Stellingen

behorende bij het proefschrift

Process Mining and Verification

van

Boudewijn van Dongen

Eindhoven, 3 juli 2007
1. The definitions of a Structured Workflow Net and an Implicit Place in [1] are in conflict, i.e. the set of SWF-nets is empty. Hence the proof that the α-algorithm is capable of rediscovering all SWF-nets is far more simple than presented there.


2. The most creative problems are often invented when solutions with high expected returns are already present.

3. Using the available work from the large area of process mining and a flexible information system, such as Declare [2], one could use process logs during process enactment to suggest how people should do their job, i.e. one could completely automate the BPM lifecycle.


4. In GPRS-capable cellular networks, dedicating channels to data traffic has a negative influence on the overall Quality of Service, especially in the area where voice loads (caused by mobile phone calls) are relatively high [3].

5. The probability of making modelling mistakes when modelling EPCs is not so much related to the complexity metrics defined in [4], but rather related to the number of join connectors in the EPC [5].


6. In this time of the Internet, fast networks and online proceedings, academic libraries storing physical copies of articles and theses are becoming obsolete. One can get a PhD without ever seeing a library on the inside.

7. Whereas the theoretical complexity of an algorithm is interesting from a scientific point of view, the actual performance of an algorithm very much depends on the programming skills of the programmer that implements it.

8. Property 6.5.6 of this thesis, stating that the labels of input and output nodes of nodes in an Instance Graphs are unique has been proven to hold for all Instance Graphs. Therefore, it was an unnecessary assumption in [6].

9. The EPC in the figure above shows a choice between “Objection entered” and “Rejection finalized”. Logically, this choice is driven by the environment, not by any information system (i.e. if an objection is not sent in time, the rejection is final). Therefore this choice should be modelled as a so-called “deferred choice” [7]. Unfortunately, Event-driven Process Chains (EPCs) do not allow for expressing this construct directly and any work-around in the EPC introduces problems relating to the understandability of the model.


10. The complexity of a password is negatively correlated to the complexity of the password policy of an organization, i.e. the more constraints that need to be satisfied, the less options there are and hence the easier it is to guess a password.

11. As first proven by the program committee of the International Conference of Application and Theory of Petri nets and Other Models of Concurrency 2006 (ATPN 2006), item 9 of Definition 7.3.10 of this thesis is a necessary condition.
Process Mining
and
Verification
The work in this thesis has been carried out under the auspices of Beta Research School for Operations Management and Logistics.

This research was supported by the Dutch Organization for Scientific Research (NWO) under project number 612.066.303

Beta Dissertation Series D96

Printed by University Press Facilities, Eindhoven
Process Mining and Verification

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr.ir. C.J. van Duijn, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op dinsdag 3 juli 2007 om 14.00 uur

door

Boudewijn Frans van Dongen

geboren te Dongen
Dit proefschrift is goedgekeurd door de promotor:

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

Copromotor:
dr.ir. H.M.W. Verbeek
## Contents

### 1 Introduction

1.1 Process Modelling ............................................ 2  
1.2 Process-aware Information Systems .......................... 3  
1.3 Process Analysis .............................................. 5  
  1.3.1 Event Logs ........................................... 6  
1.4 Process Mining and Verification ............................ 8  
  1.4.1 Log-Based Verification ................................ 8  
  1.4.2 Process Discovery .................................... 9  
  1.4.3 Conformance Checking .................................. 9  
  1.4.4 Process Model Verification ............................. 10  
1.5 Roadmap ..................................................... 11  

### 2 Preliminaries

2.1 Notations .................................................. 16  
  2.1.1 Sets, Lists and Functions ............................... 16  
  2.1.2 Graph Notations ...................................... 17  
2.2 Process Logs ................................................ 19  
  2.2.1 Event Log Requirements ................................. 19  
  2.2.2 Transactional Model .................................... 20  
  2.2.3 MXML Structure ....................................... 22  
  2.2.4 Log Filtering ......................................... 25  
  2.2.5 Classification of Logs ................................ 26  
2.3 Petri nets .................................................. 30  
  2.3.1 Concepts .............................................. 30  
  2.3.2 Workflow nets ......................................... 34  
2.4 Event-driven Process Chains ................................ 34  
  2.4.1 Concepts .............................................. 35  
2.5 The Process Mining Framework ProM ......................... 37  
2.6 Running Example ........................................... 39  

### 3 Related Work

3.1 Log-based Verification ....................................... 43  
3.2 Process Model Verification ................................ 45  
  3.2.1 Verification of models with formal semantics ...... 45  
  3.2.2 Verification of informal models ...................... 45  
  3.2.3 Execution of informal models ......................... 47  
  3.2.4 Verification by design ................................ 47  
3.3 Conformance Checking ..................................... 48  
3.4 Process Discovery ........................................... 49  
  3.4.1 Theory of Regions ..................................... 49
### 3.4.2 Process Discovery on Sequential logs

50

### 3.5 Partial Order Aggregation

54

### 3.6 Outlook

56

### 4 Log-based Verification

59

#### 4.1 Introduction

59

#### 4.2 Verifying Case-Based Statements

60

#### 4.3 Example

62

#### 4.4 The language

64

##### 4.4.1 Limitations

67

#### 4.5 Example Statements

68

##### 4.5.1 Case Finalization

68

##### 4.5.2 Task Completion

69

##### 4.5.3 Retaining Familiarity

69

#### 4.6 Conclusion

71

### 5 Process Model Verification

73

#### 5.1 Introduction

74

#### 5.2 Workflow Net Verification

74

##### 5.2.1 Soundness

75

##### 5.2.2 Relaxed soundness

76

#### 5.3 EPC Verification

78

##### 5.3.1 Reduction rules

80

##### 5.3.2 Verification of the reduced EPC

84

##### 5.3.3 Using transition invariants

92

#### 5.4 Verification of the SAP reference models

94

##### 5.4.1 Further analysis of the SAP reference models

97

#### 5.5 Conclusion

102

### 6 Process Discovery

105

#### 6.1 Introduction

106

#### 6.2 Region-based Process Discovery

107

##### 6.2.1 Theory of Regions

107

##### 6.2.2 Region-based Process Discovery

109

##### 6.2.3 Iterative Region Calculation

112

##### 6.2.4 Complexity

116

##### 6.2.5 Mining Quality

117

#### 6.3 Log Abstraction

118

##### 6.3.1 Log-based Ordering Relations

118

##### 6.3.2 Extension of Log-based Ordering Relations

121

##### 6.3.3 Summary

126

#### 6.4 The \(\alpha\)-algorithm

127

##### 6.4.1 Details of the \(\alpha\)-algorithm

130

##### 6.4.2 Mining Quality

131

#### 6.5 Partial Ordered Process Instances

131

##### 6.5.1 Causal Runs

132

##### 6.5.2 Extracting Causal Runs of Sequential Process logs

132

##### 6.5.3 Representing Instance Graphs as Petri nets

136

##### 6.5.4 Representing Instance Graphs as EPCs

137

##### 6.5.5 Summary

139

#### 6.6 Aggregation of Partial Orders

140

##### 6.6.1 Aggregating instance graphs

140

##### 6.6.2 Representing Aggregation Graphs as EPCs

145
| 6.6.3 Representing Aggregation Graphs as Petri nets | 147 |
| 6.6.4 Mining Quality | 147 |
| 6.7 Conclusion | 148 |

7 Process Discovery on Partially Ordered Logs 151

| 7.1 Aggregating MSCs | 152 |
| 7.1.1 Message Sequence Charts | 152 |
| 7.1.2 MSCs to Instance Graphs | 153 |
| 7.1.3 Aggregation of MSCs | 155 |
| 7.1.4 Bookstore Example | 157 |
| 7.1.5 Mining Quality | 158 |
| 7.2 Aggregating Runs of EPCs | 158 |
| 7.2.1 Instance EPCs | 160 |
| 7.2.2 Aggregating instance EPCs | 162 |
| 7.2.3 Mining Quality | 162 |
| 7.3 Aggregating Runs of Petri nets | 165 |
| 7.3.1 Causal Nets | 165 |
| 7.3.2 Aggregating Petri net runs | 166 |
| 7.3.3 Aggregation with Known Labels | 169 |
| 7.3.4 Aggregation with Duplicate Transition Labels | 170 |
| 7.3.5 Aggregation with Unknown Place Labels | 172 |
| 7.3.6 Mining Quality | 179 |
| 7.4 Conclusion | 179 |

8 ProM 181

| 8.1 ProMimport | 184 |
| 8.2 Log Filters | 184 |
| 8.2.1 Application of Log Filters | 184 |
| 8.2.2 Default Filter | 186 |
| 8.2.3 Start Event Filter | 188 |
| 8.2.4 Final Event Filter | 188 |
| 8.2.5 Event Filter | 188 |
| 8.2.6 Duplicate Task Filter | 188 |
| 8.2.7 Transitive Closure Filter | 190 |
| 8.2.8 Transitive Reduction Filter | 190 |
| 8.2.9 Conclusion | 190 |
| 8.3 Mining Plug-ins | 192 |
| 8.3.1 Log abstraction plug-in | 193 |
| 8.3.2 α-algorithm plug-in | 195 |
| 8.3.3 Region Miner | 195 |
| 8.3.4 Partial Order Generator | 195 |
| 8.3.5 Partial Order Aggregator | 197 |
| 8.3.6 Multi-phase Macro Plug-in | 199 |
| 8.3.7 Overview | 199 |
| 8.4 Analysis Plug-ins | 200 |
| 8.4.1 EPC Verification Plug-in | 200 |
| 8.4.2 LTL Checker | 204 |
| 8.4.3 Default LTL Checker | 204 |
| 8.4.4 Petri net Analysis | 206 |
| 8.4.5 Region Calculator | 206 |
| 8.4.6 Conclusion | 209 |
| 8.5 Conversion Plug-ins | 209 |
| 8.5.1 Aggregation Graph to EPC | 210 |
## Contents

8.5.2 EPC to Petri net ........................................... 210
8.5.3 EPC Reduction Plug-in ................................... 213
8.5.4 Regions to Petri net ..................................... 213
8.5.5 Conclusion .................................................. 214
8.6 Import and Export Plug-ins .................................. 215
8.6.1 MXML Format ............................................... 216
8.6.2 Log Connection ............................................ 216
8.6.3 Log Filter .................................................. 218
8.6.4 LTL Template Format ..................................... 218
8.6.5 AML Format ................................................ 220
8.6.6 Aris Graph Format ....................................... 220
8.6.7 VDX Format ................................................. 221
8.6.8 EPML Format ............................................... 223
8.6.9 PNML Format ............................................... 223
8.6.10 TPN Format ................................................ 223
8.6.11 Conclusion ................................................ 225
8.7 Example Walkthrough ........................................ 225
8.7.1 Mining ...................................................... 226
8.7.2 Analysis ..................................................... 226
8.7.3 Conversion .................................................. 229
8.7.4 Export ....................................................... 230
8.7.5 Conclusions ................................................ 230

9 Conclusion ....................................................... 235
9.1 Verification Contributions .................................... 235
  9.1.1 Log-based Verification .................................. 235
  9.1.2 Event-driven Process Chain Verification ............... 237
  9.1.3 ProM plug-ins ............................................ 238
9.2 Process Mining Contributions ................................ 238
  9.2.1 Log Abstraction .......................................... 240
  9.2.2 Partial Order Generation ............................... 240
  9.2.3 Partial Order Aggregation .............................. 241
  9.2.4 Other Partial Orders .................................... 241
  9.2.5 ProM plug-ins ............................................ 241
9.3 Limitations and Future Work ................................ 242
  9.3.1 Process Mining ........................................... 242
  9.3.2 Verification ............................................... 242
  9.3.3 Summary .................................................. 243

Bibliography ....................................................... 245

Summary .......................................................... 257

Samenvatting ..................................................... 261

Acknowledgements ................................................. 265

Curriculum Vitae .................................................. 267
Chapter 1

Introduction

Many of today’s businesses are supported by information systems. In the past few years, there has been a significant change in the design of these information systems. Instead of designing new, customer specific software systems for each customer, process-aware information systems using e.g., workflow management technology, have become the de-facto standard. Such information systems are typically designed to support many different businesses, by relying on a good description of the process in terms of a process model. Therefore, the difficulties in designing complex information systems are no longer in the actual “programming” of the system, but in describing the processes that need to be supported as precise as possible. The scientific discipline that studies such systems is commonly referred to as Business Process Management, or BPM.

In Figure 1.1, we present the classical BPM life cycle. In the traditional approach, you start with a process design. Then, the design is used to configure some process-aware information system in the configuration phase. After the process has been running for a while in the enactment phase and event logs have been collected, diagnostics can be used to develop another (and preferably better)
Chapter 1 Introduction

design. By taking this approach, large information systems are assumed to evolve over time into larger and better systems.

An important question in the area of business process management is whether or not people within organizations are following policies that have been introduced over time. In Chapter 4, we answer this question by introducing a means to formalize policies and rules and to verify them using an event log. Using our approach, process analysts can categorize the vast amounts of data recorded by the running information system into partitions that require further analysis in order to optimize the operational process.

Note that in practice, processes are often implicit (i.e. they are not designed as such, but emerged from daily practice) or not enforced by any system. However, when these processes are analyzed, they are captured by process models of some sort. Obviously, when making such models, mistakes should be avoided. Event-driven Process Chains (EPCs) are a well-known modelling language for the informal modelling of processes. In this thesis, we present a way to analyze EPCs while keeping in mind that these EPCs are informal, i.e. they are not intended as an executable specification of a process.

An important research question in the BPM area is whether it is possible to derive a process model describing an operational process directly from the running system, i.e. is it possible to use an event log to generate process models? In this thesis, we extend the existing work in that area in two directions, such that (1) the resulting process models are accurate, i.e. they are always guaranteed to describe the operational process fully and (2) as much information as possible is used to generate the process models, such as information on time and ordering of events in a log.

In this chapter, we introduce the concept of process modelling (Section 1.1) in more detail. Then, we give some insights into typical kinds of process-aware information systems in Section 1.2 and their uses in practice. We conclude the chapter with an introduction to process analysis in Section 1.3 and we show how the work in this thesis contributes to the research areas of process mining and verification in Section 1.4.

1.1 Process Modelling

When people talk about their business they tend to use diagrams. Especially when they want to explain how their business works and what is happening in their companies. Whether it is the management structure, or a description of the flow of goods through the various parts of a warehouse, diagrams are a useful aid in alleviating the complexity problems faced.

The reason for using diagrams is simple. We humans are very good at understanding diagrams, especially if they are accompanied by some explanatory texts, or some verbal explanations. The diagrams we use to discuss processes are
what we call *process models* and when process models are used for discussion, we typically use them as descriptive models. The process models used in this thesis focus on the control flow aspect of a process, i.e. they describe in which order activities need to be performed and hence in which way cases flow through the information system.

Increasingly, organizations are, either explicitly or implicitly, driven by processes of some sort. Examples of such processes are the handling of insurance claims in an insurance company, or the application for a residence permit in the Immigration Service, but also the flow of patients through a hospital or the delivery of newspapers. Therefore, since the 1970s, *process modelling* has become more and more popular. The idea behind process modelling is simple. We describe our business in terms of processes and the communication between processes and the environment, such that each description is unambiguous. In this way, anybody who knows the same language, is capable of understanding the process without any further explanation, by just looking at the schematics. Note that process modelling does not decrease the complexity of describing the processes under consideration. However, it helps people in making the problem at hand more insightful.

Once the formal languages are defined, all that remains is to make computers understand these process models to some extend, i.e. programs have to be developed that can interpret these process models. These models can then be used to enforce a certain model onto the people working in a process, for example when dealing with insurance claims. However, these models can also be used by more flexible systems to support the operational process without enforcing (or while only partially enforcing) the model onto the operational process. A postal worker for example will have to punch the clock when he leaves for his delivery round and when he comes back. However the order in which he delivers mail is not enforced.

Large information systems that can deal with process models in one way or the other are commonly referred to as *process-aware information systems*.

## 1.2 Process-aware Information Systems

At this moment, process-aware information systems are widely used in practice. At the basis of most of these systems lie process models of some kind. However, the way systems enforce the handling of cases is different for the various types of systems. On the one hand, there are systems that enforce a given process description onto all users, while some other systems only provide an easy way of handling access to files, without enforcing a particular process. As a result of this, information systems are used in very diverse organizations and with all kinds of expectations.

An example of a collection of models that is often used as a reference for
building large information systems such as SAP [109], is the SAP reference model. Rosemann and Van der Aalst explain in [150] that the SAP reference model is one of the most comprehensive models [57]. Its data model includes more than 4000 entity types and the reference process models cover more than 1000 business processes and inter-organizational business scenarios. These models are typically not enforced onto the people involved in a process during execution. Instead they merely serve as a guideline in the configuration of an information system.

In [82] it was shown that, even though each system has its individual advantages and disadvantages, these systems can be divided in several groups. In Figure 1.2, the authors of [72] give four types of information systems, and position them with respect to the structure of the process that is dealt with and whether they are data or process driven. In Figure 1.3, they give the trade-offs that are made for each of these four types of systems with respect to flexibility, support, performance and design effort.

Production workflow systems such as for example Staffware [172] are typically used in organizations where processes are highly standardized, and volumes are big (i.e. a lot of cases are to be dealt with in parallel). These systems not only handle data, but enforce a certain process definition to be followed by the letter. Case handling systems such as FLOWer [38] on the other hand, are typically used in environments where people have a good understanding of the complete process. This allows these so-called “knowledge workers” to handle cases with more flexibility. In the end however, the case handling system provides support by keeping structure in both the data involved and the steps required. Ad-hoc workflow systems such as InConcert [112] allow for the users to deviate completely from given processes. Process definitions are still provided, but not enforced on an execution level. They merely serve as reference models. Systems like Adept [146] allow processes to be changed at both the instance level (i.e. during execution, similar to InConcert), as well as at the type level, while migrating running instances from the old to the new process. The final category of systems,
groupware, provide the most flexibility to the users. Systems such as Lotus Notes provide a structured way to store and retrieve data, but no processes are defined at all, hence users can do tasks in any order.

The BPM lifecycle shown in Figure 1.1 shows how over time, process-aware information systems are supposed to evolve into better systems, that more accurately support the process under consideration. Traditionally, information systems play an active role in the enactment of a process. The other three phases however are typically human-centric, i.e. trained professionals are required for diagnosing process support, designing new process models and implementing new information systems which can than be enacted again. Process mining focuses on supporting these professionals throughout all phases, by analyzing information logged during enactment to gain a better insight into the process under consideration.

1.3 Process Analysis

As organizations continuously try to improve the way they do business, processes are analyzed with the purpose of increasing performance. Especially since large process aware information systems typically log the steps performed during enactment of an operational process in some sort of event log, there is plenty of input available to perform process analysis. An example of an event log is shown in Table 1.1, where a partial log of an invoice handling process is shown.

Processes, such as “invoice handling” and “order processing” are usually called the operational processes and the when people are working on such processes, they are typically involved in cases or process instances, such as “invoice 1029” or “order 2344”. Especially in process-aware information systems, these processes are

<table>
<thead>
<tr>
<th>Process</th>
<th>Case</th>
<th>Activity</th>
<th>Event type</th>
<th>Event type</th>
<th>Timestamp</th>
<th>Originator</th>
</tr>
</thead>
<tbody>
<tr>
<td>invoice handling</td>
<td>invoice</td>
<td>payment</td>
<td>start</td>
<td>10/24/2006</td>
<td>12:00</td>
<td>John</td>
</tr>
<tr>
<td></td>
<td>1029</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>invoice handling</td>
<td>invoice</td>
<td>payment</td>
<td>complete</td>
<td>10/24/2006</td>
<td>12:06</td>
<td>Mary</td>
</tr>
<tr>
<td></td>
<td>1039</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>order processing</td>
<td>order</td>
<td>shipment</td>
<td>assign</td>
<td>10/24/2006</td>
<td>12:07</td>
<td>SYSTEM</td>
</tr>
<tr>
<td></td>
<td>2344</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>invoice handling</td>
<td>invoice</td>
<td>notify</td>
<td>complete</td>
<td>10/24/2006</td>
<td>12:08</td>
<td>SYSTEM</td>
</tr>
<tr>
<td></td>
<td>1039</td>
<td>manager</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>invoice handling</td>
<td>invoice</td>
<td>payment</td>
<td>complete</td>
<td>10/24/2006</td>
<td>12:15</td>
<td>John</td>
</tr>
<tr>
<td></td>
<td>1029</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>order processing</td>
<td>order</td>
<td>shipment</td>
<td>start</td>
<td>10/24/2006</td>
<td>12:30</td>
<td>Bill</td>
</tr>
<tr>
<td></td>
<td>2344</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
typically modelled by *process models* and the information system records *events* related to these processes. These events are stored in *event logs* and each event typically refers to a *case*. In our example log of Table 1.1, there are three cases, namely “invoice 1029”, “invoice 1039” and “order 2344”. Furthermore, a company has knowledge about the *desired or undesired properties* of each operational process in some form (for example company policies, such as the statement that a manager should be notified about all payments, or the requirement that each case will be correctly handled within reasonable time).

### 1.3.1 Event Logs

Event logs can be very different in nature, i.e. an event log could show the events that occur in a specific machine that produces computer chips, or it could show the different departments visited by a patient in a hospital. However, all event logs have one thing in common: They show occurrences of *events* at specific moments in *time*, where each event refers to a specific *process* and an *instance* thereof, i.e. a *case*.

Event logs, such as the one shown in Table 1.1, but also process models serve as a basis for process analysis. One could for example consider a process model and verify whether its execution will always lead to a given outcome. Furthermore, event logs can be compared against models to see whether they match, and the event log itself can be analyzed to check whether company policy was followed.

Consider Table 1.1, which shows an example of an event log. When analyzing this small log, it is easy to see that the information system it originated from was handling two processes, i.e. “invoice handling” and “order processing”. Furthermore, we can see that between 12:00 at the 24th of October 2006 and 12:30 that day, five events were logged referring to two activities. First, John started the payment activity for “invoice 1029”, an activity which he completed 15 minutes later. Furthermore, Mary also completed a payment activity (which she probably started before) at 12:06, after which the system notified the manager. Within the “order processing” process, the “shipment” activity was assigned to “Bill” by the system and Bill started that activity at 12:30.

This little example already shows how expressive logs can be. For example, we derived the fact that the system assigned the shipment activity to Bill, since we saw the event “assign shipment” performed by “SYSTEM” and the event “start shipment” performed by Bill. If we see such assignments more often, we could derive that the information system from which this log was taken uses a push-system, i.e. the system decides who does what.

Furthermore, since in one case a manager was notified about a payment (i.e. the one made by Mary) and in another case he was not (i.e. the payment made by John), we can conclude that sometimes managers are notified about payments. Although the latter statement might seem trivial, it can be of the utmost importance. Consider for example that there is a company policy that states that
for all payments, a notification should be send to the manager. Obviously, this very simple and small log shows that this policy is violated, assuming that the notification for invoice 1029 does not take more than 15 minutes.

Recall that we stated that, from Table 1.1, we can derive the fact that the information system uses a push mechanism to assign work to people. Deriving such a statement is what we call process analysis from an organizational perspective. In the context of process mining, we distinguish three different perspectives: the process perspective, the case perspective and the organizational perspective. For each perspective, we use dedicated process analysis techniques.

**Process Perspective** The process perspective focuses on the control-flow, i.e., the ordering of activities, as shown in Table 1.1. The goal of mining this perspective is to find a good model describing the process under consideration. An example of a statement in the process perspective would be that the “shipment” activity for a specific case in the “order processing” process is always the last activity for that case.

**Case Perspective** The case perspective focuses on properties of cases (in particular data). Cases can be characterized by their path in the process or by the originators working on a case. However, cases can also be characterized by the values of the corresponding data elements. For example, saying that for “invoice 1029” the manager was not informed about a payment activity, is a property of a case, i.e. it is a statement made from a case perspective.

**Organizational Perspective** The organizational perspective focuses on the originator field (cf. Table 1.1), i.e., which performers are involved and how are they related. The goal is to either structure the organization by classifying people in terms of roles and organizational units or to show relation between individual performers.

With respect to the three analysis perspectives, it is important to realize that they cannot be seen in isolation and that they are often highly related, e.g. consider the situation where we derived the fact that John is a manager. In that case, the statement that a manager was not notified about the payment made for invoice 1029 is no longer true.

Figure 1.4 shows the relations between an operational process, the models that describe it and the event logs generated by the information system. The figure shows that a model of an operational process can be used to configure an information system that supports or controls the process under consideration. The information system then records event logs of the operational process. Furthermore, Figure 1.4 shows how the research areas of process mining and verification relate to these entities, by showing how event logs, process models and some desired or undesired properties can be used for log-based verification, process model verification, process discovery and conformance checking, which are discussed in Section 1.4.
1.4 Process Mining and Verification

This thesis is presenting methods and approaches in the areas of process mining and process verification. Both areas should be seen as part of the vast area of process analysis. Figure 1.4 graphically show the relations between event logs, (un)desired properties (such as company policy, etc) and process models. When models, event logs or properties are checked against each other then this is called verification, e.g. when a log is compared against some properties we say that that is log-based verification.

![Figure 1.4: Process Mining and Verification.](image)

In this section, we introduce each of the four relations between (un)desired properties, process models and event logs in some more detail. Furthermore, we show how the problems relating to log-based verification, process discovery, conformance checking and process model verification can be addressed from the different analysis perspectives.

1.4.1 Log-Based Verification

During the enactment of a process, the environment changes faster than the configuration of the information system supporting the process and therefore, next to a process definition, there typically exist policies that need to be followed, which are not enforced by the information system. Furthermore, for each process, performance indicators are typically used to indicate whether or not a company fulfills its own expectations.

Policies and performance indicators refer to desired and undesired properties, i.e. they are stored in some form and they can be linked to the operational processes supported by an information system. The research topic of log-based
verification focuses on how to automatically verify whether such properties hold or not, using an event log as input, e.g. using an event log such as the one in Table 1.1 to verify whether managers are notified about all payments. Log-based verification can be performed from all perspectives, e.g. by checking if normal employees are not executing the activities that should be performed by managers you apply log-based verification from an organizational perspective.

When a process is being redesigned in the design phase of the BPM lifecycle, similar log-based verification techniques can be used to verify user-statements about a process. In other words, if a user explains to a process designer how the process works, the designer can use the same techniques to objectively verify\(^1\) that statement on the recorded event log, thus reducing the possibility for error in the design phase.

### 1.4.2 Process Discovery

Even if process models are designed to the best of the designer’s capabilities, it is still not guaranteed that they indeed correctly model the process under consideration. This heavily depends on whether they fit to what the people involved in the process want. Furthermore, if the models are enforced during execution, then they are the specification of the operational process and hence they model it correctly. For this purpose, the research topic of process discovery focuses on using event logs to extract information about the operational process.

Process discovery is typically applied to gain insights into the operational process, for example to monitor patient flows in hospitals, or the routing of complex products through the different manufacturing stages. Process mining supports the manager or process analyst in finding out what actually happened in the organization. Furthermore, the discovered process models can be used to support a process designer during process design or redesign (i.e. in the design phase).

In contrast to log-based verification, the goal of process discovery is to derive some sort of model that describes the process as accurately as possible. In that respect, we could argue that process discovery focuses on the process perspective of process mining. However, process discovery techniques can be applied from all perspectives, e.g. constructing decision trees for the case perspective and social networks for the organizational perspective.

### 1.4.3 Conformance Checking

So far, we have shown how an event log can be used to check desired and undesired properties and how we can derive information about an operational process from that log. A third way in which event logs can be used is to check whether the

\(^1\)This kind of verification is often performed by asking people involved in a process and find the common consensus, which can be considered as a subjective verification.
operational process actually adheres to the modelled process. This process is called *conformance checking*.

In Section 1.2, we introduced several types of information systems. Especially when an information system is flexible, using less structured process models as a basis, *conformance checking* (or conformance testing), i.e. checking whether the operational process adheres to the given process model, is of great importance.

Consider again our example log of Table 1.1 and assume that it is policy that a manager is notified of each payment made. In a production workflow system, such a policy would be explicitly modelled in the process model, i.e. the process model explicitly shows that this notification is sent and the information system enforces that this is indeed done. In a groupware system however, this policy is not necessarily enforced, whereas the model of the process made by a process designer in the design phase does show this policy explicitly. The idea behind conformance checking is that such a conceptual model is used, together with the event log to see to what extent these two match.

Again, conformance checking can be applied from all perspectives, e.g. when checking whether only managers perform those activities that should only be performed by managers, conformance checking is applied from an organizational perspective.

### 1.4.4 Process Model Verification

Although it is important to diagnose your processes continuously, it is even more important to configure an information system in such a way that it supports the process as good as possible. Therefore, it is of the utmost importance to check if the conceptual models designed in the design phase are semantically correct. This is what is called process model verification, which we introduce in Subsection 1.4.4.

In process mining, the event log is the starting point for most algorithms, i.e. from such event logs, process models are discovered, or policies are checked. However, whether a process model is designed by a process designer, or is automatically derived from a log using process discovery, such process models can contain errors. Finding these errors is what is called *process model verification*.

In process model verification, it is assumed that a process model is present for the process under consideration. This model can be an executable specification, but it is more likely to be a conceptual model of the process. However, whether a process is modelled by an executable specification or by a conceptual model, it is still important that that model does not contain errors, i.e. the model should adhere to the syntax of the modelling language and its semantics should be such that no undesired behaviour occurs. Such clearly undesirable behaviour for example entail deadlocks (i.e. the process gets “stuck” somewhere) or livelocks (i.e. a case can never be completed). Again, process model verification techniques can be applied from all perspectives, for example by analyzing data models or
organizational models.

In this introduction, we have shown four areas of process mining and verification and how techniques belonging to each area can be applied from different perspectives. In the remainder of this thesis, we introduce several techniques and we show how to apply them from some perspectives. In which order we do so is discussed in the roadmap.

1.5 Roadmap

In Subsection 1.4.2, we introduced three perspectives from which process analysis can be performed, i.e. the process perspective, the case perspective and the organizational perspective. In this thesis, we do not consider the organizational perspective. However, we do consider the other two. Next to the theory we present in this thesis, most of the work has been implemented in our tool called the ProM framework.

For the algorithms presented in this thesis, we do not focus on one process modelling language. Instead, we use two modelling languages throughout this thesis. The first language is a formal language called Petri nets or a sub-class thereof called workflow nets. In Section 2.3, we introduce the concepts of Petri nets and workflow nets in detail, however for now it is important to realize that (1) workflow nets are widely used in practice for modelling business processes and (2) workflow nets have a formal and executable semantics, i.e. they can be used for enacting a business process directly.

The second language we use is an informal modelling language called Event-driven Process Chains (or EPCs). The details are introduced in Section 2.4, but for now, it is important to realize that (1) Event-driven Process Chains are also widely used in practice for making conceptual models, supported by systems such as SAP and the enterprise modelling tool Aris [161], and (2) Event-driven Process Chains are not meant to be executable, i.e. there do not exist clear executable semantics for these models. Instead these models are typically used by process designers to describe processes in an informal, but structured way.

After discussing the related work in the area of process mining and verification in Chapter 3, we introduce our approach towards log-based verification in Chapter 4. In that chapter, we present a language to specify desired and undesired properties and we present a means to check these properties for each recorded case in an event log, hence the approach presented in Chapter 4 focuses on process analysis. Using the approach of Chapter 4, we are able to analyze event logs in a structured way and to identify those parts of an event log that require further analysis. Furthermore, since our approach allows for the properties to be parameterized, they can be re-used in different organizations and on different event logs with relative ease.

In Chapter 5, we continue with verification, however, we introduce our veri-
fication approach towards the verification of EPCs. Well-known verification approaches for formal models, such as workflow nets, are usually only applicable if these models are directly used for the enactment of an information system. Our approach on the other hand uses well-known verification approaches for workflow nets, but we apply our approach to EPCs, i.e. an informal modelling language, hence making it applicable for a wider range of information systems. The approach we present in Chapter 5 relies on the process owner’s knowledge of an operational process, which is not explicitly present in the process model. The explicit use of human judgement is an often missing factor in the area of process model verification.

In Chapter 6, we move from verification to process mining, or more specifically process discovery, and we present several algorithms that derive a process model from an event log. One of the approaches we present is a one-step approach, which takes an event log as input and generates a process model from it, in terms of a Petri net. The other approaches first abstract from the given event log, i.e. they first derive causal dependencies and parallel relation between events. Then, they use these relations to generate process models in terms of both workflow nets and EPCs. Furthermore, one of the approaches guarantees that the resulting EPC will never be considered incorrect by the verification approach of Chapter 5.

Figure 1.5: Overview of this thesis.
In some situations, the causal dependencies and parallel relations we derived from the event log in Chapter 6 are given as input during process mining. For example, when a process is modelled in terms of a set of example scenarios, as is common when using Message Sequence Charts (MSCs), these relations are explicitly present in terms of partial orders on events. Models such as MSCs, explicitly show the causal and parallel dependencies between activities (i.e. after sending message X to company A, you send message Y to company C and at the same time, you wait for a reply from company X). Hence, these models contain more information than ordinary event logs.

Also for Workflow nets and EPCs, methods exist to describe operational processes in terms of example scenarios, usually called runs. Therefore, in Chapter 7, we introduce some algorithms for the situation where we have a set of example scenarios that can be used to derive an overall model describing the operational process. Specifically, we introduce a number of so-called aggregation algorithms that aggregate a set of partial orders, where these partial orders are specified in several languages.

Figure 1.5 graphically shows how all algorithms and approaches presented in this thesis belong together. It shows all objects, such as EPCs and Petri nets and in which sections these objects are used together. For example, Section 6.3 presents how a given event log can be abstracted from to get ordering relations.

Finally, before we conclude this thesis in Chapter 9, we first introduce our tool called ProM in Chapter 8, in which most of the work is implemented.

We conclude this chapter by summarizing the structure of this thesis:

Chapter 2 introduces preliminary concepts that we use throughout this thesis, such as workflow nets and EPCs.

Chapter 3 provides an overview of the related work in the areas of process mining and verification.

Chapter 4 introduces an approach towards log-based verification. It introduces a language for specifying properties and it shows how these properties can be checked on event logs.

Chapter 5 explains an algorithm for the verification of informal models, modelled in terms of EPCs.

Chapter 6 presents several algorithms for process discovery, i.e. to derive process models from event logs, such as the one in Table 1.1.

Chapter 7 presents an extension to the algorithms of Chapter 6, tailored towards the aggregation of partial orders found in practice, for example MSCs.

Chapter 8 then presents the ProM framework, which is a toolkit in which most of the work of this thesis is implemented.

Chapter 9 finally concludes the thesis with some final remarks.
Chapter 2

Preliminaries

Before we discuss related work in Chapter 3, we first present some useful notation and basic concepts such as process logs, Petri nets and EPCs that we use throughout this thesis. Figure 2.1 shows the part of the whole thesis we introduce in this chapter.

![Diagram](image-url)  

*Figure 2.1: Overview of preliminaries chapter.*
2.1 Notations

As the title suggest, this thesis presents several approaches towards process mining and verification. In most of the following chapters, we use mathematical notations to introduce definitions or proofs. Therefore, we start by introducing the used notations.

2.1.1 Sets, Lists and Functions

Some standard concept in mathematics are sets, lists and functions. In this subsection, we present the notation for these concepts, as well as some standard operators.

**Definition 2.1.1. (Set notation)**

For sets, we define the standard operators:

- Let $s_1, s_2$ be two elements. We construct a set $S$ containing both elements by saying $S = \{s_1, s_2\}$, i.e. we use $\{ \text{ and } \}$ for the enumeration of elements in a set,
- $s \in S$ checks whether an element $s$ is contained in $S$,
- $S = S_1 \times S_2$ is the cartesian product of two sets, i.e. $S = \{(s_1, s_2) \mid s_1 \in S_1 \land s_2 \in S_2\}$.
- The union of two sets is defined as $S = S_1 \cup S_2$, i.e. the set $S$ contains all elements of $S_1$ and $S_2$,
- The intersection of two sets as $S = S_1 \cap S_2$, i.e. the set $S$ contains all elements that are contained in both $S_1$ and $S_2$,
- Removing the elements of one set from the other is denoted as $S = S_1 \setminus S_2$, i.e. the set $S$ contains all elements of $S_1$ that are not contained in $S_2$.
- $|S|$ represents the number of elements in a set, i.e. the number of $s \in S$,
- $S \subseteq S_1$, i.e. $S$ is a subset of $S_1$,
- $S \subset S_1$ stands for $S \subseteq S_1 \land S \neq S_1$, i.e. $S$ is a proper subset of $S_1$,
- $\mathcal{P}(S) = \{S' \mid S' \subseteq S\}$ is the powerset of $S$, i.e. the set of all subsets of $S$,
- $\emptyset$ is a constant to denote an empty set, i.e. for all sets $S$ holds that $\emptyset \subseteq S$.

In this thesis, we typically use uppercase letters to denote sets and lowercase letters to denote the elements of that set. Furthermore, we use $\mathbb{N}$ to denote the set of natural number, i.e. $\mathbb{N} = \{0, 1, 2, \ldots\}$.

**Definition 2.1.2. (Function notation)**

Let $D$ and $R$ be two sets. We define $f : D \to R$ as a function, mapping the elements of $D$ to $R$, i.e. for all $d \in D$ holds that $f(d) \in R$, where we denote the application of function $f$ to the element $d$ as $f(d)$. Furthermore, we lift functions to sets, by saying that for all $D' \subseteq D$ holds that $f(D') = \{f(d) \mid d \in D'\}$. For a function $f : D \to R$, we call $\text{dom}(f) = D$ is the domain of $f$ and $\text{rng}(f) = f(\text{dom}(f))$ is the range of $f$. 

Using functions, we define the standard concept of a multi-set or bag.

**Definition 2.1.3. (Multi-set, Bag)**

Let $D$ be a set and $F : D \to \mathbb{N}$ a function mapping the elements of $D$ to the natural numbers. We say that $F$ is a bag, where we use a shorthand notation using square brackets for the enumeration of the elements of a bag, e.g. $[d_1^2, d_2, d_3^3]$ denotes a bag, where $D = \{d_1, d_2, d_3\}$ and $F(d_1) = 2$, $F(d_2) = 1$ and $F(d_3) = 3$. As a shorthand notation, we assume that for all $d \notin D$, holds that $F(d) = 0$. Furthermore, a set $S$ is a special case of a bag, i.e. the bag $F : S \to \{1\}$.

**Definition 2.1.4. (Bag notation)**

Let $X : D_1 \to \mathbb{N}$ and $Y : D_2 \to \mathbb{N}$ be two bags. We denote the sum of two bags $Z = X \cup Y$, i.e. $Z : D_1 \cup D_2 \to \mathbb{N}$, where for all $d \in D_1 \cup D_2$ holds that $Z(d) = X(d) + Y(d)$. The difference is denoted $Z = X - Y$, i.e. $Z : D' \to \mathbb{N}$, with $D' = \{d \in D_1 \mid X(d) - Y(d) > 0\}$ and for all $d \in D'$ holds that $Z(d) = X(d) - Y(d)$. The presence of an element in a bag $(a \in X) = (X(a) > 0)$, the notion of sub-bags $(X \leq Y) = (\forall d \in D_1, X(d) \leq Y(d))$, and the size of a bag $|X| = \sum_{d \in D_1} X(d)$ are defined in a straightforward way. Furthermore, all operations on bags can handle a mixture of sets and bags.

Besides sets and bags, we also use sequences of elements.

**Definition 2.1.5. (Sequence)**

Let $D$ be a set of elements. A list $\sigma \in D^*$ is a sequence of the elements of $D$, where $D^*$ is the set of all sequences composed of zero or more elements of $D$. We use $\sigma = \langle d_0, d_1, \ldots, d_n \rangle$ to denote a sequence. Furthermore, $|\sigma| = n + 1$ represents the length of the sequence and $d \in \sigma$ equals $\exists 0 \leq i < |\sigma| : \sigma_i = d$, an empty sequence is denoted by $\langle \rangle$ and we use $+$ to concatenate sequences and $<$ to denote sub-sequences, i.e. if $\sigma < \sigma'$ then there exists $\sigma_{pre}, \sigma_{post} \in \mathcal{P}(D^*)$, such that $\sigma' = \sigma_{pre} + \sigma + \sigma_{post}$.

In this thesis, we later introduce process models, which are graph-based. Therefore, we first introduce the concept of a graph.

### 2.1.2 Graph Notations

At the basis of process models, usually lie graphs. Graphs are mathematical structures, consisting of a set of nodes and edges between these nodes. A directed graph is a graph where each edge has a direction, i.e. an edge going from node $a$ to node $b$ is different from an edge going from node $b$ to node $a$.

**Definition 2.1.6. (Graph)**

Let $N$ be a set of nodes and $E \subseteq N \times N$ a set of edges. We say that $G = (N, E)$ is a graph, or more specifically a directed graph.

Since a graph is a collection of nodes, connected by edges, one can “walk” along these edges from one node to the other. Such a sequence of nodes is called a path.
Chapter 2 Preliminaries

Definition 2.1.7. (Path in a graph)
Let $G = (N, E)$ be a graph. Let $a \in N$ and $b \in N$. We define a path from $a$ to $b$ as a sequence of nodes denoted by $\langle n_1, n_2, \ldots, n_k \rangle$ with $k \geq 2$ such that $n_1 = a$ and $n_k = b$ and $\forall i \in \{1, \ldots, k-1\} \left( (n_i, n_{i+1}) \in E \right)$.

Using the concept of a path in a graph, we define whether a graph is connected or not.

Definition 2.1.8. (Connectedness)
A graph $G = (N, E)$ is weakly connected, or simply connected, if and only if, there are no two non-empty sets $N_1, N_2 \subseteq N$ such that $N_1 \cup N_2 = N$, $N_1 \cap N_2 = \emptyset$ and $E \cap ( (N_1 \times N_2) \cup (N_2 \times N_1) ) = \emptyset$. Furthermore, $G$ is strongly connected if for any two nodes $n_1, n_2 \in N$ holds that there is a path from $n_1$ to $n_2$.

Another important concept is the graph coloring. A graph coloring is a way to label the nodes of a graph in such a way that no two neighboring nodes (i.e. nodes connected by an edge) have the same label (i.e. color). A special class of graphs are the so-called bi-partite graphs. These graphs are such that they can be colored with two colors. Figure 2.2 shows two graphs, a directed graph and one bipartite graph with its two partitions (or colors).

Definition 2.1.9. (Graph coloring)
Let $G = (N, E)$ be a graph. Let $\mu$ be a finite set of colors. A function $f : N \to \mu$ is a coloring function if and only if for all $(n_1, n_2) \in E$ holds that $f(n_1) \neq f(n_2)$.

In graphs, we would like to be able to reason about predecessors and successors of nodes. Therefore, we introduce the pre-set and the post-set of a node, which can be seen as the input and the output of a node respectively.

Definition 2.1.10. (Pre-set and post-set)
Let $G = (N, E)$ be a graph and let $n \in N$. We define $\bullet n = \{ m \in N \mid (m, n) \in E \}$ as the pre-set and $n^\bullet = \{ m \in N \mid (n, m) \in E \}$ as the post-set of $n$ with respect to the graph $G$. If the context is clear, the superscript $G$ may be omitted, resulting in $\bullet n$ and $n^\bullet$. 

Figure 2.2: Two example graphs.
2.2 Process Logs

In Section 1.2, we saw that information systems serve different purposes, and that they are used in very different organizations. Therefore, it is obvious that there is a wide variety of event logs provided by such systems. In this thesis, we focus on the event logs that can be generated by process-aware information systems. Since the information in an event log highly depends on the internal data representation of each individual system, it is safe to assume that each system provides information in its own way. Therefore, we need to provide a standard for the information we need for process mining and mappings from each system to this standard.

Before introducing this standard, it is crucial to provide the minimal amount of information that needs to be present in order to do process mining. In this section, we first give some requirements with respect to this information. From these requirements, we derive a meta model in terms of a UML class diagram. Then, we introduce a formal XML definition for event logs, called MXML, to support this meta model. We conclude the section with an example of an MXML file.

2.2.1 Event Log Requirements

All process-aware information systems have one thing in common, namely the process specification. For groupware systems, such a specification is nothing more than a unstructured set of possible activities (which might not even be explicitly known to the system), while for production workflows this specification may be extremely detailed. For process mining, log files of such systems are needed as a starting point. First we give the requirements for the information needed.

When examining event logs, many events may be present more than once. To make the distinction between events, and the logged occurrences of events, we will

<table>
<thead>
<tr>
<th>Process</th>
<th>Case</th>
<th>Activity</th>
<th>Event type</th>
<th>Timestamp</th>
<th>Originator</th>
</tr>
</thead>
<tbody>
<tr>
<td>invoice</td>
<td>invoice</td>
<td>payment</td>
<td>start</td>
<td>10/24/2006</td>
<td>John</td>
</tr>
<tr>
<td>handling</td>
<td>1029</td>
<td></td>
<td></td>
<td>12:00</td>
<td></td>
</tr>
<tr>
<td>invoice</td>
<td>invoice</td>
<td>payment</td>
<td>complete</td>
<td>10/24/2006</td>
<td>Mary</td>
</tr>
<tr>
<td>handling</td>
<td>1039</td>
<td></td>
<td></td>
<td>12:06</td>
<td></td>
</tr>
<tr>
<td>order</td>
<td>order</td>
<td>shipment</td>
<td>assign</td>
<td>10/24/2006</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>processing</td>
<td>2344</td>
<td></td>
<td></td>
<td>12:07</td>
<td></td>
</tr>
<tr>
<td>invoice</td>
<td>invoice</td>
<td>payment</td>
<td>complete</td>
<td>10/24/2006</td>
<td>John</td>
</tr>
<tr>
<td>handling</td>
<td>1029</td>
<td></td>
<td></td>
<td>12:15</td>
<td></td>
</tr>
<tr>
<td>order</td>
<td>order</td>
<td>shipment</td>
<td>start</td>
<td>10/24/2006</td>
<td>Bill</td>
</tr>
<tr>
<td>processing</td>
<td>2344</td>
<td></td>
<td></td>
<td>12:30</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Example of an event log meeting all requirements.
refer to the latter by *audit trail entries* from here on. When events are logged in some information system, we need them to meet the following requirements [25] in order to be useful in the context of process mining:

1. Each audit trail entry should be an event that happened at a given point in time. It 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, as well as finishing the work-item. The process of working on the work-item itself is not.
2. Each audit trail entry should refer to one activity only, and activities should be uniquely identifiable.
3. Each audit trail entry should contain a description of the event type. For example, the activity was started or completed. This transactional information allows us to refer to the different events related to some activity, and we present this in detail in Subsection 2.2.2.
4. Each audit trail entry should refer to a specific process instance (case). We need to know, for example, for which invoice the payment activity was started.
5. Each process instance should belong to a specific process.
6. The events within each case are ordered, for example by timestamps.

Table 2.1 shows an example of a part of an event log fulfilling all requirements, where each row represents one audit trail entry. It shows 5 audit trail entries, relating to 2 processes and 3 process instances. Furthermore, for each audit trail instance, it shows who initiated this event, i.e. the originator, which is not a required attribute, but is often recorded. Using the requirements given above, we are able to make a meta model of the information that should be provided for process mining, i.e. we give the semantics of the different event types.

### 2.2.2 Transactional Model

In order to be able to talk about events recorded in an event log in a standardized way, we developed a transactional model that shows the events that can appear in a log. This model, shown in Figure 2.3, is based on analyzing the different types of logs in real-life systems (e.g., Staffware, SAP, FLOWer, etc.).

Figure 2.3 shows the event types that can occur with respect to an activity and/or a case. When an activity (or Workflow Model Element) is created, it is either “scheduled” or skipped automatically (“autoskip”). Scheduling an activity means that the control over that activity is put into the information system. The information system can now “assign” this activity to a certain person or group of persons. It is possible to “reassign” an assigned activity to another person or group of persons. This can be done by the system, or by a user. A user can (1) “start” working on an activity, (2) decide to “withdraw” the activity or (3) skip the activity manually (“manualskip”), which can even happen before
the activity was assigned. The main difference between a withdrawal and a manual skip is the fact that after the manual skip the activity has been executed correctly, while after a withdrawal it is not. The user that started an activity can “suspend” and “resume” the activity several times, but in the end the activity needs to “complete” or abort (“ate_abort”, where “ate” stands for Audit Trail Entry). Note an activity can get aborted (“pi_abort”, where “pi” stands for Process Instance) during its entire life cycle, if the case to which it belongs is aborted. The semantics described here are presented in Table 2.2.

Using the event types presented in Table 2.2, we can formally define an event log as a collection of traces.

**Definition 2.2.1. (Trace, process log, log event)**
Let $A$ be a set of activities and $E$ a set of event types like “schedule”, “complete” and so on. $\sigma \in (A \times E)^*$ is a trace, or process instance and $W \in \mathcal{P}((A \times E)^*)$ is a process log. For readability, we simply say $W \subseteq A \times E$, and we refer to the elements of $W$ as log events, i.e. unique combinations of an activity and an event type.

In Definition 2.2.1, we define a log as a set of traces. Note that in real life, logs are bags of traces, i.e. the same trace may occur more than once. However, since we often only focus on so-called noise free logs, which we define in Subsection 2.2.5 in this thesis, we will not consider occurrence frequencies of traces and therefore sets suffice.

In the following subsection, we introduce an XML format for storing event logs that include all event types of Table 2.2 by default. However, since we cannot claim that we have captured all possible event types of all systems, the format allows for user defined events.
Table 2.2: Event types and their informal semantics.

