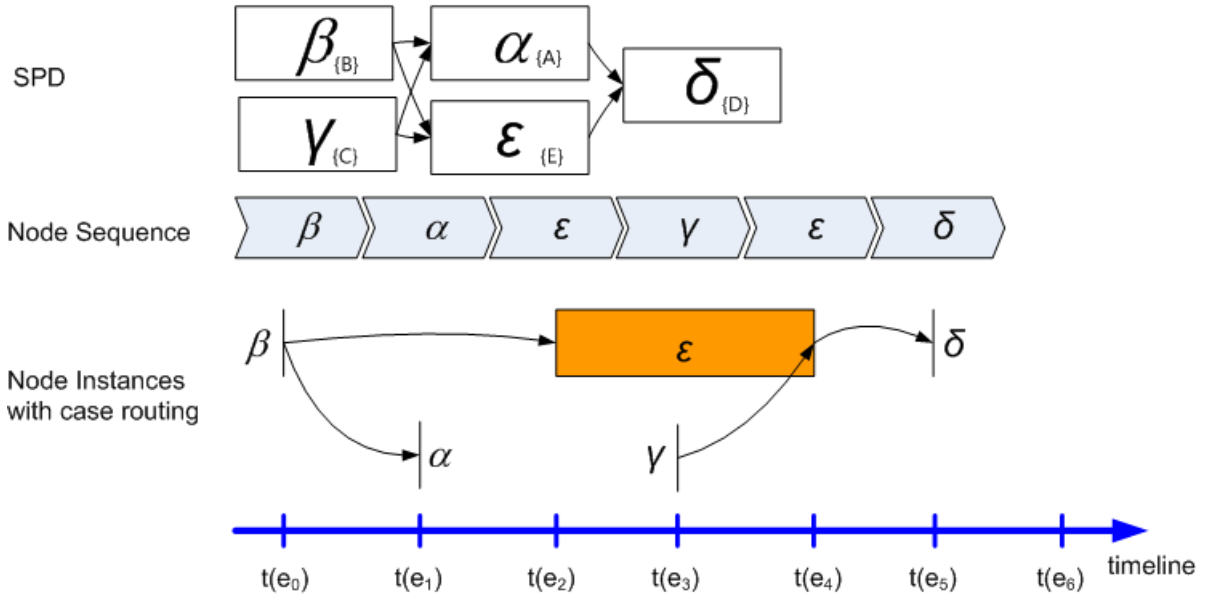


Figure 4.9: Split semantics identification example

- $tr = \langle\langle e_0, \dots, e_k \rangle\rangle, tr \in C_W$ be a sequence of events in event log W , and
- $m : E \rightarrow N$ be a function mapping an event to an SPD node that is obtained from decomposing tr to maximum fitting substraces, and
- j be an integer, $0 < j \leq k$.

We say that $m(e_j)$ has *join semantics* at time $t(e_j)$ iff $\exists_{0 \leq i < j} e_i \succ_c e_j$.

The type of the join semantics, given $S_{pred} = \{n_{pred} \in N \mid (n_{pred}, m(e_j)) \in L\}$ is

- AND-join, iff³ $\forall_{n_{pred} \in S_{pred}}, \exists_{\max(0, j-LA) \leq i < j} m(e_i) = n_{pred} \wedge e_i \succ_c e_j$
- OR-join, iff AND-join criteria does not hold and $\exists_{S'_{pred} \subset S_{pred}} |S'_{pred}| > 1 \wedge \forall_{n_{pred} \in S'_{pred}}, \exists_{\max(0, j-LA) \leq i < j} m(e_i) = n_{pred} \wedge e_i \succ_c e_j$
- XOR-join, iff OR-join criteria does not hold and $\exists_{n_{pred} \in S_{pred}, \max(0, j-LA) \leq i < j} m(e_i) = n_{pred} \wedge e_i \succ_c e_j$

Again consider Figure 4.9. At time $t(e_1)$, node instance α receives control from node instance β . According the SPD in the same figure, there are two possible predecessors node of node α : node β and node γ . Only one of the two predecessors passes control to the node. Therefore, using Definition 4.4.3, node α has XOR-Join semantics at time $t(e_1)$. As another example, consider other node sequence and node instances in Figure 4.10. Suppose that both the sequence and all of the instances are derived from the SPD of Figure 4.9 with a value of look-ahead equal to 3. At time $t(e_2)$, node α receives control from two node instances: β and γ . Thus, at time $t(e_2)$, node α has AND-join semantics as all of its predecessor nodes pass their control to the node. Using Definition 4.4.3, both node ϵ at time $t(e_3)$ and node δ at time $t(e_5)$ also have AND-Join semantics.

³with $\max : \mathbb{Z} \times \mathbb{Z}$, we denote a function that returns the maximum value between two values

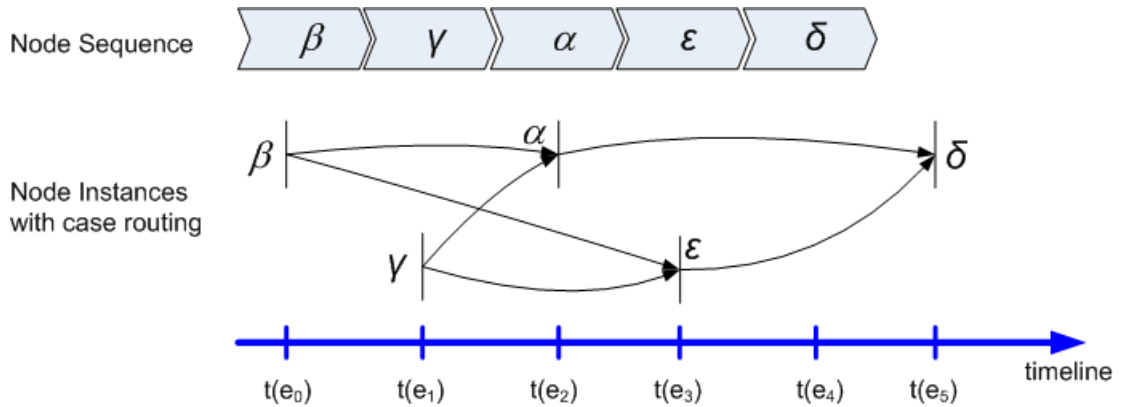


Figure 4.10: Join semantics identification example

4.5 Activity Instances Identification

As already explained in Section 2.1, activity is a work-item that is executed by a resource. In a process, an activity may be executed more than once. For instance, activity “examine patient” in a physiotherapy patient handling process may need to be executed more than once during the process, as a physiotherapy patient may need to go through several examinations before and after the therapy. We refer to an activity which is executed at a point of time during a process as *activity instance*.

A node instance consists of one or more activity instances. A node instance starts at the moment the first activity instance that refers to the node instance is created and ends at the latest moment an activity instance that refers to the node instance ends. An illustration of a node instance and its activity instances is shown in Figure 4.11. The figure shows an instance of node β that refers to three activities, namely A, B, and C. In the figure, an activity instance of each of the referred activity is shown. Notice that an activity instance may have several states. Transition from a state to another state is identified by event with a certain event type. In the figure, an event is labelled by the name of the activity that the event refers to and the type of the event inside a bracket, e.g. A(start) indicates an occurrence of event e which refers to activity A and has an event type “start”.

As stated in Section 2.1, there are several common performance metrics which can be calculated based on activity instances. Unfortunately, in most cases, real-life event logs do not provide any information of activity instances. By a simple heuristic approach, we can derive activity instances from node instances. We assume that activity instances are formed by sequences of events which follows a transactional model of an activity as shown in Figure 2.5. With this assumption, we can define activity instances as sequences of events which refer to the same SPD node instance where for each two consecutive events e, e' in the sequence, the transition of an activity’s state from the event type $et(e)$ to $et(e')$ is allowed according to the transactional model.

Activity instances can be obtained by arranging all events that forms the node instance such that the events which refer to the same activity and precede each other as permitted in the activity transactional model (see Section 2.2.1) are placed in the same activity instance. This also implies that an event can only be a part

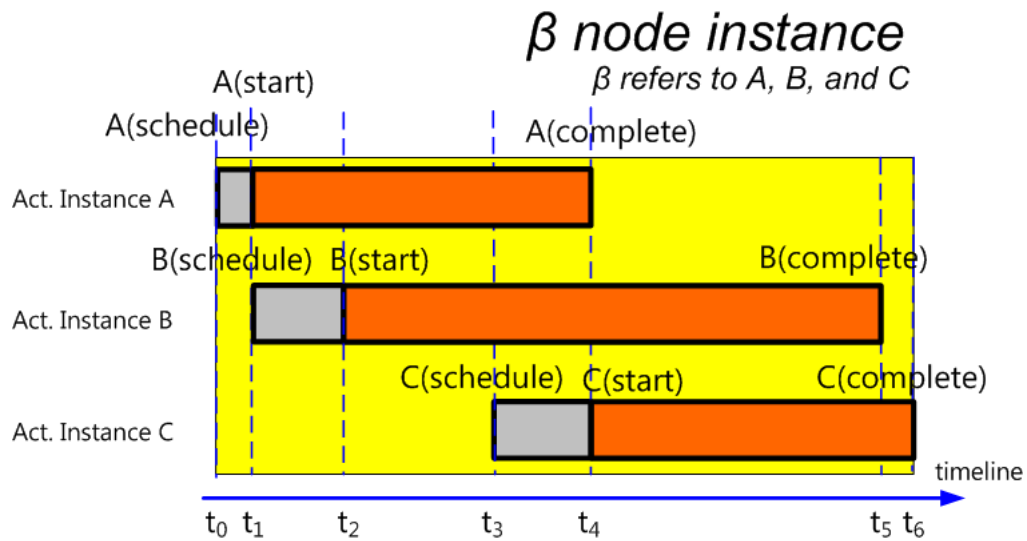


Figure 4.11: Example of activity instances inside a node instance

of one activity instance. However, multiple instances of the same activities can be performed in parallel.

Therefore, we find it is necessary to add the assumption that activities which are started early have a higher priority than other activities which are started later. Thus, we attach events as much as possible to the activity instances which are started earlier before attaching events to activity instances which are started later.

As an example, suppose that we have a node instance that consists of 10 events $\langle e_0, \dots, e_9 \rangle = \langle B(\text{schedule}), C(\text{start}), B(\text{start}), B(\text{complete}), B(\text{schedule}), B(\text{start}), B(\text{start}), C(\text{resume}), B(\text{suspend}), B(\text{complete}) \rangle$. Note that in the notation of the node instance, each event is presented by the name of activity it refers to and its event types given in brackets. Using our reference transactional model that was explained in Section 2.2.1, we can construct activity instances as illustrated in Figure 4.12. New instances are only added if events can not be attached to any existing activity instances. In our example, the occurrence of event $B(\text{start})$ at time t_2 does not create a new activity instance of B. The event is rather grouped to activity instance B-1. Note that in the figure, occurrence of the event $B(\text{schedule})$ at time t_4 initiate a new activity instance B-2 because according to our transactional model, an activity in a “completed” state does not have any other possible event which can be related to it. At time t_8 , event $B(\text{suspend})$ can be grouped to either activity instance B-2 or activity instance B-3 because at the particular time, the last event in both activity instances has event type *start* which can be succeeded by an event with event type *suspend*. This where our second assumption is needed. As activity instance B-2 is started earlier than activity instance B-3, it has higher priority than activity B-3. Therefore, the event is grouped to activity instance B-2.

4.6 Key Performance Indicators

Beside the information that can be obtained directly from event logs, both the concept of node instances and the concept of activity instances provide additional

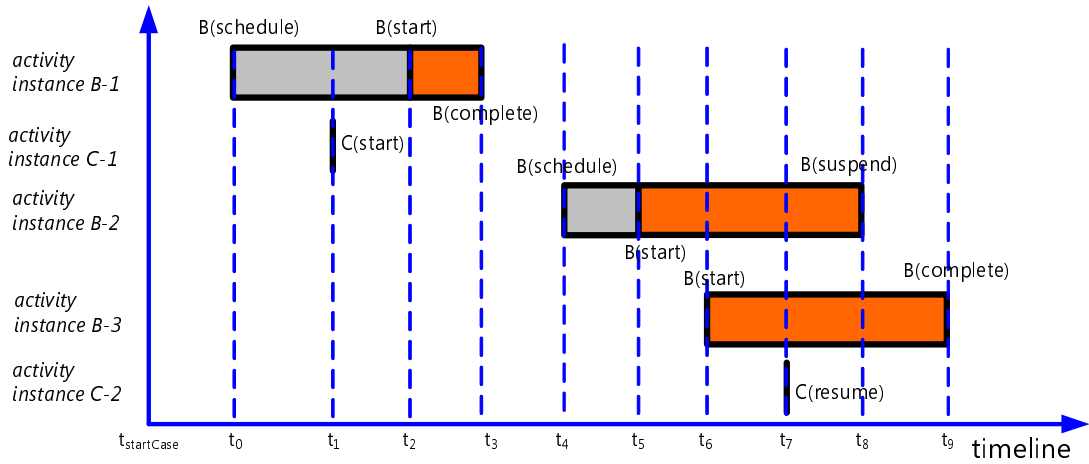


Figure 4.12: Example of constructed activity instances

information that can be used as a basis for our performance analysis. Node instances are aggregations of activity instances. Therefore, node instances provide the views of activities on a higher level of abstraction than activity instances. We argue that the two concepts complement each other during performance analysis of a process.

As an example, suppose that we have a process which is represented by a single SPD node. Two cases from the same process are shown in Figure 4.13(i) and 4.13(ii). All activity instances in both figures are constructed from events which refer to node β . At the level of node instances, the time it takes to finish the whole process in both process instance is the same ($t_5 - t_0$). However, if we analyze the instances at the level of activity instances, we gain additional insights into the cases: The first case has longer “busy” periods than the second case. The average time it takes to finish an activity instance in the process instance described in Figure 4.13(i) is $\frac{(t_3-t_0)+(t_4-t_1)+(t_5-t_2)}{3}$, while the average time it takes to finish an activity instance in the second process instance is $\frac{(t_1-t_0)+(t_2-t_2)+(t_3-t_3)}{3}$ which is obviously less than the first calculation. From the two levels of abstraction that can be provided, which level of abstraction is more relevant compared to the other depends on the purpose of the analysis.

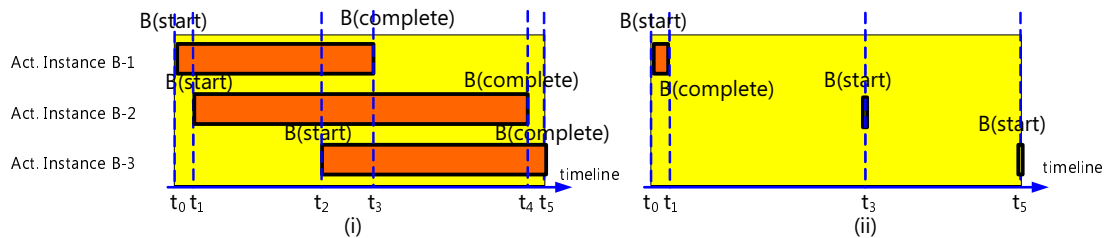


Figure 4.13: Different insights into a process from different level of abstraction

In Section 2.1, various performance metrics to measure business process performance were explained briefly. Based on the performance metrics in Section 2.1 and our approach to replay log on SPDs which was explained in Sections 4.2 to 4.5, we provide several KPIs of business processes which can be grouped into two: case-level

KPIs and model-dependent KPIs. The details of each group are explained in Section 4.6.1 and Section 4.6.2, respectively. In this chapter, the KPIs are explained informally. Some of the KPIs are also formalized in Appendix A.

4.6.1 Case-Level KPIs

Case-level KPIs refer to either performance metrics which are measured on a case level (i.e. process instance) or performance metrics which can be measured from event logs without any need for process model. Let W be an event log and S be an SPD of the log, several KPIs in this category are given as follows:

1. Case throughput time

Throughput time of a case is defined as the total amount of time spent from the moment the first event in the case occurs until the moment the last event in the case occurs. In this thesis, we calculate these statistical values:

- The average case throughput time of all cases in W .
- The minimum case throughput time of all cases in W .
- The maximum case throughput time of all cases in W .
- The standard deviation in the calculation of the average throughput time of all cases in W .
- The average case throughput time of $x\%$ cases with the lowest throughput time in W , where $0 \leq x \leq 100$.
- The average case throughput time of $y\%$ cases with the highest throughput time in W , where $0 \leq y \leq 100$.
- The average case throughput time of remainder $(100 - x - y)\%$ cases in W .

2. Number of cases

The total number of cases in W .

3. Number of traces

The total number of unique sequence of events that represent cases in W , where each event is represented by its corresponding activity and event type.

4. Executed events per resource

The average number of events that is related to a resource in W .

5. Executed activities per resource

The average number of unique activities which is performed by a resource in W .

6. Number of resources per case

The average number of resources that are involved in a case in W .

7. Number of fitting cases

The total number of sequence of events that represent a case which is also maximum fitting subtraces in W . Fitting subtraces are indentified based on S .

8. Arrival rate of cases

The number of cases that arrive per time unit in W .

9. Involved resources in all cases

The total number of resources that are involved in W .

10. Involved teams in all cases

The total number of unique set of resources that are involved in at least a case in W .

4.6.2 Process-model-related KPIs

Some performance metrics of a business process can only be calculated if a process model is known in advance. As an example, in order to calculate delays for an activity (waiting time), one needs to know which other activities are synchronized in a parallel join with the activity. This type of information is typically provided by process models. In this thesis, we use SPD as our process model. Based on our approach to replay event logs in SPDs, we define four categories of process-model-related KPIs: SPD-Node-related KPIs, SPD-Edge-related KPIs, Two-nodes performance KPIs, and Aggregated-activities KPIs. Details of each category are given in the sub sections 4.6.2.1 to 4.6.2.4.

4.6.2.1 SPD-Node-related KPIs

Let W be an event log, C be a set of all cases in event log W , and S be an SPD of the event log. Suppose that a node in the SPD n is selected as the node under inspection, several node-related KPIs that can be measured are given as follows:

1. Node activation frequency

The total number of events in C that refers to node n .

2. Node initialization frequency

The total number of cases in C that starts with an event that refers to node n .

3. Node termination frequency

The total number of cases in C that ends with an event that refers to node n .

4. Number of performers

The total number of unique resources in C that are related to at least an event in C that refers to node n .

5. Relative frequency in a case

The total number of events in C that refers to node n per case.

6. Node throughput time

The time spent to work on an instance of node n . Let ni be an instance of node n , node throughput time is the time spend between the moment the first event that refers to ni occurs and the moment the last event that refers to ni occurs. For this KPI, several statistical values are calculated:

- The average node throughput time of all instances of node n in C .
- The minimum node throughput time of all instances of node n in C .
- The maximum node throughput time of all instances of node n in C .
- The standard deviation in the calculation of the average node throughput time of all instances of node n in C .
- The average node throughput time of $x\%$ instances of node n in C with the lowest node throughput time, where $0 \leq x \leq 100$.
- The average node throughput time of $y\%$ instances of node n in C with the highest node throughput time, where $0 \leq y \leq 100$.
- The average node throughput time of remainder $(100 - x - y)\%$ instances of node n in C .

7. Node waiting time

Suppose that there is a set of node instances NI_{pred} which need to be executed before an instance of node n can be executed in a case in C . Waiting time is defined as the time between the latest moment when an event in NI_{pred} occurs and the moment the first event that refers to the instance of node n occurs. For this KPI, similar statistical values as the statistical values provided for the Node Throughput Time KPI are calculated (e.g. average node waiting time of all instances of node n in C , maximum node waiting time of all instances of node n in C).

8. Node synchronization time

Suppose that there is a set of node instances NI_{pred} which need to be executed before an event which corresponds to node n' can be executed in a case in C , where an instance of node n is also in NI_{pred} . Synchronization time is calculated for node n in the instance as the time between a moment the latest event in NI_{pred} occurs and the moment the latest event in the instance of node n occurs. For this KPI, similar statistical values as the statistical values provided for the Node Throughput Time KPI are calculated (e.g. the average node synchronization time of all instances of node n in C , maximum node synchronization time of all instances of node n in C).

9. AND-join frequency

The total number of times where an instance of node n has AND-join semantics in C .

10. AND-split-frequency

The total number of times where an instance of node n has AND-split semantics in C .

11. OR-join frequency

The total number of times where an instance of node n has OR-join semantics in C .

12. OR-split frequency

The total number of times where an instance of node n has OR-split semantics in C .

13. XOR-join frequency

The total number of times where an instance of node n has XOR-join semantics in C .

14. XOR-split frequency

The total number of times where an instance of node n has XOR-split semantics in C .

4.6.2.2 Edge-related KPIs

Edge-related KPIs are based on process controls which are passed from a node instance to other instance(s). As the controls are routed through SPD edges, we define the KPIs as edge-related KPIs. Let W be an event log, C be a set of all cases in event log W , S be an SPD of the event log, n_1 and n_2 be two nodes in S , and l be an arc from n_1 to n_2 in S . Suppose that l is selected as the edge under inspection, several edge-related KPIs that can be measured are given as follows:

1. Edge frequency

The total number of times a control is passed from instance of node n_1 to instance of node n_2 in C .

2. Edge move time

The move time of edge l is the total time spend to route a process control from an instance of node n_1 to an instance of node n_2 in C . Move time is calculated as the time spend from the moment the last event in the instance of node n_1 occurs until the event in the instance of node n_2 that receives process control from the last event

in the instance of node n_1 occurs. For this KPI, similar statistical values as the Node Throughput Time KPI are calculated (e.g. average move time of all pairs of instances of node n_1 and instances of node n_2 in C where process control is passed from n_1 instances to n_2 instances, the maximum move time of all pairs of instances of node n_1 and instances of node n_2 in C where process control is passed from n_1 instances to n_2 instances).

3. Edge violating frequency

The total number of times in C when the first event e that refers to an instance of node n_1 occurs without gaining any control from other node instance while the first event which refers to an instance of node n_2 already occurred before e and the last event which refers to the same instance of node n_2 has not occurred yet.

4.6.2.3 Two-nodes analysis

During performance analysis, it is also interesting to identify specific relation between two SPD nodes. Let W be an event log, C be a set of all cases in event log W , S be an SPD of the event log, n_1 and n_2 be two nodes in S . Suppose that n_1 is selected as the source node and n_2 is selected as the target node, several performance metrics which can be derived specific to the two nodes are:

1. Source-target pair frequency

The total number of cases in C where two events, each refers to node n_1 and n_2 , respectively, occur.

2. Number of fitting cases

The total number of cases in C where two events, each refers to node n_1 and n_2 , respectively, occur. In addition, sequence of events that form the cases must also be maximum fitting subtraces.

3. Sojourn time

The Sojourn time between node n_1 and node n_2 in a case in C is defined as the time spent between the moment the first event that refers to node n_1 occurs in the case and the moment the first event that refers to node n_2 occurs in the same case. For this KPI, similar statistical values as the Node Throughput Time KPI are calculated (e.g. the average sojourn time between node n_1 and node n_2 of all cases in C where there is an event which refers to node n_1 and another event which refers to node n_2).

4.6.2.4 Aggregated-activities KPIs

Aggregated-activities KPIs provide performance information of SPD nodes based on activity instances within instances of SPD nodes. The KPIs utilize the state of activities as explained in Section 2.1. Hence, the KPIs are calculated by considering

the event types of events that were explained in Section 2.2.1. Recall that in this thesis, we use the transactional model shown in Figure 2.5.

Let W be an event log, C be a set of all cases in event log W , S be an SPD of the event log, n be a node in SPD S , ca be a case in C , ni_n be an instance of node n in ca , AI be a set of activity instances in ni_n , and ai be an activity instance in AI . Several KPIs which can be calculated for node n from the activity instances are given as follows:

1. Frequency of activity instances

The number of activity instances that refer to instance of node n in C .

2. Aggregated-activities throughput time

The throughput time of activity instance ai is the time spent between the moment the first event in ai occurs and the moment the last event in ai occurs. The *aggregated-activities throughput time* of node n in C is the average throughput time of all activity instances that refer to instances of node n in C . Beside this KPI, several other statistical values are calculated:

- The minimum activity instance throughput time of all activity instances that refer to instances of node n in C .
- The maximum activity instance throughput time of all activity instances that refer to instances of node n in C .
- The standard deviation in the calculation of the aggregated-activities throughput time of node n in C .

3. Aggregated-activities queuing time

The queuing time of activity instance ai is the time spent between the moment the first event in ai occurs and the moment the first event that has an event type “start” in ai occurs. If there is no event with event type “start” in ai , the value of queuing time for activity instance ai is 0. The *aggregated-activities queuing time* of node n in C is the average queuing time of all activity instances that refer to instances of node n in C .

For this KPI, other related statistical values are also calculated, such as the minimum/maximum activity instance queuing time of all activity instances that refer to instances of node n in C and the standard deviation in the calculation of the aggregated-activities queuing time of node n in C .

4. Aggregated-activities service time

The service time of activity instance ai is the time spent between the moment the first event that has an event type “start” in ai occurs and the moment the last event in ai occurs. If there is no event with event type “start” in ai , the service time for activity instance ai is defined exactly the same as the throughput time of activity instance ai . The *aggregated-activities service time* of node n in C is the average service time of all activity instances that refer to instances of node n in C .

For this KPI, other related statistical values are also calculated, such as the minimum/maximum activity instance service time of all activity instances that refer to instances of node n in C and the standard deviation in the calculation of the aggregated-activities service time of node n in C .

5. Aggregated-activities start time

The start time of activity instance ai is the time spent between the moment the first event in case ca occurs and the moment the first event in ai occurs. The *aggregated-activities start time* of node n in C is the average start time of all activity instances that refer to instances of node n in C .

For this KPI, other related statistical values are also calculated, such as the minimum/maximum activity instance start time of all activity instances that refer to instances of node n in C and the standard deviation in the calculation of the aggregated-activities start time of node n in C .

6. Aggregated-activities intersection time

Let n' be a node in S , $ni_{n'}$ be an instance of node n' in ca , AI' be a set of activity instances in $ni_{n'}$, and ai' be an activity instance in AI' .

Intersection time between activity instance ai and activity instance ai' is the time span between the latest moment when the first event of each activity instance occurs and the earliest moment when any of the last event of each activity instance occurs. To explain the core idea, we use an example as shown in Figure 4.14.

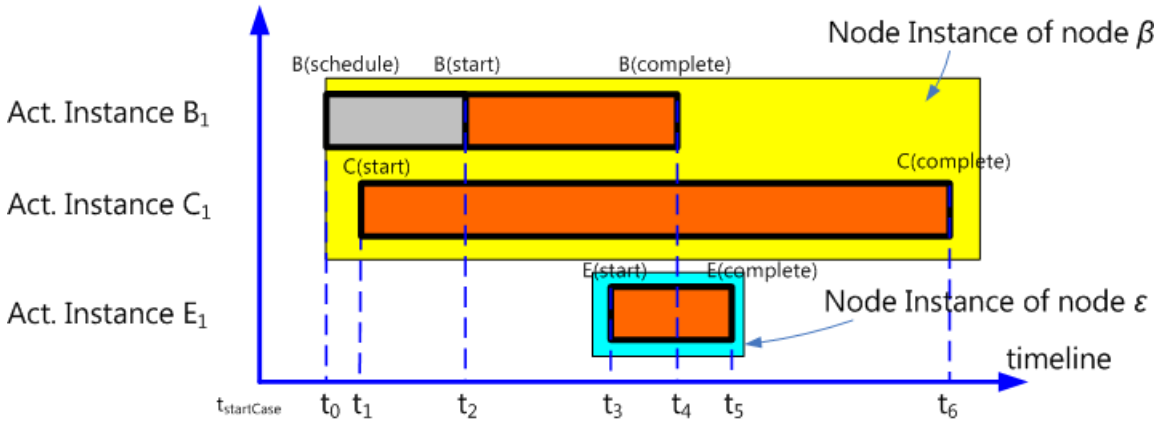


Figure 4.14: Example of two node instances in the same case

Suppose that we want to calculate intersection time between activity instance B_1 and activity instance C_1 in Figure 4.14. Based on the figure, the first event in activity instance C_1 occurs later than the first event in activity instance B_1 ($t_1 > t_0$). The last event in activity instance B_1 occurs earlier than the last event in activity instance C_1 ($t_4 < t_6$). Thus, based on the definition of activity instance intersection time, intersection time between activity instance B_1 and activity instance C_1 is $t_4 - t_1$.

The *aggregated-activities intersection time* between node n and node n' is the average intersection time between all activity instances that refer to instances of node

n in C and all activity instances that refer to instances of node n' in the same C . Other related statistical values are also calculated, such as the minimum/maximum activity instance intersection time of all activity instance intersection time between activity instances that refer to instances of node n in C and activity instances that refer to instances of node n' in C . The standard deviation in the calculation of the aggregated-activities intersection time is also calculated.

The aggregated-activities intersection time gives an indication, how much time that instances of activities in two different node instances overlap. Suppose that we want to calculate intersection time between node β and node ϵ in Figure 4.14. First, we need to calculate the intersection time between each activity instance of node β and each activity instance of node ϵ *which occurs in the same case*. All of the intersection time are summed and then divided by the number of activity instances of both node β and node ϵ *in all cases*.

As example, see again Figure 4.14. Aggregated-activities intersection time between node β and node ϵ is calculated from both intersection between activity instance B_1 and activity instance E_1 ($t_4 - t_3$) and intersection between activity instance C_1 and activity instance E_1 ($t_5 - t_3$). Then, sum of the intersection time is divided by the number of activity instances of node β . The result is then divided again by the number of activity instances of node ϵ . Hence, the aggregated-activities intersection time between node β and node ϵ in Figure 4.14 can be formulated as $\frac{(t_4 - t_3) + (t_5 - t_3)}{2 * 1}$.

Other KPIs can be calculated in more straightforward way than the calculation of aggregated-activities intersection time, as they must only consider activity instances of a single node. For instance, to calculate the throughput time of node β , we calculate the average throughput time of activity instance B_1 and activity instance C_1 . The calculation will provide us the value $\frac{(t_4 - t_0) + (t_6 - t_1)}{2}$ as node β 's aggregated-activities throughput time. Using a similar approach, aggregated-activities waiting time, aggregated-activities service time, and aggregated-activities start time of node β can be calculated.

