Implementation of a Process Mining Tool on top of an Event Graph Database

Hernandez Siles, Valdemar

Award date:
2021

Department of Mathematics and Computer Science
Process Analytics

Implementation of a Process Mining Tool on top of an Event Graph Database

Master Thesis End Project Report

Valdemar Hernandez Siles

Supervisors:
Dr. Dirk Fahland
TU/e
d.fahland@tue.nl

15-08-2021

Abstract

Graph-based data models can be used to store multi-dimensional event data and describe the relations between multiple entity identifiers and sequential information in one data structure. There are existing data models that describe how event data can be represented inside a graph database, however, none of them describe how a graph database can be used for an in-depth analysis that includes the execution of multiple process mining-related activities. Therefore, it becomes relevant to determine if it is possible to build upon these graph-based data models to obtain a viable alternative to execute these activities and provide significant results for a process mining project. In this thesis, we present a tool built on top of a graph database that is able to execute the essential activities that should be executed in a process mining project with the help of a tool. We provide the details on its implementation and the execution of these activities through a user interface. Moreover, we present an extension to one of the existing data models that allows us to execute and analyze the results a process discovery algorithm, the Heuristic Miner. We show that it is not only possible to execute process mining on top of a graph database, but there are additional advantages obtained from its usage.

Contents

Chapter 1

Introduction

In this Chapter, we provide the introduction to this thesis. First, we provide the context of this work. Then, we present the state of the art related to process mining on relational and graph databases. Then, we define our research questions and present the methodology used to conduct our research. Finally, we describe the results of our work.

1.1 Context

Process mining projects aim to discover, monitor and improve real processes by extracting knowledge from event logs stored in information systems. These event logs represent the starting point for process mining, since all process mining techniques assume that it is possible to obtain a sequential order of events that represent the activities executed in a process [3]. Event logs define a case identifier to store the activities as one sequence of events per case. In turn, these sequences can be queried for behavioral properties.

In practice, processes may involve multiple related entities, where every event is linked to more than one case identifier, in which case, we say the event data describes the behavior of multiple dimensions. Relational databases (RDBs) can store this behavior through *1:n* and *n:m* relations between events and case identifiers. However, to analyze the behavior of multi-dimensional data stored in an RDB, it is necessary to extract the data into a sequential format, which may lead to false behavioral information.

Graph-based data models can overcome these issues since they can describe the relations between multiple entities and sequential paths in one data structure. Events and case identifiers can be stored as nodes, and the information between events and cases can be stored as relationships. Then, the sequential order of events can be represented through paths of nodes

connected by relationships.

In [1], Esser and Fahland propose a data model for multi-dimensional event data based on labeled property graphs capable of representing multiple different, related entities and the relations between entities and events through correlation relationships. This data model avoids the shortcomings related to multi-dimensional data found in existing data models.

Given the benefits of this data model, it becomes relevant to determine if it is possible to build upon it and provide a viable alternative for process mining analysis, allowing the discovery and improvement of real processes through a graph database.

1.2 State of the Art

Dijkman, R., Gao, J., Syamsiyah, A. et al [26] present a first step towards in-database process mining, where they define a relational algebraic operator to extract the "directly follows" relations from a log stored in an RDB. However, they state that it is hard to work with cases and propose the aggregation of properties from an event level to a case level in the future.

In another study, Romero, M. and Rodriguez, A. [27] propose a graph-based approach to modeling events and their interrelations, but their focus is mainly on extending the graph data models to include temporal and spatial settings and manage different levels of granularity to represent events and their relationships.

In a previous master's thesis, Türkyılmaz [30] discusses how to conduct process mining analysis on an event log stored in a graph database. However, as part of the findings, it is mentioned that future work is still needed to make deeper process mining analysis and include the implementation of different process mining algorithms.

Finally, the graph-based data model proposed by Esser and Fahland [1] allows the modeling and querying of multi-dimensional event data, but also mention that a more general data model could allow the development of new event data analysis and process mining techniques.

1.3 Research Questions

As we can see from Section 1.2, there is no graph-based data model that not only provides a correct representation of multi-dimensional event data, but allows the execution of other process mining-related activities and techniques to provide an in-depth analysis on the process being evaluated.

The data model from [1] provides a promising foundation to do this, but it remains to be seen if this graph-based model can be used to execute process mining activities and techniques and store models that describe the behavior of the processes. This is why our aim is to determine if it is possible to build upon this data model to obtain a viable alternative to execute the activities that provide significant results in a process mining project. Based on this objective, we define the following research questions:

RQ1. Is it possible to build on top of the data model proposed in [1] to execute process mining-related activities in a graph database?

RQ2. Can we store process models in the graph database? If so, how would the execution of process mining change in this environment?

RQ3. Is there an added benefit on the joint storage of different process mining components in the graph database?

1.4 Methodology

We built a tool in Java that, through its connection to a graph database, allows us to execute the most essential activities in a process mining project while using the data model presented in [1] as its foundation.

First, in Chapter 3, we analyze the stages of the PM^2 methodology to identify all the activities that a process mining tool should perform during a process mining project. Based on this analysis, we define a 5-layer architecture to guide the implementation of the tool. Then, by identifying the most essential process mining-related activities, we delimit the scope of our project, which resulted in the removal of the third layer of the architecture from our implementation.

Secondly, in Chapter 4, we start by presenting the layout of the user interface for the tool and discussing how we handle multi-dimensional data. Then, we describe how we adapt the data model from [1] into our tool through the first two layers of the 5-layer architecture. In the first layer, we implement functionalities related to the import of event data into the graph database. In the second layer, we implement functionalities related to the creation and enrichment of event logs by defining entity identifiers and event classes. For every functionality, we describe its queries and how they can be executed through the user interface, together with the visualization of the results.

Thirdly, in Chapter 5, we implement the fourth layer of the architecture by implementing functionalities related to process discovery. First, we describe our proposal to extend the data model [1], allowing us to define how

process models can be represented through nodes and relations in the graph database. Then, we discuss our implementation of an existing process discovery algorithm, the Heuristic Miner, through the graph data model. After describing the algorithm, we present our implementation, once again detailing the queries and interactions with the user interface needed to execute this activity and visualize the results through our tool.

Finally, in Chapter 6, we implement the fifth layer of the architecture by implementing functionalities related to the diagnosis of the process. Given our extended data model, we identify two activities that can be implemented in our tool to help in the diagnosis, the model comparison and the model querying.

For the model comparison, we first implement a transformation of the results of the Heuristic Miner algorithm into Petri nets, the most common modeling language, to account for the fact that distinct discovery algorithms may generate different outputs. Then, we present how the model comparison can be executed in our tool. As for the model querying, we first define the queries that find process models stored in the graph database based on graph patterns. Then, we present how the model querying can be executed in our tool.

1.5 Results

We defined four experiments to evaluate our implementation and help us determine the answer to the research questions.

Experiment 1 In the first experiment, we use our tool to run a process mining exploratory analysis in a real dataset to help us determine if the activities implemented in the tool provide significant results for the analysis. Then, we use the process mining tool ProM to run the same analysis and be able to compare the results and identify the differences in their execution.

From this experiment we could determine that our tool does execute successfully the distinct process mining-related activities implemented, allowing us to obtain significant insights for the process analysis. In terms of usability, even if there are changes in the steps that need to be executed to obtain the results, the interactions between the user and the tool are similar to what has to be done in ProM (63 against 66 interactions respectively), but in terms of performance, ProM proved to be better since it was able to execute its activities in less than 2 minutes, while our tool took around 14 minutes.

Additionally, our tool proved to have a couple of advantages. First, it allowed us to execute a model comparison as part of the process mining

analysis, something that ProM does not provide. Also, the model obtained by our tool does a better job in describing the behavior of the process due to our handling of data dimensionality through graph structures.

Experiment 2 In the second experiment, we dive deeper into the performance of our tool, testing its scalability with larger datasets. We generated 5 random subsets of data, which contained 20, 40, 80, 160, and 320 cases respectively. For each of them, we registered the time it took from the import of the data into the graph database until the visualization of the discovered process model. Similar to the previous experiment, we ran the same experiment in ProM to provide context on our results.

In this case, we saw that our tool can still be improved on this regard, since the scale up is significantly larger than ProM. For the event log containing 160 cases, our tool took around 85 minutes to complete its tasks, while ProM took 1 minute. Then, for the event log containing 320 cases, the time went up to 330 minutes, while ProM had a more linear scale up and the tasks took 2 minutes overall to complete.

So even if it is possible to execute several process mining-related activities, including the creation and storage of process models, the performance of the tool must be improved to be considered a more viable option to work with during process mining projects.

Experiment 3 In the third experiment, we analyze what additional benefits can we obtain from the storage of different process mining components in the graph database. To do this, we define a query that takes advantage of a graph relation defined by us as an extension to the original data model from [1].

Using this relation, which connects the process models with the event data, we were able to obtain additional insights on how well a given process model is representing the behavior of the event log.

We realized that further extensions for our tool can be made by expanding on the approach used in this experiment to include additional process mining-related activities such as conformance checking.

Experiment 4 Finally, in the fourth experiment we dive further in the analysis of the benefits of executing process mining in a graph environment, where we try to identify how our tool contributes to the discipline of process mining based on the functionalities implemented.

To do this, we refer to the Process Mining Manifesto [3] to identify which challenges process mining is currently facing and identify which of them are

being addressed by our implementation.

From the results of this experiment, we identified that 17 of our functionalities help address 4 of the process mining challenges, letting us conclude that our tool does not only prove that it is possible to execute process mining on top of a graph database, but it also provides additional value to the discipline in general.