<table>
<thead>
<tr>
<th>Event type</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>schedule</td>
<td>An activity has been scheduled to be executed. At this point, it is not assigned to any user.</td>
</tr>
<tr>
<td>assign</td>
<td>The activity has now been assigned to a single user, i.e. that user should start the activity or re-assign it</td>
</tr>
<tr>
<td>re-assign</td>
<td>The activity was assigned to one user, but is now re-assigned to another user. Note that this does not lead to a change in state.</td>
</tr>
<tr>
<td>start</td>
<td>The activity is now started by the user. This implies that no other user can start the same activity any more.</td>
</tr>
<tr>
<td>suspend</td>
<td>If a user decided to stop working on an activity, the activity is suspended for a while, after which it needs to be resumed.</td>
</tr>
<tr>
<td>resume</td>
<td>When an activity was suspended, it has to be resumed again.</td>
</tr>
<tr>
<td>complete</td>
<td>Finally, the activity is completed by the user.</td>
</tr>
<tr>
<td>autoskip</td>
<td>Some information systems allow for an activity to be skipped, event before it is created, i.e. the activity was never available for execution, but is skipped by the system.</td>
</tr>
<tr>
<td>manualskip</td>
<td>In contrast to skipping an activity automatically, a user can skip an activity if it is scheduled for execution, or assigned to that user.</td>
</tr>
<tr>
<td>withdraw</td>
<td>If an activity is scheduled for execution, or assigned to a user, it can be withdrawn, i.e. the system decides that the execution of this activity is no longer necessary.</td>
</tr>
<tr>
<td>ate_abort</td>
<td>Once a user has started the execution of an activity, the system can no longer withdraw it. However, the user can abort the execution, even if it is currently suspended.</td>
</tr>
<tr>
<td>pi_abort</td>
<td>An activity is always executed in the context of a case, or process instance. Therefore, in every state of the activity it is possible that the case is aborted.</td>
</tr>
</tbody>
</table>

2.2.3 MXML Structure

To store the event logs that we defined in this section, we have developed an XML format, called MXML, used by our process mining framework ProM. In Figure 2.4 a schema definition is given for the MXML format.

Most of the elements in the XML schema have been discussed before and are self-explanatory. However, there are two exceptions. First of all, there is the “Data” element, which allows for storing arbitrary textual data, and contains a list of “Attribute” elements. On every level, it can be used to store information about the environment in which the log was created. Second, there is the “Source” element. This element can be used to store information about the information system this log originated from. It can in itself contain a data element, to store information about the information system. It can for example be used to store
configuration settings.

Table 2.3 shows the formalization following Definition 2.2.1, of the event log of Table 2.1. Since that event log contains two processes, i.e. “invoice handling” and “order processing”, we need two process logs to express that event log. Note that we abstract from most of the information contained in that log, i.e. the timestamps and originators, but we keep the ordering within each process instance.

Table 2.3: The log of Table 2.1 formalized as two processes.

<table>
<thead>
<tr>
<th>Process</th>
<th>Process Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>invoice handling</td>
<td>{ \ldots, (payment, start), (payment, complete), \ldots, }</td>
</tr>
<tr>
<td></td>
<td>{ \ldots, (payment, complete), \ldots }</td>
</tr>
<tr>
<td>order processing</td>
<td>{ \ldots, (shipment, assign), (shipment, start), \ldots }</td>
</tr>
</tbody>
</table>

Table 2.4 shows the event log of Table 2.1 in the MXML format. In this thesis, event logs, such as the one in Table 2.4, form the starting point for several process mining algorithms. However, for these algorithms, we typically consider only one process, in which case we refer to the log as a *process log*.

So far, we formalized the concept of an event log, as well as the semantics thereof and we presented three ways to represent such logs, i.e. using tables, MXML and a more abstract formal notation.
Table 2.4: MXML representation of the log in Table 2.1.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<WorkflowLog xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="WorkflowLog.xsd"
description="Log of residence permit application model">
  <Source program="process-aware Information System"></Source>
  <Process id="0" description="invoice handling">
    <ProcessInstance id="invoice1029" description="Handling of invoice 1029">
      ...
      <AuditTrailEntry>
        <WorkflowModelElement>payment</WorkflowModelElement>
        <EventType>start</EventType>
        <Timestamp>2006-10-24T12:00:00.000+01:00</Timestamp>
        <Originator>John</Originator>
      </AuditTrailEntry>
      <AuditTrailEntry>
        <WorkflowModelElement>payment</WorkflowModelElement>
        <EventType>complete</EventType>
        <Timestamp>2006-10-24T12:15:00.000+01:00</Timestamp>
        <Originator>John</Originator>
      </AuditTrailEntry>
      ...
    </ProcessInstance>
    <ProcessInstance id="invoice1039" description="Handling of invoice 1039">
      ...
    </ProcessInstance>
  </Process>
  <Process id="1" description="order processing">
    <ProcessInstance id="order 2344" description="Processing order 2344">
      ...
      <AuditTrailEntry>
        <WorkflowModelElement>shipment</WorkflowModelElement>
        <EventType>assign</EventType>
        <Timestamp>2006-10-24T12:07:00.000+01:00</Timestamp>
        <Originator>SYSTEM</Originator>
      </AuditTrailEntry>
      <AuditTrailEntry>
        <WorkflowModelElement>shipment</WorkflowModelElement>
        <EventType>start</EventType>
        <Timestamp>2006-10-24T12:30:00.000+01:00</Timestamp>
        <Originator>Bill</Originator>
      </AuditTrailEntry>
      ...
    </ProcessInstance>
    ...
  </Process>
</WorkflowLog>
```
2.2.4 Log Filtering

The next step for many applications is to filter the log, i.e. to transform the log into another process log. Filtering a log is nothing more than adding information to, or removing information from the process log. Recall for example the transactional model we presented in Figure 2.3. A first step in process mining often is to remove all event types except the “complete” events, i.e. the events referring to the successful completion of an activity. The reason for this is simple, i.e. by removing all other events and even discarding cases that were aborted, we obtain a process log that is easier to analyse or mine and less sensitive to noise (see Subsection 2.2.5 for the definition of noise).

A log filter therefore can considered to be a function that transforms one process instance into another process instance, by adding or removing information.

**Definition 2.2.2. (Log filter)**

Let $A$ be a set of activities and $E$ be a set of event types, such that $T \subseteq (A \times E)$ is a set of log events and $W \in \mathcal{P}(T^*)$ a process log over $T$. Furthermore, let $A'$ also be a set of activities and $E'$ also be a set of event types, such that $T' \subseteq (A' \times E')$ is a set of log events. We define a log filter $f$ as a function $f : \mathcal{P}(T^*) \rightarrow \mathcal{P}(T'^*)$, i.e. a function that takes a trace as input and produces a different trace as output, as long as the requirements of Section 2.2.1 are met. We call $f(W)$ a filtered log.

Note that if $W$ is a process log, then $f(W)$ is again a process log, hence filters can be applied one after another and the result remains a process log.

Filtering is an important step when using process mining techniques on process logs. In Chapter 6 for example, we introduce a process mining algorithm that requires the first log event of all process instances to be the same log event. It is easy to see that a log filter could be used to insert such a log event in all process instances. Table 2.5 shows a selection of common filters, their parameters and their results, when applied to a single trace $\sigma$. Note that each log filter requires some parameters.

<table>
<thead>
<tr>
<th>Name (f)</th>
<th>Parameters</th>
<th>Activities $(A')$</th>
<th>Event Types $(E')$</th>
<th>Log Events $(T')$</th>
<th>Result $(f(\sigma))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add initial event filter</td>
<td>$a \notin A$</td>
<td>$A \cup {a}$</td>
<td>$E \cup {\text{complete}}$</td>
<td>$T \cup {(a, \text{complete})}$</td>
<td>$\langle (a, \text{complete}) \rangle + \sigma$</td>
</tr>
<tr>
<td>Add final event filter</td>
<td>$a \notin A$</td>
<td>$A \cup {a}$</td>
<td>$E \cup {\text{complete}}$</td>
<td>$T \cup {(a, \text{complete})}$</td>
<td>$\sigma + \langle (a, \text{complete}) \rangle$</td>
</tr>
<tr>
<td>Initial events filter</td>
<td>$I \subseteq T$</td>
<td>$A$</td>
<td>$E \cup {\text{complete}}$</td>
<td>$T$</td>
<td>if $\sigma_0 \in I$ then $\sigma$ else $\langle \rangle$</td>
</tr>
<tr>
<td>Final events filter</td>
<td>$F \subseteq T$</td>
<td>$A$</td>
<td>$E$</td>
<td>$T$</td>
<td>if $\sigma_{</td>
</tr>
</tbody>
</table>

Note that we assume that $A$, $E$, $T$ and $W$ are known, and that $\sigma \in W$. 

Table 2.5: Four examples of log filters.
As we mentioned in Section 3.4, different process mining techniques have different requirements with respect to the log content. Therefore, in the next subsections, we classify logs into several categories, where we note that a log that falls into one category could become part of another by filtering it the right way.

2.2.5 Classification of Logs

In order to be able to define process mining techniques, we need to classify logs. For our purpose of control flow discovery, we need to consider two dimensions. First, we need to be able to say something about the completeness of information in the log and second we need to be able to say whether the log is accurate or not, i.e. whether it contains noise.

To exemplify our classification, we consider a travel agency process. In this process, a trip needs to be booked that always includes a hotel reservation and a rental car reservation. Furthermore, it either includes a bus ticket or a plane ticket to be booked. Booking the hotel, rental car, plane ticket or bus ticket can be done in any order, after which an invoice is created.

Noisy Logs

Consider an information system that logs its transactions. Assuming that the logging module does not change timestamps by itself, the transactions will appear in the right order in the log, and hence the log should be accurate. However in the context of process mining, noise can also mean that a log contains exceptional behaviour. For example, if in a log, activity a was performed by person b at time c then that is what happened. However, it might be that some of these events do not belong to the normal behaviour of the process. For example if a person makes a mistake while entering data in a financial system and the system administrator needs to be called to “undo” that mistake. One could argue that that is exceptional behaviour and should therefore be considered as noise. In this thesis, we will not deal with such kind of logs. For a more elaborate discussion on noise and how it can be dealt with, we refer to Chapter 6 of [128], where all low-frequent behaviour is considered noise.

In our example, a noisy log could show that a plane ticket was reserved, after which the customer decided to go by bus anyway and therefore, a bus ticket was reserved as well. Since that does not correspond to the normal (i.e. frequent) behaviour, we consider it to be noise.

Just like noise, completeness is far from straightforward. A log always shows recorded behaviour. Without explicit knowledge of the underlying process, there is no way to say whether or not a log shows all behaviour needed to draw the right conclusions. Nonetheless, the assumption that a log is complete will be necessary to prove the correctness and effectiveness of our algorithms. We consider globally
complete and \textit{locally} complete logs, as will be explained in the remainder of this subsection.

\section*{Globally Complete Logs}

When a process log is said to be \textit{globally complete}, we mean that the log contains all the possible behaviour of the underlying system, i.e. it shows all possible executions of a process.

\textbf{Definition 2.2.3. (Globally complete log)}

Let $T$ be a set of log events and $L \in \mathcal{P}(T^*)$ be the set of all possible traces of some model or process. Furthermore, let $W \in \mathcal{P}(T^*)$ be a process log over $T$. We say that $W$ is a globally complete log if and only if $W = L$.

A globally complete log for our travel agency example contains the 12 possible traces, such as shown in Table 2.6. Note that it is unlikely for this log to occur in practice.

This type of log is very interesting from a theoretical point of view, since all behaviour is known. In fact in the work on the Theory of Regions, which we discuss in Section 3.4 in more detail, it has been shown that from a log like this, a model describing the underlying process exactly can always be discovered. However, logs like this are rarely seen in practice for a very simple reason. Consider a process where 10 tasks have to be performed in parallel. This leads to 3628800 possible execution orders (i.e. 10!), for just these 10 tasks. Therefore, we need to consider the fact that sometimes logs do not show the globally complete behaviour of a system.
Table 2.7: Example of a locally complete log of a travel booking process.

<table>
<thead>
<tr>
<th>Process Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Book Hotel, Book Car, Book Plane, Create Invoice),</td>
</tr>
<tr>
<td>(Book Car, Book Hotel, Book Plane, Create Invoice),</td>
</tr>
<tr>
<td>(Book Car, Book Hotel, Book Bus, Create Invoice),</td>
</tr>
<tr>
<td>(Book Plane, Book Hotel, Book Car, Create Invoice),</td>
</tr>
<tr>
<td>(Book Hotel, Book Bus, Book Car, Create Invoice),</td>
</tr>
<tr>
<td>(Book Car, Book Bus, Book Hotel, Create Invoice),</td>
</tr>
<tr>
<td>(Book Car, Book Plane, Book Hotel, Create Invoice)</td>
</tr>
</tbody>
</table>

Locally Complete Logs

If a log is not globally complete, it can still be *locally complete* with respect to a certain algorithm. By that we mean that the log does not contain all the possible behaviour of the underlying system, but for that specific algorithm, it contains enough information. For example the \( \alpha \)-algorithm, which we present in detail in Section 6.4, requires a log such that if a log event can follow another log event in the process then this should be recorded at least once.

**Definition 2.2.4. (Locally complete log)**

Let \( T \) be a set of log events and \( L \subseteq \mathcal{P}(T^*) \) be the set of all possible traces of some model or process. Furthermore, let \( W \subseteq \mathcal{P}(T^*) \) be a process log over \( T \). We say that \( W \) is a locally complete log if and only if for all two events \( t_1, t_2 \in T \) holds that if there exists a \( \sigma \in L \) with \( \langle t_1, t_2 \rangle \triangleleft \sigma \) then there exists a \( \sigma' \in W \) with \( \langle t_1, t_2 \rangle \triangleleft \sigma' \).

A locally complete log for our travel agency example does not necessarily contain all 12 traces of Table 2.6\(^1\). Instead, a log containing the 7 traces of Table 2.7 is locally complete.

Again, a locally complete log is interesting from a theoretical point of view, since the knowledge about predecessors and successors of an activity will prove to be enough to come up with a good description of the underlying process. Furthermore, logs like this are more likely to be seen in practice, as less cases are necessary to get locally complete information.

The definition of a locally complete log seems simple, but hides a rather complex problem. Consider a process that is being executed in a company. Over time, hundreds, or even thousands of cases have been executed and logged. Now, we look at the process log. Is there anyone who can tell us if the log is locally complete or not? In fact, the answer is probably no. Therefore, in this thesis, we will always have to *assume* that the log is locally complete. Under this assumption, a process mining algorithm can be applied and the resulting model

\(^1\)Note that each globally complete log is also locally complete, but there are many different locally complete logs.
Table 2.8: Example of an incomplete log of a travel booking process.

<table>
<thead>
<tr>
<th>Process Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ Book Hotel, Book Car, Book Plane, Create Invoice },</td>
</tr>
<tr>
<td>{ Book Hotel, Book Car, Book Bus, Create Invoice },</td>
</tr>
<tr>
<td>{ Book Hotel, Book Plane, Book Car, Create Invoice },</td>
</tr>
<tr>
<td>{ Book Hotel, Book Bus, Book Car, Create Invoice }</td>
</tr>
</tbody>
</table>

should be checked against the log to see whether it is capable of reproducing it (i.e. conformance testing).

However, using the same process mining algorithms one could predict whether a log is locally complete or not. Consider a process log, which you divide into $k$ partitions. Then, if you apply a mining algorithm which requires a locally complete log as input onto $k - 1$ partitions at the time and for all $k$ combinations the result is the same process model, it is safe to assume that the log is locally complete. This procedure is well-known in machine learning [137] and it is usually referred to as $k$-fold checking.

Incomplete Logs

Obviously, the assumption that a log is complete, either local or global, is not a nice one. Without knowing the process by which the log was generated, the question whether or not a log is complete can in fact not be answered. Therefore, we have to define incomplete logs. Incomplete logs are logs that do contain a number of process instances, but of which we do not know if these instances form a complete log. Note that this is subtly different from saying that they are not complete.

As an example, consider the process log of Table 2.8, where we again show a log of the travel booking process. However, this log shows that the travel agent always first books a hotel and only then a rental car, bus ticket or plane ticket. Obviously, this is an incomplete log of our process. However, if we would not have the process definition, we would not know that it is incomplete. In fact, if we assume that the hotel booking has to be performed first, then this log would be globally complete for that process.

In many of the algorithms we present in this thesis, we have to assume that the log we start from is complete. However, we will always give at least some insight into the behaviour of the algorithm with respect to incomplete logs, i.e. we give some insight into the robustness of the algorithms.

In this section, we introduced a standard for storing event logs generated by process-aware information systems. For this, we provide requirements, a data model and an XML format called MXML. Furthermore, we provided some insights into the completion of logs and noise. In the remainder of this thesis, we will assume the process logs to be noise-free, unless otherwise indicated. The
practical applicability of MXML will become apparent in Chapter 8, where we introduce a tooling framework that is based on the MXML format.

2.3 Petri nets

Petri nets are a formal language that can be used to specify processes. Since the language has a formal and executable semantics, processes modelled in terms of a Petri net can be executed by an information system. For an elaborate introduction to Petri nets, we refer to [67, 138, 147]. For sake of completeness, we mention that the Petri nets we use in this thesis correspond to a well-known class of Petri nets, namely Place/Transition nets.

2.3.1 Concepts

A Petri net consists of two modeling elements, namely places and transitions. These elements are the nodes of a bipartite graph, partitioned into places and transitions. When a Petri net is represented visually, we draw transitions as boxes and places as circles. Figure 2.5 shows an example of a marked P/T-net, containing 11 transitions, i.e. $T = \{A, B, C, D, E, F, G, H, I, J, K\}$ and 10 places, of which we typically do not show the labels, since they do not correspond to active elements of a P/T-net. The roles of places and transitions in the description of the process are defined as follows:

**Transitions**, which typically correspond to either an activity which needs to be executed, or to a “silent” step that takes care of routing.

**Places**, which are used to define the preconditions and postconditions of transitions.

Transitions and places are connected through directed arcs in such a way that the places and transitions make up the partitions in a bipartite graph (no place is connected to a place and no transition is connected to a transition). The places that are in the pre-set of a transition are called its input places and the places in the post-set of a transition its output places.

To denote the state of a process execution the concept of tokens is used. In Figure 2.5, three places contain a token denoted by the black dot. A token is placed inside a place to show that a certain condition holds. Each place can contain arbitrarily many of such tokens. When a transition execution occurs (in other words, a transition fires), one token is removed from each of the input places and one token is produced for each of the output places. Note that this restricts the behaviour in such a way that a transition can only occur when there is at least one token in each of the input places. The distribution of tokens over the places is called a state, better known as marking in Petri net jargon. Formally, a P/T net with some initial marking is defined as follows.
Definition 2.3.1. (Place/Transition net)
\( \varphi = (P, T, F) \) is a Place/Transition net (or P/T-net) if:
- \( P \) is a finite, non-empty set of places,
- \( T \) is a finite, non-empty set of transitions, such that \( P \cap T = \emptyset \),
- \( F \subseteq (P \times T) \cup (T \times P) \) is the flow relation of the net,

Definition 2.3.2. (Marking, Marked P/T-net)
A marking \( M \) is a bag over the set of places \( P \), i.e. \( M : P \rightarrow \mathbb{N} \) and a marked P/T-net is a pair \( (\varphi, M_0) \), where \( \varphi = (P, T, F) \) is a P/T-net and where \( M_0 \) is the marking of the net. The set of all marked P/T-nets is denoted \( \mathcal{N} \).

In this thesis, we restrict ourselves to P/T-nets where for all transitions \( t \) holds that \( \bullet t \neq \emptyset \) and \( t \bullet \neq \emptyset \) and for all places \( p \) holds that \( \bullet p \cup p \bullet \neq \emptyset \). In other words, all places have at least one incident edge and all transitions have at least one incoming and one outgoing edge.

As we stated before, Petri nets are used to describe processes and therefore, to describe dynamic behaviour. So-far, we have only defined the static structure of a Petri net. Therefore, we now define the dynamics. The dynamics of a Petri net are defined using the concepts of a marking and a firing rule. However, first we state when a transition is enabled.

Definition 2.3.3. (Enabled transition)
Let \( \varphi = ((P, T, F), M_0) \) be a marked P/T-net. Transition \( t \in T \) is enabled, denoted \( (\varphi, M_0)[t] \), if and only if \( \bullet t \leq M_0 \).

In other words, a transition is enabled if each of its input places contains at least one token. In Figure 2.5 for example, the transitions \( F, G, H \) and \( I \) are all enabled. If a transition is enabled, it can fire. Transitions fire one by one using the following firing rule.
Definition 2.3.4. (Firing rule)
Let $\varphi = ((P, T, F), M_0)$ be a marked P/T-net. The firing rule $\rightarrow \subseteq N \times T \times N$ is the smallest relation satisfying for any $((P, T, F), M_0) \in N$ and any $t \in T$, $(\varphi, M_0)[t] \Rightarrow (\varphi, M_0)[t] (\varphi, (M_0 - \bullet t) \uplus \bullet t)$.

The firing rule says that if a transition is enabled then it can fire and when it does, it removes exactly one token from each of its input places and adds exactly one token to each of its output places. If in Figure 2.5 transition $G$ would fire, then the input place of $G$ contains no tokens after the firing and the output place of $G$ contains one token after the firing.

When considering a marked P/T-net with some enabled transition, firing that transition leads to another marking in which other transitions can be enabled again. Therefore, transitions can be fired in sequences. Such sequence of transition firings is what we call a firing sequence.

Definition 2.3.5. (Firing sequence)
Let $\varphi = ((P, T, F), M_0)$, be a marked P/T-net. A sequence $\langle t_1, \ldots, t_n \rangle \in T^*$ is called a firing sequence of $\varphi, M_0)$, if and only if, there exist markings $M_1, \ldots, M_n$, such that for all $i$ with $0 \leq i < n$, $(\varphi, M_i)[t_{i+1}] (\varphi, M_{i+1})$. Note that for $n = 0$, $\langle \rangle$ is a firing sequence of $\varphi, M_0)$. A Sequence $\sigma$ is said to be enabled in marking $M_0$, denoted $(\varphi, M_0)[\sigma]$. Firing the sequence $\sigma$ results in a marking $M_n$, denoted $(\varphi, M_0)[\sigma] (\varphi, M_n)$.

The distribution of tokens over places is what we call a marking. Since the firing rule defines how one marking can be transformed into another marking, we can define a set of reachable markings.

Definition 2.3.6. (Reachable markings)
Let $(\varphi, M_0)$ be a marked P/T-net in $N$. A marking $M$ is reachable from the initial marking $M_0$ if and only if there exists a firing sequence $\sigma$, such that $(\varphi, M_0)[\sigma] (\varphi, M)$. The set of reachable markings of $(\varphi, M_0)$ is denoted $[\varphi, M_0] = \{ M \mid \exists_{\sigma \in T^*}(\varphi, M_0)[\sigma] (\varphi, M)\}$.

Using the dynamics of a Petri net, we can define some properties of Petri nets, which will play an important role in this thesis. First, we introduce the notion of boundedness and safeness for those Petri nets where the maximum number of tokens in each place is limited. Figure 2.6 shows four marked P/T-nets, of which (b) is unbounded, i.e. the middle place can accumulate tokens. All other nets are bounded, i.e. (a) is bounded (since the maximum number of tokens in any place is 2) and (c) and (d) are safe, i.e. the maximum number of tokens in each place is one.

Definition 2.3.7. (Boundedness, Safeness)
A marked net $\varphi = ((P, T, F), M_0)$ is bounded if and only if the set of reachable markings $[\varphi, M_0]$ is finite. It is safe, if and only if, for any $M \in [\varphi, M_0]$ and any $p \in P$, $M(p) \leq 1$. Note that safeness implies boundedness.
The firing rule of a Petri net only states that a transition can fire if there are enough tokens present. However, what we are also interested in is whether transitions can become enabled or not. If a transition can not be enabled, it is called dead. Furthermore, if all transitions in a net can become enabled over and over again, the net is called live. In Figure 2.6, P/T-net (c) contains a dead transition $C$ and nets (b) and (d) are live. Net (a) is an example that does not contain dead transitions, but is not live.

**Definition 2.3.8. (Dead transitions, liveness)**

Let $(\varnothing = (P, T, F), M_0)$ be a marked P/T-net. A transition $t \in T$ is dead in $(\varnothing, M_0)$ if and only if there is no reachable marking $M \in [\varnothing, M_0]$ such that $(\varnothing, M)[t]$. $(\varnothing, M_0)$ is live if and only if, for every reachable marking $M \in [\varnothing, M_0]$ every $t \in T$ is not dead in $(\varnothing, M)$.

Note that liveness implies the absence of dead transitions, but not the other way around.

Finally, since we use Petri nets to describe processes, we define the notion of an implicit place, which is a place, that can be removed from the Petri net, without changing the behaviour at all. In Figure 2.6, all nets contain implicit places, since all output places are implicit. However, net (d) contains two implicit places which are not output places, i.e. either one of the places between $A$ and $B$ can be removed (they are both implicit), but they cannot both be removed at the same time.

**Definition 2.3.9. (Implicit place)**

Let $\varnothing = (P, T, F)$ be a P/T-net with initial marking $M_0$. A place $p \in P$ is called implicit in $(\varnothing, M_0)$ if and only if, for all reachable markings $M \in [\varnothing, M_0]$ and transitions $t \in p \bullet, M \geq t \setminus \{p\} \Rightarrow M \geq t$.

Figure 2.6: Examples of marked P/T-nets.
Petri nets are a nice formalism to describe dynamic systems. However, we are mainly interested in describing workflow processes, which have some special properties. Therefore, we introduce a sub-class of Petri nets, namely workflow nets.

### 2.3.2 Workflow nets

Workflow nets, or WF-nets, are a subclass of Petri nets, tailored towards workflow modeling and analysis. For example in [175], several analysis techniques are presented towards soundness verification of WF-nets.

Basically, WF-nets correspond to P/T-nets with some structural properties. These structural properties are such that for each case that is executed, we can clearly see the initial and final state. Furthermore, they ensure that there are no unreachable parts of the model, i.e.:

- The initial marking marks exactly one place (the initial place or source place), and this place is the only place without incoming arcs,
- There is exactly one final place or sink place, i.e. a place without outgoing arcs,
- Each place or transition is on a path that starts in the initial place and ends in the final place.

Formally, we define WF-nets as follows:

**Definition 2.3.10. (Workflow net)**

A P/T-net $\varphi = (P, T, F)$ is a workflow net (or WF-net) if and only if:

- There exists exactly one $p_i \in P$, such that $p_i = \emptyset$, i.e. the source place,
- There exists exactly one $p_f \in P$, such that $p_f = \emptyset$, i.e. the sink place,
- Each place and transition is on a path from $p_i$ to $p_f$.

Although workflow nets provide a nice theoretical basis for the work presented in this thesis, their application is not restricted to theory. In fact, many tools, such as YAWL [1], Staffware [172] or Protos [140] use workflow nets, or Petri nets as a basis, even though they typically extend the concept to make it more expressive/applicable, while retaining some of the nice theoretical properties.

### 2.4 Event-driven Process Chains

To describe process models in an informal language, closer to practically used modelling languages, we introduce the concept of *Event-driven Process Chains* (EPCs) [113, 114, 162] as a modelling language and consider some properties of these EPCs. This choice is motivated by the widespread use of EPCs. For example, EPCs are the main language of the ARIS Toolset, which is an industry
standard enterprise modelling toolset. Furthermore, the Aris for MySAP reference model is modelled in the EPC language and is widely used to configure the ERP system SAP (i.e. SAP R/3).

2.4.1 Concepts

The aim of EPCs is to provide an intuitive modeling language to model business processes. They were introduced by Keller, Nüttgens and Scheer in 1992 [113]. It is important to realize that the language is not intended to make a formal specification of a business process, but to make a conceptual sketch of the process under consideration. An EPC consists of three main elements. Combined, these elements define the flow of a business process as a chain of events. The elements used are:

Functions, which are the basic building blocks. A function corresponds to an activity (task, process step) which needs to be executed.

Events, which describe the situation before and/or after a function is executed. Functions are linked by events. An event may correspond to the postcondition of one function and act as a precondition of another function.

Connectors, which can be used to connect functions and events. This way, the flow of control is specified. There are three types of connectors: \( \land \) (and), \( \times \) (xor) and \( \lor \) (or).

Functions, events and connectors can be connected with edges in such a way that (1) events have at most one incoming edge and at most one outgoing edge, but at least one incident edge (i.e. an incoming or an outgoing edge), (2) functions have precisely one incoming edge and precisely one outgoing edge, (3) connectors have either one incoming edge and multiple outgoing edges, or multiple incoming edges and one outgoing edge, and (4) in every path, functions and events alternate (no two functions are connected and no two events are connected, not even when there are connectors in between). Furthermore, a more relaxed requirement of EPCs (i.e. a modelling guideline) is that an event should never be followed by a choice connector. The latter requirement relates to the implementation where all components of an information system that can handle a certain event, should handle it and no selection is made between those components.

The idea behind Event-driven Process Chains (EPCs) is to provide an intuitive modeling language to model business processes. Figure 2.7 shows an example of an EPC. The 9 functions are drawn as rectangles with rounded corners, the 19 events as hexagons and the connectors as circles containing their type, i.e. \( \land, \times, \lor \).

Next to the formal definition of EPCs given below in Definition 2.4.1, other restrictions on the language can be given as well. For example, one can enforce to avoid loops consisting of only connectors, or nodes that cannot be reached from an initial event.
Figure 2.7: Example of an EPC with 9 functions, 19 events and 9 connectors, describing the EPC verification process described in Section 5.3.
**Definition 2.4.1. (Event-driven Process Chain)**

\( \varepsilon = (F, E, C_{\text{and}}, C_{\text{xor}}, C_{\text{or}}, A) \) is an EPC if and only if:

- \( F \) is a finite set of functions,
- \( E \) is a finite set of events,
- \( C = C_{\text{and}} \cup C_{\text{xor}} \cup C_{\text{or}} \) is a finite set of connectors,
- All sets \( F, E, C_{\text{and}}, C_{\text{xor}}, C_{\text{or}} \) are pairwise disjoint,
- \( A \subseteq (F \cup E \cup C) \times (F \cup E \cup C) \) is the flow relation of the net, such that for the graph \( G = (F \cup E \cup C, A) \) holds that:
  - for all \( f \in F \) holds that \( |f^G| = |f^G| = 1 \),
  - for all \( e \in E \) holds that \( |e^G| \leq 1 \) and \( |e^G| \leq 1 \),
  - for all \( c \in C \) holds that either \( |c^G| = 1 \) and \( |c^G| > 1 \) or \( |c^G| > 1 \) and \( |c^G| = 1 \),
  - for all \( f_1, f_2 \in F \) holds that there is no \( \{c_1, \ldots, c_n\} \subseteq C \), such that \( \{(f_1, c_1), (c_1, c_2), \ldots, (c_{n-1}, c_n), (c_n, f_2)\} \subseteq A \),
  - for all \( e_1, e_2 \in E \) holds that there is no \( \{c_1, \ldots, c_n\} \subseteq C \), such that \( \{(e_1, c_1), (c_1, c_2), \ldots, (c_{n-1}, c_n), (c_n, e_2)\} \subseteq A \).

The EPC language was originally not intended as a formal specification language. Therefore, finding a clear executable semantics for EPCs has lead to extensive discussions and many different interpretations of EPCs. In this thesis, we will take a pragmatic approach, and use the concept of relaxed soundness as defined in [63] as an executable semantics. The idea behind this is that the possible executions of the EPC is limited by those executions that can eventually reach a desired final outcome, i.e. if a deadlock situation can be avoided in the future execution, then it will be avoided.

### 2.5 The Process Mining Framework ProM

In this thesis, we present several analysis algorithms for EPCs and Petri nets, as well as mining algorithms to discover these models from event logs. To prove that our approaches work, we have developed a tool called the **ProM framework**. The original idea behind the tool was to enable researchers to benefit from each other by providing a common, plug-able environment for process mining research. However, over time the framework has evolved into something more than just a mining framework. Currently, the framework can import and export Petri nets, EPCs, and other models in over 20 different formats. Furthermore, it contains more than 30 analysis plug-ins that analyse properties of these models or logs, more than 25 mining plug-ins that implement specific process mining algorithms and more than 20 conversion plug-ins to convert one model into the other. Figure 2.8 shows a screenshot of ProM, where a log has been opened. Furthermore,
Figure 2.8: Overview of ProM with (a) a log, (b) its summary, (c) a Petri net, and (d) an EPC.
it shows a summary of that log and the results of process mining in terms of an EPC and a WF-net. A more elaborate overview of ProM will be presented in Chapter 8. However, we introduce it here because we frequently refer to it.

2.6 Running Example

Throughout the thesis, we will use an example of a process to explain our work and to show how our algorithms can be used in practice. Our running example considers the process of getting a Dutch residence permit for foreigners who want to come to work at Eindhoven University of Technology. A simplified, though accurate, version of the internal process of the Dutch immigration service is shown as an EPC in Figure 2.9, as well as on the cover of this thesis. The idea is that the university starts the process by applying for an “MVV advice” after which the process is initiated by the event “Advice application entered”. MVV stands for “Machtiging Voorlopig Verblijf”, which is Dutch for a temporary visa the foreigner needs in order to come to the Netherlands the first time. The immigration service now decides on the MVV and the result is either negative, followed by the sending of a rejection letter, or a positive advice, after which the foreigner needs to apply for an MVV himself or herself.

At this point, the foreigner applies for an MVV (initial event “Application for MVV entered” occurs) and the process continues by making a decision on that application (function “Decide on MVV”). Again, the result is either positive or negative and if the MVV is denied, a letter is sent to the foreigner (function “Send MVV rejection letter”). However, if the MVV is granted, an acceptance letter is sent (function “Send MVV acceptance letter”) and at the same time, all relevant documents are sent to the Town Hall of Eindhoven (function “Send documents to Town Hall”).

Once the foreigner has arrived in the Netherlands, he or she goes to the Town Hall and all documents are checked against their conditions (i.e. is there really a contract with the university, etc.). If all conditions are met, the residence permit is granted and sent to the applicants address in the Netherlands. However, if these conditions are not met, the residence permit is denied and a rejection letter is sent. Then, two things can happen. Either the foreigner leaves the country within 28 days, or an objection against the decision is entered (event “Objection entered” occurs). In the latter case, the objection is evaluated (function “Evaluate objection”) and again the residence permit can be granted or denied.

Finally, we note that it is possible to apply for an MVV without having the university ask for an advice, i.e. the initial event “Advice application entered” might never occur, unlike the initial event “Application for MVV entered”.

Besides the model presented in Figure 2.9, we also created a process log following this process, which we use extensively in the chapter on process discovery. An example of the contents of that log can be found in Table 2.9, which shows six
Figure 2.9: EPC describing the process of getting a Dutch residence permit.
audit trail entries, two of which contain additional data, next to the obligatory information of workflow model element, eventtype and timestamp. The data elements in the log can be used by many algorithms, however in this thesis we do not use them.
Table 2.9: Part of an MXML log file of our running example.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<WorkflowLog xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="WorkflowLog.xsd"
    description="Log of residence permit application model">
    <Source program="CPN Tools"></Source>
    <Process id="0" description="Permit Application">
        <ProcessInstance id="MVV00001" description="MVV application 00001">
            <AuditTrailEntry>
                <WorkflowModelElement>Decide on MVV advice</WorkflowModelElement>
                <EventType>start</EventType>
                <Timestamp>2002-04-08T10:55:00.000+01:00</Timestamp>
                <Originator>John</Originator>
            </AuditTrailEntry>
            <AuditTrailEntry>
                <Data>
                    <Attribute name="result">accepted</Attribute>
                </Data>
                <WorkflowModelElement>Decide on MVV advice</WorkflowModelElement>
                <EventType>complete</EventType>
                <Timestamp>2002-04-08T11:55:00.000+01:00</Timestamp>
                <Originator>John</Originator>
            </AuditTrailEntry>
            <AuditTrailEntry>
                <WorkflowModelElement>Decide on MVV</WorkflowModelElement>
                <EventType>start</EventType>
                <Timestamp>2002-04-09T10:55:00.000+01:00</Timestamp>
                <Originator>John</Originator>
            </AuditTrailEntry>
            <AuditTrailEntry>
                <Data>
                    <Attribute name="result">denied</Attribute>
                </Data>
                <WorkflowModelElement>Decide on MVV</WorkflowModelElement>
                <EventType>complete</EventType>
                <Timestamp>2002-04-09T12:55:00.000+01:00</Timestamp>
                <Originator>John</Originator>
            </AuditTrailEntry>
            <AuditTrailEntry>
                <WorkflowModelElement>Send MVV rejection letter</WorkflowModelElement>
                <EventType>start</EventType>
                <Timestamp>2002-04-09T13:55:00.000+01:00</Timestamp>
                <Originator>John</Originator>
            </AuditTrailEntry>
            <AuditTrailEntry>
                <WorkflowModelElement>Send MVV rejection letter</WorkflowModelElement>
                <EventType>complete</EventType>
                <Timestamp>2002-04-09T14:00.000+01:00</Timestamp>
                <Originator>John</Originator>
            </AuditTrailEntry>
        </ProcessInstance>
    </Process>
</WorkflowLog>
```
Chapter 3

Related Work

This thesis presents several algorithms for the verification and mining of processes. Since the work presented in this thesis builds on prior work in different areas, such as process mining and verification, we present this related work in a separate chapter, using the terminology introduced in Chapter 1.

In Figure 3.1, we show the overview of the work on process mining and verification using the framework presented in Section 1.3. This figure shows that process mining and verification can be split into four categories, i.e. log-based verification, process model verification, conformance checking and process discovery. Furthermore, in Chapter 1, we presented three perspectives from which techniques in these categories can be applied, i.e. the process perspective, the case perspective and the organizational perspective.

In this chapter, we present related work in all categories in separate sections, i.e. log-based verification in Section 3.1, process model verification in Section 3.2, process discovery in sections 3.4 and 3.5 and conformance checking in Section 3.3. As stated in Chapter 1 however, this thesis focuses on the process perspective and the case perspective. Therefore, we do not extensively discuss the related work on process mining and verification from an organizational perspective. Instead, we refer to [24] and references there.

3.1 Log-based Verification

In Chapter 4, we introduce a language to verify properties on an event log, i.e. by checking the value of a property written in Linear Temporal Logic, or LTL.

Monitoring events with the goal to verify certain properties has been investigated in several domains, e.g., in the context of requirements engineering [85, 148, 149] and program monitoring [88, 101, 102]. The work of Robinson [148, 149] on requirements engineering is related, as he suggests the use of LTL for the verification of properties. Important differences between his approach and ours are the focus on real-time monitoring (with model-checking capabilities
to warn for future problems) and the coding required to check the desired properties. The following quote taken from [148] illustrates the focus of his work:

“Execution monitoring of requirements is a technique that tracks the run-time behaviour of a system and notes when it deviates from its design-time specification. Requirements monitoring is useful when it is too difficult (e.g., intractable) to prove system properties. To aid analysis, assumptions are made as part of the requirements definition activity. The requirements and assumptions are monitored at run-time. Should any such conditions fail, a procedure can be invoked (e.g., notification to the designer).”

In a technical sense, the work of Havelund et al. [101, 102] is highly related. Havelund et al. propose three ways to evaluate LTL formulas: (1) automata-based, (2) using rewriting (based on Maude), (3) and using dynamic programming. We use the latter approach, i.e. dynamic programming.

More recent work of Van der Aalst and Pesic [20,141] shows that LTL formulas can not only be used for the verification of properties on process logs, but also for the execution of business processes. In their approach, a process is specified by giving desired and undesired properties, in terms of LTL formulas. The system then lets its users perform any task, as long as the undesired properties are not violated, while at the same time it makes sure that a case is only completed if all desirable properties are fulfilled.

For a more elaborate description of the LTL language and LTL checker we define in Chapter 4, we refer to the manual of the LTL Checker [41], which can be found on our website www.processmining.org.

![Figure 3.1: Process Mining and Verification.](image-url)
### 3.2 Process Model Verification

In Chapter 5 of this thesis, we introduce an algorithm for the verification of EPCs. Event-driven Process Chains were introduced in [113,114,162] and since the mid-nineties, a lot of work has been done on the verification of process models, in particular workflow models.

In 1996, Sadiq and Orlowska [157] were among the first ones to point out that modeling a business process (or workflow) can lead to problems like livelock and deadlock. In their paper, they present a way to overcome syntactical errors, but they ignore the semantical errors, i.e. they focus on the syntax to show that no deadlocks and livelocks occur.

Nowadays, most work that is conducted is focusing on semantical issues, i.e. “given a number of possible initial states, can the process specified always lead to an acceptable final state” and similar questions. The work that has been conducted on verification in the last decade can roughly be put into three main categories. In this section, we present these categories and give relevant literature for each of them.

#### 3.2.1 Verification of models with formal semantics

In the first category we consider the work that has been done on the verification of modeling languages with formal semantics. One of the most prominent examples of such a language is the Petri net language [67,138,147]. Since Petri nets have a formal mathematical definition, they lend themselves to great extent for formal verification methods. Especially in the field of workflow management, Petri nets have proven to be a solid theoretical foundation for the specification of processes. This, however, led to the need of verification techniques, tailored towards Petri nets that represent workflows. In the work of Van der Aalst and many others [5, 17, 63, 104, 175, 176] these techniques are used extensively for verification of different classes of workflow definitions. However, the result is the same for all approaches. *Given a process definition, the verification tool provides an answer in terms of “correct” or “incorrect”, (i.e. the model is sound or unsound).*

Not all modeling languages have a formal semantics. On the contrary, the most widely used modeling techniques, such as UML, BPMN and EPCs provide merely an informal representation of a process. These modeling techniques therefore require a different approach to verification.

#### 3.2.2 Verification of informal models

Modeling processes in a real-life situation is often done in a less formal language. People tend to understand informal models easily, and even if models are not executable, they can help a great deal when discussing process definitions. However, at some point in time, these models usually have to be translated into a
specification that can be executed by an information system, or used in some kind of analysis, such as simulation. This translation is usually done by computer scientists, which explains the fact that researchers in that area have been trying to formalize informal models for many years now. Especially in the field of workflow management, a lot of work has been done on translating informal models to Petri nets. Many people have worked on the translation of EPCs or BPMN models to Petri nets, cf., [4, 62, 69, 120]. The basic idea of these authors however is the same: “Restrict the class of models to a subclass for which we can generate a sound Petri net”. As a result, the ideas are appealing from a scientific point of view, but not useful from a practical point of view. Nonetheless, verification approaches for formal models, especially models such as workflow nets form a good basis for the analysis of informal models, however, they should be used differently.

Also non-Petri net based approaches have been proposed for the verification of informal modeling languages. One of these ideas is graph reduction. Since most modeling languages are graph-based, it seems a good idea to reduce the complexity of the verification problem by looking at a reduced problem, in such a way that correctness is not violated by the reduction, i.e. if a model is not correct before the reduction, it will not be correct after the reduction and if the model is correct before the reduction, it will be correct after the reduction. From the discussion on graph reduction techniques started by Sadiq and Orlowska in 1999 [158, 159] and followed up by many authors including Van der Aalst et al. in [15] and Lin et al in [123], as well as similar reduction-based approaches presented in [64, 138, 185], it becomes clear that again the modeling language is restricted to fit the verification process. In general this means that the more advanced routing constructs cannot be verified, while these constructs are what makes informal models easy to use. For an extensive overview of the existing work on EPC verification, using both graph reduction techniques, as well as translations to models with exact semantics, we refer to [129].

The verification approach we present in Chapter 5, is an approach towards the verification of EPCs, without considering an exact executable semantics of EPCs. Instead, the approach acknowledges the conceptual setting in which EPCs are typically used and relies on knowledge that a process owner has about a process. In [78–80], our verification approach was applied to a part of Aris for MySAP reference model, where it was shown that this reference model contains some errors which can be discovered. The Aris for MySAP reference model has been described in [58, 115] and is referred to in many research papers (see e.g. [84, 119, 135, 151, 173]). The extensive database of this reference model contains almost 10,000 sub-models, most of them EPCs. The SAP reference model was shown to contain a lower bound of 5% erroneous EPCs [12,130,131].

In a number of recent publications [132–134], a similar verification approach towards EPCs, based on WofYAWL [177,178], was used to check all EPCs in the
SAP reference model. The focus of these publications however is on predicting errors, i.e. determining characteristics of an EPC that predict whether or not the EPC contains errors.

### 3.2.3 Execution of informal models

The tendency to capture informal elements by using smarter semantics is reflected by recent papers, cf. [62, 117]. However, in these papers, the problem of verification is looked at from a different perspective, i.e. instead of defining subclasses of models to fit verification algorithms, the authors try to give a formal semantics to an informal modeling language. Even though all these authors have different approaches, the goal in every case is similar: try to give a formal executable semantics for an informal model.

Obviously, verification is strongly related to the efficient execution of models. Most approaches presented in this section, rely on executable semantics of the process model under consideration. As an example, we mention YAWL models [187]. YAWL models use an OR-join of which the intuitive idea is taken from EPCs. To obtain executable semantics for YAWL models, YAWL models are mapped onto reset nets to decide whether an OR-join is enabled or not in [186].

In the context of EPCs the possibility to provide executable semantics has been investigated in [118], where executable semantics are proven to exist for a large sub-class of all EPCs. In [56] an approach is presented to efficiently calculate the state space of an EPC, thereby providing executable semantics for the EPC. The authors mainly motivate this work from the viewpoint of simulation/execution although their approach can also be used for verification purposes. Because of the semantical problems in some EPCs [118] the algorithm does not always provide a result. Moreover, the authors also point out the need for “chain elimination” to reduce the state space of large models. In [129], a different semantics for EPCs is provided, that uses both state and context of an EPC.

### 3.2.4 Verification by design

The last category of verification methods is somewhat of a by-stander. Instead of doing verification of a model given in a specific language, it is also possible to give a language in such a way that the result is always free of deadlocks and livelocks. An example of such a modeling language is IBM MQSeries Workflow [121]. This language uses a specific structure for modeling, which will always lead to a correct and executable specification. However, modeling processes using this language requires advanced technical skills and the resulting model is usually far from intuitive, due to the enforces structure.
3.3 Conformance Checking

Although a first step towards translating process models into LTL formulas has been presented in [81], it makes more sense to directly compare a process log to the process model instead of an LTL translation of a process model. The process of comparing an event log against a process model is called conformance checking. Instead of some desired or undesired properties, a process model (e.g., a Petri net or an EPC) is used to verify whether the log satisfies some behavioral and structural properties, or the other way around, i.e. whether the model satisfies some structural or behavioral properties based on the log.

Especially when processes are modelled using informal or conceptual models which are, during enactment, not enforced on the users of an information system, conformance of a log to such a model is a relevant question. For example, people using the SAP R/3 system are not limited by process models described in the SAP R/3 Reference Model database [114].

Deviations from the specified “normal process” may be desirable but may also point to inefficiencies or even fraud. New legislation such as the Sarbanes-Oxley (SOX) Act [160] and increased emphasis on corporate governance has triggered the need for improved auditing systems [110]. For example, Section 404 of the SOX Act states two requirements: (1) Section 404(a) describes management’s responsibility for establishing and maintaining an adequate internal control structure and procedures for financial reporting and assessing the effectiveness of internal control over financial reporting, and (2) Section 404(b) describes the independent auditors responsibility for attesting to, and reporting on, management’s internal control assessment. Both requirements suggest an increased need for the detailed auditing of business activities. To audit an organization, these business activities need to be monitored, i.e. one needs to check whether an operational process adheres to the specified model.

The work in [13, 152, 153] demonstrates that there is not a simple answer to the question whether an event log matches a given process model. Therefore the authors propose an approach that considers two dimensions, i.e. the fitness, that measures to what extent the model is capable of reproducing the log under consideration, and appropriateness, measuring to what extent the model is a likely candidate to model the process that generated the log. Consider for example a Petri net with one place and all transitions consuming one token from and producing one token in that place. It is easy to see that that net can reproduce any log. However, it is not a very appropriate model since it does not give any information about causal dependencies. Therefore, in [13,152,153], the appropriateness is considered from both a structural and a behavioural point of view.

The work of Cook et al. [48,51] is closely related to conformance checking. In [51] the concept of process validation is introduced, i.e. two event streams are compared, an event stream coming from the model and an event stream coming
from real-life observations. Only in the last part of the paper an attempt is made to selectively try to match possible event streams of the model to a given event stream. As a result, only fitness is considered and the time-complexity is problematic as the state-space of the model needs to be explored (which is not necessarily finite as discussed in Section 2.3.1). In [48] these results are extended to include time aspects.

The notion of conformance has also been discussed before in the context of security [7], business alignment [6], and genetic mining [19,128]. However, in each of the papers mentioned only fitness is considered and appropriateness is mostly ignored.

3.4 Process Discovery

Next to log-based verification and process model verification, this thesis also addresses the topic of process discovery, i.e. given a process log we show several algorithms for deriving process models describing the operational process the log originated from. In this section, we introduce the related work on process discovery, where we start by introducing the related work on the theory or regions.

3.4.1 Theory of Regions

Process discovery is closely related to the so-called Theory of Regions, i.e. the theory where a Petri net is constructed (synthesized) of which the state space is bisimilar to the given transition system. This method, where “Regions” are defined as sets of states in the transition system and then translated to places in a Petri net is often called synthesis. Note that the method of synthesis is different from process discovery, since the input is a description of the complete behaviour of a system, instead of a set of executed traces.

We refer to [68] and [83] for the synthesis of safe Petri nets and [40] for more general cases. In these papers, the initial input describing the behaviour of the process is given in the form of transition systems (where the events are known but the states are anonymous). Typically, in process mining, the observed behaviour is not complete (as it is in a transition system) and it is not known, which process executions lead to which states (black box).

In a recent paper [124], regions are defined for partial orders of events representing runs. These regions correspond to places of a Place/Transition net, which can generate these partial orders. In contrast to our work, the considered partial orders are any linearizations of causal orders, i.e., two ordered events can either occur in a sequence (then there is a causal run with a condition “between” the events) or they can occur concurrently. Consequently, conditions representing tokens on places are not considered in these partial orders whereas our approach heavily depends on these conditions.
The application of the Theory of Regions in the context of process mining has been addressed in [23, 155], where the authors address process mining in a software engineering setting. One of the challenges faced there is to find state information in event logs. In [23, 155], the authors propose several ways of doing so. Furthermore, their approach is implemented in ProM by making a link between the event logs of ProM and a well-known tool tailored towards the application of the Theory of Regions, called Petrify [53].

Finally, it is worth mentioning that regions have been used in many different settings, e.g. in the synthesis and verification of asynchronous circuits (e.g. [52]) or in the verification of security properties (e.g. [44]).

The process discovery approach that we present in Section 6.2, uses the Theory of Regions [54] to construct process models without any implied behaviour, i.e. the resulting process models can reproduce the log under consideration and nothing more. We do so using an iterative algorithm that does not require the entire transition system to be built in memory. However, as the logs are seldom globally complete, the goal usually is to obtain a process model that does contain implied behaviour.