Chapter 5

Performance Projection

In Chapter 3, we presented SPDs as conceptual process models and we have shown how to calculate performance values based on the SPDs in Chapter 4. In this chapter, we propose two models to project performance information onto. First, in Section 5.1 we present Fuzzy Performance Models (FPDs) which basically are a direct projection of performance information onto SPDs. In Section 5.2, we show how to project performance information when focusing on a specific SPD node in Aggregated Activities Performance Diagrams (AAPDs). Note that only the performance information of important KPIs is projected onto the models.

5.1 Fuzzy Performance Diagram (FPD)

A Fuzzy Performance Diagram (FPD) is a visualization of an SPD with projected performance information. Each node in an FPD has a one-to-one relation with a node in an SPD. Each arc in the FPD also has a one-to-one relation with an arc in an SPD. FPDs are designed to show both performance information and control flow information of a process in an easily interpretable manner. The information is obtained through log replay on an SPD model which was explained in Chapter 4. In an FPD, the information is projected onto each node of the SPD and onto each edge of the SPD (the way this projection is done is highly influenced by both Fuzzy models [26] and extended Petri nets in [16,28]). An example of an FPD is shown in Figure 5.1.

An FPD utilizes shape, size, and colors to provide comprehensive yet compact information about the performance of a process and case routing in a process. An

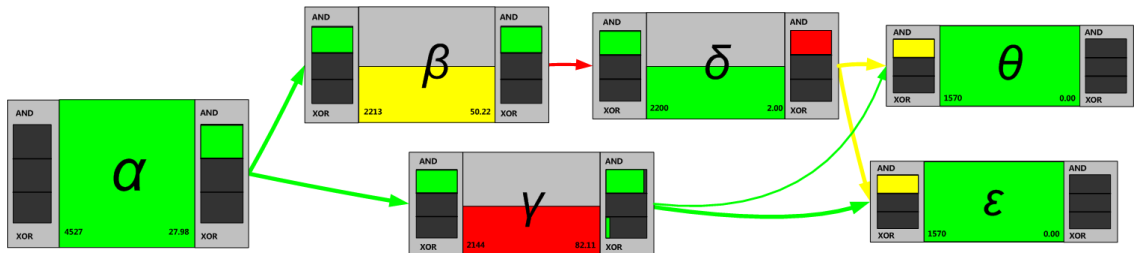


Figure 5.1: Example of an FPD

FPD node shows several types of performance information (see Figure 5.2 and Figure 5.3). The height of an FPD node indicates the *frequency of activity instances* referred to by the node (See Section 4.6.2.4, point 1). The more activities a node refers to are executed, the higher the node. Hence, nodes with a high frequency of activity instances are easily distinguished in an FPD compared to other nodes with a low frequency of activity instances. As frequency often indicates the level of importance of an activity, this feature is useful to help process owners distinguish important nodes from unimportant ones in a process.

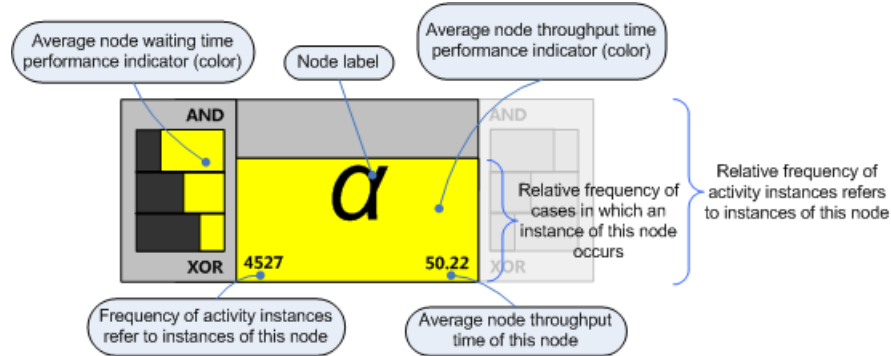


Figure 5.2: Example of an FPD node (1)

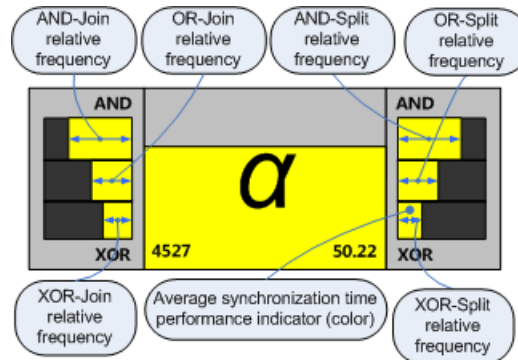


Figure 5.3: Example of an FPD node (2)

An FPD node shows the exact value of two of the most important KPIs: the *frequency of activity instances* referred to by the node and the *average node throughput time* which is calculated based on node instances (see Section 4.6.2.1, point 6). Other KPIs are indicated by bars and colors to avoid information overload on the node. The height of the colored box inside the node shows the ratio of cases in which the node occurs compared to all cases in the event log. The color of the box indicates the average throughput time of the node compared to other nodes. A red color indicates that the average node throughput time value is relatively high compared to the average throughput times of other nodes in the same process. A green color indicates that the value is relatively low, while a yellow color indicates that the value is relatively moderate.

With this visualization, an FPD node which occurs in many cases and has a high frequency of activity instances is easily distinguished from all other nodes due to its

big size and its full color. In contrast, an FPD node which only occurs in several cases and has a low frequency of activity instances is less noticeable, as it has a small size and less-attractive color (dominated by grey color as the base color of an FPD node).

More than just performance information, an FPD node also provides insights into the types of *splits* and *joins semantics* it has, i.e. by indicating to what extent both the split semantics and join semantics tends to *XOR*, *AND*, or *OR* (see Figure 5.2 and Figure 5.3). The type of split/join is indicated by colored horizontal bars on both sides of the node. The width of the colored horizontal bar for certain semantics indicates the tendency of the node towards the semantics (i.e. the percentage of the node classified as XOR, AND, and OR). The tendency is calculated using the semantics frequency of the node as already explained in Section 4.6.2.1, point 9 to 14.

The average *synchronization time* and the average *waiting time* of a node are represented by the colors of the horizontal bars (see Section 4.6.2.1, point 7 and 8). The average waiting time is indicated by the color of the bars on the left side of the node, while the average synchronization time is indicated by the color of the horizontal bars on the right side of the node. A red colored horizontal bar on the left side of the node means that the average waiting time of the node is relatively high compared to the average waiting time of other nodes. A green color on the same horizontal bar indicates a relatively low average waiting time compared to others, while a yellow color indicates an intermediate average waiting time compared to others. The similar concept also holds for color of the horizontal bar on the right side as an indicator of the node's average synchronization time.

An edge in an FPD indicates both case routing and performance. The thicker an edge from a source node to a target node, the more often cases were routed from the source node to the target node (see Figure 5.4). This width corresponds to the *edge frequency* KPI in Section 4.6.2.2, point 1. The color of an edge indicates whether the average time spent on it is relatively high (red), medium (yellow) or low (green) compared to the average time spent on other edges. The average time spent on an edge corresponds to the average *edge move time* KPI that was explained in Section 4.6.2.2, point 2.

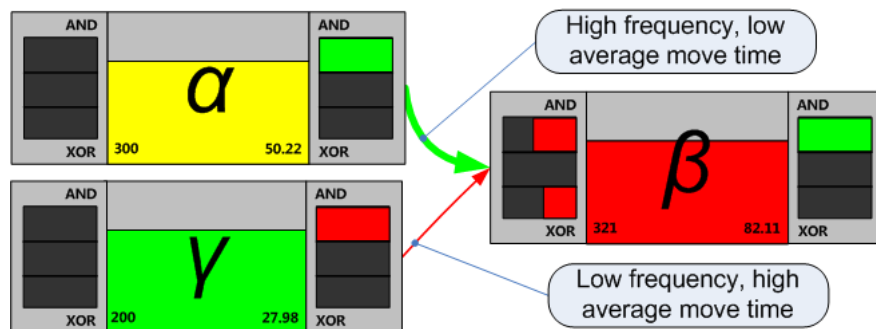


Figure 5.4: Control flow indication in FPD edges (See also Figure 5.3)

Consider an example log with six different activities (A, B, C, D, E, F, G). Activity A always occurs once in every case and requires medium time to be finished. Activities B, C, D, and E only require a small amount of time to be finished. Ac-

tivities C, D, and F do not occur in any cases where either activity B or E occurs, and vice versa. In any case where activities B and E occur, they are executed in sequence. The time spent between the end of execution of activity B and the start of execution of activity E in every case where they occur is always long. Activity G is always performed at the end of all cases. Activities A, C, and D only require a small amount of time compared to activity G. Although activity F does not occur as frequently as activity G, each occurrence of F takes approximately the same amount of time as activity G.

Based on the log, the SPD is constructed as shown in Figure 5.5. Then, the FPD in Figure 5.6 is constructed by replaying the log in the SPD. The FPD figure shows that in total there were 2023 instances of activities which refer to node α . The average node throughput time of node α is 27.98 time units and still considered low compared to the average throughput times of other nodes. More cases were routed from node α to node β than from node α to node δ . The time spent on the route from node α to node β is relatively high compared to the time spent on the route between any other nodes. In the figure, we can also see that both node α and node γ occur in approximately all cases, but not node β and node δ . As indicated by the ratio between the height of yellow-colored box inside node β and the height of node β , instance of node β only occurs in approximately 66.67% of all cases in the log. Although both node α and node δ occur in approximately all cases in the log, the frequency of activity instances refer to node α is bigger than the frequency of activity instances refer to node γ .

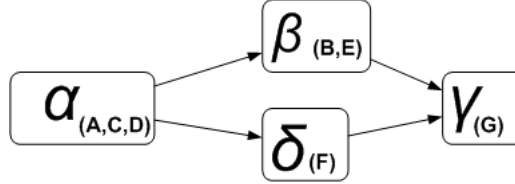


Figure 5.5: Activity mapping in an example SPD

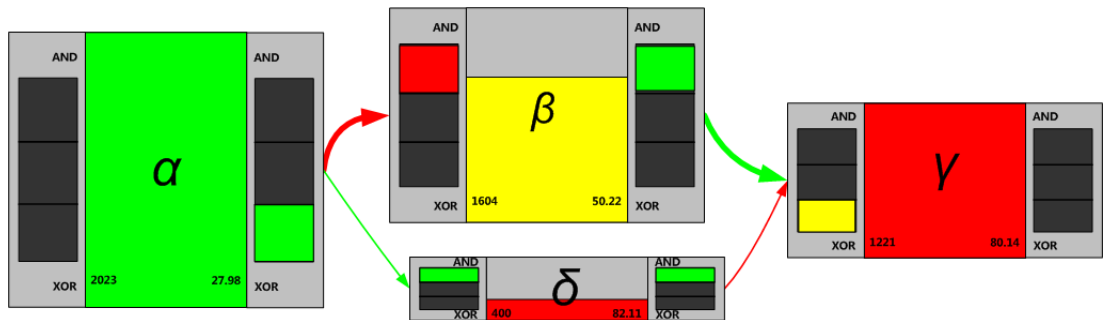


Figure 5.6: Performance and control flow information in an example FPD

The green color of the AND-split bar of node δ indicates that on average, δ 's synchronization time is low compared to the others. However, the time spent to route cases from node δ to node γ is high. In contrast, the time spent to route cases from node β to node γ is low and apparently more frequent. The waiting time of

node γ is medium, as shown by the yellow color. Each FPD node in Figure 5.6 has a strong tendency to a certain join/split semantics, i.e. no nodes indicate a mix of join and split semantics. For instance, node γ only has an XOR-join semantics and no split semantics. During log replay, the node has never indicated any other join semantics apart of XOR-join.

5.2 Aggregated Activities Performance Diagram (AAPD)

Although FPDs provide intuitive insights into the performance of a process, projected onto a given model, there is sometimes a need to focus on a single cluster of activities. Therefore, we developed the Aggregated Activities Performance Diagram (AAPD). An AAPD is a simple diagram consisting of rectangular elements. Each element has a one-to-one relationship with a node in an FPD, hence, an AAPD element refers to an SPD node, and hence, to one or more activities in a log. An AAPD is designed to show the time spent between activities in a process and to show activities which often run in parallel. It is complementary to an FPD.

An example of an AAPD is shown in Figure 5.7. Every AAPD has one focus element which determines the cases that are being considered. Only cases which contain at least one event referred to by the focus element are considered in the AAPD. Besides the focus element, each AAPD contains the other relevant elements which correspond to nodes of the FPD from which it is constructed. Suppose that the AAPD in Figure 5.7 is derived from FPD in Figure 5.6 and the log described in our example. In our example, we selected node β (referring to activities B and E) as our focus element. This implies that node δ (referring to activity F) is not shown, as the activity F does not occur in any case that contains B or E . The other nodes are shown in the AAPD.

Each relevant FPD node is shown in the AAPD as a rectangular element, such that the width indicates the sum of the average *aggregated-activities queuing time* and the average *aggregated-activities service time* for all corresponding activities in the selected cases. For each rectangular element, the width of the area that is colored grey indicates the value of aggregated-activities queuing time, and the width of the area that is colored by other color (green, yellow, or red) indicates the value of average aggregated-activities service time. In an element, the color of the area that is not colored by grey indicates the average *aggregated-activities throughput time* of the element compared to other elements. A red color indicates that the average aggregated-activities throughput time value is relatively high compared to the average aggregated-activities throughput time of other elements. A green color indicates that the value is relatively low, while a yellow color indicates that the value is relatively moderate. Aggregated-activities throughput time, aggregated-activities queuing time and aggregated-activities service time are calculated based on the approach which was explained in Section 4.6.2.4, point 2 to 4.

The height of the element is determined by the percentage of cases in which any of the represented activities occurs, relative to the cases determined by the focus element. In this way, a focus element always has the biggest height and is easily

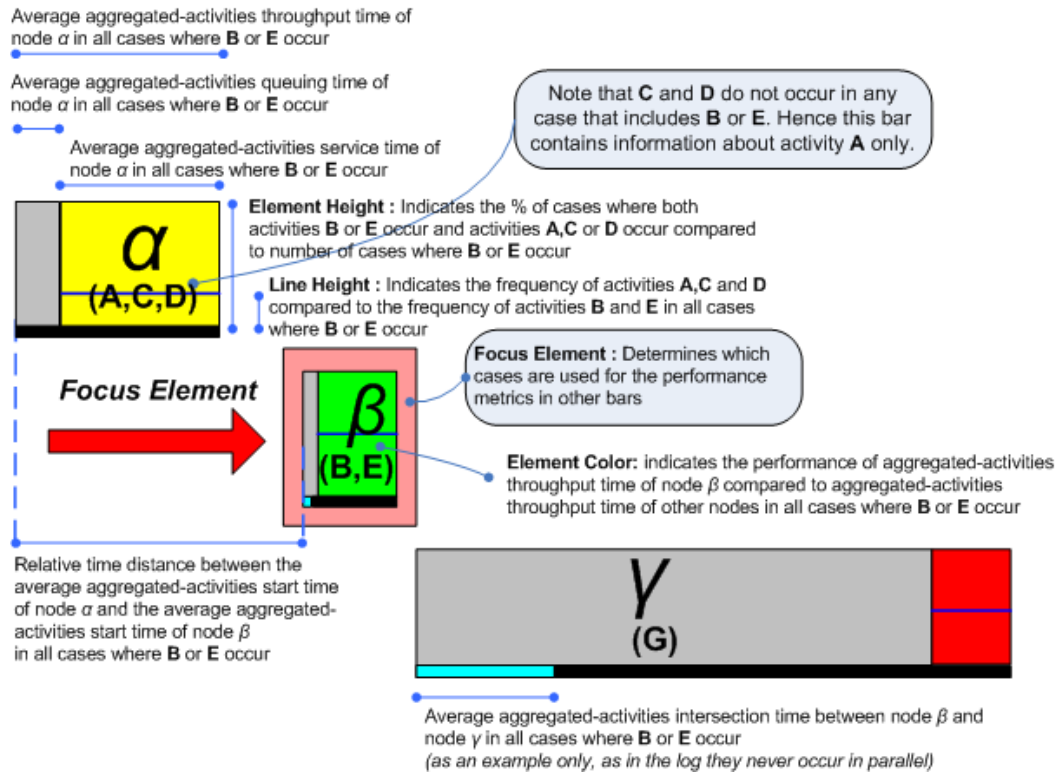


Figure 5.7: Example of an AAPD

distinguished from other elements. In addition, a pink-colored border is appended to the focus element in order to make it more distinguishable. The position along the horizontal axis of all nodes is determined by the relative average *aggregated-activities start time* of each element compared to the average aggregated-activities start time of the focus element (see Section 4.6.2.4, point 5). The horizontal distance between focus element and each other element is given on a logarithmic scale, where the focus element's start time becomes the base point for horizontal distance scaling.

Another indicator in each element of the AAPD is a horizontal line inside the big rectangle. This indicator shows the *frequency of activity instances* which are represented by the element, relative to the frequency of activity instances which are represented by the focus element (see Section 4.6.2.4, point 1). In the example of Figure 5.7, the height of the line in element α (referring to activities **A**, **C**, and **D**) is 50% of element β (referring to activities **B** and **E**) as each case that contains **B** or **E** contains them both, but only one **A**.

Finally, using an indicator below the element, the average time when activities of the element are performed in the same timespan as activities of the focus element is presented. The average intersection time between an element and focus element is calculated based on the *aggregated-activities intersection time* which was explained in Section 4.6.2.4, point 6. Note that in Figure 5.7, aggregated-activities intersection time for element γ (referring to activity **G**) is artificial, as **G** does not occur in parallel with either **B** or **E**. In addition, notice that the color that indicates the average aggregated-activities throughput time of a node in an AAPD may be different than the color that indicates the average node throughput time of the same node in its

corresponding FPD. For example, see the color of FPD node and AAPD element of both node α and node β in our example in Figure 5.6 and Figure 5.7.

As in an FPD node, the AAPD visualization helps a human analyst to distinguish important clusters of activities from unimportant ones. Furthermore, it provides an indication of control flow, i.e. which activities often come before/after another, and how are they conducted (in sequence/parallel). The advantage of AAPDs over FPDs is that they utilize the notion of activity instances. Consider for example again the two process instances in Figure 4.13. Suppose that we only use node instances-based metrics to analyse the instances. In this case, it looks as if both instances require the same effort as they both have the same throughput time. This type of insights into performance of the process are provided by FPDs. However, if we analyze both instances by considering the throughput time of activity instances, we can identify that the process instance 4.13(i) requires more effort to be finished. This type of insights into performance of the process can not be identified by FPDs, but it can be identified using AAPDs.

Chapter 6

Implementation

To evaluate our work, we implemented our approach using the ProM framework¹. Section 6.1 provides an overview about all implemented plugins. Section 6.2 provides an explanation of all SPD plugins which are implemented to visualize SPDs and to map activities onto SPD nodes. Then, the implementation of our performance measurement plugins is explained in Section 6.3. For a more detailed explanation about the implemented plugins, please refer to Appendix B (design) and Appendix C (user manual).

6.1 Plugins Overview

In this section, we provide an overview of all plugins that we implemented based on the ProM architecture which was explained in Section 2.2.3. For convenience, we provide the architecture of the framework that was presented earlier again in Figure 6.1. Based on our analysis in Chapter 3, we proposed SPDs as process models which can describe any processes in an intuitive way. By definition, SPDs are directed graphs. In the ProM architecture, various types of graphs have been implemented, including directed graphs. Thus, we extended the framework's existing directed graph class to create our SPD class. The connection class between SPDs and event logs was also implemented by extending the generic connection class which is provided by ProM. Both connection classes and graph classes are members of the model package. Extension of existing classes in ProM makes our implemented classes recognizable by the framework, i.e. their objects can be stored and retrieved from the *Object Pool* as conveniently as other types of objects which already exist. Details of the SPD plugins is given in Section 6.2.

To calculate performance values as explained in Chapter 4, both an SPD and an event log are required as inputs for log replay. Both SPDs and event logs are objects which are stored in the *Object Pool*. Based on our analysis of the ProM architecture, the most suitable component to retrieve objects from the *Object Pool* and use the objects are the *Plugin* components. Therefore, we implemented our event log replay as a class in the *Plugin* component. In ProM framework, all plugins are located in a plugin package. Details of the log replay class are given in Section 6.3.1.

¹Our work is available in ProM 2008, not in ProM 5.1

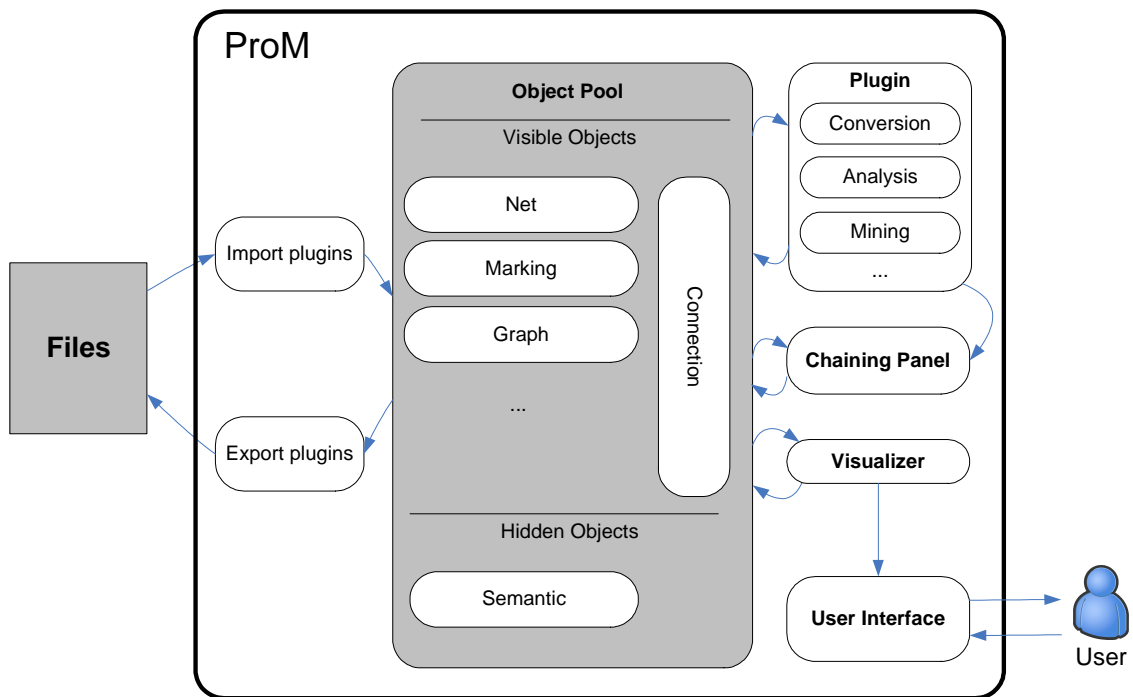


Figure 6.1: The new ProM architecture

Based on our description in Chapter 4 and Chapter 5, log replay produces KPI values, FPDs, and AAPDs. Each of these results are only useful if they can be visualized. Therefore, they are implemented as classes in the model package, the same as SPDs. To visualize FPDs, AAPDs, and all KPI values, we implemented various classes as part of the *Visualizer* component. The implemented FPD visualizer and AAPD visualizer are explained in Section 6.3.2 and Section 6.3.3, respectively.

6.2 SPD Plugins

The GUI of the implemented plug-in is shown in Figure 6.2. SPDs are displayed on the top panel of the GUI. In the right side of the display, there is a zooming panel to adjust SPD visualization. In addition, small panel is placed on top of the zooming panel to help users navigate through the displayed SPDs. Clicking on a displayed SPD node will make the node's label and activities that the node refers to visible on the bottom panel.

SPD-related plugins are implemented to perform these following functions:

- Visualize SPDs. Given an SPD object and an event log, the plugin should be able to visualize the SPD in a friendly Graphical User Interface (GUI). Activities which are mapped to each of its nodes must also be visualized.
- Manually create mappings between activities in a given event log and SPD nodes in a given SPD.

In the latest version of ProM framework, a plug-in may have multiple variants, each with different input parameters. The SPD plug-in is implemented with two variants. The first variant accepts two input parameters: an SPD and an event log. Given an

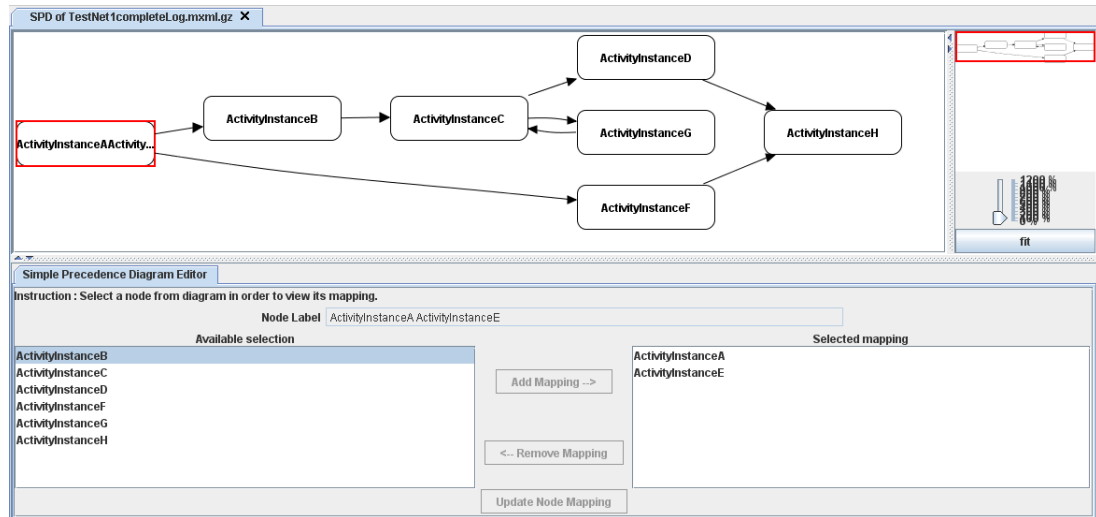


Figure 6.2: SPD Visualization

SPD object and an event log object, the plug-in searches for a connection between the SPD and the event log in the framework's *Object Pool*. If such a connection does not exist, the SPD is visualized in editing mode, e.g. a user can manually map each SPD node in the SPD to activities in the event log. After all nodes in the SPD are mapped to one or more activities in the event log, a connection object is created to store the mapping. If there is already a connection object which connects the event log to the SPD, the SPD is visualized in a read-only mode, e.g. no mapping modification is allowed.

The second variant of the plug-in only accepts an SPD as an input parameter. This variant searches for a connection between the SPD object and any event log object in the framework's *Object Pool*. If there is such a connection, the SPD is visualized together with this mappings. But if there is no such connection, an error message is shown on the screen.

To use the plug-in, the user needs to ensure that both an SPD object and an event log exist in the *Object Pool*. Then, either the SPD or both the SPD and the event log need to be selected and visualized. For a more detailed explanation, please refer to Appendix C.

Currently, a plug-in which mines SPDs from event logs is already available in the nightly build of ProM 2008. The plug-in is named as the SPD Miner and uses the so-called Fuzzy k-medoid clustering [23]. An explanation of the mining algorithm was explained in Section 3.2.2. Other SPD-related plugins such as conversion plug-in and import/export plug-in are still under development.

6.3 Performance Measurement Plugins

Performance measurement plugins are implemented to provide these following functions:

- Replay an event log in an SPD.
- Visualization of:

- FPD and all FPD-related performance values,
- AAPD and all AAPD-related performance values, and
- Global settings of time units and percentage boundaries.

To provide all of the functions, a separate plug-in is implemented for each function. Details of these plugins are given in Sections 6.3.1 to 6.3.4.

6.3.1 Event Log Replay Plug-in

The event log replay plug-in implements the replay log approach which was explained in Sections 4.2 to 4.5. In addition to replaying logs, the plug-in also calculates all KPIs which were explained in Section 4.6. There are two main variants which are implemented for the plug-in. The first variant accepts an SPD and an event log. The second variant only accepts an SPD. The second variant searches through the framework's *Object Pool* to find the event log which is represented by the SPD before it can perform the log replay. If there is no event log which is connected to the SPD, no action is taken by the plug-in.

To use the plug-in, we need an SPD and an event log which is connected to the SPD by a connection object. Before the event log is replayed in the SPD, the plug-in requires two input parameters from users. The first input is the size of look-ahead window during the replay. The second input is the maximal number of states which must be considered before a random approach is performed. It affects the way SPD node are identified. A state corresponds to a trace of SPD nodes which is considered in the process of searching a maximum fitting subtrace. If the number of possible traces exceeds a given upperbound, the plug-in stops its calculation and chooses the latest candidate randomly. Although this step does not was not mentioned in the approach we described in Section 4.2, this limitation is necessary in practice to prevent possible memory problems. Only after the two values are inserted, the plug-in starts to replay the log.

The log replay plug-in produces following output objects:

- An `FPD`, which represents an FPD and performance information projected onto it.
- An `AAPD`, which represents an AAPD and performance information projected onto it.
- The `CaseKPIData`, which stores all case-level KPIs as described in Section 4.6.1.
- The `FPDElementPerformanceMeasurementData`, which stores all SPD-node-related KPIs and SPD-edge-related KPIs as described in Section 4.6.2.1 and Section 4.6.2.2, respectively.
- The `TwoFPDNodesPerformanceData`, which stores all performance metrics which are specific for pairs of SPD nodes as described in Section 4.6.2.3.
- The `GlobalSettingsData`, which stores global configuration to visualize performance information (e.g. time unit).

6.3.2 FPD Visualization Plug-in

The FPD visualization plug-in accepts an FPD object which is produced by the log replay plug-in and visualizes it. This plug-in has only one variant. A screenshot of the visualization is shown in Figure 6.3. The FPD is shown in the top panel together with a zoom panel and small navigation panel on the right side. The bottom panel displays detailed performance information which is related to the FPD.