These experiments helped us identify that it is not only possible to execute process mining-related activities on top of a graph database, but there are added benefits in doing so, such as the correct representation of multi-dimensional data, the model comparison, or the connection between process models and event data. In addition, the ease with which these activities can be performed through the user interface allows for more flexibility in the process mining analysis, where at any point we can define new connections in the data to analyze the process from different perspectives.

Chapter 2

Background

In this Chapter, we provide the concepts, definitions, and work related to this thesis. First, we provide the definition of process mining. Then, we provide additional details on the discipline of process mining through the description of the PM^2 methodology and the Process Mining Manifesto. Then, we describe graph databases and Neo4j, an open-source, NoSQL database that implements the property graph model. Then, we define the CSV files, which we use as input for our tool. Then, we describe GraphStream, a Java library used for the modeling and analysis of dynamic graphs. Finally, we describe the existing research related to the use of graph databases for process mining, with special emphasis on a graph data model proposed by Esser and Fahland.

2.1 Preliminaries

2.1.1 Process Mining

Wil van der Aalst [6] defines process mining as the missing link between *data science* and *process science*. Process mining can be placed between machine learning and data mining on side and process modeling and analysis on the other. The objective of this discipline is to discover, monitor and improve real processes by extracting knowledge from the event logs available in information systems.

There are three main types of process mining, *discovery*, which takes an event log and produces a model without any a-priori information, *conformance*, where an existing process model is compared with an event log of the same process, and *enhancement*, where the idea is to extend or improve an existing process model.

In this thesis we discuss the requirements needed for a tool, built on top of a graph database, to help in the execution of an in-depth process mining analysis.

2.1.2 PM2 Methodology

PM^2 is a methodology that guides the execution of process mining projects [2]. This methodology guides organisations performing process mining projects whose aim is to improve process performance or ensure their compliance to rules and regulations.

The methodology consists of 6 stages related to the inputs and outputs of the data objects, models, and four goals: (1) research question definitions, (2) performance findings, (3) compliance findings, and (4) improvement ideas. The 6 stages of the methodology are the following [2]:

1. *Stage 1 - Planning.* The objective of this stage is to set up the project and determine the research questions.

2. *Stage 2 - Extraction.* The objective of this stage is to extract event data and, optionally, process models.

3. *Stage 3 - Data Processing.* The objective of this stage is to create event logs as different views of the event data and process logs in such a way that their format is optimal for the next stage.

4. *Stage 4 - Mining and Analysis.* The objective of this stage is to apply process mining techniques on event logs and aim to answer the research questions.

5. *Stage 5 - Evaluation.* The objective of this stage is to relate the analysis findings to improvement ideas that achieve the project's goals.

6. *Stage 6 - Process Improvement and Support.* The objective of this stage is to use the gained insights to modify the actual process execution.

2.1.3 Process Mining Manifesto

The Process Mining Manifesto is a manifesto written by members and supporters of the *IEEE Task Force on Process Mining* whose goal is to promote the research, development, education, implementation, evolution, and understanding of process mining [3].

Among the contents of the manifesto, which include the state of the art of the discipline and its guiding principles, it is also included a list of challenges that need to be addressed. The list describes the following 11 challenges [3]:

C1. *Finding, Merging, and Cleaning Event Data.* It still takes considerable efforts to extract event data suitable for process mining.

C2. *Dealing with Complex Event Logs Having Diverse Characteristics.* Extremely large event logs may be difficult to handle, whereas other extremely small event logs do not provide enough data to make reliable conclusions.

C3. *Creating Representative Benchmarks.* Since process mining is an emerging technology, good benchmarks are still missing.

C4. *Dealing with Concept Drift.* This refers to the need to address the situation where the process is changing while being analyzed.

C5. *Improving the Representational Bias Used for Process Discovery.* The selection of a target language may limit the search space, where results that cannot be represented by the language cannot be discovered.

C6. *Balancing Between Quality Criteria such as Fitness, Simplicity, Precision, and Generalization.* Improved process mining algorithms need to be developed to better balance these four competing quality dimensions.

C7. *Cross-Organizational Mining.* New analysis techniques need to be developed to handle scenarios where the event logs of multiple organizations are available for analysis.

C8. *Providing Operational Support.* Process mining should not be restricted to offline analysis and should also be used for online operational support.

C9. *Combining Process Mining With Other Types of Analysis.* Combining automated process mining techniques with interactive visual analytics could allow us to extract more insights from event data.

C10. *Improving Usability for Non-Experts.* The end-user interactions with the results of process mining can be very valuable, but intuitive user interfaces are required to allow them to happen.

C11. *Improving Understandability for Non-Experts.* To avoid problems with the understanding of the outputs or reaching incorrect conclusions, the results should be presented in a suitable representation.

2.1.4 Graph Databases

A graph database is a database that stores information in sets of nodes and relationships. The *labeled property graph* is the most popular form of graph models [23], which has the following characteristics:

- Contains nodes and relationships.
- Nodes contain properties (key-value pairs).
- Nodes can be labeled with one or more labels.
- Relationships are named and directed, and always have a start and end node.
- Relationships can also contain properties.

Neo4j

Neo4j is an open-source, NoSQL database that implements the property graph model. Neo4j uses Cypher as its specific graph database query language to store and retrieve information [23]. Cypher allows us to find data inside the database that matches a specific pattern. For example, for the graph pattern shown in Figure 2.1, we can define the following Cypher query to help us find the mutual friends of Tony:

```
MATCH (a:Person{name:'Tony'})-[:KNOWS]->(b)-[:KNOWS]->(c),
  (a)-[:KNOWS]->(c)
RETURN b, c
```

The **MATCH** Cypher clause shown in the query takes a pattern as an input, and allows us to find the nodes and relationships that match the pattern. Then, the **RETURN** clause specifies which nodes, relationships, and properties should be included in the output of the query. In our example, we are returning the (:PERSON) nodes that represent Steve and Bruce.

Other important Cypher clauses include:

- **WHERE** Provides criteria for filtering results.
- **CREATE** Creates nodes and relationships.
- **MERGE** Ensures that the given pattern exists in the graph. If it does not exist, it is created.
- **DELETE** Removes nodes, relationships and properties from the database.
- **SET** Set property values.

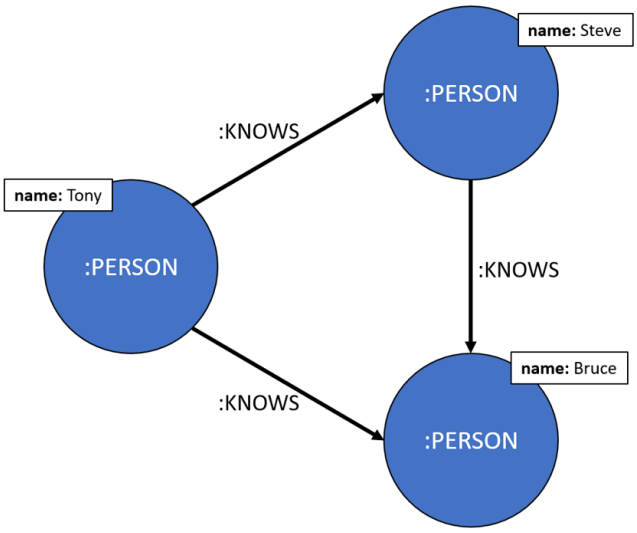


Figure 2.1: Example graph pattern that describes the "knows" relation between three persons.

- **WITH** Merges results from two or more queries.
- **DISTINCT** Removes duplicates in the result.
- **ORDER BY** Order the results based on a given property.
- **UNWIND** Expands the rows that have a list property into several rows, with one row per item in the list.
- **LOADCSV** Load data into the database from a CSV file.

2.1.5 Comma Separated Values (CSV) File

The Comma Separated Values (CSV) file is the most prominent file format to represent tabular data. As mentioned in [24], CSV data is a standard way for exchanging and converting data between different related applications. According to [24], two of its key format specifications include:

1. CSV is a 2D format constructed by rows and columns of data, each row containing multiple cells.
2. CSV is a "text-based" format, which makes it flexible for processing with all types of textual applications.

In this thesis, we use CSV files as the main input to represent the event data that is meant to go through a process mining analysis, where each row represents a single event.

2.1.6 GraphStream

GraphStream is a Java library for the modeling and analysis of dynamic graphs [25]. This library lets us generate, import, export, measure, layout and visualize graphs.

GraphStream also allows us to store any kind of data attribute on the nodes and relationships of the graph, such as numbers, strings or any object.

The Java code below shows an example of how a graph can be defined and visualized in a panel. Executing this code can produce the result shown in Figure 2.2.

```
MultiGraph graph = new MultiGraph("g");

graph.addNode("A");
graph.addNode("B");
graph.addNode("C");

graph.addEdge("AB", "A", "B");
graph.addEdge("BC", "B", "C");
graph.addEdge("CA", "C", "A");

g.getNode("A").setAttribute("ui.color", Color.BLUE);
g.getNode("B").setAttribute("ui.color", Color.YELLOW);
g.getNode("C").setAttribute("ui.color", Color.WHITE);

graph.display();
```

In this thesis, we include the GraphStream library in the Java implementation of our tool to display the graphs that represent the event data, taking advantage of the functionalities provided by the library to format the nodes and edges and provide accurate representations of the data.