### 3.4.2 Process Discovery on Sequential logs

Historically, Cook et al. [49], Agrawal et al. [28] and Datta [60] were the first to work on the process discovery problem addressed in this thesis. Below, we discuss the related work in this area on a per author basis.

**Cook et al.**

Cook and Wolf have investigated the process discovery problem in the context of software engineering processes. In [49] they describe three methods for process discovery: one using neural networks, one using a purely algorithmic approach, and one Markovian approach. The authors consider the latter two the most promising approaches. The purely algorithmic approach builds a finite state machine where states are fused if their futures (in terms of possible behaviour in the next \( k \) steps) are identical. The Markovian approach uses a mixture of algorithmic and statistical methods and is able to deal with noise. Note that the results presented in [49] are limited to sequential behaviour.

Cook and Wolf extend their work to concurrent processes in [50]. They propose specific metrics (entropy, event type counts, periodicity, and causality) and use these metrics to discover models out of event streams. However, they do not provide an approach to generate explicit process models. Recall that the final goal of process discovery is to find explicit representations for a broad range of process models, i.e., we want to be able to generate a concrete Petri net rather than a set of dependency relations between events.
Agrawal et al.

The idea of applying process mining in the context of workflow management was first introduced in [28]. This work is based on workflow graphs, which are inspired by workflow products such as IBM MQSeries workflow (formerly known as Flowmark) and InConcert. In this paper, two problems are defined. The first problem is to find a workflow graph generating events appearing in a given workflow log. The second problem is to find the definitions of edge conditions. A concrete algorithm is given for tackling the first problem. The approach is quite different from other approaches: Because of their definition of workflow graphs there is no need to identify the nature (AND or OR) of joins and splits. As shown in [116], workflow graphs use true and false tokens which do not allow for cyclic graphs. Nevertheless, [28] partially deals with iteration by enumerating all occurrences of a given task and then folding the graph. However, the resulting conformal graph is not a complete model. In [127], a tool based on these algorithms is presented.

Pinter et al.

In [89, 143], Pinter et al. extend the work of Agrawal, by assuming that each event in the log refers either to the start or to the completion of an activity. This information is used to derive explicit parallel relations between activities, in a similar way as we do in Section 6.3.

Datta

In [60], Datta considers the problem of process mining as a tool for business process redesign or BPR [96,97]. In BPR, the starting point is typically assumed to be a set of process models that describe the current process. These models are then analyzed and better process models are constructed. The question that Datta addresses in [60] is how to get this initial set of models.

In [60], Datta proposes three strategies for extracting so-called “AS-IS” process models, based on stochastic modelling and finite state machine synthesis. The results of the procedural strategies proposed in [60] are provided as process activity graphs, which are state-based models of business processes. Such state-based models describe all possible paths individually and hence no solution is provided to “discover” parallelism. Therefore we expect that in highly parallel processes, the three strategies result in too complex models. However, there is no implementation available to verify this.

Herbst et al.

Herbst and Karagiannis also address the issue of process mining in the context of workflow management [105–107] using an inductive approach. The work presented in [107] is limited to sequential models. The approach described in
Chapter 3 Related Work

[105, 106] also allows for concurrency. It uses stochastic task graphs as an intermediate representation and it generates a workflow model described in the ADONIS modeling language. In the induction step task nodes are merged and split in order to discover the underlying process. A notable difference with other approaches is that the same task can appear multiple times in the workflow model, i.e., the approach allows for duplicate tasks. The graph generation technique is similar to the approach of [28,127]. The nature of splits and joins (i.e., AND or OR) is discovered in the transformation step, where the stochastic task graph is transformed into an ADONIS workflow model with block-structured splits and joins.

Schimm

Schimm [163–167] has developed a mining tool suitable for discovering hierarchically structured workflow processes. This requires all splits and joins to be balanced. However, in contrast to other approaches, he tries to generate a complete and minimal model, i.e. the model can reproduce the log and there is no smaller model that can do so. Like Pinter et al. in [89, 143], Schimm assumes that events either refer to the start or completion of an activity and he uses this information to detect explicit parallelism.

Greco et al.

Greco et al. [91, 92] present a process mining algorithm tailored towards discovering process models that describe the process at different levels of abstraction. Of the two step approach they present, the first step is implemented as the Disjunctive Workflow Miner plug-in in the process mining framework ProM, which we discuss extensively in Chapter 8.

Van der Aalst et al.

In contrast to the previous papers, the work of Van der Aalst et al. [26,126,182] is characterized by the focus on business processes with concurrent behaviour (rather than adding ad-hoc mechanisms to capture parallelism). In [182] a heuristic approach using rather simple metrics is used to construct so-called “dependency/frequency tables” and “dependency/frequency graphs”.

In [10] the EMiT tool is presented which uses an extended version of the $\alpha$-algorithm of [26] to incorporate timing information. EMiT has been replaced by a set of $s$ in ProM [11,74]. For a detailed description of the $\alpha$-algorithm, we refer to Section 6.4 and for a proof of its correctness we refer to [26]. For a detailed explanation of the constructs the $\alpha$-algorithm does not correctly mine, such as short loops, see [32,128].
Alves de Medeiros et al.

In [19,128], Alves de Meideros et al. propose two genetic algorithms towards process mining. They describe how to represent a process model using a so-called heuristic net and for these nets, they define the main ingredients of genetic algorithms.

The main ingredients for a genetic algorithm are:

- The *individuals*, i.e. the heuristic nets that form a *population*,
- The *fitness* that determines the quality of an individual compared to the input log,
- The *genetic operators*, i.e. the *crossover* and *mutation* operators that make that the population evolves into a better, i.e. a more fit, population over time,
- The *selection mechanism* that determine how to select individuals from a population that will be responsible for generating the next generation,
- The *stop criteria* that determine when to stop the algorithm.

The idea behind a genetic algorithm is that an initial population of random process models is constructed. Then, for each of the individuals the fitness is calculated and through some selection mechanism, some individuals are selected to produce the next generation by crossover and mutation. The goal of genetic mining is to get a heuristic net with the highest possible fitness, i.e. a net that best describes the log under consideration. Note that the fitness measurement used in this work, relates to the appropriateness presented in Section 3.3.

The concept of a heuristic net used in [19,128] was introduced by Weijters et al. in [182]. This notation turns out to be convenient for some discovery algorithms and can be mapped onto Petri nets and vice versa (cf. [19]).

Weijters et al.

The approach by Weijters et al. [16,182] provides an extension to the first step in the $\alpha$-algorithm, i.e. a heuristic approach is presented for determining causal and parallel dependencies between events. These dependencies then serve as an input for the $\alpha$-algorithm, thus making it applicable in situations where the process log contains noise, i.e. exceptional behaviour that we do not want to appear in a process model. Again, the work of Weiters et al. is implemented in the ProM framework.

Wen et al.

Two other extensions of the $\alpha$-algorithm are proposed by Wen et al. in [183,184], the first of which again assumes start and complete events to be present in the log and the resulting Petri net shows only activities, i.e. not the individual events.
Using this approach the problems the $\alpha$-algorithm has with short loops (loops of length one or two in a Petri net) are tackled. The second approach extends the $\alpha$-algorithm by explicitly searching for specific Petri net constructs, by looking at relations between events that are further apart in the event log, causally related in a more indirect way. This way, it is possible to correctly discover a bigger class of Petri nets than the class of nets the $\alpha$-algorithm can discover.

Others

More from a theoretical point of view, the process discovery problem discussed in this thesis is related to the work discussed in [37, 90, 144]. In these papers the limits of inductive inference are explored. For example, in [90] it is shown that the computational problem of finding a minimum finite-state acceptor compatible with given data is NP-hard. Several of the more generic concepts discussed in these papers could be translated to the domain of process mining. It is possible to interpret process discovery problems as inductive inference problems specified in terms of rules, a hypothesis space, examples, and criteria for successful inference. The comparison with literature in this domain raises interesting questions for process mining, e.g., how to deal with negative examples (i.e., suppose that besides log $W$ there is a log $V$ of traces that are not possible, e.g., added by a domain expert). However, despite the many relations with the work described in [37, 90, 144] there are also many differences, e.g., we are mining at the net level rather than sequential or lower level representations (e.g., Markov chains, finite state machines, or regular expressions).

Website

In this section, we gave an extensive overview of the research in the area of process discovery. However, since the field of process mining is very active, we refer to our website www.processmining.org for an overview of the latest work conducted in the area.

3.5 Partial Order Aggregation

In Chapter 7, we present algorithms for process discovery in case that the process logs do not just contain sequences of events, but rather partially ordered events. We will focus on Message Sequence Charts (MSCs) [100, 156], instance-EPCs and causal runs as input, but many other notations are available, e.g., UML Sequence Diagrams (SDs), Communication Diagrams (CDs), Interaction Overview Diagrams (IODs), and Harel’s Live Sequence Charts (LSCs).

In their basic form, these notations model individual scenarios, i.e., a particular example behaviour of the process/system and not the full behaviour. Although
technically, these scenarios are not logs, they could be seen as such, i.e. the specification of a complete system can be derived by aggregating these scenario based models.

Many of the approaches towards specifying individual scenarios have been extended with composition constructs to model a set of example behaviours or even the full process/system behaviour. The term “high-level MSCs” is commonly used to refer to MSCs which are composed using operators such as “sequence”, “iteration”, “parallel composition”, “choice”, etc. We consider these high-level MSCs as a less appropriate starting point for process discovery, i.e., if one just wants to model example behaviour, then the basic MSCs are more suitable. Moreover, if one wants to model the full system behaviour, traditional techniques such as EPCs and Petri nets seem more appropriate.

Many researchers have been working on the synthesis of scenario-based models (i.e. process discovery, where the scenarios are considered as input), in particular the generation of process models from different variants of MSCs. In [122] an excellent overview of 21 approaches is given. This overview shows that existing approaches are very different and typically have problems dealing with concurrency. Other problems are related to performance, implied scenarios (i.e., the model allows for more behaviour than what has actually been observed), and consistency (e.g., the synthesized model contains deadlocks).

It is impossible to give an overview of all approaches reported in literature. Therefore, we only describe some representative examples. Harel et al. [59,98,99] have worked on the notion of Live Sequence Charts (LSCs). The primary goal has been to turn LSCs (“play-in”) into an executable system (“play out”) without necessarily constructing an explicit process model. However, in [99] the synthesis of LSCs into statecharts is described. Note that this approach is very different from the notion of process discovery presented in this thesis, i.e., through the so-called prechart of LSCs the links between the various MSCs are made explicit. Hence there is no real synthesis in the sense of deriving a process model from example scenarios.

The latter observation holds for many other approaches, e.g., several authors assume “state conditions” or similar concepts to make the linking of partial orders explicit [86,174]. In a way, sets of scenarios are explicitly encoded in high-level models. However, there are also approaches that really derive process models from partial orders without some explicit a-priori encoding. An example is the work by Alur et al. [30,31]. In [30,31] two problems are discussed: the inference of additional (possibly undesirable) implied behaviour and the construction of incorrect models (e.g., models having potential deadlocks).

Finally, we would like to mention the approach described in [168]. Like many other authors, these authors provide formal semantics of MSCs in terms of Petri nets. The authors also synthesize MSCs into an overall Petri net. However, they assume that there is a Petri net gluing all MSCs together, i.e., sets of scenarios
are explicitly encoded in high-level MSCs.

Next to the aggregation of scenario based models, in Chapter 7, we also present three algorithms for the synthesis of Petri nets from so-called causal runs, i.e. models describing execution scenarios in terms of Petri nets.

The generation of process models from causal runs has been investigated before. The first publication on this topic is [170]. Here the input is assumed to be the set of all possible runs. These runs are folded, i.e., events representing the occurrence of the same transition are identified, and so are conditions representing a token on the same place. In [66] a similar folding approach is taken, but there the authors start with a set of causal runs, which is not necessarily complete. [66] does not present any concrete algorithm for the aggregation of runs but rather concentrates on correctness criteria for the derived system net.

### 3.6 Outlook

In this chapter, we introduced the related work in the areas of process mining and verification. In the remainder of this thesis, we revisit the topics of log-based verification, process model verification and process discovery and we extend the existing work with new algorithms or approaches.

In Chapter 4 the existing work on log-based verification is extended by an LTL based approach for checking LTL formulas on an event log. Our approach is based on event logs in the MXML format and has proven to be easy to use. Furthermore, like most of the work presented in this thesis, it is fully supported by an implementation in ProM.

In Chapter 5 we present an interactive approach towards the verification of EPCs. This approach uses knowledge that a process designer typically has about the process, which is not stored in the model directly. Furthermore, if a model contains errors they are explicitly shown in the model, but if the model does not contain errors, the result is either that the model is trivially correct, or that it can be correct if special care is taken during enactment. Again, this approach is fully supported by an implementation in ProM.

In Chapter 6 we move from verification to discovery, i.e. we present several approaches towards the discovery of process models from event logs. In contrast to most known approaches, our main result is a multi-step algorithm that always results in a correct EPC, i.e. an EPC that can generate the event log under consideration and is correct according to our EPC verification approach. This property however comes at a cost, since the resulting process model might contain a lot of implied behaviour, i.e. behaviour that is allowed in the process model, but not seen in the log.

Finally, Chapter 7 introduces different algorithms towards the aggregation of partial orders, i.e. we present algorithms to generate process models from a set of partially ordered examples. These partial orders can be specified in terms of
EPCs and Petri net, but also in terms of MSCs.

The ProM framework that contains implementations of most of the algorithms and approaches presented in this thesis, is extensively discussed in Chapter 8, where we introduce the many plug-ins developed in the context of this thesis in detail.
Chapter 4

Log-based Verification

In the classical BPM enactment phase, a process that is being executed is often described by a model. This model may be descriptive or normative and may even be used for enactment. However, besides such models, a company typically has policies that should be followed, or business rules that should be obeyed. These policies are usually formulated as properties on the case level of a process, e.g., each claim involving more than 10,000 euros should be handled by a senior employee.

Furthermore, when improving an information system in the process diagnosis phase, process experts gather information about the process under consideration, typically by collecting statements about the process from the people that are most involved in this process.

What remains unclear is how both the policies and collected statements can be checked for their validity, i.e., how can process owners know whether the right policy is being followed and whether the collected statements accurately describe their operational process. In this chapter, we present a means to check the validity of any case-based statement, if process logs are present, i.e., we present log-based verification.

4.1 Introduction

In Figure 4.1, we again present the relations between event logs, process models and (un)desired properties. When people are working on a process, they typically have to follow policies (also called rules or procedures). And when companies introduce new rules or policies, these are communicated to the people involved on paper. Furthermore, it is often made their responsibility to follow policies, i.e., the new policies are not enforced by the information system supporting the process. In other words, these rules and policies describe (in words) the desired and undesired properties an operational process should adhere to.

Furthermore, for model-driven information systems, once a process designer
makes a model of a process in the process design phase, this model will reflect the ideas of the designer about that process. In fact, to come up with a process model that correctly reflects the actual behaviour of a process is an extremely complex task. Typically, process designers will spend an enormous amount of their time gathering the ideas of people involved in the process. These ideas will then be collected and based on this a model is constructed, which is later used to implement an information system. However, although each person involved in an operational process has its own view on that process, usually that person does not have the overall picture.

Therefore, policies and statements made by a person should be objectively verified against what is actually going on in the organization, i.e. using a process log and a set of (un)desired properties, each case should be analyzed to see whether or not it satisfies each given property. More precisely, for each case in a process log, we want to check if a specific statement is valid for that case or not, i.e. we consider case-based statements.

In Section 4.2 we present an approach based on linear temporal logic, or LTL to perform such a verification, where we assume that logs are available for the process under consideration. After introducing the LTL based approach, we present some example queries in Section 4.5 some conclusions. Figure 4.2 shows the part of the whole thesis we present in this chapter.

4.2 Verifying Case-Based Statements

The starting point for log-based verification is a process log and some case-based statements that need to be checked, i.e., given a process log we want to verify
properties, such as the 4-eyes principle. The 4-eyes principle states that although authorized to execute two activities, a person is not allowed to execute both activities for the same case. For example, a manager may submit a request (e.g., to purchase equipment, to make a trip, or to work overtime) and a manager may also approve requests. However, it may be desirable to apply the 4-eyes principle implying that a request cannot be approved by the same manager who submitted it. If there is an event log recording the events “submit request” and “approve request”, such that for both events it is logged who executed it, the 4-eyes principle can be verified easily.

More difficult are those properties that relate to the ordering or presence of activities. For example, activity A may only occur if it is preceded by activity B or activity C and immediately followed by activity D. Therefore, we propose an approach based on temporal logic \[125, 145\]. Since we are considering case-based statements and, as we presented in Section 2.2, cases are linear sequences of events, we use an extension of Linear Temporal Logic (LTL) \[88, 101, 102\] tailored towards process logs holding information on activities, cases (i.e., process instances), timestamps, originators (the person or resource executing the activity), and related data.

Although the 4-eyes principle is an example of a statement that needs to be

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure42.png}
\caption{Overview of log-based verification chapter.}
\end{figure}
checked during the enactment of a process, LTL can also be used during the process design phase to verify statements that describe the process itself. Instead of requiring from a person to describe what is going on in an organization, it is typically more feasible to ask people about the (un)expected and/or (un)desirable properties. These properties can be directly compared with the event log, so that a process designer immediately knows if the person that is being interviewed knows enough about the process. Note that, in principle, one can first build a model and then compare the model and some property using model checking. We do not do this because we would like to verify statements as early as possible in the design phase.

The result of our verification approach is not a simple answer stating that a statement is true or false. Instead our approach points out exactly for which cases a statement was violated and for which cases it was not. As a result, a process designer might for example find out that a policy was violated twice, but in both cases the manager approved of this violation.

4.3 Example

Before we present the LTL-based language in Section 4.4, we first show a small example of our approach. Consider the event log in Table 4.1. Recall that this is a log of our running example presented in Section 2.6.

Consider the situation where the Immigration Service found that there was a problem with the integrity of John and that they need to reconsider all cases in which John denied a foreigner an MVV (recall that an MVV is some sort of temporary visa). In terms of a property, this should be specified as: Did John eventually execute the activity “Decide on MVV”, while the result of that activity was “denied”?

Checking this property on an event log requires to go through all process instances (i.e. cases) individually to see whether the outcome is true or false. In our example log of Table 4.1, the answer to this question for process instance 0 is true, i.e. this case should be reconsidered.

This small example nicely shows that the language should make use of as many aspects of the event log as possible (e.g. data elements, audit trail entry labels, etc.). In Section 2.2, we defined event logs as a linear ordering on events. Furthermore, in Subsection 2.2.3, we presented the MXML format for storing logs. Since this MXML format serves as a basis for our process mining framework ProM and allows for storing data attributes in a specific way, it is ver suitable for use with our LTL approach. Before we introduce some example statements and their applicability in practice in Section 4.5, we start by defining a language for specifying questions on a process log stored in the MXML format.
Table 4.1: Part of an MXML log file of our running example.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<WorkflowLog xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="WorkflowLog.xsd"
  description="Log of residence permit application model">
  <Source program="CPN Tools"></Source>
  <Process id="0" description="Permit Application">
    <ProcessInstance id="MVV00001" description="MVV application 0001">
      <AuditTrailEntry>
        <WorkflowModelElement>Decide on MVV advice</WorkflowModelElement>
        <EventType>start</EventType>
        <Timestamp>2002-04-08T10:55:00.000+01:00</Timestamp>
        <Originator>John</Originator>
      </AuditTrailEntry>
      <AuditTrailEntry>
        <Data>
          <Attribute name="result">accepted</Attribute>
        </Data>
        <WorkflowModelElement>Decide on MVV advice</WorkflowModelElement>
        <EventType>complete</EventType>
        <Timestamp>2002-04-08T11:55:00.000+01:00</Timestamp>
        <Originator>John</Originator>
      </AuditTrailEntry>
      <AuditTrailEntry>
        <WorkflowModelElement>Decide on MVV</WorkflowModelElement>
        <EventType>start</EventType>
        <Timestamp>2002-04-09T10:55:00.000+01:00</Timestamp>
        <Originator>John</Originator>
      </AuditTrailEntry>
      <AuditTrailEntry>
        <Data>
          <Attribute name="result">denied</Attribute>
        </Data>
        <WorkflowModelElement>Decide on MVV</WorkflowModelElement>
        <EventType>complete</EventType>
        <Timestamp>2002-04-09T11:55:00.000+01:00</Timestamp>
        <Originator>John</Originator>
      </AuditTrailEntry>
      <AuditTrailEntry>
        <WorkflowModelElement>Send MVV rejection letter</WorkflowModelElement>
        <EventType>start</EventType>
        <Timestamp>2002-04-09T12:55:00.000+01:00</Timestamp>
        <Originator>John</Originator>
      </AuditTrailEntry>
      <AuditTrailEntry>
        <WorkflowModelElement>Send MVV rejection letter</WorkflowModelElement>
        <EventType>complete</EventType>
        <Timestamp>2002-04-09T13:55:00.000+01:00</Timestamp>
        <Originator>John</Originator>
      </AuditTrailEntry>
    </ProcessInstance>
  </Process>
</WorkflowLog>
```
4.4 The language

Before being able to check properties on an event log, a concrete language for formulating dynamic properties is needed. Given the fact that we consider behavioural properties where ordering and timing are relevant, some temporal logic seems to be the best basis to start from [125,145]. There are two likely candidates: Computational Tree Logic (CTL) and Linear Temporal Logic (LTL) [88,101,102]. Given the linear nature of each process instance (or case) within a process log, LTL is the obvious choice. It would only make sense to use CTL if first a model (e.g., an automaton) was built before evaluating the property, and as we mentioned before, we do not assume a model to be present.

It is not sufficient to select LTL as a means to specify statements, i.e., as a language. To easily specify statements in the context of MXML, a more dedicated language is needed that exploits the structure shown in Figure 2.4. Therefore, in addition to standard logical operators, we need dedicated operands to address the properties of process instances and audit trail entries. For example, it should be easy to use the various attributes of a process (both standard ones such as process instances, audit trail entries, timestamps, originators and event types, and context specific ones such as data values).

We developed a powerful language [9] to reason about cases that includes type definitions, renaming, formulas, subformulas, regular expressions, date expressions, propositional logic, universal and existential quantification, and temporal operators such as nexttime ($\diamond F$), eventually ($\lozenge F$), always ($\Box F$), and until ($F U G$). Since the formulas specify properties on a case level, the scope of each of these operators is the case that is being considered, e.g., “eventually” means “eventually in this case”. A complete description of the language is given in [41]. To illustrate the language we use the examples shown in Table 4.2.

The language contains two default identifiers for the current process instance $pi$ and the current audit trail entry $ate$. When a formula is checked on a process instance, the notation $pi.X$ is used to refer to some attribute $X$ of that process instance. The notation $ate.X$ is used to refer to some attribute $X$ of an audit trail entry ($ate$). Furthermore, there are several predefined attributes, e.g., $ate.WorkflowModelElement$ refers to the activity being executed. $ate.Originator$ is the resource executing it. $ate.Timestamp$ is the timestamp of the event. $ate.EventType$ is the type of the event (i.e., schedule, assign, etc. as presented in Section 2.2.2). Note that the names of these attributes correspond to the names in the MXML format presented in Section 2.2.3.

The three set declarations shown in Table 4.2 (lines 1-3) declare that $ate.WorkflowModelElement$, $ate.Originator$, and $ate.EventType$ can be used for quantification, e.g., $ate.WorkflowModelElement$ may refer to the activity related to the current event but may also be used to range over all activities appearing in a case.

Line 5 declares the DateTime format used when specifying a value (note
that this allows for shorter notations than the standard XML format. Line 6 declares a data attribute at the level of an event. The data attribute result is used to record the outcome of a single review in the running example. Line 7 shows a data attribute at the level of a case. Note that both attributes are of type string. To allow for shorter/customized names our language allows for renaming. As shown in lines 9-11, \texttt{ate.Originator}, \texttt{ate.Timestamp}, and \texttt{ate.WorkflowModelElement} are renamed to \texttt{person}, \texttt{timestamp}, and \texttt{activity} respectively. These names are used in the remainder of Table 4.2.

The goal of our dedicated LTL language is to specify properties, which are described using the \texttt{formula} construct. Formulas may be nested and can have parameters. Furthermore, to hide formulas that are only used indirectly, the \texttt{subformula} construct should be used. Table 4.2 describes five formulas and two subformulas, which could be used to verify a review process of some journal, where for each submitted paper a new process instance is started. Lines 13-14 specify a formula without any parameters. The property holds for a given event log if for each process instance (i.e. for each paper), there is an audit trail entry referring to the activity accept, or reject, but not both. To formulate this, both temporal and logical operators are used: $<>$ is the syntax for the temporal operator eventually $(\Diamond F)$, $\leftrightarrow$ denotes “if and only if”, and $!$ is the negation. Line 14 uses the shorthand \texttt{activity} defined in Line 11 twice. \texttt{activity == "accept"} is true if the \texttt{WorkflowModelElement} attribute of the current audit trail entry points to the acceptance activity. Hence, $<>(\texttt{activity == "accept"})$ holds if the acceptance activity was executed. Similarly, $<>(\texttt{activity == "reject"})$ holds if the rejection activity was executed. Using $\leftrightarrow$ and $!$ we can formulate that exactly one of these two should hold. The formula \texttt{accept_or_reject_but_not_both} can be evaluated for each case in the log. If it holds for all cases, it holds for the entire log.

Lines 16-19 define the property that any decision about a paper should be directly followed by a rejection, acceptance or invitation of a new reviewer. The following logical and temporal operators are used to achieve this: $[]$ to denote the always operator $(\Box F)$, $\rightarrow$ for implication, $\_0$ denote the nexttime operator $(F)$, and $\lor$ for the logical or. The part $[](\texttt{activity == "decide"} \rightarrow$ states that it should always be the case that if the current activity is decide, the next activity should be one of the three mentioned. This is expressed in the second part, which starts with $\_0$ to indicate that immediately after the decision step the current activity should be one of the three mentioned.

The formula specified in lines 24-28 uses the parameterized subformula defined in lines 21-22. The subformula states whether at some point in time activity $a$ was executed by person $p$. Note that both parameters are typed through the declarations in the top part of Table 4.2. Formula \texttt{not_the_same_reviewer} calls the subformula six times to express the requirement that no reviewer should review the same paper twice. Note that universal quantification over the set
Table 4.2: Some LTL formulas for a paper review example [180].