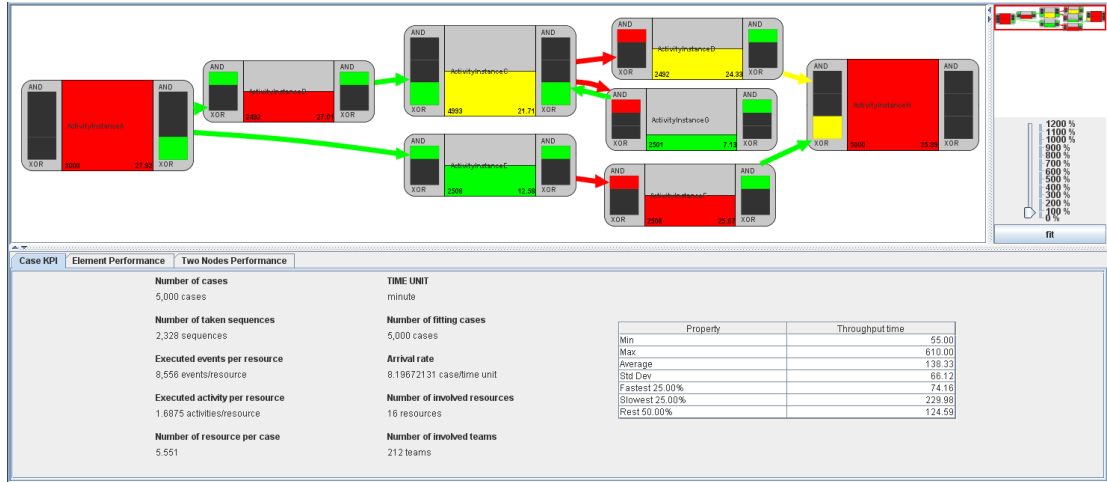


Figure 6.3: FPD visualization

There are three tabs containing performance information that are displayed by the FPD visualization plug-in on the bottom panel. The first tab is the **Case KPI** tab which shows all case level KPIs which were described in Section 4.6.1. The second tab is the **Element Performance** tab which shows either node or edge related KPIs which were described in Section 4.6.2.1 and Section 4.6.2.2. When this tab is active and an element in the displayed FPD is clicked (either an FPD node or an FPD edge), the KPIs related to that element are displayed in the tab.

The last tab is the **Two Nodes Performance** tab which shows the specific KPIs of pair of FPD nodes as was described in Section 4.6.2.3. To use this panel, a pair of nodes needs to be selected before the sojourn time between the pair is displayed in the tab. Selection of the nodes can be performed using either the provided combo boxes or the combination between both the provided radio buttons and the panel that displays FPDs.

To determine the color which indicate performance metrics (e.g. the throughput time color, queuing time color, and synchronization time color of nodes, or move time of edges), lower and upper bound values of each performance metric are calculated based on the minimum, the maximum, and the average of each performance metric value. Suppose that the absolute value of difference between the average value and the minimum value is v_1 , and the absolute value of difference between the average value and the maximum value is v_2 , the lower bound is determined by subtracting the average value with half of the minimum value between v_1 and v_2 . The upper bound is determined by adding the average value with half of the minimum value between v_1 and v_2 .

For example, given an FPD with 5 nodes: $\alpha, \beta, \gamma, \delta$ and ϵ . Suppose that the

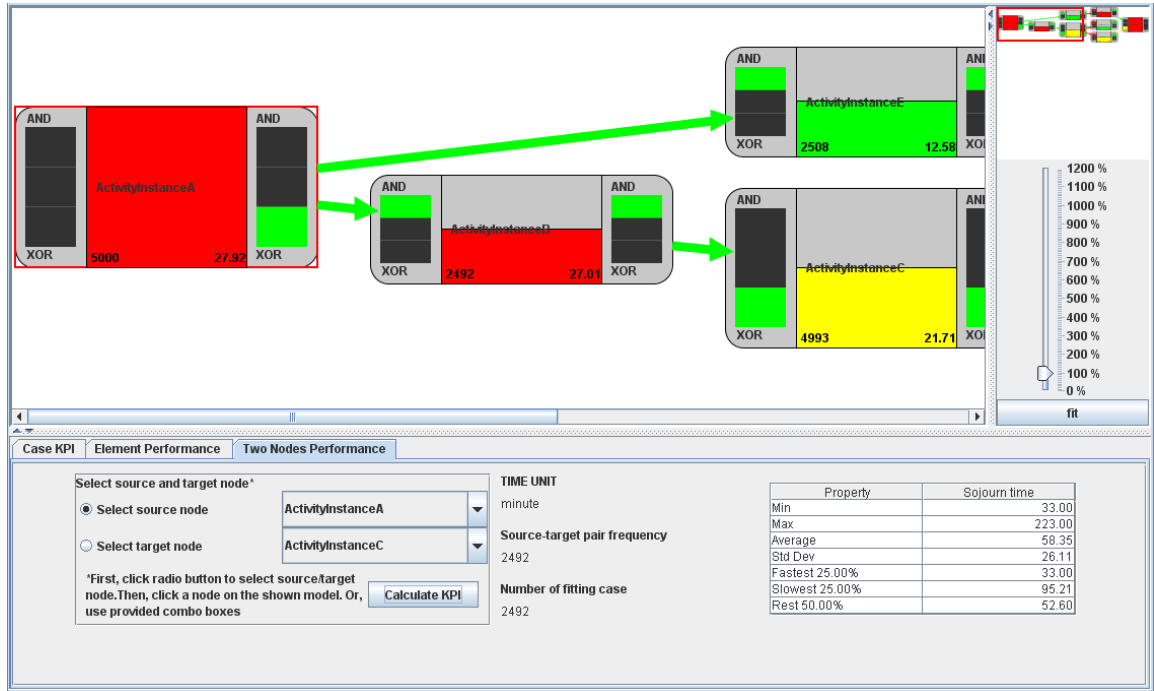


Figure 6.4: Using Two Nodes Performance panel provided by FPD visualization plug-in

Node	α	β	γ	δ	ϵ
case 1	12.0	2.3	4.3	23.0	8.6
case 2	11.0	3.3	2.6	11.0	8.7
case 3	10.0	2.9	3.4	20.0	9.5
case 4	9.0	3.1	1.3	16.0	8.6

Table 6.1: Throughput time table

node throughput time for each node in 4 different cases is shown in Table 6.1. Note that in this example, only one instance occurs for each node in a case. First, we calculate the average node throughput time for each node in all cases. The average node throughput times for nodes $\alpha, \beta, \gamma, \delta$ and ϵ in all cases are 10.5, 2.9, 2.9, 17.5, and 8.85, respectively. Then, we calculate the average of the values avg and search for both the minimum min and the maximum max value. In our example, $avg = \frac{10.5+2.9+2.9+17.5+8.85}{5} = 8.53$. min and max are 2.9 and 17.5, respectively. Next, we compare the distance between avg and min to the distance between avg and max . In our example, distance between avg and min is $8.53 - 2.9 = 5.63$, while the distance between avg and max is $17.5 - 8.53 = 8.97$. We take the minimum distance and divide it by 2 to produce value x . Lower bound $bound_{min}$ for each performance metric is given by $avg - x$, and the upper bound $bound_{max}$ is given by $avg + x$. In our example, $x = \frac{5.63}{2} = 2.815$. Thus, $bound_{min} = 8.53 - 2.815 = 5.715$ and $bound_{max} = 8.53 + 2.815 = 11.345$.

Using the bounds that we calculated before, we can now determine the color of node throughput time of each node. Suppose that the average node throughput time for a node n is avg_n , if $avg_n < bound_{min}$, the color of the node is green. If

$avg_n > bound_{max}$, the color of the node is red. In any other case, the color of the node is yellow. Using our example in Table 6.1, average node throughput time for node α is $\frac{12.0+11.0+10.0+9.0}{4} = 10.5$. As $10.5 \not\leq 5.715$ and $10.5 \not\geq 11.345$, the throughput time color of node α is yellow.

6.3.3 AAPD Visualization Plug-in

The AAPD visualization plug-in accepts an AAPD object which is produced by the log replay plug-in and visualizes it. A screenshot of the visualization is shown in Figure 6.5.

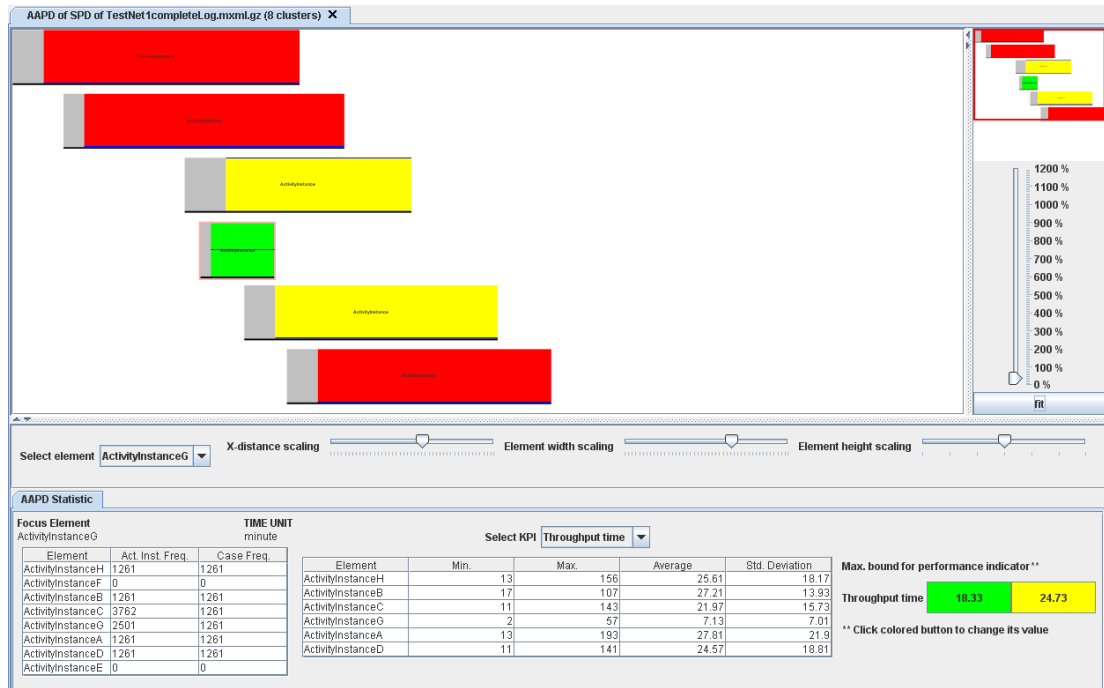
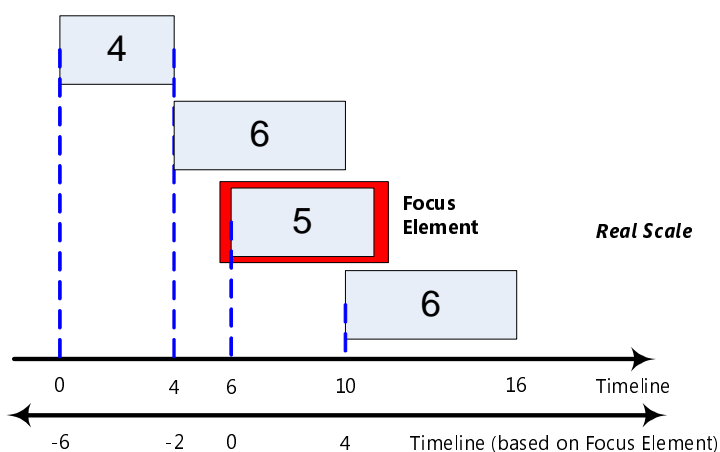


Figure 6.5: AAPD visualization

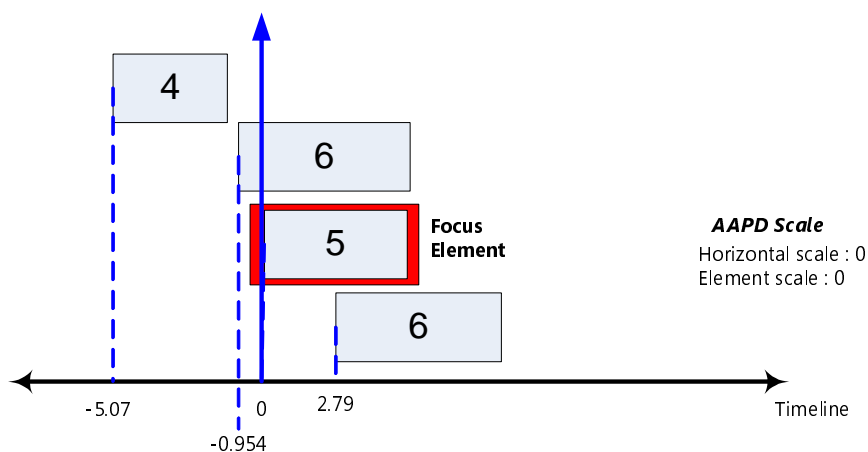
The width, the horizontal placement, and horizontal distance between elements in AAPDs indicates either performance values or precedence relations between activities. In some cases, the distance between elements may be so large that the zoom level we need in order to see all the elements in one screen makes that the elements cannot be recognized anymore. Therefore, we provide three sliders, each to adjust the horizontal distance between elements and focus elements, the width of the elements, and the height of the elements. Note that with these possible adjustments, elements may be visualized differently from the way they should in reality in return of a more meaningful visualization. However, taking each element's average aggregated-activities start time as a reference point for the element, precedence relations between elements are preserved. To visualize AAPDs, the concept of a roadmap is used as a metaphor. Rather than showing all details, a roadmap emphasizes highways and large cities over dirt roads and small towns to provide a more meaningful visualization.

Figure 6.6 shows a comparison between an AAPD with all of its elements' dimensions (size of elements and distance between elements) visualized in a linear scale

and another AAPD with its horizontal distance between elements scaled *logarithmically*. Each AAPD element has a label which indicates the width of the element. The AAPD in Figure 6.6b exposes the difference of horizontal distance between each element and the focus element. This exposure is better shown when the horizontal scaling is used as shown in both Figure 6.7a and Figure 6.7b. Elements with average aggregated-activities start time relatively close to the average aggregated-activities start time of the focus element are shown closer to the focus element, while elements with average aggregated-activities start time relatively far from the focus element's aggregated-activities start time are shown further from the focus element. We argue that this way of visualization provides better insights into one single focus element than if the distance is shown in a linear scale. A user can easily see which elements have the average aggregated-activities start time relatively close to or far from the average aggregated-activities start time of the focus element. In addition, the vertical ordering of elements in an AAPD is ordered by the aggregated-activities start time of each element.

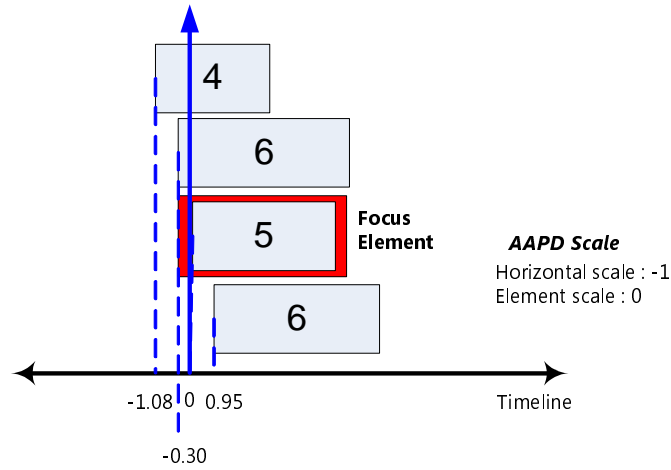


(a) AAPD with horizontal distance between elements in linear scale

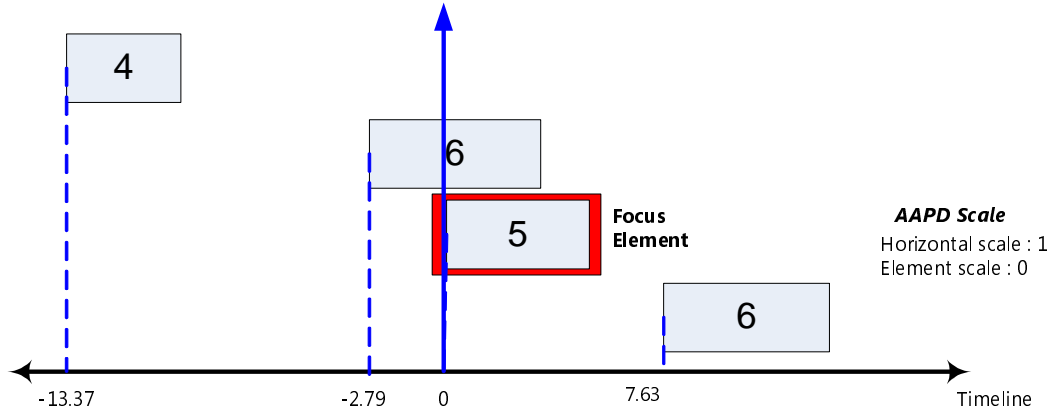


(b) AAPD with horizontal distance between elements in logarithmic scale (horizontal scaling 0, element scaling 0)

Figure 6.6: Example of comparison between AAPD with horizontal distance in linear scale and AAPD with horizontal distance in logarithmic scale



(a) AAPD with horizontal distance between elements in logarithmic scale (horizontal scaling -1, element scaling 0)

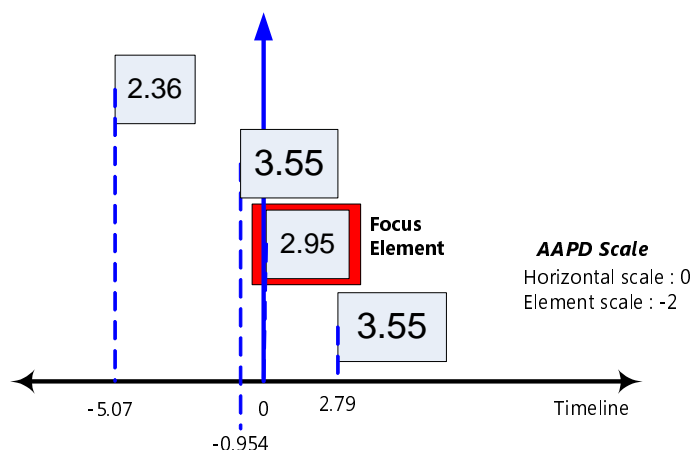


(b) AAPD with horizontal distance between elements in logarithmic scale (horizontal scaling 1, element scaling 0)

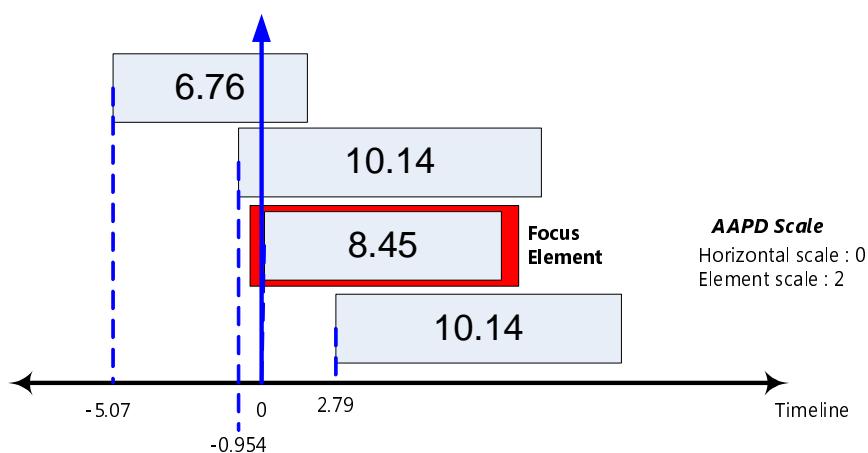
Figure 6.7: Example of adjustment to AAPD horizontal scaling value

The width of each element is scaled *linearly*. With the linear scale, a user can compare precisely both the aggregated-activities queuing time and the aggregated-activities service time of the focus element to both the aggregated-activities queuing time and the aggregated-activities service time of other elements. Examples of adjustment to the element's width scaling of AAPD are shown in Figure 6.8a and Figure 6.8b.

An example of a combination between the horizontal distance scaling and the element's width scaling is shown in Figure 6.9. A user needs to be aware that although the ratio of horizontal distance between elements is different than it is in reality (due to the logarithmic scaling), the ratio of the size of elements is preserved (due to the linear scaling). Thus, the ratio of the average aggregated-activities queuing time, the average aggregated-activities service time, the average aggregated-activities throughput time, and the average aggregated-activities intersection time between elements is always preserved.



(a) AAPD with horizontal distance between elements in logarithmic scale (horizontal scaling 0, element scaling -2)



(b) AAPD with horizontal distance between elements in logarithmic scale (horizontal scaling 0, element scaling 2)

Figure 6.8: Example of adjustment to AAPD element scaling value

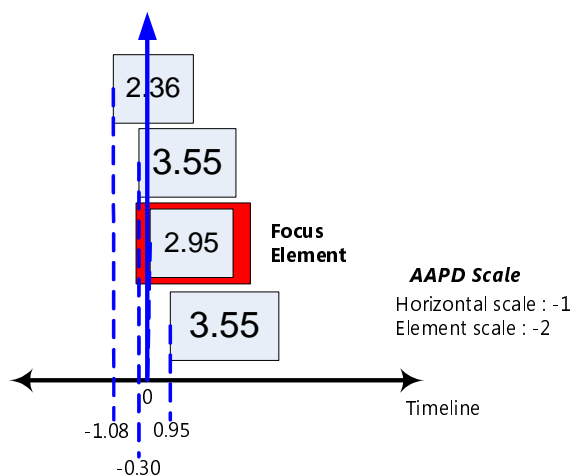


Figure 6.9: Example of adjustment to horizontal scaling value and element scaling value in AAPD shown in Figure 6.6b

The approach to determine the color of throughput time for each element in an AAPD is similar to the approach to determine the color of throughput time for each node in an FPD. The color is determined based on the minimum, the maximum, and the average of average throughput time of all elements in the AAPD.

6.3.4 Global Settings Visualization Plug-in

Event logs may record event timestamps in the order of milliseconds. For analysis purposes, millisecond time units may not be convenient. Therefore, we implemented class `GlobalSettingsData` to store the time unit which can be used by both FPD visualization plug-in and AAPD visualization plug-in. In addition, class `GlobalSettingsData` also stores the values of the percentage boundaries which are needed to visualize statistical performance values in form of tables, such as the table showing statistical values of the case throughput time (see Section 4.6.1, point 1) in Figure 6.10. In the figure, the percentage boundary for fast cases is 32%, while the percentage boundary for slow cases is 12%. This means that the average case throughput time value in column “Throughput time” with corresponding property “Fastest 32.00%” is calculated based on 32% of the cases with the fastest throughput time, and the average case throughput time value in column “Throughput time” with corresponding property “Slowest 12.00%” is calculated based on only 12% of the cases with the slowest throughput time.

Property	Throughput time
Min	0.18
Max	362,893.45
Average	176,783.43
Std Dev	105,755.99
Fastest 32.00%	53,931.72
Slowest 12.00%	333,124.55
Rest 56.00%	213,430.99

Figure 6.10: Example of a detailed statistical measurement

To adjust the values of both the time units in use and the percentage boundaries, we implemented the global settings visualization plug-in. This plug-in provides an interface to modify the values in objects of class `GlobalSettingsData`. A screenshot of the provided interface is shown in Figure 6.11.

Figure 6.11: Interface to set the values in objects of class `GlobalSettingsData`

There are several options for the time unit which are provided by the combo box: millisecond, second, minute, hour, and day. With these options, performance

values can be displayed in the most convenient time unit according to the user. Both the fastest percentage and slowest percentage determine the boundary for statistical values that are shown together with several KPIs, such as the throughput time of a case, the waiting time of an FPD node, and the sojourn time between two FPD nodes.

Chapter 7

Evaluation

To evaluate the correctness, applicability, and performance of our ideas and implementation, several evaluations are performed. Section 7.1 provides an evaluation result for the node and semantics identification which is performed by our replay log approach. In Section 7.2, we provide a case study that shows the applicability of the approach to provide performance insights into real life processes. Section 7.3 provides the performance evaluation result for our plugins.

7.1 Node and Semantics Identification

Our replay approach is started by identifying node sequences. To evaluate this, we created a set of traces of events which are generated from a Petri net. The Petri net is shown in Figure 7.1 and the set of traces is shown in Table 7.1. Using the approach we described in Section 3.2, we obtained an SPD of the net as shown in Figure 7.2. The result of replaying the traces in the SPD is given in Table 7.2. As seen in the table, our approach successfully identified each trace of nodes.

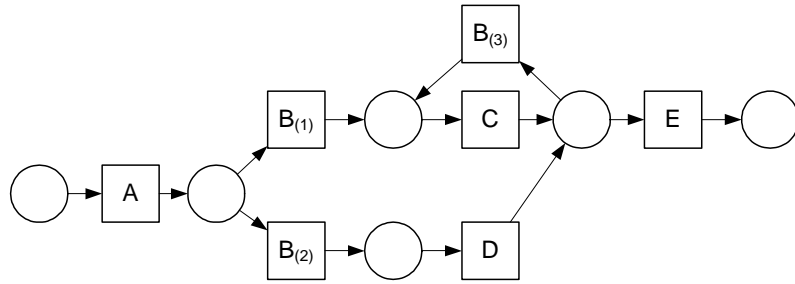


Figure 7.1: Petri net for evaluation purpose

Beside node sequence identification, we also predict semantics of SPD nodes by identifying control which is passed between events of different node instances. In order to measure how accurately the replay determines semantics of SPD nodes, we generate two event logs, each from different Petri nets which are shown in Figure 7.3 and Figure 7.4. Both logs are created to identify whether XOR-split, XOR-join, AND-split, and AND-join semantics can be properly identified by our approach. To

Trace	Generated From
A-B-D-E	A - B ₍₂₎ - D - E
A-B-D-B-C-E	A - B ₍₂₎ - D - B ₍₃₎ - C - E
A-B-C-B-C-E	A - B ₍₁₎ - C - B ₍₃₎ - C - E
A-B-C-E	A - B ₍₁₎ - C - E

Table 7.1: Traces for evaluation purpose

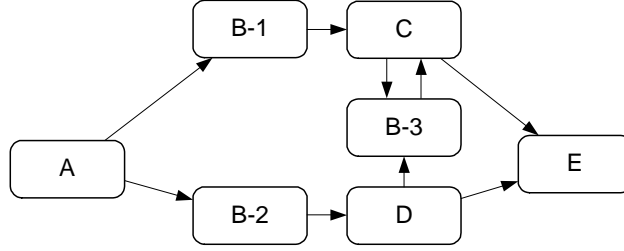


Figure 7.2: SPD of Petri net in Figure 7.1

generate such logs from Petri nets, we used the CPN Tools¹ [33]. The log which is generated from the Petri net in Figure 7.3 is referred to as **TestNet1completeLog**, and the log which is generated by the Petri net in Figure 7.4 is referred to as **TestNet2completeLog**.

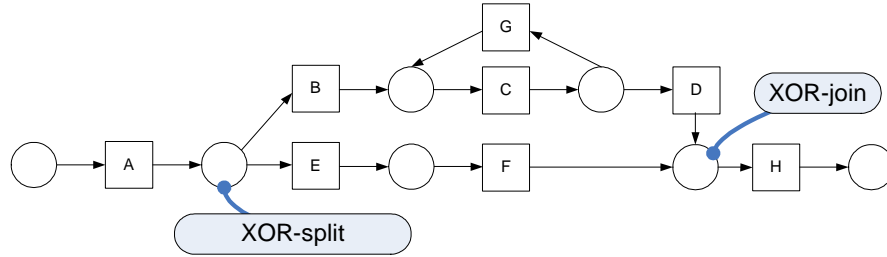


Figure 7.3: Petri net 1 for evaluation

From our experiments, identification of node semantics is sensitive to the value of the look-ahead. Large look-ahead values may not always lead to a better identification, as well as small look-ahead values. For instance, log replay of the trace A-B-D-B-C-E (see Table 7.1) in the SPD of Figure 7.2 provides better semantics prediction when small look-ahead window values are used. Suppose that value greater than 3 is used as a look-ahead value. In that case, node D in Figure 7.2 is mistakenly identified to have AND-split semantics (rather than XOR-split) because both B-3 and E as its successors are within the look-ahead window. With a look-ahead window value smaller than 3, node D is correctly identified as an XOR-split. In contrast, another experiment to replay **TestNet2completeLog** on an SPD in Figure 7.2 shows that the bigger the value of look-ahead window, the better the AND-join semantics can be predicted (see Figure 7.5). Note that in Figure 7.5, changes in

¹<http://wiki.daimi.au.dk/cpntools/cpntools.wiki>

Trace of events	Identified trace of SPD nodes	Correct trace of SPD nodes
A-B-D-E	A - B-2 - D - E	A - B ₍₂₎ - D - E
A-B-D-B-C-E	A - B-2 - D - B-3 - C - E	A - B ₍₂₎ - D - B ₍₃₎ - C - E
A-B-C-B-C-E	A - B-1 - C - B-3 - C - E	A - B ₍₁₎ - C - B ₍₃₎ - C - E
A-B-C-E	A - B-1 - C - E	A - B ₍₁₎ - C - E

Table 7.2: Node identification evaluation

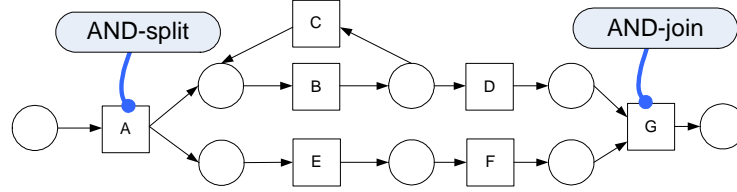


Figure 7.4: Petri net 2 for evaluation

the arc color are caused by automatic layout of FPD nodes. Color of the arcs that pointing to node G do not change during the experiments.

7.2 Real-life Log Analysis

To evaluate whether our proposed approach and model can really provide useful insights into real life processes, we tested our approach against a real life event log. We used an event log called “bezwaar WOZ” from a Dutch municipality [20, 43]. The process described in this log is the process of handling objections filed against real estate taxes.

In order to analyze the process, beside an event log, we also need SPDs of the process. To obtain SPDs, we use the SPD Miner plug-in which is already implemented in ProM 2008. The plug-in works based on the approach given in Section 3.2. One of the advantages of using this plug-in is that we can define any number of clusters (nodes) that we want in the output SPD.

To gain preliminary insights into the way activities are performed in the process, we constructed an SPD with the number of nodes the same as the number of unique activities in the log. In the log, there are 18 uniquely labeled activities. Thus, we obtain an SPD with 18 nodes, each refers to a unique activity. Then, the log is replayed on the obtained SPD to obtain an FPD and an AAPD. The obtained FPD is shown in Figure 7.6.