2.2 Related Work

2.2.1 Process Mining on Databases

Dijkman, R., Gao, J., Syamsiyah, A. et al [26] present a first step towards in-database process mining. They define a relational algebraic operator to extract the "directly follows" relations from a log stored in a relational database. This operator is meant to facilitate exploratory process mining, but it must

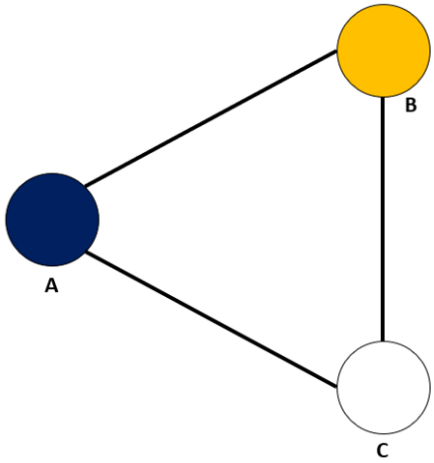


Figure 2.2: Example visualization using the GraphStream Java library.

be constructed with a complex SQL query. However, as part of their findings, they report that it is hard to work with "cases" in a SQL query, because "a log table has a row for each event that happens rather than each case that completes" [26].

Romero, M. and Rodriguez, A. [27] propose a graph-based approach to modeling events and their interrelations. In order to make explicit relations between events, they extend graph data models to include temporal and spatial settings and to manage different levels of granularity to represent events and their relationships. Their main contribution is to "address the modeling, representation, and query specification of events from a different perspective" [27].

Türkyılmaz [30] discusses how to conduct process mining analysis on an event log stored in a graph database. The paper discusses what the schema and the data preparation should be to make proper process mining analysis on graphs. However, it is also mentioned that deep process mining analysis is missing and "future work could look into the implementation of different process mining algorithms" [30].

2.2.2 Multi-Dimensional Event Data in Graph Databases

In [1], Esser and Fahland propose a "general data model for multi-dimensional event data based on *labeled property graphs* that allows storing structural and temporal relations in a single, integrated graph-based data structure in a systematic way".

Their data model allows them to query for multi-dimensional event data

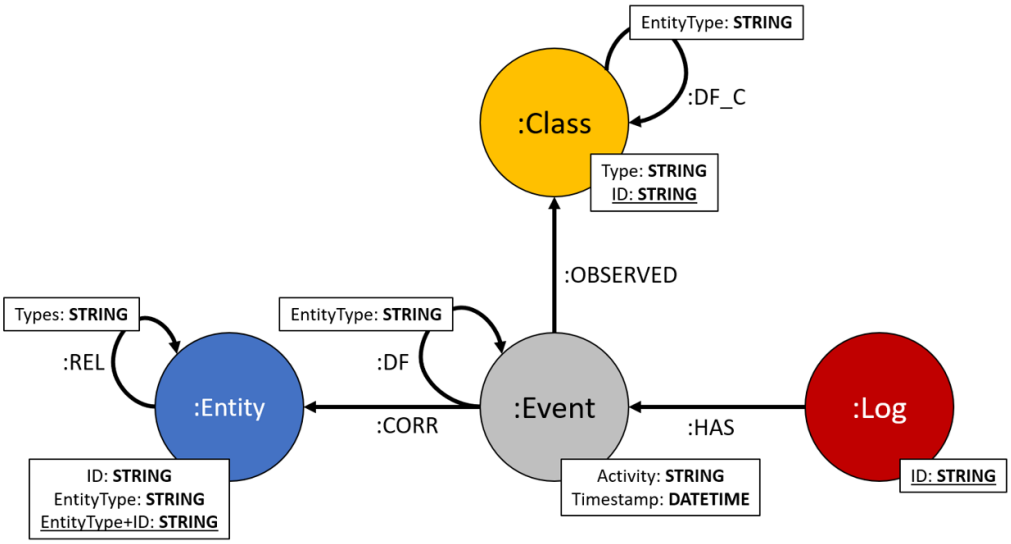


Figure 2.3: Data model proposed in [1] to model multi-dimensional event data in labeled property graphs.

of multiple related entities. They mention that, to analyze the behavior of event data that involves multiple identifiable entities, data must be extracted in a sequential format; however, the correlation of events under a single entity that needs to be done to *flatten* the data and obtain a single-dimensional event log may lead to false behavioral information known as *convergence* and *divergence*.

Convergence A convergent event log contains entries where one activity is executed in several process instances at once. In a relational database, this can be recognized by a *1:n* relation from an event to the process instance [28].

Divergence A divergent event log contains entries where the same activity is performed several times in one process instance. In a relational database, this can be recognized by a *n:1* relation from events to the process instance [28].

The data model presented in [1] can be observed in Figure 2.3. The Event node represents a single event from the event log, with the event's *activity* and *timestamp* as properties. Event nodes can be connected between them through :DF relations, which are determined based on the entity identifiers. Then, the Log nodes allows them to identify which events belong to each log.

The Entity node is used to identify the entity identifier to which each

event is correlated, with the *EntityType* property describing the type of entity, the *ID* property describing the entity identifier, and their combined values serving to provide a unique identifier to the node in the database. The relations between entities are described through the :REL relations.

Finally, the Class node is used to specify event classes. The Class nodes have a unique *ID* property and a *Type* property that is the same for all event classes defined based on a single attribute or a combination of multiple attributes. The :DF_C relations between classes describe the aggregated directly-follows relations between the events related to them.

Esser and Fahland mention that this data model can be seen as a multi-dimensional event log, where events of each entity are ordered by "their" directly-follows relation leading to a partial order of events, avoiding shortcomings of existing event data models and supporting multiple perspectives on the event data from different case identifiers at once.

Chapter 3

Building a Process Mining Tool on top of a Graph Database

In this chapter, we describe how we can build a process mining tool that is able to execute the most essential activities within a process mining project while using the data model presented in [1] as its foundation. First, we identify the activities that a process mining tool should perform throughout a process mining project, analysing how they are impacted when considering a tool built on top of a graph database. Then, we describe the architecture used as a guideline to implement the tool, identifying the activities that provide the minimum viable result for a process mining project, thus defining the final scope of the implementation.

3.1 Identifying Process Mining Activities

In this section, we use the PM^2 methodology, described in Section 2.1.2, as the main reference to identify the activities that need to be carried out during a process mining project. For each of its six stages, we first describe their objectives and identify the activities that can be executed with a process mining tool. Then, we analyze the considerations this entails for a tool built on top of a graph database.

First Stage - Planning The first stage of the PM^2 methodology is *Planning*, whose main objectives are setting up the start of the project and determining what knowledge should be obtained in the form research questions. The three activities that should be carried out in this stage are *Identifying research questions*, which should be answered with the available data, *Selecting business processes* that are meant to be analyzed, and *Composing the*

project team with business owners, system experts and process analysts that will be involved in the project. Although these activities are important to start the project and define its goals, an interaction with a process mining tool is not yet required.

Second Stage - Extraction The objective of the second stage, *Extraction*, is to extract event data and process models. The three activities to be performed in this stage are *Determining the scope* or identifying the data that will be extracted for the analysis, *Extracting event data*, where the event data is created based on the scope determined in the previous activity, and *Transferring process knowledge*, where there is an exchange of information within the project team to ensure the same understanding of the process. The identification and extraction of the most relevant data that will be used as an input for the next stage depends more on domain knowledge and the interaction between team members than the usage of process mining tools, but this stage is still relevant for our tool.

The *Extracting event data* activity at this stage gets the source data into a format ready for process mining, but, for a tool built on top of a graph database, the extracted event data cannot be used directly as an input. For this particular tool, we must consider that the data has to be adapted for the graph database environment so it can be used to execute activities from subsequent stages. This is why we must define an additional activity to be considered for our tool related to this stage. We call this activity *Importing data*, where we import the extracted event data into the graph database, producing the appropriate inputs for our tool to execute the activities from the upcoming stages.

Third Stage - Data Processing In the third stage, *Data Processing*, event logs must be created from the event data in order to provide the necessary inputs for the mining and analysis stage. The four activities executed during this stage are *Creating views* or event logs by defining event classes and process instances to provide different perspectives of the data, *Aggregating events* to reduce the complexity of the event data, *Enriching logs* by adding additional attributes to the logs, and *Filtering logs* by removing events or partitioning the event log. These four activities, while they can be executed before we start using a process mining tool, we can also make the argument that they can be executed inside it, which we should discuss more in depth.

The data processing stage must ensure that the data is suitable for the analysis that will be performed later on, and even though it can be argued that these activities do not require a process mining-specific tool to execute

them (e.g. running a Python script that generates an event log from the data or filters events based on their attributes), tools such as ProM have shown that there is a benefit in performing them within a dedicated process mining tool, since it provides the right context to define events and traces to allow its users to perform filters or aggregations on the data more intuitively. This is especially true considering the data model proposed in [1], where Entity and Class nodes can be defined to generate different views of the event data, enriching the log at the same time with new relations between Event nodes. Thus, a process mining tool based on this graph data model should be capable of taking advantage of the schema to create and enrich logs while also allowing its users to filter the log or aggregate events in a subsequent step.

Another thing to note related to the processing stage is found on a recent case study on process mining [4], which mentions that the processing stage may also address bringing the data into the right format, such as updating the timestamps so they can be correctly interpreted by the tool, therefore, it is important to clearly define the type of inputs accepted by the tool (e.g. file types, content format) so the processing stage can take this into consideration and the tool can be used successfully.

Fourth Stage - Mining and Analysis The objective of the fourth stage, *Mining and Analysis*, is to apply process mining techniques on event logs in an attempt to answer the research questions. The four types of activities included in this stage are *Process Discovery*, where techniques are used to obtain a process model, *Conformance Checking* to detect inconsistencies between a process model and its corresponding event log, *Enhancement*, which refers to extending, improving or repairing a process model based on information about the actual process, and *Process Analytics*, where data mining techniques or visual analytics can be applied to improve the process models. Once again, these four activities can be executed with the help of a process mining tool and should be discussed further.

We can say that any tool with process mining capabilities should be able to execute at least one of these activities, something that becomes apparent by looking at the similarities between these four activities and those mentioned in the Process Mining Manifesto, which include process discovery, conformance checking, deviation monitoring, model extension and model repair among others [3]. Therefore, a tool that plans to build around the data model presented in [1] must be able to adapt or create process mining techniques that execute the activities from the fourth stage in a manner suitable for the current setting.