```plaintext
1 set ate.WorkflowModelElement;
2 set ate.Originator;
3 set ate.EventType;
4
5 date ate.Timestamp := "yyyy-MM-dd";
6 string ate.result;
7 string pi.title;
8
9 rename ate.Originator as person;
10 rename ate.Timestamp as timestamp;
11 rename ate.WorkflowModelElement as activity;
12
13 formula accept_or_reject_but_not_both() :=
14 (<> (activity == "accept") <-> !(<> (activity == "reject")));
15
16 formula action_follows_decision() :=
17 []( (activity == "decide" -> _O( ((activity == "accept" \/
18       activity == "reject") \/
19       activity == "invite additional reviewer") )))
20
21 subformula execute( p : person, a : activity ) :=
22 <> ( (activity == a /\ person == p ) );
23
24 formula not_the_same_reviewer() :=
25 forall[ p : person |
26 (((execute(p,"get review 1") \/ !execute(p,"get review 2") ) \/
27 (((execute(p,"get review 1") ) \/ !execute(p,"get review 3") ) \/
28 (((execute(p,"get review 2") ) \/ !execute(p,"get review 3") ) )
29
30 subformula accept(a : activity ) :=
31 <> ( (activity == a /\ ate.result == "accept" ) );
32
33 formula dont_reject_paper_unjustified() :=
34 (((accept("get review 1") \/ accept("get review 2") ) \/
35 accept("get review 3")
36 -> <> ( activity == "accept" ) );
37
38 formula started_before_finished_after(start_time:timestamp,
39    end_time:timestamp ) :=
40 (<>( timestamp < start_time ) \/ <> ( timestamp > end_time ));
```
people involved is used (cf. \texttt{forall[p:person | ...]) where \texttt{person} is renamed in Line 9 and declared to be a set type in Line 2. When evaluating the formula on a process log, the set of persons contains all the originators that are involved in the process, not only the ones involved in a single process instance.

The formula specified in lines 33-36 uses the parameterized subformula defined in lines 30-31. The subformula checks whether there is some event corresponding to activity \(a\) that has a data attribute \texttt{result} set to value \texttt{accept}, i.e., \texttt{ate.result == "accept"}. Note that \texttt{ate.result} was declared in Line 6. Formula \texttt{dont_reject_paper_unjustified} states that a paper with three positive reviews (three accepts) should be accepted for the journal.

The last formula in Table 4.2 (lines 38-40) shows that it is also possible to use timestamps. The formula has two parameters (start and end time) and it holds if each case was started before the given start time and ended after the given end time.

The formulas shown in Table 4.2 are specific for a paper selection process of a journal. However, since formulas can be parameterized, many generic properties can be defined, e.g., the \textit{4-eyes principle}. Recall that this principle states that, although authorized to execute two activities, a person is not allowed to execute both activities for the same case. The following formula can be used to verify this:

\begin{verbatim}
formula four_eyes_principle(a1:activity,a2:activity) :=
    forall[p:person | (!(execute(p,a1)) \(\lor\) !(execute(p,a2)))];
\end{verbatim}

In the context of our running example that we presented in Section 2.6, this formula could be called with the parameters “Send permit rejection letter” and “Evaluate objection”. It then checks whether there is a single person that first denies an applicant a residence permit and then also handles the objection to that decision.

### 4.4.1 Limitations

Although the LTL language we present in this chapter is very powerful, it has limitations [45]. In LTL formulas, you are not able to count, i.e. you cannot define a formula stating that if activity \(A\) is executed \(n\) times, then activity \(B\) should also be executed \(n\) times. Although it is possible to define separate formulas for all different values of \(n\), a single formula, where \(n\) is a parameter, cannot be constructed.

Another limitation of the LTL-based approach is that it assumes time to be linear, i.e. the temporal operators reason about the order of events, but not about the physical time when the events occurred, or the physical time spent between events. It is for example impossible to check whether \texttt{activity A} was completed within two hours after it started, unless this duration is explicitly recorded as a data element.
The example of the 4-eyes principle and the formulas in Table 4.2 provide an impression of the LTL language we developed, which, despite its limitations, has proven to be very useful in the context of process mining. In Section 4.5, we present a set of example formulas with generic parameters. These formulas and many others are contained by default in the ProM framework, which we discuss in Chapter 8.

### 4.5 Example Statements

In Subection 4.4, we presented a language for writing statements about process logs. In this section, we show some example of such statements. For each statement, we show exactly how they are written in LTL files that can be read by our process mining framework ProM (cf. Chapter 8) and what they express.

#### 4.5.1 Case Finalization

In business process management, it is usually important that cases are completed at a certain point in time. Furthermore, a case is typically completed if one of the possible final activities has been executed. As an example, we assume that there are three possible outcomes and we specify a formula checking whether at least one of them is eventually executed.

The formula in Table 4.3 consists of the following parts. First, lines 1-3 defines the formula and its parameters, i.e. two activities $A$, $B$ and $C$. Recall from Table 4.2 that activity is a renaming of $at.WorkflowModelElement$, i.e.

<table>
<thead>
<tr>
<th>Table 4.3: Eventually one of two activities occurs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>formula eventually_one_of_activities_A_B_C( A: activity, B: activity, C: activity ) :=</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>&lt;h2&gt;Does one of the activities A, B or C occur?&lt;/h2&gt;</td>
</tr>
<tr>
<td>&lt;p&gt; Compute if there is an activity of which the name is contained in {A,B,C}.&lt;/p&gt;</td>
</tr>
<tr>
<td>&lt;p&gt; Arguments:&lt;br&gt;</td>
</tr>
<tr>
<td>&lt;ul&gt;</td>
</tr>
<tr>
<td>&lt;li&gt;&lt;b&gt;A&lt;/b&gt; of type set (&lt;i&gt;at.WorkflowModelElement&lt;/i&gt;)&lt;/li&gt;</td>
</tr>
<tr>
<td>&lt;li&gt;&lt;b&gt;B&lt;/b&gt; of type set (&lt;i&gt;at.WorkflowModelElement&lt;/i&gt;)&lt;/li&gt;</td>
</tr>
<tr>
<td>&lt;li&gt;&lt;b&gt;C&lt;/b&gt; of type set (&lt;i&gt;at.WorkflowModelElement&lt;/i&gt;)&lt;/li&gt;</td>
</tr>
<tr>
<td>&lt;/ul&gt;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>&lt;&gt; ( ( activity == A \ activity == B ) \ activity == C ) ;</td>
</tr>
</tbody>
</table>
of the workflow model element contained in an audit trail entry as specified in Section 2.2.

Lines 4-15 then define a HTML based description of the formula, i.e. what does it check and what parameters does it contain. This description is used by ProM and displayed as a textual definition of the formula.

Line 16 then finally defines the formula, i.e. it says that \textit{eventually} the activity equals $A$, or the activity equals $B$, the activity equals $C$.

Although our verification approach is applicable in the context of process mining (i.e. when a process log is present), this formula illustrates that our LTL approach could also be used to check if deadlines are met, i.e. as soon as a case is started, you schedule the formula of Table 4.3 to be checked when the deadline expires. If the formula’s result is not true, then the deadline was not met and you send a message to some manager.

### 4.5.2 Task Completion

Another example of a relevant question is a formula that checks for each case whether all tasks within the case that were started have been completed, i.e. for all activities, if there is a “start” event, then there should later be a “complete” event.

Table 4.4 shows this formula in LTL. Lines 1 and 2 define the formula and its parameters $E$ and $F$ and lines 3-12 explain the formula in HTML text. The actual formula is defined in lines 13-18, where \texttt{forall} activities is checked whether the combination of that activity with event $E$ always leads to the combination of that activity with the event $F$. Note that in this \texttt{forall} statement, the “activity” is used at a type level, (i.e. for all activities $a$ that are present in the process log), whereas in line 14, the always statement uses “activity” to refer to an instance of an activity, namely the activity at the current position in the process instance when checking this formula.

Note that this formula does not consider loops or multiple executions, i.e. it might be that activity $x$ is started many times, but completed only once afterwards. Requiring that each activity is completed before it can be started again can be incorporated into an LTL-formula. For an extensive discussion on how LTL can be used for defining such control flow related statements we refer to [20, 21, 142].

### 4.5.3 Retaining Familiarity

Consider the resource pattern of retaining familiarity [18, 116], that states that one person should execute a number of tasks within the same case, i.e. if person $x$ executes task $a$, then person $x$ should also execute task $b$. Again, this can be translated into our LTL language and Table 4.5 shows how, using one subformula and one formula. The subformula of lines 1 and 2 tells whether a certain person
Table 4.4: Eventually both events for one activity occur.

\begin{verbatim}
formula forall_activities_always_event_E_implies_eventually_event_F( 
    E: event, F: event) := 
{ 
  <h2>Hold for all activities that if event E occurs, 
  then eventually event F occurs too</h2>
  <p> Arguments:<br>
  <ul>
    <li><b>E</b> of type set (<i>ate.EventType</i>)</li>
    <li><b>F</b> of type set (<i>ate.EventType</i>)</li>
  </ul>
}</p>
for all[ a: activity | 
  [() ( ( activity == a /\ event == E ) 
    -> 
    <>(( activity == a /\ event == F ))
  ) ];
\end{verbatim}

Table 4.5: Eventually one person performs two tasks.

\begin{verbatim}
subformula P_does_A( P: person, A: activity ) := {}
( person == P /\ activity == A);

formula one_person_doing_task_A_and_B( A: activity, B: activity ) := 
{ 
  <h2>Is there a person doing task A and B?</h2>
  <p>
  <li><b>A</b> of type set ( <i>ate.WorkflowModelElement</i>)</li>
  <li><b>B</b> of type set ( <i>ate.WorkflowModelElement</i>)</li>
  <ul></p>
} exists[ p: person | 
( <>(( P_does_A( p, A ) ) /\ <>(( P_does_A( p, B ) ) ) ));
\end{verbatim}
executed a given activity. The formula of lines 4-13 checks if there is one person that executes both tasks $A$ and $B$, where it uses the subformula twice.

4.6 Conclusion

In this chapter, we introduced an approach based on linear temporal logic, or LTL, to verify case-based statements about a process against an execution history of that process. Process logs that accurately describe what has been done by whom in the history of a process serve as an input for the objective verification of such statements.

First of all, this approach is useful during the enactment phase of business process management, since it allows a system to check whether policies are followed, and pinpoint the cases for which a policy was violated.

Using the LTL-based approach, process logs can be split up in smaller logs of which certain properties are known, i.e. a log can be split up into one partition where a policy is followed and one where this is not the case. Using this mechanism, logs can be pre-processed for process discovery, in such a way that only that part of the log that relates to the most urgent problems can be analyzed first.

Furthermore, since our LTL-based approach does not require any explicit process model, it helps a process designer in the design phase of business process management. During this phase, the process designer collects statements about the process under consideration and LTL can be used to verify the information received. If not, the right information can be acquired from the right people even before the actual modelling started.

Finally, we would like to mention that the LTL-formulas presented in this chapter can not only be used for the verification of properties on an event log, but also for giving an executable specification of a process. For this, we refer to [20, 21, 142], where a tool is presented that can execute sets of LTL formulas, i.e. the tool uses LTL formulas as an executable language for business process modelling.
Chapter 5

Process Model Verification

In the introduction, we presented the concept of process mining. Especially where it concerns control flow discovery (i.e. constructing a process model from a set of execution traces), it is important to note that the resulting process model should be a correct model. Moreover, for all process models, whether they are made by a process designer or derived using a process discovery algorithm, it is important that they possess certain properties. Obviously, these models should be a good

Figure 5.1: Overview of process model verification chapter.
abstraction of the process under consideration. Examples of properties that check whether that is the case are: that every case that is started should be finished at some point, or that no case can result in an undesired state. In this chapter, we introduce techniques to check such properties on a process model. More precisely, we introduce process model verification, as shown in Figure 5.1.

5.1 Introduction

Recall the classical BPM life cycle which we presented in Figure 1.1. It shows that a process is consecutively diagnosed, designed, configured and enacted, after which this whole process starts again. In Chapter 4, we presented an approach to objectively verify statements about a process against the actual process enactment, by using an LTL based approach on process logs.

In this chapter, we focus on the situation where process models are provided. Such models could either be used for enactment, e.g. in a workflow environment, or be used as a reference as is the case in the SAP reference model mentioned in Section 1.2. However, whether a process model is used during enactment or not, it is of the utmost importance that the model is correct, i.e. it correctly models the process under consideration.

In Section 5.3, we present an algorithm for the verification of Event driven Process Chains, a typical conceptual modelling language. Our approach towards EPC verification relies heavily on the verification concepts known for workflow nets. Therefore, we first introduce those in Section 5.2. We conclude this chapter with a case study presented in Section 5.4, where we apply our verification technique to part of the SAP reference model.

5.2 Workflow Net Verification

Checking a process model for problems is not as easy as it sounds. Workflow nets however, have unambiguous semantics and it is possible to use a variety of Petri net based analysis techniques that have been developed in the past to answer different questions relating to verification.

Given a marked Petri net, i.e., a Petri net and some initial marking, an interesting question is which markings can be reached. That is, what is the set of reachable markings (also called the state space) for that marked Petri net? Related to this is the question whether the state space is finite, i.e., whether the number of reachable markings is finite. If this is the case, the marked Petri net is called bounded, i.e. for every place we can find some upper bound on the number of tokens inside that place. If this upper bound is 1, then the marked Petri net is called safe. A marked Petri net is called live if every transition can be fired from any reachable marking (possibly after some other transitions have been fired.
first). It should be noted that although the state space of a bounded Petri net is finite, it can be too big for any computer to compute.

A standard approach to decide boundedness and safeness for a marked Petri net is to construct a coverability graph [67,138,147]. For many marked Petri nets it is possible to construct the coverability graph: If the net is unbounded, there are infinitely many reachable markings, but these can be grouped in such a way that the coverability graph is still finite. Furthermore, if the marked Petri net happens to be bounded (which we can check using the coverability graph), then the coverability graph is identical to the state space. As a result, we can check liveness if the marked Petri net is bounded.

In this thesis, we mostly consider workflow nets (WF-nets), which we defined in Section 2.3.2. WF-nets are a subclass of Petri nets tailored towards workflow modeling and analysis. Based on WF-nets, correctness notions such as soundness [5,14], and relaxed soundness [62,63] have been defined, which we introduce in detail, since they play an important role in our work on EPC verification which we present in Section 5.3.

5.2.1 Soundness

For our approach, the notions of soundness and relaxed soundness are highly relevant, therefore we describe these in more detail. Recall from Section 2.3.2 that a WF-net has one source place and one sink place. Let us assume that $p_i$ is the source place and $p_f$ is the sink place. Place $p_i$ is the entry point for new cases (i.e., process instances), while place $p_f$ is the exit point. Ideally, every case that enters the WF-net (by adding a token to place $p_i$) should exit it exactly once while leaving no references to that case behind in the WF-net (no tokens should be left behind, except in $p_f$, in which one token should reside). Furthermore, every part of the process should be viable, that is, every transition in the corresponding WF-net should be potentially executable. Together these properties correspond

![Figure 5.2: Workflow net representing the process of our running example of Section 2.6.](image-url)
to the notion of soundness [5,14], i.e., a WF-net is sound if and only if:

- From every marking reachable from $[p_i]$, the marking $[p_f]$ is reachable (completion is always possible).
- Every transition is included in at least one firing sequence starting from $[p_i]$ (no dead transitions).

Recall that $[p_i]$ denotes the marking with one token in place $p_i$ and $[p_f]$ denotes the marking with one token in place $p_f$. Formally, we can define soundness as follows:

**Definition 5.2.1. (Soundness)**
Let $\varphi = (P,T,F)$ be a WF-net with input place $p_i$ and output place $p_f$. $\varphi$ is sound if and only if:

1. **option to complete**: for any marking $M \in [\varphi,[p_i]]$, $[p_f] \in [\varphi,M]$, and
2. **absence of dead tasks**: $(\varphi,[p_i])$ contains no dead transitions.

The set of all sound WF-nets is denoted $\mathcal{W}$.

Consider Figure 5.2, that shows a WF-net of our running example as explained in Section 2.6. Note that it contains one extra step, namely the transition “timeout, no objection received”. This transition is necessary in the WF-net, while it is not there in the EPC of Figure 2.9, since the WF-net does not contain events to express this situation. It is not difficult to verify that this process satisfies the requirements of soundness, i.e. there is always the option to complete and if the process completes, it does so properly with one token in the final place. Furthermore, there are no dead tasks.

Some verification techniques require the addition of an extra transition connecting the sink place back to the source place. Such a short-circuited WF-net can be used to express soundness in terms of well-known Petri net properties: A WF-net is sound if and only if its short-circuited net is live and bounded [3]. Liveness and boundedness are two well-known properties supported by a variety of analysis tools and techniques [67,138,147]. This transformation is based on the observation that an execution path that moves a token from place $p_i$ to place $p_f$ corresponds to a cyclic execution path in the short-circuited net: By executing the short-circuiting transition once, the token is back in place $p_i$. An example of a tool tailored towards the analysis of WF-nets is Wo\lan [179].

### 5.2.2 Relaxed soundness

In some circumstances, the soundness property is too restrictive. Usually, a designer of a process knows that certain situations will not occur. As a result, certain execution paths in the corresponding WF-net should be considered impossible. Thus, certain reachable states should be considered unreachable. Note
that in verification of processes, we are often forced to abstract from data, applications, and human behaviour, and that it is typically impossible to model the behaviour of humans and applications. However, by abstracting from these aspects, typically more execution paths become possible in the model. The notion of relaxed soundness [62, 63] aims at dealing with this phenomenon. A WF-net is called relaxed sound if every transition can contribute to proper completion, i.e., for every transition there is at least one execution of the WF-net starting in state $p_i$ and ending in state $p_f$ which involves the execution of this transition.

Formally, we define relaxed soundness as:

**Definition 5.2.2. (Relaxed soundness)**

Let $\varphi = (P,T,F)$ be a WF-net with input place $p_i$ and output place $p_f$. A transition $t \in T$ is relaxed sound if and only if, there exists a firing sequence $\sigma \in T^*$, such that $(\varphi, [p_i]) [\sigma] (\varphi, [p_f])$ and $t \in \sigma$. The workflow net $\varphi$ is relaxed sound if and only if all transitions $t \in T$ are relaxed sound.

As mentioned before, every case that enters a WF-net should exit it exactly once while leaving no references to that case behind in the WF-net (no tokens should be left behind). Thus, the ultimate goal of a WF-net is to move from place $p_i$ to place $p_f$. The notion of relaxed soundness brings this goal down to the level of transitions: every transition should aid in at least one possible scenario moving a token from place $p_i$ to place $p_f$. A transition that cannot aid in moving a token from place $p_i$ to place $p_f$ cannot help the WF-net in achieving its goal. Hence, such a transition has to be erroneous.

Consider Figure 5.3, where we show a modified version of the WF-net shown in Figure 5.2. To this WF-net, we added the constraint that an MVV acceptance letter can only be sent after the documents have been sent to the Town Hall, but before the conditions are checked with the applicant. It is clear that this WF-net is no longer sound, since the process can deadlock if the conditions are checked with the applicant before the acceptance letter is sent.
Interestingly, the example in Figure 5.3 also shows the practical use of relaxed soundness, since it does not sound unlikely that an applicant will not come to the Town Hall to check the application, before it got a letter that the MVV is accepted, hence the deadlock will never occur. In the sound model for Figure 5.2 the deadlock could also not occur, but it could happen that the applicant got the letter of acceptance before the papers were sent to the Town Hall, which means that the applicant can call the Town Hall for an appointment while they do not have the appropriate dossier yet.

The concept of relaxed soundness is highly relevant when looking at correctness of EPCs. In EPCs, the notion of tokens is not present. Therefore, the modelling language does not have a clear execution semantics. However, in the next section, we introduce a verification technique of EPCs, where we translate the EPC to a WF net and assume that the EPC is correct if and only if that WF net is relaxed sound.\footnote{In essence, this means that this verification approach defines an execution semantics for sound EPCs, since each firing sequence of the WF-net that results in the final marking \([p_f]\) corresponds to a possible execution of the EPC.} Note that we do not use the term “sound” for EPCs, since this might confuse the reader.

## 5.3 EPC Verification

As we stated before, the EPC modelling language is used by some leading industrial modelling tools, such as Aris PPM, SAP, etc. By their success in business, EPCs have proven to be an intuitive modelling language for humans. However, because of the lack of clear execution semantics, the question whether or not an EPC is correct is not as trivial as it might seem. In this section, we introduce a verification approach for EPCs that consists of two steps. First, we introduce reduction rules to eliminate the “easy” constructs for which it is generally accepted that they are correct. If necessary, in the second step, we translate the EPC into a safe WF net and use the concept of relaxed soundness to verify the WF-net. The second step provides the designer of the EPC with one of the following three answers:

**The WF-net is sound:** The original EPC is correct and no further reviewing is necessary.

**The WF-net is relaxed sound (but not sound):** The original EPC can be correct, but the designer needs to assess some problematic constructs. If the designer assesses the EPC to be incorrect, corrections are necessary.

**The WF-net is not relaxed sound:** The original EPC is incorrect, regardless of the designer’s assessment. Corrections are necessary.

Figure 5.4 shows the EPC of our running example, that we have seen before in Figure 2.9. We illustrate our verification approach using this EPC.
Figure 5.4: EPC of our running example presented in Section 2.6.
To understand the rationale of our approach, it is important to see that we address the issue of verification from a designer’s perspective instead of from a formal perspective, where the result would always be sound or not sound without any need for interpretation of the process model. Instead, our verification process consists of two main parts. First we take the EPC that is defined by a process designer and, using simple reduction rules, we reduce the net. As a result, the designer can focus on the possibly problematic areas. Second, we translate the result into a Petri net and use variants of existing WF-net-based verification techniques to give feedback to the designer. By using an extended version of the relaxed soundness notion for general Petri nets and, if needed, human judgment, the correctness of the model is assessed. Especially the explicit use of human judgment is an often missing factor in the area of process model verification.

Although relaxed soundness it typically defined for WF-nets, where the initial and final markings are known, we use an extended version of this concept. The idea behind relaxed soundness was that each transition should contribute to at least one firing sequence going from the given initial marking to the given final marking. In our approach, we generalize this, by allowing the EPC designer to manually identify the possible initial markings and the possible final markings. Each transition then has to contribute in at least one firing sequence going from one of the initial markings to one of the final markings.

In the previous section, we introduced relevant notions such as (relaxed) soundness for WF-nets. In the remainder of this section, we show the process of EPC verification. In Subsection 5.3.1, we reduce the verification problem of a large EPC to that of a smaller (and possibly trivial) EPC. Then, in Subsection 5.3.2, we present the whole approach and show how we can use WF-net based techniques to further analyze the reduced EPC. Finally, Subsection 5.3.3 discusses an extension to the initial algorithm, that uses so-called invariants if the state space is too large to be constructed in reasonable time.

5.3.1 Reduction rules

EPCs may contain a large number of functions, events and connectors. However, for the verification of EPCs, not all of these elements are of interest. In particular, we are interested in the routing constructs that are used in the EPC, since that is where the errors can be. Furthermore, it is obvious that some constructs are trivially correct. For example, an arbitrary long sequence of events and functions is clearly correct and does not need to be considered in detail in the verification process, assuming relaxed soundness is what we aim for (In [118], removing such chains of events and functions is called “chain elimination”). Moreover, if a split of some type is followed by a join of the same type, the exact semantics of the splits and joins is irrelevant as under any semantics this will be considered
correct\textsuperscript{2}. In this section, we introduce a set of reduction rules. These rules can be applied to any EPC in such a way that, if the EPC is correct before the reduction, then the result after reduction is correct and if the EPC is not correct before reduction, then the result after reduction is not correct, i.e. these rules are correctness preserving in the EPC, or relaxed soundness preserving in the WF-net that is generated from an EPC. However, we do not intend these rules to be complete. Instead, they merely help to speed up the verification process, by removing trivial parts before going to the more demanding steps in process (i.e., both in terms of computation time and human interpretation).

Without presenting the translation to WF-nets just yet, it is important to note that for a reduction rule to be relaxed soundness preserving in the WF-net translation of an EPC, it needs to be correctness preserving in an EPC, where we have to take both all nodes and all edges into account. In other words, an EPC is correct if all functions, events, connectors and edges are contained in at least one correct execution sequence.

It is easily seen that the application of the reduction rules does not result in an EPC, since functions and events no longer alternate. However, for the process of verification, this is not a problem and we will refer to this reduced model as a reduced EPC. There are several reduction rules, i.e. rules to reduce trivial constructs, simple split/joins, similar splits/joins, trivial loops and complex OR loops. Below, we discuss each of these reduction rules separately.

**Trivial constructs**

Figure 5.5 shows the reduction rules for trivial constructs, i.e. it shows that a function $f$, an event $e$ or a connector with type $t_1$ with precisely one ingoing and one outgoing edge can be removed completely. Functions, intermediate events and connectors with single input and outputs can safely be removed without disturbing the routing behaviour of the EPC and since we are only interested in the routing behaviour, it is easy to see that this rule is correctness preserving.

The rule that can be used to reduce trivial constructs is correctness preserving, since:

1. If the EPC is correct before reduction, then there is a correct trace with this function, connector or event, but also with the incoming and outgoing edge. Since we do not remove the edges, the same trace is still a correct trace for the new edge combining the two original edges.

2. If the EPC is incorrect before reduction, then there is no correct trace with this function, connector or event. However, this implies that there is no correct trace with the incoming or outgoing edge of this node, hence after removing the function, connector of event, the result is still incorrect.

\textsuperscript{2}At least in any semantics that would comply with the intuitive human interpretation of EPCs.
Figure 5.5: Trivial construct.

Figure 5.6: Simple split/join.

Figure 5.7: Similar joins.

Figure 5.8: Similar splits.

Figure 5.9: XOR loop.
Simple splits/joins

Figure 5.6 shows the reduction rule for a split that is followed by a join connector. This rule can be applied if both connectors are of the same type (i.e. AND, OR or XOR), or if the join connector is of type OR. This rule is correctness preserving, since:

1. If the EPC is correct before reduction, then there is a correct trace for both connectors and all edges between them. This will not change by removing edges, since one of the edges remains, hence the EPC is correct after reduction.

2. If the EPC is incorrect before reduction and there is no correct trace containing the first connector, then there is no correct trace containing the incoming edge of that connector and such a trace still does not exist in the reduced EPC. If there is no correct trace containing the second connector, this can only be if either the problem is caused by the other inputs of the second connector, or if at least one of the incoming edges is not correct, in which case, by symmetry, they all are incorrect. In both cases the EPC cannot become correct after reduction since the remaining edge serves as a proxy for the removed edges.

Similar splits/joins

Figures 5.7 and 5.8 show the rules for two connectors of the same type that directly follow each other. These two connectors can then be merged into one connector of the same type. Note that syntactical restrictions of (reduced) EPCs do not allow for more than one edge between the first and the second connector, since connectors are either a split or a join and never both.

The rule that can be used to reduce similar splits and joins is correctness preserving, since:

1. If the EPC is correct before reduction, then the correct trace containing the first connector must also contain the second connector at a later position. Therefore, it is easy to see that the same trace with only the second connector is a correct trace in the reduced EPC.

2. If any of the input or output edges is incorrect before reduction, they remain incorrect after. Therefore, we can assume that they are correct, in which case the EPC before reduction can only be incorrect if the edge between the connectors is not, which is not possible if all input or output edges are correct.

The rule in Figure 5.8 is correctness preserving for symmetric reasons.
Trivial Loops

Figure 5.9 shows a reduction rule that deals with loops. In this case, it is straightforward that the edge can be removed while preserving correctness, i.e. the edge going back does not introduce new states in the state space. Note that due to syntactical rules the XOR-join connector on top cannot have more than one outgoing arc (i.e. it is a join-connector).

(Complex OR Loops)

Figure 5.10 shows an optional sixth rule. Unlike the others it is not always correctness preserving. The rule assumes that the intended semantics is safe (i.e., no multiple activations of functions and no events that are marked multiple times). This implies that if $t_1$ is an OR-join either the backward arc is taken or any combination of the other arcs, i.e. in this case, we assume that the semantics of the rule is such as in Figure 5.11, after which we can consecutively apply the rule of Figure 5.9 and the rule of Figure 5.5 twice.

It is clear that none of the reduction rules will make the reduced EPC incorrect if the original was correct, and they will not make the reduced EPC correct if the original was incorrect. This implies that if the EPC reduces to the trivial EPC (i.e. an EPC with one initial event, one final event and an edge from the initial event to the final event) then the original EPC is correct. However, since this is rarely the case, we can now proceed with the verification process using this reduced EPC. The result can directly be translated back to the original EPC.

5.3.2 Verification of the reduced EPC

In the previous subsection, we introduced reduction rules for EPCs in such a way that we can use a reduced EPC for the verification process. In this section,
we will translate the reduced EPC into a WF-net. This is also the part of the verification process where user interaction plays an important role. The user has to provide us with possible combinations of initial events. These combinations are then translated into transitions that generate the correct initializations in the WF-net. By calculating the state space, we can then provide the user with all possible combinations of final events that can happen. It is again up to the user to divide those into a set of desired and undesired combinations. Using this information we go to the final stage, where we use a simple coloring algorithm on the state space to decide whether the reduced EPC is correct. This is then translated back to the original EPC.

In this section we will use the EPC shown in Figure 5.4 to describe the overall verification approach in more detail and focus on the second phase of the algorithm where the Petri net mapping for the reduced EPC is used for deciding (relaxed) soundness. The result of applying our reduction rules to that EPC is shown in Figure 5.12. Note that the whole approach described in this section is implemented in the context of the ProM framework (cf. Chapter 8).

**User interaction 1**

As we stated before, the process of EPC verification relies on user interaction. The first point of interaction is when the user has to specify which combinations of initial events can appear to initiate the process described by the EPC. Using this information from the user, we can calculate which initial markings are possible for the Petri net that we will build. If we consider the example from Figure 5.4, then there are two combinations of events that can start the process. The process is initiated, either by the event “Application for MVV entered”, or by both events

---

**Figure 5.12:** The EPC of Figure 5.4 after applying our reduction rules.
“Application for MVV entered” and “Advice application entered”. Note that this knowledge is not present in the EPC. Instead, this is the kind of knowledge that a process designer has about the underlying process. Furthermore, the statement that the process is always started by one of these two possibilities is easily verified on a process log using LTL, as described in Section 4.2. Recall from Section 2.6 that the event “Advice application entered” never occurs in isolation, i.e. it is always followed by “Application for MVV entered”.

Now that we know the possible initializations of the process, we can transform the EPC into a Petri net.

**Translation to Petri net**

Many authors have described algorithms to translate EPCs to Petri nets. In this thesis, we use a modified version of the translation proposed in [62,63]. The translation presented in [62,63] gives a translation into normal Petri nets, whereas we use the same translation algorithm, but enforce the result to be a safe Petri net. The reasons for doing so are twofold. First, events in EPCs either have occurred or not, i.e. only when the occurrence of an event has been dealt with by the system, the same event can occur again for a specific case. Therefore, it makes no sense to map an event onto a place that can contain multiple tokens. Second, the state space of a safe Petri net is more likely to be small, hence, it is more likely that we will be able to construct the state space. Furthermore, enforcing safeness is similar to the interpretation of EPCs in [118] and corresponds to the intuitive nature of EPCs.

To enforce safeness, we duplicate all places with arcs in the reversed direction, i.e., every “normal place” gets a “complement place” that initially contains one token. Each time a token is produced for the normal place, a token is consumed from the complement place. If the token on the normal place is consumed, a token is returned to the complement place. Therefore, the sum of tokens in a normal place and its complement place is always 1. This way the resulting net is forced to be safe. Figure 5.13 shows the translation from a reduced EPC to a safe Petri net. It shows that each connector is translated into transitions, such that these transitions mimic the intended behaviour of the connectors. In other words, the AND-join connector is translated into a transition that requires a token from its two unmarked input places and produces a token on its unmarked output place. Furthermore, each place is accompanied by a counter place that carries a token. This is to ensure that the resulting Petri net is safe, i.e. there is always exactly one token in either a place or its corresponding counter place.

The result of the transformation process for our running example of Figure 5.4 is shown in Figure 5.14. Note that in the layout of the Petri net the reduced EPC from Figure 5.12 is visible. Also note that, by definition, the reduced EPC only contains events and connectors.

Besides the translation, the Petri net in Figure 5.14 also shows how the process
Figure 5.13: Translations of a reduced EPC to a safe Petri net.
can be initialized, i.e. the transition $t_{01}$ represents the occurrence of only the event “Application for MVV entered”, whereas transition $t_{02}$ also initiates the event “Advice application entered”. Note that we do not translate the EPC into a WF-net. Instead, we use a generalized approach towards relaxed soundness, where we say that each transition should occur in at least one firing sequence going from the initial marking to one of a set of final markings. In the first user interaction, the user specified which combinations of events can initialize the EPC. Therefore, in Figure 5.14, we introduced the transitions $t_{01}$ and $t_{02}$, that consume the initial token and activate the corresponding events. All possible final markings reachable from the current marking will be divided into desirable and undesirable markings in the next interaction phase.

**State space generation**

Figure 5.15 shows the state space of the Petri net as shown in Figure 5.14. For ease of reference, we have highlighted the initial state (the open dot). As we discussed before, a state space of a bounded Petri net is always finite. Still, it may be too complex to be constructed (within reasonable time). However, given the fact that the Petri net results from a reduced EPC and that the Petri net is safe by construction, we are able to construct the state space for the vast majority of EPCs found in practice. Moreover, based on our experiences so far (with EPCs up to 100 functions and events), it seems possible to construct the state space without any problems for all EPCs encountered in practice (especially after reduction). Nevertheless, it is also possible to use transition invariants if
Figure 5.15: State space of the Petri net as shown in Figure 5.14

the state space cannot be constructed. We discuss this in detail in Section 5.3.3.

User interaction 2

Now that we have calculated the state space, we are able to provide the user with details about the possible outcomes of the process. In our example, there are many different outcomes that were not intended to be there. The reason for this is the informal definition of the OR-connector in the process. From the description of the verification process in this thesis it will become clear that for the OR-connector you either have both events “Application for MVV entered” and “Positive advice”, or you only have “Application for MVV entered”. However, from the structure of the EPC, this is not clear.

In our case, we leave the exact semantics of the OR-join to the process designer, i.e. the constructed state space contains all possible behaviours. Therefore, from all possible final states, we require the user to select those possible outcomes that correspond to correct executions of the process. In our case, there are 10 possible outcomes, of which the user selects only 6. The outcomes that are not desirable are those that for example show that a residence permit or
MVV were both rejected and accepted at the same time. Note that the question whether this can or cannot happen in practice can be answered using the LTL approach of Chapter 4 again.

The decision process

Finally, we have all the ingredients we need to decide whether the EPC is correct. We have a state space, of which we know all initial states and all allowed final states. The first step toward the final decision is to color everything from the state space that appears on a path from a initial state to one of the allowed final states. The colored part of the state space then describes all the behaviour that the user allows. Then, we look for all transitions that do not have a colored edge in the state space. We call those transitions “not covered”. Figure 5.16 shows the colored state space, where the black parts represent the allowed behaviour.

In principle, transitions that are not covered show that there is possibly incorrect behaviour. Translating this back to an EPC would result in saying that some connectors are used incorrectly. This is indeed the case for connectors of type XOR and AND. However, for join connectors of type OR, we need to perform an
additional step. When people use connectors of type OR, they do not necessarily want all the possible behaviour to appear. As mentioned earlier, the designer knows that the OR-connector in the example EPC will either use the event “Application for MVV entered” or will synchronize and use the events “Application for MVV entered” and “Positive advice”. This knowledge is implicitly taken into account when the allowed final states were selected and therefore, although transition $t_{23}$ is not covered in Figure 5.16, we can tell that this does not indicate a problem, since the transition $t_{23}$ in the Petri net of Figure 5.14 cannot occur in practice.

Using that knowledge, the last step is to remove such transitions from the Petri net, and to color the state space again. Note that we cannot just remove all uncovered transitions relating to OR-joins. Instead, we need to make sure for each of the outgoing edges of the connector at least one transition remains in the WF-net, otherwise, we might disconnect parts of the net making it relaxed sound where it wasn’t before. Therefore, when removing transitions, we need to check whether the covered transitions fulfill this requirement. If not, then no transitions are removed.
Finally, there are three possible outcomes:

**The EPC is correct,** which is the case if the entire state space is colored. If the EPC is correct, then it is always possible to execute the process without ending up in some undesired state.

**The EPC can be correct,** which is the case if the state space is not entirely colored, but all transitions are covered, which is the case for our EPC of Figure 5.4. This result tells the designer that the EPC can be executed, but special care has to be taken to make sure that an execution does not end up in some undesired state.

Consider for example Figure 5.17, where the choices following functions $A$ and $B$ need to be synchronized to get proper execution paths. However, this synchronization is not made explicit in the model.

**The EPC is incorrect,** which is the case when not all transitions are covered. Basically this means that there is some part of the EPC that cannot be executed without running into some undesired behaviour. Consider for example Figure 5.18, where an obvious modeling mistake is depicted.

### 5.3.3 Using transition invariants

For well-structured EPCs (e.g., a 1-to-1 correspondence between splits and joins) the approach discussed in previous section is very effective. The reduction rules immediately remove all the complexity. However, for more complex “spaghetti-like” diagrams the reduced EPC may still be large and exhibit a lot of parallelism. Note that for an EPC with 10 functions in parallel there are at least $2^{10} = 1024$ possible states, and that for an OR-split/join construct with 10 functions to choose from, there are $2^{10} - 1 = 1023$ possible sets of functions that can all be executed in parallel. Hence, if the reduction rules fail to reduce the EPC sufficiently, the construction of the state space may simply take too much time.

To address this problem we can use the concept of *transition invariants*. Note that it might be possible that even the Petri net translation of the reduced EPC is too large to construct (consider a OR-join connector with 24 incoming edges, which would require $2^{24} - 1 = 16,777,215$ transitions.) In that case, verification of the EPC is not possible using our approach. However, one should wonder if such an EPC would be practically useful at all.

A transition invariant assigns a weight to each transition of a Petri net, such that if every transition fires the specified number of times, the initial state is restored, where negative weights correspond to “backward firing” of the transition (i.e. consuming tokens from the output places and producing them in the input places). If none of the weights is negative, then the invariant is called *semi-positive*. The net effect of executing every transition as many times as indicated by a semi-positive transition invariant is zero. Note that a transition invariant does not specify any order and that transition invariants are structural, i.e. they
do not depend on the initial state. Moreover, a transition invariant does not state that it is indeed possible to fire each transition as many times as indicated by the transition weights.

If the state space is too complex to construct, we make use of so-called transition invariants. Recall that every relaxed sound transition is covered by some path from the initial marking \([i]\) to the final marking \([o]\). Therefore, if we add a short-circuiting transition \(t\) that is enabled in \([o]\) and produces the marking \([i]\), a relaxed sound transition should be covered by a semi-positive transition invariant. The benefit of this is that \textit{transition invariants do not require the state space to be constructed, and it is straightforward to show that a transition that cannot be covered by any of the transition invariants, can also not be covered by relaxed soundness.}

For the analysis it suffices to focus on transition invariants where the short-circuiting transition \(t\) has either weight 0 or 1. The former corresponds to a cycle in the WF-net itself, the latter to an execution path from \([i]\) to \([o]\). Using standard algorithms [46] it is possible to calculate a set of minimal semi-positive transition invariants, and to select the relevant ones in a second step. Note that again it is straightforward to generalize this idea to the situation where more than one initial marking exists and more than one final marking is acceptable.

As we stated before, it might be possible that the transitions covered by some transition invariant cannot be executed (because some tokens are lacking). As a result, there may be a transition that is covered by some transition invariant in the short-circuited net, although it is not covered by any execution path from state \([i]\) to state \([o]\). As an example, consider the reduced EPC as shown in Figure 5.19.

The thick path shown in this figure corresponds to a transition invariant in the corresponding short-circuited Petri net. As a result, the two AND-connectors in the middle are covered by transition invariants. However, these connectors are not covered by relaxed soundness (as there is no executable sequence that uses these connectors, because the top connector would block). This example shows that if all transitions are covered by invariants, this is not a sufficient condition for relaxed soundness.

Examples such as the reduced EPC in Figure 5.19 are rare. In most cases invariants and relaxed soundness will yield identical results. Moreover, the designer should realize that all errors discovered using invariants are indeed errors in the EPC, but that some errors may remain undetected. As a result, we feel that our approach can also support the designer of an EPC even if the state space of that EPC is too complex to construct.
5.4 Verification of the SAP reference models

The application of the verification approach presented in this chapter is based on a basic assumption: It assumes that the designer of a model has a good understanding of the actual business process that was modelled, and that he knows which combinations of events may actually initiate the process in real life. Typically, reference models are used by consultants that do indeed have a good understanding of the process under consideration. Besides, they know under what circumstances processes can start, and which outcomes of the execution are desired and which are not.

In Section 1.2, we introduced the SAP reference model. Since this reference model contains many EPCs, our verification approach seems to be well suited for the verification of the SAP reference model. As we did in [80], we focus on the procurement module of the ARIS for MySAP reference model database, since it can be seen as a representative subset of the whole reference model. The procurement module contains several sub-modules and we analyzed all the models from these modules using the approach presented in Section 5.3. Surprisingly, already in the first model (Internal Procurement) there were structural errors. In Figure 5.20, we show a screenshot of the EPC verification plug-in while analyzing the Internal Procurement EPC. It clearly shows that an AND-split is later joined...
Figure 5.20: Fragment of the “Internal Procurement” EPC showing a structural error.

by an XOR-join. Recall Figure 5.18, where we have shown that this is incorrectly modelled. As a result, if this model would not be repaired, payments could be made for goods that were never received. Obviously, this is not desirable. In this case, the problem can easily be repaired. If the XOR-join at the bottom is changed into an AND-join, the model is correct: The EPC cannot be reduced to the trivial EPC but the corresponding WF-net is sound.

The results of our analysis of the whole procurement module are presented in Table 5.1, which contains three columns. The first column shows the name of the module. The second contains the verification result. Note that the SAP reference model does not come with possible sets of initial events required for our analysis. Therefore, we used our own understanding of business processes to define such possible initializations. We use “I” for incorrect models (i.e., the corresponding Petri net is not relaxed sound), “S” for syntactically correct models (i.e., soundness can be decided by just applying the reduction rules), and “C” for semantically correct ones (i.e., sound or relaxed sound). The final column gives short explanation of the error found.

In addition, we applied the analysis using transition invariants. From which, we again were able to conclude that the processes “Internal Procurement” and
Table 5.1: Results for analysis of the procurement module.

<table>
<thead>
<tr>
<th>Module name</th>
<th>Result</th>
<th>Implication of the problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal Procurement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>→ Goods Receipt</td>
<td>I</td>
<td>Payments can be done for goods never received.</td>
</tr>
<tr>
<td>→ Invoice Verification</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>→ Purchase Requisition</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>→ Purchasing</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>→ Warehouse stores</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>Pipeline Processing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>→ Invoice Verification</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>→ Pipeline Withdrawal</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>Materials and External Services</td>
<td></td>
<td></td>
</tr>
<tr>
<td>→ Goods Receipt</td>
<td>I</td>
<td>An invoice can be paid for ordered goods (not services) that have not yet been delivered.</td>
</tr>
<tr>
<td>→ Invoice Verification</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>→ Purchase Requisition</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>→ Purchasing</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>→ Service Entry Sheet</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>→ Transportation</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>→ Warehouse/Stores</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>Procurement on a Consignment basis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>→ Goods Receipt</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>→ Invoice Verification</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>→ Purchase Requisition</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>→ Purchasing</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>→ Warehouse/Stores</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>Procurement via Subcontracting</td>
<td></td>
<td></td>
</tr>
<tr>
<td>→ Goods Receipt</td>
<td>I</td>
<td>An invoice that is received twice will be paid twice.</td>
</tr>
<tr>
<td>→ Invoice Verification</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>→ Provision of Components</td>
<td>C</td>
<td>When materials are simultaneously placed into the stock and removed from it, erroneous behaviour occurs. Operational procedures should avoid this.</td>
</tr>
<tr>
<td>→ Purchase Requisition</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>→ Purchasing</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>→ Transportation</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>→ Warehouse/Stores</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>Return Deliveries</td>
<td></td>
<td></td>
</tr>
<tr>
<td>→ Invoice Verification</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>→ Outbound Shipments</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>→ Quality Notification</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>→ Shipping</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>→ Warehouse</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>Source Administration</td>
<td></td>
<td></td>
</tr>
<tr>
<td>→ Outline Purchase Agreements</td>
<td>C</td>
<td>Redundant objects are present.</td>
</tr>
<tr>
<td>→ RFQ/Quotation</td>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>

"Procurement via Subcontracting" were incorrect, i.e., the use of invariants (rather than constructing the state space) also allowed us to find errors. This strengthens our belief that examples as shown in Figure 5.19 are rare, and that, in general, the technique with transition invariants is applicable in practice. Using this technique, we were also able to conclude that an OR-join in the "Outline Purchase Agreements" (from the Module "Source Administration") could be replaced by an AND-join without changing the allowed behaviour of the process.
5.4.1 Further analysis of the SAP reference models

From the previous section it seems that we can conclude that most errors are made in the higher level models. Using this as a guide, we tried to find problems in the entire reference model. In fact, in the high level EPCs, it is not hard to find these mistakes. These high level models are usually more complex than the lower level models (i.e. they contain more functions, events and connectors). Therefore, errors are more likely to be introduced there. Instead of giving a detailed list of all the errors we have found, we would like to mention three observations that we made during this guided model selection. Note that we did not find these errors using any tools, but merely by manually analyzing the reference model, while being guided by the structure of it.

Mixed process