From the FPD, we gain a visualization of the performance of the process in a low level of abstraction. Only from the FPD, we obtain an information that activities “SYSDELWACHT”, “OZ14 Plan. taxeren”, “OZ10 Horen”, “OZ09 Wacht. Beoord”, and “OZ18 Uitspr wacht” are rarely occurs in any cases in the log. From the height of boxes inside each node in the FPD, we notice that no activities ever occur in all cases. This fact shows that the process depicted by the FPD does not have any exact starting or ending activities. From the FPD, we also obtain information that activities that consume more time to be finished than the others are “OZ08 Beoordelen” (average node throughput time 21.29 days) and “OZ16 Uitspraak” (av-

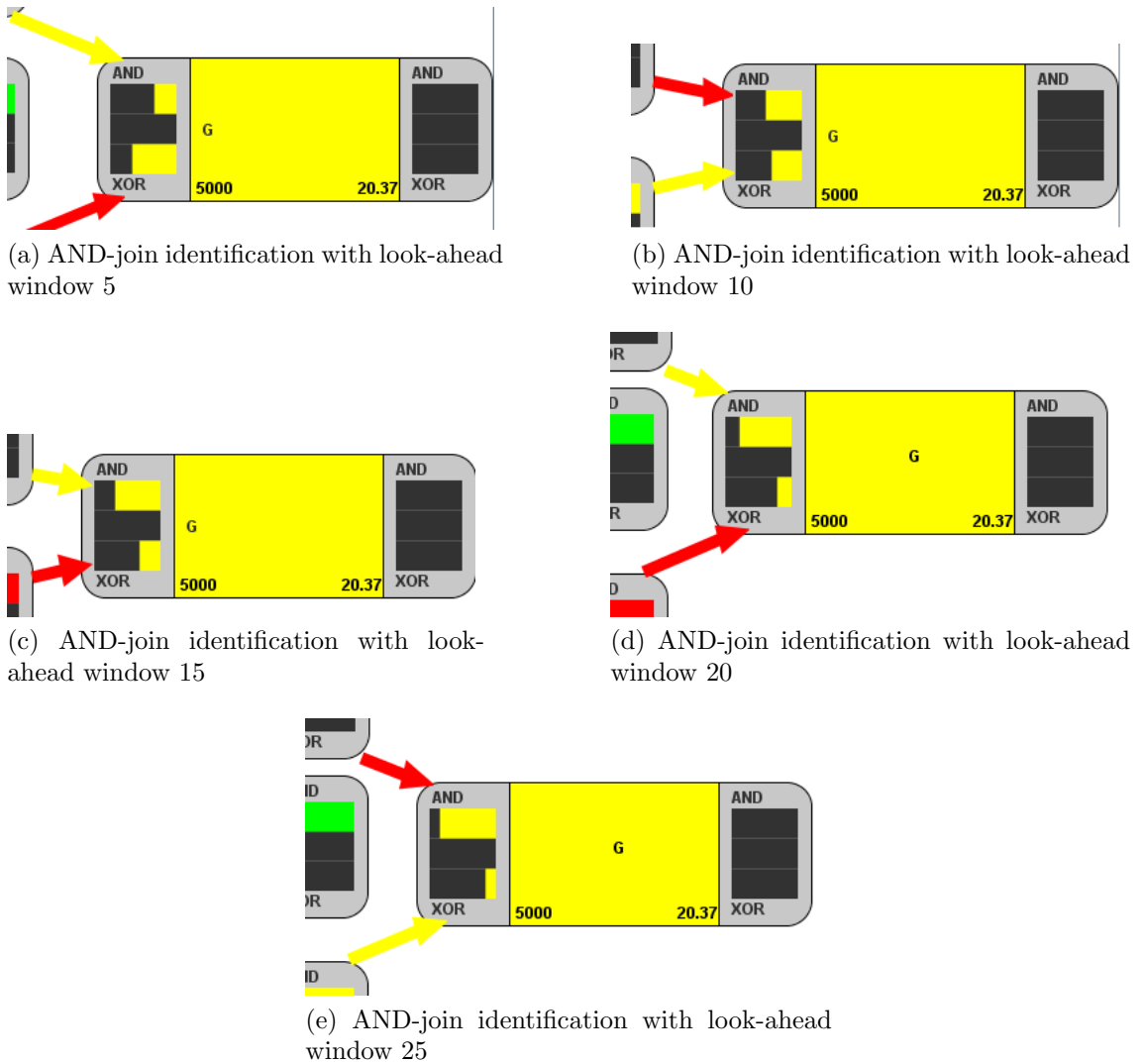


Figure 7.5: Prediction of AND-join semantics of node G which refers to transition G of the Petri net in Figure 7.4 with several look-ahead window values

erage node throughput time 126.87 days). Activity “OZ08 Beoordelen” occurs the most in all cases compared to other activities (1620 cases).

To obtain information about the time ordering of activities, we look at the obtained AAPD of the process. To gain the most global view of the process, we select a focus element which refers to a set of activities that occurs in most cases. Thus, we select element “OZ08 Beoordelen” as the focus element (see Figure 7.7). Note that the horizontal distance between elements in AAPD in Figure 7.7 is already scaled down. Based on the figure, most time during the execution of activities is spent on waiting for resources rather than actually doing the activity. Most of the elements in the AAPD shown in Figure 7.7 have longer average aggregated-activities queuing time than average aggregated-activities service time.

Based on the constructed AAPD, we gain insights into the way activities are executed in the process. Due to its occurrence in most of the cases and its position in the AAPD, activity that refers to element “OZ02 Voorbereiden” (translated as “prepare” in English) may be the earliest activity that starts a normal process.

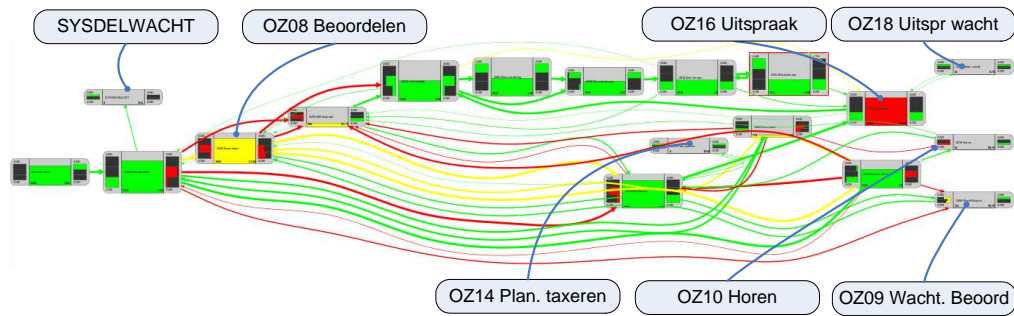


Figure 7.6: The constructed FPD of “bezwaar WOZ” log with 18 nodes

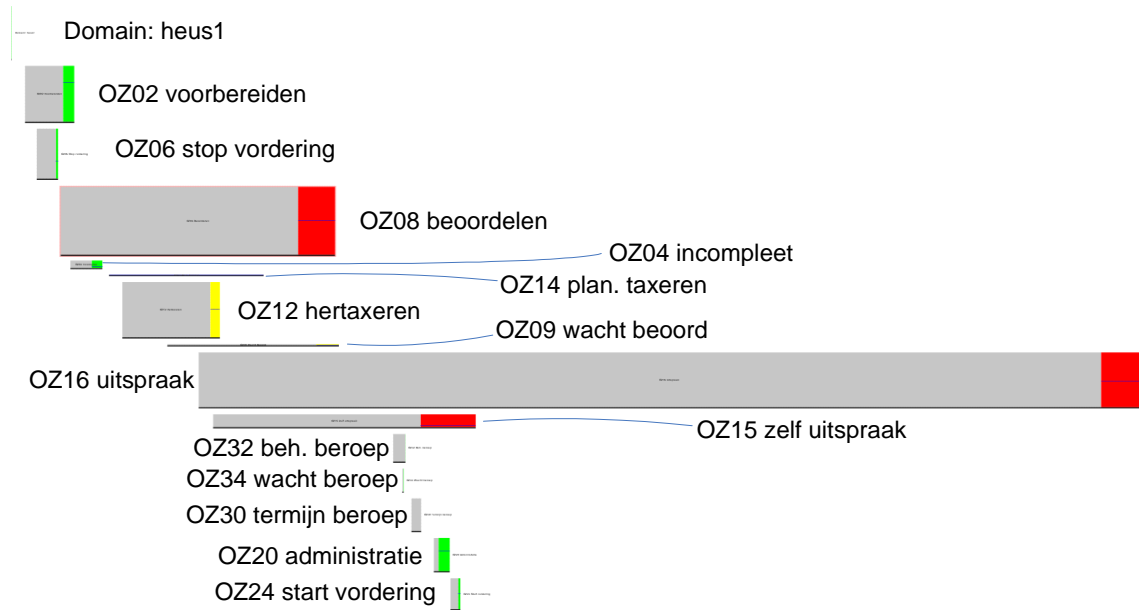


Figure 7.7: The constructed AAPD of “bezwaar WOZ” log with 18 elements

The activity that refers to element “Domain: heus” may only be a dummy activity in the event log, as it has the least aggregated-activities start time and a very small value of aggregated-activities service time compared to others but occurs in many cases. Hence, this activity may be better filtered out during performance calculations. The activity that refers to element “OZ16 uitspraak” has the highest average aggregated throughput time and occurs in almost all cases. Thus, this activity must be important to the process.

Only in a relatively small number of cases, activities that refers to element “OZ14 plan. taxeren” and “OZ09 wacht beoord” (means “awaiting assessment” in English) are performed. This may indicate that the two activities are only executed in exceptions cases. Hence, to analyze exceptional cases, either of the two elements can be selected as the focus element. In contrast to the two elements, the height of both element “OZ12 hertaxeren” (means “re-estimate” in English) and “OZ16 uitspraak” element (means “judgement” in English) are high (almost as high as the height of the focus element), indicating that the activity that refers to either one of the elements occurs at least once in almost all cases.

Elements “OZ32 beh. beroep” (predicted to be “behandeling beroep” which

means “treatment appeal” in English), “OZ34 wacht beroep” (means “wait appeal” in English), and “OZ30 termijn beroep” (means “term action” in English) have low average aggregated-activity throughput time. The difference between each of their average aggregated-activity start time is not so much. The value of average aggregated-activity queuing time in both element “OZ32 beh. beroep” and element “OZ30 termijn beroep” are much bigger than the value of each of their average aggregated-activity service time. This may indicate certain relationship between activities that refers to element “OZ32 beh. beroep” and activities that refers to element “OZ30 termijn beroep”.

Our previous analysis shows that both FPD and AAPD that are constructed from an SPD that describe a process at a relatively low level of abstraction can provide insights into the process. To show that they can also be used to gain insights into processes from SPDs with high level of abstractions, we construct an SPD of event log “bezwaar WOZ” with 5 nodes using the so-called SPD Miner plugin. The constructed SPD is shown in Figure 7.8. As shown in the figure, all nodes in the constructed SPD overlap eachother. They all contain the Domain:heus1, OZ14, OZ18, and SYSDELWACHT activity. All other activities belong to at most 1 node. For convenience, we also label each SPD node with a unique greek letter.

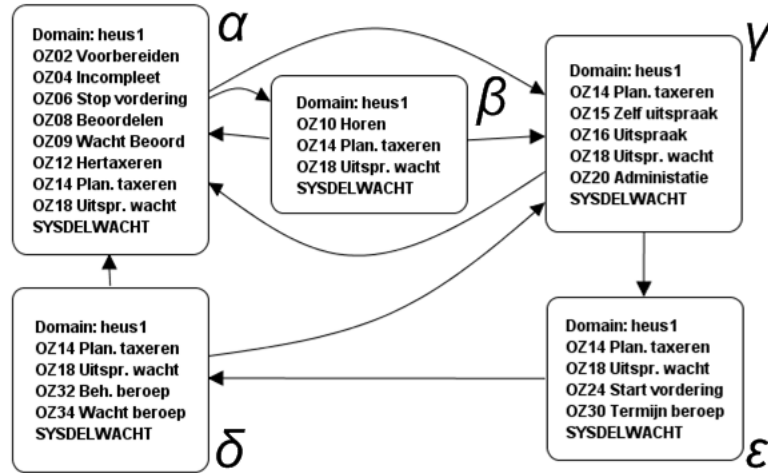


Figure 7.8: The constructed SPD of “bezwaar WOZ” log with 5 nodes

After we obtained an SPD, the next step is to replay the log in the SPD. We use the 4 as our look-ahead value to replay the log in the SPD and set the maximal number of states to 5000. With these values, we obtained the FPD shown in Figure 7.9. Note that in this study case, we use “day” as time unit.

As shown in Figure 7.9, only from the sizes and colors of nodes and edges, important activities and paths in the process can be easily recognized. In addition, the figure also gives us insights into the types of node splits and joins, i.e. by indicating to what extend these tend to be XOR, AND, or OR. The big width of edges from node α to node γ (and vice versa), from node γ to node ϵ , and from node ϵ to node δ indicates that they are important paths in the process. From the four edges, only the edge from node α to node γ has a red color, which indicates its low performance. Thus, efforts to improve the performance of this edge may lead to a

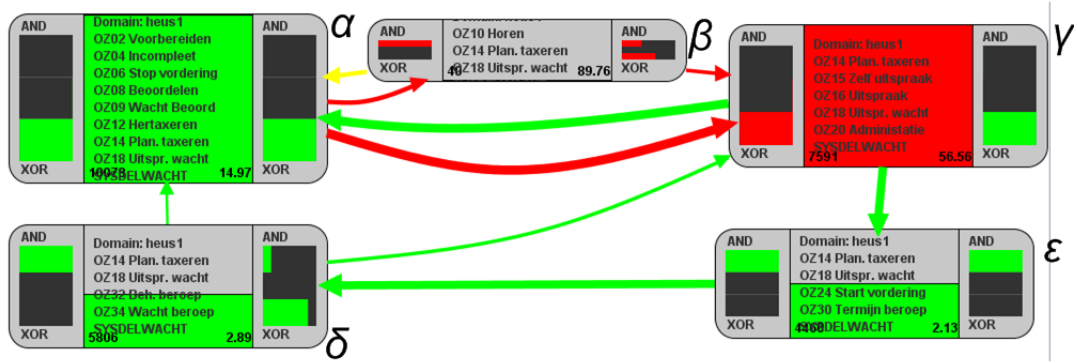


Figure 7.9: The constructed FPD of “bezwaar WOZ” from SPD in Figure 7.8

significant improvement of the overall process.

Node β has a relatively small height compared to the other FPD nodes. A small height indicates that the node does not have as much activity instances referring it as other nodes. Indeed, the node is only referred to by 44 activity instances, far less than other nodes which are referred to by at least 4000 activity instances. The color of the node is mostly grey, which means that it does not occur in as many cases as other nodes. Its only incoming edge from node α has a red color with a small width. The red color means that it takes a relatively long time to pass control from an event which refers to node α to another event which refers to node β , while the small width indicates that the process control is not passed frequently. Each of its outgoing edges has a red color and a yellow color. With the information we gain from the height, color, and edges, node β appears to be referred to by activities which are only executed in exceptional cases.

From all nodes, node γ is easily recognized by its red color which covers all parts of the node and its relatively high height. This indicates that the node instance of node γ occurs frequently in all cases, with a relatively high average throughput time. On average, the throughput time of node γ is 56.56 days, which is indeed high compared to the average throughput time other nodes (except for node β). The node occurs in 1980 cases, almost equal to the total number of cases in the log (1982 cases). With the red color for all of its incoming edges, in addition to the red color for its waiting time, node γ appears to be the bottleneck in the process.

To complete our analysis, consider the AAPD of the process in Figure 7.10. The AAPD shows elements and distances between elements relative to the focus element on a logarithmic scale. In the AAPD, element γ is chosen as the focus element, considering that it is one of the mostly executed nodes (with 7591 activity instances) and appears in almost all cases (1980 cases out of possible 1982). From the figure, we see that on average, activity instances of node α occur before the others, followed by activity instances of node β , γ , δ , and ϵ .

To gain better insights into throughput times of activity instances, we scale the horizontal distance between AAPD elements such that all elements are left aligned (see Figure 7.11). From the figure, we can make a comparison of each element’s throughput time. On average, activity instances of node β have the highest throughput time compared to activity instances of other nodes, followed by activity



Figure 7.10: AAPD of “bezwaar WOZ” log with a default settings (all scaling has a zero value)

instances of node γ . Considering the FPD in Figure 7.9 which shows us that both node β and γ have low performance throughput time, this indicates that the low performance is caused by the low performance of each node’s activity instances. Thus, by improving activity executions within each node, the throughput time of each node in the FPD can be improved.



Figure 7.11: AAPD of “bezwaar WOZ” log with horizontal scaling adjustment

We can also adjust the size of the each AAPD element to gain insights into other aspects of the process. For example, we provide an AAPD with adjusted horizontal scaling, element width, and element height in Figure 7.12. Element β ’s small height compared to others shows that activity instances of node β only appear in a small number of cases compared to other activity instances of other nodes, such as node γ and node α . The figure also shows that for activity instances of node α , β , and γ , most time is spent on waiting for resources rather than actually doing the activity. From the position of blue horizontal line comparison between element α and element γ , we gain additional information that the number of activity instances of node α is slightly higher than the number of activity instances of node γ although activity instances of both node α and node γ appear in approximately equal number of cases.

In this section, we showed that both FPDs and AAPDs that are constructed from SPDs at any level of abstraction can provide insights into performance of processes. With addition of global description about the processes, we can even make educated guesses about the way activities are performed in the processes. However, without sufficient knowledge about the underlying process and a clear motivation behind the way SPDs are constructed, it is difficult to verify all obtained insights into the processes, especially from SPDs that describe processes at relatively high level of abstractions.

7.3 Performance Evaluation

In order to prove that our performance analysis approach is robust enough to be used in real cases, we also performed a performance evaluation of the log replay plug-in that is implemented as a proof of concept of the approach. The evaluation is

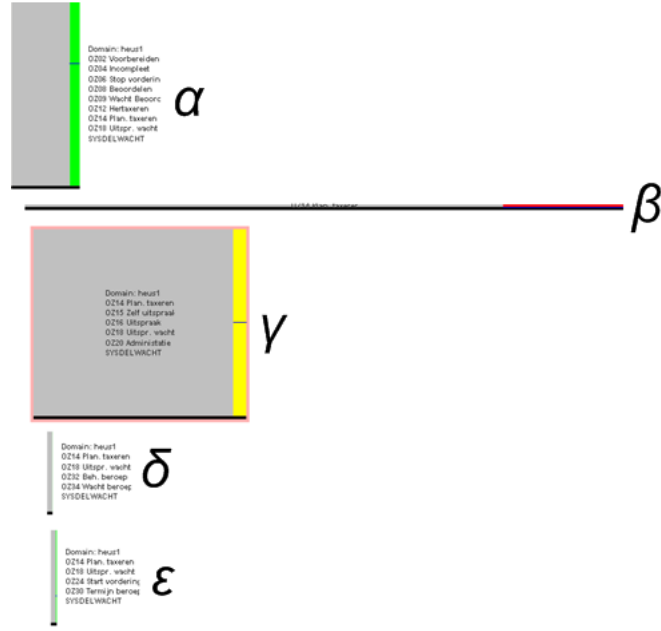


Figure 7.12: AAPD of “bezwaar WOZ” log with horizontal scaling, width scaling, and height scaling adjustment

performed using various event logs. In addition to real-life event logs, we also used the two generated event logs `TestNet1completeLog` and `TestNet2completeLog`. A description of each testing event log is given in Table 7.3².

Event Log	Cases	Events	Activities	Event Types	Events per Case			Activities per case		
					Min	Mean	Max	Min	Mean	Max
logs_BezwaarWOZ	1982	48453	87	7	3	24	104	3	18	39
OriginalLogVestia	223	10664	74	1	4	47	1524	4	29	43
OriginalWMOLog	876	5497	10	1	1	6	12	1	6	8
wholeLog	24	154966	720	2	2820	6456	16250	310	420	508
Bike-allEvents	20	1549	92	10	77	77	81	65	66	68
rws_data_new	14279	119021	15	1	1	8	69	1	7	12
TestNet1completeLog	5000	136904	40	5	12	27	147	12	19	30
TestNet2completeLog	5000	199686	35	5	18	39	138	18	26	35

Table 7.3: Metadata of testing event logs

To test the performance, we used a computer with Intel Core 2 Duo, 2.4 GHz processor and 2 GB memory. Using the same procedure as the analysis of real-life event log in Section 7.2, we evaluate the performance of our implementation. For each event log, we generated several SPDs with different numbers of clusters (nodes) using the SPD miner plug-in. Then, we replayed the event log in the generated SPDs

²The “Activities” column in Table 7.3 indicates the number of unique pairs of activities and event types exists in an event log. Let W be an event log with only one case with two events e and e' , both refers to the same activity. Suppose that event e and event e' have different event types (e.g. e has event type “start” and e' has event type “complete”), the number of unique pairs of activities and event types in W is 2.

and record the time it takes to perform the replay. The result of this performance evaluation is given in Table 7.4.

Event Log	Number of Clusters															
	1	5	6	7	8	9	10	15	18	20	25	45	74	90	180	360
logs BezwaarWOZ	3.843	7.250	8.203	7.515	6.750	8.093	6.454	6.422	8.969	-	-	-	-	-	-	-
OriginalLogVestia	0.594	18.750	20.953	28.594	26.235	26.500	27.765	25.562	17.703	15.281	2.937	7.234	12.969	-	-	-
OriginalWMOLog	0.281	0.391	0.453	0.516	0.500	0.594	0.766	-	-	-	-	-	-	-	-	-
wholeLog	29.594	41.875	51.688	58.735	50.859	69.047	46.625	55.141	127.360	147.047	216.016	470.125	438.719	570.969	1841.843	4360.313
Bike-allEvents	0.359	1.718	1.250	7.157	0.484	6.734	13.156	16.578	1.860	0.375	0.422	0.703	-	-	-	-
nws_data_new	6.765	9.578	9.765	13.281	12.468	13.375	17.187	37.797	-	-	-	-	-	-	-	-
TestNet1completeLog	5.015	8.078	6.750	7.235	8.000	-	-	-	-	-	-	-	-	-	-	-
TestNet2completeLog	7.609	12.719	13.500	15.281	-	-	-	-	-	-	-	-	-	-	-	-

Table 7.4: Performance of replay log plug-in (time unit is given in seconds)

From Table 7.4, our implementation works arguably fast when dealing with real-life event logs. Given an SPD with a number of clusters below or equal to 10, the algorithm can always finish its calculation in less than a minute. In some experiments, the calculation takes even less than one second (see the experiments with the “originalWMOLog” event log). Only the set of experiments with the “wholeLog” event log shows a considerably low performance. But considering the complexity of the “wholeLog” log (approximately 6457 events per case), our implementation has a satisfying performance.

To identify the relation between the plug-in’s performance and the number of clusters, we normalized the values in Table 7.4 by dividing each performance value by the number of events and the number of cases in its corresponding event log. The normalized performance table is given in Table 7.5. As can be seen from the table, the replay plug-in always works better for some logs rather than some other logs, regardless of the number of clusters. This indicates that the performance of the plug-in depends on the complexity of the event log which is determined by the routing of cases (the control flow). We also notice that in the end, the performance of the plug-in always decrease when the number of clusters reach its maximum (when the number of clusters equal to the number of activities). With a maximum number of cluster, node sequence identification can be performed faster as each node is referred to exactly one activity, but KPI calculation takes more time as the number of SPD nodes and the number of SPD edges increases. Figure 7.13 shows the tendency of performance degradation as the number of node increases.

Event Log	Number of Clusters															
	1	5	6	7	8	9	10	15	18	20	25	45	74	90	180	360
logs BezwaarWOZ	0.040	0.075	0.085	0.078	0.070	0.084	0.067	0.067	0.093	-	-	-	-	-	-	-
OriginalLogVestia	0.250	7.885	8.811	12.024	11.032	11.143	11.675	10.749	7.444	6.426	1.235	3.042	5.454	-	-	-
OriginalWMOLog	0.058	0.081	0.094	0.107	0.104	0.123	0.159	-	-	-	-	-	-	-	-	-
wholeLog	7.957	11.259	13.898	15.792	13.675	18.565	12.536	14.826	34.244	39.537	58.082	126.405	117.961	153.520	495.228	1172.384
Bike-allEvents	11.588	55.455	40.349	231.020	15.623	217.366	424.661	535.119	60.039	12.105	13.622	22.692	-	-	-	-
nws_data_new	0.004	0.006	0.006	0.008	0.007	0.008	0.010	0.022	-	-	-	-	-	-	-	-
TestNet1completeLog	0.007	0.012	0.010	0.011	0.012	-	-	-	-	-	-	-	-	-	-	-
TestNet2completeLog	0.008	0.013	0.014	0.015	-	-	-	-	-	-	-	-	-	-	-	-

Table 7.5: Performance of replay log plug-in per case per event (time unit is given in microsecond/ 10^{-6} second)

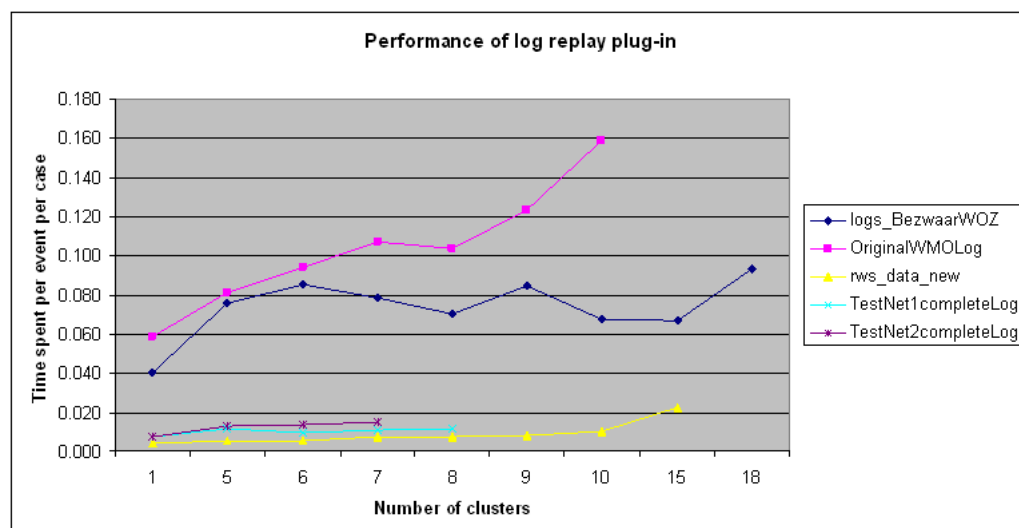


Figure 7.13: Performance degradation with increasing number of clusters (nodes)

Chapter 8

Conclusion and Recommendation

In this master thesis, we analyzed problems encountered when analyzing business process performance. The first problem that is tackled in this master thesis is the problem to model processes intuitively, regardless of their complexity. We showed that even complex business processes can be shown intuitively by process models which support both activity abstraction and aggregation, and have relaxed semantics. In Chapter 3, we proposed Simple Precedence Diagrams (SPDs) as an example of such models. We also presented several possible approaches to obtain SPDs: by converting already existing process models to SPDs, by discovering SPDs directly from the event logs using process discovery techniques, or by simply letting process model experts draw the SPDs. With these three approaches, we argue that we can always obtain SPDs for any given event logs.

The next issue we tackled in this thesis is how to calculate performance information based on the SPD models and the corresponding event log. A heuristic approach which exploits all available information is needed to calculate performance based on the models. In Chapter 4, we provided an approach to replay event logs in SPDs. Based on this log replay, we can obtain both SPD node instances and activity instances which become the basis of our Key Performance Indicators (KPIs). In Chapter 4, we also list various KPIs which can be obtained from either SPD node instances or activity instances. In addition, we also list various commonly calculated KPIs that can be calculated directly from the event logs. By the end of the chapter, we successfully showed that performance information can be calculated from a pair of an SPD and an event log.

The last problem we tackled in this thesis is how to project the performance information onto process models, such that insights into the performance of the process can be obtained intuitively. In Chapter 5, we proposed two models to project performance information onto: Fuzzy Performance Diagrams (FPDs) and Aggregated Activities Performance Diagrams (AAPDs). FPDs provide insights into the bottlenecks of processes by coloring the nodes and edges of an SPD according to their relative performance. AAPDs provide insights into activity instances within processes and show the performance of elements with respect to a single focus element. FPDs and AAPDs are complementary to each other and are tightly related as an AAPD element has a one-to-one relation with an FPD node. We argue that both FPDs and AAPDs provide intuitive performance information of processes to human analysts.

As a proof of concept, we have implemented our solutions in the ProM framework (see Chapter 6). The plugins have been tested using various real life event logs and two simulated event logs (see Chapter 7). The evaluation shows that our proposed approach manages to provide useful performance insights into processes intuitively, regardless of their complexity. The performance of the implemented plugins is reasonably fast and can easily handle real-life cases.

However, there are still some weaknesses of our proposed approach. First, SPDs and their corresponding AAPDs and FPDs can only be interpreted subjectively. Thus, knowledge about the process under consideration and motivations behind the structure of SPDs is crucial. Although we've showed that useful insights into processes can be obtained easily, further verifications involving process owners and process analysts are still needed. Second, the current log replay approach is sensitive to the look-ahead value. An unsuitable look-ahead value may lead to misleading performance measurement results. Moreover, the approach is memory-consuming, specifically when identifying the node sequences of trace of events. Currently, the memory usage is limited by allowing human analyst to set a maximum state space value. Further investigation is needed in order to minimize this risk without involving any human analysts.

Improvements can be made by predicting the look-ahead value from structural analysis of the SPD. Another alternative is to perform a modification to the approach such that the look-ahead value depends on the event under inspection, the construction of process models, and additional information which may be gained from human analysts. Another way of improvements can also be performed by modifying the definition of SPD such that there are two type of nodes in SPD: nodes that refer to sets of activities and nodes that refer to sequences of activities. With this modification, the level of subjectivity to interpret SPDs can be decreased with the risk of complicating the structure needed to describe processes. Note that this improvement may lead to modification of the currently implemented log replay.