Fifth Stage - Evaluation Then, the fifth stage, *Evaluation*, refers to the association of the findings from the previous stage with improvements that address the initial objectives. The two activities for this stage are *Diagnose*, which refers to the understanding of the process model discovered, highlighting the results that deviate from the expectations, and *Verify and Validate*, where the findings are compared against the original data to identify improvements. While the second activity requires direct interaction between the stakeholders to evaluate the findings, the diagnosis can be more effective by using a tool that provides an optimal interpretation of the results.

The evaluation and correct understanding of the results depend to some extent on the effectiveness with which results are presented. The Process Mining Manifesto acknowledges the importance of working with process mining tools that provide a good representation of the results by mentioning three current process mining challenges related to this topic [3]:

- Challenge C9. This challenge mentions the need to combine process mining with visual analytics for a better understanding of large and complex datasets.
- Challenge C10. This challenge mentions that it is important to consider the creation of user-friendly interfaces and the linking of event data with process models to provide valuable interactions with end-users
- Challenge C11. This challenge mentions that results should be presented using a suitable representation.

Therefore, a process mining tool working on top of a graph database must also consider a way to extract the data from events and models, stored in the form of nodes and edges, and present the results in a suitable manner.

Sixth Stage - Process Improvement and Support Finally, the objective of the sixth stage, *Process Improvement and Support*, is to use the insights obtained to define actions that modify the current process. The two activities executed in this stage are *Implementing improvements* based on the results of the project, and *Supporting operations*, where, for structured processes, results can be used to detect problematic cases or suggest corrective actions. While the results obtained from the process mining tools are the inputs for this stage, no further interaction with the tools is needed at this stage.

After analyzing the 6 stages proposed by the PM^2 methodology, we identified 10 activities that can be executed with the help of a process mining tool. These activities are marked in boldface in Table 3.1, where we also specify

	Stage	Activity	Inputs	Outputs	Included?	Layer
2	Extraction	Determining the scope	Research questions and Information Systems	Event Data	-	-
		Extracting event data			-	-
		Transferring process knowledge			-	-
		Importing data*			Yes	1
3	Data Processing	**Creating views**	Event Data	Event Logs	Yes	2
		Aggregating events			No	-
		Enriching logs			Yes	2
		Filtering logs	Event Data Process Models		No	-
4	Mining and Analysis	**Process discovery**	Event Logs	Process Model	Yes	4
		Enhancement	Event Logs Process Models		No	-
		Conformance checking		Diagnostics	No	-
		Process analytics		Data Mining / Visual Analytics Results	No	-
5	Evaluation	**Diagnose**	Process Models	Improvement Ideas	Yes	5
		Verify and Validate	Performance and Compliance Findings	New research questions	-	-

Table 3.1: PM^2 methodology activities. Activities in boldface are those that can be executed with the help from a process mining tool. *The *Importing Data* activity was added by us considering that data has to be adapted to execute subsequent activities on top of a graph database.

their corresponding inputs and outputs (according to the methodology) in the fourth and fifth columns. The next step is to define an architecture for the tool that is able to address these activities.

3.2 Defining the Tool Architecture

After identifying the activities that can be executed with a process mining tool, in this section we define an architecture for the tool, providing a guideline for its implementation that specifies when these activities are executed. We define the architecture based on the activities defined in Table 3.1, describing how their inputs and outputs are correlated and how they translate into the graph database environment.

By analysing the stages of the PM^2 methodology that contain the activities identified in the previous section, together with the interaction with ProM, it was possible to define an architecture for the tool that would ensure that each activity was implemented and executed in a logical order, providing the appropriate inputs for subsequent activities. If we look at the inputs and outputs of each activity defined in Table 3.1, we can describe two examples of how the architecture was defined: 1) The second activity identified was *Creating views*, where Entities and Class nodes are defined to create differ-

ent connections in the log, but to do this, first it is necessary to import the event data into the graph database through the *Importing data* activity so the Event nodes that compose the log can be used as the inputs to create views, and 2) To perform the *Diagnose* activity, we must have previously generated process models that can be visualized to understand and analyze the results. Following a similar thought process for all the activities resulted in the model shown in Figure 3.1.

The model consists of 5 layers: (1) Event Layer, (2) Entity and Behavior Layer, (3) View Layer, (4) Model Layer, and (5) Model Analysis Layer. These layers help us group the different types of nodes and relations that exist in the database. In turn, these different types of nodes represent the inputs and outputs of the 10 distinct process mining activities we identified towards the end of Section 3.1. Next, we describe the objectives and activities addressed by each layer.

Event Layer The *Event Layer* addresses the *Importing data* activity through the upload of the extracted data into the graph database, transforming the isolated events into Event nodes, with the event attributes stored as node properties. The input for this layer is the extracted data from relevant information systems and the output is a set of Event nodes imported into the database representing the event data.

Entity and Behavior Layer The *Entity and Behavior Layer* addresses the *Creating views* and *Enriching logs* activities, where functionalities that create different views of the data and add information to the log can be executed. For the graph data model, this is done by defining the entity identifiers, which provide a temporal ordering for the events, and the event classes, which describe the aggregated behavior per entity type. The input for this layer is the event data in the form of the isolated Event nodes and the outputs are the Event, Entity and Class nodes together with the relations that define their correlations, all of which represent our version of the views and event logs.

View Layer The *View Layer* addresses the *Aggregating events* and *Filtering logs* activities, where functionalities that create data subsets to reduce the complexity of the event log can be executed. The input for this layer is the connected Event nodes together with the Entity and Class nodes and the output is the subset of nodes that compose the processed log.

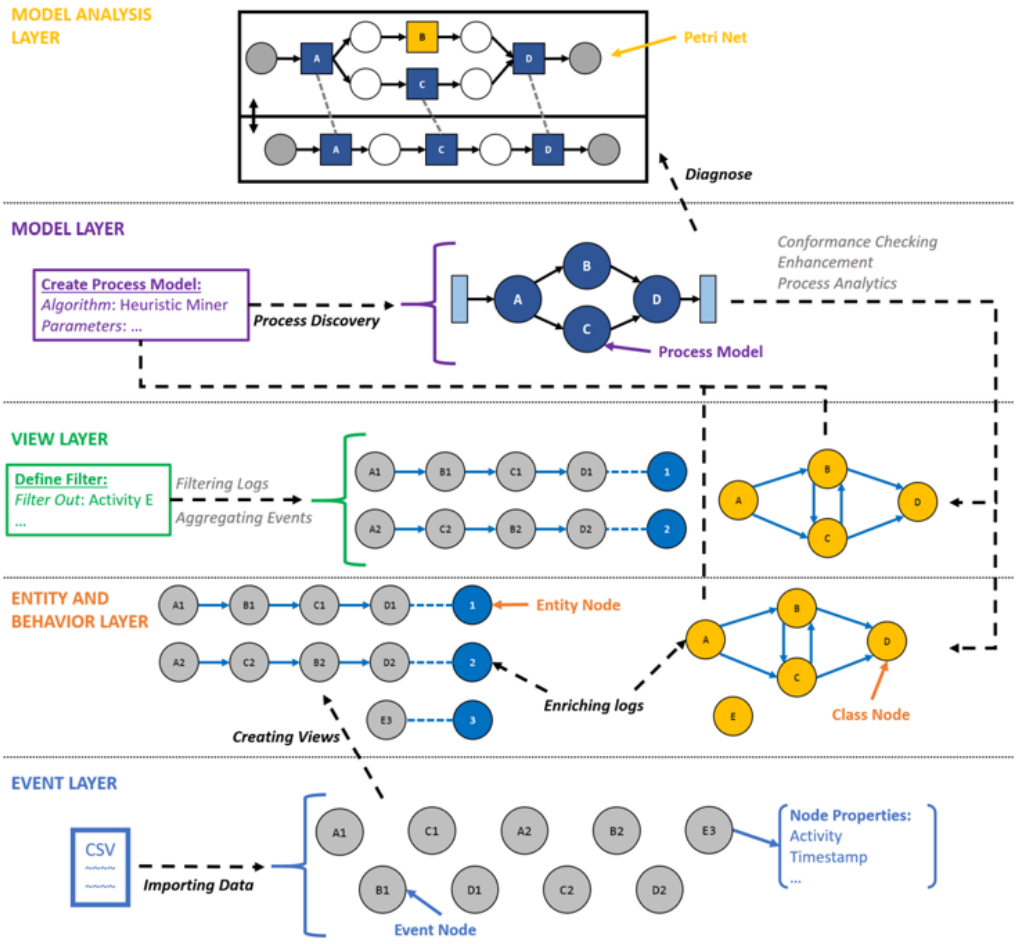


Figure 3.1: 5-layer architecture used for the implementation of the tool. From the 10 activities identified in Section 3.1 (marked in *italics*), 5 of them (marked also in **boldface**) are addressed by our implementation.

Model Layer The fourth layer, *Model Layer*, addresses the *Process discovery*, *Conformance checking*, *Enhancement*, and *Process analytics* activities, where the process mining algorithms or other data mining techniques can be executed. For this layer in the graph data model, the input is either the set of correlated Event, Entity and Class nodes conforming the event log to be analyzed or an already existing model on which algorithms are executed, and the outputs are either new process models or diagnostic information about the existing models.

Model Analysis Layer Then, the fifth layer, *Model Analysis Layer*, addresses the *Diagnose* activity, where those functionalities that help in understanding the process models can be executed. The inputs for this layer are the database elements containing the information about the process models and the output is a visual representation of those elements.

Now that the architecture has been defined, we describe our approach to delimit the scope of our implementation.