The first observation is that the errors are not always simple mistakes like an XOR connector that should be AND connector. Surprisingly, some models have logical errors that transcend the level of a single connector and indicate that the process is fundamentally flawed. When making a process model, it is important to be absolutely clear what the process is about once it is instantiated (i.e., what is the definition of the “case” being handled in the EPC). It is important not to mix different processes into one diagram. A nice illustration of this problem is shown in Figure 5.21. This EPC is taken from the “Recruitment” module. The left-hand side of the process is about a vacant position that is created by event “Need to recruit has arisen”. The right-hand side however, is about individual applications. This part is triggered by the event “Enterprise receives application”. Note that there may be many applications for a single position. This makes it unclear whether the process is about filling the vacant position or about dealing with the applications. This leads to all kinds of problems. Figure 5.21 highlights two of these problems. The first problem is that most applications (all except the first one) deadlock because of the AND-join connector that requires a position for each application. The second problem is of a similar nature. Looping back will create a deadlock at the same location. The only way to resolve this problem is to split the EPC in two separate EPCs: one for the vacant position and one for dealing with applications.

Inconsistent models

The second observation is that often one particular initial event is applied in several (sub)models. Take, for example, the event “Deliveries need to be planned”. This event occurs in 15 different EPCs! Every time it occurs, it is joined with the event “Delivery is relevant for shipment”. However, in some models this is done via an XOR-join, and in some models via an AND-join. In Figure 5.22, we show these two events, used in the “Consignment Processing” module, where they are
Chapter 5 Process Model Verification

Problem 1: one position needs to be synchronized with multiple applications. The AND-join cannot cope with this.

Problem 2: The process always loops back. This creates another deadlock (same AND-join).

This event represents the opening of a position ("Need to recruit has arisen").

This event represents a single application ("Enterprise receives application").

**Figure 5.21**: The "Recruitment" EPC with fundamental flaws.
joined by an $XOR$-join. However, in Figure 5.23, we show the same two events in an $AND$-join configuration. Since these two events are always followed by something that refers to transportation, it seems that they should always appear in an $AND$-join configuration. However, only a designer with deep knowledge of the process that is modelled can decide if that is the case.

**Re-use**

The third observation, that shows a common problem, is the effect of re-use. Typically, many different organizations have very similar processes. Therefore, when building reference models, it is a good idea to use one model to create another one. The new model is then changed in such a way that it fits the needs of the new organization better. Figure 5.24 shows a screenshot of the ARIS toolset, showing two models, namely “$Q$-notification with Complaint Against Vendor” on top and “Internal Quality Notification” below. These two models are exactly alike, except that in the top-model, a vendor’s complaint score can be updated. Here, one of the models has been correctly re-used to create the other.

In Figure 5.25, two models are shown for which the re-use was performed incorrectly. The model on the left hand side represents the handling of a “Service Order” and on the right hand side it represents the handling of a “Maintenance Order”. They are very similar, except that the latter does not make a distinction between maintenance at a customer site and at an internal site. Both models however, contain the same mistake: If services are to be entered, the rightmost event called “Services are to be Entered” occurs. However, when that is the case, due to the $XOR$-split in front of it, the function “Overall Completion Confirmation” will never be able to execute. Solving this problem requires a good understanding of the modelled situation since many correct solutions are possible. As both models have the same problem, the process designer should have had the tools to detect this while he derived the second model from the first.
Figure 5.24: Re-use of a correct model.
Figure 5.25: Re-use of an incorrect model.
Summary

The three observations discussed above show that based on a detailed analysis of the EPC reference models we discovered some general problems that should be addressed urgently. It is difficult to take the reference models seriously if they are not 100 percent correct. Note that the reference models are supposed to represent “best practices”. Currently, this is clearly not the case. Note that the ARIS toolset offers the so-called “ARIS Semantic Check”. This involves the checking of rules such as:

- **Path begins with a start event.** This rule checks whether all start objects are of the Event type.
- **Path ends with an event or function.** This rule checks whether all end objects are of the Event or Function type.
- **Function or rule after a joining rule.** This rule checks whether the successor of a joining rule is of the Event or Function type.
- **Event after splitting rule.** This rule checks whether the successors of a splitting rule are of the Event type.
- **No OR or XOR after a single event.** This rule checks whether there is no opening OR or XOR rule (distributor) after events.

Unfortunately, these checks are purely syntactic and will not identify any of the errors mentioned. Therefore, ARIS clearly offers too little verification support and the support that is offered is definitely not aiming at the semantic level. Therefore, we feel that the approach presented in this chapter is a valuable addition to the area of EPC verification, as it allows a process designer to easily identify erroneous parts, which can then be corrected as early as possible.

To conclude this section we would like to mention that we could analyze all EPCs using the approach presented in Section 5.3 without the use of transition invariants, i.e., the state spaces of the models where rather small.

5.5 Conclusion

In this chapter, we introduced an algorithm to verify whether a conceptual model in terms of an EPC is correct, incorrect, or could potentially be correct. Especially the latter statement is new in the area of verification.

In our approach it is assumed that a process designer has knowledge of a process which is not explicitly expressed in the EPC, especially regarding the possible initializations and correct final states of a process. Using this knowledge, the verification approach provides a result related to the relaxed soundness of Petri nets, where the result can be that the EPC is correct, but only under

---

3Note that this text is taking directly from the “Semantic Check” report generated by ARIS.
the explicit assumptions made by the process designer. The choice for relaxed soundness is motivated by the fact that this relates to a human interpretation of conceptual models.

Finally, for the unlikely situation that an EPC is too complex to be analyzed using our approach, we introduced an approximation algorithm, that can find some, but not all errors, as long as the EPC is not too complex to be translated into a safe Petri net.
Chapter 6

Process Discovery

In Chapter 4, we presented an LTL-based approach to objectively verify statements about a process, using a process log, which we introduced in Section 2.2. In this chapter, we go one step further. We show that the same process log that was used to verify statements can be used to gain meaningful insights in the process under consideration, i.e. by deriving a process model from a log.

In this chapter, as shown in Figure 6.1, we introduce several algorithms to-
wards process discovery, i.e. towards discovering a process model from an event log. We start from an event log following Definition 2.2.1. Then, we apply different techniques to construct a WF-net (Definition 2.3.10) or an EPC (Definition 2.4.1).

In the region-based approach of Section 6.2, we directly derive a Petri net from an event log. The other approaches first abstract from the log by deriving log-based ordering relations, such as causal and parallel relations between log events (Section 6.3). These ordering relations are used by the $\alpha$-algorithm of Section 6.4 to again derive a Petri net describing the process, however this time the result is a WF-net. The last algorithm we present in this chapter uses the ordering relations to generate process models for each individual case (i.e. by generating partial orders on the events, Section 6.5) and then aggregates these partial orders in one model (Section 6.6). The result of the latter algorithm can be translated into both WF-nets and EPCs.

The models generated by any of the presented algorithms can for example be used by a process owner to run simulations, or to gain insights in the way cases flow through an organization. Furthermore, these models can serve as a basis for process modeling, in which case the process designer can refine or extend the models manually, to more accurately describe the processes of an organization.

6.1 Introduction

In Chapter 1, we presented process discovery as one of the main challenges of process mining in general. Recall the definition of a process log that we presented in Section 2.2, where we explained that a process log is an ordered set of events. Furthermore, we introduced the concepts of globally complete and locally complete logs, where the first is a log that explicitly contains all possible executions, whereas the latter is only complete in the sense that it explicitly shows if two activities can succeed each other directly, i.e. a local notion of completeness.

In this chapter, we present some process discovery approaches in detail. From the related work on process mining, we know that for all process discovery algorithms, assumptions have to be made on the available information, i.e. whether the logs are complete or not and to what extent they are complete. Therefore, this chapter is organized in the following way.

First, we introduce a process mining technique based on the Theory of Regions that was discussed before in Section 3.4.1. For this algorithm it is important that the input is a globally complete log. Then, in Section 6.3, we present a way to abstract from the log, in such a way that we obtain information about causalities and parallelism between log events. To derive this information, the requirements on the completeness of the log are much less restrictive than for the region based approach. We use this information in sections 6.4, 6.5 and 6.6 to automatically generate process models from a log.
In the chapter, we will use the example introduced in Section 2.6 to illustrate our algorithms. For this example, we took a process log containing both start and complete events for each of the activities. Note that the log is locally complete, but not globally complete (since there is a loop in the process, it cannot be globally complete).

6.2 Region-based Process Discovery

State-based models are widely used for the formal specification and verification of systems. However, the drawback of using such models is the fact that they only represent in which way activities can directly follow each other, i.e. in which state, which activity can lead to which new state. Real causality, concurrency and conflicts between activities are not represented as such, but need to be derived from the sequences of activities presented in the model.

**Definition 6.2.1. (Transition system)**
A labelled state transition system is a triple \((S, \Lambda, \rightarrow)\), where \(S\) is a set of states, \(\Lambda\) is a set of labels, and \(\rightarrow \subseteq S \times \Lambda \times S\) is a ternary relation. If \(p, q \in S\) and \(a \in \Lambda\), \((p, a, q) \in \rightarrow\) is usually written as \(p \xrightarrow{a} q\). This represents the fact that there is a transition from state \(p\) to state \(q\), by executing a transition labelled \(a\). Furthermore, in this thesis, we assume that a transition system is connected.

Describing a process using a transition system is a cumbersome task, especially when the process contains a lot of parallel behaviour. This is not desirable for any business process designer and therefore, there is a need to transform state-based models into models that do represent concurrency and choices directly, such as Petri nets.

6.2.1 Theory of Regions

For the purpose of deriving Petri nets from transition systems, the concept of regions was introduced in [83], where these regions served as intermediate objects, between a transition system on the one hand and a Petri net on the other hand. This process, to go from a transition system with anonymous states to a Petri net, is called *synthesis* and the goal is to generate a Petri net that exactly mimics the behaviour of the state-based model.
Definition 6.2.2. (Region in a transition system)
Let $TS = (S, \Lambda, \rightarrow)$ be a transition system. We say that $R \subseteq S$ is a region of $TS$ if and only if for all $(p, a, q), (p', a, q') \in \rightarrow$ holds that:

- if $p \in R$ and $q \notin R$ then $p' \in R$ and $q' \notin R$, i.e. all transitions labelled $a$ exit the region, and we say that $R$ is a pre-region of $a$,
- if $p \notin R$ and $q \in R$ then $p' \notin R$ and $q' \in R$, i.e. all transitions labelled $a$ enter the region, and we say that $R$ is a post-region of $a$.
- if $(p \in R) \Leftrightarrow (q \in R)$ then $(p' \in R) \Leftrightarrow (q' \in R)$, i.e. all transitions labelled $a$ do not cross the region.

It is easy to see that there are two trivial regions, i.e. $\emptyset$ and $S$ are regions. The collection of all regions of a transition system $TS$ is called $\mathcal{R}(TS)$. A region $R \in \mathcal{R}(TS)$ is said to be minimal if and only if for all $R' \subset R$ with $R' \neq \emptyset$ holds that $R' \notin \mathcal{R}(TS)$. The set of all minimal regions is denoted by $\mathcal{R}_{\text{min}}(TS)$. Furthermore, it is important to note that regions do not depend on one label $\alpha$, i.e. they always depend on the entire set of labels in the transition system.

When reasoning about regions and events, we use a generic notation for retrieving pre-regions and post-regions of events and entering and exiting activities of regions.

Definition 6.2.3. (Pre-set and post-set for events and regions)
Let $TS = (S, \Lambda, \rightarrow)$ be a transition system and $a \in \Lambda$ an activity label. The pre-region set and the post-region set of $a$ are the sets of regions defined as follows:

- $^{TS}a = \{R \in \mathcal{R}(TS) | \forall(s, a, s') \in \rightarrow: s \in R \land s' \notin R\}$ and
- $^{TS}a = \{R \in \mathcal{R}(TS) | \forall(s, a, s') \in \rightarrow: s \notin R \land s' \in R\}$

Given a region $R \in \mathcal{R}(TS)$, $^{TS}R = \{a \in \Lambda | R \in ^{TS}a\}$ and $R^{TS} = \{a \in \Lambda | R \in ^{TS}a\}$. Note that if the context is clear, we omit the superscript $TS$, i.e. $\circ R$ and $R\circ$.

As we explained before, in process discovery, we do not have a state-based model to start with. However, the Theory of Regions can still be applied. Assume that we have a process log and we want to obtain a Petri net that exactly describes what we have seen in the log, i.e. the Petri net should be able to generate each process instance in the log and nothing more. Obviously, if our log would be a state-based model, the Theory of Regions would apply directly. However, there is one problem with our logs:

Our process instances are sequences of events and do not carry any state information, so there is no relation between different process instances, i.e. we cannot tell which sequences of activities lead to similar global states.

To solve the issue above, we need to make assumptions about the process log. In the remainder of this section, we present an iterative algorithm [75], that makes such assumptions.
6.2.2 Region-based Process Discovery

Since the Theory of Regions, when applied to a state-based model, results in a Petri net, of which the behaviour exactly matches the given model (i.e. trace equivalence), process discovery would only make sense if we have seen all possible behaviour. Therefore, we say that region-based process discovery is only useful in the situation where our process log is globally complete, i.e. as explained in Section 2.2.5, it shows all possible behaviour of the process.

The assumption that our log is globally complete alone is not enough. We still have the problem stated in the previous section that we need to identify global states. To tackle this problem, we take a naive approach. We assume that the initial state is known and the same for all cases, i.e. we identify some notion of a global “initial state”.

After deriving a Petri net using the Theory of Regions, we need to be able to identify the initial state in the Petri net, i.e. we need to specify the initial marking. By assuming that all cases start with the same unique activity, which always appears as the first audit trail entry in each process instance, we can easily do so in the resulting Petri net, i.e. we mark the input places of the corresponding transition. Requiring that each process instance starts with the same activity might seem restrictive, but it is trivial to add such an activity to each process instance of a log.

The first step in region based process mining is to convert the process log into a transition system. This translation is rather trivial, and Figure 6.2 gives an example of such a transition system of the process log with three process instances shown in Table 6.1. The transitions in Figure 6.2 correspond to some of the functions of the EPC of our running example.

**Definition 6.2.4. (Trace to transition system)**

Let $T$ be a set of log events and let $W$ be a globally complete process log over $T$, i.e., $W \in \mathcal{P}(T^*)$. Furthermore, let $\sigma \in W$ be an arbitrary trace. We define $TS(\sigma) = (S_\sigma, \Lambda_\sigma, \rightarrow_\sigma)$ to be a transition system, such that:

- $S_\sigma = \{(\sigma, i) \in \{\sigma\} \times \mathbb{N} \mid 0 \leq i < |\sigma|\} \cup \{(W, -1)\}$, i.e. the set of states consists of all indices in all process instances, as well as a global state $(W, -1)$, which is the initial state,
- $\Lambda_\sigma = \{t \mid t \in \sigma\}$, i.e. the set of labels is the set of log events,
- $\rightarrow_\sigma = \{((\sigma, i), (\sigma, i + 1)) \in S \times T \times S\}$, i.e. the trace is represented as a sequence of state transitions, starting in the initial state. The transitions between each two states are made by the activity at the given position in the trace.

Using the translation from a single trace to a transition system, we can translate an entire log to a transition system. Again this is a straightforward translation and the result of our example log is shown in Figure 6.2.
Table 6.1: Example event log

<table>
<thead>
<tr>
<th>Case</th>
<th>Event</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Decide on MVV</td>
<td>12:00</td>
</tr>
<tr>
<td>2</td>
<td>Decide on MVV</td>
<td>12:02</td>
</tr>
<tr>
<td>1</td>
<td>Send MVV rejection letter</td>
<td>12:03</td>
</tr>
<tr>
<td>3</td>
<td>Decide on MVV</td>
<td>12:43</td>
</tr>
<tr>
<td>2</td>
<td>Send documents to Town Hall</td>
<td>12:46</td>
</tr>
<tr>
<td>2</td>
<td>Send MVV acceptance letter</td>
<td>12:47</td>
</tr>
<tr>
<td>3</td>
<td>Send MVV acceptance letter</td>
<td>13:11</td>
</tr>
<tr>
<td>3</td>
<td>Send documents to Town Hall</td>
<td>13:15</td>
</tr>
</tbody>
</table>

Figure 6.2: Transition system of Table 6.1.

Definition 6.2.5. (Process log to transition system)

Let $T$ be a set of log events and let $W$ be a globally complete process log over $T$, i.e., $W \in \mathcal{P}(T^*)$. We define $TS(W) = (S, \Lambda, \rightarrow)$ to be a transition system, such that:

- $S = \bigcup_{\sigma \in W} S_{\sigma}$, i.e. the union over the states of the transition system translations of each individual trace,
- $\Lambda = T$, i.e. the set of labels is the set of log events,
- $\rightarrow = \bigcup_{\sigma \in W} \rightarrow_{\sigma}$, i.e. each trace is represented as a sequence of state transitions, starting from the common initial state. The transition between each state is made by the activity at the given position in the trace.

Once a process log is converted into the transition system, we can use the Theory of Regions to generate a Petri net from it. The idea is that each log event from the log is represented as a transition in the transition system and therefore, we know that each log event has a set of pre-regions and post-regions in the transition system, which may represent the input and output places in the Petri net. Note that many algorithms in the area of Petri net synthesis have been developed, to come up with smaller Petri nets in terms of the number of places, or with free-choice Petri nets by introducing multiple transitions with the same name [40]. However, in this thesis, we present the most basic algorithm, presented in [68] and [83], where minimal regions are translated into places.
Section 6.2 Region-based Process Discovery

Definition 6.2.6. (Region-based mining algorithm)
Let $T$ be a set of log events, $W$ a process log over $T$ with $t_i \in T$ the initial event in $W$ and $TS(W) = (S, \Lambda, \rightarrow)$ to be the transition system generated from that using Definition 6.2.5. We define a marked Petri net $\varphi = ((P, \Lambda, F), M_0)$, synthesized from $TS(W)$, as follows:

- $P = \mathcal{R}_{\text{min}}(TS(W))$, i.e. each place corresponds to a minimal region in the transition system,
- $F = \{(R, t) \in P \times \Lambda \mid R \in \circ t\} \cup \{(t, R) \in \Lambda \times P \mid R \in \to t\}$, i.e. each transition is connected to an input place if the corresponding region is a pre-region and to an output place if the corresponding region is a post-region,
- $M_0 = \bullet t_1$, i.e. the initial transition has all its input places marked, with one token.

In Figure 6.3, we show the 6 minimal regions of the example transition system and their translation into a Petri net. Note that this Petri net is not a WF-net (there are multiple output places). In general, the result of the region-based approach presented here is not a WF-net and due to the fact that each case might end in a different final state, there is no generic algorithm for translating the Petri net into a WF-net.

It is important to note that this algorithm presented in Definition 6.2.6 has been proven to work on so-called elementary transition systems only, i.e. on transition systems where each two different states have to belong to two different sets of regions, and if a state $s$ is included in all pre-regions of an event $e$, then that event $e$ must be enabled in $s$. Our translation from process logs to transition
systems does not enforce this, i.e. the resulting transition system is not necessarily an elementary transition system. However, for now, we assume that this is the case and at the end of Subsection 6.2.3, we show an example where the transition system is not elementary and the approach still works.

The definition of Petri net synthesis results in a Petri net of which the state space is bisimilar to the original transition system \([40,68,83]\). Without going into details about bisimilarity, it is enough to realize that this implies that the state space is trace equivalent with the original transition system and hence the Petri net can generate exactly those traces we observed in the log and nothing more.

Although the algorithm presented in Definition 6.2.6 only considers minimal regions, there are many ways to improve the result \([40]\). Another way of improving the result is by making a better transition system out of the log. However, that would require us to have state information in the log, or to estimate such information, for example by saying that the same sequences of events in different traces lead to the same global state. In \([23,155]\), several algorithms are presented that do so. However, under the assumption of a globally complete log, such estimations should never introduce new traces that we did not observe in the log.

Furthermore, the approach presented in this section is rather complex in terms of computation time. The transition system that is generated from the log is exactly as big as the log, which can be a problem for real-life applications. In the worst case, the set of regions of this transition system is exponential in the size of the transition system and thus the required calculation time is exponential in the size of the log. One way to deal with this problem is to build the actual set of regions iteratively, i.e. by adding instances on the fly.

### 6.2.3 Iterative Region Calculation

Our naive approach towards using the Theory of Regions in the context of process discovery has the problem that it requires the entire transition system to be built in memory. Since process logs can easily contain thousands of cases, referring to hundreds of events each, the resulting transition system may be too large to be stored in computer memory. Therefore, in this section, we use the structure of our transition system to introduce an iterative approach.

Recall that the transition system we built from the process logs in Definition 6.2.5 is basically a straightforward sum over a set of sequential transition systems with a known initial state. In this section, we show that if we have two transition systems with equal initial states, we can calculate the regions of the combination of these two transition systems without constructing the transition system itself. Finally, after iterating over all traces individually, we translate the resulting set of regions to a Petri net. This process is shown in Figure 6.4

In Definition 6.2.2, we defined a region as set of states, such that each transition in the transition system either \(\text{enters, exits}\) or \(\text{does not cross}\) the region. Since our aim is to obtain the regions of an unknown transition system by com-
bining the regions of two smaller transition systems with similar initial and final states, we introduce the concept of compatible transition systems and compatible regions.

**Definition 6.2.7. (Compatible transition systems)**

Let $TS_1 = (S_1, \Lambda_1, \rightarrow_1)$ and $TS_2 = (S_2, \Lambda_2, \rightarrow_2)$ be two transition systems. We say that $TS_1$ and $TS_2$ are compatible if and only if:

- $|S_1 \cap S_2| = 1$, i.e. there is only one common state and,
- For $s \in S_1 \cap S_2$ holds that there is no $p \in S_1 \cup S_2$ and $\alpha \in \Lambda_1 \cup \Lambda_2$ with $(p, \alpha, s) \in \rightarrow_1 \cup \rightarrow_2$, i.e. the common state is an initial state.

Two transition systems are compatible if they share a common initial state, but no other states. It is easily seen that the translation of two traces from one process log to two transition systems yields two compatible transition systems.

For compatible transition systems, we define compatible regions.

**Definition 6.2.8. (Compatible regions)**

Let $TS_1 = (S_1, \Lambda_1, \rightarrow_1)$ and $TS_2 = (S_2, \Lambda_2, \rightarrow_2)$ be two compatible transition systems. Let $R_1 \in \mathcal{R}(TS_1)$ and $R_2 \in \mathcal{R}(TS_2)$. We say that $R_1$ is compatible with $R_2$, denoted by $R_1 \leftrightarrow R_2$ if and only if

- $(TS_1 \circ R_1 \setminus TS_2 \circ R_2) \cap \Lambda_2 = \emptyset$,
- $(TS_2 \circ R_2 \setminus TS_1 \circ R_1) \cap \Lambda_1 = \emptyset$,
- $(R_1 TS_1 \circ R_2 TS_2) \cap \Lambda_2 = \emptyset$,
- $(R_2 TS_2 \circ R_1 TS_1) \cap \Lambda_1 = \emptyset$,
- $\forall s \in S_1 \cap S_2$ holds $s \in R_1$ if and only if $s \in R_2$.

A region of one transition system is compatible with a region of another transition system if all transitions that enter the first region also enter the second region, or do not appear at all in the second transition system. Similarly, this has to hold

![Figure 6.4](image-url)
for all exiting transitions. Furthermore, if a common state appears in one region, it should appear in the other region as well.

The first step towards our iterative approach is to define how to add two compatible transition systems, where we use the earlier translation of a process log to a transition system, i.e. Definition 6.2.5.

**Definition 6.2.9. (Adding compatible transition systems)**

Let $TS_1 = (S_1, \Lambda_1, \rightarrow_1)$ and $TS_2 = (S_2, \Lambda_2, \rightarrow_2)$ be two compatible transition systems. We define $TS = (S, \Lambda, \rightarrow)$ as the sum of the two transition system, denoted by $TS = TS_1 \oplus TS_2$, such that:

- $S = S_1 \cup S_2$, i.e. the union over the states of both transition systems,
- $\Lambda = \Lambda_1 \cup \Lambda_2$, i.e. the union over the labels of both transition systems,
- $\rightarrow = \rightarrow_1 \cup \rightarrow_2$, i.e. the union over the transitions of both transition systems.

**Property 6.2.10. (Adding yields a compatible transition system)**

Let $TS_1 = (S_1, \Lambda_1, \rightarrow_1)$, $TS_2 = (S_2, \Lambda_2, \rightarrow_2)$ and $TS_3 = (S_3, \Lambda_3, \rightarrow_3)$ be three pairwise compatible transition systems and let $TS = (S, \Lambda, \rightarrow) = TS_1 \oplus TS_2$ be the sum over the first two. $TS$ is compatible with $TS_3$.

**Proof.** For $TS$ and $TS_3$ to be compatible, we need to show that there is one common initial state. Let $s_1 \in S_1 \cap S_2$ be the common initial state of $TS_1$ and $TS_2$. Since there is only one initial state, we know that this is the initial state of both $TS_3$ and of $TS$, hence $TS$ and $TS_3$ share one initial state and hence $TS$ and $TS_3$ are compatible. \qed

It remains to be shown that we are able to calculate the set of regions of the sum of two transition systems from the sets of regions of the transition systems we are adding.

**Property 6.2.11. (Region summation is possible)**

Let $TS_1 = (S_1, \Lambda_1, \rightarrow_1)$ and $TS_2 = (S_2, \Lambda_2, \rightarrow_2)$ be two compatible transition systems. Furthermore, let $TS = (S, \Lambda, \rightarrow) = TS_1 \oplus TS_2$ be the sum over both transition systems. We show that $\mathcal{R}(TS) = D$, where $D = \{R \in \mathcal{P}(S) \mid \exists R_1 \in \mathcal{R}(TS_1) \exists R_2 \in \mathcal{R}(TS_2) R_2 \leftrightarrow R_1 \land R = R_1 \cup R_2\}$

**Proof.** Assume that $R = R_1 \cup R_2$ with $R_1 = R \cap S_1$ and $R_2 = R \cap S_2$. It is easy to see that $R_1 \in \mathcal{R}(TS_1)$ and $R_2 \in \mathcal{R}(TS_2)$. The question remains whether $R_1 \leftrightarrow R_2$, however since $R \in \mathcal{R}(TS)$, we know that all events entering $R$ also enter $R_1$, or do not appear in $\Lambda_1$, i.e. $\tau^S \circ R \cap \Lambda_1 = \tau^{TS_1} \circ R_1$. Similarly, all events entering $R$ also enter $R_2$, or do not appear in $\Lambda_2$, hence $R_1 \leftrightarrow R_2$. \qed

In Property 6.2.11, we have shown that if we have two compatible transition systems $TS_1$ and $TS_2$, then we can calculate the regions of the sum of $TS_1$ and $TS_2$, using the regions of the individual transition systems. If we have a large collection of compatible transition systems, adding two of them up to a new one yields a transition system which is compatible with all others as shown
Section 6.2 Region-based Process Discovery

in Property 6.2.10. Since we have provided an algorithm in Definition 6.2.4 to translate all traces of a process log to a collection of compatible transition systems, we have constructed an iterative algorithm for the calculations presented in Subsection 6.2.2.

With Property 6.2.12, we conclude this subsection. Property 6.2.12 shows that the iterative algorithm yields the same transition system as Definition 6.2.5. Combining this with Property 6.2.10 and Property 6.2.11 leads to the conclusion that the set of regions resulting from our iterative approach indeed yields the set of regions of the transition system obtained by applying Definition 6.2.5 directly.

**Property 6.2.12. (Iterative approach works)**

Let $T$ be a set of log events, let $W$ be a globally complete process log over $T$, i.e., $W \in \mathcal{P}(T^*)$ and let $TS(W) = (S, \Lambda, \rightarrow)$ be a transition system. Furthermore let $TS' = \bigoplus_{\sigma \in W} TS(\sigma) = (S', \Lambda', \rightarrow')$ be the transition system gained by adding all transition systems corresponding to each instance. We show that $TS = TS'$.

**Proof.**

- From Definition 6.2.5, we know that
  \[ S = \bigcup_{\sigma \in W} S_{\sigma} \text{ and } \rightarrow = \bigcup_{\sigma \in W} S_{\sigma}(\Lambda_{\sigma}, \rightarrow_{\sigma}). \]

- From Definition 6.2.9, we know that
  \[ S' = \bigcup_{\sigma \in W} S_{\sigma} \text{ and } \rightarrow' = \bigcup_{\sigma \in W} S_{\sigma}(\Lambda_{\sigma}, \rightarrow_{\sigma}), \]

- From Definition 6.2.5, we know that $\Lambda = T$. Furthermore, from Definition 6.2.9, we know that $\Lambda' = \bigcup_{\sigma \in W} \{t \mid t \in \sigma\}$. Assuming that for all $t \in T$ there is at least one $\sigma \in W$, such that $t \in \sigma$, we know that $\Lambda = \Lambda'$.

Note that by showing that the iterative approach works in Property 6.2.12, we can apply the synthesis algorithm of Definition 6.2.6 on the set of regions that results from the iterative algorithm.

Recall that the algorithm presented in Definition 6.2.6 has been proven to work on so-called elementary transition systems only. Elementary transition systems are such that each two different states belong to two different sets of regions, and if a state $s$ is included in all pre-regions of an event $e$, then that event $e$ must be enabled in state $s$.

Even though our algorithm does not guarantee that the resulting transition system is an elementary transition system, the result can still be very insightful. Consider for example a log with three cases, i.e. case 1: $A,B,D$; case 2: $A,C,D$ and case 3: $A,B,C,D$. Figure 6.5 shows why the log with three instances does not result in an elementary transition system. The three highlighted states all appear in all pre-regions of transitions $B, C$ and $D$, i.e. these three different states do not belong to different sets of regions. Therefore, no region-based algorithm.
can distinguish between these states and in all the highlighted states \((\text{case1}, 0)\), \((\text{case2}, 0)\) and \((\text{case3}, 0)\) the transitions \(B\), \(C\) and \(D\) are assumed to be enabled. Hence, in the resulting Petri net, there is only one marking that enabled both \(B\) and \(C\) and \(D\).

In the work of Cortadella et al. [52–54], there are many proposed algorithms to resolve the problems with non-elementary transition systems. However, at this point, it is unknown whether these algorithms can be made iterative in the way presented in this section and hence the complexity of these algorithms when dealing with real life logs becomes a problem.

### 6.2.4 Complexity

In Subsection 6.2.5, we mentioned that the synthesis algorithm of Definition 6.2.6 requires the full transition system to be built in memory. The space required to do so is obviously linear in the size of the log, i.e. the space requirement is linear in the size of the log. However, our experience with process mining has shown that typical process logs found in practice are too big to be stored in memory.

In process mining, all algorithms have to make a trade-off between computation time on the one hand and space requirements on the other. In [128] for example, a genetic-algorithm approach toward process mining is presented, where the algorithm scales linearly in the size of the log, i.e. if a log contains twice the number of cases, but the same number of different events, the algorithm is twice as slow, but requires the same amount of memory.

When a full transition system is stored in memory, the calculation of only minimal regions is simpler then in our iterative approach, i.e. using a breadth-first search, all minimal regions could be found, without considering larger regions. Our iterative approach requires all regions to be calculated, after which the minimal regions need to be found in the set of all regions. Therefore, the computation time is larger with our iterative approach. In fact, it is not too hard to see that
our iterative approach scales linearly in the size of the log, i.e. by adding more cases, while keeping the number of different activities equal, we need more processing time, but not more memory, which is commonly accepted as a desirable result for process mining algorithms.

6.2.5 Mining Quality

It is straightforward to see that the approach presented in this section indeed leads to a Petri net and due to the fact that the state space of that net is bisimilar to the original transition system (if that transition system is elementary), the state space is also trace equivalent with the transition system. Furthermore, since the transition system contains exactly those traces that are represented in the event log, the Petri net can reproduce the event log exactly, even if the log is not globally complete, but in that case the result is too restrictive. However, there are several problems which make this approach less useful in practice.

Although the Petri net resulting from applying the Theory of Regions to a process log can reproduce that log, there is no clear terminating state. In fact there are many so-called dead markings, which all correspond to the final state of one or more traces. Even adding an extra final activity to each trace will not solve this, since this only guarantees that the output places of the corresponding transition will be marked. However, it does not enforce that no other places are also marked. This violates the requirement of proper completion in soundness and furthermore makes the resulting Petri net hard to understand. Since most process designers are interested in having a sound process model, we consider this to be a great limitation.

The biggest problem of the Theory of Regions in the context of process discovery is that the resulting Petri net mimics the behaviour of the log exactly, i.e. the Theory of Regions was developed to generate a compact representation of a transition system in terms of a Petri net, which is very different from the goal of process mining, i.e. to generate a model that can reproduce the log, but allows for more behaviour. The latter is not possible with the Theory of Regions and therefore the Theory of Regions is only directly applicable if the log is globally complete.

Furthermore, if a process contains loops, then this implies that (1) a log can never be globally complete and that (2) cases lead to identical global states at several points during execution. Since our approach does not identify these global states, the resulting Petri net will never contain loops, which is unlikely to correspond to real life, where loops are often contained in processes. In [23,155], an approach to process mining is proposed that uses the Theory of Regions and where global states are derived from the log. However, the approach presented there has the downside that the entire transition system has to be built in memory.

To overcome the problems relating to the size of the log (or the size of the derived transition system), as well as the requirement of a globally complete log,
we consider a different approach towards process discovery. In the next section, we go back to the event log and we analyze that log in such a way that we derive information about the operational process, without constructing a model for it yet. Instead, we first abstract from the log and later, in sections 6.4, 6.5 and 6.6, we use this abstraction to generate a process model.

6.3 Log Abstraction

In process discovery, we typically see that logs are not globally complete, or in the context of loops will never be globally complete. Instead, process logs, at best, show the common practice of executing activities in a process. Therefore, we argue that the first papers really addressing the problem of process discovery appeared around 1995, when Cook et al. [47–51] started to analyze recorded behaviour of processes in the context of software engineering, acknowledging the fact that information was not complete and that a model was to be discovered that reproduces at least the log under consideration, but it may allow for more behaviour.

Requiring that the discovered process model reproduces at least the log under consideration, intuitively leads to the idea of abstracting from the information in the log before applying a process discovery algorithm. Therefore, in this section, we introduce some ways to abstract from the log to so-called log-based ordering relations. Later, in sections 6.4, 6.5 and 6.6, we present process discovery algorithms that use the ordering relations to generate a model of the log, i.e. that do the actual process discovery.

6.3.1 Log-based Ordering Relations

In order to be able to discover a process model from a log, we at least need to make some assumptions, therefore we assume our logs to be locally complete, instead of globally complete. Recall from Section 2.2.5 that this implies that we have not seen all possible sequences of events, but if two activities can be performed right after each other, we have observed that at least once. For example, a process with three activities A, B and C in parallel, requires six sequences to be globally complete, i.e. \( ABC, ACB, BAC, BCA, CAB \) and \( CBA \). However if we only have observed \( ABC, ACB, CBA, CAB \) it is already locally complete, since we have seen all possible pairs of subsequent activities \( AB, BA, AC, CA, BC \) and \( CB \) at least once. The more tasks are in parallel, the greater the difference in minimal size is between the locally and globally complete log and therefore the more likely it is that the log is locally complete. Note that this not only applies to parallelism, but also to independent choices. In this section, we exploit the local completeness property, i.e. we take a process log as input and we introduce abstract information about the underlying process.
The process modelling languages used in this thesis are such that they explicitly express causal dependencies between activities, as well as moments of choice and possible parallelism. More specifically, using process logs, we determine the following relations between two activities:

- $→$, representing a causal dependency between them,
- $∥$, representing a parallel relation between them, and
- $#$, representing the fact that there is no parallel or causal relation.

Formally, we define the properties that have to be satisfied by any ordering relations as follows:

**Definition 6.3.1. (Ordering relations)**

Let $T$ be a set of log events and let $W$ be a locally complete process log over $T$, i.e., $W ∈ \mathcal{P}(T^*)$. We define ordering relations, $→_W \subseteq T \times T$, $∥_W : T \rightarrow T$ and $\#_W : T \rightarrow T$ over $T$, such that for $t_1, t_2 ∈ T$:

- $t_1∥_W t_2$ implies that $t_2∥_W t_1$ and not $t_1#$ $t_2$ and not $t_2 →_W t_1$, i.e. if two events are in parallel, there is a relation and they do not have a causal dependency.
- $t_1#$ $t_2$ implies that $t_2#$ $t_1$ and not $t_1∥_W t_2$ and not $t_1 →_W t_2$ and not $t_2 →_W t_1$, i.e. if two events do not have a relation, then they do not have a causal or parallel relation.
- $t_1 →_W t_2$ implies that not $t_1∥_W t_2$ and not $t_1#$ $t_2$, i.e. if two events are causally dependent, they have a relation and are not in parallel.

Note that in contrast to $∥_W$ and $\#_W$, $→_W$ is not a function, hence it is possible to have both $t_1 → t_2$ and $t_2 → t_1$, i.e. $t_1$ both causally succeeds and precedes $t_2$.

Consider Figure 6.6, where we graphically show a process log containing three process instances and six activities, $A$, $B$, $C$, $D$, $E$ and $F$. Furthermore, the log shows both the starting and the completing of each activity, thus leading to twelve log events, i.e. $A_{\text{start}}$, $A_{\text{complete}}$, $B_{\text{start}}$, etc. Table 6.2 shows a collection of ordering relations for this log, i.e. it shows that $A_{\text{start}}$ is causally followed by $A_{\text{complete}}$ and that $B_{\text{complete}}$ is in parallel with $C_{\text{complete}}$. The way in which these basic ordering relations are derived from a process log is formalized in Definition 6.3.2.

Note that in Table 6.2, we use “s” to denote start events and “c” to denote complete events. Furthermore, the relations presented in that table are not arbitrarily chosen. Instead, the ordering relations presented in the table are called
Table 6.2: Basic ordering relations.

<table>
<thead>
<tr>
<th></th>
<th>As</th>
<th>Ac</th>
<th>Bs</th>
<th>Bc</th>
<th>Cs</th>
<th>Cc</th>
<th>Ds</th>
<th>Dc</th>
<th>Es</th>
<th>Ec</th>
<th>Fs</th>
<th>Fc</th>
</tr>
</thead>
<tbody>
<tr>
<td>As</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>Ac</td>
<td>←</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>Bs</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>Bc</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>Cs</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>Cc</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>Ds</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>→</td>
<td>#</td>
<td>#</td>
<td>→</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>Dc</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>Es</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>Ec</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>Fs</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>Fc</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
<td>←</td>
</tr>
</tbody>
</table>

The idea behind the ordering relations of Definition 6.3.2 is simple. If two activities appear right after each other in one specific order, but never in the other order, we assume them to be causally related. If two activities appear in any order, we assume them to be in parallel and otherwise we assume them to be unrelated.

Consider the process log of Figure 6.6 again. It is not hard to describe the underlying process in words, i.e. the process starts with the execution of activity A. Then, activities B and C are executed in parallel, followed by either one of the activities D, E or F. Activities E and F are final activities, but after the execution of activity D, activities B and C are executed again, hence there is no relation between activity D on the one hand and E and F on the other hand.

Consider Table 6.2, where we presented the basic ordering relations for the example of Figure 6.6. Although intuitively, we say that activities B and C are executed in parallel and can be succeeded by activities D, E and F, this does not show in that table, i.e. $B_{start}$ is not in parallel with anything, while we expect...
it to be in parallel with both $C_{\text{start}}$ and $C_{\text{complete}}$. This shows that we need extensions to the basic relations, where we make use of our knowledge of process logs.

### 6.3.2 Extension of Log-based Ordering Relations

In sections 6.4, 6.5 and 6.6, we present algorithms that use the ordering relations for the generation of a process model. Therefore, it is of great importance to calculate the ordering relations as precisely as possible.

As we saw from the definition of local completeness, the more activities are in parallel, the greater the difference becomes between a locally complete log and a globally complete log. Therefore, when it comes to determining which activities are concurrent, it is of the utmost importance to use as much information as possible. For this purpose, we refine the log-based ordering relations in two ways. First, we introduce a concept taken from [33, 34], to find causal dependencies between activities that so far appeared to be in parallel and thereafter we make use of the transactional model presented in Section 2.2.2 to even further refine the relations found.

**Short Loops**

The original log-based ordering relations do not consider anything more than the direct succession, i.e. through relation $\succ_M$. However, this leads to problems in the context of short loops, as discussed in [26]. Short loops are loops of activities of length one or two. Using the ordering relations defined so far, we would consider such activities to be in parallel, while in fact they are not. Therefore, we take the approach presented in [33, 34] where we look two steps ahead and if we see two log events $A$ and $B$ that appear in the log as both $A;B;A$ and as $B;A;B$ then we say that there is a causal dependency from $A$ to $B$ and the other way around, thus implying that they are no longer in parallel.

The solution to the problem of length one loops also presented in [33, 34] is specific to one algorithm, since it requires post-processing of results produced that algorithm. Therefore, in this thesis, we will take a slightly different approach and we say that if log event $A$ can directly succeed itself in a log, then there is a causal dependency from that event to itself and not a parallel relation. Since this assumption does not influence events that do not follow themselves directly, we can safely say that a log event is always causally related to itself, hence the diagonal of Table 6.3 only shows causal dependencies.

To summarize, to deal with short loops, we change the basic ordering relations in the following way:
Table 6.3: Extended ordering relations.

<table>
<thead>
<tr>
<th></th>
<th>A_s</th>
<th>A_c</th>
<th>B_s</th>
<th>B_c</th>
<th>C_s</th>
<th>C_c</th>
<th>D_s</th>
<th>D_c</th>
<th>E_s</th>
<th>E_c</th>
<th>F_s</th>
<th>F_c</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_s</td>
<td>←</td>
<td>→</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>A_c</td>
<td>←</td>
<td>→</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>B_s</td>
<td>#</td>
<td>←</td>
<td>→</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>B_c</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>→</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>C_s</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>→</td>
<td>→</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>C_c</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>→</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>D_s</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>→</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>D_c</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>E_s</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>→</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>E_c</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>F_s</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>→</td>
</tr>
<tr>
<td>F_c</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>→</td>
</tr>
</tbody>
</table>

Definition 6.3.3. (Extended log-based ordering relations)

Let $T$ be a set of log events and let $W$ be a locally complete process log over $T$, i.e., $W \in \mathcal{P}(T^*)$. Let $a, b \in T$ and let:

- $a >_W b$ if and only if there is a trace $\sigma \in W$ and $i \in \{0, \ldots, |\sigma| - 2\}$ such that $\sigma_i = a$ and $\sigma_{i+1} = b$,
- $a <_W b$ if and only if there is a trace $\sigma \in W$ and $i \in \{0, \ldots, |\sigma| - 3\}$ such that $\sigma_i = a$ and $\sigma_{i+1} = b$ and $\sigma_{i+2} = a$,
- $a \circ_W b$ if and only if $a <_W b$ and $b <_W a$,

We define the extended ordering relations as:

- $a \rightarrow^\text{ext} W b$ if and only if $a = b$ or ($a >_W b$ and ($b \not< W a$ or $a \circ_W b$)), and
- $a \parallel^\text{ext} W b$ if and only if $a \neq b$ and $a >_W b$ and $b >_W a$ and $a \not< W b$, and
- $a \#^\text{ext} W b$ if and only if $a \neq b$ and $a \not< W b$ and $b \not< W a$.

Again, it is easy to see that these relations are indeed ordering relations according to Definition 6.3.1.

Our example log of Figure 6.6 clearly does not contain short loops. Therefore the only change to the ordering relations of Table 6.2 is the fact that the diagonal now shows causal dependencies.

So far, the ordering relations that we have defined are based purely on the sequential structure of the log. However, as we have seen in Section 2.2, process logs contain more information than just the sequences of events. In fact, most process logs contain the notion of events relating to the same activity, i.e. scheduling versus completion of an activity. Therefore, we use this information to further refine the ordering relations.
Using Time and the Transactional Model

In Subsection 6.3.1, we stated that we want to derive causal dependencies between log events, as well as parallel relations from a process log. To this end, we introduced several relations that exploit the order of events in the log. However, a typical process log contains more information than just the ordering on events and therefore we use this information to further refine the relations on events.

In Section 2.2, we defined a process log, such that each audit trail entry refers to one activity (called “workflow model element” in the MXML format) and one event type, such as “schedule”, “start” or “complete”. Recall Figure 2.3 shown in Section 2.2.2, where we presented these event types as well as some causal dependencies between them. These causal dependencies are exactly what we exploit in this section. We do so, by introducing a relation on activities, not on log events, which tells if these activities are in parallel or not.

**Definition 6.3.4. (Parallel activities)**

Let $A$ be a set of activities and $E$ be a set of event types (i.e. such as “schedule”, “start”, “complete”, etc). Let $T \subseteq A \times E$ be a set of log events and let $W$ be a process log over $T$. We say that two activities $a, b \in A$ are in parallel with respect to two event types $e, e' \in E$, i.e. $a \parallel_{W}^{e, e'} b$ if and only if there is a trace $\sigma \in W$ and $i, j \in \{0, \ldots, |\sigma| - 1\}$, such that $(a, e) = \sigma_i$ and $(a, e') = \sigma_j$ with $i < j$ and for all $k : i < k < j$ holds that $\sigma_k \neq (a, e)$ and $\sigma_k \neq (a, e')$ and

- There is an $l : i < l < j$, such that $\sigma_l = (b, e)$ and there is an $m > l$, such that $\sigma_m = (b, e')$, or,
- There is an $l : i < l < j$, such that $\sigma_l = (b, e')$ and there is an $m < l$, such that $\sigma_m = (b, e)$.

The idea of Definition 6.3.4 is shown in Figure 6.7, where two possible situations are shown from which we conclude that activities $A$ and $B$ are in parallel, i.e. if activity $A$ is scheduled at time $t_1$ (i.e. the event log would show an audit trail entry referring to activity $A$ and event type “schedule”) and completed at time $t_3$ and activity $B$ is scheduled at time $t_2$, (i.e. after $t_1$, but before $t_3$) and completed some time later at time $t_4$, we can conclude that activities $A$ and $B$ are in parallel, and hence that all their events are in parallel. Note that it is easy to see that the relation described in Definition 6.3.4 is symmetric, i.e. if activity $A$ is in parallel with activity $B$, then activity $B$ is also in parallel with activity $A$.

**Figure 6.7:** Activities that overlap in time.
However, just assuming that all events of two activities are in parallel introduces a problem. Consider for example the case such as shown in Figure 6.7 and let us assume that these are the only two possibilities in which the activities $A$ and $B$ appear in the log, i.e. the log will always show that $A$ is scheduled before $B$, but the completion of these two activities can be in both orders. It is clear that the “schedule” event of activity $B$ is always directly preceded by the “schedule” event of activity $A$ and therefore has no other causal predecessors according to our ordering relations.

Therefore, when we derive that the log-events $A_{\text{schedule}}$ and $B_{\text{schedule}}$ are in parallel because activities $A$ and $B$ are in parallel, then all causal predecessors of $A_{\text{schedule}}$ should become causal predecessors of $B_{\text{schedule}}$ as well. By symmetry, the same holds to the causal successors of the final events of each activity.

**Definition 6.3.5. (Transactional parallel extension)**

Let $A$ be a set of activities and $E$ be a set of event types (i.e. such as “schedule”, “start”, “complete”, etc). Let $T \subseteq A \times E$ be a set of log events and let $W$ be a process log over $T$. Furthermore, let $\rightarrow_W, \#_W$ and $\parallel_W$ be ordering relations (i.e. following Definition 6.3.1) on the elements of $T$ and let $i, f \in E$ represent an initial and final event type. We define the transactional parallel relations for $(a, e_1), (b, e_2) \in T$ as:

- $(a, e_1)\parallel_W (b, e_2)$ if and only if $(a, e_1)\parallel_W (b, e_2)$ or $a \stackrel{i, f}{\parallel_W} b$,
- $(a, e_1)\rightarrow_W (b, e_2)$ if and only if not $(a, e_1)\parallel_W (b, e_2)$ and
  
  - $(a, e_1)\rightarrow_W (b, e_2)$ if and only if not $(a, e_1)\parallel_W (b, e_2)$ and not $(b, e_2)\parallel_W (a, e_1)$ and not $(a, e_1)\parallel_W (b, e_2)$.

<table>
<thead>
<tr>
<th></th>
<th>$A_s$</th>
<th>$A_c$</th>
<th>$B_s$</th>
<th>$B_c$</th>
<th>$C_s$</th>
<th>$C_c$</th>
<th>$D_s$</th>
<th>$D_c$</th>
<th>$E_s$</th>
<th>$E_c$</th>
<th>$F_s$</th>
<th>$F_c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_s$</td>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
</tr>
<tr>
<td>$A_c$</td>
<td>$\leftarrow$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
</tr>
<tr>
<td>$B_s$</td>
<td>$#$</td>
<td>$\leftarrow$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
</tr>
<tr>
<td>$B_c$</td>
<td>$#$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
</tr>
<tr>
<td>$C_s$</td>
<td>$#$</td>
<td>$\leftarrow$</td>
<td>$#$</td>
<td>$#$</td>
<td>$\leftarrow$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
</tr>
<tr>
<td>$C_c$</td>
<td>$#$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
</tr>
<tr>
<td>$D_s$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
</tr>
<tr>
<td>$D_c$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
</tr>
<tr>
<td>$E_s$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
</tr>
<tr>
<td>$E_c$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
</tr>
<tr>
<td>$F_s$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
</tr>
<tr>
<td>$F_c$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
<td>$\rightarrow$</td>
<td>$#$</td>
</tr>
</tbody>
</table>
Table 6.5: Causal extension to ordering relations of Table 6.4.

<table>
<thead>
<tr>
<th></th>
<th>A_s</th>
<th>A_c</th>
<th>B_s</th>
<th>B_c</th>
<th>C_s</th>
<th>C_c</th>
<th>D_s</th>
<th>D_c</th>
<th>E_s</th>
<th>E_c</th>
<th>F_s</th>
<th>F_c</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_s</td>
<td>→</td>
<td>→</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>A_c</td>
<td>←</td>
<td>←</td>
<td>→</td>
<td>→</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>B_s</td>
<td>#</td>
<td>←</td>
<td>→</td>
<td>→</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>B_c</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>C_s</td>
<td>#</td>
<td>←</td>
<td>→</td>
<td>→</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>C_c</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>→</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>D_s</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>D_c</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>→</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>E_s</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>E_c</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>F_s</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>F_c</td>
<td>#</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>←</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
</tbody>
</table>