As future work, the conversion technique from any process models to SPDs as described in Section 3.2.1 still left to be evaluated and implemented. Implementation of the technique also need to cover the implementation of GPM. In addition, based on our experiments, the most expensive computation of our implemented approach lies within node sequence identification. Other algorithms for finding a decomposition in fitting subtraces may exist such that the performance of our replay plugin can be improved. The implementation of such algorithms is also a part of our future work.

Bibliography

- [1] Automated Business Process Discovery. <http://www.fujitsu.com/global/services/software/interstage/abpd/>.
- [2] Lombardi Teamworks 7. <http://www.lombardisoftware.com/enterprise-bpm-software.php>.
- [3] Metastorm Business Process Management. http://www.metastorm.com/products/business_process_management.asp.
- [4] Oracle BPA Suite. <http://www.oracle.com/technologies/soa/bpa-suite.html>.
- [5] Pegasystem SmartBPM Suite. <http://www.pegasystem.com/products/>.
- [6] Process Discovery - the First Step of BPM. http://www.lombardisoftware.com/downloads/Lombardi_ProcessDiscovery_TheFirstStepofBPM_WP.pdf.
- [7] Savvion Business Manager. http://www.savvion.com/business_manager.
- [8] Software AG WebMethods. <http://www.softwareag.com/corporate/products/wm/default.asp>.
- [9] WebSphere Business Monitor. http://www-01.ibm.com/software/integration/wbmonitor/features/?S_CMP=rnav.
- [10] What is Performance Measurement? <http://www.bpir.com/what-is-performance-measurement-bpir.com.html>.
- [11] Getting Started with Business Process Management, 2007. http://www.softwareag.com/Corporate/Images/SAG_BPM_Get_Started_WP_Dec07-web_tcm16-34221.pdf.
- [12] White paper: Business process improvement. Technical report, Metastorm Inc., July 2007. downloaded on 12 February 2009.
- [13] A. Neely and M. Gregory and K. Platts. Performance measurement system design: A literature review and research agenda. *International Journal of Operations & Production Management*, 25, 2005.
- [14] W.M.P. van der Aalst. Exploring the Process Dimension of Workflow Management. Technical report, Eindhoven University of Technology, Eindhoven, 1997. Computing Science Reports 97/13.
- [15] W.M.P. van der Aalst. Trends in Business Process Analysis: From Verification to Process Mining. In J. Cordeiro J. Cardoso and J. Filipe, editors, *Proceedings of the 9th International Conference on Enterprise Information Systems (ICEIS 2007)*, pages 12–22. Institute for Systems and Technologies of Information, Control and Communication, INSTICC, Medeira, Portugal, 2007.

- [16] W.M.P. van der Aalst and B.F. van Dongen. Discovering Workflow Performance Models from Timed Logs. In S. Tai Y. Han and D. Wikarski, editors, *International Conference on Engineering and Deployment of Cooperative Information Systems (EDCIS 2002)*, volume 2480 of *Lecture Notes in Computer Science*, pages 45–63. Springer-Verlag, 2002.
- [17] W.M.P. van der Aalst, V. Rubin, H.M.W. Verbeek, B.F. van Dongen, E. Kindler, and C.W. Günther. Process mining: a two-step approach to balance between underfitting and overfitting. *Software and Systems Modeling*, 2009.
- [18] W.M.P. van der Aalst, A.J.M.M. Weijters, and L. Maruster. Workflow Mining: Which Processes can be Rediscovered? Technical report, Eindhoven University of Technology, Eindhoven, 2002.
- [19] James C. Bezdek. *Pattern Recognition with Fuzzy Objective Function Algorithms*. Kluwer Academic Publishers, Norwell, MA, USA, 1981.
- [20] A.K. Alves de Medeiros. *Genetic Process Mining*. PhD thesis, Eindhoven University of Technology, Eindhoven, 2006.
- [21] A.K. Alves de Medeiros, A.J.M.M. Weijters, and W.M.P. van der Aalst. *Genetic Process Mining: An Experimental Evaluation*, volume 14 of *Data Mining and Knowledge Discovery*, pages 245–304. Springer Science+Business Media, 2007.
- [22] B.F. van Dongen. *Process Mining and Verification*. PhD thesis, Eindhoven University of Technology, Eindhoven, July 2007.
- [23] B.F. van Dongen and A. Adriansyah. Process Mining: Fuzzy Clustering and Performance Visualization. In *Proceedings of the 5th International Workshop on Business Process Intelligence (BPI 2009)*, 2009 (to appear).
- [24] Equifax. Bpm helps equifax align its global operations. [http://interfacing.com/uploads/File/equifaxcasestory\(1\).pdf](http://interfacing.com/uploads/File/equifaxcasestory(1).pdf).
- [25] C.W. Gunther. *Process Mining in Flexible Environments*. PhD thesis, Eindhoven University of Technology, Eindhoven, December 2008.
- [26] C.W. Gunther and W.M.P. van der Aalst. Fuzzy Mining: Adaptive Process Simplification Based on Multi-perspective Metrics. In P. Dadam G. Alonso and M. Rosemann, editors, *International Conference on Business Process Management (BPM 2007)*, volume 4714 of *Lecture Notes in Computer Science*, pages 328–343. Springer-Verlag, Berlin, 2007.
- [27] J.B. Hill, M. Cantara, M. Kerremans, and D.C. Plummer. Magic Quadrant for Business Process Management Suites, 2009. <http://mediaproducts.gartner.com/reprints/lombardi/article2/article2.html>.
- [28] P.T.G. Hornix. Performance Analysis of Business Processes through Process Mining. Master’s thesis, Eindhoven University of Technology, Eindhoven, 2007.
- [29] M.H. Jansen-Vullers, M.W.N.C. Loosschilder, P.A.M. Kleingeld, and H.A. Reijers. Performance Measures to Evaluate the Impact of Best Practices. In B. Pernici and J.A. Gulla, editors, *Proceedings of Workshops and Doctoral Consortium of the 19th International Conference on Advanced Information Systems Engineering (BPMDs workshop)*, volume 1, pages 359–368. Tapir Academic Press, Trondheim, 2007.

- [30] W. Jeremy. The case for business process management. 2009. <http://www.bptrends.com/publicationfiles/04-09-CS-Case-for-BPM-TIBC0.doc.pdf>.
- [31] J. Jeston and J. Nelis. *Business process management: practical guidelines to successful implementations*. Butterworth-Heinemann, 2006.
- [32] L. Josh. Runner-up:dickerson financial corporation. 2009. <http://www.bptrends.com/publicationfiles/05-09-CS-OMG-BPT-Award-DickersonFinancial.doc.pdf>.
- [33] J. Kurt, L. M. Kristensen, and W. Lisa. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(3-4):213–254, 2007.
- [34] M. J. Lebas. International Journal of Production Economics. In *Proceedings of the 12th International Conference on Production Research*, volume 41 of *International Journal of Production Economics*, pages 23–35. Elsevier B.V., October 1995.
- [35] W. Mark. Walking the bpm talk. how appian uses its own technology to drive its business. <http://www.bptrends.com/publicationfiles/09-08-CS-Appian-MarcWilson.doc-cap-82708.pdf>.
- [36] S. Douwe P. F., L. Fortuin, and Paul P.M. Stoop. Towards consistent performance management systems. *International Journal of Operations & Production Management*, 16:27–37, 1996.
- [37] M. Pesic. *Constraint-Based Workflow Management Systems: Shifting Control to Users*. PhD thesis, Eindhoven University of Technology, Eindhoven, October 2008.
- [38] A. Polyvyanyy, S. Smirnov, and M. Weske. Process model abstraction: A slider approach. *Enterprise Distributed Object Computing Conference, IEEE International*, 0:325–331, 2008.
- [39] J.P. Roberts and R. Andy. Executive Summary Improving Business Processes, 2009. http://www.gartner.com/resources/168000/168003/executive_summary_improving__168003.pdf.
- [40] A. Rozinat, I.S.M. de Jong, C.W. Gunther, and W.M.P. van der Aalst. Process mining of test processes: A case study. Technical report, Eindhoven University of Technology, Eindhoven, 2007. BETA Working Paper Series, WP 220.
- [41] A. Rozinat and W.M.P. van der Aalst. Conformance testing: Measuring the fit and appropriateness of event logs and process models. In C. Bussler, editor, *BPM 2005 Workshops (Workshop on Business Process Intelligence)*, volume 3812 of *Lecture Notes in Computer Science*, pages 163–176. Springer-Verlag, 2006.
- [42] L. Sang and A. Arben. *TQM and BPR: symbiosis and a new approach for integration*, volume 35. Emerald Group Publishing Limited, 1997.
- [43] W.M.P. van der Aalst, M. Dumas, O. Chun, A. Rozinat, and H.M.W. Verbeek. Conformance checking of service behavior. *ACM Trans. Internet Technol.*, 8(3):1–30, 2003.

- [44] W.M.P. van der Aalst, D. Hauschildt, and H.M.W. Verbeek. A Petri-net-based Tool to Analyze Workflows. In B. Farwer, D. Moldt, and M.O. Stehr, editors, *Proceedings of Petri Nets in System Engineering (PNSE'97)*, Lecture Notes in Computer Science, pages 78–90. University of Hamburg, 1997.
- [45] W.M.P. van der Aalst and A.H.M. ter Hofstede. Workflow Patterns: On the Expressive Power of (Petri-net-based) Workflow Languages. In K. Jensen, editor, *Proceedings of the Fourth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2002)*, volume 560 of *DAIMI*, pages 1–20. University of Aarhus, 2002.
- [46] W.M.P. van der Aalst, B.F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A.J.M.M. Weijters. Workflow mining: a survey of issues and approaches. *Data Knowl. Eng.*, 47(2):237–267, 2003.
- [47] B.F. van Dongen, A.K. Alves de Medeiros, H.M.W. Verbeek, A.J.M.M. Weijters, and W.M.P. van der Aalst. *The ProM framework: A New Era in Process Mining Tool Support*, volume 3536 of *Lecture Notes in Computer Science*, pages 444–454. Springer-Verlag, Berlin, 2005.
- [48] A.J.M.M. Weijters, W.M.P. van der Aalst, and A.K. Alves de Medeiros. Process mining with the heuristics miner-algorithm. Technical report, Eindhoven University of Technology, Eindhoven, 2006. BETA Working Paper Series, WP 166.
- [49] M. Weske. *Business process management. Concepts, Languages, Architectures*. Springer-Verlag Berlin Heidelberg, 2007.
- [50] M. zur Muehlen. Business Process Analytics Format (BPAF), February 2009. <http://www.bpm-research.com/wp-content/uploads/2009/02/2009-02-20-wfmc-tc-1015-business-process-analytics-format-r1.pdf>.

Appendix A

KPI Formalization

This appendix provides formalizations of several KPIs that are introduced in Section 4.6 of this report.

A.1 Case-level KPIs

Case-level KPIs refer to either performance metrics which are measured on a case level (i.e. process instance) or performance metrics which can be measured from event logs without any need for process model. Let

- $W = (E, ET, A, R, C, t, et, a, r, c)$ be an event log,
- C_W be a set of sequences of events in event log W that represents cases,
- $S = (W, N, L, l_a, l_n)$ be an SPD of the event log W ,
- $tr = \langle\langle e_0, \dots, e_k \rangle\rangle, tr \in C_W$ be a sequence of events in event log W .

Several KPIs in this category are given as follows:

1. Case throughput time

Let thr_c be a function that accepts a sequence of events $tr = \langle\langle e_0, \dots, e_k \rangle\rangle \in C_W$ and returns $t(e_k) - t(e_0)$, the calculated statistical values are formalized as:

- The average case throughput time: $\frac{\sum_{tr \in C_W} thr_c(tr)}{|C_W|}$.
- The minimum case throughput time¹: $MIN_{tr \in C_W} thr_c(tr)$.
- The maximum case throughput time²: $MAX_{tr \in C_W} thr_c(tr)$.
- The average case throughput time of $x\%$ cases with the lowest throughput time in W , where $0 \leq x \leq 100$:

$$\frac{\sum_{tr \in sub_{min}(C_W, x)} thr_c(tr)}{|sub_{min}(C_W, x)|}, \text{ where } sub_{min}(C_W, x) \subseteq C_W, tr' \in sub_{min}(C_W, x) \Leftrightarrow$$

$$\frac{|\{tr'' \in C_W \mid thr_c(tr'') \leq thr_c(tr')\}|}{|C|} \leq \frac{x}{100}.$$

¹With MIN , we denote a function which returns the minimum value from a set of values

²With MAX , we denote a function which returns the maximum value from a set of values

- The average case throughput time of $y\%$ cases with the highest throughput time in W , where $0 \leq y \leq 100$:

$$\frac{\sum_{tr \in sub_{max}(C_W, y) \text{ } thr_c(tr)}}{|sub_{max}(C_W, y)|}, \text{ where } sub_{max}(C_W, y) \subseteq C_W, tr' \in sub_{max}(C_W, y) \Leftrightarrow \frac{|\{tr'' \in C_W \mid thr_c(tr'') \geq thr_c(tr')\}|}{|C|} \geq \frac{y}{100}.$$
- The average case throughput time of remainder $(100 - x - y)\%$ cases in W :

$$\frac{\sum_{tr \in sub_{rem}(C_W, x, y) \text{ } thr_c(tr)}}{|sub_{rem}(C_W, x, y)|}, \text{ where } sub_{rem}(C_W, x, y) \subseteq C_W, sub_{rem}(C_W, x, y) = C_W \setminus (sub_{min}(C_W, x) \cup sub_{max}(C_W, y)).$$

2. Number of cases

The total number of cases in W : $|C|$.

3. Number of traces

The total number of unique sequence of events that represent cases in W ³:

$|sub_{unique}(C_W)|$, where $tr' \in sub_{unique}(C_W) \Leftrightarrow \nexists_{tr', tr'' \in C_W} tr'' \in sub_{unique}(C_W) \wedge \#_{tr'} = \#_{tr''} \wedge \forall_{0 \leq i < \#_{tr'}} [a(tr'_i) = a(tr''_i) \wedge et(tr'_i) = et(tr''_i)]$.

4. Executed events per resource

The average number of events that is related to a resource in W : $\frac{|E|}{|R|}$.

5. Executed activities per resource

The average number of unique activities which is performed by a resource in W :

$$\frac{\sum_{res \in R} |\{a(e) \mid e \in E \wedge r(e) = res\}|}{|R|}.$$

6. Number of resources per case

The average number of resources that are involved in a case in W : $\frac{\sum_{ca \in C} |\{r(e) \mid e \in E \wedge c(e) = ca\}|}{|C|}$.

7. Number of fitting cases

The total number of sequence of events that represent a case which is also maximum fitting substraces in W according to S : $|\{c_W \in C_W \mid fs(S, c_W) = "true"\}|$, where fs is defined as a function that accepts an SPD and a sequence of events $\langle e_s, \dots, e_{s+l} \rangle$ and returns "true" if the sequence is fitting trace according to Definition 4.2.3 in Section 4.2.

8. Arrival rate of cases

The number of cases that arrive per time unit in W : $\frac{|C|}{MAX_{e \in E}(t(e)) - MIN_{e' \in E}(t(e'))}$

9. Involved resources in all cases

The total number of resources in log event W : $|R|$.

³With $\#(\langle e_0, \dots, e_k \rangle)$, we denote a function $\#$ that returns the length of sequence $\langle e_0, \dots, e_k \rangle$

10. Involved teams in all cases

The total number of unique set of resources that are involved in at least a case in W : $|\{\{r(e) \mid e \in E \wedge c(e) = ca\} \mid ca \in C\}|$.

A.2 Process-model-related KPIs

Process-model-related KPIs can only be calculated if a process model is known in advance. In this section and its subsections, let

- $W = (E, ET, A, R, C, t, et, a, r, c)$ be an event log,
- C_W be a set of sequences of events in event log W that represents cases,
- $S = (W, N, L, l_a, l_n)$ be an SPD of the event log W ,
- LA be a look-ahead value ($LA \in \mathbb{N}_1$),
- $tr = \langle\langle e_0, \dots, e_k \rangle\rangle, tr \in C_W$ be a sequence of events in event log W , and
- $m : E \rightarrow N$ be a function mapping an event to an SPD node that is obtained from decomposing tr to maximum fitting subtraces

Given the formalization above, KPIs for each category of process-model-related KPIs are given in the following sections.

A.2.1 SPD-Node-related KPIs

Suppose that an SPD node $n \in N$ is selected as the node under inspection, formalization of several SPD-node-related KPIs that can be measured are given as follows:

1. Node activation frequency

The total number of events in C that refers to node n : $|\{e \in E \mid m(e) = n\}|$.

2. Node initialization frequency

The total number of cases in C that starts with an event that refers to node n :⁴ $|\{cw \in C_W \mid m(cw_0) = n\}|$.

3. Node termination frequency

The total number of cases in C that ends with an event that refers to node n : $|\{cw \in C_W \mid m(cw_{\#_{cw}-1}) = n\}|$.

4. Number of performers

The total number of unique resources in C that are related to at least an event in C that refers to node n . $|\{r(e) \mid e \in E \wedge m(e) = n\}|$.

⁴With tr_x , we denote the event with index x in the sequence of events tr

5. Relative frequency in a case

The total number of events in C that refers to node n per case: $\frac{|\{e \in E \mid m(e)=n\}|}{|C|}$

6. Node throughput time

The time spent to work on an instance of node n . Let $ni = \langle e_{i_0}, \dots, e_{i_j} \rangle$ be an instance of node $n \in N$ in the sequence of events tr . The throughput time of node instance ni is calculated as $t(e_{i_j}) - t(e_{i_0})$. Let NI be a set of all instances of node n in C . The calculated statistical values are formalized as follows:

- The average node throughput time of all instances of node n in C : $\frac{\sum_{ni \in NI} t(ni_{\#ni-1}) - t(ni_0)}{|NI|}$.
- The minimum node throughput time of all instances of node n in C : $MIN_{ni \in NI} t(ni_{\#ni-1}) - t(ni_0)$
- The maximum node throughput time of all instances of node n in C : $MAX_{ni \in NI} t(ni_{\#ni-1}) - t(ni_0)$
- The average node throughput time of $x\%$ instances of node n in C with the lowest node throughput time, where $0 \leq x \leq 100$: $\frac{\sum_{ni \in sub_{min}(NI, x)} t(ni_{\#ni-1}) - t(ni_0)}{|sub_{min}(NI, x)|}$, where $sub_{min}(NI, x) \subseteq NI$. Let $ni' \in NI$, $ni' \in sub_{min}(NI, x) \Leftrightarrow \frac{|\{ni'' \in NI \mid (t(ni''_{\#ni-1}) - t(ni'_0)) \leq (t(ni'_{\#ni-1}) - t(ni'_0))\}|}{|NI|} \leq \frac{x}{100}$
- The average node throughput time of $y\%$ instances of node n in C with the highest node throughput time, where $0 \leq y \leq 100$: $\frac{\sum_{ni \in sub_{max}(NI, y)} t(ni_{\#ni-1}) - t(ni_0)}{|sub_{max}(NI, y)|}$, where $sub_{max}(NI, y) \subseteq NI$. Let $ni' \in NI$, $ni' \in sub_{max}(NI, y) \Leftrightarrow \frac{|\{ni'' \in NI \mid (t(ni''_{\#ni-1}) - t(ni'_0)) \geq (t(ni'_{\#ni-1}) - t(ni'_0))\}|}{|NI|} \leq \frac{y}{100}$
- The average node throughput time of remainder $(100 - x - y)\%$ instances of node n in C : $\frac{\sum_{ni \in sub_{rem}(NI, x, y)} t(ni_{\#ni-1}) - t(ni_0)}{|sub_{rem}(NI, x, y)|}$, where $sub_{rem}(NI, x, y) = NI \setminus (sub_{min}(NI, x) \cup sub_{max}(NI, y))$.

A.2.2 Edge-related KPIs

Suppose that an SPD edge $l \in L$ which connects a source node n_1 to a destination node n_2 ($n_1, n_2 \in N$) is the edge under inspection, edge-related KPIs which can be calculated for l are given as follows:

1. Edge frequency

The total number of times a control is passed from instance of node n_1 to instance of node n_2 in C : $|\{(e_1, e_2) \in E \times E \mid m(e_1) = n_1 \wedge m(e_2) = n_2 \wedge e_1 \succ_c e_2\}|$.

2. Edge move time

Move time from node n_1 to an instance of node n_2 is the total time spend to route a process control from an instance of node n_1 to an instance of node n_2 in C . Let $MT = \{(e_1, e_2) \in E \times E \mid m(e_1) = n_1 \wedge m(e_2) = n_2 \wedge c(e_1) = c(e_2) \wedge e_1 \succ_c e_2\}$ be

a set of pairs of events where control is passed from an instance of node n_1 to an instance of node n_2 in C . The calculated statistical values are formalized as follows:

- The average move time of all pairs of instances of node n_1 and instances of node n_2 in C where process control is passed from n_1 instances to n_2 instances: $\frac{\sum_{(e_1, e_2) \in MT} t(e_2) - t(e_1)}{|MT|}$.
- The minimum move time of all pairs of instances of node n_1 and instances of node n_2 in C where process control is passed from n_1 instances to n_2 instances: $MIN_{(e_1, e_2) \in MT} t(e_2) - t(e_1)$.
- The maximum move time of all pairs of instances of node n_1 and instances of node n_2 in C where process control is passed from n_1 instances to n_2 instances: $MAX_{(e_1, e_2) \in MT} t(e_2) - t(e_1)$.
- The average move time of $x\%$ pairs of instances of node n_1 and instances of node n_2 in C where process control is passed from n_1 instances to n_2 instances with the lowest move time, $0 \leq x \leq 100$: $\frac{\sum_{(e_1, e_2) \in sub_{min}(MT, x)} t(e_2) - t(e_1)}{|sub_{min}(MT, x)|}$, where $sub_{min}(MT, x) \subseteq MT$. Let $e'_1, e'_2 \in E, (e'_1, e'_2) \in MT, (e'_1, e'_2) \in sub_{min}(MT, x) \Leftrightarrow \frac{|\{(e''_1, e''_2) \in MT \mid (t(e''_2) - t(e''_1)) \leq (t(e'_2) - t(e'_1))\}|}{|MT|} \leq \frac{x}{100}$.
- The average move time of $y\%$ pairs of instances of node n_1 and instances of node n_2 in C where process control is passed from n_1 instances to n_2 instances with the highest move time, $0 \leq y \leq 100$: $\frac{\sum_{(e_1, e_2) \in sub_{max}(MT, y)} t(e_2) - t(e_1)}{|sub_{max}(MT, y)|}$, where $sub_{max}(MT, y) \subseteq MT$. Let $e'_1, e'_2 \in E, (e'_1, e'_2) \in MT, (e'_1, e'_2) \in sub_{max}(MT, y) \Leftrightarrow \frac{|\{(e''_1, e''_2) \in MT \mid (t(e''_2) - t(e''_1)) \geq (t(e'_2) - t(e'_1))\}|}{|MT|} \leq \frac{y}{100}$.
- The average move time of remainder $(100 - x - y)\%$ pairs of instances of node n_1 and instances of node n_2 in C where process control is passed from n_1 instances to n_2 instances: $\frac{\sum_{(e_1, e_2) \in sub_{rem}(MT, x, y)} t(e_2) - t(e_1)}{|sub_{rem}(MT, x, y)|}$, where $sub_{rem}(MT, x, y) = MT \setminus (sub_{min}(MT, x) \cup sub_{max}(MT, y))$.

3. Edge Violating frequency

This KPI can be formalized as $|\{(e_1, e_2, e_3) \in E \times E \times E \mid m(e_2) = n_1 \wedge m(e_1) = m(e_3) = n_2 \wedge e_1 \models e_3 \wedge t(e_1) < t(e_2) < t(e_3) \wedge c(e_1) = c(e_2) = c(e_3) \wedge \nexists_{e_4 \in E} e_4 \succ_c e_2 \wedge \nexists_{e_5 \in E} e_5 \models e_2\}|$.

A.2.3 Two-nodes analysis

Given two nodes n_1 and n_2 ($n_1, n_2 \in N$), several performance metrics which can be derived specific to the two nodes are:

1. Source-target pair frequency

The total number of cases in C where two events, each refers to node n_1 and n_2 , respectively, occur. This can be formalized as: $|\{c(e_1) \mid e_1 \in E \wedge e_2 \in E \wedge m(e_1) = n_1 \wedge m(e_2) = n_2 \wedge c(e_1) = c(e_2)\}|$

2. Number of fitting cases

The total number of unique cases in C where two events, each refers to node n_1 and n_2 occur, and the sequence of events that form the cases are also maximum fitting subtraces. This can be formalized as: $|\{c(e_1)|e_1 \in E \wedge e_2 \in E \wedge m(e_1) = n_1 \wedge m(e_2) = n_2 \wedge c(e_1) = c(e_2) \wedge mfs(S, c(e_1)) = \text{"true"}\}|$, where mfs is defined as a function that accepts an SPD and a sequence of events $\langle e_s, \dots, e_{s+l} \rangle$ and returns "true" if the sequence is fitting trace according to Definition 4.2.3 in Section 4.2.

3. Sojourn time

The Sojourn time between node n_1 and node n_2 in a case in C is defined as the time spent between the moment the first event that refers to node n_1 occurs in the case and the moment the first event that refers to node n_2 occurs in the same case. Let $SJ = \{(e_1, e_2) \in E \times E \mid m(e_1) = n_1 \wedge m(e_2) = n_2 \wedge c(e_1) = c(e_2) \wedge \nexists_{e'_1 \in E} [c(e_1) = c(e'_1) \wedge t(e'_1) < t(e_1) \wedge m(e'_1) = n_1] \wedge \nexists_{e'_2 \in E} [c(e_2) = c(e'_2) \wedge t(e'_2) < t(e_2) \wedge m(e'_2) = n_2]\}$ be a set of pairs of events as basis of sojourn time calculation between node n_1 and node n_2 , the calculated statistical values are formalized as follows:

- The average sojourn time between node n_1 and node n_2 of all cases in C where there is an event which refers to node n_1 and another event which refers to node n_2 : $\frac{\sum_{(e_1, e_2) \in SJ} t(e_2) - t(e_1)}{|SJ|}$.
- The minimum sojourn time between node n_1 and node n_2 of all cases in C where there is an event which refers to node n_1 and another event which refers to node n_2 : $MIN_{(e_1, e_2) \in SJ} t(e_2) - t(e_1)$.
- The maximum sojourn time between node n_1 and node n_2 of all cases in C where there is an event which refers to node n_1 and another event which refers to node n_2 : $MAX_{(e_1, e_2) \in SJ} t(e_2) - t(e_1)$.
- The average sojourn time between node n_1 and node n_2 of $x\%$ cases with the lowest sojourn time in C where there is an event which refers to node n_1 and another event which refers to node n_2 : $\frac{\sum_{(e_1, e_2) \in sub_{min}(SJ, x)} t(e_2) - t(e_1)}{|sub_{min}(SJ, x)|}$, where $sub_{min}(SJ, x) \subseteq SJ$. Let $e'_1, e'_2 \in E, (e'_1, e'_2) \in SJ, (e'_1, e'_2) \in sub_{min}(SJ, x) \Leftrightarrow \frac{|\{(e'_1, e'_2) \in SJ \mid (t(e'_2) - t(e'_1)) \leq (t(e_2) - t(e_1))\}|}{|SJ|} \leq \frac{x}{100}$.
- The average sojourn time between node n_1 and node n_2 of $y\%$ cases with the highest sojourn time in C where there is an event which refers to node n_1 and another event which refers to node n_2 : $\frac{\sum_{(e_1, e_2) \in sub_{max}(SJ, y)} t(e_2) - t(e_1)}{|sub_{max}(SJ, y)|}$, where $sub_{max}(SJ, y) \subseteq SJ$. Let $e'_1, e'_2 \in E, (e'_1, e'_2) \in SJ, (e'_1, e'_2) \in sub_{max}(SJ, y) \Leftrightarrow \frac{|\{(e'_1, e'_2) \in SJ \mid (t(e'_2) - t(e'_1)) \geq (t(e_2) - t(e_1))\}|}{|SJ|} \leq \frac{y}{100}$.
- The average sojourn time between node n_1 and node n_2 of remainder $(100 - x - y)\%$ cases in C where there is an event which refers to node n_1 and another event which refers to node n_2 : $\frac{\sum_{(e_1, e_2) \in sub_{rem}(SJ, x, y)} t(e_2) - t(e_1)}{|sub_{rem}(SJ, x, y)|}$, where $sub_{rem}(SJ, x, y) = SJ \setminus (sub_{min}(SJ, x) \cup sub_{max}(SJ, y))$.

Appendix B

Implementation Design

In this appendix, the architectural design and the classes for all implemented plugins are explained. We divide the explanations to two sections. Section B.1 provides explanations about SPD plugins whose main functions are to visualize SPDs and to provide a convenient GUI to map SPD nodes to activities in event logs. Section B.2 covers all plugins related to the log replay plug-in, including the visualization plug-in of both FPD and AAPD.

B.1 SPD Plug-in

An SPD is basically a directed graph consisting of nodes and arcs. In the ProM framework, an abstract implementation of directed graphs is already provided inside package `org.processmining.models.graphbased.directed`. Therefore, to implement SPDs, we extend the classes and interfaces in the package as shown in Figure B.1.

Abstraction of a directed graph is provided by the class `DirectedGraph`. The class uses the class `AbstractDirectedGraphNode` and the class `AbstractDirectedGraphEdge` as an abstraction of graph node and edge, respectively. All classes which extend the class `DirectedGraph` are mostly implemented as interfaces. Therefore, we also implement SPD interface as an extension of the class `DirectedGraph`. The interface is implemented by the class `SPDImpl`, which then uses the abstract class `SPDNode` as representation of an SPD node and the abstract class `SPDEdge` as a representation of an SPD edge. An instantiable class for node of the graph is implemented as class `SPDNodeElement`, while an instantiable class for the edge is implemented as class `SPDArcElement`. In addition, as most directed graphs in ProM have a factory class, the class `SPDFactory` is also implemented as a factory class for SPD.