3.3 Delimiting the Project Scope

Once the architecture has been established, the next step is to delimit the scope of the tool. In this section, we describe the scope of our implementation by identifying the most essential activities to successfully use the tool for process mining. To do this, we analyze if any of the 10 activities originally established can be considered as "non-essential", keeping only those activities whose execution provide the minimum viable working tool that can provide relevant results for a process mining project.

The first pair of activities that can be discarded are the *Aggregating events* and *Filtering logs* activities. Although reducing the complexity of the logs can help to remove uncommon events or traces and focus the analysis on specific parts of the data, the process mining algorithm can still be executed on a full event log. Therefore, these two activities, composing in full the third layer, are non-essential for the implementation.

Then, we can look at the types of process mining mentioned in the Process Mining Manifesto to discard activities from the fourth layer and further delimit the scope of the tool. First, the Manifesto mentions three basic types of process mining: discovery, conformance checking, and enhancement [3], which directly correlate to three of the four activities addressed in the fourth layer. Given that the *Process analytics* activity is not mentioned as a basic type of process mining, it can be discarded from the implementation. Then, from the three types of process mining, both the conformance checking and

enhancement require an existing model as an input, so it is important to first define how a process model can be discovered using the graph data model before working with those two activities. Considering also the fact that the Manifesto mentions that process discovery is "the most prominent process mining technique" [3], we need to include this activity to show that it is possible to execute process mining with our tool.

The results from this analysis are displayed in the last two columns of Table 3.1, where we specify which activities were included and which layer from the architecture defined in Section 3.2 addresses them. In the end, we considered five activities as "non-essential" to obtain the minimum viable working tool that can be built around the data model from [1], with only the third layer from the model being fully excluded. These activities are displayed in *italics and grey font* in Figure 3.1. The five remaining activities considered for our implementation are: *Importing Data*, addressed in the first layer, *Creating views* and *Enriching logs*, addressed in the second layer, *Process discovery*, addressed in the fourth layer, and finally *Diagnose*, addressed in the fifth layer. These activities are displayed in ***italics and black font*** in Figure 3.1.

The details on the implementation of these 4 layers are presented in Chapters 4, 5, and 6.

Chapter 4

Creating the User Interface for the Event Graph Data Model

In Chapter 3, we define the scope of the process mining tool under the 5-layer model. In this chapter, we discuss how the first and second layer of the architecture shown in Figure 3.1 are implemented. In Section 4.1, we present the layout for the user interface of the tool. In Section 4.2, we describe the scenarios that must be accounted for to deal with multi-dimensional data. Then, to implement the Event Layer and address the *Importing Data* activity, in Section 4.3 we describe how event data is imported into the graph database. To implement the Entity and Behavior Layer and address the *Creating views* and *Enriching logs* activities, we describe the creating of entities, directly-follows relations, entity type attributes and classes in Sections 4.4, 4.5, 4.6, and 4.7 respectively.

4.1 User Interface Layout

In this section, we describe the layout of the user interface by providing details on the purpose of each of its components.

Now that the scope of the tool has been defined, the next thing we need is to define a layout that can help us translate the architecture from Figure 3.1 into a user interface.

The layout was designed with the objective of providing the user with an intuitive look into what the tool can do and in which order it can be done, which resulted in the user interface shown in Figure 4.1. The layout of the user interface is described next based on the labels from Figure 4.1:

A. *Tool Menu.* The layout contains a menu on top, standard for many software applications, that can be used to store functions such as opening

files, interacting with the application or displaying additional details about the tool.

B. *Log Label.* The layout includes a label indicating which event log is currently selected, providing a reference for the users to know which log will be used as input or will be affected by the different functionalities.

C. *Functions Panels.* The layout also contains 7 panels on the left-hand side that allow us to group the tool functions according to their functionality. These panels are ordered in such a way that they emulate the 5-layer architecture, providing the users with a visual guide to know the order on which different functions can be executed. The 7 panels are:

   1. Logs. The *Logs* panel is meant to contain functions related to the imported event logs, directly relating it to the *Event Layer*.
   2. Graph Data. The *Graph Data* panel was placed next since users can already interact with the event logs after they are imported. This panel allows the users to obtain more information about not only the Event nodes pertaining to a particular event log, but also the Entity and Class nodes that are addressed in the next two panels.
   3. Entities. The *Entities* panel contains those functions related to the definition of entity identifiers.
   4. Classes. The *Classes* panel contains those functions related to the definition of event classes. Together with the *Entities* panel, these two panels represent the *Entity and Behavior Layer*.
   5. Filters. The *Filters* panel is meant to contain the functions related to the *View Layer*, which, as discussed earlier, was left out of the current scope, serving only as a placeholder.
   6. Algorithms. The *Algorithms* panel is meant to contain the functions related to the *Model Layer*, such as the process discovery algorithms.
   7. Models. The *Models* panel represents the *Model Analysis Layer*, so inside this panel we can find those functionalities that help in understanding the process models.

D. *Graph Panel.* The right side of the user interface is completely allocated to display graphs, which could be representing anything that is stored in the graph database, from events to models.

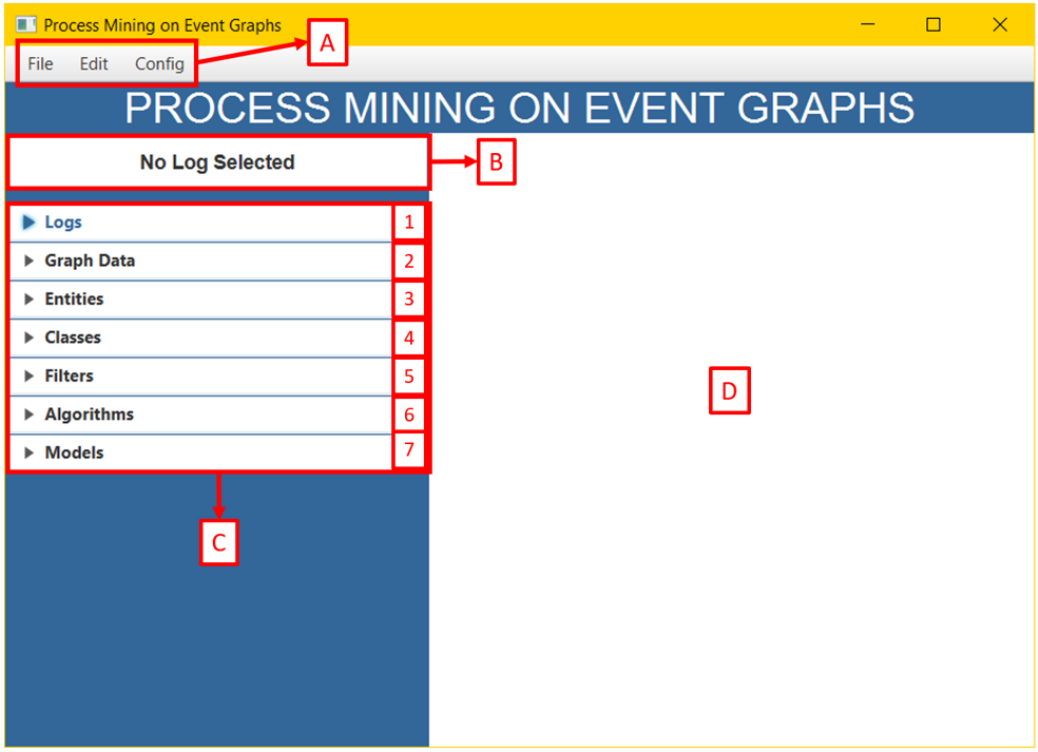


Figure 4.1: User Interface Layout. (A) Tool Menu. (B) Log Label. (C) Functions Panels. (D) Graph Panel. The tool functions are grouped by functionality in the 7 panels inside (C).

Now that the main layout of the user interface has been presented, we need to discuss the impact of potential multi-dimensionality in the extracted event data used as an input for the tool. This factor must be taken into account during the implementation of the first two layers, the *Event* and *Entity and Behavior* layers, which is why it is important to discuss them next.

4.2 Scenarios for Data Dimensionality

In this section, we discuss the scenarios that must be accounted by the tool to handle event data that contains multiple dimensions. First, we provide a brief description on how the data model proposed in [1] allows us to model multiple dimensions. Then, we describe three scenarios that change the way in which events should be correlated to an entity identifier. Finally, we describe why these scenarios must be considered by the tool based on their impact for the process mining analysis.

As mentioned in [1], a process event log is a collection of recorded events structured into a specific view on an information system from the perspective of one specific entity, and, to analyse its behavior, data must be extracted in sequential format. However, information systems can host multiple identifiable entities, and in this case the data extraction also requires *flattening* the data to correlate all events under a single entity or case identifier. This transformation of the data may lead to false behavioral information if those distinct entities are not considered.

As we discussed in 2.2.2, this issue is addressed by the graph-based data model presented in [1], which allows the definition of multiple entity identifiers and then defines directly-follows relations between events related to the same entity only, turning the data model into a multi-dimensional event log and avoiding the issues related to convergence and divergence.

Based on the flexibility provided by this graph data model to define multiple entity identifiers, which represent distinct dimensions of the event data, we identify three scenarios that change the way in which events can be correlated through directly-follows relations depending on how the data dimensionality is defined.

To explain these scenarios, we can use the event data from Table 4.1, which contains a simplified example of the events registered for an order and its delivery. Table 4.1 shows the 6 events that correlate the order 40 with deliveries 514 and 623, indicating the time at which the resources carried out the action for each event. Using this table, we can now describe the three scenarios.