Definition 6.3.5 defines a method where a set of ordering relations is changed into another set of ordering relations, taking time into account for one specific initial and final event, such as “start” and “complete”. The result of this procedure, when applied to our relations of Table 6.3 is shown in Table 6.4. Note that in this table indeed activities B and C are completely in parallel, i.e. B_{start}, B_{complete}, C_{start} and C_{complete} are all in parallel with each other. The resulting relations can again serve as input for the same change with different event types, which allows to get as much information regarding parallelism as possible, thus defining the relations between events as precise as possible.

Using Causal Dependencies from the Transactional Model

In the previous subsection, we exploited the transactional model of Figure 2.3 to obtain as much information as possible about parallelism from an event log. The final way in which we exploit the transactional model of Figure 2.3 is by assuming explicit causal dependencies between events that refer to the same activity. In the example of Figure 6.6, it is clear that the log event B_{start} is always followed by C_{start} and never directly by B_{complete}. Hence in the ordering relations we defined so far, the obvious causal dependency between B_{start} and B_{complete} is not detected. Therefore, the causal dependencies of the transactional model are incorporated in the ordering relations.

To formalize how these dependencies should be incorporated, we need to assume that we do not only have a set of possible event types, but also all causal dependencies between them. In Definition 6.3.6, we therefore assume that we have a relation C that represents all causal dependencies between event types.
Definition 6.3.6. (Transactional causal extension)
Let $A$ be a set of activities and $E$ be a set of event types (i.e. such as “schedule”, “start”, “complete”, etc). Let $T \subseteq A \times E$ be a set of log events and let $W$ be a process log over $T$. Furthermore, let $\rightarrow_W, \#_W$ and $\parallel_W$ be ordering relations (i.e. following Definition 6.3.1) and assume that $C \subseteq E \times E$ represents the causal dependencies between the events in $E$ as shown in Figure 2.3, and $C^\circ$ is the transitive closure of $C$. We define the transactional causal relations for $(a, e_1), (b, e_2) \in T$ as:

- $(a, e_1) \rightarrow_{tc}^W (b, e_2)$ if and only if $(a, e_1) \rightarrow_W (b, e_2)$ or $[a = b$ and $(e_1, e_2) \in C^\circ$ and $\not\exists (a, e_3) \in T$ with $\{(e_1, e_3), (e_3, e_2)\} \subseteq C^\circ$],
- $(a, e_1) \parallel_{tc}^W (b, e_2)$ if and only if $(a, e_1) \parallel_W (b, e_2)$ and not $[(a, e_1) \rightarrow_{tc}^W (b, e_2) \text{ or } (b, e_2) \rightarrow_{tc}^W (a, e_1)]$,
- $(a, e_1) \#_{tc}^W (b, e_2)$ if not $(a, e_1) \rightarrow_{tc}^W (b, e_2)$ and not $(b, e_2) \rightarrow_{tc}^W (a, e_1)$ and not $(a, e_1) \parallel_{tc}^W (b, e_2)$.

Applying the procedure described in Definition 6.3.6 to the ordering relations shown in Table 6.4 yields the relations shown in Table 6.5. Note that now the causal dependency between $B_{\text{start}}$ and $B_{\text{complete}}$ is found. Furthermore, this table shows all causal and parallel dependencies that we would expect to find based on the example log of Figure 6.6, i.e. it shows that:

- Activities $A$ and $D$ causally precede both activities $B$ and $C$, or more precisely, the events $A_{\text{complete}}$ and $D_{\text{complete}}$ causally precede the events $B_{\text{start}}$ and $C_{\text{start}},$
- All events of activities $B$ and $C$ are in parallel (i.e. all the events relating to $B$ and $C$ are in parallel),
- Activities $D$, $E$ and $F$ causally succeed both activities $B$ and $C$, or more precisely, the events $D_{\text{start}}, E_{\text{start}}$ and $F_{\text{start}}$ causally succeed the events $B_{\text{complete}}$ and $C_{\text{complete}},$ and
- The start event of each activity causally relates to the complete event of the same activity.

6.3.3 Summary
In this section, we presented several ways to abstract from a log file by determining causal and parallel relations on the events in the log. Obviously, this still does not give us the process model, we are interested in. Therefore, in the following three sections, we present algorithms that use these abstract log relations to generate a process model in a common language. These algorithms, such as the one presented in Section 6.4, use ordering relations, which basically can be any of the ordering relations we presented in this section, i.e. any set of ordering relations as defined in Definition 6.3.1 can be used. The only exception is the
algorithm presented in Section 6.5, which requires that for all events \( e \) holds that \( e \rightarrow e \).

## 6.4 The \( \alpha \)-algorithm

A well-known process discovery algorithm that is based on the log relations that we presented in Section 6.3 is called the \( \alpha \)-algorithm. In this section, we introduce this algorithm and show some of its applications and limitations.

The \( \alpha \)-algorithm was introduced in [26], where the following algorithm was presented, which generates a WF-net using the log events of the process log as transitions.

**Definition 6.4.1. (Mining algorithm \( \alpha \))**

Let \( T \) be a set of log events and \( W \) a process log over \( T \). Furthermore, let \( \rightarrow_W, \#_W \) and \( \|_W \) be ordering relations on the elements of \( T \) (i.e.

1. \( T_W = \{ t \in T \mid \exists \sigma \in W \ t \in \sigma \} \),
2. \( T_I = \{ t \in T \mid \exists \sigma \in W \ t = \sigma_0 \} \),
3. \( T_O = \{ t \in T \mid \exists \sigma \in W \ t = \sigma_{|\sigma|-1} \} \),
4. \( X_W = \{ (A, B) \mid A \subseteq T_W \land B \subseteq T_W \land \forall a \in A \forall b \in B a \rightarrow_W b \land \forall a_1, a_2 \in A a_1 \#_W a_2 \land \forall b_1, b_2 \in B b_1 \#_W b_2 \} \),
5. \( Y_W = \{ (A, B) \in X_W \mid \forall (A', B') \in X_W A \subseteq A' \land B \subseteq B' \implies (A, B) = (A', B') \} \),
6. \( P_W = \{ p_{(A, B)} \mid (A, B) \in Y_W \} \cup \{ i_W, o_W \} \),
7. \( F_W = \{ (a, p_{(A, B)}) \mid (A, B) \in Y_W \land a \in A \} \cup \{ (p_{(A, B)}, b) \mid (A, B) \in Y_W \land b \in B \} \cup \{ (i_W, t) \mid t \in T_I \} \cup \{ (t, o_W) \mid t \in T_O \} \),
8. \( \alpha(W) = (P_W, T_W, F_W) \).

The \( \alpha \)-algorithm is remarkably simple, in the sense that it uses only the information in the ordering relations to generate a WF-net. In the first three steps of the algorithm, the set of transitions is determined and those transitions are identified that were an initial activity or a final activity in a process instance. Then, in steps 4, 5 and 6, the places of the Petri net are determined by defining their input and output sets. Similar to the Theory of Regions, each place refers to a set of incoming transitions and a set of outgoing transitions. However, how these transitions are divided over the sets is determined by their log-based relations, which eliminates the need for calculating regions. Finally, the transitions and places are linked and the Petri net is completed.

Figure 6.8 shows the result of applying the \( \alpha \)-algorithm to Table 6.5. Recall that this table contains log-based ordering relations derived from only three process instances shown in Figure 6.6. In Figure 6.8, we explicitly labelled the places according to the \( \alpha \)-algorithms logic. However, in the future, we typically do not show these labels, since they are not very informative.
The Petri net shown in Figure 6.8 is a sound workflow net that is indeed capable of reproducing the three process instances of Figure 6.6. Furthermore, it is an abstract representation of these three processes, since it allows that activity $D$ is executed more than once. Typically, as we show in Section 6.4.1, if process logs are generated by certain well-structured processes, the result of the $\alpha$-algorithm is such a sound WF net. However, as we will show next, this is not always the case.

In Section 2.6, we presented an example process of a residence permit application. From this process, we obtained a log file, which is locally complete and we calculated the ordering relations as described in Definition 6.3.2. The result of the $\alpha$-algorithm applied to this log, considering only complete events, is shown in Figure 6.9. Note that the transition “Evaluate objection” is disconnected in this figure. The reason for this is that it has no causal predecessors, since it is in a loop of length two with its predecessor “Send permit rejection letter”.

However, in Section 6.3, we introduced several extensions to the ordering relations, which we can all use to gain better results. If we apply all our extensions to the log-based ordering relations, introduced in Subsection 6.3.2 and we use the process log of our running example to generate a process model again, we get a different result as shown in Figure 6.10, where we condensed the separate transitions for start and complete events into one.

Note that both the Petri nets shown in Figure 6.9 and Figure 6.10 are not sound. It is a well-known fact [26,33] that the $\alpha$-algorithm is only guaranteed to produce a sound WF-net under certain conditions, such as “no transitions can be skipped”, “there are no short loops” and “choices are made locally”.

The process log used to generate the Petri nets of Figure 6.9 and Figure 6.10 does not meet all the conditions for the $\alpha$-algorithm. One of the reasons the algorithm produces an unsound net is that (according to the EPC of Figure 2.9) in the beginning of the process, the “Decide on MVV advice” is not necessarily executed (it can be skipped). Hence in our log, both “Decide on MVV advice” and “Decide on MVV” are initial transitions although they are also causally related, thus leading to a wrong net.
What is important to realize however, is that the Petri net of Figure 6.10, although still not sound, is a better description of the process log than the net of Figure 6.9, since it for example explicitly shows the relations between “Send residence permit rejection letter” and “Evaluate objection”.

Although the α-algorithm sounds promising, it has different drawbacks. First, our experience with real life logs has shown that logs are often not coming from structured processes and since the approach only works if the Petri nets belong to a certain class of nets, this might be a problem. In the next subsection, we present the algorithm in more detail and we show under which conditions the result is indeed a sound WF-net.
6.4.1 Details of the α-algorithm

The ordering relations presented in Definition 6.3.2, can be constructed for any process log. However, one of the conditions for the α-algorithm to succeed is that there is a minimal amount of information in the log. For this purpose, the authors of [26] define the concept of a complete process log, which follows our idea of a locally complete log, i.e. if two transitions can follow each other directly in a trace, this should be observed at least once.

Besides the necessary condition that the process log is a complete one, [26] gives more conditions for the α-algorithm to be correct. First of all the paper demands that the WF-net that generated the process log is of a certain class of nets called structured WF-nets or SWF-nets and that the net does not contain any loops of length one and two. Figure 6.11 graphically shows the forbidden constructs of SWF-nets.

Definition 6.4.2. (SWF-net)

A WF-net $\varphi = (P, T, F)$ is an SWF-net (Structured WF-net) if and only if:

1. For all $p \in P$ and $t \in T$ with $(p, t) \in F$: $|p \bullet| > 1$ implies $|\bullet t| = 1$.
2. For all $p \in P$ and $t \in T$ with $(p, t) \in F$: $|\bullet t| > 1$ implies $|\bullet p| = 1$.
3. For all $p \in P$ with $p \bullet \neq \emptyset$ holds that $p$ is no implicit place.

The problem of dealing with loops of length one and two has been dealt with by extending the log-based ordering relations as described in Section 6.3, and enhancing the α-algorithm slightly, as described in [33,34]. However, the requirement of a SWF net has not been lifted in [33,34], although already in [26], it has been shown that there are Petri nets outside of the class of SWF nets, for which the α-algorithm produces a behavioural equivalent WF-net, such as the WF-net shown in Figure 6.8.

Interestingly, although the class of SWF nets is too restrictive, it seems to be the best we can do while still basing the restrictions on the structure of the net. Since the SWF nets were introduced to enforce that connecting places imply causal relations, we could say that the class of nets the α-algorithm can correctly discover are those for which the state-space reflects this, i.e. if there is a place

![Figure 6.11: Two forbidden constructs in SWF-nets.](image-url)
connecting transition $A$ to transition $B$ then there should be a state in the state-
space with $A$ as an incoming arc and $B$ as an outgoing arc, but there should not
be a state for which this is the other way around.

However, since the goal of process discovery is not to start with a model, but to start with a process log, we think that it is more important to retrieve
as much information as possible from the log and then generate a model that is
as sound, capable of replaying the log and easy to understand. Therefore, if the
$\alpha$-algorithm is executed on a process log and comes up with a model that violates
this requirement, we should use other algorithms to find better results. For that
purpose, we present alternative mining algorithms in sections 6.5 and 6.6, which
are based on the ordering relations defined in Section 6.3 as well, but produces
better results than the $\alpha$-algorithm on less complete information.

6.4.2 Mining Quality

Using the extensions to the ordering relations presented above, the $\alpha$-algorithm
can mine a large class of process logs. However, the $\alpha$-algorithm will not produce a
sensible result for all locally complete logs. Some resulting Petri nets may contain
deadlocks, or disconnected transitions and so on. Moreover, the conditions under
which the $\alpha$-algorithm has been proven to produce good results are such that not
all real-world processes can be modelled under these conditions. Hence, this leads
to a big problem in the context of process mining, where we would like to know
on the basis of a process log, whether the algorithm produces a decent result.
Therefore, in the next sections, we introduce another way of process discovery,
that always produces a result which is sound and can replay the log file, while
still explicitly expressing the causal dependencies between activities that were
captured by the log-based ordering relations.

6.5 Partial Ordered Process Instances

Many process mining algorithms in practice, take a whole world approach to
process discovery, i.e. they go from a process log to a model of the whole process.
For example in case of the $\alpha$-algorithm, the process log is taken as input and
after applying some abstraction algorithm and the actual discovery algorithm, a
process model is generated as output. In this section however, we present the first
step of a different discovery algorithm. The input is similar the the $\alpha$-algorithm,
i.e. a set of abstractions of the log. However, the result is not a single process
model, but a process model for each individual process instance in the log.

Using the log-based ordering relations discussed in Section 6.3, we translate
each process instance in the log, into a partial order on its audit trail entries
called a causal run, which is the first step of our mining approach. Then, in
Section 6.6 we present algorithms to aggregate these partial orders (the second
step) and translations into process models (the third step).

6.5.1 Causal Runs

The process instances we defined in Section 2.2 are basically linearly ordered lists of events. However, by adding explicit knowledge about causal dependencies such process instances could be represented by partial orders. For example, if we know that the completion of activity $A$ always leads to the scheduling of activity $B$ and the cancellation of activity $C$, that would not be expressible as a linear order.

A partial order of events that describes exactly one execution of a process is called a causal run. A causal run can always be represented as an acyclic process model without choices. The reason that there are no choices in a causal run is that the run represents a process instance that was executed in the past, i.e. all choices that had to be made have already been made and therefore no choices remain. The fact that there are no choices implies that we can interpret any two activities that have no causal dependency in the transitive closure of the partial order to be in parallel, but we will come back to that later.

One well-known application of causal runs is the use of Message Sequence Charts (MSCs) to model communication between processes in terms of example scenarios as discussed in Section 3.5. However, in this thesis, we focus on EPCs and Petri nets, as they are more commonly used to model business processes as a whole. Therefore, in the remainder of this section, we introduce a method to translate each process instance of a process log into a causal run, i.e. a partial order on events, and we show how these partial orders can be represented as Petri nets or as EPCs.

6.5.2 Extracting Causal Runs of Sequential Process logs

In Section 6.3, we introduced a method to calculate ordering relations between log events in a process log. In this section, we take some ordering relations, i.e. the causal relations, denoted by $\rightarrow_W$, such that for all log events $e$ holds that $e \rightarrow_W e$, and we use this relation to translate process instances into causal runs\textsuperscript{1}. Then, we show how to represent these causal runs as instance EPCs or Petri nets.

Consider a process log with a linear ordering on events and a set of causal dependencies on these events. There are two things we know about each of the process instances in the log, which are that the initial state is always the same and that the final state is always the same. Note that this assumption is different from the Theory of Regions, where we only assumed the initial state to be unique. However, in the context of business processes, we feel that it is valid to assume that each case ends in a given state, i.e. for example the state “case closed”. To translate a linear order of events into a causal runs, we again

\textsuperscript{1}Note that it is trivial to change any ordering relation following Definition 6.3.1, such that this condition is fulfilled.
take a naive approach. For each event in the process instance, we find the closest causal predecessor and closest causal successor, where we introduce special nodes to identify the start and the end of the case. This idea was first presented in [71].

To be able to introduce an initial and a final node, we need to know the maximum number of events that occurs in one process instance of a log.

**Definition 6.5.1. (Maximum instance size)**

Let $T$ be a set of log events and let $W$ be a process log over $T$. We say that $l_w = \max_{\sigma \in W}(|\sigma|)$ is the maximum instance size.

For each process instance, we define the domain of that instance, so we can make a distinction between the audit trail entries on the one hand (i.e. the executions of a log event) and log events on the other hand.

**Definition 6.5.2. (Process instance domain)**

Let $T$ be a set of log events and let $W$ be a process log over $T$. Let $l_W$ be the maximum instance size of $W$, then we say that $D^+_\sigma = D_\sigma \cup \{-1, l_W\}$ is the extended instance domain of $\sigma$.

Using the domain of an instance, we can link each audit trail entry in the process instance to a specific log event, i.e., $i \in D_\sigma$ can be used to represent the $i$-th element in $\sigma$. In an instance graph, the instance $\sigma$ is extended with an ordering relation $\prec_\sigma$ to reflect some causal relation.

**Definition 6.5.3. (Instance graph)**

Let $T$ be a set of log events and let $W$ be a process log over $T$. Furthermore, let $\rightarrow_W$ be a causal ordering relation following Definition 6.3.1 representing the causal dependencies between elements of $T$ and let $l_W$ be the maximum instance size of $W$. Let $\sigma \in W$ be a process instance. We define $\prec_\sigma \subseteq D^+_\sigma \times D^+_\sigma$ as a relation, such that for $i, j \in D_\sigma$ with $i < j$:

- $-1 \prec_\sigma i$ if and only if for all $0 \leq k < i$ holds that $\sigma_k \not\rightarrow_W \sigma_i$,
- $i \prec_\sigma l_W$ if and only if for all $i < k < |\sigma|$ holds that $\sigma_i \not\rightarrow_W \sigma_k$,
- $i \prec_\sigma j$ if and only if $\sigma_i \rightarrow_W \sigma_j$ and there does not exist a $k : i < k < j$ such that $\sigma_i \rightarrow_W \sigma_k$ or there does not exist a $k : i < k < j$ such that $\sigma_k \rightarrow_W \sigma_j$.

We say that $IG_\sigma = (D^+_\sigma, \{(i, j) \in D^+_\sigma \times D^+_\sigma : i \prec_\sigma j\})$ is an instance graph of $\sigma$.

Figure 6.12 shows four examples of instance graphs. The initial node is called “-1” and the final node is called $l$, which represents a global maximum number, i.e. the maximum instance size. All other nodes refer to a number in the domain of the process instance and for clarity, we put the labels of the log event to which they refer in the figure. Note that instances (b) and (d) are not finished yet, i.e. the process is not finished when the documents are sent to the Town Hall.

The construction of the instance graph using the procedure above is fairly straightforward. Basically, each element of a sequence of log events is connected
to its closest causal predecessor and closest causal successor, i.e. in Figure 6.12(b),
the event “Send documents to Town Hall” occurs later than “Send MVV acceptance letter”,
but they are in parallel and therefore, the closest causal predecessor of “Send documents to Town Hall”
is the activity “Decide on MVV”.

As a basis for the construction of the instance graph, we use the ordering relations presented in Section 6.3.
This yields some interesting properties, that we use later in the aggregation process, the first of which is that the generated
instance graphs are acyclic, i.e. they are partial orders.

**Property 6.5.4. (Instance graphs are acyclic)**

Let $T$ be a set of log events and let $W$ be a process log over $T$. Furthermore, let $\sigma \in W$ be a process instance and let $IG_\sigma = (N, E)$ be the instance graph of $\sigma$.
The graph $(N, E)$ is acyclic, i.e. for all non-empty paths $(n_1, \ldots, n_k) : n_1 \neq n_k$

**Proof.** Assume that there is a cycle in the graph $(N, E)$, this implies that there is
at least one edge $e \in E$, such that $e = (i, j)$ and $j < i$. However, since $i \prec_\sigma j$
implies $i < j$ this is impossible according to Definition 6.5.3.

Besides being acyclic, each instance graph is minimal. This implies that if, in an
instance graph, there is a path of length 3 or more between two nodes, then there is
no edge between these nodes. This property will be required by the aggregation
process later.

**Property 6.5.5. (Instance graphs are minimal)**

Let $T$ be a set of log events and let $W$ be a process log over $T$ with $l_W$ the
maximum instance size of $W$. Furthermore, let $\sigma \in W$ be a process instance and let $IG_\sigma = (N, E)$ be the instance graph of $\sigma$. Then, for all $(a, b) \in E$ holds that $(a, b)$ is the only path from $a$ to $b$.
Proof. Let \( e \in E \) be an edge, such that \( e = (a, b) \) and assume that there is a path \( (a, i_1, \ldots, i_n, b) \) in \( IG_\sigma \). We know that \( a < i_1 < \ldots < i_n < b \) from Definition 6.5.3. We need to consider three cases:

1. If \( a = -1 \) then \( (a, b) \in E \) implies that for all \( 0 \leq k < b \) holds that \( \sigma_k \not\rightarrow_W \sigma_b \), hence \( (a, i_1) \notin E \),

2. If \( b = l_W \) then \( (a, b) \in E \) implies that for all \( a < k < |\sigma| \) holds that \( \sigma_a \not\rightarrow_W \sigma_k \), hence \( (i_n, b) \notin E \),

3. Otherwise, \( (a, b) \in E \) implies that for all \( a < k < b \) holds that \( \sigma_a \not\rightarrow_W \sigma_k \) or for all \( a < k < b \) holds that \( \sigma_k \not\rightarrow_W \sigma_b \), hence \( (a, i_1) \in E \Rightarrow (i_n, b) \notin E \) and \( (i_n, b) \in E \Rightarrow (a, i_1) \notin E \).

Finally, it is of interest to realize that one event can never cause to two instantiations of another event, or be caused by two instantiations of another event. Recall that the nodes of an instance graph are the indices of events in a process instance and therefore, we show that in instance graph, input and output nodes refer to unique log events, at least if we assume that log events are always causally related to themselves i.e. for all events \( e \) holds that \( e \rightarrow e \). In the aggregation process that we present in Section 6.6, we rely on this assumption to be able to determine the types of splits and joins. Violating this assumption would lead to a situation where too many splits and joins would be determined to be of type OR.

Property 6.5.6. (Input and output nodes have unique labels)

Let \( T \) be a set of log events and let \( W \) be a process log over \( T \). Furthermore, let \( \sigma \in W \) be a process instance, let \( IG_\sigma = (N, E) \) be the instance graph of \( \sigma \). Then, for all \( a \in D_\sigma \) holds that \( |a^{IG_\sigma}| = |\{\sigma_b \mid b \in a^{IG_\sigma}\}| \) and symmetrically \( |^{IG_\sigma}a| = |\{\sigma_b \mid b \in IG_\sigma a\}| \).

Proof. Assume \( a \in D_\sigma \) and let \( B = a^{IG_\sigma} \). Furthermore, let \( b_1, b_2 \in B \) with \( b_1 < b_2 \), such that \( \sigma_{b_1} = \sigma_{b_2} \). However, in this case, the edge \( (a, b_2) \in E \) cannot exist. The closest causal successor of \( a \) is not \( b_2 \), since \( b_1 \) is a closer causal successor. Furthermore, \( a \) is not the closes causal predecessor of \( b_2 \), since by definition (Definition 6.3.3), we know that the \( b_1 \) is a closer causal predecessor of \( b_2 \). Therefore, the edge \( (a, b_2) \) cannot exist. The proof for the pre-set is symmetrical.

To show that instance graphs are not merely a theoretical entity, we show that it is rather trivial to translate these partial orders into more readable models, such as Petri nets and EPCs.
6.5.3 Representing Instance Graphs as Petri nets

The instance graph we have just introduced can easily be translated to a Petri net. However, we need a special class of Petri nets, named labeled P/T nets, since each transition is uniquely identifiable by the sequence number in the process instance, but each transition also refers to a log event, which is not necessarily unique in the process instance.

**Definition 6.5.7. (Labelled Place/Transition net)**

$\ell_\varphi = (P, T, F, \Lambda, \gamma)$ is a labelled place/transition net (or labelled P/T-net) if and only if:

- $(P, T, F)$ is a P/T net,
- $\Lambda$ is a finite, non-empty set of labels,
- $\gamma : T \rightarrow \Lambda$ is a labelling function, labelling elements of $T$ with elements of $\Lambda$.

Definition 6.5.7 states that a labelled P/T net is a P/T net, where each transition has a label. Using this concept, we can convert an instance graph in a labelled P/T net.

**Definition 6.5.8. (Instance graph as labelled P/T net)**

Let $T$ be a set of log events and let $W$ be a process log over $T$. Furthermore, let $\sigma \in W$ be a process instance and let $IG_\sigma = (N, E)$ be the instance graph of $\sigma$ and let $l_W$ be the maximum instance size of $W$. We say that $\ell_\varphi IG_\sigma = (P, T', F, \Lambda, \gamma)$ is the labelled P/T net representation of $IG_\sigma$ if and only if:

- $P = \{p_e | e \in E\} \cup \{p_i, p_f\}$ is the set of places, where $p_i \neq p_f$ and $\{p_e | e \in E\} \cap \{p_i, p_f\} = \emptyset$, i.e. each edge in the instance graph is represented as a place and we have a unique initial and final place,
- $T' = N$ is the set of transitions, which will be labelled with elements of $\Lambda$, i.e. the observed log events,
- $F = \bigcup_{(n,m) \in E} \{(m, p_{(m,n)}), (p_{(m,n)}, n)\} \cup \{(p_i, n) | n^{IG_\sigma} = \emptyset\} \cup \{(n, p_f) | n^{IG_\sigma} = \emptyset\}$, i.e. each edge in the instance graph is translated into one place with two edges,
- $\Lambda = T \cup \{\tau\}$ is the set of labels, where $\tau \notin T$, i.e. $\tau$ is a label that does not appear in the log,
- $\gamma : N \rightarrow \Lambda$ is the labelling function, such that $\gamma(-1) = \tau$ and $\gamma(l) = \tau$ and for all $n \in D_\sigma$ holds that $\gamma(n) = \sigma_n$.

Figure 6.13 shows the labelled P/T net representation of the four instance graphs shown in Figure 6.12. Note that we use black transitions to depict the transitions labelled with $\tau$.

The representation of instance graphs in terms of labelled P/T nets is rather straightforward. The nodes in the instance graph are represented as transitions,
such that the label of each of them is $\tau$ for the initial and final node and the log event from the log for all the others. The places are created by translating each edge into a place with one incoming and one outgoing arc. Finally, one initial and one final place is created and connected to the initial and final node. The reason that this translation is straightforward is that there are no choices, i.e. we do not have to worry about how places should be fit between transitions. This observation also helps us in the translation to EPCs, where the only connectors of interest are the AND connectors.

### 6.5.4 Representing Instance Graphs as EPCs

Translating instance graphs to instance EPCs is less straightforward. Since the instance graph only shows which log events are appearing in the log, and these log events relate to functions in an EPC. Furthermore, the splits and joins in the instance graph correspond to connectors in an EPC. However, we have no information about the events we should use in the EPC. Therefore, we introduce an event before each function that serves as the precondition of that function. Again, we first introduce a labelled EPC to accommodate the fact that a function
Definition 6.5.9. (Labelled Event-driven Process Chain)

$\ell \in (F, E, C_{\text{and}}, C_{\text{xor}}, C_{\text{or}}, \Lambda, \gamma)$ is a labelled EPC if and only if:

- $(F, E, C_{\text{and}}, C_{\text{xor}}, C_{\text{or}}, A)$ is an EPC, and
- $\Lambda$ is a finite, non-empty set of labels,
- $\gamma : F \cup E \rightarrow \Lambda$ is a labelling function, labelling elements of $F$ and $E$ with elements of $\Lambda$.

In Figure 6.14, we show the EPC representation of the instance graphs of Figure 6.12.

The definition of a labelled EPC allows us to label events and functions with the same label and since we have to introduce events ourselves. We do so by introducing an event before each function and we label the events we introduce with the label of the succeeding function. This makes the translation rather simple. First, all nodes of the instance graph are translated to connected pairs of an event followed by a function with the same label. These labels are the log events from the process instance, except for the initial and final node, i.e. the nodes with number $-1$ and $l$, where we use the labels $\tau_{\text{initial}}$ and $\tau_{\text{final}}$ respectively. In the second step, for all events where more than one edge comes in and for all functions where more than one edge goes out, an AND connector is introduced and the edges are connected to these connectors. Finally, a new event

![Diagram](image-url)  
Figure 6.14: Instance graphs of Figure 6.12 as EPCs.
labelled with $\tau$ is introduced that serves as the result of the function labelled $\tau_{\text{final}}$.

**Definition 6.5.10. (Instance graph as EPC)**

Let $T$ be a set of log events and let $W$ be a process log over $T$. Furthermore, let $\sigma \in W$ be a process instance and let $IG_\sigma = (N, E)$ be the instance graph of $\sigma$ and let $l_W$ be the maximum instance size of $W$. We say that $l_{IG_\sigma} = (F, E, C_{\text{and}}, C_{\text{xor}}, C_{\text{or}}, A, \Lambda, \gamma)$ is the labelled EPC representation of $IG_\sigma$ if and only if:

- $F = N \setminus \{l_W\}$ is the set of functions (i.e. $F = D_\sigma \cup \{-1\}$),
- $E = \{e_n \mid n \in N\}$, i.e. each function is preceded by one event,
- $C_{\text{xor}} = C_{\text{or}} = \emptyset$, i.e. there are no choice connectors,
- $C_{\text{and}} = C_j \cup C_s$, where $C_j = \{c_n^{\text{join}} \mid n \in N \land |^{1G_\sigma} \cdot n| > 1\}$ and $C_s = \{c_n^{\text{split}} \mid n \in N \land |n^{1G_\sigma} \cdot | > 1\}$, i.e. the AND-connectors are split up into join and split connectors,
- $A = \bigcup_{n \in F} \{(e_n, n)\} \cup \bigcup_{(n, m) \in E} \{(n, e_m)\} \cup \bigcup_{(n, m) \in E} \{(e_n^{\text{split}}, c_m^{\text{join}}), (n, c_n^{\text{split}}), (c_m^{\text{join}}, e_m)\} \cup \bigcup_{(n, m) \in E} \{(c_n^{\text{split}}, e_m), (n, c_n^{\text{split}})\}$
- $\Lambda = T \cup \{\tau_{\text{initial}}, \tau_{\text{final}}\}$ is the set of labels, where $\tau_{\text{initial}}, \tau_{\text{final}} \notin T$, i.e. the new labels are added for the new events and functions,
- $\gamma : F \cup E \rightarrow \Lambda$ is the labelling function, such that $\gamma(-1) = \gamma(e_{-1}) = \tau_{\text{initial}}$ and $\gamma(e_{l_W}) = \tau_{\text{final}}$ and for all $n \in D_\sigma$ holds that $\gamma(n) = \gamma(e_n) = \sigma_n$.

### 6.5.5 Summary

In this section, we presented causal runs. These causal runs are constructed from a process log by using the ordering relations of Section 6.3. More specifically, the causal relations are used for each log event in a process instance to find its closest causal predecessor and causal successor. The result is a partial order and in Property 6.5.5, we have shown that this partial order is minimal.

Although we have shown that we can now represent each individual process instance as a labelled Petri net or EPC, the goal of process discovery is to find one EPC or Petri net for the whole log. Therefore, in Section 6.6, we show how we can use the instance graph defined in this section to construct a process model for the whole log.
6.6 Aggregation of Partial Orders

The goal of process discovery in the context of process mining is to generate a process model that represents the log file as good as possible. A desirable property is that it should at least be able to reproduce the log. In Section 6.4, the $\alpha$-algorithm was presented, which has proven to be a very powerful algorithm, but for a limited set of processes. Furthermore, in Section 6.5, we have introduced the concept of causal runs and how these can be constructed from a process log. In this section, we introduce an aggregation algorithm that uses the instance graph to construct a model, i.e. where, in the first step, we generated instance graphs, we now present the second step, where we aggregate these instance graphs into a process model.

6.6.1 Aggregating instance graphs

Let us assume that the only information we have about a process is the log that we obtained. In that case, we can use the log-based ordering relations that we have presented in Section 6.3 to generate partial orders for each process instance, called instance graphs, which we introduced in Subsection 6.5.2. In this section, we present an algorithm to aggregate these instance graphs into an aggregated process graph and we show how this graph can be translated into an EPC and how, using the techniques from Chapter 5, this EPC can be translated into a Petri net as well.

The process is as follows [73]. First, we project each instance graph on its set of labels, i.e. we introduce loops in the instance. Then, we use these projections and aggregate them into an aggregation graph.

**Definition 6.6.1. (Instance graph projection)**

Let $T$ be a set of log events, let $W$ be a process log over $T$ and let $l_W$ be the maximum instance size of $W$. Furthermore, let $\sigma \in W$ be a process instance and let $IG_\sigma = (N, E)$ be the instance graph of $\sigma$. We define $\Pi(IG_\sigma) = (N', E', l')$ as the projection of this instance graph onto $T$ such that:

- $N' = T \cup \{t_s, t_f\}$ is the set of nodes, where $\{t_s, t_f\} \cap T = \emptyset$ and $t_s \neq t_f$,
- $E' = \{(\sigma_a, \sigma_b) | (a, b) \in E\}$, where we assume that $\sigma_{-1} = t_s$ and $\sigma_{l_W} = t_f$,
- $l' : N' \cup E' \rightarrow \mathbb{N}$ where
  - for all $t \in N'$ we define $l'(t) = |\{a \in N | \sigma_a = t\}|$
  - for all $(a, b) \in E'$, we define $l'((a, b)) = |\{(a', b') \in E | \sigma_{a'} = a \land \sigma_{b'} = b\}|$, where again $\sigma_{-1} = t_s$ and $\sigma_{l_W} = t_f$.

In Definition 6.6.1 we build an instance graph in such a way that all nodes are now elements of the set of log events $T$, or one of the fictive nodes $t_s$ and $t_f$. Furthermore, edges are constructed in such a way that if two task executions were connected in the instance graph then the tasks are connected in the projection.
Finally, for each task execution that was not preceded or succeeded by any other execution there now is an edge connecting the task to the start or final node respectively. The result is a graph where each node lies on a path from the start node to the final node. Furthermore, we introduced a labeling function $l'$ that gives the number of times a task was executed (or an edge was taken) in an instance.

In Figure 6.15, we show an example of an instance graph, concerning four log events, i.e. $A$, $B$, $C$ and $D$. Note that both $B$ and $D$ occur two times and $C$ occurs three times. Figure 6.16 shows the instance graph projection of this instance graph, where indeed the labels of $B$ and $D$ are 2 an the label of $C$ equals 3. These instance graph projections have some interesting properties, which are presented in detail in [73]. We briefly repeat these properties here in an informal way.

**Projection preserves paths**

Making a projection of an instance graph onto the set of log events (or tasks in [73]), preserves paths in the graph. For an instance graph, we know that every time a node has two outgoing or incoming edges, this represents a parallel split or join respectively. There are no choices, since each node represents an audit trail entry and not a log event. Clearly this no longer holds for projection graphs where nodes represent log events and not individual executions thereof. Therefore, when making an instance graph projection, we loose explicit information about parallelism. However, important information about the causal structure of the process is preserved, i.e. we are still able to see whether splits and joins in the projection are XOR, OR or AND splits and joins.
Exclusive choices can be found in projections

In an instance graph, the labels of nodes represent how often a log event appeared in a process instance and the labels of edges show how often that edge was taken in the execution. If a node has label $x$ and the sum of the labels of the outgoing edges equals $x$, then this is an exclusive or split, i.e. never were two edges taken at once. Note that this holds on the input side as well. In Figure 6.16 for example, $C$ is an XOR join on $A$ and $D$, since the label of $C$ equals 3 and the sum of the edges coming from $A$ and $D$ also equals 3. In Figure 6.17, we show the types of all splits and joins.

True parallelism can be found in projections

If the label of a node is $x$ and each outgoing edge also has label $x$, then we know that this refers to a parallel split, i.e. always all edges were taken in execution. Note that this holds on the input side as well. In Figure 6.16 for example, $A$ is an AND split on $B$ and $C$, since the label of $A$ equals 1 and both the labels of the edges going to $B$ and $C$ also equal 1.

OR splits and joins can be found in projections

If a split or join is not an exclusive choice or a parallel split/join then it is a multi choice, or OR, i.e. every execution of a task a selection of the output edges is taken. In Figure 6.16 for example, $D$ is an OR split on $B$ and $C$, since the label of $D$ equals 2 and the sum of the labels of the edges going to $B$ and $C$ is greater than 2, but they are not both 2.
Figure 6.18 shows the instance projections of the four instance graphs of Figure 6.12. Note that these projections do not contain loops and hence all labels of both nodes and edges are 1. However, the first step in the aggregation process, i.e. the transformation from an instance graph into an instance graph projection, is an important step, since all instance graph projections share their sets of nodes and hence we can aggregate two instance projections, while we cannot aggregate instance graphs.

Furthermore, the properties of instance graph projections with respect to the labels of nodes and edges are essential in the process of translating the aggregation graph into an EPC (or Petri net). However, before presenting these translations, we give an algorithm to generate an aggregation graph. This graph is a straightforward summation over a set of instance graph projections. Therefore, the set of nodes is simply the union set of all instance graph projections.

**Definition 6.6.2. (Aggregation graph)**

Let $T$ be a set of tasks and let $S$ be a set of instance graph projections over $T$. We define $\beta(S) = (N_S, E_S, l_S)$ to be the aggregation graph over $S$ such that:

- $N_S = \bigcup_{(N,E,l) \in S} N$, i.e. the set of nodes.
- $E_S = \bigcup_{(N,E,l) \in S} E$, i.e. the set of edges.
- For all $n \in N_S \cup E_S$ we define $l_S(n) = \sum_{(N,E,l) \in S} l(n)$, where we assume that $l(n) = 0$ if $n \not\in N \cup E$.

In essence, the aggregation graph is the straightforward sum over a set of instance graph projections. Note that each instance projection contains the start node $t_s$ and the final node $t_f$. Therefore, the aggregation graph also contains these unique nodes $t_s$ and $t_f$ in such a way that $t_s$ is the only source node and $t_f$ is the only sink node. The labels represent the number of times a task is executed or a causal relation is present in some instance graph.

Figure 6.19 shows the resulting aggregation graph after aggregating the instance graph projections presented in Figure 6.18, whereas Figure 6.20 shows the entire aggregation graph, after applying the algorithms to a locally complete log of our residence permit application process.

In [73], we have proven that for an aggregation graph, the same properties hold as for an instance graph projection, i.e. of each split and join we can decide whether it is an exclusive choice, a multi-choice or a true parallel split or join based on the labels of nodes and edges. This information is used to translate and aggregation graph to EPCs and Petri nets.

After aggregating a set of instance graphs into an aggregation graph the result has to be visualized. Although the aggregation graph as shown in Figure 6.20 contains all the information needed, it is still hard for a human to understand. Therefore, we translate the graph into a format that is more intuitive, i.e. EPCs. The choice for EPCs is based on the fact that it is a well known concept, and
Figure 6.19: Aggregation graph with all its labels.

Figure 6.20: Complete aggregation graph of our running example.
it allows for all necessary routing constructs. However, in Subsection 6.6.3, we show how this EPC can be used to construct a labelled Petri net.

### 6.6.2 Representing Aggregation Graphs as EPCs

In the labelled aggregation graph, the labels of the nodes represent the number of times a task is executed in the log. The labels of the edges represent the number of times an edge is taken in some instance. Similar to instance graph projections, it is easily seen that there are three possibilities for the incoming as well as for the outgoing edges of a node.

1. The *sum* over the labels of all incoming (outgoing) edges of some node equals the label of the node.
2. *Each of* the labels of *all* incoming (outgoing) edges of some node equal the label of the node.
3. *Not all* the labels of all incoming (outgoing) edges of some node equal the label of the node, and the *sum* over the labels is *greater* than the label of the node.

We use the three possibilities for the labels to determine the type of connectors we use in the transformation process to EPCs. These points are symmetrical for ingoing and outgoing edges at all nodes. Therefore, if we talk about incoming edges, we also implicitly refer to the symmetrical case for outgoing edges. Transforming an aggregation graph into an EPC is straightforward and we only give a sketch of the translation algorithm.

**Definition 6.6.3. (Translation to EPC)**

Let $S$ be a set of instance graphs and let $\beta(S) = (N_S, E_S, l_S)$ be the labelled aggregation graph over $S$. We define the aggregated EPC as:

$$EPC = (F, E, C_{\text{and}}, C_{\text{xor}}, C_{\text{or}}, A),$$

where:

- $F = N_S \setminus \{t_f\}$ is the set of functions,
- $E = \{e_n \mid n \in N_S\}$ is the set of events,
- $C_{\text{and}} \cup C_{\text{xor}} \cup C_{\text{or}}$ is the set of connectors, constructed as shown in Figure 6.21,
- $A$ is the set of edges, constructed as shown in Figure 6.21.

Figure 6.21 defines the transformation of the aggregation graph into an EPC. These rules are rather straightforward. First, each node $n$ from the aggregation graph is translated into a combination of an event $e_n$ and a function $n$ and an edge is added in between. Then, the incoming and outgoing edges at all nodes are transformed into connectors. Note that for connectors that have only one ingoing or outgoing arc, the type is ambiguous, since both *and* and *xor* can be applied. However, since these connectors are actually no real connectors, we remove all of
them that have only one ingoing and one outgoing edge, and replace them with a normal edge. For the initial and final nodes $t_s$ and $t_f$, special rules are made.

Figure 6.22 shows the final node of Figure 6.20 in detail. Clearly, the sum over the labels of the edges going into that node is greater than the label of the node itself and they are not all the same, therefore, in the translation the result will be an OR-join connector as shown in Figure 6.23.

An important property of the resulting EPC is that it is indeed capable of reproducing all original instance graphs. This can easily be seen from the fact that an EPC is merely a syntactically different representation of a labelled aggregation graph. We already mentioned that each path in an instance graph is also a path in an aggregation graph. Therefore, it is also a path in an aggregated EPC. Furthermore, we mentioned that the types of splits and joins can be found in an aggregation graph. Based on these properties, the transformation into an EPC is made. Therefore, it is clear that for the aggregated EPC the same properties hold. All the instance graphs that we started with in the first place are executions of the EPC.

Especially when OR-connectors are used, the EPC can allow for more behaviour than was observed in the log. However, recall Section 5.3, where we stated that an EPC can be correct if the underlying Petri net is relaxed sound, under the condition that special care is taken with respect to invisible transitions that relate to OR-connectors. The EPC that we obtained by translating the aggregation graph will always yield this result, or better. In other words the
underlying Petri net will always be relaxed sound. The reason for that is simple, since each process instance is a relaxed sound execution of the EPC, where for all OR connectors some configuration was selected. Since there are no edges introduced in the aggregation graph that are not observed at least once in a process instance, there are no dead paths and the underlying Petri net is relaxed sound.

6.6.3 Representing Aggregation Graphs as Petri nets

In Section 5.3, we introduced an algorithm to translate EPCs to Petri nets. Obviously, the same approach can be applied to the aggregation EPC we generated from our MXML log. Recall that in this translation, each function is translated into a transition with the same label and that each connector is translated into a set of \( \tau \)-labelled (or invisible) transitions. However, using the original instance graphs, we can go one step further, i.e. for each invisible transition belonging to an OR-connector, we can decide whether or not it belongs in the Petri net, based on its occurrence in an instance graph. Assume there is an invisible transition that after the execution for function \( a \) enables functions \( b \) and \( c \) but there are also invisible transitions that enable only \( b \) and only \( c \) after \( a \). Each of these transitions can be removed from the Petri net if there is no instance graph in which \( a \) was causally followed by \( b \) and \( c \), or only \( b \), only \( c \) respectively. If we assume that after \( a \) we have never seen only \( c \), but always only \( b \) or both \( b \) and \( c \), then the invisible transition linking \( a \) and \( c \) can be removed while the resulting net is still able to reproduce all process instances. This procedure is described in detail in Section 6 of [73], where it is called restriction.

If we consider the final node of Figure 6.20 again, the OR-join connector of Figure 6.23 should be translated into 15 invisible transitions. However, in Figure 6.25, we show that only three are necessary and all others would be removed in the restriction process.

6.6.4 Mining Quality

Although our process mining approach using causal runs is based on the same log-based ordering relations as the \( \alpha \)-algorithm, it always outperforms that algorithm, since the resulting model is guaranteed to be able to reproduce the
log. Furthermore, the $\alpha$-algorithm only works if the log satisfies certain properties, whereas the multi-phase approach, i.e. generating instance nets, aggregating instance nets, converting the aggregation graph to an EPC and to a Petri net consecutively and finally restricting the Petri net to the instances, does not have such requirements.

However, when we consider process logs which were generated using a Petri net fulfilling the requirements of the $\alpha$-algorithm, i.e. a SWF net without short loops the requirements on the completeness of the log are the same, since the log-based ordering relations serve as a basis for both approaches. Furthermore, if the logs are complete, the resulting Petri nets are behaviourally equivalent, in the sense that they can produce exactly the same set of traces in terms of visible transitions.

### 6.7 Conclusion

In this chapter, we presented several approaches to process discovery. These approaches included:

**The Theory of Regions,** where the resulting Petri net can exactly reproduce the given process log and nothing more. Although this sometimes is a desirable feature, the resulting Petri nets are not WF-nets and therefore not very useful in practice. Furthermore, the algorithm works directly on a state space which has to be generated from the log and this makes the algorithm very time consuming.

**The $\alpha$-algorithm,** which uses log-based ordering relations to derive knowledge about causal dependencies and parallelism and then uses these relations to construct a WF-net. The problem with this algorithm is that the result is only sound if the process log satisfies certain properties which cannot be derived from the log directly.
An instance based algorithm, where the same ordering relations are used as for the \( \alpha \)-algorithm, but where there is a separation between determining the structure of each individual process instance and an aggregation step, where these structures are aggregated. However, the resulting model, which can always reproduce the entire process log, regardless of the quality of the information, can be extremely generic if the causal relations cannot be determined accurately enough.

All these algorithms nicely show the diversity of process discovery, or process mining in general. We have clearly shown that the process discovery heavily depends on both the knowledge on the input and the required output format. If you know that a process log shows all the behaviour you will ever see and you are looking for a Petri net that compactly describes that log, then the Theory of Regions is the most logical choice for a process mining algorithm. The result however is a Petri net and not a workflow net, i.e. it is less suitable as input for process design.

If a process log does not show all possible behaviour and you are looking for a workflow net that can reproduce the process log, but might allow for more behaviour, the \( \alpha \)-algorithm presented in Section 6.4 can give good results. However, if the process log does not adhere to certain properties which cannot be checked beforehand, the \( \alpha \)-algorithm might produce unsound WF-nets. Therefore, the two step approach presented in section 6.5 and 6.6 is guaranteed to give a process model that is relaxed sound and can reproduce the process log.

To conclude, it is important to realize that that process discovery is not an automated step, i.e. in the context of the BPM life cycle, process discovery is not meant to automate the process design phase. Instead, it should be used as a tool for supporting that phase, by using different algorithms and comparing the results.
Chapter 7

Process Discovery on Partially Ordered Logs

In Chapter 6, we presented a multi-phase approach to process discovery. In that approach, we have shown how to abstract from a process log by calculating some ordering relations. Then, these ordering relations were used to describe each individual process instance as a partial order on its audit trail entries. By
aggregating these partial orders, we produced a process model in terms of an
EPC that has proven to be correct. Furthermore, we presented an algorithm to
translate that EPC into a relaxed sound workflow net.

In this chapter, as shown in Figure 7.1, we again consider partial orders
on events that represent cases and we use them to generate aggregated process
models. However, in contrast to Chapter 6, we assume that the partial orders
are stored in the log by an information system, i.e. they are presented as input
to our algorithms and not constructed from the log. Since these partial orders
appear in real life in several contexts, we subsequently assume that our input is
a set of so-called MSCs in Section 7.1, instance EPCs in Section 7.2 or Petri net
runs in Section 7.3.

7.1 Aggregating MSCs

Message Sequence Charts (MSCs) [100,156] are a well-known language to specify
communication between agents, and are supported by many tools, standards, and
approaches, e.g., the Object Management Group (OMG) has decided to adapt a
variant of them called Sequence Charts in the UML notational framework [93].
In this section we look at MSCs that are restricted to only using agents and
messages. We do not consider structured language features such as choice and
iteration that the UML 2.0 standard introduces, i.e., we consider basic MSCs
rather than high-level MSCs. The reason for abstracting from these high-level
constructs is that users typically use MSCs only to model example scenarios
and not complete process models. Therefore, most users do not use routing constructs
such as parallel routing, iteration, choice, etc. in MSCs.

When developing a system it is often useful to describe requirements for the
system by MSCs were each MSC depicts a single scenario. The strength of MSCs
is that they depict a single scenario using an intuitive notation. Therefore, they
are easy to understand. However, this strength can at the same time also be
considered a weakness. How does one consolidate several scenarios from the
same system? This is far from trivial. For example, two MSCs may be similar up
to a certain point, after which they diverge. This point corresponds to a choice
if we look at them in combination, but this is not clear just by looking at one
of the MSCs. Synchronization points are not discovered just by looking at one
diagram, but by looking at multiple MSCs in relation to one another.

7.1.1 Message Sequence Charts

Since several variants of MSC exists, we focus on MSCs with only two syntactical
constructs, i.e., agents and messages. Agents can be used to denote a wide
variety of entities ranging from software components and web services to people
and organizations. A message is passed from one agent to the other and therefore
each message has a, (not necessarily different) sender and receiver agent. An agent has a lifeline representing a sequence of messages that the agent is the sender or receiver of.

In Figure 7.2 we show two examples of the MSCs that we consider. Each of them consists of three agents, Agent A, Agent B, and Agent C. These agents are represented by their name and their lifelines. These lifelines are connected to each other by messages, which are represented by labelled arrows. Internal events (i.e., messages where the sender and receiver are the same agent) are represented by a box on the lifeline of the corresponding process, again with a label.

The process shown in both MSCs of Figure 7.2 starts by the sending of message \( a \) by Agent A. Then message \( a \) is received by Agent B. Next Agent A sends a message \( a \) to Agent C. The overall process continues like that until Agent C sends a message \( c \) that is received by Agent A and Agent B sends a message \( b \) that is received by Agent A.

This section presents our approach of generating a process model from a set of MSCs, i.e. how to aggregate a set of MSCs. This approach is based on the aggregation approach presented in Section 6.6, since we simply provide a translation of MSCs to instance graphs, where we relate messages to activities and agents to the people involved in the process, i.e. the originators.

### 7.1.2 MSCs to Instance Graphs

In our translation, each MSC is translated into one process instance. The reason for this is simple. Since each MSC describes one possible execution scenario of the system, it corresponds nicely to the notion of a case also referred to as process instance. Furthermore, the sending of a message is seen as an activity, which is performed by the agent from which the message originates.

In Section 6.5, we have presented a partially ordered representation of a case using the concept of an instance graph. These instance graphs serve as a basis for the aggregation algorithm presented in Section 6.6. Therefore, in this subsection, we present an algorithm to translate MSCs into instance graphs.

In the translation of MSCs to instance graphs, all messages are translated into two audit trail entries, one referring to the sending of the message and one

---

**Figure 7.2:** Two example MSCs of three agents communicating.
referring to the receiving of the message. To accomplish this, we made sure that both audit trail entries refer to the same activity (i.e., the message). The audit trail entry that refers to the sending of the message has event type `start`. Receiving audit trail entries have type `complete`.

Consider for example the MSC of Figure 7.2, where the first three events are: (1) the sending of message `a` by Agent A, (2) the receiving of message `a` by Agent B and (3) the sending of message `a` by Agent A. These events are represented in MXML by the audit trail entries in Table 7.1

In the translation of MSCs to instance graphs, causal relations between audit trail entries, which in MXML are stored in the data part of the audit trail entries,

Table 7.1: Example MXML log with partial order stored in attributes.