To store the mapping from nodes in an SPD to activities in an event log, the class `LogSPDConnection` is created as an extension of the class `AbstractLogModelConnection` which is provided in the framework (see Figure B.2). The class `AbstractLogModelConnection` stores the mapping between nodes in a graph and activities in an event log. Therefore, it is suitable to store the mapping between SPD nodes and activities in an event log. To enable user maps activities in the event log to nodes in the SPD, a GUI class `SPDEditorPanel` and an SPD visualizer class `SPDVisualization` are created. The class `SPDVisualization` has the `visualize()` method which accepts

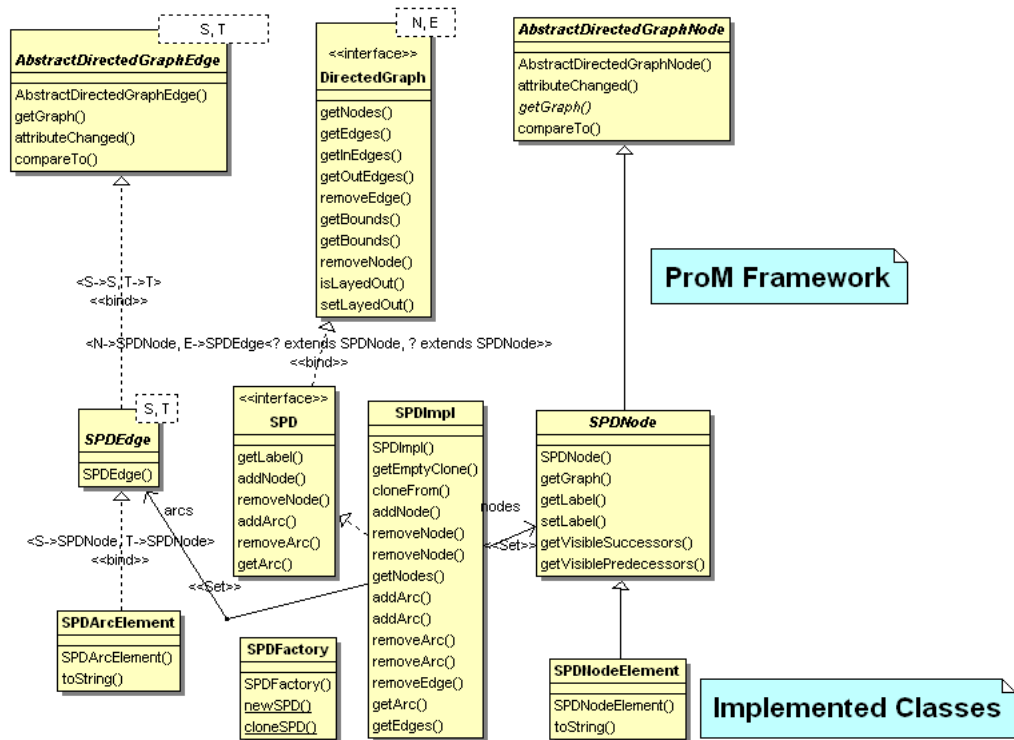


Figure B.1: SPD class design

both an SPD and an event log as its input parameters (or only an SPD if there is a connection object which links the SPD to the event log). If the SPD is not linked to the event log, an object of class `SPDEditorPanel` provides a mapping panel so that a user can map the nodes of the SPD to the activities in the event log. After all SPD nodes are mapped, the object of class `SPDEditorPanel` creates an object of class `LogSPDConnection` to store the mapping between the SPD nodes and the event log in the ProM's *Object Pool*. A screenshot of a visualized object of class `SPDEditorPanel` is shown in Figure B.3.

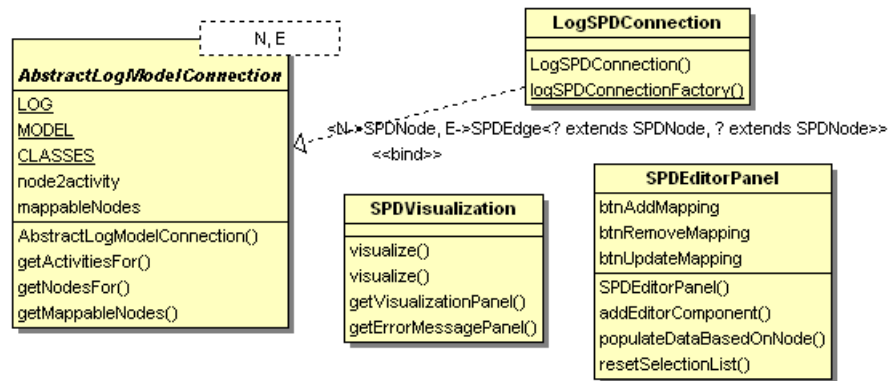


Figure B.2: Classes to map nodes in an SPD to activities in an event log

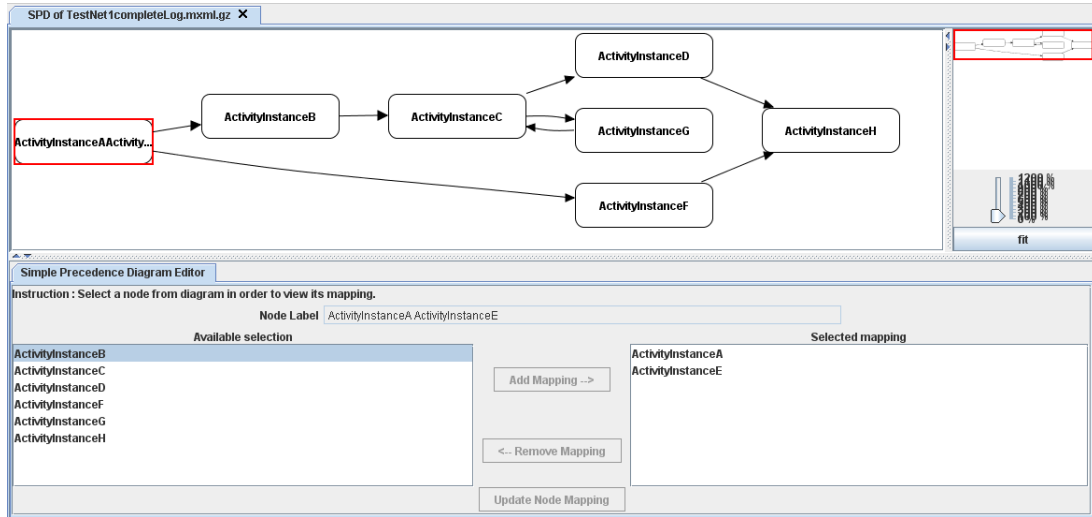


Figure B.3: Screenshot of SPDEditorPanel

B.2 Performance Measurement

To replay a log in an SPD, we need an event log object and an SPD object whose nodes are already mapped to activities in the event log object. As the mapping between activities and nodes in an SPD is stored in a connection object, the connection object is also required to replay the log. These three inputs are used to generate an FPD and an AAPD with all related KPIs. Before the implementation of the log replay is described, first we provide the implementation of both FPD and AAPD as the models to project performance information. The implementation of both models is explained in Section B.2.1. Then, the implementation of log replay classes is explained in Section B.2.2.

B.2.1 Models to Project Performance Information

The design of the classes to represent an FPD is shown in Figure B.4. For simplicity, methods of all classes in the figure are not shown. All methods in the classes are basically getter and setter methods for their attributes. Similar to SPDs, FPDs are basically directed graphs consist of nodes and arcs. Therefore, FPDs can be implemented just like SPDs, i.e. using the same set of superclasses which are already provided by the ProM framework as implementation of SPDs.

An FPD is represented as an interface class `FPD` which extends the class `DirectedGraph`. The real implementation of the FPD lies in the class `FPDImpl`. Node and edge class for the FPD are extended from class `FPDNode` and class `FPDEdge`, respectively. A factory class is also constructed with the name `FPDFactory`. Notice that unlike the class `SPDNode`, class `FPDNode` has many attributes to store performance information. The class `FPDEdge` also has several attributes to store performance information.

The design of classes to implement AAPD is presented in Figure B.5. AAPD can be seen as a directed graph with invisible arcs. An element in AAPD is similar to a node in a directed graph. Therefore, as can be seen in the figure, the implementation of AAPD is similar to the implementation of both FPD and SPD. The main

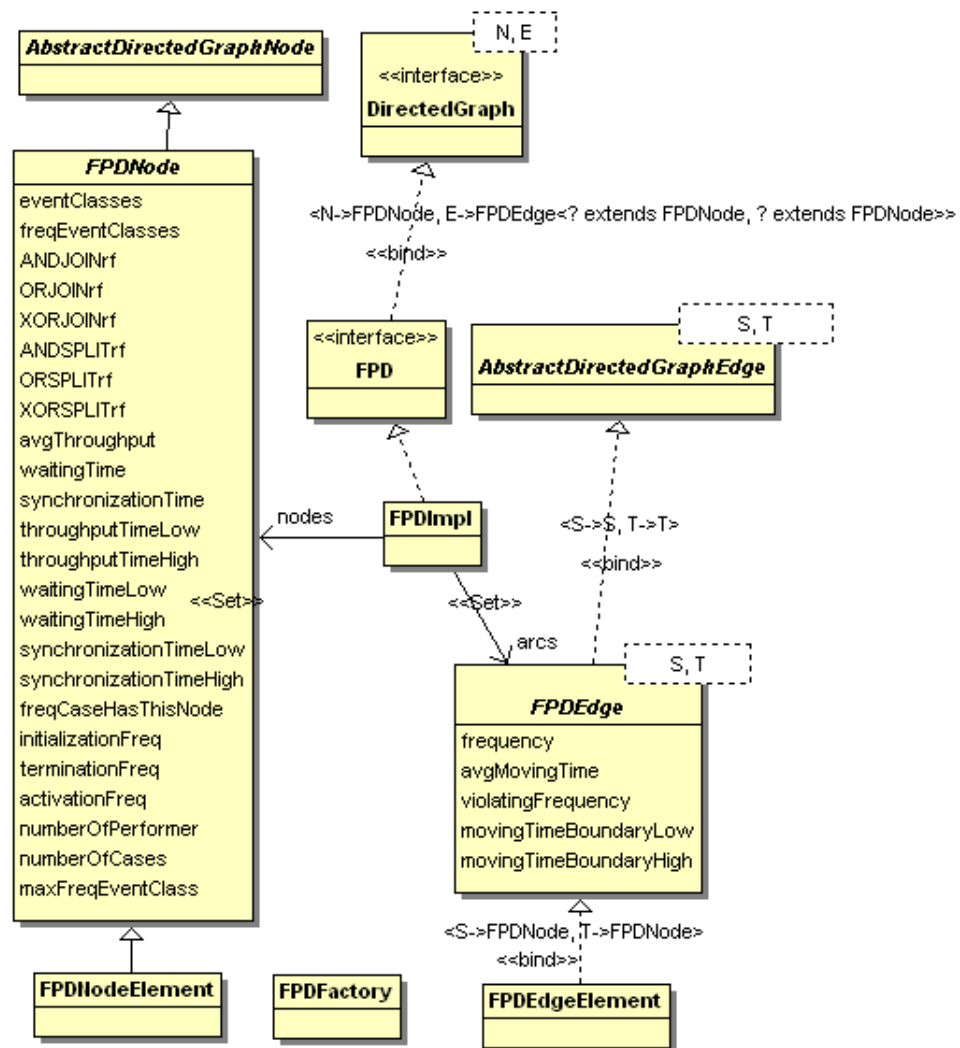


Figure B.4: FPD class design

difference between AAPD and both FPD and SPD is that AAPD elements are set to have a static position.

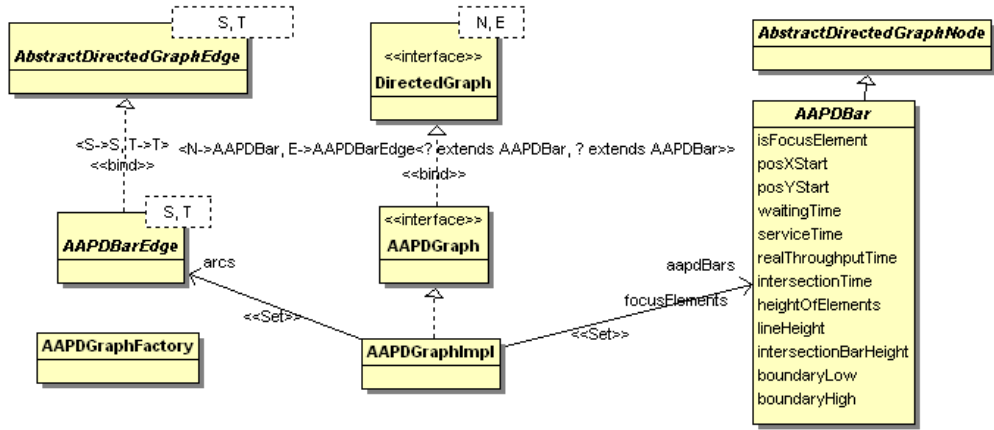


Figure B.5: AAPD class design

B.2.2 Log Replay Plug-in

In order to replay an event log on an SPD, several classes are implemented as shown in Figure B.6.

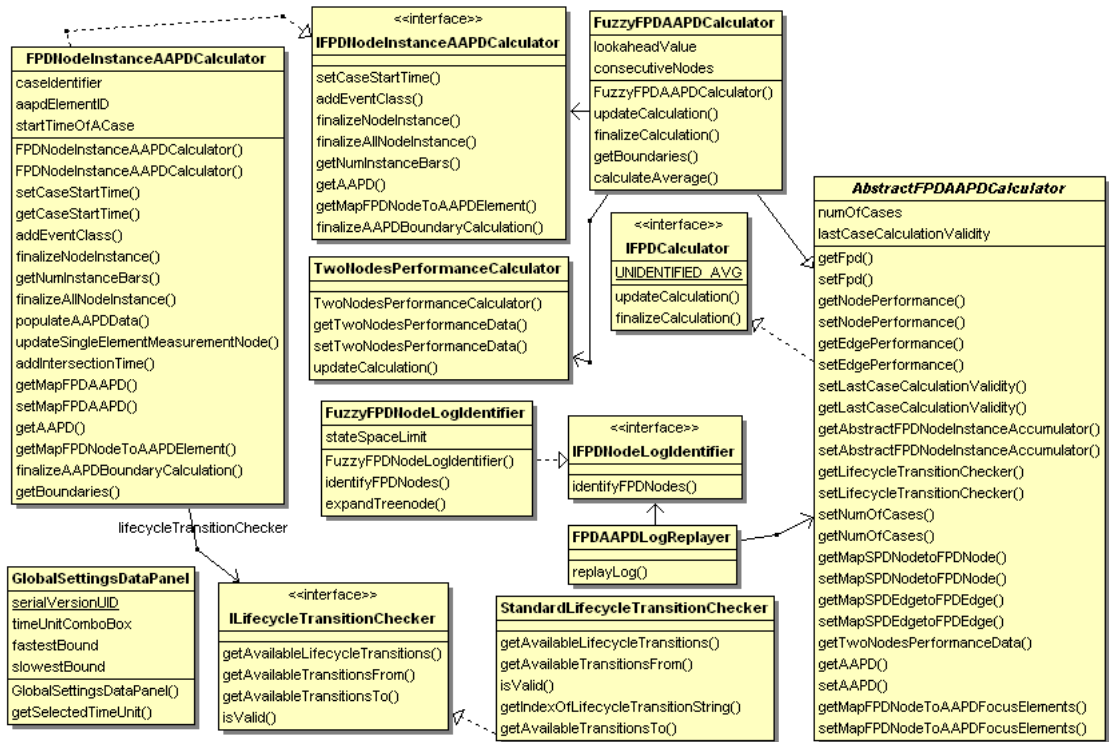


Figure B.6: Design of classes to perform log replay

The main class that responsible to perform the log replay is the class `FPDAAPDLogReplayer`. To perform a log replay, apart of an event log and an SPD, an object of the class `LogSPDConnection` which links the event log and the SPD is also required. Log replay is performed mainly with the help from two other classes: the interface class `IFPDNodeLogIdentifier` and the class `AbstractFPDAAPDCalculator`.

The former is an interface for classes which identify which node in an SPD is referred to by an event in an event log. Our approach to identify nodes is implemented in the class `FuzzyFPDNodeLogIdentifier` that implements the interface class `IFPDNodeLogIdentifier`.

The class `AbstractFPDAAPDCalculator` is a superclass of all classes which are responsible to calculate all performance information based on sequences of SPD nodes. In the class, several basic methods to calculate performance information are provided. The class `AbstractFPDAAPDCalculator` is an abstract class that implements class `IFPDCalculator`, an interface which describes several methods to calculate performance information in an FPD. The class `FuzzyFPDAAPDCalculator` is implemented as an extension of the abstract class and is responsible to calculate performance information which is projected onto either FPD or AAPD.

To add flexibility to performed log replay, the `replayLog` method of the class `FPDAAPDLogReplayer` may accept several parameters: object of class that implements interface `IFPDNodeLogIdentifier`, object of class that implements interface `ILifecycleTransitionChecker`, and object of subclasses of the abstract class `AbstractFPDAAPDCalculator`. Both parameters `IFPDNodeLogIdentifier` and `AbstractFPDAAPDCalculator` enable node identification and performance calculation to be performed in various way by simply passing suitable subclasses. The `ILifecycleTransitionChecker` is an interface for classes to validate whether transitions from an activity's state to another state are valid based on a transactional model in use. In this thesis, the class which checks whether activity state transition is valid is the class `StandardLifecycleTransitionChecker`.

The class `FuzzyFPDAAPDCalculator` is responsible to calculate all performance information from a sequence of SPD nodes. Its `updateCalculation` method accept the sequence as input parameter and calculates all of the performance information. After all traces are calculated, the `finalizeCalculation` method is executed to calculate the values of performance metrics which can only be calculated after all traces are analyzed (e.g. average throughput time of a case). The class `FuzzyFPDAAPDCalculator` uses the interface `IFPDNodeInstanceAAPDCalculator` to calculate both FPD-related and AAPD-related KPIs. This interface is implemented by the class `FPDNodeInstanceAAPDCalculator`. Again, the purpose of having an interface (interface `IFPDNodeInstanceAAPDCalculator`) rather than a class is to make other possible approaches to be easily implemented in the future. Finally, the class `TwoNodesPerformanceCalculator` is implemented to calculate performance analysis related to pair of nodes in the output FPD.

Log replay produces several objects which are then stored in the *Object Pool*. The objects are instances of these classes (see Figure B.7):

- `FPD`, the class represents an FPD and performance information projected onto it.
- `AAPD`, the class represents an AAPD and performance information projected onto it.
- `CaseKPIData`, the class stores all case-related KPIs.
- `FPDElementPerformanceMeasurementData`, the class stores all FPD-node-related KPIs and FPD-edge-related KPIs.

- **TwoFPDNodesPerformanceData**, the class stores all performance metrics which are specific for pair of FPD nodes.
- **GlobalSettingsData**, the class stores global configuration to visualize performance information (e.g. time unit).

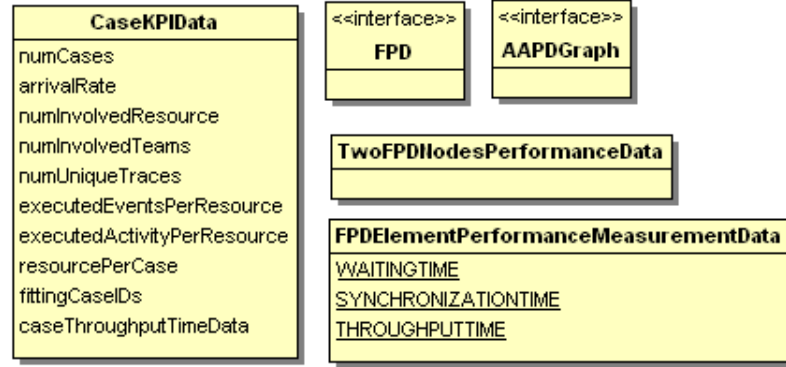


Figure B.7: Classes to store the result of log replay

In order to link the objects which are produced from a single log replay, two connection classes are implemented. The first connection class, **FPDAAPDConnection**, links an FPD and an AAPD. The second connection class, **FPDLogReplayConnection**, links all objects which result from the log replay.

B.2.3 Performance Information Visualization

After all performance information is calculated, the next step is to visualize the information in a user-friendly manner. The first performance information to be visualized is FPD-related information. In order to visualize all performance metrics related to an FPD in a compact way, we refer to the concept of performance dashboard. A common performance dashboard consists of several panels. One of the panels provides an overview about performance of all activities, and the other panels provide detailed information about performance. In our case, the main panel of the dashboard shows the FPD, and the other panels provide KPI values that are related to it. Screenshot of the implemented dashboard is given in Figure B.8.

The design of classes which forms the visualization in Figure B.8 is shown in Figure B.9. The class **FPDVisualization** is the main *Visualizer* class which accepts an FPD object. The **visualize()** method of the class accepts the FPD object and returns an object of class **FPDInformationPanel** which represents the GUI. The main panel of the GUI is implemented using class **JGraphVisualizationPanel** which is already provided by the ProM framework. This class visualizes graphs which are extended from class **DirectedGraph**, including their zooming panel and navigation panels.

For additional panels, several classes are implemented, each represents a group of similar information. Case-level KPIs are visualized in a panel which is provided by the class **CaseKPIInformationPanel**. The class **ElementPerformancePanel** is a superclass of both class **EdgePerformancePanel** and class **NodePerformancePanel**.

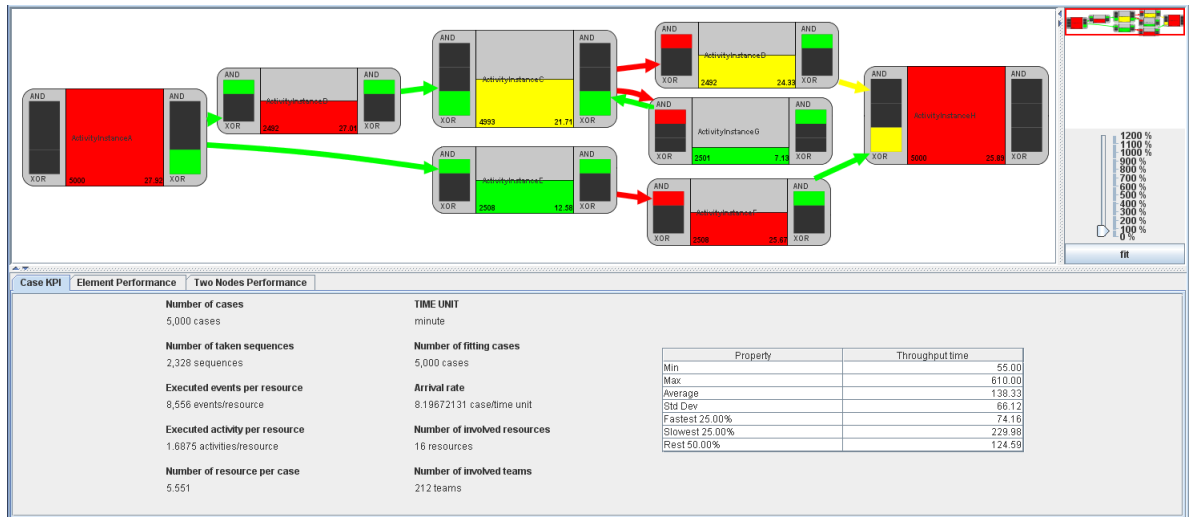


Figure B.8: FPD visualization

and each visualizes FPD-node-related performance information and FPD-edge-related performance information, respectively. Finally, the class `TwoNodesInformationPanel` visualizes two-nodes-analysis performance information.

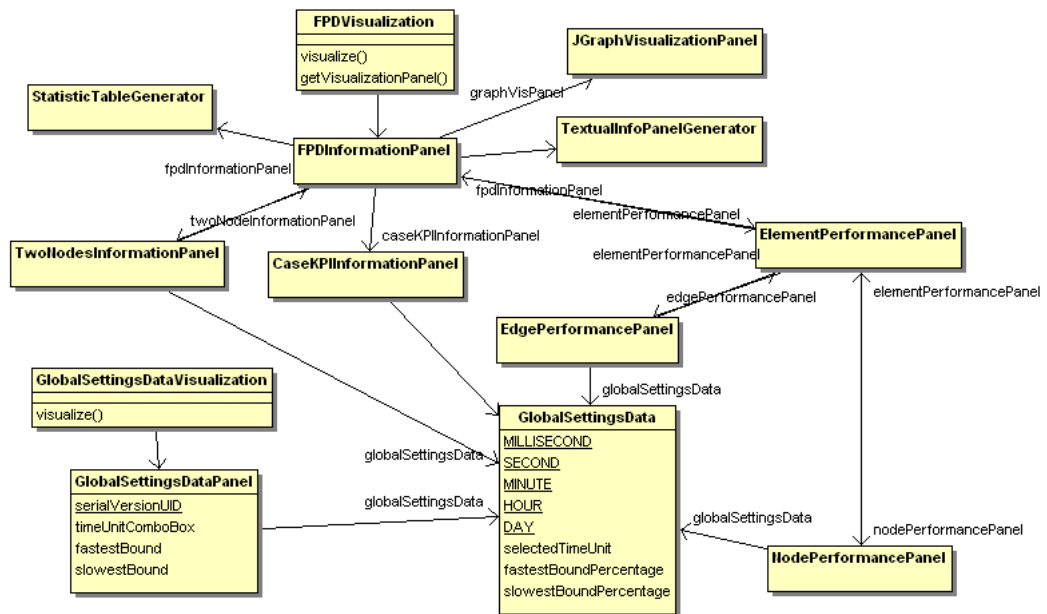


Figure B.9: FPD visualization class design

Other than the previously mentioned classes to visualize performance information, some other classes are needed to help them performing their function. The class `TextualInfoPanelGenerator` class helps the generation of textual information (see Figure B.10), while the class `StatisticTableGenerator` helps the generation of table information (see Figure B.11). The class `GlobalSettingsData` stores the information about time unit which is used to show performance information and precentage bounds which are going to be visualized for several per-

formance information. Hence, it is used by several classes. The object of class `GlobalSettingsDataVisualization` visualizes objects of class `GlobalSettingsData` with the help of the class `GlobalSettingsDataPanel` so that user can modify the values in the objects.

Number of cases	TIME UNIT
5,000 cases	minute
Number of taken sequences	Number of fitting cases
2,090 sequences	5,000 cases
Executed events per resource	Arrival rate
8,556 events/resource	8.19672131 case/time unit
Executed activity per resource	Number of involved resources
1.6875 activities/resource	16 resources
Number of resource per case	Number of involved teams
5.551	212 teams

Figure B.10: Example of textual information

Property	Throughput time
Min	55.00
Max	610.00
Average	138.33
Std Dev	66.12
Fastest 25.00%	74.16
Slowest 25.00%	229.98
Rest 50.00%	124.59

Figure B.11: Example of table information

Visualization of AAPD follows the way FPD is visualized. AAPD is visualized in a dashboard consisting of three parts: the top, the middle, and the bottom part. The AAPD graph is visualized on the top part of the dashboard, while the bottom part provides the detail of performance information related to the AAPD (see Figure B.12). The middle part of the dashboard provides a combo box to change the focus element of the displayed AAPD and several slide controls to adjust the appearance of the AAPD graph, e.g. X-coordinate scaling (distance between elements), element width scaling, and element height scaling.

Design of the classes to implement the AAPD visualization in Figure B.12 is shown in Figure B.13. The main class that is responsible to visualize an AAPD object is the class `AAPDVisualization`. The `visualize()` method of the class accepts an object of class `AAPD` and returns an object of class `AAPDInformationPanel` which represents the GUI. Similar to the main panel of FPD visualization, the GUI is implemented using the class `JGraphVisualizationPanel`.

When any of the AAPD scales are adjusted using any of the sliders in the middle panel, a new AAPD graph is generated. To generate a new AAPD, the class `AAPDGraphGenerator` is used. The class `AAPDStatisticPanel` generates tables which are used to show performance values, such as throughput time table, queuing time table, and service time table. This class also utilizes the class `AAPDStatisticTableGenerator` to perform its task generating the tables. When a table is generated, an object of class `GlobalSettingsData` is used to adjust the time unit for all performance values which are shown.