Event	Order	Time	Action	Delivery
e1	50	2018-12-20 11:03	Receive Order	
e2	50	2018-12-23 11:11	Pack Order	
e3	50	2018-12-23 11:46	Add Item	514
e4	50	2018-12-23 11:49	Ship Parcel	514
e5	50	2018-12-23 11:51	Add Item	623
e6	50	2018-12-23 11:59	Ship Parcel	623

Table 4.1: Event Data. Simplified events of an order.

S1. Event data describes a single dimension Even when the data model from [1] allows us to work with multiple dimensions, it is still possible that we need to work on an event log that describes the events based on a single entity. In this scenario, once we identify the column that defines the entity, we can make the directly-follows connection between the events related to that entity. The top section of Figure 4.2 shows how the events would be connected if the "Order" column in Table 4.1 is identified as the only entity in the event log. To represent the actions, we use the first letter of each word (e.g. "Receive Order" is represented as "RO" in the figure). As we can see, all the events are connected between them since the event data indicates that they belong to the same order (order 50).

S2. Event data describes multiple dimensions In this second scenario, we take advantage of the flexibility provided by the graph data model of [1] to define multiple entity identifiers in the event data. The middle section of Figure 4.2 shows how the events would be connected if we defined "Delivery" as a second entity after the "Order" entity. In this case, we can see how the directly-follows paths for deliveries 514 (events *e3* and *e4*) and 623 (events *e5* and *e6*) cross paths with the path from order 50.

S3. Event data describes multiple *independent* dimensions In this third scenario, we have events that belong to one dimension but were correlated to a second one during the data extraction to obtain a single event log from multiple sources. The event data from Table 4.1 may also represent this scenario if the data from orders and deliveries was *flattened* under the "Order" entity to obtain the event data. We can see how the events should be connected in this scenario in the bottom section of Figure 4.2, where the connection with order 50 does not include events *e3* until *e6* since, for these events, the order attribute only represents the correlation with the orders,

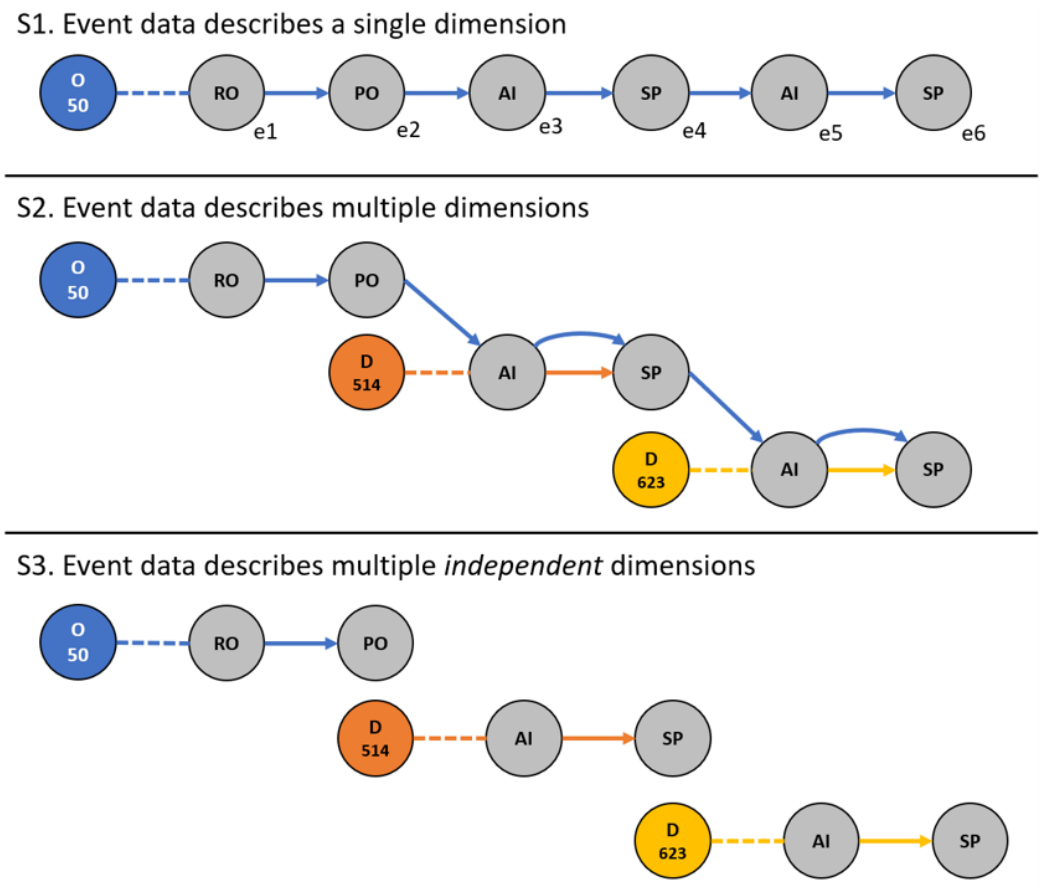


Figure 4.2: Distinct scenarios (S1, S2, and S3) where the connections between events change depending on the dimensionality of the event data.

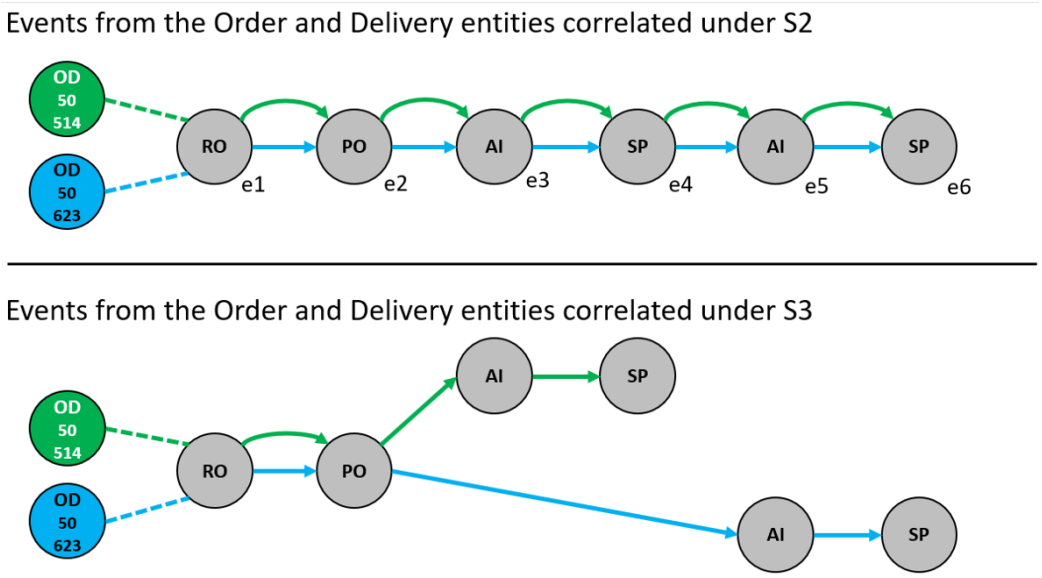


Figure 4.3: Events from the Order and Delivery entities correlated under scenarios S2 and S3.

but the "Delivery" dimension is independent from the "Order" dimension.

The difference between scenarios S2 and S3 becomes more relevant when we start to correlate events from multiple entities, something that the graph data model from [1] also allows. We can see the difference in the connections in Figure 4.3.

If we correlate the events from the "Order" and "Delivery" dimensions considering scenario S2, as shown in the top section of Figure 4.3, any event that is part of either order 50 or delivery 514 will be part of a path, and any event that is part of either order 50 or delivery 623 will be part of a different path. These connections imply that, for any given order and its corresponding delivery, a "Ship Parcel" activity can be followed by an "Add Item" activity, as exemplified by the connection between events *e4* and *e5*.

In contrast, under scenario S3, as shown in the bottom section of Figure 4.3, only the events that are exclusively from order 50 and the events that are exclusively from delivery 514 will be part of a path, and only the events that are exclusively from order 50 and the events that are exclusively from delivery 623 will be part of another path. These connections imply that, for any given order and its corresponding deliveries, the "Ship Parcel" activity marks the end of the path, as exemplified by the events *e4* and *e6*.

As we can see, depending on the context, choosing the wrong scenario for a particular event log could create false connections between the events, leading

to a different interpretation of the results and changing the analysis made on the process being evaluated. This presents relevant considerations for the implementation of the tool, which must be able to handle the 3 scenarios to allow its users to make their analysis under the correct assumptions regarding data dimensionality.

Now that we have seen how the definition of dimensions in the event data may impact the process mining results, we can continue with the description of the tool functionalities that were implemented, starting with the data import in the *Event Layer*.

4.3 Importing Event Data

In this section, we describe how the Event Layer was implemented. First, we describe how the event data is represented in the graph data model from [1]. Then, we discuss the considerations that need to be accounted for to implement the Event Layer in our tool. Finally, we describe its implementation, where we provide details on how the event data is imported to the database through our user interface.

4.3.1 Defining the Event Data

The objective of the *Event Layer* in the architecture we defined in Section 3.2 is to address the *Importing data* activity, where we must import the extracted event data into the database. In other words, our implementation for this layer must ensure that the source event data is in the correct format for process mining on top of a graph database, allowing us to use it to execute the subsequent activities. Therefore, we should start by defining how the extracted event data is represented in this context.

The extracted event data is the only input for the Event layer, which we assume is contained in a CSV file in the form of an event table, where each row describes one event. Table 4.2 shows an example of the event data. This data in particular consists of the events registered for four different orders, stored under a CSV file named "Orders.csv".

In [1], event data such as the one from Table 4.2 is represented in the graph database through the Event nodes. Each Event node represents a distinct event from the source data, and contains two mandatory properties, the *activity* and the *timestamp*. The additional event attributes can also be modeled as additional node properties.