```
<?xml version="1.0" encoding="UTF-8"?>
<WorkflowLog xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="WorkflowLog.xsd"
    description="Log of residence permit application model">
    <Source program="MSC modelling tool"></Source>
    <Process id="0" description="Translation of MSCs to instance graphs">
        <ProcessInstance id="id-instance-0">
            <Data>
                <Attribute name="isPartialOrder">true</Attribute>
            </Data>
            <AuditTrailEntry>
                <Data>
                    <Attribute name="ATE_id">id1</Attribute>
                    <Attribute name="ATE_post">id2,id3</Attribute>
                    <Attribute name="ATE_pre"></Attribute>
                </Data>
                <WorkflowModelElement>a</WorkflowModelElement>
                <EventType>start</EventType>
                <Originator>Process A</Originator>
            </AuditTrailEntry>
            <AuditTrailEntry>
                <Data>
                    <Attribute name="ATE_id">id2</Attribute>
                    <Attribute name="ATE_post">id6</Attribute>
                    <Attribute name="ATE_pre">id1</Attribute>
                </Data>
                <WorkflowModelElement>a</WorkflowModelElement>
                <EventType>complete</EventType>
                <Originator>Process B</Originator>
            </AuditTrailEntry>
            <AuditTrailEntry>
                <Data>
                    <Attribute name="ATE_id">id3</Attribute>
                    <Attribute name="ATE_post">id4,id5</Attribute>
                    <Attribute name="ATE_pre">id1</Attribute>
                </Data>
                <WorkflowModelElement>a</WorkflowModelElement>
                <EventType>start</EventType>
                <Originator>Process A</Originator>
            </AuditTrailEntry>
        </ProcessInstance>
    </Process>
</WorkflowLog>
```
are built in a trivial way. If an audit trial entry refers to the sending of a message, the preset of that audit trail entry is the event that happened before it on the lifeline of that process. If the audit trail entry refers to the receiving of a message, the preset also contains the audit trail entry referring to the sending of the same message. The postsets are built in a similar fashion.

Table 7.1 shows that for each audit trail entry, there are data elements `ATE_id`, `ATE_post` and `ATE_pre`. These data elements are understood by ProM (cf. Chapter 8) and ProM understands that each process instance for which the data element `isPartialOrder` is set to true actually represents an instance graph, allowing us to mix partial and linear orders in one log file. The instance graphs resulting from this translation for the two MSCs of Figure 7.2 are shown in Figure 7.3, where we used the same notation as Figure 6.12.

![Figure 7.3: MSCs of Figure 7.2 as instance graphs.](image)

### 7.1.3 Aggregation of MSCs

The final phase of the MSC aggregation approach is simply to apply the aggregation algorithm presented in Section 6.6 on the partially ordered traces generated from the MSCs. In the context of our process discovery problem, this algorithm seems to fit perfectly, however, there are two important requirements, namely that partial orders are minimal and that input and output sets are uniquely labelled.
Partial Orders need to be minimal.

The aggregation algorithm presented in 6.6 assumes that the partial orders used as input are minimal (cf. Property 6.5.5), i.e., if there is an edge between two nodes, there is no other path between them. This requirement is clearly not met by our MSCs, i.e., if Process A sends message a to Process B and then gets message b back from Process B, i.e., there are two paths between the audit trail entry referring to the sending message a and the one referring to the receiving of message b (see Figure 7.2). Therefore, we apply the concept of transitive reduction [29] to each MSC, before aggregating them.

The process of transitive reduction is the opposite of transitive closure, i.e. edges are removed from a graph if there is another path in that graph connecting the source and the target node. In general, the transitive reduction of a graph is not unique. However, for a-cyclic graphs it is, which makes this technique applicable in our situation.

Input and Output sets are uniquely labelled.

The second requirement for our aggregation algorithm is that no single audit trail entry is preceded or succeeded by two audit trail entries with the same label twice (cf. Property 6.5.6). Violating this requirement would result in the situation where the types of splits and joins can not be determined in the aggregation process. However, in MSCs, this requirement might be violated.

Consider Figure 7.4, where the start event of message a by Agent B is followed by two complete events of the same message, i.e. the one coming in from Agent A and the one going to Agent C. If we make sure that such a situation does not occur, we can be certain that the aggregation algorithm produces a model that can replay the MSC. Furthermore, it is trivial to detect this situation in an MSC and it can be corrected by adding an internal event on the lifeline of Agent B, between the sending of message a to Agent C and the receiving of message a from Agent A.
7.1.4 Bookstore Example

In this subsection we present an application of our process discovery approach to a system modelled by MSCs. The system that we describe is an online bookstore. The system contains the following four MSC agents:

**Customer** The person that wants to buy a book.

**Bookstore** Handles the customer request to buy a book. The virtual bookstore always contacts a publisher to see if it can handle the order. If the bookstore cannot find a suitable publisher, the customer’s order is rejected. If the bookstore finds a publisher, a shipper is selected and the book is shipped. During shipping, the bookstore sends a bill to the customer.

**Publisher** May or may not have the book that the customer wants to buy.

**Shipper** May or may not be able to carry out the shipping order.

We start with some message sequence charts. In Figure 7.5 we show two examples where the customer orders a book at the bookstore. The bookstore then tries to get the book from different publishers (one in the left hand figure and two in the right hand figure) until it gives up and tells the customer that the order cannot be fulfilled. Note that in both cases, the shipper does not do anything.

We started from 10 of such MSCs, containing 23 different messages and we translated them into MXML using the plug-in for ProM we developed. Then, we used the process mining technique described in Section 6.6 to obtain process models for the bookstore, the publisher, the shipper and the customer, by projecting each process instance on the corresponding audit trail entries.

The three process models depicted in Figure 7.6 are the result of projecting the log onto the bookstore, i.e., we only took into account the audit trail entries.

![Figure 7.5: Two MSCs describing scenarios in our bookstore example.](image)
that had the bookstore as an originator. The Petri net at the top of the screenshot shows the whole bookstore process discovered using the approach described in this section. ProM can transform models from one notation to another, e.g., mapping Petri nets onto EPCs, YAWL, BPEL, etc (we introduce these conversion plug-ins in detail in Section 8.5). The EPC in the bottom-right corner and the YAWL model in the bottom-left corner, show only a part of the process. The EPC shows the part of the Petri net in the small oval and the visible YAWL model corresponds to the part in the big oval. Furthermore, a dialog is shown resulting from an analysis of the EPC model. The EPC analysis plug-in in ProM reports that the EPC is a correct EPC, as defined in Chapter 5.

7.1.5 Mining Quality

This section presented a new approach to synthesize a process model from MSCs. In contrast to typical process discovery, the input is not a process log with sequences of events, but a set of MSCs, explicitly showing scenarios of possible behaviour (where the causalities are stored in the MXML log file). The downside of this approach is twofold:

1. The example scenarios show what should happen in practice, not what is actually happening. Therefore, the resulting Petri net model or EPC also shows what should happen and not what actually happens, i.e. the discovered model might not conform to the log.

2. Example scenarios are typically far from exhaustive, i.e. if 5 scenarios are presented to model the intended behaviour of a process, the process might show hundreds of other scenarios once implemented in an information system. Therefore, special care has to be taken when using the resulting model when configuring an information system.

The two issues mentioned above make that the quality of the process model resulting from the aggregation of MSCs heavily depends on the quality of the input. Although the aggregation algorithm of Section 6.6 guarantees to result in a model that can replay the log, this does always not hold for MSCs, i.e. only if the MSCs indeed exactly describe the operational process this property does hold.

Finally, it is important to note that the ideas presented in this section are not limited to MSCs and can be applied to other event logs containing explicit causal dependencies.

7.2 Aggregating Runs of EPCs

The goal of process mining is to extract information about processes from a process log. In Chapter 6, we presented several algorithms that assume the
Section 7.2 Aggregating Runs of EPCs

Figure 7.6: ProM showing the mined bookstore process in different languages.
process log under consideration to contain linearly ordered sequences of events. In this section, we consider the situation where the initial process log already contains more information than just those sequences of events, i.e. it contains causal runs of EPCs or so-called \textit{instance EPCs}.

### 7.2.1 Instance EPCs

Since EPCs are a conceptual model for describing business processes, i.e. they are not intended to be executable, it seems unnatural that there exists something like a causal run of an EPC. However, we first encountered instance EPCs of EPCs in the performance monitoring toolset Aris PPM (IDS Scheer, [111]). Simply put, instance EPCs are a subclass of EPCs, namely EPCs without loops and without choice connectors, i.e. there are no OR and XOR connectors.

\textbf{Definition 7.2.1. (Instance EPC)}
\newline
Let $\ell\varepsilon = (N, E, C_{\text{and}}, C_{\text{xor}}, C_{\text{or}}, A, \Lambda, \gamma)$ be a labelled EPC. We say that $\ell\varepsilon$ is an instance EPC, if and only if $C_{\text{or}} = C_{\text{xor}} = \emptyset$ and the EPC is acyclic.

In Figure 7.7, we show three instance EPCs relating to our running example from Section 2.6. Note that the EPC in Figure 7.7(a) is only a fragment of an EPC, since the event “MVV granted” triggers new parts of the process. These example EPCs clearly show that the choice connectors that are present in the original EPC (cf. Figure 2.9) do not appear in the instance. Instead, these choices have been made and therefore, the connectors are no longer needed.
In Aris PPM, instance EPCs are used as an analysis method to allow the user of a process management system to drill down to specific cases and see how they went through the system. In fact, the instance EPC is used to display what actually happened in a specific case in a human-readable format. The construction of these EPCs is done by first manually introducing links from milestones in a process management system to a fragment of an EPC in PPM. Then, every time a milestone is reached in a case, the corresponding fragment is automatically put in the instance EPC and connected to the already existing fragments.

In Subsection 6.5.4, we have shown that we can translate instance graphs to EPCs and it is clear that the resulting EPC is indeed an instance EPC. Furthermore, in Section 6.6, we presented an algorithm to aggregate instance graphs into an aggregation graph, and we have again shown how this aggregation graph can be represented as an EPC. Recall that in both translations, we needed to generate a set of events by assuming that each function is preceded by one event with the same label. However, if our starting point is not a process log, but a set of instance EPCs, we have explicit knowledge about the events, both about their labels as about their connections, which makes the aggregation process more simple.

Figure 7.8: An example of an instance EPC in Aris PPM.
7.2.2 Aggregating instance EPCs

As we stated before, instance EPCs are EPCs that represent a single process instance. Furthermore, they contain both explicit references to functions and events. Especially the latter is what sets them apart from the EPC representation of an instance graph, i.e. each EPC representation of an instance graph is an instance EPC, but not all instance EPCs could be an EPC representation of an instance graph.

Consider for example Figure 7.9, where we show two possible ways to express an AND-split after function $A$ to functions $D$ and $E$. Note that Figure 7.9(a) could be part of an EPC representation of an instance graph (Definition 6.5.10), since functions $D$ and $E$ are directly preceded by two events $B$ and $C$, although these events should in that case be labelled $D$ and $E$ respectively (cf. Definition 6.5.10). Figure 7.9(b) cannot be part of an EPC representation of an instance graph, since functions $D$ and $E$ are not directly preceded by an event.

The process of aggregating instance EPCs is remarkably simple to explain. The idea is that you just map elements with the same label onto each other, while keeping consistency in the way your connectors take care of routing. Here the OR connector of EPCs is particularly useful. For example, if function $f$ generated two events $a$ and $b$ in one instance EPC and only generated event $a$ in another, then the connector connecting function $f$ and these two events simply becomes an OR-split connector.

The procedure described above is implemented in the Aris PPM tool, where a user can select a number of instance EPCs, for example those EPCs that represent cases that missed a certain deadline, and use the aggregation mechanism to see if these cases always follow a certain process. If the instance EPCs from Figure 7.7 would be aggregated, the result is the EPC as shown in Figure 7.10, which not surprisingly corresponds exactly with the original EPC as presented in Figure 2.9.

Note that if one would start with a normal sequential process log and use the approach of Section 6.5 to generate instance graphs for each process instance, these instance graphs could be represented as instance EPCs and the aggregation approach of this section could be used. However, this has no advantage over the approach of Section 6.6 since, in terms of the resulting EPC, the result would be the same.

7.2.3 Mining Quality

Although the idea behind aggregating EPCs is interesting, it has some problems. First of all, as EPCs are rarely used as executable models, it is highly unlikely that instance EPCs can be found in real life. In Aris PPM for example, these instance EPCs are generated by, at design time, manually mapping milestones in some information system to fragments of instance EPCs. Although this is a valid approach for the purpose of performance monitoring, it is not considered process
Section 7.2 Aggregating Runs of EPCs

(a) Figure 7.9: Two part of instance EPCs.

(b) Figure 7.10: Instance EPCs of Figure 7.7 aggregated into one EPC.
Figure 7.11: Aggregated EPC in Aris PPM.
mining, since deep knowledge of the process under consideration is required to construct such mappings.

Furthermore, by introducing the OR connector, information about what is actually happening is lost. The OR-join does not always have a clear semantics and may be interpreted in various ways, thus leading to confusion about the resulting process model.

Since we are now able to aggregate sets of generated instance graphs, as well as sets of instance EPCs, an interesting question is whether we can aggregate sets of causal runs of Petri nets, or causal nets.

### 7.3 Aggregating Runs of Petri nets

In Section 7.2, we have shown that instance EPCs contain explicit information about events that EPC representations of instance graphs do not. A similar property holds for Petri net representations of instance graphs, where the causal dependencies between two transitions are made explicit through places, but it is unknown how many places should appear between two transitions.

Therefore, in this section, we consider the situation where our input for process mining is not just a process log, but a set of causal runs of Petri nets or causal nets, which contain explicit information about places. Using these causal nets, we present three aggregation algorithms, each of which has different requirements on the information in the Petri nets, with respect to labels of transitions and places.

#### 7.3.1 Causal Nets

Similar to instance EPCs, causal runs of Petri nets are Petri nets without loops and without choices. More specifically, the Petri nets are safe, i.e. there is never more than one token in a place. This again can be explained by the absence of choice, since if a place would contain two tokens then the next transition that fires has to choose a token and there are no choices in the model.

Figure 7.12 shows two examples of causal runs of Petri nets. Note that in both examples, all elements (i.e. places and transitions) are labelled and that for Figure 7.12(b) the process is not finished when the event (transition) “Send documents to Town Hall” produces a token in condition (place) p5.

To illustrate the idea of causal runs of Petri nets, we introduce some concepts taken from [65]. First, we introduce the notion of a causal net, this is a Petri net specification of one process instance of a process.

**Definition 7.3.1. (Causal net)**

A marked P/T net \( \varphi = ((C, E, K), S_0) \) is called a causal net, if and only if:

- For every place \( c \in C \) holds that \( |\cdot c| \leq 1 \) and \( |c\cdot| \leq 1 \),
- The transitive closure of \( K \) is irreflexive, i.e. it is a partial order on \( C \cup E \),
- For all places \( c \in C \) holds that \( S_0(c) = 1 \) if \( \cdot c = \emptyset \) and \( S_0(c) = 0 \) if \( c \neq \emptyset \).
In causal nets, we refer to places as \textit{conditions} and to transitions as \textit{events}. Furthermore, we still restrict ourselves to nets where for all $e \in E$ holds that $\bullet e \neq \emptyset$.

Before, in Section 6.5.3, we have shown how we can generate instance graphs for all process instances in an event log, and how these instance graphs can be represented as Petri nets. Such Petri nets do not contain labels of places and hence they are similar to the input of the $NPL$-algorithm presented in this section. However, there is one important difference and that is the fact that the instance graphs do not contain information about \textit{how many} places there should be between two transitions, i.e. a causal dependency in the instance graph is always represented by 1 place and not more, whereas the causal nets in this section do contain that information (cf. Figure 7.12(a) places $p_2$ and $p_3$). Furthermore, causal nets are not necessarily minimal.

A causal net is typically generated by a process (remember that it represents an instance of a process). Therefore, each transition and place in a causal net should refer to a transition and place of some process specification respectively. This reference is made by mapping the conditions and events of a causal net onto places and transitions of a Petri net. We call the combination of a causal net and such a mapping a \textit{run}.

\textbf{Definition 7.3.2. (Petri net run)}

A Petri net run $\rho = (N, \alpha, \beta)$ of a marked P/T-net $((P, T, F), M_0)$ is a causal net $N = ((C, E, K), S_0)$, together with two mappings $\alpha : C \to P$ and $\beta : E \to T$, such that:

- For each event (transition) $e \in E$, the mapping $\alpha$ induces a bijection from $\bullet e$ to $\bullet \beta(e)$ and a bijection from $e \bullet$ to $\beta(e) \bullet$.
- $\alpha(S_0) = M_0$ where $\alpha$ is generalized to markings by $\alpha : (C \to \mathbb{N}) \to (P \to \mathbb{N})$, such that $\alpha(S_0)(p) = \sum_{c | \alpha(c) = p} S_0(c)$.

To avoid confusion, the marked P/T net $((P, T, F), M_0)$ is called \textit{system net} in the sequel and when we have a system net and all possible runs for that system net, we say that we have the causal behaviour of that system net.

\textbf{Definition 7.3.3. (Causal behaviour)}

The causal behaviour of a P/T net $((P, T, F), M_0)$ is defined as its set of possible runs.

In Subsection 7.3.2, we take a set of runs as a starting point. From these runs, we generate a system net describing the behaviour of all individual runs.

\subsection*{7.3.2 Aggregating Petri net runs}

Generally, when looking at Petri net runs, it is assumed that a Petri net, or so-called system net, is used to generate these runs. In this section, we introduce an approach that takes a set of runs as a starting point. From this set of runs, a
system net is constructed. However, to construct a system net, we need to find a mapping from all the events and conditions in the causal nets to the transitions and places in the constructed system net, i.e. for each causal net, we need to define mappings $\alpha$ and $\beta$, such that the causal net becomes a run of the generated system net.

From Definition 7.3.2, we know that there should exist a bijection between all places in the pre- or post-set of an event in some causal net, and the pre- or post-set of a transition in a system net. Therefore, two conditions belonging to the pre- or post-set of an event should not be mapped onto the same label. Note that this label corresponds to a place in the system net. Furthermore, this restriction is merely another way to express the fact that our P/T nets do not allow for more than one edge between a place and a transition or vice versa. To be able to express this property formally, we define a labelling function on the nodes of a graph as a function that does not give the same label to two nodes that have a common successor, or a common predecessor.

**Definition 7.3.4. (Labelling function)**

Let $\mu$ be a set of labels. Let $G = (N, E)$ be a graph. Let $R = \{(n_1, n_2) \in N \times N | n_1 \bullet \cap n_2 \bullet \neq \emptyset \vee n_1 \cap n_2 \neq \emptyset\}$. We define $f : N \rightarrow \mu$ to be a labelling function if and only if $f$ is a coloring function on the graph $(N, R)$ (cf. Definition 2.1.9 for the definition of a coloring function).

In this thesis, we focus on the aggregation of runs that originate from a Petri net with a clearly defined starting state and completion state, which is a natural assumption in the context of workflow management systems [5,14,82]. However,
Chapter 7 Process Discovery on Partially Ordered Logs

it applies to many other domains where processes are instantiated for specific cases. Hence, we will limit ourselves to runs originating from WF-nets.

Recall that a WF-net is a Petri net with one input place and one output place. When looking at runs of a WF-net, we can therefore conclude that in the initial marking, there is always exactly one condition containing a token and all the other conditions do not contain tokens. A set of causal nets that have this property is called a causal set.

**Definition 7.3.5. (Causal set)**

Let \( n \in \mathbb{N} \) and let \( \Phi = \{(C_i, E_i, K_i, S_i) \mid 0 \leq i < n\} \) be a set of causal nets. We call this set a causal set if and only if for all \( 0 \leq i < n \) holds that:

- All sets \( C_i, E_i, K_i, C_j, E_j \) and \( K_j \) are disjoint.
- \( \sum_{c \in C_i} S_i(c) = 1 \), i.e. there is exactly one condition with an empty preset,
- For all \( e \in E_i \) with \( c \in \bullet e \) such that \( S_i(c) = 1 \) holds that \( \{c\} = \bullet e \), i.e. each event in the postset of one of these conditions has only this initially marked condition in its preset,
- For all \( c \in C \) with \( c\bullet = \emptyset \) holds that \( \forall e \in \bullet c \ e\bullet = \{c\} \), i.e. each event in the preset of a condition with empty postset (representing a token on the place \( p_{out} \)) has only this condition in its postset.

To aggregate the causal nets in a causal set, we need to assume that these nets can all be runs of the same system net. Using Definition 7.3.2, we know that for this assumption to be valid, we need two mappings \( \alpha \) and \( \beta \). Using the definition of a system net and the relation between system nets and runs, we can conclude that any aggregation algorithm for Petri net runs should have the following functionality:

- It should provide the set of places \( P \) of the system net.
- It should provide the set of transitions \( T \) of the system net.
- It should provide the flow relation \( F \) of the system net.
- It should provide the initial marking \( M_0 \) of the system net.
- For each causal net in the causal set, it should provide the mappings \( \alpha_i : C_i \rightarrow P \) and \( \beta_i : E_i \rightarrow T \), in such a way that for all causal nets, \( \alpha_i(S_i) \) is the same (i.e. they have the same initial marking) and they induce bijections between pre- and post-sets of events and their corresponding transitions.

From Definition 7.3.2, we know that each event that appears in a causal net has a corresponding transition in the original system net. More important, however, is the fact that bijections exist between the pre- and post-sets of this event and the corresponding transitions. In order to express this in terms of labelling functions of causal nets, we formalize this using the notion of transition equivalence.
**Definition 7.3.6. (Transition equivalence)**

Let $\Phi = \{ N_i = ((C_i, E_i, K_i), S_i) \mid 0 \leq i < n \}$ be a causal set, let $\mu, \nu$ be two sets of labels, such that $\mu \cap \nu = \emptyset$ and let $\Psi = \{ (\alpha_i : C_i \to \mu, \beta_i : E_i \to \nu) \mid 0 \leq i < n \}$ be a set of labelling functions of $((C_i, E_i, K_i), S_i)$. We define $(\Phi, \Psi)$ to respect transition equivalence if and only if for each $i, j \in \{0, \ldots, n-1\}$, $e_i \in E_i$ and $e_j \in E_j$ with $\beta_i(e_i) = \beta_j(e_j)$ the following holds:

- for each $(p_i, e_i) \in K_i$ we have a $(p_j, e_j) \in K_j$ such that $\alpha_i(p_i) = \alpha_j(p_j),$
- for each $(e_i, p_i) \in K_i$ we have a $(e_j, p_j) \in K_j$ such that $\alpha_i(p_i) = \alpha_j(p_j)$.

Using the concepts of a causal set and transition equivalence, we introduce three aggregation algorithms with different requirements on the available information. First, in Subsection 7.3.3, we introduce an algorithm to aggregate causal nets where all places and transitions have known labels. Then, in Subsection 7.3.4, we show an algorithm that can deal with the situation where different transitions have the same label. The final algorithm, presented in Subsection 7.3.5, deals with the situation where transitions are correctly labelled, but places are not labelled at all. These algorithms have been presented before in [76, 77] and for the proofs that these algorithms indeed produce correct system nets, we refer to those papers.

### 7.3.3 Aggregation with Known Labels

In this section, we present an aggregation algorithm that assumes that we know all mapping functions, and that these mapping functions adhere to the definition of a run. To illustrate the aggregation process, we make use of a running example that will come back later in this section. Consider Figure 7.13 where four fragments of runs are shown. It is important to realize that these are not complete runs, but merely fragments of runs. We do assume however that the events $A, B, C, D, E, F$ and $G$ do not appear in any other fragment of each run.

Our first aggregation algorithm in the context of causal nets is called the $ALK$ aggregation algorithm (short for “All Labels Known”). This algorithm has been investigated before in [42] and assumes all information such as presented in Figure 7.13 to be present, i.e. it assumes known labels for events and known labels for conditions. These labels refer to concrete transitions and places in the aggregated system net.

![Figure 7.13: Four examples of fragments of runs, showing conditions referring to places $p_1$ and $p_2$ and their neighborhoods.](image-url)
**Definition 7.3.7. (ALK aggregation algorithm)**

Let $\mu, \nu$ be two sets of labels, such that $\mu \cap \nu = \emptyset$. Let $\Phi$ be a causal set of size $n$ and let $((C_i, E_i, K_i), S_i) \in \Phi$ correspond to a causal net with $0 \leq i < n$. Furthermore, let $\{(\alpha_i : C_i \rightarrow \mu, \beta_i : E_i \rightarrow \nu) \mid 0 \leq i < n\}$ be a set of labelling functions respecting transition equivalence, such that for all causal nets $\alpha_i(S_i)$ is the same. We construct the system net $((P, T, F), M_0)$ belonging to these runs as follows:

- $P = \bigcup_{0 \leq i < n} \text{rng}(\alpha_i)$ is the set of places (note that $P \subseteq \mu$)
- $T = \bigcup_{0 \leq i < n} \text{rng}(\beta_i)$ is the set of transitions (note that $T \subseteq \nu$),
- $F = \bigcup_{0 \leq i < n} \{(\alpha_i(c), \beta_i(e)) \mid (c, e) \in K_i \cap (C_i \times E_i)\} \cup \bigcup_{0 \leq i < n} \{(\beta_i(e), \alpha_i(c)) \mid (e, c) \in K_i \cap (E_i \times C_i)\}$
  - is the flow relation,
- $M_0 = \alpha_0(S_0)$ is the initial marking.

The result of the ALK aggregation algorithm from Definition 7.3.7 for the fragments presented in Figure 7.13 is shown in Figure 7.14.

It is easy to see that the aggregated net shown in Figure 7.14 can actually generate the runs shown in Figure 7.13, a property that has been proven in [76, 77].

The aggregation algorithm presented in Definition 7.3.7 is a rather trivial aggregation over a set of runs. However, it is assumed that the mapping functions $\alpha_i$ and $\beta_i$ are known for each causal net, in such a way that Definition 7.3.2 is followed. Furthermore, we assume two sets of labels $\mu$ and $\nu$ to be known. However, when applying these techniques in the context of process mining, it is often not realistic to assume that all of these are present. Therefore, we relax some of these assumptions to obtain more usable process mining algorithms.

### 7.3.4 Aggregation with Duplicate Transition Labels

In the domain of process mining, the problem of so-called “duplicate transitions” (i.e. several transitions with the same label) is well-known (cf. [8, 35, 108]). Therefore, there is a need for algorithms to find out which events actually belong to which transition. Therefore, unlike before, we assume that the causal set used to
generate the system net and the labelling functions, does not respect transition equivalence and we introduce an algorithm to change the labelling function for events in such a way that this property holds again.

We assume that we have causal nets with labelling functions, where some events have the same label, even though they may refer to different transitions. This is illustrated in Figure 7.15. Note that this figure is similar to Figure 7.13, except that we now labelled events F and G both with a new label X.

In terms of an aggregation algorithm, the problem of duplicate labels translates to the situation where the property of transition equivalence (Property 7.3.6) is not satisfied. Since the aggregation algorithm presented in the previous section only works if this property holds, we provide an algorithm to redefine the labelling functions for events. Furthermore, we will prove that after applying this algorithm, the desired property holds.

**Definition 7.3.8. (Relabelling algorithm)**

Let $\mu, \nu$ be two sets of labels, such that $\mu \cap \nu = \emptyset$. Let $\Phi = \{N_i \mid 0 \leq i < n \land N_i = ((C_i, E_i, K_i), S_i)\}$ be a causal set, and let $\Psi = \{((\alpha_i : C_i \rightarrow \mu, \beta_i : E_i \rightarrow \nu) \mid 0 \leq i < n\}$ be a set of labelling functions in $((C_i, E_i, K_i), S_i)$, such that $\alpha_i(S_i)$ is the same for all causal nets. Furthermore, assume that $\mu$ and $\nu$ are minimal, i.e. $\bigcup_{0 \leq i < n} \text{rng}(\alpha_i) = \mu$ and $\bigcup_{0 \leq i < n} \text{rng}(\beta_i) = \nu$. Let $E^* = \bigcup_{0 \leq i < n} E_i$ be the set of all events in the causal set. We define the relabelling algorithm as follows:

1. Define $\sim \subseteq E^* \times E^*$ as an equivalence relation on the elements of $E^*$ in such a way that $e_i \sim e_j$ for some $i, j \in \{0, \ldots, n - 1\}$ with $e_i \in E_i$ and $e_j \in E_j$ if and only if $\beta_i(e_i) = \beta_j(e_j)$, and $\alpha_i(e_i^{N_i}) = \alpha_j(e_j^{N_j})$.

2. For each $e \in E^*$, we say $\text{eqvl}(e) = \{e' \in E^* \mid e \sim e'\}$.

3. Let $\nu'$ be the set of equivalence classes of $\sim$, i.e. $\nu' = \{\text{eqvl}(e) \mid e \in E^*\}$.

4. For all causal nets $((C_i, E_i, K_i), S_i)$ and labelling functions $\alpha_i$, define a labelling function $\beta'_i : E_i \rightarrow \nu'$ such that for an event $e_i$, $\beta'_i(e_i) = \text{eqvl}(e_i)$, i.e. it returns the equivalence class of $\sim$ containing $e_i$.

After re-labelling the events, the fragment of the run shown in Figure 7.15(d) is relabelled to include the pre- and postconditions. Figure 7.16(a) shows the fragment before relabelling, whereas Figure 7.16(b) shows the fragment after relabelling. However, in Figure 7.16(b) we only show the relabelling with respect to the post conditions.
Applying the ALK algorithm of Definition 7.3.7 to the relabelled runs yields the result as shown in Figure 7.17. Note that we do not show the $\nu'$ labels explicitly, i.e. $B$ refers to the equivalence class of events labelled $B$ and $X_{p1}$ refers to the equivalence class of events labelled $X$, having output place $p1$.

![Figure 7.16](image1)

*Figure 7.16:* The original and relabelled fragment of Figure 7.15(d).

![Figure 7.17](image2)

*Figure 7.17:* A fragment of the aggregated net.

Again, the proof that this algorithm is capable of finding events that have the same label, but correspond to different transitions in the system net has been proven in [76, 77]. However, when no transition labels are known at all, the same algorithm can be applied to find all transition labels, by using an initial $\nu = \{\tau\}$ and initial mapping functions $\beta_i$, mapping everything onto $\tau$. However, in that case, no distinction can be made between events that have the same pre- and post set, but should have different labels.

### 7.3.5 Aggregation with Unknown Place Labels

Before, we have shown a way to identify the transitions in a system net, based on the labels of events in causal nets. However, are more relevant question is: what if condition labels are not known? For the aggregation algorithm we present in this section, we take one step back. We assume all events to refer to the correct transition and we try to identify the labels of conditions. In Figure 7.18, we again show our small example of the aggregation problem, only this time there are no labels for conditions $p1$ and $p2$, which we did have in figures 7.13 and 7.15.
Consider the four runs of Figure 7.18. Recall that they show parts of causal nets, in such a way that the tasks $A, B, C, D, E, F$ and $G$ do not appear in any other way in another causal net. In contrast to the algorithms presented previously, we cannot always derive a unique aggregated system net for causal nets if we do not have labels for the conditions. Instead, we define an aggregation class, describing a class of WF-nets that could have generated these causal nets. Table 7.2 shows some requirements all WF-nets in the aggregation class of our example should satisfy.

The information in Table 7.2 is derived using the concept of a segment, which can be considered to be the context of a condition in a causal net.

**Definition 7.3.9. (Segment)**

Let $N = ((C, E, K), S_0)$ be a causal net and let $N' = (C', E_{in}, E_{out})$ be such that $C' \subseteq C$, $E_{in} \cup E_{out} \subseteq E$ and $E_{in} \neq \emptyset$ and $E_{out} \neq \emptyset$. We call $N'$ a segment if and only if:

- For all $c \in C'$ holds that $\bullet c \subseteq E_{in}$ and $c \bullet \subseteq E_{out}$, and
- For all $e \in E_{in}$ holds that $e \bullet \subseteq C'$, and
- For all $e \in E_{out}$ holds that $\bullet e \subseteq C'$, and

We say that $E_{in}$ represents the input events and $E_{out}$ the output events. Furthermore, a segment is minimal if and only if $C'$ is minimal, i.e. there does not exist a segment $N'' = (C'', E'_{in}, E'_{out})$ with $C'' \subset C'$ and $C'' \neq \emptyset$.

Figure 7.19 shows a minimal segment in a causal run, where $E_{in}$ contains two
transitions, $E_{out}$ also contains two transitions and $C'$ contains 4 places. For the fragments of Figure 7.18, it is easy to see that each of them contains only one minimal segment, where the input events are the events on the left hand side and the output events are the events on the right hand side.

The meaning of a segment is as follows. If we have a run and a segment in that run, then we know that after each of the events in the input set of the segment occurred, all the events in the output set occurred in the execution represented by this run. This translates directly to a marking in a system net, since the occurrence of a set of transitions would lead to some marking (i.e. a bag over places), which enables another set of transitions. Furthermore, each transition only produces one token in each output place. Combining this leads to the fact that for each segment in a causal net, the aggregated system net should show that the bag of places following the transitions corresponding to the input events of the segment should be the same as the bag of places preceding the transitions corresponding to the output set of events, as indicated in Table 7.2.

Clearly, when looking only at these fragments, what we are looking for are the places that should be put between tasks $A, E, F$ and $G$ on the one hand, and $B, C$ and $D$ on the other hand. Therefore, we only focus on this part of the causal nets. For this specific example, there are two possibilities, both of which are equally correct, namely the two WF-net fragments shown in Figure 7.20.

From the small example, we have seen that it is possible to take a set of causal nets without labels for any of the conditions (but with labels for all the events) and to define a class of WF-nets that could be system nets of the causal nets. In

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
Fragment & Conclusions \\
\hline
Fig. 7.18 (a) & $A \bullet = \bullet B \uplus \bullet C$ \\
Fig. 7.18 (b) & $A \bullet = \bullet D$ \\
Fig. 7.18 (c) & $E \bullet = \bullet B$ \\
Fig. 7.18 (d) & $F \bullet \uplus G \bullet = \bullet D$ \\
\hline
\end{tabular}
\caption{Information derived from runs shown in Figure 7.18.}
\end{table}

Figure 7.20: Two fragments of aggregated nets.
the remainder of this section, we show that this is indeed possible for all causal sets.

Before actually presenting the NCL algorithm (which stands for “No Condition Labels”), we first take a look at a more intuitive example. Consider Figure 7.21, where we present three causal nets, each of which corresponds to a paper review process. In the first causal net, three reviewers are invited to review the paper and after the three reviews are received, the paper is accepted. In the second causal net, only two reviews are received (the third one is not received on time), but the paper is rejected nonetheless (apparently the two reviewers that replied rejected the paper). In the third example only one review is received in time, and therefore an additional reviewer is invited, who hands in the review in time, but does not accept the paper.

As we stated before, we define an aggregation class of a causal set, that contains all WF-nets that are capable of generating the causal nets in the causal set. The information needed for this aggregation class comes directly from the

![Diagram](attachment:image.png)

**Figure 7.21:** Three causal nets of a review process of a paper.
Table 7.3: Information derived from review example.

<table>
<thead>
<tr>
<th>Causal net</th>
<th>Conclusions on transitions in the aggregation class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fig. 7.21 (a)</td>
<td><img src="image1" alt="Diagram" /></td>
</tr>
<tr>
<td>“Invite reviewers”</td>
<td><img src="image1" alt="Diagram" /></td>
</tr>
<tr>
<td>“Invite reviewers”</td>
<td><img src="image1" alt="Diagram" /></td>
</tr>
<tr>
<td>“Get review 1”</td>
<td><img src="image1" alt="Diagram" /></td>
</tr>
<tr>
<td>“Get review 2”</td>
<td><img src="image1" alt="Diagram" /></td>
</tr>
<tr>
<td>“Get review 3”</td>
<td><img src="image1" alt="Diagram" /></td>
</tr>
<tr>
<td>“Collect &amp; Decide”</td>
<td><img src="image1" alt="Diagram" /></td>
</tr>
<tr>
<td>“Accept paper”</td>
<td><img src="image1" alt="Diagram" /></td>
</tr>
<tr>
<td></td>
<td><img src="image1" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image1" alt="Diagram" /></td>
<td><img src="image1" alt="Diagram" /></td>
</tr>
</tbody>
</table>

| Fig. 7.21 (b) | ![Diagram](image1) |
| “Invite reviewers” | ![Diagram](image1) |
| “Invite reviewers” | ![Diagram](image1) |
| “Get review 1” | ![Diagram](image1) |
| “Get review 2” | ![Diagram](image1) |
| “Time-out review 3” | ![Diagram](image1) |
| “Collect & Decide” | ![Diagram](image1) |
| “Reject paper” | ![Diagram](image1) |
| | ![Diagram](image1) |
| ![Diagram](image1) | ![Diagram](image1) |

| Fig. 7.21 (c) | ![Diagram](image1) |
| “Invite reviewers” | ![Diagram](image1) |
| “Invite reviewers” | ![Diagram](image1) |
| “Time-out review 1” | ![Diagram](image1) |
| “Time-out review 2” | ![Diagram](image1) |
| “Get review 3” | ![Diagram](image1) |
| “Collect & Decide” | ![Diagram](image1) |
| “Invite add. reviewer” | ![Diagram](image1) |
| “Get add. review” | ![Diagram](image1) |
| “Reject paper” | ![Diagram](image1) |
| | ![Diagram](image1) |
causal nets, using segments. In Table 7.3, we present the conclusions we can draw based on the three causal nets. Note that in this table, just like in Table 7.2, we consider bags of pre- and post-sets of transitions in the aggregation class. The information in this table is obtained from the causal nets in the following way. Consider for example Figure 7.21(a), where “invite reviewers” is followed by “Get review 1”, “Get review 2” and “Get review 3”. This implies that the bag of output places of “invite reviewers”, should be the same as the sum over the bags of the input places of “Get review 1”, “Get review 2” and “Get review 3”.

Consider the information presented in Table 7.3 and the two Petri nets in Figure 7.22, where both nets adhere to all constraints of Table 7.3. As this example again shows, there is no unique Petri net satisfying all constraints. Instead, there is a class of nets satisfying all constraints and therefore, we use the constraints sketched in Table 7.3 to define an aggregation class.

![Diagram](image)

**Figure 7.22:** Two possible aggregated nets, both obeying the constraints of Table 7.3.
Definition 7.3.10. (Aggregation class)

Let $\Phi = \{(C_i, E_i, K_i, S_i) \mid 0 \leq i < n\}$ be a causal set, and let $\varphi = ((P, T, F), M_0)$ be a marked WF-net. For each causal net, let $\beta_i : E_i \to T$ be a mapping from the events of that causal net to $T$, such that $\beta_i$ is a labelling function for $((C_i, E_i, K_i), S_i)$. We define $A_\Phi$, the aggregation class of $\Phi$, as the set of all pairs $((\varphi, B), \text{such that the following conditions are satisfied:})$

1. $T = \bigcup_{0 \leq i < n} \text{rng}(\beta_i)$ is the set of transitions, i.e. each transition appears as an event at least once in some causal net,

2. $B$ is the set of all labelling functions, i.e. $B = \{\beta_i \mid 0 \leq i < n\}$. We use $\beta_i \in B$ to denote the labelling function for events belonging to $((C_i, E_i, K_i), S_i) \in \Phi$.

3. For all $p \in P$ holds that $\bullet p \cup p \bullet \neq \emptyset$,

4. $M_0 = [p_{\text{ini}}]$ and $\bullet p_{\text{ini}} = \emptyset$,

5. For each causal net $\gamma = ((C_i, E_i, K_i), S_i)$, with $e \in E_i$ and $\beta_i(e) = t$ holds that if $S_i(\bullet e) = 1$ then $p_{\text{ini}} \in \bullet t$,

6. For each causal net $\gamma = ((C_i, E_i, K_i), S_i)$, with $e \in E_i$ and $\beta_i(e) = t$ holds that $|t\bullet| = |e\bullet|$ and $|\bullet t| = |\bullet e|$,

7. For each causal net $\gamma = ((C_i, E_i, K_i), S_i)$, with $e \in E_i$, $\beta_i(e) = t$ and $T' \subseteq T$ holds that $|t\bullet \cap \bigcup_{t' \in T'} (\bullet t')| \geq \sum_{e' \in E_i, \beta(e') \in T'} |e'\bullet \cap \bullet e'|$,

8. For each causal net $\gamma = ((C_i, E_i, K_i), S_i)$, with $e \in E_i$, $\beta_i(e) = t$ and $T' \subseteq T$ holds that $|\bigcup_{t' \in T'} (t'\bullet) \cap \bullet t| \geq \sum_{e' \in E_i, \beta(e') \in T'} |e'\bullet \cap \bullet e|$,

9. For each causal net $\gamma = ((C_i, E_i, K_i), S_i)$ and minimal segment $(C'_i, E_{\text{in}}, E_{\text{out}})$ of $\gamma$, holds that $\bigcup_{e \in E_{\text{in}}} (\bullet \beta_i(e)) = \bigcup_{e \in E_{\text{out}}} (\bullet \beta_i(e))$.

Figure 7.23 is shown to provide more insight into the $9^{th}$ condition of Definition 7.3.10. In the lower causal net of that figure, there is a token travelling from $A$ to $D$ and from $B$ to $C$. The upper causal net on the other hand only connects $A$ and $C$. Assuming that these are the only causal nets in which these transitions appear, we know that the condition between $A$ and $D$ and between $B$ and $C$ should represent a token in the same place, since there is a minimal segment $\{(c_4, c_5, c_6), \{A, B\}, \{C, D\}\}$ in the lower causal net and therefore, $A \bullet \cup B \bullet = \bullet C \cup \bullet D = [p_1, 2p_2]$.

Definition 7.3.10 defines a finite class of WF nets for a causal set. What remains to be given are the conditions under which it is a finite non-empty class of Petri nets and the proof that each Petri net with its mappings is indeed a system net for the causal set. For this, we again refer to [76, 77], where it is shown that each element of the aggregation class can indeed serve as a system net for the runs and that if we start from a sound WF-net as a system net, generate a set of runs and remove the labels of places, the original WF-net is in the aggregation class.
7.3.6 Mining Quality

In this section, we presented the concept of causal nets, i.e. partial orders on events in the form of Petri nets. Using these causal nets, we presented three aggregation algorithms to construct a so-called system net, i.e. an aggregated net. Each of these algorithms has different requirements on the causal nets. For the first algorithm, all places and transitions need to be labelled and this labelling needs to adhere to the concept of transition equivalence. The second algorithm lifts that restriction by finding equivalence classes of transitions and is thus able to discover Petri nets with duplicate transitions. The first two algorithms make use of the labels of places. However, the third algorithm does not, i.e. it assumes that places are not labelled. The result is than no longer a unique aggregated WF-net, but a class of nets which are correct system nets.

Unlike EPCs, Petri nets are executable specifications and therefore, it is more likely that causal nets are found in real life, thus making the aggregation algorithms presented in this section a valuable addition to the field of process discovery.

7.4 Conclusion

In this chapter, we presented three partial order-based aggregation approaches, where, in contrast to the input for the algorithms in Chapter 6, the input is no longer a process log, but a set of partially ordered cases, in the form of Message Sequence Charts, instance EPCs or Petri net runs.

Recall that the aim of process discovery is to aid a process designer in improving the support for a process by an information system. Hence, it heavily depends on the specific information that can be taken from an information system, which process discovery approach best suits the designer. Therefore, if the logs taken from an information system contains more information than just sequences of events, i.e. in the form of partial orders, specialized aggregation algorithms as
presented in Section 7.2 for EPCs and Section 7.3 for Petri nets are the better choice.

To conclude, it is important to realize that the EPC-based aggregation approach presented in this chapter is used in practice in the commercial tool Aris PPM. Most situations where this tool is implemented are typical examples of situations where more than just a sequential log is available as input.
Chapter 8

ProM

For most of the techniques presented in this thesis, good tool support is essential, i.e. a verification approach for EPCs is only useful if there is a corresponding implementation that can work with EPCs taken from modelling tools used in industry. Furthermore, any implementation of process mining algorithms, such as the ones presented in Chapter 6, should allow for its result to be analyzed and compared against other results. The (Pro)cess (M)ining framework ProM

![Diagram](image.png)

**Figure 8.1:** Overview of the part of this thesis implemented in ProM.
was developed for this purpose. Although ProM started out as a framework for process mining, it has become much more versatile, currently including many analysis and conversion algorithms, as well as import and export functionality for many formalisms, such as EPCs and Petri nets.

The ProM framework has been developed as a workbench for process mining and related subjects, such as verification and analysis. In Chapter 2, we presented the MXML log format, which lies at the basis of the process mining algorithms contained in ProM, such as the ones presented in Chapter 6 and Chapter 7. ProM however is not just a collection of process mining algorithms. Instead, it contains implementations of a large number of related subjects, such as analysis or conversion algorithms, e.g. including the verification algorithms presented in Chapter 4 and Chapter 5.

In this chapter, we discuss the ProM framework in detail and we specifically discuss the implementations that have been developed in the context of this thesis, an overview of which is provided in Figure 8.1. However, we first elaborate on the architecture of ProM in general.

An important feature of the ProM framework is that it allows for interaction between a large number of so-called plug-ins. A plug-in is basically the implementation of an algorithm that is of some use in the process mining area, where the implementation agrees with the framework. Such plug-ins can be added to the entire framework with relative ease: Once the plug-in is ready it can be added to the framework by adding its name to an ini-file. Note that there is no need to modify the ProM framework (e.g., recompiling the code) when adding new plug-ins, i.e., it is a truly “plug-able” environment.

In Figure 8.2, we show an overview of the framework that we developed. It explains the relations between the framework, the process log format MXML, external tools and the plug-ins. As Figure 8.2 shows, the ProM framework can read files in the MXML format, which it does through the Log Reader component. This component is able to deal with large data sets and complex log files. Furthermore, it sorts the events within each case on their timestamps before actually presenting the data to the framework. (If no timestamps are present, the order in the XML file is preserved.) Furthermore, the log reader uses Log Filters, that filter the log by manipulating process instances. In Section 8.2, we discuss some of the standard filters in detail.

Through the Import plug-ins a wide variety of models, or objects, can be loaded ranging from a Petri net to a set of LTL formulas. The Mining plug-ins are implementations of process mining algorithms which store their results in memory, and visualize these results in a window on the ProM desktop. The framework allows plug-ins to operate on each others results in a standardized way. Typically, the mining results contain some kind of visualization, e.g., displaying a Petri net or an EPC. The Analysis plug-ins take a number of models or other objects and perform some analysis on it, for example by checking for soundness.
The Conversion plug-ins take a model and transform it into another format, e.g., transforming an EPC into a Petri net.

Next to EPCs and Petri nets, of which we present several import and export formats in Section 8.6, ProM supports the following formalisms in some way:

**Aggregation graphs**, which we introduced in Section 6.6. These aggregation graphs can be converted to EPCs.

**YAWL models**, which can be read and executed by the workflow management system YAWL. YAWL models can be obtained by converting a WF-net. Furthermore, YAWL models can be converted into EPCs.

**Heuristic nets**, which are a special class of nets used in [32,128] and which can be converted into EPCs and Petri nets.

**Causal Footprints**, which were introduced in [81] and represent abstractions of process models. They can be obtained from an EPC or a Petri net.

**BPEL processes**, which can be imported using an import plug-in for BPEL 1.1. These models can then be converted to Petri nets.
In the remainder of this section, we first present the ProM\textit{import} framework, which is a general framework for converting process logs taken from different sources into MXML. Then, we discuss the log filters and the five plug-in types in detail and we show which plug-ins of each type have been implemented in the context of this thesis. In Section 8.7, we conclude this chapter with an example walkthrough of ProM, i.e. we show how to use ProM on a practical example.

### 8.1 ProM\textit{import}

In Section 2.2, we introduced the MXML format for storing process logs. However, each information system in practice has its own logging functionality and its own logging format and therefore, to allow for process mining in a standard way, translations need to be defined from many proprietary formats to MXML. For this purpose, the ProM\textit{import} framework has been developed by Günther et al. [94]. This framework provides an intuitive interface between many known information systems and MXML.

A well known tool for modelling and simulating colored Petri nets is CPN Tools [55] and we have used CPN Tools to generate log files by simulating Petri net models. Figure 8.3 shows the user interface of ProM\textit{import}, where the CPN Tools filter is selected to convert CPN Tools logs to MXML.

Once a process log has been translated to MXML, it can be used by the ProM framework. Since process logs can typically be very large and contain a lot of information, the first step in process mining will often be the filtering of the log.

### 8.2 Log Filters

In Section 2.2.4, we presented the concept of a log filter, i.e. a function that typically transforms one process instance into another process instance, for example by removing all but the “complete” events. In the context of this thesis, many of such log filters have been developed, which we discuss in detail below. However, we first introduce how filters are applied to a log.

#### 8.2.1 Application of Log Filters

Filtering logs is a procedure by which a log is made easier to analyze and typically less sensitive to noise. Most of the filters presented in this section are removing information from a process instance, i.e. by projecting it onto a set of tasks or by removing specific instances. In this context it is important to realize that a process instance originally is a linear order on the audit trail entries and that, by removing an audit trail entry, the resulting ordering is still linear.

Instead of removing information, information can also be added. For example in the case where an assumption is made that all cases start with the same log
event, as we assumed in our discussion about the Theory of Regions. A log filter could just add such an initial event to each instance and the assumption is valid. ProM does contain such a log filter, called the “Add Artificial Start Task Log Filter”, which was presented in [128].

Furthermore, it can be important to apply more than one log filter at once, e.g. one could add one filter that removes all schedule events from a log and one filter that checks whether $A_{\text{complete}}$ is the first event of each process instance. Obviously, the order in which these filters are applied matters. Therefore, ProM allows the user to stack the log filters. When applying the stacked filters, they are applied top-down.

Figure 8.4, shows an example of a stack of log filters. It shows three filters, the first of which calculated the transitive closure of a partial order (cf. Subsection 8.2.7), the second of which removes all audit trail entries that do not correspond to a complete event (cf. Subsection 8.2.2) and the third of which calculates the transitive reduction of a partial order (cf. Subsection 8.2.8). When applying this stack of filters, ProM applies them top-down.

Finally, we note that if a log filter removes all audit trail entries from a pro-

![Figure 8.3: The ProMimport framework.](image-url)
cess instance, ProM automatically discards that process instance, when reading through the log.

8.2.2 Default Filter

The first filter that was developed originated from an earlier implementation of the α-algorithm, cf. [10, 70]. Recall the structure of MXML logs, presented in Figure 2.4, where it is shown that a log can contain more than one process, that each process consists of zero or more process instances and that each process instance consists of zero or more audit trail entries, which always refer to a workflow model element (i.e. an activity) and an event type.

The Default Log Filter is a filter that allows the user to select one process out of the ones contained in one MXML file. Each process instance that does not belong to that process is then transformed into an empty instance, which the ProM framework automatically ignores. Furthermore, in the Default Log Filter, the user has three options for each event type that appears in the log:

- **Discard instance**, in which case the whole process instance is ignored if it contains one audit trail entry referring to this event type,
- **Ignore**, in which case the filter will remove all audit trail entries from the log that refer to this event type,
- **Include**, in which case no action is taken on an audit trail entry referring to this event type (provided that the process instance it belongs to is not discarded.).
Section 8.2 Log Filters

Figure 8.5: Settings dialog of default filter.

Figure 8.6: Final event filter settings.
The first of the three is typically useful to filter out process instances that show undesired behaviour, such as the abortion of a case or a task. The “ignore” option is typically used to filter the log to such an extent that it only shows the most basic behaviour, i.e. for example by removing all events except the “complete” events.

Figure 8.5 shows the settings dialog for the default log filter. In this case, the filter is set to accept the “DEFAULT” process, to include “complete” events and to ignore “start” events.

In some algorithms, the log is required to start with a specific log event (recall that a log event is a combination of an activity and an event type, such as \(A_{\text{assign}}\)). For this purpose, we developed the start event log filter.

### 8.2.3 Start Event Filter

The start event log filter is a rather simple filter that checks if the first audit trail entry of a process instance (specifically, its activity and event type) is contained in a given set of allowed log events. If this is not the case then the process instance is discarded.

Similar to the start event, we can also be interested in the final event, therefore we developed the final event log filter.

### 8.2.4 Final Event Filter

This final event log filter is very similar to the start event log filter, however, it checks whether the final audit trail entry belongs to a given set of log events. The settings dialog for this filter, which is the same as the settings dialog for the start event filter is shown in Figure 8.6, where the filter is configured to only accept finished cases, i.e. those cases where the final task is either “Send MVV acceptance letter”, “Send MVV rejection letter”, “Send permit rejection letter” or “Send residence permit”.

### 8.2.5 Event Filter

Especially logs taken from large and complex information systems typically contain many tasks. In such situations, it can be useful to project the log onto a set of tasks, i.e. to only consider a specific set of tasks and just ignore all others. That is what the event filter does, i.e. a set of tasks can be selected after which all others are removed from all process instances.

### 8.2.6 Duplicate Task Filter

The Theory of Regions approach presented in Section 6.2 only works if a transition system does not contain two successive edges referring to the same transition.
Figure 8.7: Both the default filter and the final event filter applied to the log.
Therefore, the *duplicate task log filter* can be used, which removes all but the first of a sequence of the same audit trail entries.

### 8.2.7 Transitive Closure Filter

In Chapter 6 and Chapter 7, we introduced several algorithms that work on a partial order of events instead of on a linear one. In ProM, such a partial order is reflected in the data part of the audit trail entries, by storing pointers to the predecessors and successors explicitly. However, in a partial order, we cannot remove an audit trail entry while still keeping the ordering intact, e.g. consider the situation where audit trail entry $A$ is followed by $B$ and $B$ is followed by $C$. If $B$ would be removed, the relation between $A$ and $C$ is not automatically restored. Therefore, we developed two filters specifically for this purpose.

The first of these two filters is the *transitive closure filter* calculates the transitive closure of a partial order and stores this in the audit trail entries’ data field. Once the transitive closure of a partial order is calculated, audit trail entries can be removed without disturbing that order.

The second filter that does the opposite of the transitive closure filter is the *transitive reduction filter*.

### 8.2.8 Transitive Reduction Filter

Recall that our aggregation algorithm presented in Section 6.6 requires the partial order to be minimal. Therefore, we implemented a filter that calculates the transitive reduction on a partial order.

A requirement for the aggregation algorithm presented in Section 6.6 is that the partial order on the events in a case is minimal. Therefore, we need to calculate the transitive reduction, using the transitive reduction filter. Recall that the transitive reduction of a partial order is unique.

Figure 8.8 shows how filters should be applied to a partially ordered log, in order to remove any start event. In the top left corner of the figure, it shows that the first of three filters is the transitive closure filter. The second filter is the default filter with the settings as shown in Figure 8.5 and the third filter is the transitive reduction filter, that removes implied causal dependencies, as well as any references to audit trail entries that no longer exist. Since filters are applied to the log in the order in which they appear in the list, the transitive closure and reduction filters should always be applied in the same way, i.e. the closure filter as the first in the list and the reduction filter as the last in the list.

### 8.2.9 Conclusion

In this section, we presented several log filters, that allow the user of ProM to filter the information contained in a log. The purpose of this filtering is twofold.
Figure 8.8: Removing start events, using the transitive closure and reduction filters.
First, using filters, one can make sure that certain requirements of specific process mining algorithm are fulfilled, i.e. for example by introducing initial and final audit trail entries referring to new log events. The second application of filters however is to obtain better results from the process mining algorithms used.

Consider a process log with a large number of audit trail entries and process instances that reflect the behaviour of a certain process in detail. In such a case, the direct application of process mining algorithms to such a log may lead to unreadable, large process models. Therefore, a first step in process mining often is to project the log onto complete events and to discard all process instances that show “abnormal” behaviour, e.g. withdraw events or abort events.

When a process model is generated by applying a process mining plug-in to a log containing only complete events, the user can go back to the original log and investigate a part of the log in detail, i.e. for example by only considering a set of tasks that seem to have a certain relation in the mined process model, thus making process mining an interactive process.

Recall the log-based verification approach presented in Chapter 4, where we introduced an LTL-based approach to check if all process instances satisfy a certain property. In essence, this could be seen as a filter, i.e. if the process instance satisfies the property, it is not changed, whereas if it does not, all its audit trail entries are removed. However, in ProM this is not implemented as such.

Finally, it is important to realize that the order in which filters are applied to a log influences the result. It might for example be useful to first select a compulsory initial event and then project the log on a set of events not containing that initial event. The other way around however would always yield an empty log.

### 8.3 Mining Plug-ins

Once a process log is filtered, a mining plug-in can be started. Although in this thesis, the focus has been on discovering the control flow, there are many other kinds of mining plug-ins in ProM, for example plug-ins to discover social networks or organizational structures. Because these plug-ins are not within the scope of this thesis we will not discuss them here, but we discuss some of these plug-ins in the walktrough that we present in Section 8.7.

The input of a mining plug-in always consists of a process log in the MXML format, where it is assumed that all algorithm-specific assumptions hold. The plug-in that implements the aggregation algorithm presented in Section 6.6 for example, requires that the process instances it receives as input contain specific data attributes that specify the partial order on the audit trail entries in that log. If this information is not there, the plug-in simply does not work.
8.3.1 Log abstraction plug-in

In Section 6.3, we have shown how a log can be used for the calculation of causal and parallel dependencies. Since this is a common algorithm, shared by more than one plug-in, it is implemented in the log abstraction plug-in, which is an abstract implementation from which several others are derived, such as the $\alpha$-algorithm plug-in presented in 8.3.2, the partial order generator of Subsection 8.3.4 and the multi-phase macro plug-in of Subsection 8.3.6.

The user interface provided by the log abstraction plug-in allows the user to enforce causal dependencies based on the transactional model presented in Section 2.2 and to derive parallel relations based on time. Figure 8.9 shows a screenshot of this user interface, where the configuration for log abstraction is done in such a way that the causal dependencies implied by the transactional model are indeed used. Furthermore, if two activities overlap in time, considering the “start” event as the first event and the “complete” event as the last event, then parallelism is implied between all events relating to these two activities.
Figure 8.10: Result of the o-algorithm applied to a log with only complete events.

Figure 8.11: Result of the o-algorithm applied to a log with start and complete events.
8.3.2 \(\alpha\)-algorithm plug-in

The first plug-in that was derived from the log abstraction plug-in realizes the \(\alpha\)-algorithm presented in Section 6.4. After choosing the right settings in the dialog provided by the log abstraction plug-in, the plug-in then applies the \(\alpha\)-algorithm and the result is presented as a Petri net. Figure 8.10 shows the result of the \(\alpha\)-algorithm plug-in when applied to our example log, where we considered only the “complete” events. Figure 8.11 shows a similar result, but this time also including the “start” events. Note that this result is exactly the same as the Petri net presented in Figure 6.10, except for the layout. Since this layout is not determined by the process mining algorithm, ProM uses an external library called DOT (cf. [39]) for generating layout information for all graph-based models.

At this point, it is important to realize that the result of this plug-in is not only a Petri net, but that it also contains a reference to the filtered MXML log it was generated from. Therefore, this result can be used by other plug-ins that require the combination of a log and a Petri net as input, such as the conformance checker [152].

8.3.3 Region Miner

The iterative approach that uses the Theory of Regions for process mining presented in Section 6.2 is implemented in ProM in the region miner. As described in that section, the approach results in a Petri net that can reproduce the given log exactly, however each instance has to start with the same log event. Therefore, we applied the algorithm to our example log, taking into account only “complete” events and those cases that start with “Decide on MVV”. The result is shown in Figure 8.12.

8.3.4 Partial Order Generator

In Section 6.5, we presented an algorithm to convert process instances to partial orders. This is implemented in the partial order generator, which is based on the same log abstraction plug-in as the \(\alpha\)-algorithm plug-in, i.e. the plug-in also allows the user to enforce causal dependencies based on the transactional model and to derive parallel relations based on time.

The result of this plug-in is a new MXML log, that now contains the information about the partial orders for each process instance. ProM is able to recognize this information and therefore each instance is presented as an acyclic directed graph. Figure 8.13 shows this feature of ProM. The process instance that was represented as a linear order in Figure 8.7 is now represented as a partial order in Figure 8.13.
Figure 8.12: Mined Petri net using the Iterative Theory of Regions algorithm.
Section 8.3 Mining Plug-ins

8.3.5 Partial Order Aggregator

An MXML log containing process instances that represent partial orders on their events can serve as input for the aggregation algorithm.

The partial order aggregation algorithm presented in Section 6.6 is implemented in the partial order aggregator. The input is an MXML process log that contains the information about the partial order in the data part of each process instance and audit trail entry. If for some process instance, this information is not there, the process instance is simply skipped in the aggregation process.

The only setting of this plug-in is the option to let the aggregation algorithm first calculate the transitive reduction of each process instance before the aggregation. Note that the aggregation algorithm presented in Section 6.6 requires the partial orders to be minimal, which can be enforced by this option. For our example however, in the interest of readability, we decided to ignore the start events, by introducing the combination of log filters as shown in Figure 8.8.

![Diagram](image-url)
Figure 8.14: Aggregation graph and EPC translation of our example log.
The result of the *partial order aggregator* is an aggregation graph, as shown in Figure 8.14, i.e. a directed graph with labels on both nodes and edges. Note that this graph is similar to the one depicted in Figure 6.20, where only the labels of nodes and edges are different. In Subsection 8.5.1, we introduce the conversion plug-in that can convert this aggregation graph to an EPC as presented in Section 6.6.2. The result of this translation in ProM is shown in Figure 8.14 as well. Note that the layout of the models in ProM is generated by an external library called DOT [87].

**8.3.6 Multi-phase Macro Plug-in**

In Chapter 6, we presented a multi-phase algorithm to obtain a correct EPC describing our log. The first phase is the log abstraction, the second phase the generation of partial orders, the third phase the aggregation of these partial orders and the final phase the conversion to EPCs. Although these phases have been implemented in separate plug-ins, we implemented a *multi-phase macro plug-in* that performs exactly that sequence, i.e. it enables the user to choose the settings for the partial order generator and then it consecutively applies the *partial order generator*, the *partial order aggregator* and the conversion to an EPC. Therefore, the EPC in Figure 8.14 is the result of the *multi-phase macro plug-in*, after selecting the right settings in the dialog shown in Figure 8.9.

**8.3.7 Overview**

Besides the plug-ins presented in this section, the current version of ProM provides many more mining plug-ins, each with their own requirements and resulting formats. Since the number of plug-ins increases constantly, we refer to the website www.processmining.org for the latest version of ProM with a complete overview of its plug-ins. However, we would like to mention the following plug-ins explicitly:

- The genetic mining algorithms presented in [128] are implemented in two mining plug-ins, namely the *Genetic algorithm plug-in* and the *Duplicate Tasks GA plug-in*. Both plug-ins result in a Heuristics net, which can be converted to both an EPC and a Petri net,
- The first step of the mining approach proposed in [91, 92] is implemented in the *DWS mining plug-in*, the result of which is a collection of Heuristics Nets,
- The *Social network miner* presented in [24] implements a mining algorithm that results in a social network describing relations between the originators in a log.

Using a mining plug-in on a log to obtain a model of some sort is typically the first stage in the context of process mining. The resulting model may need some additional analysis or needs to be converted into another format. For this purpose,
ProM contains analysis and conversion plug-ins, as shown in the remainder of this Chapter.

### 8.4 Analysis Plug-ins

Process models, but also other mining results are typically static structures that compactly describe information that was generated from the log. These models however are not the final result. Instead, there may be interesting questions that can only be answered by analyzing the results of mining in a larger context. Analysis plug-ins serve this purpose, i.e. they take a number of models or other structures as input and then perform analysis techniques that relate to the question under investigation.

#### 8.4.1 EPC Verification Plug-in

In Chapter 5, we presented an EPC verification approach (cf. Section 5.3), which is implemented in ProM as an analysis plug-in. The input of this *EPC verification plug-in* is simply an EPC, and the analysis leads to one of three possible conclusions: (1) the EPC is correct, that (2) the EPC can be correct or that (3) the EPC contains errors.

To show the functionality of the *EPC verification plug-in* we use a real-life example taken from [80]. These examples resulted from a case-study within a large Dutch bank. We applied the verification plug-in of ProM to several processes within this bank, including the trade execution process. We imported the EPC describing that trade process into the ProM framework using another plug-in, namely the *ARIS graph format import*, which we will present in Subsection 8.6.6. Furthermore, we started the *EPC verification plug-in*, which resulted in the screen presented in Figure 8.15.

The plug-in informed us that there was only one initial event, i.e. “trade executed (deal made)”’. Therefore, as possible initial event sets, we selected the set containing only this initial event (this is not reflected in the screenshot). Behind the scenes, the plug-in now maps the EPC onto a marked Petri net, constructs the state space for that Petri net, and computes the possible final states for that Petri net. Finally, as Figure 8.16 shows, the plug-in divided the 11 possible outcomes into desired and undesired states.