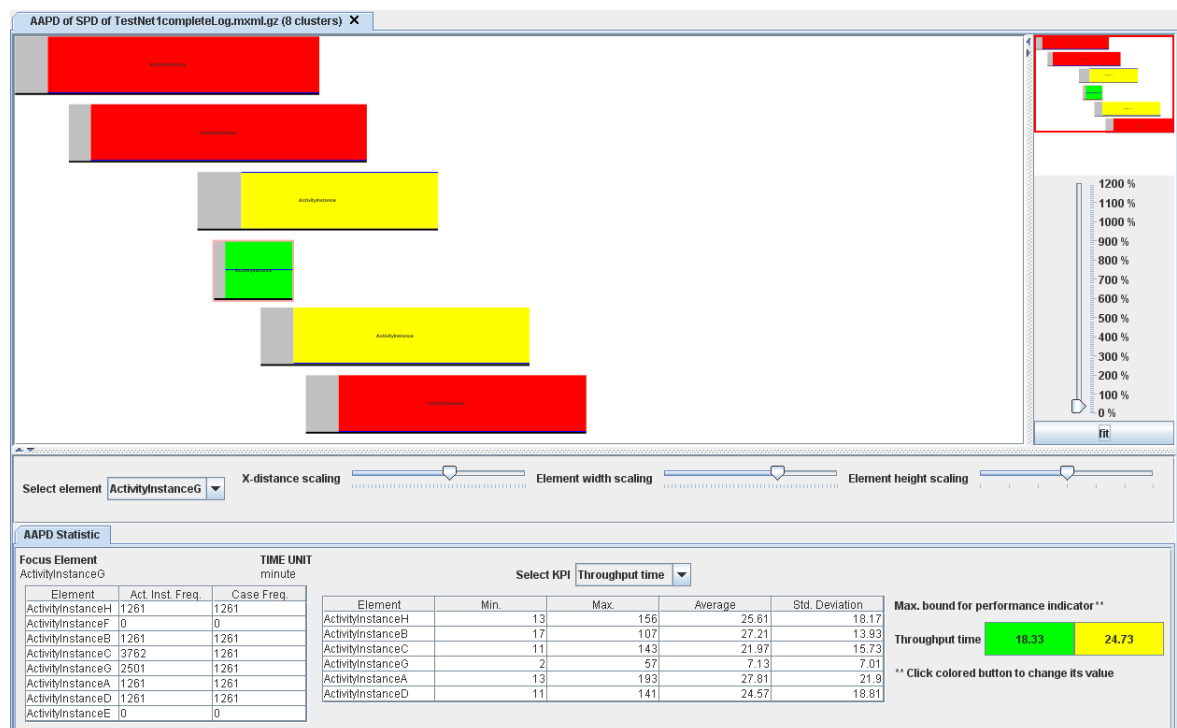


Figure B.12: AAPD visualization

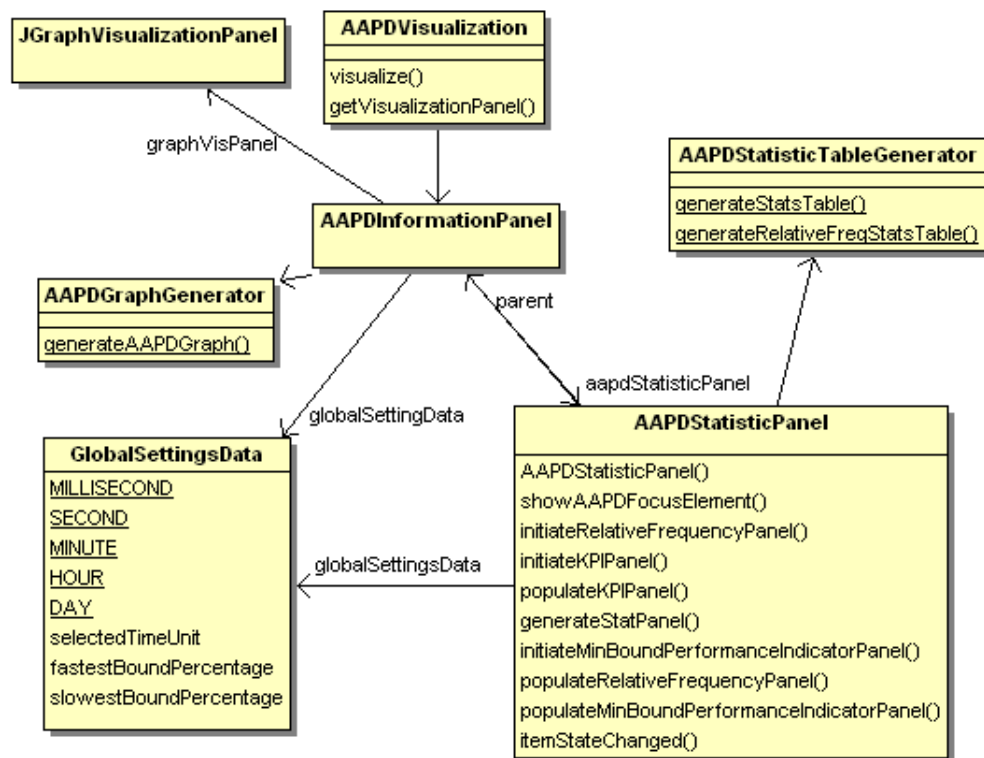


Figure B.13: AAPD visualization class design

Appendix C

User Manual

This appendix provides a brief user manual for all ProM plugins which are implemented in this thesis, including the SPD Miner plug-in that was not a part of the implementation work reported in this thesis.

C.1 SPD Miner Plug-in

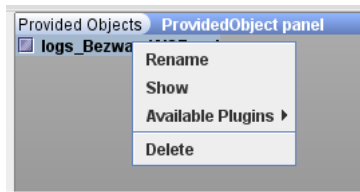
C.1.1 Introduction

The SPD Miner plug-in accepts an event log object and constructs an SPD object based on the event log using the Fuzzy k-Medoid clustering approach. This plug-in accepts a user-defined value to determine how many clusters are generated from the event log object. Beside the SPD object, this plug-in also creates a connection object that links the object to the event log object. The plug-in produces random outputs from event logs, i.e. two SPD objects which are both constructed from the same log object using this plug-in may not be similar in terms of nodes and arcs.

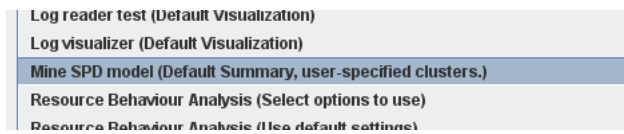
C.1.2 Using SPD Miner Plug-in

To use the SPD miner plug-in, we need an event log object in the *Provided Objects* panel of ProM. Suppose that such object exists in the panel, right-click on the object and select “**Available Plugins**” > “**Mine SPD Model (Default Summary, user-specified clusters)**” (see Figure C.1). A user dialog which ask for the number of clusters to be generated will be shown (see Figure C.2). Choose one of the values provided by the displayed combo box, and then click “**OK**”. The selected value determines how many SPD nodes will exist in the constructed SPD object. Note that the maximum value in the combo box is always equal to the number of activities in the event log, and the minimum value in the combo box is always equal to 1.

After all previous steps are performed, the plug-in starts to calculate the SPD of the log. During the process, a progress bar is shown in the *Plugin* panel as shown in Figure C.3. The result of the plug-in is an SPD object which is stored in *Provided Objects* panel. In addition, the plug-in also creates a connection object that links the SPD object to the event log object.



(a) Selection of event log



(b) Selection of plug-in

Figure C.1: Using SPD Miner

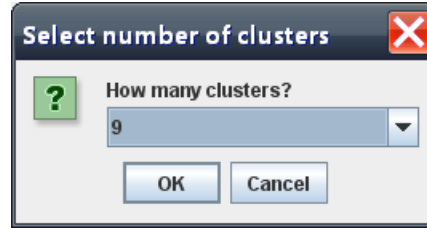


Figure C.2: Dialog which ask for the number of SPD clusters



Figure C.3: Progress bar that indicates that the SPD Miner plug-in is processing

C.2 SPD Visualization Plug-in

C.2.1 Introduction

This plug-in visualizes SPD objects and their node mapping to activities in event log objects. An example of a visualized SPD object is shown in Figure C.4. As seen in the figure, the SPD is displayed on the top panel of the GUI. On the right side of the SPD, there is a zooming panel to adjust the size of SPD being shown on the screen. In addition, small panel on top of the zooming panel can be used to navigate through the displayed SPD quickly. The user can drag the red box in the small panel to the desired part of the SPD which wants to be visualized. Clicking on a displayed SPD node will make the node's label and all activities which the node refers to visualized on the bottom panel.

SPD Visualization plug-in accepts an SPD object to be visualized and an event log object. This plug-in does not only visualize the SPD object, but also provides an interface to map nodes in the SPD object to activities in the event log object if such mapping does not exist before. The three buttons in the bottom panel: **“Add Mapping”**, **“Remove Mapping”**, and **“Update Node Mapping”** can be used to create such mapping. All three buttons are only enabled if there is no mapping between the nodes in the selected SPD object and the activities in the selected event log object. After all nodes in the SPD object are mapped to activities in the event log object, these three buttons are disabled.

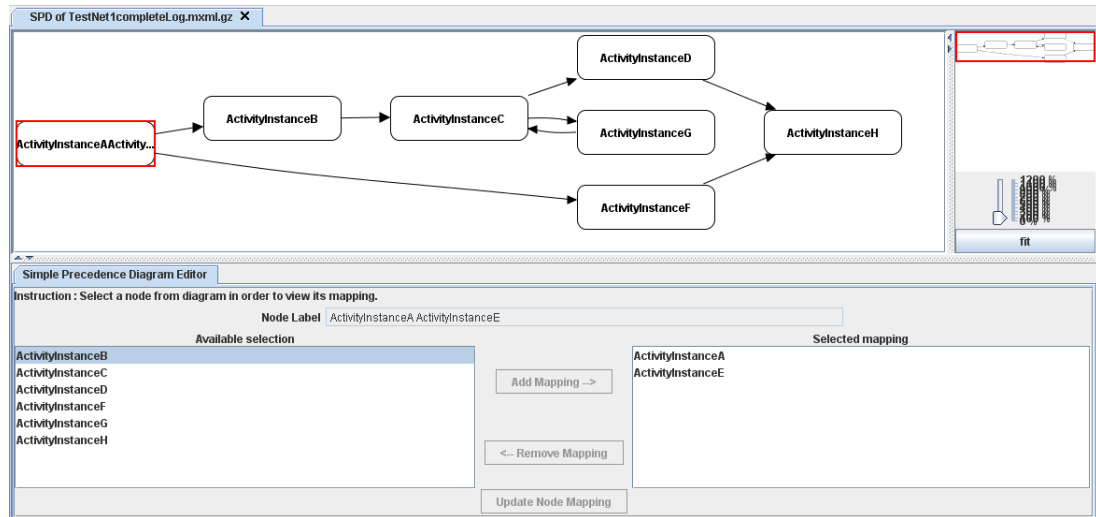
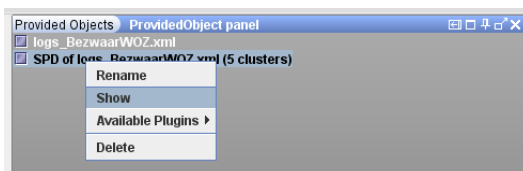


Figure C.4: SPD Visualization

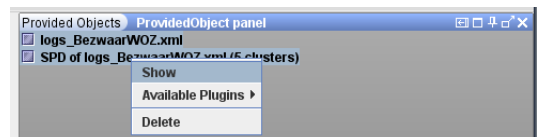
C.2.2 How to Use

C.2.2.1 Visualize SPD

To visualize an SPD object, right-click on the SPD object to be visualized. The SPD object should be located on the *Provided Objects* panel. Then, click “**Show**” (see Figure C.5a). If there is no event log object which is linked to the SPD object, an error message will be displayed as shown in Figure C.6. If there is such event log object, one of the event log object which is linked by a connection object is selected. Then, the SPD object and the mappings between the activities in the selected log object and the nodes in the SPD object are visualized as shown in Figure C.4.



(a) Variant 1 (without event log object as input parameter)



(b) Variant 2 (with event log object as input parameter)

Figure C.5: How to use SPD Visualization plug-in

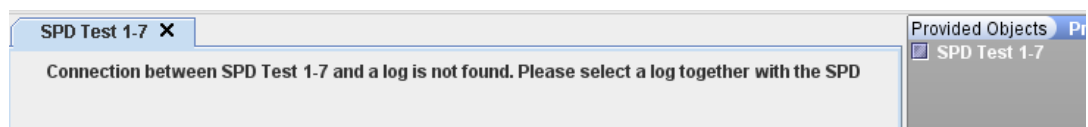


Figure C.6: Error message if there is no event log which is mapped to the selected SPD

An SPD object can be linked to more than one event log objects. To show a specific pair of SPD object and event log object, select both objects from the list of objects in *Provided Objects* panel by simply clicking them. Use button “Ctrl”

or “Shift” to select more than one object in the panel by holding the button while clicking at the desired objects. Then, right-click on any of the selected object and click “**Show**” (as shown in Figure C.5b).

C.2.2.2 Mapping SPD nodes to activities

Mapping between the SPD nodes in an SPD object and the activities in an event log object can only be performed if there is no connection object in ProM’s *Object Pool* which links both objects. To do the mapping, select both the event log object and the SPD object from the *Provided Objects* panel. Right click on one of the selected objects and click “**Show**” (see Figure C.7). Suppose that there is no connection object that links the two objects, the same display as Figure C.4 will be shown with all the three mapping-related buttons set to enabled.

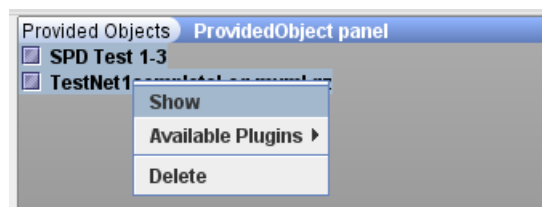


Figure C.7: The first step of mapping SPD nodes to activity

To map an SPD node to one or more activities, click the node on the top panel. The details of the node will be shown in the bottom panel. Then, select the activities to be mapped to the node from the available selection list which is located on the left side of the bottom panel. Click the “**Add Mapping**” button to move the selected activities to the selected mapping list. Then, click the “**Update Node Mapping**” button to save the mapping. In a similar way, we can remove one or more activities from the selected mapping list and place it back to the selected activities list using the “**Remove Mapping**” button. Repeat the mapping steps to map all nodes in the SPD object to activities in the event log object.

Only after all nodes in the SPD object are mapped to activities, a dialog window appears as shown in Figure C.8. Click “**OK**” to close the dialog. After the dialog is closed, all three mapping-related buttons are disabled and a connection object that links the SPD object and the event log object is created.

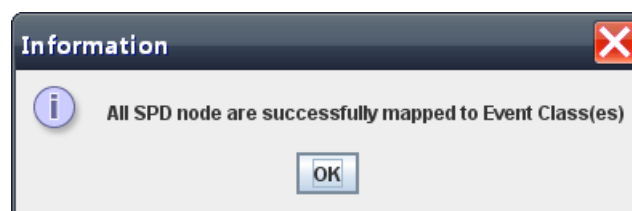


Figure C.8: Popup window after all nodes are mapped

C.3 Event Log Replay Plug-in

C.3.1 Introduction

The log replay plug-in calculates KPI values of a process based on the process' event log and an SPD which describes the process. This plug-in accepts both an event log object and an SPD object. Both objects must be linked, i.e. there is a connection object in the *Object Pool* which links the SPD object and the event log object. This plug-in produces five different objects, consisting of four objects that store the performance values of the process and an object to store visualization configuration values. These objects include an FPD object, an AAPD object, and a Global settings object, each can be visualized by a different plug-in.

C.3.2 How to Use

There are two variants of this plug-in. The first variant requires only an SPD object as its input. This variant searches a connection object which links the SPD object to any log object. If such an object is not found, the plug-in throws an exception message which is displayed in the ProM's *Message Panel*. The second variant of this plug-in requires both an SPD object and an event log object. The same exception message is thrown if there is no connection object which links the SPD object to the event log object.

To use the first variant of this plug-in, select an SPD object from the *Provided Objects* panel. Right click on the object and select **“Available Plugins”** > **“Replay Log in Simple Precedence Diagram (SPD) (From SPD)”** as shown in Figure C.9.

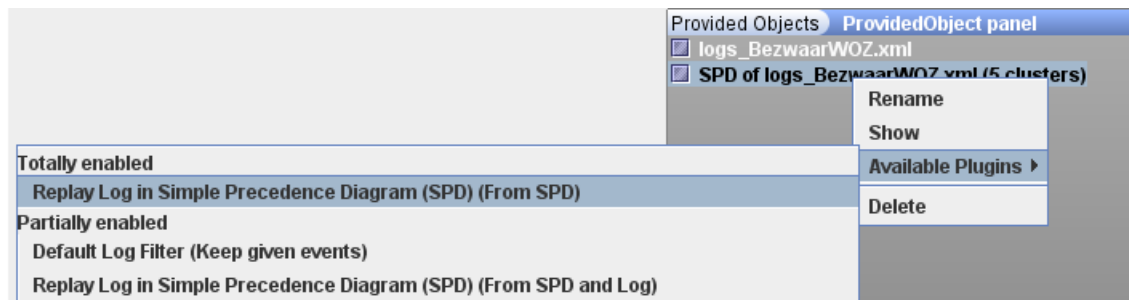


Figure C.9: How to use the Event Log Replay Plug-in

Then, a dialog window will appear as shown in Figure C.10. Select the look-ahead value and then click **“OK”**.

Another dialog window will be shown as in Figure C.11. Insert the value of maximum generated states before random selection is performed during maximum fitting subtrace identification phase of the log replay. Note that the inserted value must be positive integer. After that, click **“OK”** to start replaying the event log object on the SPD object.

Replay process takes some time, depending on the complexity of the event logs and the values of input parameters. Progress of the replay process is shown in the *Plugins* panel as shown in Figure C.12. After the plug-in finishes processing, six new

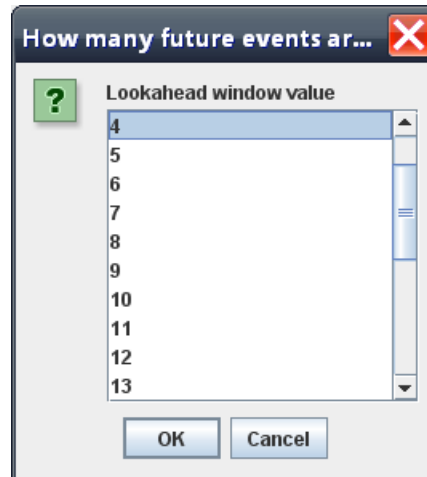


Figure C.10: Dialog to determine look-ahead value

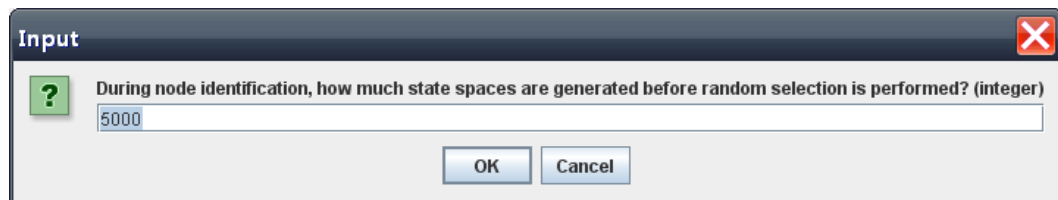


Figure C.11: Dialog to adjust the value of maximum state space in the search of maximum fitting subtraces

objects are added to the *Provided Objects* panel as the outputs of the replay: the *FPD* object, the *Case KPI data* object, the *Elements' performance* object, the *Two nodes performance* object, the *AAPD* object, and the *Global settings* object (see Figure C.13). Beside the six objects, the plug-in also creates a connection object which links all of the six objects.

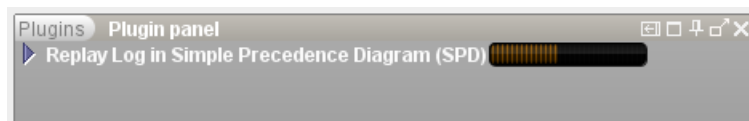


Figure C.12: Event log replay progress bar

To use the second variant of this plug-in, select an event log object and an SPD object from the list of objects in the *Provided Objects* panel. Then, right-click on any of the selected objects and perform the same steps as already explained for the first variant.

C.4 FPD Visualization

C.4.1 Introduction

This plug-in visualizes an FPD object and all related-performance values. The plug-in accepts an FPD object and visualizes it together with FPD-related performance

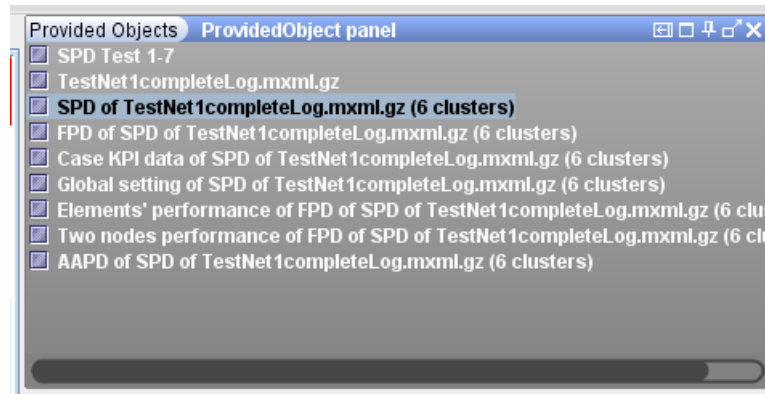


Figure C.13: Output objects of log replay plug-in

information. The information is obtained from other objects which are linked to the object: the *Case KPI data* object, the *Elements' performance* object, the *Two nodes performance* object, and the *Global settings* object.

The FPD object is visualized in a performance dashboard style, as shown in Figure C.14. FPD is shown in the top left panel. The top right panel provides a zoom panel with a slider to help the user adjust the zoom level of the displayed FPD. On top of the zoom panel, there is a small navigation panel that shows the whole FPD and the part of the FPD which is currently shown (it is indicated by the red box). The bottom panel displays detailed performance information which is related to the FPD.

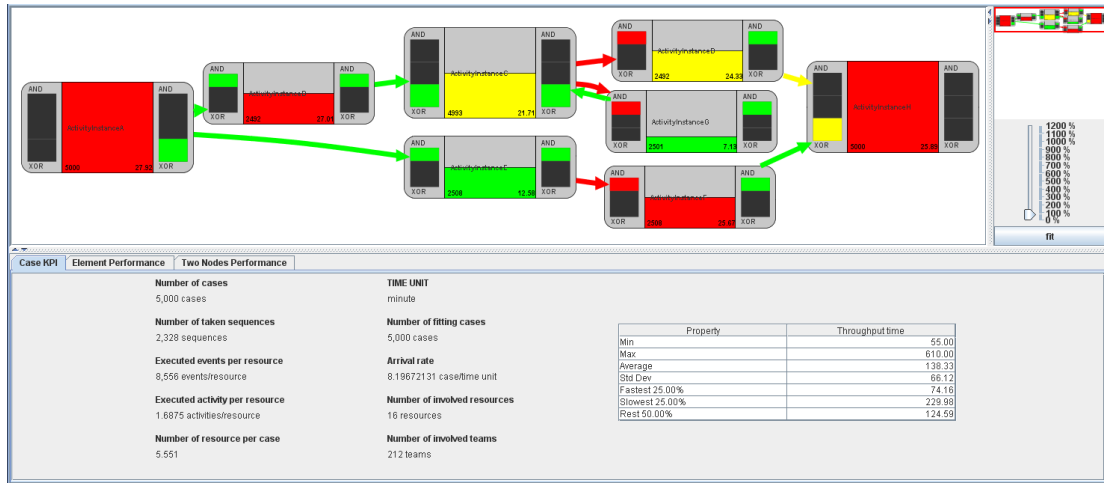


Figure C.14: FPD visualization

There are three types of information which are provided in the bottom panel. The first is the case-level information type that provides various KPIs of a process that can be calculated without a process model. This information is shown in the “**Case KPI**” tab. The second type of information is element-related KPIs that provides various performance information related to nodes and edges of the displayed FPD. This information is shown in the “**Element Performance**” tab. Finally, the third type provides performance information which relate two nodes in the displayed FPD. This type of information is provided in the “**Two Nodes Performance**” tab.

C.4.2 Performance Information

Each tab in the bottom panel of the FPD visualization panel provides different types of information. Details of information provided by each tab is given as follows:

Case-level KPIs Panel

Case-level KPIs Panel is shown in Figure C.15. In general, this panel provides information of performance of cases in an event log and information about the log. The information include:

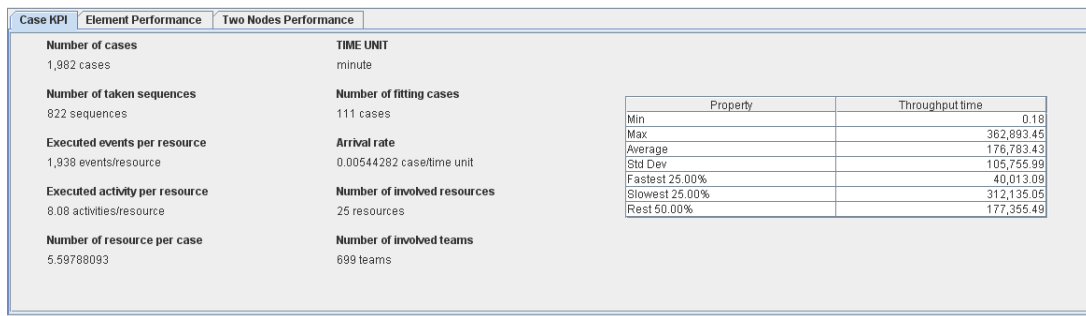


Figure C.15: The Case-level KPIs panel

- **Case throughput time:** The time spent to handle a case in the event log, including the minimum time, the maximum time, the average time, and standard deviation value of the average time. In addition, the average throughput time of the fastest $m\%$ cases, the slowest $n\%$ cases, and the rest $(100 - m - n)\%$ cases are also provided. The values of both m and n can be modified by modifying the *Global setting* object.
- **Number of cases:** The total number of cases in the event log.
- **Number of taken sequences:** The total number of unique traces of events from all cases in the event log.
- **Executed events per resource:** The average number of events that is correlated to a resource in the event log.
- **Executed activities per resource:** The average number of unique activities which is performed by a resource.
- **Number of resources per case:** The average number of involved resources in a case.
- **Number of fitting cases:** The total number of traces of events in all cases which is also maximum fitting subtraces.
- **Arrival rate:** The average number of cases that arrive per time unit.
- **Number of involved resources:** The total number of resources in the log event.
- **Number of involved teams:** The total number of unique set of resources which are allocated to a trace in the event log.

Element Performance

Element performance panel provides information related to the nodes and the edges of the displayed FPD. The information can be categorized to node-related KPIs and edge-related KPIs. Node-related KPIs are displayed as shown in Figure C.16, and edge-related KPIs are displayed as shown in Figure C.17.

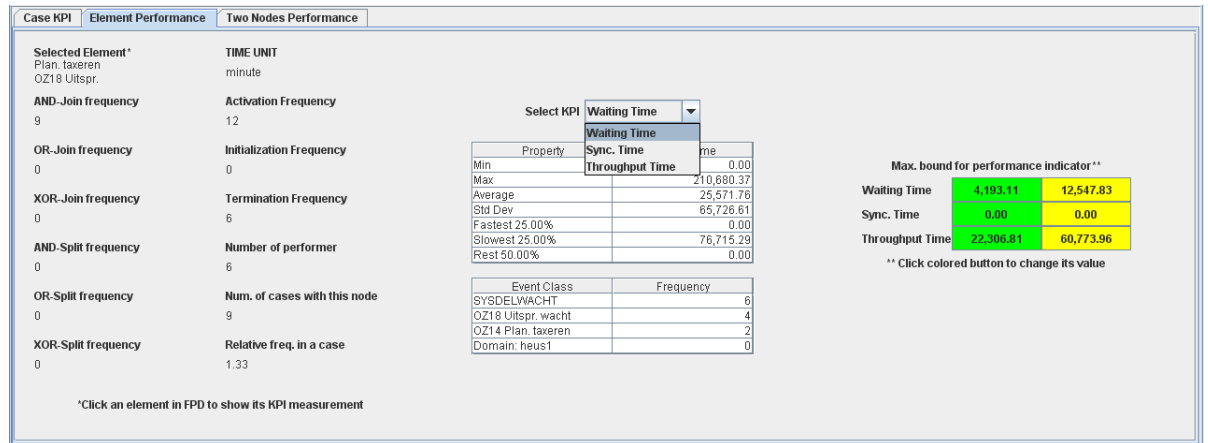


Figure C.16: Example display of Node-related KPIs panel

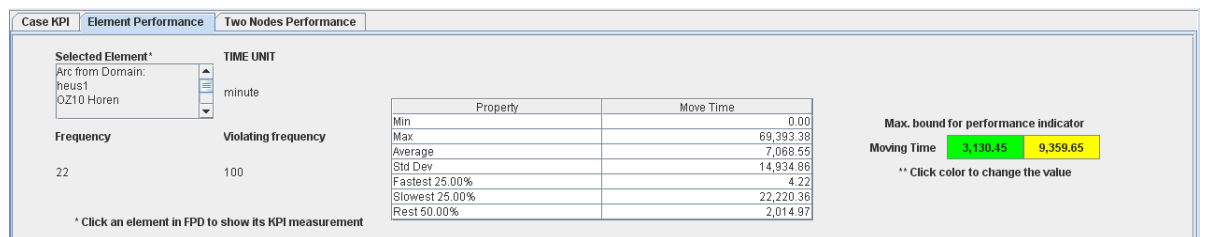


Figure C.17: Example display of Edge-related KPIs panel

Given a node in an FPD as a node under inspection, the node-related KPIs of the node are given as follows:

- **Node activation frequency:** The total number of events in an event log which is mapped to the node.
- **Node initialization frequency:** The total number of cases which is started with an event which is mapped to the node.
- **Node termination frequency:** The total number of cases which is ended with an event which is mapped to the node.
- **Number of performers:** The total number of unique resources which is correlated with any event which is mapped to the node.
- **Number of cases with this node:** The total number of cases in which there is an event which is mapped to the node.
- **Relative frequency in a case:** The node activation frequency per case.
- **AND-join frequency:** The total number of times where the node has AND-join semantics.

- **AND-split-frequency:** The total number of times where the node has AND-split semantics.
- **OR-join frequency:** The total number of times where the node has OR-join semantics.
- **OR-split frequency:** The total number of times where the node has OR-split semantics.
- **XOR-join frequency:** The total number of times where the node has XOR-join semantics.
- **XOR-split frequency:** The total number of times where the node has XOR-split semantics.
- **Node throughput time:** the time spent to perform instances of the node, including the minimum time, the maximum time, the average time, and standard deviation value of the average time. In addition, the average node throughput time of the fastest $m\%$ cases, the slowest $n\%$ cases, and the rest $(100 - m - n)\%$ of the cases are also provided. Value of both m and n can be modified by modifying the *Global setting* object.
- **Node waiting time:** Suppose that there is a set of node instances E_{pred} which need to be executed before an event which corresponds to the node can be executed. Waiting time is defined as the time between the latest moment when all events in E_{pred} is finished and the moment the first event which is mapped to the node occurs. The provided time include the minimum time, the maximum time, the average time, and standard deviation value of the average time. In addition, the average node waiting time of the fastest $m\%$ cases, the slowest $n\%$ cases, and the rest $(100 - m - n)\%$ cases are also provided. Value of both m and n can be modified by modifying the *Global setting* object.
- **Node synchronization time:** Suppose that all node instances in a set of node instances E_{pred} need to be executed before an event which refers to a node instance is executed, where an instance of the node under inspection is also in E_{pred} . Synchronization time is the time between a moment the latest event in E_{pred} is finished and the moment the node instance of the node under inspection is finished. The provided time include the minimum time, the maximum time, the average time, and standard deviation value of the average time. In addition, the average node synchronization time of the fastest $m\%$ cases, the slowest $n\%$ cases, and the rest $(100 - m - n)\%$ cases are also provided. Value of both m and n can be modified by modifying the *Global setting* object.