To identify which events belong to which log and allow us to store event data from multiple logs in the database, a second type of node is introduced

Event	Order	Time	Action	Life-cycle	User	Delivery	Type
1	40	2018-12-20 11:02	Receive Order	Complete	System		Phone
2	40	2018-12-20 12:09	Pack Order	Start	Aaron		Phone
3	40	2018-12-20 12:10	Add Item	Complete	Bob	432	Phone
4	40	2018-12-20 12:10	Add Item	Complete	Bob	432	Phone
5	40	2018-12-20 12:15	Ship Parcel	Complete	Cooper	432	Phone
6	40	2018-12-20 16:00	Pack Order	Complete	Aaron		Phone
7	40	2018-12-22 07:23	Receive Payment	Complete	System		Phone
8	40	2018-12-22 07:24	Archive	Complete	Matthew		Phone
9	50	2018-12-20 11:03	Receive Order	Complete	Sean		Phone
10	50	2018-12-23 23:11	Pack Order	Start	Bob		Phone
11	50	2018-12-23 23:46	Add Item	Complete	Bob	514	Phone
12	50	2018-12-23 23:49	Ship Parcel	Complete	Sean	514	Phone
13	50	2018-12-23 23:51	Add Item	Complete	Bob	623	Phone
14	50	2018-12-23 23:59	Ship Parcel	Complete	Bob	623	Phone
15	50	2018-12-27 09:01	Pack Order	Complete	Aaron		Phone
16	50	2018-12-27 09:02	Archive	Complete	Aaron		Phone
17	60	2018-12-20 11:04	Receive Order	Complete	System		Online
18	60	2018-12-20 11:12	Pack Order	Start	Bob		Online
19	60	2018-12-21 09:17	Add Item	Complete	System	623	Online
20	60	2018-12-21 09:23	Pack Order	Complete	Aaron		Online
21	60	2018-12-21 09:28	Receive Payment	Complete	Cooper		Online
22	60	2018-12-21 10:15	Archive	Complete	Matthew		Online
23	70	2018-12-20 11:05	Receive Order	Complete	Cooper		Online
24	70	2018-12-20 14:05	Pack Order	Start	Aaron		Online
25	70	2018-12-20 14:08	Add Item	Complete	Aaron	775	Online
26	70	2018-12-22 08:07	Add Item	Complete	Matthew	775	Online
27	70	2018-12-22 09:01	Ship Parcel	Complete	Matthew	775	Online
28	70	2018-12-22 09:05	Pack Order	Complete	Aaron		Online
29	70	2018-12-22 09:36	Receive Payment	Complete	Aaron		Online
30	70	2018-12-22 10:15	Archive	Complete	Cooper		Online

Table 4.2: Event Data

in [1] in the form of Log nodes. Each Log node represents a distinct event log imported into the database and is linked to its corresponding Event nodes through the :HAS relationship. The only property assigned for this node is the *ID*.

There are several queries already defined in [1] to import the event data into the graph database. First, to import the events, they defined a query to import each row as an Event node and set each event attribute as a node property. The name of these properties is defined based on the column name, which is assumed to be specified in the first row of the CSV file. Then, they defined a query to create the Log node, defining its ID based on the LogID property of the recently created Event nodes. Finally, they created a query to connect each event with the Log node through the :HAS relation. The nodes that should be connected are identified based on the match between LogID property of the Event nodes and the ID property of the Log node.

To implement the Event Layer and address the *Data Import* activity, we must be able to execute the import of the event data into the graph database.

4.3.2 Data Import Considerations

While the queries provide a way for us to import the event data into the database, we still need to define how these queries can be executed by users through the user interface of our tool.

However, defining a functionality in the user interface for the data import is not enough to consider the implementation complete. First, we must implement a functionality to let the users define which of the three scenarios defined in Section 4.2 will be used to handle multiple dimensions on the event log. Handling the dimensionality at a later stage could lead to inconsistencies in the directly-follows relations between events (e.g. an entity identifier is defined considering the log as one-dimensional, but then the log is redefined as multi-dimensional, making the previous connections obsolete), so we need to consider this from the start. Then, the user interface itself requires additional functionalities to provide feedback to the users on the status of the database, especially since we do not expect users to interact directly with the database through Cypher queries. Therefore, we also need to consider that the user interface should display the information from the database in a suitable manner, which might also result in changes to the original queries. For example, we know that one of the mandatory properties in an Event node is the activity, so we must find a way to let the user select which attribute from the original event data is the activity; then, in order to allow the user to recall which attribute was selected during the import, we must store this selection somehow in the database, potentially altering the queries

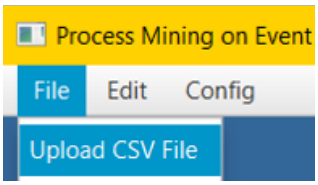


Figure 4.4: Menu option to upload a CSV file containing the event data.

that create the Event and Log nodes.

Additionally, there are two more considerations that should be addressed by the tool in the Event Layer. The first consideration comes from the need to prevent the creation of false connections between nodes in the database, while the second consideration comes from the need to be able to delete the event data, which can be required for a different number of reasons, such as liberating space from the database or redefining the properties of a log.

In summary, to complete the implementation of the Event layer and address the *Importing Data* activity, we must define functionalities in the tool that allow us, through the user interface, to import the data, define the dimensionality, provide feedback on the information stored in the database, prevent the creation of false connections between nodes, and deleting the data.

4.3.3 Implementing the Data Import

To import the data through the user interface, we need to start by defining a way to allow the user to select the file containing the event data that will be imported into the graph database.

To allow the user to select the file, we created a menu option in the user interface, as seen in Figure 4.4, allowing the user to select the CSV file containing the event data (Figure 4.5). The "Orders.csv" file shown in the figure contains the event data from Table 4.2. This event data will be used as our running example to describe all the functionalities of the tool.

Once the CSV file has been selected, we look into the data to find which column describes the timestamp of the events, which is one of the parameters needed for the query that imports the events. We assume that this column has one of the following names (ignoring case): "Timestamp", "Time", "Start", or "End".

Then, to have the users identify the Activity attribute, which is the second parameter needed for the import query, we cannot use the same approach as the one used for the Timestamp, since the possible names that the Activity attribute can be assigned in the event data are greater, so instead of

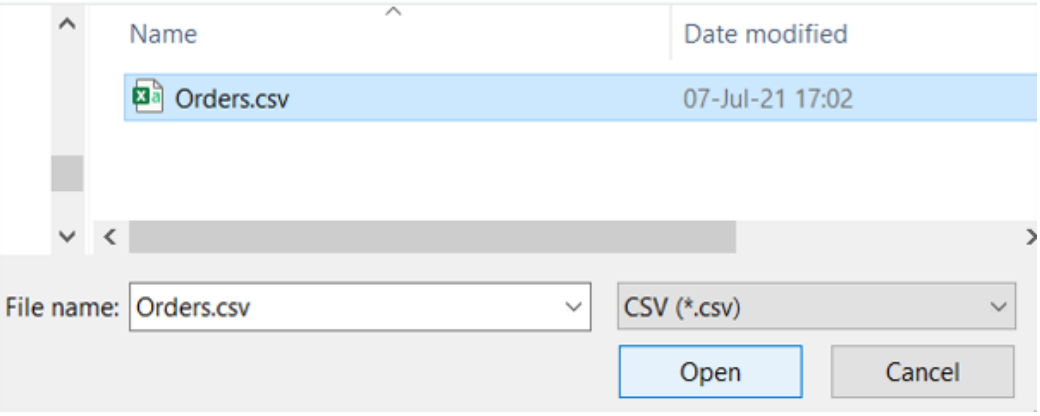


Figure 4.5: After selecting the *Upload a CSV File* option, a window shows up for the user to select the file.

configuring the tool to expect a column with a specific name, we decided to allow the user to select the attribute that represents the Activity of the event in a subsequent window, as shown in Figure 4.6. For our running example, we select the "Action" attribute as the Activity attribute. We can also see at the top of the figure that the "Time" column has already been identified as the Timestamp attribute of the event data.

Then, we decided to let users choose which event attributes to upload so they can avoid loading unnecessary data into the database. To do this, a subsequent window appears with the remaining attributes, allowing the user to select 0 or more attributes to include as additional node properties during the import, as shown in Figure 4.7. For our running example, we select the remaining 6 attributes to be included as properties in the Event nodes.

Then, to allow users to define the dimensionality of the data through the user interface, we included two buttons at the bottom of the window shown in Figure 4.7. Clicking on the "Finish" button on this window will run the import queries considering that the data does not have independent dimensions, thus, indicating that the events will be connected considering the scenarios S1 and S2 defined in Section 4.2. Clicking on the "Next" button will display a new window, shown in Figure 4.8, where users can provide the details on the independent dimensions of the event log.

The window shown in Figure 4.8 lets the user specify the independent dimensions for the event data, indicating that the events will be connected considering the scenario S3. An example was added at the top to let the users know what the practical effect is of choosing the independent dimensions. From the list on the left, users can choose which attributes, from those already selected in the previous window (Figure 4.7), should be considered

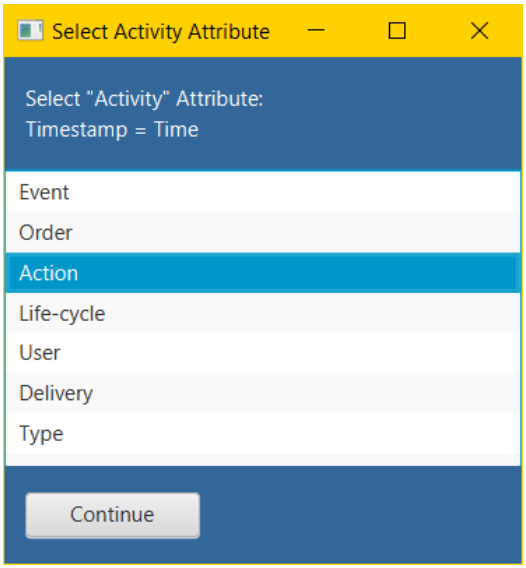


Figure 4.6: After selecting the file, the user can select which of the attributes will be stored as the event's Activity.