At this point, we had to decide which final states were desired (i.e. final states to keep) and which were undesired (i.e. final states to ignore). Note that the plug-in already categorized the final states based on the fact whether the final states included non-final events: if a final state includes a non-final event, then the plug-in proposes to ignore that final state, since this indicates a process that terminated while some events have not been dealt with. By simply accepting the categorization as proposed by the plug-in, the result is that “The EPC can be
Section 8.4 Analysis Plug-ins

Figure 8.15: The trade process after starting the EPC verification plug-in.
Figure 8.16: The trade process with its undesired (a) and desired (b) final states and the verification result (c). The EPC can be corrected, but allows for undesired behaviour.
Figure 8.17: The trade process of Figure 8.16 with the new verification result (c) and highlighted part.
correct, but allows for undesired behaviour”.

After some discussion with the process owner at the bank, we were informed that two of the final states that the plug-in proposed to keep, were in fact undesired. Therefore, we moved these two states to the undesired list and ran the verification again, which now lead to the following message: “The EPC contains structural errors”.

The plug-in highlights the problematic parts it identified, as shown in Figure 8.17. From this information, we deduced that the EPC contained a problem with respect to synchronization and choice: Two parallel branches were started, but from one of these branches it was possible to jump back and to start both branches again. As a result, the other branch could be started over and over again, without ever completing.

The EPC verification plug-in is a plug-in that only requires one object as input, namely the EPC. Some plug-ins however might require more than one input, an example of which is the LTL checker.

8.4.2 LTL Checker

The LTL language for log-based verification is presented in Chapter 4. The implementation of this verification approach in ProM is called the LTL checker and it requires two objects as input. First, it requires an MXML log file and second, a set of LTL formulas in the format described in Section 4.2 and in [41]. These LTL formulas can be imported using an import plug-in for LTL files, which we present in Subsection 8.6.4.

When the LTL checker is used to verify a formula on a log, it stores the result of that formula in the data part of each process instance for which it was checked, i.e. the data attribute’s name is the LTL expression that was checked and the value is the result of that formula.

Although many formulas can be formulated, ProM contains a special version of the LTL checker that comes with a large set of default formulas and therefore only requires a process log as input.

8.4.3 Default LTL Checker

The default LTL checker contains a large set of pre-defined LTL formulas. Figure 8.18 shows the settings dialog of the LTL checker, which in this case was started with the default set of formulas. It shows that the formula “exists_person_doing_task_A_and_B” is selected and that the parameters “A” and “B” are given the values “Send permit rejection letter” and “Evaluate objection” respectively.

Figure 8.18 also shows that the user has to select how the LTL checker should check the results, i.e. whether it should check the whole log, or stop at the first failure or success. Furthermore, the user can choose to ignore those process
Section 8.4 Analysis Plug-ins

Figure 8.18: Settings dialog of the LTL checker.

Figure 8.19: Result dialog of the LTL checker.
instances for which the formula has been checked before and the result was stored in the data part of the process instance.

The result of checking the formula shown in Figure 8.18 on our example log is shown in Figure 8.19. In this figure, the formula shown in Figure 8.18 is checked and the log is divided into two, i.e. in those process instances for which the result equals true and those for which the result equals false. Note that in our process log, there were three cases that violated the formula. As shown in the figure, in process instance “1949”, “William” executed both the activity “Send permit rejection letter” and “Evaluate objection”, which we consider to be undesirable.

At this point, we like to stress that the Default LTL Checker is almost the same as the LTL Checker. However it does not require a set of LTL formulas to be imported into ProM, since it is equipped with a default set of LTL formulas. Since these formulas are stored in a file, the default set of formulas can easily be extended by the user of ProM.

8.4.4 Petri net Analysis

Besides the verification of EPCs or checking LTL formulas on logs, ProM contains many other analysis plug-ins. One of which is a plug-in tailored towards the analysis of Petri nets, i.e. the Petri net analysis plug-in. This plug-in takes a Petri net as input and, on request, calculates the transition and place invariants, the coverability graph and the reachability graph (cf. [67,138,169]).

Figure 8.20 shows the Petri net analysis plug-in showing the state space of the Petri net of Figure 8.10. This state space can provide a user insight into the behaviour of the Petri net, but it can also be used to obtain information about regions.

8.4.5 Region Calculator

In Section 6.2, we presented the Theory of Regions. Recall that a region is a set of states in a state space with certain properties regarding the incoming and outgoing transitions. Using the region calculator, ProM can calculate the regions of a state space. Note that there are two options, i.e. it can calculate only the minimal regions, or all regions. Figure 8.21 shows the state space of Figure 8.20, with a list of all its regions. Furthermore, one region is selected and highlighted in the graph.

Once a set of regions is calculated from a state space, we can use the synthesis algorithm presented in Section 6.2 (cf. Definition 6.2.6) to generate a Petri net again. This process is implemented as a conversion plug-in, hence we discuss this plug-in in Subsection 8.5.4.
Figure 8.20: State space of the Petri net of Figure 8.10.
Figure 8.21: State space of Figure 8.20 with regions.
8.4.6 Conclusion

In this section, we presented several analysis plug-ins, all with a different purpose. The overview given in this section is far from exhaustive, i.e. there are many other analysis plug-ins, such as:

- The conformance checker, which calculates to what extent a process log and a Petri net match. For related work on this subject, we refer to Section 3.3,
- Several plug-ins for the analysis of heuristic nets, since these nets have very specific properties. All of these plug-ins are discussed in detail in [128],
- Footprint similarity plug-in, a plug-in that calculates a metric saying how similar two models are, with respect to their causal dependencies and moment of choice,
- The Social network analyzer, which allows the user to analyze social networks [24,171],
- Performance analysis plug-in, which uses the timestamps stored in logs and a Petri net to derive performance information about a process. This performance data is visually represented in the Petri net.

ProM currently provides 37 of such analysis plug-ins. However, the plug-ins presented above have not been developed in the context of this thesis and therefore, we refer to www.processmining.org for more information on them and all other analysis plug-ins.

Although many analysis techniques have been developed for specific purposes, such as for the verification of EPCs, the real power of ProM lies in the fact that most formalisms can be converted back and forward, which allows the user to use many more analysis techniques. For converting mining results from one formalism into other formalisms, ProM contains conversion plug-ins.

8.5 Conversion Plug-ins

Conversion plug-ins in ProM are meant for the translation of one formalism into another. For example in Section 6.6, we presented a mining algorithm that produces an EPC as result and we mentioned that this EPC can be translated into a Petri net. The latter is an example of a conversion plug-in in ProM.

One of the important lessons we learned while using ProM is that it is fairly easy to convert one model into another model if one is willing to accept some loss of information or precision. For example, there exist many interpretations of the semantics of EPCs (cf. the “Vicious Circle” discussion in [118]). Nevertheless, rough translations from EPCs to YAWL and Petri nets can be very useful because they are applicable in most practical cases. Moreover, some operations on EPCs such as reduction and verification can be applied without selecting one particular semantical interpretation, as explained in Chapter 5.
Therefore, we advocate a pragmatic approach towards conversion and for the many formalisms described in the introduction of this chapter, various of such pragmatic conversion plug-ins are available. However, we focus on those that were implemented in the context of this thesis and for the latest plug-ins in ProM, we again refer to www.processmining.org.

8.5.1 Aggregation Graph to EPC

In Subsection 8.3.6, we presented a macro plug-in implementing the algorithms of Section 6.5 and Section 6.6. Recall that this macro plug-in converted an aggregation graph to an EPC. This conversion is implemented separately in a conversion plug-in. Figure 8.22 again shows an aggregation graph on the left hand side and the resulting EPC on the right hand side, after using the \textit{aggregation graph to EPC plug-in}.

The next plug-in we introduce is the conversion from EPCs to Petri nets. Note that in Figure 8.22 we highlighted two connectors, which we will refer to in the Petri net translation of this EPC.

8.5.2 EPC to Petri net

In Section 6.6, we stated that the EPC resulting from the aggregation approach can be converted into a Petri net, using the standard translation algorithm presented in [61]. This translation is implemented in the context of ProM as a the \textit{EPC to Petri net conversion plug-in} and the result of the conversion of the EPC shown in Figure 8.22 is presented in Figure 8.23.

Note that the Petri net of Figure 8.23 shows a labelled Petri net, where the transitions relating to functions are labelled with the function labels and the transitions relating to connectors are labelled with $\tau$ and shown in black. Furthermore, in the background, the conversion plug-in used a special set of reduction rules to remove as many of those $\tau$-labelled transitions (or invisible transitions, as they do not correspond to events visible in the log) as possible, without changing the behaviour.

However, the resulting Petri net still shows a lot of $\tau$-labelled transitions. Especially the two highlighted parts contain many $\tau$-labelled transitions. These transitions correspond to the OR-split and the OR-join of Figure 8.22, i.e since the OR-split has 3 outgoing edges, $2^3 - 1 = 7$ transitions are required and for the OR-join with 4 incoming edges, $2^4 - 1 = 15$ transitions are required. However, as shown in Section 6.6.3, unused transitions can be removed from the Petri net in a process called restriction. The restriction presented there and in Section 6 of [73] has been implemented in ProM as part of the conformance checker, which we show in Subsection 8.7.2.
Section 8.5 Conversion Plug-ins

Figure 8.22: Aggregation graph and EPC translation.
Figure 8.23: Petri net translation of the EPC in Figure 8.22. Note the "explosion" of possible transitions relating to the OR-split and OR-join connectors.
Section 8.5 Conversion Plug-ins

Figure 8.24: Reduction of the EPC in Figure 8.22.

8.5.3 EPC Reduction Plug-in

The EPC verification approach, presented in Section 5.3, uses a translation from EPCs to Petri nets that is slightly different than the one presented in Subsection 8.5.2, since it applies the translation to a reduced EPC (recall that this is an EPC without functions etc.), and it results in a safe Petri net. The reduction process, i.e. reducing an EPC using the reduction rules of Section 5.3.1 is available as a the EPC reduction plug-in.

The EPC reduction plug-in applies all reduction rules presented in Section 5.3.1 to the EPC. When applied to the EPC of Figure 8.22, the result is a small reduced EPC with two events and 4 connectors, as shown in Figure 8.24.

8.5.4 Regions to Petri net

The last plug-in we present in this section, is the plug-in that transforms a set of regions into a Petri net, using the Theory of Regions described in Section 6.2, i.e. the Regions to Petri net plug-in. When applied to the state space of Figure 8.20 with the set of minimal regions, the result is again Figure 8.10. However, when applied to the same state space and the set of all its regions, the result is the very dense Petri net shown in Figure 8.25. Note that (although impossible to see directly) this Petri net has the same state space as shown in Figure 8.20.
In this section, we presented four conversion plug-ins for the translation of one formalism to the other. Many more plug-ins are available, including translations from Petri nets to EPCs. It is exactly this capability that makes ProM a very useful tool for process mining. A mining algorithm that provides its result in one formalism can also be used in the context of other formalisms, as long as there exists a conversion plug-in supporting the translation, which might not always be trivial.

Since the analysis and conversion plug-ins are not limited to the results of a process mining algorithm, but can be used in a larger context ProM allows for reading and writing files in various formats.
8.6 Import and Export Plug-ins

So far, we presented implementations of algorithms in the ProM framework. However, many of the implemented algorithms are also useful outside of the process discovery context, i.e. we would like ProM to be able to import models from other tools. Furthermore, the results of process discovery, or conversion should be exported for further use in other (sometimes commercial) tools. For this purpose, ProM contains import and export plug-ins.

In addition to the plug-ins we discuss individually in this section, ProM contains the following plug-ins to import and export the different formalisms to the following formats:

- AGNA, a tool for the analysis of social networks [27],
- BPEL 1.1, a standard defined for storing BPEL 1.1 models [36],
- CPN Tools 1.4.0, the native format of CPN-tools 1.4.0 for storing Petri nets [55],
- CPN Tools 2.0, the native format of CPN-tools 2.0 for storing Petri nets [55],
- DOT file, for storing any graph-based model such that it can be used by DOT [39],
- FSM, the native format of state-space visualization tools FSMview [95] and DiaGraphica [181],
- HN file, a ProM native format for storing Heuristic Nets,
- NetMiner, the native format of the social network analysis tool NetMiner [139],
- Org Model, a ProM native format for storing organizational models,
- Protos XML Export, which writes a Petri net to a Protos XML export file. For more information about Protos, we refer to [140],
- CSV file, a comma-separated values file containing case-related data that can be read by for example Microsoft Excel,
- Petrify files, ProM can export transition systems to and import Petri nets from the Theory of Regions tool Petrify [53]. Note that this plug-in works on the Transition System object which is the result of the mining plug-in called Transition System Generator, presented in [23,155], and
- YAWL files, containing YAWL models which can be executed in YAWL [187].

Import plug-ins basically read a file to generate some sort of internal model. For example EPCs which are exported by the Aris Toolset can directly be read into ProM, so the EPC verification plug-in can be used to check if the modelled EPC contains errors. Export plug-ins write these internal models back to file for further use in the respective tools.
8.6.1 MXML Format

The main format in which ProM can read and write is the MXML format presented in Section 2.2.3. ProM can read process logs in this format using the MXML import plug-in and is also capable of exporting filtered logs to the same format, i.e. through the MXML export plug-in. Note that we used this plug-in before, when we exported the generated partial orders to an MXML log file in Subsection 8.3.4.

Although the export plug-ins contained in ProM all write a file in a pre-defined format, ProM contains two types of import plug-ins. The first type are import plug-ins that just read a file and produce some internal object, such as the log filter import plug-in presented in Subsection 8.6.3 and the LTL import plug-in presented in Subsection 8.6.4. The second type however are those plug-ins that import some model that can be connected to an MXML log.

8.6.2 Log Connection

Connecting a model to a log file is a process that works as follows. When opening a file, the user needs to select a process log file to connect to, as shown in Figure 8.26. Then ProM reads the file using the right import plug-in and constructs the internal representation. This representation contains a number of objects that can be connected to events in the log, i.e. a Petri net contains transitions and an EPC contains functions that can connect to events in the log.

ProM collects the objects that can be connected to log events and it collects all log events from the selected log. Then a dialog is shown, where the user has to select a mapping between the objects in the model on the one hand and the events in the log on the other hand. The mapping from the objects in the model to the events in the log needs to be made manually. However, based on the labels of the objects and events, ProM makes a suggestion for the correct mapping automatically, which is shown in the dialog presented in Figure 8.27.

When deciding on the correct mapping, the user has the option not to map an object to anything (i.e. to make it invisible) or to map it to a non-existing event, i.e. an event that is not in the log. Finally, when the user accepts the mapping, ProM checks whether all events in the log are linked to at least one object. If this is not the case, a dialog is presented as shown in Figure 8.28. Here, the user can choose to automatically filter out those events, or to correct the mapping.

Mapping objects in a model to events in a log can be useful, for example in the context of performance analysis, in which case a log is replayed in a Petri net to calculate performance characteristics.

In the remainder of this section, we present the import plug-ins developed in the context of this thesis and if a plug-in can connect to a log, we explicitly mention how.
Figure 8.26: Opening a TPN file to be connected to “Filtered Log Without Partial Order.MXML”.

Figure 8.27: When connecting our example log to a TPN file containing the Petri net of Figure 8.10, ProM suggests a mapping from the transitions in the Petri net to the events in the log based on the labels of these transitions and events.
8.6.3 Log Filter

Each log filter in ProM is required to be able to read and write its settings from and to XML. When writing a filter to file, using the export plug-in, ProM takes care that the produced XML file can again be read by ProM, using the log filter import plug-in.

The import plug-in for log filters is different from most plug-ins, since the result is not a dialog of any kind. Instead, the plug-in produces a global object that from that point on is available in the filter selection part of ProM, as shown in Figure 8.29, where we show the availability of the filter stored in “D:\filter.xml”.

Furthermore, when a list of log filters is exported as one filter, the import plug-in produces a so-called abstract filter that is a wrapper around the original list of filters. Therefore, if the log filter we just imported is selected the settings dialog shown in Figure 8.30 is presented, showing the original list of filters, that we used before in Figure 8.8.

8.6.4 LTL Template Format

Another plug-in that does not produce a model that can connect to a log is the LTL template import plug-in. This plug-in reads text files containing LTL formulas as presented in Section 4.2. The result is a dialog showing the text file, as presented in Figure 8.31. As long as this dialog is not closed, the LTL formulas are available for use in the LTL checker. Note that for LTL template files, no export plug-in is available. The reason for this is that ProM does not contain functionality to create or change LTL formulas, hence an export would be less useful.

Figure 8.32 shows the analysis dialog that can be accessed through the analysis menu. As shown in this figure, the LTL checker requires two inputs, the first of which is a process log and the second of which is the LTL template file we just imported.
Figure 8.29: The imported filter is available in the filter selection dialog.

Figure 8.30: The settings dialog of an abstract log filter.
A well-known tool for modelling EPCs is the Aris toolset [161]. This toolset contains export and import functionality for EPCs in the form of AML files. These AML files contain much more information than just EPCs. Since ProM does not provide support for all AML objects, the import plug-in ignores those objects it does not support, such as data objects, system objects, etc. What ProM does support however is the hierarchy of EPCs that the AML files contain. Therefore, if an AML file is opened, the result is a hierarchy of EPCs, presented as shown in Figure 8.33, i.e. as a tree structure on the left hand side of the dialog.

8.6.6 Aris Graph Format

The Aris graph format is a format for storing EPCs, which is used by the performance monitoring tool Aris PPM [111]. It is a simple XML-based format in which a list of EPCs can be stored (cf. Section 7.2). When importing, ProM considers this list to be a flat hierarchy, i.e. it shows all EPCs in a dialog, similar
Figure 8.32: The LTL template file selected as input for the LTL checker.

to Figure 8.33.

When importing EPCs, ProM checks whether all EPCs adhere to the structural rules of EPCs that we presented in Section 2.4. If not, a warning dialog is shown, pointing the user to the problematic areas. An example of such a dialog is shown in Figure 8.34.

8.6.7 VDX Format

Similar to AML, the VDX format is an XML-based format for storing EPCs that are modelled using Microsoft Visio 2003. Again, many objects are ignored in the import process, but the hierarchy (if present) is respected in the resulting EPC dialog. Currently, no export to the VDX format is available in ProM, however, we expect it to be developed in the future.

Note that all EPC import plug-ins return a collection of EPCs, which can be connected to a process log. In other words, when importing EPCs in any of the presented formats, the user can map the functions in those EPCs to events in a selected process log.
Figure 8.33: The hierarchy of EPCs collected from the Dutch bank mentioned in Subsection 8.4.1.
Figure 8.34: Warning dialog, showing that one of the imported EPCs does not meet the criteria of Definition 2.4.1.

### 8.6.8 EPML Format

Whereas the previously mentioned file formats for EPCs were proprietary formats of commercial tools, the EPML format [136] for storing EPCs is a format used by the open source framework EPCtools [117]. Similar to VDX and AML, this format allows for storing hierarchies of EPCs, as well as many objects that are not supported by the ProM framework and therefore ignored in the import.

### 8.6.9 PNML Format

Similar to EPML, PNML is a standard format for storing Petri nets [43]. Again, the format allows for storing many objects that are not supported by ProM, such as inhibitor arcs and reset arcs. Since PNML is an evolving standard, ProM has been designed to be able to read and write PNML in the same format as the process modelling tool Yasper [103]. Figure 8.35 shows the Petri net of Figure 8.10 in Yasper.

### 8.6.10 TPN Format

The last plug-in we discuss in this section is an import plug-in capable of reading Petri nets in the TPN format. The TPN format was presented in [2] as the native format of the IAT analysis tool. Furthermore, it is used by the WF net verification tool Wo\nlan. Wo\nlan is a tool implementing the verification approach for WF nets, presented in Section 5.2, although currently most of the functionality of Wo\nlan is also contained in the ProM framework directly.

Plug-ins that read Petri nets will be able to connect to a given log, in which case the transitions of the Petri net can be mapped onto the events in the log.
Figure 8.35: The Petri net of Figure 8.10 in Yasper.
8.6.11 Conclusion

The ProM framework can import models in many different formats, of which we discussed a small set here. The mere fact that ProM can do so enables it to compare models designed in different tools, by different users. Furthermore, by also allowing the user to export models in different languages, ProM bridges the gap between several commercial and public tools, thus allowing a process designer to build a model in one tool and analyze it in another tool, while translating the results of this analysis back to the original modelling tool.

Currently, the ProM framework contains 38 mining plug-ins, 43 analysis plug-ins, 18 import plug-ins, 30 export plug-ins, 32 conversion plug-ins and 20 log filters, bringing the total number of plug-ins to 181.

8.7 Example Walkthrough

In Figure 8.2, we presented a graphical overview of the ProM framework. Furthermore, in sections 8.2 through 8.6, we presented several plug-ins implemented in the context of this thesis. We conclude this chapter with an example of how to use ProM, i.e. we take an example and walk through ProM, showing how it can be used on that example, to illustrate the powerful features of ProM (cf. [180]).

Figure 8.36 shows an EPC (Event-driven Process Chain) describing a review process [180] which we used before in Section 7.3.5. In principle each paper

![Figure 8.36: The example review process model [180].]
should be reviewed by three people. However, reviewers may be tardy resulting in time-outs. After a while the reviews are collected and based on the result a paper is rejected, a paper is accepted, or an additional reviewer is invited.

The EPC shown in Figure 8.36 could have been imported into ProM from ARIS, using the import plug-in of Subsection 8.6.5, ARIS PPM using the import plug-in of Subsection 8.6.6, or EPC Tools using the import plug-in of Subsection 8.6.8. Instead of imported, the EPC could have been discovered using some process discovery plug-in or be the result of some conversion.

Once a model such as the EPC shown in Figure 8.36 is in the ProM framework, it can be used as a starting point for analysis and model conversion. In this section, we focus on the example of the review process and show how related mining, analysis, conversion, import and export plug-ins can be used.

8.7.1 Mining

Mining plug-ins like the α-algorithm [26] and social network analyzer [22] extract models from event logs and most mining plug-ins discover process models represented in terms of Petri nets, EPCs, etc. However, some mining plug-ins also address other perspectives such as the data or organizational perspective.

Starting point for our review example is a log containing events related to the reviewing of papers. Based on such events we can automatically construct a social network as shown in Figure 8.37. Using the same log, we can also construct a process model as shown in Figure 8.38. This model has been created using the α-algorithm presented in Section 6.4.

8.7.2 Analysis

After obtaining a process model using process mining or by simply loading the model from another tool, we can analyse it using one of the available analysis plug-ins for this model type. Because the process model is a Petri net, we can only start a Petri net analysis plug-in. The framework is capable of determining at runtime which plug-ins can handle the current model, and it will only offer plug-ins that can handle the current model to the user. In addition to classical analysis tools such as a verification tool, ProM also offers a conformance checker and an LTL checker as described below.

Conformance Checker

As an example, and to show how versatile ProM is, we can analyze to what extent another log fits the mined review process model. For this reason, we open a different log than the one used to derive the process model. This way, we can analyze how a model derived from one log conforms to a different log (for example a log of the same process taken earlier). We start the conformance
Figure 8.37: A social network derived by ProM (smaller windows show the export into NetMiner).
Figure 8.38: A snippet of the results of the conformance checker when comparing the model of Figure 8.38 with a log containing "noise".

Figure 8.39: The Petri net resulting from applying the -algorithm on some XML log.
checker [153] plug-in with the combination of the process model and the log as input (note that ProM automatically offers this combination to the conformance plug-in in the analysis menu). Figure 8.39 shows a snippet of the results. From these results, we learn that (amongst others):

- The log does not fit the model entirely, as the fitness $\approx 0.89$ (if the log would fit the model, the fitness would be 1).
- In 65 out of 100 cases, the process ended just before the “decide” task.
- In 35 cases, the “decide” task was executed, but in 6 of those cases, there was no token available in the place before that transition (hence the $-6$ in that place). In those remaining 6 cases, an execution of the “decide” task had to be inserted to allow logged successors (like “accept” and “reject”) to execute.

**LTL Checker**

When using the LTL checker introduced in Subsection 8.4.2, we checked whether in all cases the 4-eyes principle was satisfied. Figure 8.40 shows that this is not the case for the tasks “get review 2” and “get review 3”: “John” has done both reviews.

### 8.7.3 Conversion

After we have analyzed the process model (a Petri net), we can convert it into other process models. For example, we can convert it into an EPC or a YAWL model. However, before doing so, we declare the four “time-out” transitions in Figure 8.38 to be invisible. Figure 8.41 shows the result. The four “time-out”
transitions did not correspond to any real activities in the process, i.e., they were only there for routing purposes (to bypass the “get review” tasks). When converting one model to another we can use such information.

**From a Petri net to an EPC**

First, we convert the Petri net shown in Figure 8.41 into an EPC using the algorithm presented in [61]. The resulting EPC has the same structure as the one in Figure 8.36. Of course, after converting the Petri net to an EPC, different plug-ins may be applied to the process model. For example, we could check the correctness of the resulting EPC using the approach presented in Chapter 5, in which case ProM will report that this EPC is trivially correct.

**From a Petri net to a YAWL Model**

Figure 8.42 shows the result from converting the Petri net into a YAWL model. Note that, in this case, the conversion plug-in is able to remove all routing transitions (i.e., the invisible transitions in Figure 8.41) from the resulting process model. Removing the invisible transitions introduces an OR-join and an OR-split, moreover conditions (corresponding to Petri net places) are only introduced when needed. Clearly, such a “smart” translation is far from trivial, since the splits and joins have to be derived from blocks of several places and transitions in the Petri net. Similarly, there are innovative conversions from EPCs to YAWL and conversions from heuristics nets (used for genetic mining) to Petri nets.

**8.7.4 Export**

Of course, we can also export any model to file. For example, we can export the converted YAWL model to a YAWL file, which can be uploaded right-away to a YAWL engine. Figure 8.43 shows the result after we’ve uploaded the file: a YAWL model with ID “WFNet28922354” has been uploaded. Note that most fields (specification ID, specification name, documentation, . . .) are generated by ProM. Figure 8.43 also shows a work list for the uploaded process. Currently, three work items are available in the work list: One for the task “invite reviewers”, one for “decide”, and one for “collect reviews”, all for different cases (3, 4 and 5).

The social network shown in Figure 8.37 for example can be exported both to NetMiner and to AGNA, both of which are specialized tools for the analysis of social networks.

**8.7.5 Conclusions**

Figure 8.44 provides an overview of the different ways we used ProM on the review example. The numbers on the edges refer to the order in which we applied all algorithms.
Figure 8.41: The Petri net with the “time-out” transition made invisible.

Figure 8.42: The mined review process model converted to a YAWL model.
Figure 8.43: The YAWL model uploaded to a YAWL server, and a worksheet thereof.
It is important to note that in the process described Figure 8.44 we only partially used the broad functionality of ProM. However, Figure 8.44 nicely demonstrates how versatile the ProM framework is, and how it can link different external tools together.
Chapter 9

Conclusion

The work presented in this thesis is motivated by the omnipresence of event logs and the relevance of good models. An increasing number of processes and systems is being monitored. For example, any enterprise information system is recording business events. Moreover, professional and consumer systems are recording events. For example, high-end X-ray and copier systems log all relevant events and distribute these over the Internet. Similarly, process models are readily available and their importance is increasing. Increasingly, information systems are configured on the basis of process models (cf. Workflow management systems, and all kinds of other systems containing a Workflow engine).

Moreover, process models are used for all kinds of analysis (e.g. simulation). The availability of event logs and the need for high-quality process models has been the starting point of this thesis.

In the remainder, we summarize our findings. First, we discuss our contributions related to verification. Then, we summarize our results in the field of process discovery. Finally, we highlight some limitations and discuss future work.

9.1 Verification Contributions

In the introduction, we introduced two approaches towards verification, i.e. log-based verification and process model verification. The first of the two takes recorded events of some information system as input and uses them to verify desired or undesired properties. The latter takes a process model and reports on errors contained in that model.

9.1.1 Log-based Verification

The goal of log-based verification is to use the events recorded in a process log to verify properties of a process. Our approach, presented in Chapter 4, is based on linear temporal logic, i.e. a logic containing specific operators towards reasoning
about time in a linear sense. Examples of these operators are “eventually” and “always”. Where typical support systems focus on the diagnosis of a process being enacted, e.g. by measuring performance on a daily basis, or by alerting management about specific undesired situations, our approach allows for a broader use.

For example, a well known property is the 4-eyes principle, i.e. although authorized to execute two activities $A$ and $B$, one person should not execute both activities for the same case. In our language this would translate to a formula stating whether for one case, there is one person that eventually executes activity $A$ and eventually executes activity $B$. Although many support systems would be able to monitor this, our approach allows for the parametrization of the statement, hence the LTL-formula specifying the 4-eyes principle can be re-used for different activities.

Since LTL formulas are checked on individual cases, our LTL-based approach can be used for example in process discovery, by splitting an event log into two partitions, one that does adhere to a certain desired property, and one that does not. Gaining insights into the cases that fall into any of these two categories can be extremely useful when diagnosing a process and this way, our LTL-based approach can be used to see in which cases company policy was violated. By examining the cases that indeed violated policy during the diagnosis phase of the
BPM lifecycle, one could optimize the running system.

Furthermore, in the process design phase, our approach can be used by a process designer to objectively verify user-statements about a process. Where a process designer so-far was forced to collect statements of as many users as possible to find the “common consensus”, it now suffices to only look into the incorrect statements in more detail, thus giving more focus in the questions asked to the people involved in the process under consideration.

Consider for example the situation where, according to the surgeon in charge, patients in a hospital always follow a certain path throughout their treatment. It is not unlikely to realize that there might be exceptions. When the surgeon states that each patient first gets treatment $A$ and then treatment $B$, this can now be checked on the fly, for example to find out that this does not hold if the patient is older than 55, in which case the two treatments are swapped. Hence, a process designer can use our approach to objectively verify user-statements about the process that is being (re)designed. Note that if a process model is already present, such decisions can also be detected using decision mining [154].

Using objectively verified user statements, a process designer often builds process models in a conceptual language. These models are typically used as input for the process configuration phase, for which it is of the utmost importance that the conceptual models are correct. Therefore, the second contribution of this thesis in the context of verification is an approach towards verification of a specific conceptual modelling language, i.e. Event-driven Process Chains or EPCs.

9.1.2 Event-driven Process Chain Verification

The EPC modelling language is used by some leading industrial modelling tools, e.g. Aris PPM, SAP, etc. By their success in business, EPCs have proven to be an intuitive modelling language for humans. However, because of the lack of clear execution semantics, the question whether or not an EPC is correct is not as trivial as it might seem.

In Chapter 5, we use the well-known concept of relaxed soundness for workflow nets (WF-nets) to present an approach towards verification of EPCs. The two main differences with existing verification approaches are the involvement of the user in the verification process and the possible outcomes.

User interaction

In our approach, we assume that the user of the verification tool has knowledge about the process modelled by the EPC that is not made explicit in that EPC. Therefore, in our verification approach, we interact with the user at two moments. First, the user is required to state the possible initializations of the process, i.e. which events can occur together that will trigger this model to start. Second, the
user is required to separate the possible outcomes of the process in desired and undesired outcomes.

Results

In contrast to existing verification approached, the result of our EPC verification approach is not binary (i.e. “correct” or “incorrect”). An EPC can of course be trivially correct, or definitely incorrect, however, there is a possible third outcome. If the EPC is said to be possibly correct, then the EPC can be used for the configuration of an information system since, under the assumptions made about the possible initializations and desired outcomes, the EPC has relaxed sound semantics. However, special care has to be taken in certain parts of the model when translating the EPC into an executable specification, as certain execution paths of the EPC should be avoided.

9.1.3 ProM plug-ins

The verification approaches presented in this thesis are tailored towards aiding a process designer in diagnosing and designing a process, which is supported by the fact that all the verification algorithms presented in this thesis have been implemented in the context of the ProM framework, as presented in Chapter 8.

The process logs that served as an input for the log-based verification, can also be used to automatically discover a process model, helping the process designer to diagnose or design a process even better.

9.2 Process Mining Contributions

In Figure 9.2, we show the contributions of this thesis in the area of process mining, with references to the sections in this thesis where we presented the respective algorithms.

In Chapter 6, we have shown that the same process log that was used to verify statements using log-based verification can be used to gain meaningful insights in the process under consideration, i.e. by deriving a process model from a log. We introduce several different algorithms that generate process models from event logs automatically. These models can then, for example, be used by a process owner to run simulations, or to gain insights in the way cases flow through an organization. Furthermore, these models can serve as a basis for process modeling, in which case a process designer can refine or extend the models manually, to more accurately describe the processes of an organization.

The algorithms presented in this thesis automatically generate process models in terms of EPCs or WF-nets, thereby focussing on process discovery from a process perspective. However, the applicability of the presented algorithms is not limited to the process perspective. Consider for example a large event log, where
in two cases, the 4-eyes principle was violated. In that situation, one could first abstract from the whole log using the abstraction algorithms of Section 6.3. Then, in the next step, the algorithms presented in Section 6.5 can be used to generate partial orders for each of the two cases. By now analyzing these two cases, one might be able to discover important information about the reasons for violating the 4-eyes principle. Obviously, this example shows that process discovery can also focus on process discovery from a case perspective.

Although all algorithms in this thesis look at process discovery from a process perspective, there are several major differences, with respect to the requirements on the event log and the quality of the resulting model. The Theory of Regions approach of Section 6.2 for example, directly generates a Petri net that exactly matches the log under consideration, under the assumption that the log is globally complete, i.e. it shows all possible behaviour. The other approaches construct less specific, but more generic process models, i.e. the resulting model can reproduce the log, but allows for more executions. Furthermore, those algorithm have less requirements on the information in the log, i.e. the log only needs to be locally complete, which means that all direct successions need to be recorded at least once.

Under the assumption that a log is locally complete, we have shown that we

![Figure 9.2: Overview of mining part of this thesis.](image-url)
can abstract from that log by deriving abstract ordering relations between events in the log. These abstractions can then be used by the $\alpha$ algorithm presented in Section 6.4 to generate a WF-net. Furthermore, the approach presented in Section 6.5 uses the abstraction of the log to transform each case into a partial order on its events. And using the algorithm presented in Section 6.6 these partial orders can be aggregated into a process model, in terms of an EPC or Petri net.

9.2.1 Log Abstraction

A process log contains linearly ordered events, i.e. for each case it shows which events occurred in which order. In Section 6.3, we presented an approach to use these logs and derive relations between these events in terms of causality and parallelism. Causal dependencies describe which events always precede or succeed which other events and parallel relations describes which events can occur in any given order. To derive these relations, we make extensive use of the information in the process log, i.e. the linear ordering of events and the concept of activities and event types.

We formalized the contents of process logs in Section 2.2, where we introduced a meta model describing the information in the log, specifically regarding the different events that can occur with respect to one activity, i.e. an activity can be started, completed, etc. This information is used when abstracting from a log, e.g. if one activity $A$ is started after another activity $B$, but $A$ completed before $B$ completes, then it is reasonable to assume that all events relating to activities $A$ and $B$ are in parallel, i.e. there are no causal dependencies between them.

The abstraction of the log in terms of causal dependencies and parallel relations serves as an input for a well-known process mining algorithm called the $\alpha$-algorithm. The $\alpha$-algorithm presented in Section 6.4 transforms an abstraction of the log into a Petri net. However, the result of this algorithm is not necessarily a correct process model. Therefore, we introduced an approach in two stages that always produces a relaxed-sound process model.

9.2.2 Partial Order Generation

The first step of our two-stage approach is to translate each case into a partial order on its events, as presented in Section 6.5. This is simply done by assuming that each event has exactly one event that triggered it and that each event triggered exactly one other event. In other words, we relate each event to its closest causal predecessor and its closest causal successor.

Furthermore, we provided translations of these partial orders to Petri nets and EPCs, explicitly showing which parts have been executed in parallel. This provides a case-by-case insight into the behaviour of the process, much like the commercial performance monitoring tool Aris PPM, that uses a similar idea towards visualizing individual cases using EPCs. Aris PPM however can not gen-
erate these representations based on simple event logs. Instead, it needs to be manually configured to link events from the running system to parts of the EPCs representing cases.

9.2.3 Partial Order Aggregation

Although describing each case as a partial order on its events can be useful for getting insights into the behaviour of individual cases, the goal of process discovery remains the construction of a process model describing the process from which all cases originated. Therefore, in Section 6.6, we introduced an algorithm that, when each case is represented as a partial order, produces a process model for an entire log. This process of going from partial orders to one model is called aggregation. Again, an algorithm is presented to translate the resulting model into an EPC, which in its turn can be translated into a Petri net.

In Section 6.6, we also prove that if we start with a process log, use the log abstractions of Section 6.3, and aggregate the partial orders generated by the algorithm of Section 6.5, the result is a correct EPC. In other words, the verification approach of Chapter 5 will consider the generated EPCs to be correct.

9.2.4 Other Partial Orders

The idea of using partial orders to describe executions is not new. In fact, the commercial performance modelling tool Aris PPM uses a similar concept of instance EPCs. Furthermore, Message Sequence Charts, or MSCs, are a well-known partial order based language to specify communication between agents. For Petri nets, similar concepts exist known as occurrence nets or runs.

Therefore, in sections 7.2 and 7.3, we presented approaches towards the aggregation of instance EPCs and causal runs respectively. In these sections, we have shown that the explicit information about causal dependencies and parallel relations stored in partial orders such as instance EPCs, MSCs and Petri net runs can be used to derive process models, i.e. to do process discovery.

9.2.5 ProM plug-ins

Most of the process mining algorithms presented in this thesis have also been implemented in the context of the ProM framework. Moreover, the log abstraction algorithm serves as a basis for many other plug-ins in the ProM framework, where they have proven to be a valuable addition to the initial relations presented in the context of the $\alpha$-algorithm in [26].
9.3 Limitations and Future Work

In this thesis, we presented several approaches and algorithms that help a process designer during the consecutive phases of the BPM life cycle, i.e. we presented LTL verification and process discovery for the diagnosis and design phase and EPC verification for use during the design phase. In this final section, we present some of the limitations of these approaches as well as opportunities for future research.

9.3.1 Process Mining

Since our process mining approach using partial orders is guaranteed to produce a correct EPC, it is a powerful approach. However, as many other algorithms, it has been proven to work very well in the context of administrative processes (such as insurance claim handling, call-center routing, etc), but the approach may perform poorly in environments with less structured processes, such as in hospitals. Although the result may be a correct EPC, it is often a spaghetti-like process model, not giving much insights in the process to a process designer.

The reason for the poor performance in more dynamic environments is twofold. First of all, for event logs taken from information systems used in these environments, it is not clear which properties are satisfied. It is impossible to say whether such a log shows all possible interleavings of two activities, since that implies knowledge about the process, which you do not have (the more dynamic a process is, the less knowledge you typically possess). The second problem is that event logs typically log information on a rather low level, whereas people talk about processes on a very high level, and there is no given way to link the low-level activities to those high level activities.

Furthermore, our approach produces EPCs and Petri nets, which are rather simple modelling languages, i.e. although they can express more complex routing patterns, the modelling of more refined routing constructs is far from trivial. Therefore, we feel that we should express process models on a higher level, using a more suitable language. An example of such a language would be the approach presented in [21,142]. The system presented there allows for the execution of process models, specified in terms of LTL formulas, much like the ones we presented in Chapter 4, but based on a graphical language. However, further research is needed to extract such LTL formulas from event logs, i.e. to do process discovery in such a way that the result is a set of LTL formulas.

9.3.2 Verification

The EPC verification approach presented in this thesis is different from existing approaches in the sense that it assumes the user to have knowledge about the process which is not explicitly contained in the process model. However, it
makes the same, somewhat misleading, assumption as many other approaches, i.e. it assumes that the model correctly models the process under consideration. Obviously, if the process model contains errors, it is not a correct model of the process, however, if the model does not contain errors, it can still be wrong (i.e. the model does not reflect the real desired behaviour).

The research area of conformance checking focuses on this problem, i.e. to what extent does a process model fit a process, where the process is expressed as an event log. In other words, a given process model is checked to what extent it can reproduce a process log. However, this approach has a similar problem as the current problems in process mining, i.e. how do we relate the low-level events in a process log to the activities expressed in the process model.

### 9.3.3 Summary

We conclude this section, and therefore this thesis, with the expectation that process mining and verification, in the near future, will play an important role in the enactment phase of the BPM life cycle. Using verification and mining techniques, we will be able to recommend what users should do to optimize their performance. The ultimate goal is to enable a new breed of Process Aware Information Systems that provide a better balance between flexibility and support.


Process Mining and Verification

Summary

As organizations continuously try to improve the way they do business, operational processes are analyzed with the purpose of gaining insights into the actual behaviour of these processes. Especially since large information systems typically log the steps performed during enactment of an operational process in some sort of event log, there is plenty of input available for this analysis.

Operational processes such as “invoice handling” and “order processing”, especially when analyzed or described, are typically modelled by process models. Typically, any information system supporting users in the execution of the operational process records events related to these processes. These events are stored in event logs. Furthermore, a company has knowledge about the desired or undesired properties of each operational process in some form (for example company policies, such as the requirement that a manager should be notified about all payments exceeding 10,000 euros, or more generic requirements, such as the requirement that each case that was started will eventually be finished).

This thesis addresses two research areas: process mining and process verification. The algorithms and approaches presented in these two areas can be used for the analysis of operational processes from several perspectives, namely: (a) a case perspective, (b) an organizational perspective and (c) a process perspective, although the focus of this thesis is on the case and the process perspective.

The research area of process verification addressed in this thesis focuses on the desired and undesired properties mentioned before. When process models are analyzed, these models are verified to possess certain properties, relating to the usability of the models, e.g. will all cases that are started eventually finish in a desired final state? This type of analysis is referred to as process model verification.

In the area of process model verification, this thesis presents a method for
the verification of Event-driven Process Chains, or EPCs. These EPCs are an informal modeling language, for which a formal executable semantics can not be given in general. Because of the informal semantics of EPCs, verification is far from trivial. In contrast to the verification of formal models, the answer whether a model is correct or not can not always be given, when the EPC is all that is available. Instead, the approach presented in this thesis uses the implicit knowledge that a process owner has about the possible initial and final states of a process. Therefore, the verification approach could result in the answer that a process is correct, but only under the assumptions made by the process owner.

The main contribution of the EPC verification approach is twofold. First, the verification approach allows for conceptual models to be checked for problems. When designing a large information system, it is of great importance to detect mistakes as early as possible. Since EPCs are conceptual models, used before the actual implementation phase, a verification approach for EPCs is an important contribution. Second, if the result of the verification approach is that the EPC is not incorrect (i.e. it is correct under the assumptions made by the process owner), an executable semantics for that specific EPC is a by-product of the verification process. Therefore, the EPC that started out as a conceptual model of an operational process is now an executable specification of that process.

Another type of verification addressed in this thesis is so-called log-based verification. This thesis presents a log-based verification approach using Linear Temporal Logic (LTL). This logic allows for the formal specification of properties relating to cases, in terms of temporal operators, such as “eventually” and “always” (e.g. always if activity “sale made” is executed then eventually the activity “receive payment” should be executed). The approach presented in this thesis can be applied from any of the three given perspectives. An example of a property from a case perspective that can be formalized using LTL is the question whether a case was completed on time, i.e. “Is the duration of a case less than 5 days?” From an organizational perspective, questions like “Are there any people that both submitted a request and approved that same request?” can be asked.

Besides the EPC verification approach and the LTL approach, this thesis presents several process mining algorithms. The research area of process mining focuses mainly on the analysis of an operational process in general, i.e. using logs that are created during enactment of an operational process, the operational process is analyzed. Within the broad area of process mining, this thesis focuses on process discovery, i.e. how to use the event log to derive a process model that describes the operational process. In order to derive process models from event logs, different types of event logs are considered, namely totally ordered and partially ordered logs.

The starting point for most process discovery algorithms and approaches presented in this thesis is a process log. A process log is an event log of an operational process, such that (1) each event in the log refers to an activity, (2) each event in
the log refers to a case, (3) each event describes what happened for that activity (e.g. “schedule”, “start” or “complete”) and (4) the events in the log are ordered in some way. Furthermore, process logs may contain information about people (i.e. who initiated each event), time (i.e. when did the event occur) and other data (e.g. which amount was involved in an insurance claim).

At the basis of the first category of algorithms presented in this thesis, lies a process log in which the events are totally ordered. Using the totally ordered log, the first approach, which is based on the Theory of Regions, generates a process model in terms of a Petri net in a single step. The other two approaches first abstract from the log by determining causal dependencies and parallel relations between events. The $\alpha$-algorithm then uses these relations to again generate a Petri net directly. The final approach uses these events to convert each case to a partial order on its events, which can be used by an aggregation algorithm. This aggregation algorithm is such that by aggregating the partial orders, the types of splits and joins in the process model can be discovered and the result is an EPC or a Petri net. The main contribution of this algorithm is that the resulting EPC will never be considered incorrect by the EPC verification approach mentioned before, under the assumption that the initial and final states observed in the log are desired states.

For the second category of algorithms, it is assumed that the process log contains partially ordered cases to begin with, which are aggregated into a process model. Again, three algorithms are presented, the first of which focuses on the aggregation of Message Sequence Charts (MSCs). MSCs are a scenario-based language, which typically are partial orders on events. The second algorithm focuses on the aggregation of runs of EPCs, which are used by the commercial performance monitoring tool Aris PPM. The third and final algorithm in this category considers partial orders in the form of causal runs, which are Petri nets describing individual cases. The aggregation algorithm for causal runs yields a single Petri net if the labels of all places are known and a class of Petri nets describing the operational process if the labels of places are unknown, which is typically the case in process mining.

For most of the algorithms in this thesis, good tool support is essential, i.e. a verification approach for EPCs is only useful if there is a corresponding implementation that can work with EPCs taken from modelling tools used in industry. Furthermore, any implementation of process mining algorithms should allow for its result to be analyzed and compared against other results. The (Pro)cess (M)ining framework ProM was developed for this purpose and most of the algorithms presented in this thesis are implemented in ProM. Furthermore, ProM currently includes many analysis and conversion algorithms, as well as import and export functionality for many formalisms, such as EPCs and Petri nets.
Process Mining and Verification

Samenvatting

Organisaties proberen voortdurend om hun bedrijfsvoering te optimaliseren. Operationele processen worden geanalyseerd om inzicht te krijgen in het werkelijke gedrag van deze processen. Er is veel informatie beschikbaar voor deze analyse, temeer doordat grote informatiesystemen typisch “event logs” bijhouden van de stappen die gezet zijn gedurende de uitvoering van een operationeel proces.

Operationele processen, zoals “facturering” en “afhandeling bestelling”, worden typisch gemodelleerd met behulp van procesmodellen, zeker wanneer deze geanalyseerd of beschreven worden. Informatiesystemen die gebruikers ondersteunen tijdens het uitvoeren van operationele processen slaan in het algemeen “events” op die aan deze processen gerelateerd zijn. Deze “events” worden opgeslagen in zogenaamde “event logs”. Tevens hebben de bedrijven kennis over gewenste en ongewenste eigenschappen van ieder operationeel proces in een of andere vorm. (Een voorbeeld hiervan is een beleid waarbij iedere betaling van meer dan 10.000 euro gemeld moet worden aan de leidinggevende, of meer algemeen dat iedere casus waaraan wordt begonnen uiteindelijk ook wordt afgemaakt.)

In dit proefschrift worden twee onderzoeksgebieden gepresenteerd, namelijk “process mining” en “process verification”. De algoritmen en methoden die gepresenteerd worden in beide onderzoeksgebieden kunnen toegepast worden voor de analyse van operationele processen vanuit verschillende perspectieven, te weten (a) het casus perspectief, (b) het organisatie perspectief en (c) het proces perspectief. Dit proefschrift richt zich alleen op het casus perspectief en het proces perspectief.

Het onderzoeksgebied “process verification” richt zich op de eerder genoemde gewenste en ongewenste eigenschappen van procesmodellen. Gecontroleerd wordt of deze modellen bepaalde eigenschappen bezitten die gerelateerd zijn aan
Samenvatting

hun bruikbaarheid. (Zullen alle zaken waaraan begonnen wordt uiteindelijk op de juiste manier afgewerkt kunnen worden?) Dit soort analyse wordt ook wel procesmodel-verificatie genoemd.

In het vakgebied van procesmodel-verificatie presenteert dit proefschrift een methode voor de verificatie van “Event-driven Proces Chains”, of EPCs. Deze EPCs zijn een informele modelleertaal, waarvoor een formele en uitvoerbare semantiek in het algemeen niet bestaat. Juist door dit informele karakter van EPCs is de verificatie daarvan verre van triviaal. In tegenstelling tot de verificatie van formele modellen, kan bij EPCs de vraag of een model al dan niet correct is, niet altijd gegeven worden op basis van alléén dat model. Daarom gebruikt de methode in dit proefschrift ook impliciete kennis die de proceseigenaar heeft over de mogelijke begin- en eindtoestanden van het operationele proces. Om die reden kan het antwoord van de verificatie-methode zijn dat het model correct is, onder de aannamen die door de proceseigenaar gemaakt zijn.

De belangrijkste bijdrage van de EPC verificatiemethode is tweeledig. Ten eerste staat de methode toe dat conceptuele modellen gecontroleerd worden op bepaalde gewenste eigenschappen. Als grote informatiesystemen ontworpen worden is het erg belangrijk om ongewenste eigenschappen zo vroeg mogelijk op te sporen. Aangezien EPCs conceptuele modellen zijn, die gebruikt worden voordat een informatiesysteem ook daadwerkelijk geïmplementeerd wordt, is een verificatiemethode voor EPCs een belangrijke bijdrage. Ten tweede is het zo dat als de verificatie-methode zegt dat de EPC niet incorrect is (oftewel correct onder de aannamen die door de proceseigenaar gemaakt zijn), dan is een uitvoerbare semantiek een bijproduct van de verificatie. Daarom kan een EPC die in eerste instantie een conceptueel model was van een operationeel proces nu gebruikt worden bij de uitvoering van dat proces.

Een ander type verificatie waar in dit proefschrift aandacht aan besteed wordt is de zogenaamde “log-based” verificatie. Dit proefschrift presenteert een “log-based” verificatie-methode waarbij gebruikt gemaakt wordt van Lineaire Temporele Logica (LTL). Deze logica staat toe om eigenschappen van zaken te beschrijven met temporele operatoren, zoals “uiteindelijk” en “altijd” (bijvoorbeeld: Altijd als de activiteit “verkoop afronden” uitgevoerd wordt, moet uiteindelijk de activiteit “ontvang betaling” uitgevoerd worden). De methode gepresenteerd in dit proefschrift kan uitgevoerd worden vanuit alle drie de eerder genoemde perspectieven. Een voorbeeld van een eigenschap vanuit een casusperspectief die geformaliseerd kan worden met LTL is de vraag of die casus op tijd afgerond was, ofwel “Heeft de casus minder dan 5 dagen geduurd?”. Vanuit een organisatieperspectief kunnen vragen als “Zijn er medewerkers die een verzoek hebben ingediend en dat zelf hebben goedgekeurd?” gesteld worden.

Naast de EPC verificatie-methode en de LTL methode, worden in dit proefschrift verschillende “process mining” algoritmen gepresenteerd. Het onderzoeksgebied van “process mining” richt zich op de analyse van operationele processen
in het algemeen. Met gebruik van de logs die gemaakt zijn tijdens de uitvoering van een proces, wordt dat proces geanalyseerd. Binnen het brede spectrum van “process mining” richt dit proefschrift zich op “process discovery”: Hoe kan een “event log” gebruikt worden om een procesmodel af te leiden dat het operatieve proces beschrijft? Om procesmodellen af te leiden uit event logs worden twee verschillende types logs beschouwd, namelijk *totaal geordende* en *partieel geordende* logs.

Het beginpunt voor de meeste “process discovery” algoritmen en methoden in dit proefschrift is een *proces-log*. Een proces-log is een “event log”, waarin (1) ieder “event” aan een activiteit refereert, (2) ieder “event” aan een casus refereert, (3) ieder “event” beschrijft wat er met betrekking tot de activiteit plaatsvond (bijvoorbeeld “gepland”, “gestart”, of “afgesloten”) en (4) alle “events” op een of andere manier geordend zijn. Tevens kunnen proces-logs informatie bevatten over mensen (bijvoorbeeld, wie een “event” veroorzaakte), alsmede andere data (bijvoorbeeld, wie een “event” veroorzaakte), alsmede andere data (bijvoorbeeld, wie een “event” veroorzaakte).

De basis voor de eerste categorie van algoritmen in dit proefschrift wordt gevormd door proces-logs, waarin de “events” *totaal geordend* zijn. Met behulp van deze totale ordening genereert de eerste methode, gebaseerd op de “Theory of Regions”, een procesmodel in de vorm van een *Petri-net* in een enkele stap. De andere twee methoden abstraheren eerst van de proces-log door het ontdekken van causale afhankelijkheden en parallelle relaties tussen “events”. Het *α-algoritme* gebruikt deze relaties om ook een Petri-net te genereren. De laatste methode gebruikt de relaties om iedere casus te converteren in een *partiële ordening* op de events, welke gebruikt kan worden door een *aggregatie* algoritme. Dit aggregatie-algoritme zorgt ervoor dat tijdens de aggregatie van de partiële ordeningen de types van de ingangen en uitgangen bepaald worden. Het resultaat van het algoritme is een Petri-net, of een EPC. De belangrijkste bijdrage van dit algoritme is dat de resulterende EPC nooit als incorrect beschouwd wordt door het eerder genoemde verificatie-algoritme, onder de aanname dat de begin- en eindtoestanden die in de log geobserveerd zijn ook wenselijk zijn.

Voor de algoritmen in de tweede categorie van “process discovery”, wordt aangenomen dat de proces-log initieel al *partieel geordende casussen* bevat, die geaggregeerd worden in een procesmodel. Opnieuw worden drie algoritmen gepresenteerd, waarvan de eerste zich richt op het aggregeren van “Message Sequence Charts” (MSCs). MSCs beschrijven met scenario’s mogelijke procesuitvoeringen en deze scenario’s zijn partiële ordeningen op “events”. Het tweede algoritme richt zich op het aggregeren van “runs” van EPCs, die gebruikt worden in het commerciële “process performance monitoring tool” Aris PPM. Het derde en laatste algoritme in deze categorie beschouwt partiële ordeningen in de vorm van “causal runs”, oftewel Petri-netten die individuele casussen beschrijven. Dit aggregatie-algoritme resulteert in een enkel Petri-net als de labels van alle plaatsen bekend zijn en in een klasse van Petri-netten als deze labels niet bekend zijn. Dit laatste
is typisch het geval in “process mining”.

Voor de meeste algoritmen in dit proefschrift is een goede implementatie van belang. Een verificatie-methode voor EPCs is namelijk alleen bruikbaar als er software bestaat die EPCs kan lezen vanuit verschillende modeller-programma’s. Tevens moet men in staat zijn om de resultaten van ieder “process mining” algoritme te analyseren en te vergelijken met andere resultaten. Met dit doel is het “(Pro)cess (M)ining framework ProM” ontwikkeld en de meeste algoritmen die gepresenteerd worden in dit proefschrift zijn daarin geïmplementeerd. Daarnaast bevat ProM op dit moment vele andere analyse- en vertalingsalgoritmen, alsmede import- en exportfunctionaliteit voor vele formalismes, zoals EPCs en Petri-netten.
Acknowledgements

In the beginning of 2002, still during my master studies in Computer Science, I started an internship in the area of process mining. At that time, Ana Karla Alves de Medeiros just started her PhD in that same area, and while she was working on all the compulsory PhD courses, I started on the implementation of the α-algorithm and with that I started to enjoy the mysteries of process mining. From September 2003, when Wil van der Aalst hired me as a PhD candidate in his research group, I have had a lot of support from many people, but first and foremost, I would like to thank Ana Karla, for always taking the time to discuss my work during the four years we shared an office.

I cannot possibly mention all the people I met during my PhD work and with whom I had stimulating discussions. However, I would like to thank some of them in particular. First, I thank Jeroen van Luin, Eric Verbeek, Jochem Vonk and Remco Dijkman for always starting the working day with lively discussion over the right amount of coffee. Furthermore, I thank all my colleagues from the Information Systems group in the Department of Technology Management, currently lead by Paul Grefen, for providing me with a very pleasant working environment, as well as many possibilities for (international) collaborations.

Outside of my research group, I would like to thank Kristian Bisgaard Lassen, Tobias Blickle, Mark van den Brand, Nadia Busi, Jörg Desel, Kees van Hee, Ekkart Kindler, Ton de Kok, Jan Mendling and Vladimir Rubin in particular. They all played an important part in the research presented in this thesis, as without their comments, suggestions and corrections, I would never have been able to finish this thesis on time. Especially when it comes to the formal definitions and proofs I have to thank Eric Verbeek for helping me to identify and solve problems, inconsistencies and loopholes.

A year before I concluded the work on this thesis, Wil van der Aalst moved to the Computer Science department and I am grateful to him for making it possible for me to join his new research group as a postdoc. In that postdoc project, I will continue to work in the area of process mining, but I will also focus on the visualization of the results, which until now has not received enough attention.

Boudewijn van Dongen
Eindhoven, May 22, 2007
Curriculum Vitae

Boudewijn Frans van Dongen was born on 26th of December 1979 in Dongen, The Netherlands, where he also grew up. From 1992 to 1998, he went to high school at the dr. Schaepman College, later Cambreur College in Dongen.

After finishing highschool, he studied Computer Science at the Eindhoven University of Technology from 1998 to 2003. In the first two years, he obtained a degree (propadeuse) in Computer Science, as well as in Mathematics. In 2003, his studies were concluded with a Master of Science degree in Computer Science, after a 9 months project within the well-known telecommunications company Vodafone. The title of his thesis is “GPRS Dimensioning, an algorithmic approach”.

Following his Master studies, from September 2003 Boudewijn became a PhD candidate in the department of Technology Management of the Eindhoven University of Technology. The focus of his doctoral studies, supervised by prof.dr.ir. Van der Aalst and co-supervised by dr.ir. Verbeek, was on process mining and verification and lead to a number of publications in international journals, as well as on international conferences and workshops. His doctoral studies were completed in July 2007 and resulted in a thesis entitled “Process Mining and Verification”.

Boudewijn’s work has had great impact, as shown by many publications, such as the survey paper in the ISI journal “Data and Knowledge Engineering”, which was selected as hot paper (January 2005) by ESI Special Topics and an application paper, which has been in the top 3 of most downloaded articles of the ISI-journal “Information Systems” since the article appeared online. Furthermore, in the area of process verification, Boudewijn has not only published in the scientific community, but his work has also been published in popular magazines (e.g. “De automatiseringsgids” (in Dutch), “BPTrends” and “iX-Magazine” (in German)) and has therefore lead to great visibility in industry.

Finally, Boudewijn has been (and until today still is) heavily involved in the development of the process mining framework ProM (www.processmining.org). This process mining framework is a plugable environment in which researchers are enabled to implement their work on process mining and related subjects. Combining efforts from researchers all over the world, the framework currently contains over 180 plug-ins, capable of performing all kinds of analysis. Further-
more, it is capable of importing from and exporting to many industrial tools, such as the Aris Toolset, Microsoft Visio, Protos, CPN tools, etc.

Currently, Boudewijn is working as a Postdoc in the Computer Science department of the Eindhoven University of Technology. His main research area in this project is the combination of Process Mining and Visualization, i.e. on the question how to present the results of process mining techniques to end users in an intuitive way? To this end, he works in the Architecture Information Systems group lead by prof.dr.ir. Van der Aalst. Boudewijn can be reached at b.f.v.dongen@tue.nl.
<xml version="1.0" encoding="UTF-8"?>
<!-- MXML version 1.0 -->
<!-- This is an event log created to be analyzed by the process mining framework ProM. -->
<!-- ProM can freely be obtained from http://www.processmining.org/ -->
  <Process id="Curriculum Vitae" description="Academic Life">
    <ProcessInstance id="bfvdongen" description="Boudewijn van Dongen">
      <AuditTrailEntry>
        <Data>
          <Attribute name="University">Eindhoven University of Technology</Attribute>
          <Attribute name="Department">Mathematics and Computer Science</Attribute>
        </Data>
        <WorkflowElement>Master Studies</WorkflowElement>
        <EventType>start</EventType>
        <Timestamp>1998-08-31T08:45:00.000+01:00</Timestamp>
      </AuditTrailEntry>
      <AuditTrailEntry>
        <Data>
          <Attribute name="Master Thesis">GPRS Dimensioning, an algorithmic approach</Attribute>
        </Data>
        <WorkflowElement>Master Studies</WorkflowElement>
        <EventType>complete</EventType>
        <Timestamp>2003-08-28T16:00:00.000+01:00</Timestamp>
      </AuditTrailEntry>
      <AuditTrailEntry>
        <Data>
          <Attribute name="University">Eindhoven University of Technology</Attribute>
          <Attribute name="Department">Technology Management</Attribute>
        </Data>
        <WorkflowElement>PhD Project</WorkflowElement>
        <EventType>start</EventType>
        <Timestamp>2003-09-01T08:45:00.000+01:00</Timestamp>
      </AuditTrailEntry>
      <AuditTrailEntry>
        <Data>
          <Attribute name="PhD Thesis">Process Mining and Verification</Attribute>
        </Data>
        <WorkflowElement>PhD Project</WorkflowElement>
        <EventType>complete</EventType>
        <Timestamp>2007-07-03T15:25:00.000</Timestamp>
      </AuditTrailEntry>
    </ProcessInstance>
  </Process>
</WorkflowLog>