Given an edge which connects a source node n_1 to a destination node n_2 , the edge-related KPIs which are provided are given as follows:

- **Frequency:** The total number of times a process control was passed from the instance of node n_1 to the instance of node n_2 .
- **Edge move time:** The time spend to pass process control from the source node n_1 to the destination node n_2 . The provided time include the minimum time, the maximum time, the average time, and standard deviation value of the average time. In addition, the average of the fastest $m\%$ cases, the slowest $n\%$ cases, and the rest $(100 - m - n)\%$ cases are also provided. Value of both m and n can be modified by modifying the *Global setting* object.

- **Violating frequency:** The total number of times when a new instance of the source node n_1 occurs without gaining any control from other node instance while an instance of the destination node n_2 has a process control.

Two Nodes Performance

Two nodes performance panel provides KPI values which focus on only a pair of FPD nodes. The panel is shown in Figure C.18. Several KPIs which are provided in this panel for a pair of nodes n_1 and n_2 are given as follows:

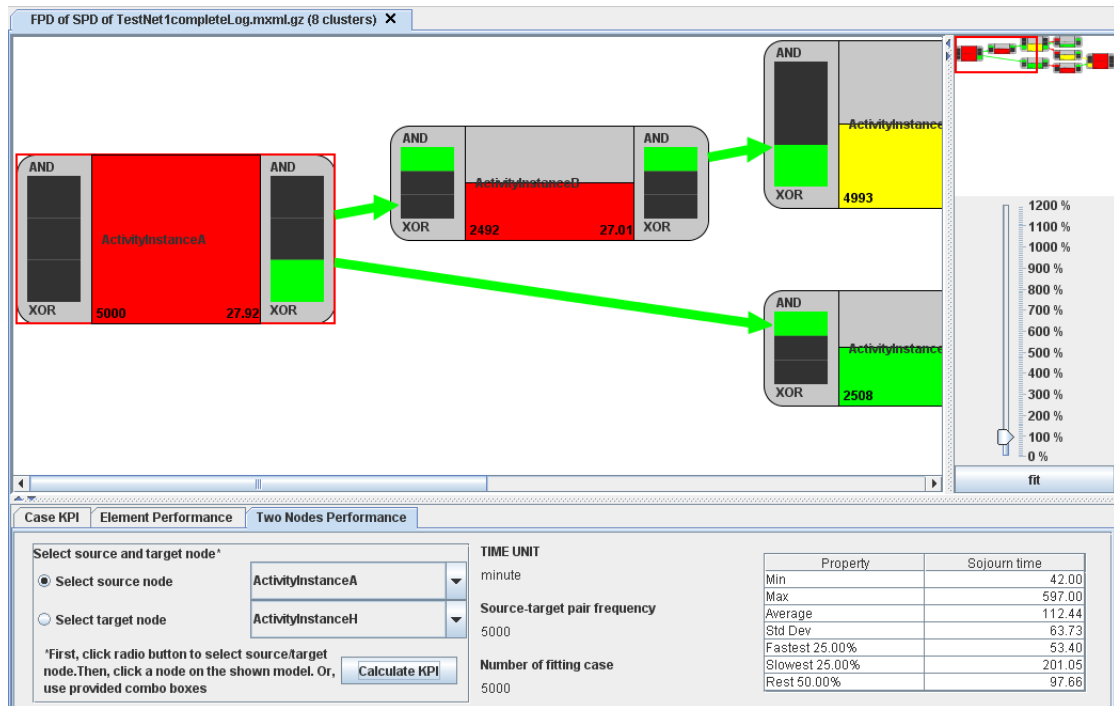


Figure C.18: Example display of the Two Nodes Performance panel

- **Source-target pair frequency:** The total number of traces of events in an event log where two events, each refers to node n_1 and n_2 , respectively, occurred.
- **Number of fitting cases:** The total number of traces in all cases where the trace is also a maximum fitting subtrace.
- **Sojourn time:** The time spent between the first occurrence of events which correspond to node n_1 and the first occurrence of events which correspond to node n_2 in a case where the two events exist. The provided time include the minimum time, the maximum time, the average time, and standard deviation value of the average time. In addition, the average of the fastest $m\%$ cases, the slowest $n\%$ cases, and the rest $(100 - m - n)\%$ cases are also provided. Value of both m and n can be modified by modifying the *Global setting* object.

C.4.3 How to Use

To use the FPD visualization plug-in, right click on an FPD object which is located in the *Provided Objects* panel. Then, select **“Show”** (see Figure C.19). A similar panel as shown in Figure C.14 will be shown. To see case-related KPIs, click on the **“Case KPI”** tab. Similarly, to see the Element-related KPIs and the Two nodes performance KPIs, click on the **“Element Performance”** tab and the **“Two Nodes Performance”** tab, respectively.

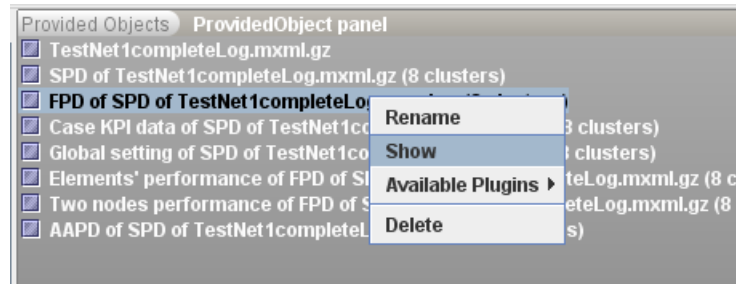
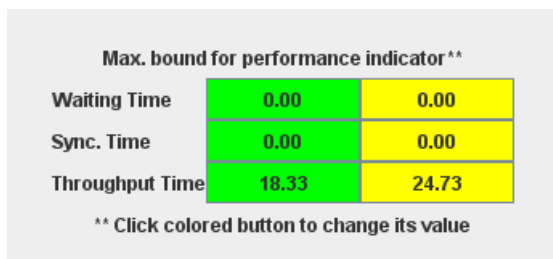


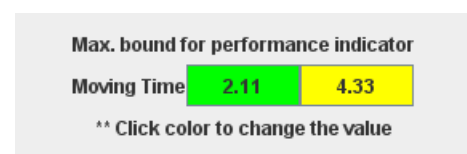
Figure C.19: How to use the FPD visualization plug-in

To show the KPI values which are related to a node or an edge, first, click on the **“Element Performance”** tab. Then, click on either the node or the edge of interest in the displayed FPD. The KPI values for the selected element will be shown at the bottom panel. We can also adjust the boundary values which are used to determine the performance color of nodes or edges using the yellow and green boxes which are located on the right side of the bottom panel. Figure C.20a shows the boxes that can be used to adjust boundary values which are related to FPD nodes, and Figure C.20b shows the boxes that can be used to adjust boundary values which are related to FPD edges. To change the boundary value, click on the box. A dialog window will be shown. Insert the new boundary value in the dialog, and then click **“OK”**. As an example, Figure C.21 shows the dialog which is shown when the green box to adjust the Throughput Time performance boundary of a node is clicked.

To show KPI values which are related to two nodes, click on the **“Two Nodes Performance”** tab. Then, select the source node and the target node using the provided combo box at the bottom panel. After that, click the **“Calculate KPI”** button to show the KPI values. As another alternative, the nodes can also be selected using the visualized FPD nodes. To select a source node, rather than using



(a) Boxes to adjust performance color of a node



(b) Boxes to adjust performance color of an edge

Figure C.20: Display of boxes to adjust the boundary of performance color

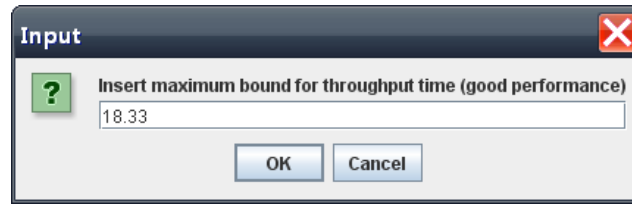


Figure C.21: Dialog window to modify the value of node throughput time performance boundary

the provided combo box, click an FPD node which is displayed on the top panel after clicking the radio button beside the **“Select source node”** label (see Figure C.22). The selection of a target node can also be performed in the same way as the selection of a source node, except that the radio button that must be clicked is the one beside the **“Select target node”** label.

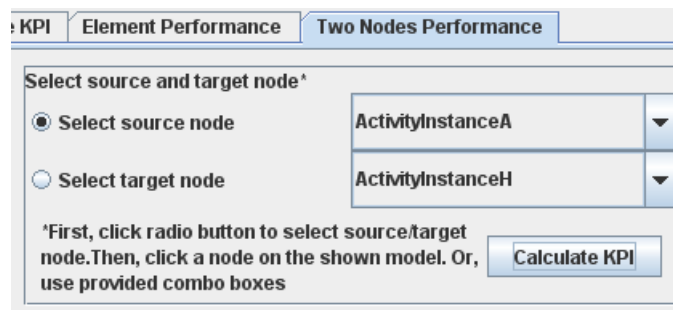


Figure C.22: Example of a selected radio button beside the **“Select source node”** label

C.5 AAPD Visualization

C.5.1 Introduction

This plug-in visualizes an AAPD object and all related-performance values. The plug-in accepts an AAPD object and visualizes it together with AAPD-related performance information. Similar to the FPD visualization plug-in, this plug-in also utilize the *Global settings* object.

The AAPD object is visualized as shown in Figure C.23. An AAPD is shown in the top left panel. The top right panel provides a zoom panel with a slider to help the user adjust the zoom level of the displayed AAPD. On top of the zoom panel, there is a small navigation panel that shows the whole AAPD and the part of the AAPD which is currently shown (it is indicated by the red box). The middle panel provides a combo box to select the focus element of the AAPD and several sliders, each to adjust the scaling of the displayed AAPD. The scaling include horizontal distance (X-distance), element width, and element height. The bottom panel displays detailed performance information which is related to the displayed AAPD.

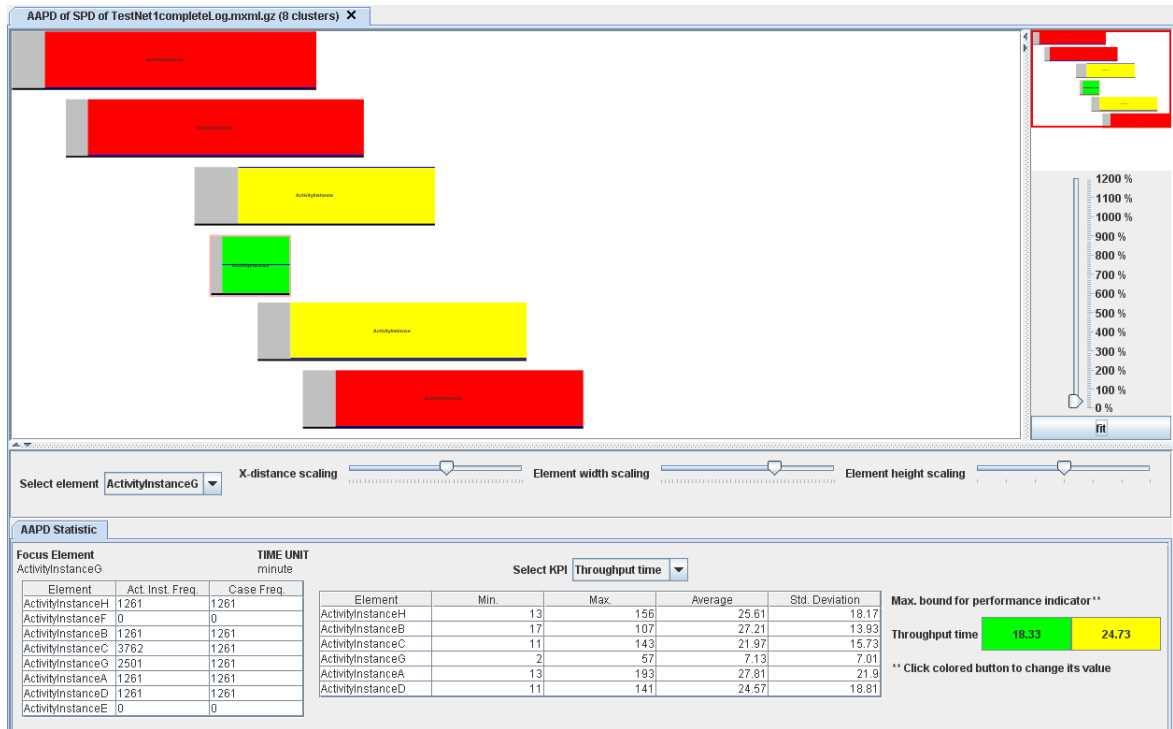


Figure C.23: AAPD visualization

C.5.2 Performance Information

The information which is provided by the AAPD visualization plug-in can be described as follows:

- **Activity Instances Frequency (Act. Inst. Freq.):** The total number of activity instances of an element which occurs in the cases where activity instances of the focus element occur.
- **Case Frequency (Case Freq.):** The total number of cases of where activity instances of an element occurs in the cases where activity instances of the focus element occur.
- **Aggregated-activities Throughput time:** The time a resource spend to perform an activity instance which refers to an element in the cases where activity instances of the focus element occur. The provided time include the minimum time, the maximum time, the average time, and standard deviation value of the average time.
- **Aggregated-activities Queuing time:** The time a scheduled activity which refers to an element spends waiting for a resource to become available in the cases where activity instances of the focus element occur. The provided time include the minimum time, the maximum time, the average time, and standard deviation value of the average time.
- **Aggregated-activities Service time:** The time that resources spend on doing an activity instance which refers to an element in the cases where activity instances of the focus element occurs. The provided time include the minimum time, the maximum time, the average time, and standard deviation value of

the average time.

- **Aggregated-activities Start time:** The time between the moment first event in cases occurs and the moment first event in the activity instance which refers to an element occurs in the cases where activity instances of the focus element occur. The provided time include the minimum time, the maximum time, the average time, and standard deviation value of the average time.
- **Aggregated-activities Intersection time:** The time span when an activity instance which refers to an element and an activity instance which refers to the focus element in the same case are performed in the same time. The provided time include the minimum time, the maximum time, the average time, and standard deviation value of the average time.

C.5.3 How to Use

To use the AAPD visualization plug-in, right click on an AAPD object which is located in the *Provided Objects* panel. Then, select “**Show**” (see Figure C.24). A similar panel as shown in Figure C.23 will be shown.

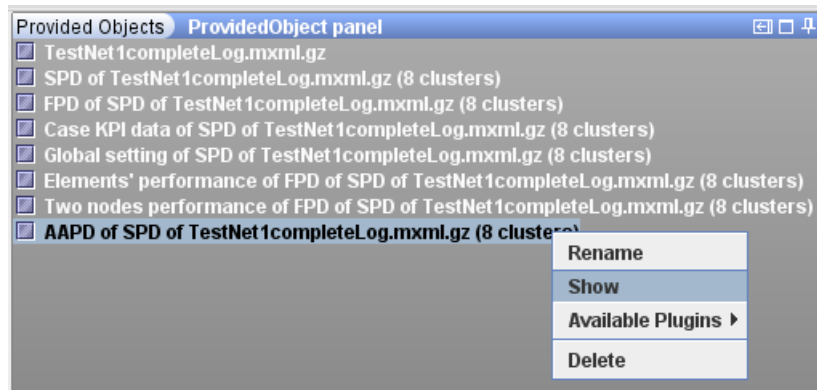


Figure C.24: How to use the AAPD visualization plug-in

To adjust the visualization of AAPD, the three sliders which are provided in the middle panel can be used (see Figure C.25). The first slider can be used to adjust the horizontal distance between elements and the focus element on the displayed SPD. The distance is scaled logarithmically, such that the elements with average starting time close to the average start time of the focus element look relatively closer than they should, while the elements with average starting time far from the focus element look relatively further than they should. The second slider can be used to adjust the width of elements in a linear scale, and the third slider is used to adjust the height of elements in a linear scale.

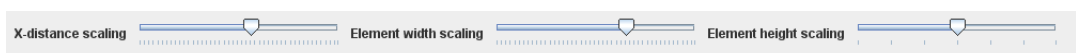


Figure C.25: The three sliders to adjust AAPD visualization

To change the focus element of the displayed AAPD, use the combo box which is located on the left side of the sliders components. Select the focus element from the

available items. After selection, a new AAPD with the selected element as the focus element will be shown in the top panel. The throughput time, start time, service time, waiting time, and intersection time for each element is shown by selecting one of the items in the combo box just beside the “**Select KPI**” label.

Similar to the FPD visualization plug-in, AAPD visualization plug-in provides boxes that can be used to adjust performance boundary values of throughput time the focus element. Figure C.26 shows the boxes that can be used to adjust the boundary values. To change the boundary value, click on the box. A dialog window similar to the dialog in C.21 will be shown. Insert the new boundary value in the dialog, and then click “**OK**”.

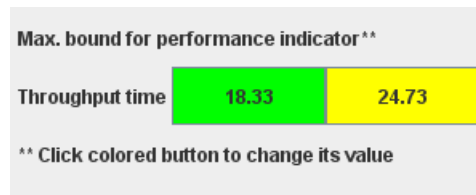


Figure C.26: Boxes to adjust the performance color of AAPD element

C.6 Global Setting GUI

C.6.1 Introduction

This plug-in visualizes a *Global settings* object which are used by both FPD visualization plug-in and AAPD visualization plug-in and provide an interface to modify the values of the variables. This plug-in only requires a Global settings object. Example of the visualization of the object is shown in Figure C.27.

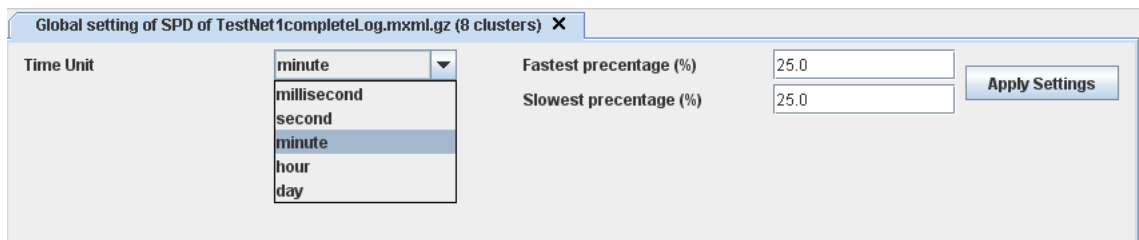


Figure C.27: Global settings object visualization

A Global settings object stores information of the time unit which is used to visualize both FPD and AAPD performance values. It also stores information of percentage boundaries which are used to show detailed statistical performance tables, such as throughput time of cases table and waiting time of a node table. As an example, Figure C.28 shows the throughput time of a case table from the “**Model-independent KPIs**” panel. In the figure, the percentage boundary for fast cases is 32%, while the percentage boundary for slow cases is 12%. This means that the average case throughput time value in column “Throughput time” with corresponding property “Fastest 32.00%” is calculated based on 32% of the cases with the fastest

throughput time, and the average case throughput time value in column “Throughput time” with corresponding property “Slowest 12.00%” is calculated based on only 12% of the cases with the slowest throughput time.

Property	Throughput time
Min	0.18
Max	362,893.45
Average	176,783.43
Std Dev	105,755.99
Fastest 32.00%	53,931.72
Slowest 12.00%	333,124.55
Rest 56.00%	213,430.99

Figure C.28: Example of a detailed statistical performance table

C.6.2 How to Use

To display a GUI that can be used to modify the values of properties within a *Global settings* object, right click the object on the *Provided Objects* panel. Then, select “**Show**” as shown in Figure C.29. After this, the object will be visualized as shown in Figure C.27.

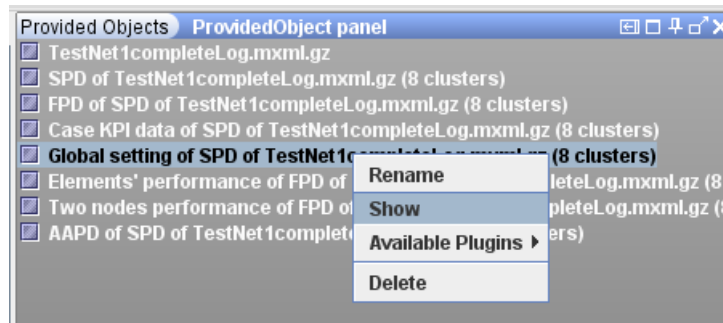


Figure C.29: How to use Global setting visualization plug-in

To change the value of time unit and percentage boundaries, select the time unit value using the provided combo box and enter the new value of percentage boundaries. The sum of both percentage boundaries must not be more than 100. Then, click “**Apply Settings**” to adjust the value. After the click, a popup window will be displayed as shown in Figure C.30. Click “**OK**” to close the window. In order to see the effect of the values modification of Global settings object, close all opened panels and then use either FPD visualization to visualizes FPD objects or AAPD visualization plug-in to visualizes AAPD objects. The time unit and percentage will be shown according to the values of variables in the Global setting object.

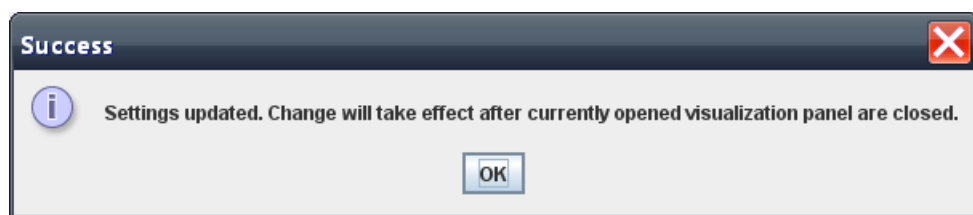


Figure C.30: Popup window after Global settings object is successfully modified

Appendix D

Evaluation

In this appendix, we provide additional information on the evaluations reported in Chapter 7. Evaluations consist of semantic identification evaluation and plug-in correctness evaluation.

D.1 Semantics Identification Evaluation

D.1.1 Purpose

The purpose of this evaluation is to identify to what extent our log replay approach manages to identify splits/joins semantics of SPD nodes.

D.1.2 Procedure

To perform the evaluation, two Petri nets as shown in Figure D.1 and Figure D.2 are created in the CPN Tools. In order to generate event logs with various event types as described in our transactional model (see Section 2.2.1) each transition in both Petri nets is implemented using 5 different transitions as described in Figure D.3. Notice that with the division to 5 different transitions, an activity instance is always started with event type “schedule” and ended with event type “complete”. The log which is generated from the Petri net in Figure D.1 is referred to as **TestNet1completeLog**, and the log which is generated by the Petri net in Figure D.2 is referred to as **TestNet2completeLog**.

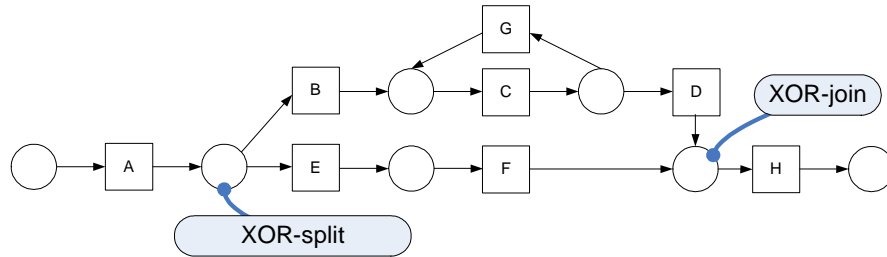


Figure D.1: Petri net 1 for evaluation

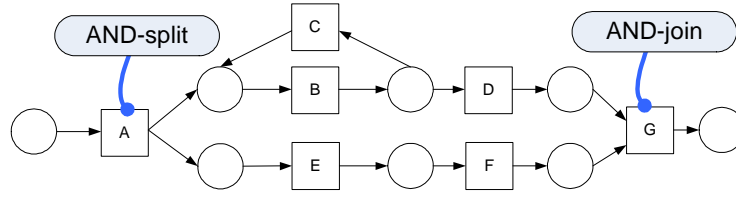


Figure D.2: Petri net 2 for evaluation

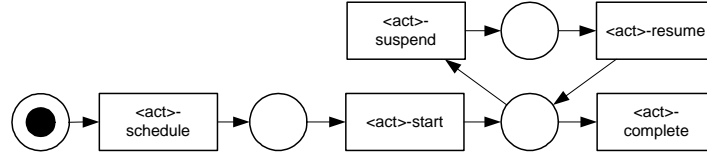


Figure D.3: Decomposition of each transition in both Figure D.1 and Figure D.2

Before log replay can be executed, we need SPDs to represent each of the Petri nets in figures D.1 and D.2. Based on the conversion approach in Section 3.2.1, the SPDs are shown in figures D.4 and D.5, respectively. To generate these two SPDs, we implement a plug-in for each SPD which generates the SPD and put it in the *Object Pool*. Note that we have to map each SPD node to activities manually before log replay can be executed.

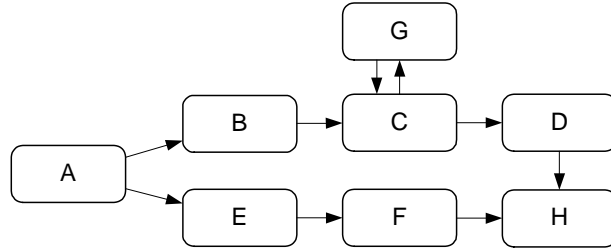


Figure D.4: SPD of the Petri net in Figure D.1

Then, event logs are replayed in the created SPDs. Log replay is performed several times, each with a different value of look-ahead window. We choose the value of look-ahead window equal to 5, 10, 15, 20, and 25. The limit of state space is set to 5000 for all experiments. Node semantics which are identified during log replay are recorded and compared to the correct semantics.

Based on the Petri net in which each SPD was generated from, the correct semantics are given as follows:

- Nodes of the SPD in Figure D.4:
 - XOR-split: node A, node C
 - AND-split: node B, node E, node G, node F, node D
 - XOR-join: node C, node H
 - AND-join: node B, node D, node E, node F, node G
- Nodes of the SPD in Figure D.5:

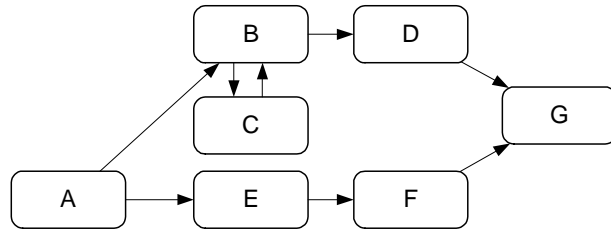


Figure D.5: SPD of the Petri net in Figure D.2

- XOR-split: node B
- AND-split: node A, node C, node D, node E node F
- XOR-join: node B
- AND-join: node C, node D, node E, node F, node G

D.1.3 Result

Replay result of experiments with event log `TestNet1completeLog` shows that for all nodes, all semantics are successfully identified in all experiments. However, the experiments with event log `TestNet2completeLog` show that there is a problem to identify AND-join semantics of node G when small values are used as look-ahead window. In this experiment, the bigger the value look-ahead window, the better prediction of the semantics (see Figure 7.5 in Section 7.1).

D.2 Multi-level of Abstraction Evaluation

D.2.1 Purpose

This evaluation is performed to validates the correctness of log replay implementation. In this evaluation, we check whether the replay log approach gives consistent performance values whenever it is used to extract performance information of a single process which is represented by several process models, each with a different level of abstraction.

D.2.2 Procedure

Several SPDs, each with different level of activity abstraction, are created to show the same process as Petri net in Figure D.1. These SPDs are shown in Figure D.6. Then, `TestNet1completeLog` is replayed in each of the SPDs. Consistency of performance values generated from the log replay in each experiments is then investigated.

D.2.3 Result

Based on our experiments, log replay in different process models with a different level of abstraction of the same process provide a consistent result. Average throughput

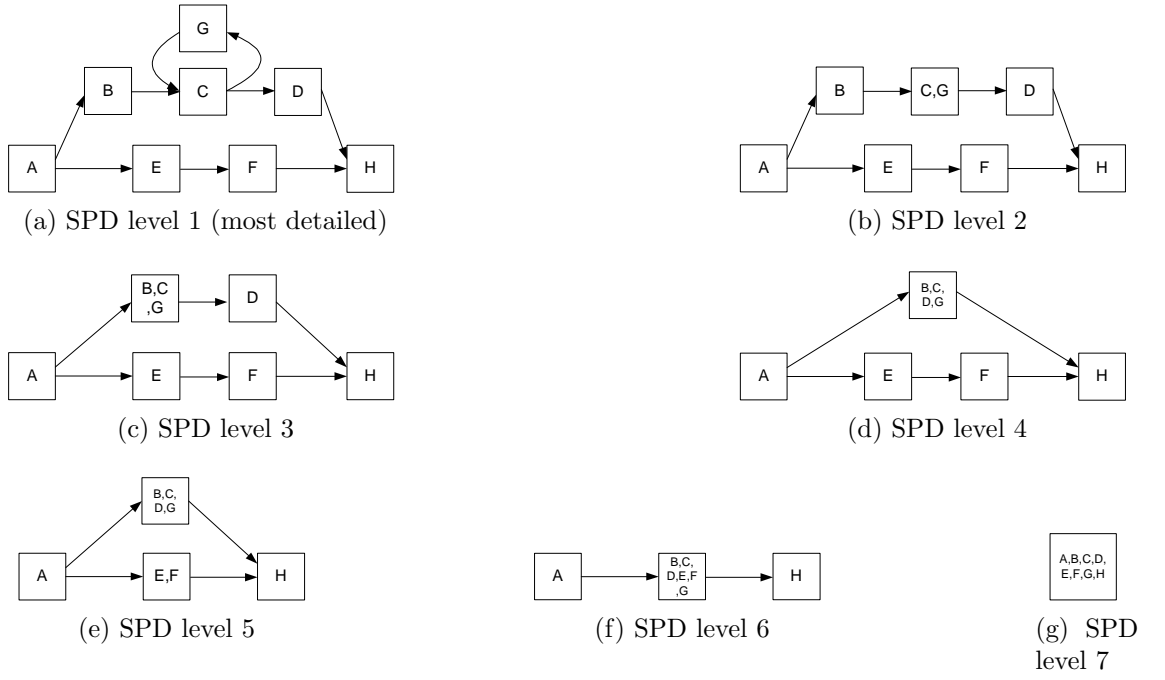


Figure D.6: SPDs of the Petri net in Figure D.1, each with a different level of activity abstraction

time of a node which aggregates several nodes in the model with one lower level of abstraction always the same with the average time spent between the moment the first event which refers to any of the abstracted nodes occurs and the moment the last event which refers to any of the abstracted nodes occurs. Hence, our log replay is successfully validated.