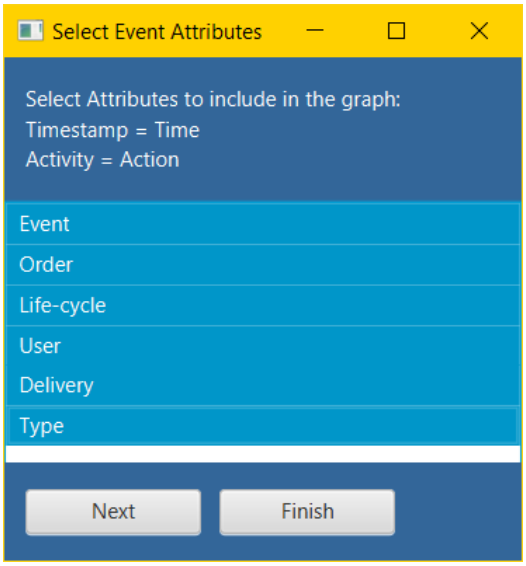


Figure 4.7: During the import, the user can click "Next" after selecting the event attributes to define the independent dimensions of the data or "Finish" to import the data with no independent dimensions.

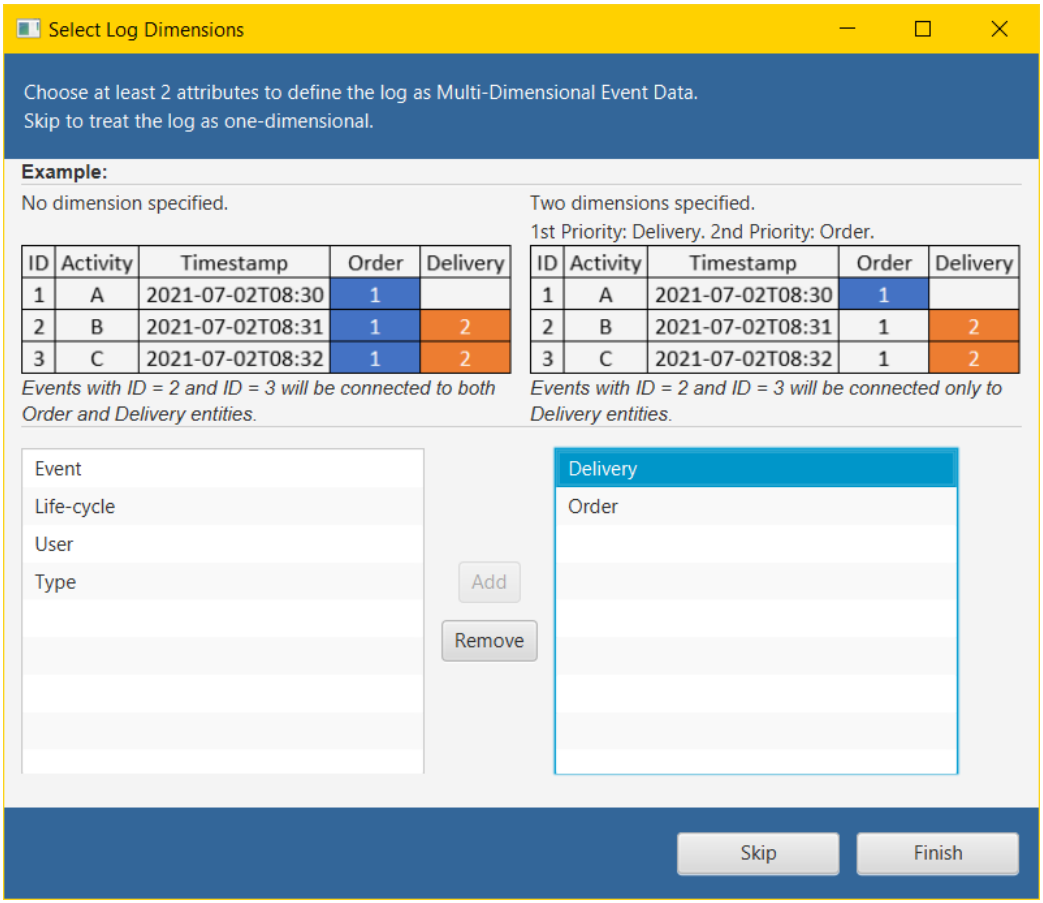


Figure 4.8: During the import, the user can select the independent dimensions to consider for the event log.

as independent dimensions for the event log. The order in which the attributes are added as dimensions is important, since those on top will have priority over those at the bottom when the tool needs to decide the entity to which events will be associated with. Clicking on the "Skip" button on this window will have the same effect as clicking on the "Finish" button in the previous window (Figure 4.7), but clicking on the "Finish" button here will run the import queries considering that the data has at least 2 independent dimensions.

For our running example, we specify the "Order" and the "Delivery" attributes as independent dimensions, which means that when either of these attributes is involved in the connection of events, the connection will be done considering the scenario S3, but for the other attributes, such as "User", the events will still be connected considering the scenario S2. Note that in Figure 4.8 the "Delivery" attribute is placed first so the events that have attributes set for both "Order" and "Delivery" (such as events 3-5 in Table 4.2) are considered as part of the "Delivery" dimension.

Now that we have defined how the query parameters (log, timestamp, activity, event attributes, and the optional independent dimensions) are selected by the user through the user interface, we can take a closer look at how the queries from the original paper were adapted in our implementation.

The query that imports the data is shown below. We added the *PERIODIC COMMIT* to prevent the query from failing due to memory constraints in case the CSV file contains large amounts of data, and we defined the Log ID as the file name to prevent the need of an additional selection from the user to import the data.

```
1 USING PERIODIC COMMIT
2 LOAD CSV WITH HEADERS FROM "file:///filename.csv" AS line
3 CREATE (:Event {Log:"filename.csv", Timestamp:datetime(line.
    timestamp), Activity:line.activity, Attribute1:line.
    Attribute1,..., AttributeN:line.AttributeN})
```

The query that creates the Log node is shown below. Again, we use the CSV file name to define the ID property. To keep a record of the selections made by the user, we defined four additional properties, the "timestampCol", indicating the name of the column identified as the timestamp attribute, the "activityCol", indicating the name of the column selected as the activity attribute, the "attributesSelected", indicating the additional attributes selected by the user, and "Dimensions", indicating the independent dimensions specified by the user (only if they were specified, otherwise, the property is not defined). In our running example, the value for the "attributesSelected" property is "Event|Order|Life-cycle|User|Delivery|Type", while the value for the "Dimensions" property is "Delivery|Order".

```
CREATE (:Log {ID:"filename.csv", timestampCol:"
    timestampColumnName", activityCol:"activityColumnName",
    attributesSelected:"Attribute1|...|AttributeN", dimensions:"
    Dimension1|...|DimensionN"})
```

Finally, the query shown below correlates the Event nodes with the Log node through the :HAS relationships.

```
MATCH (e:Event {Log:"filename.csv"})
MATCH (l:Log {ID:"filename.csv"})
CREATE (l)-[:HAS]->(e)
```

Once these queries are executed, the event data is stored in the graph database as Event and Log nodes, completing the main objective of the *Event Layer*. However, we still need to describe how we can visualize the event data recently imported to provide feedback to the users on the status of the database, how we prevent the creation of false connections between nodes, and how we can delete the data through the user interface. The way in which these three functionalities are addressed is described next.

Visualizing the Event Data

In order to provide the users with feedback from the tool and show them data that currently exists on the database, it became important to implement some sort of visual aid in the user interface to display which logs and events have already been imported.

Visualizing Logs The users can visualize the details of the existing event logs in the database through the Logs panel, as shown in Figure 4.9.

The first table in the panel shows a list of the logs currently on the database, obtained by querying over the existing Log nodes. This allows the user to check which data is already available to work with. The second table, which is populated after clicking on the "View Log Details" button, displays additional details of the log, allowing the user to review which attributes were included during the import, helping them check if the current version of the imported log contains the information they require.

Visualizing Events Then, users can visualize the event data pertaining to an imported log by expanding the Graph Data panel, as shown in Figure 4.10.

The Log label lets the users and the tool know which log is being used as an input to make the queries to the database and retrieve the information. To visualize the Event nodes on the Graph panel, we decided to implement the "View" button next to the *Instance Level* label, which runs a query to

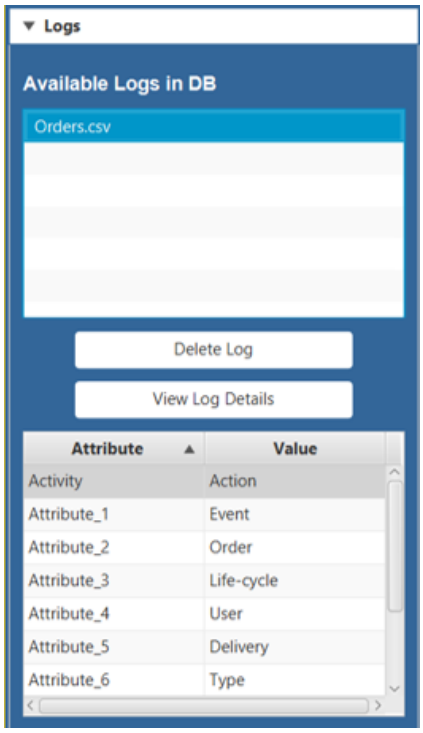


Figure 4.9: Logs panel showing the logs imported into the database and its details.

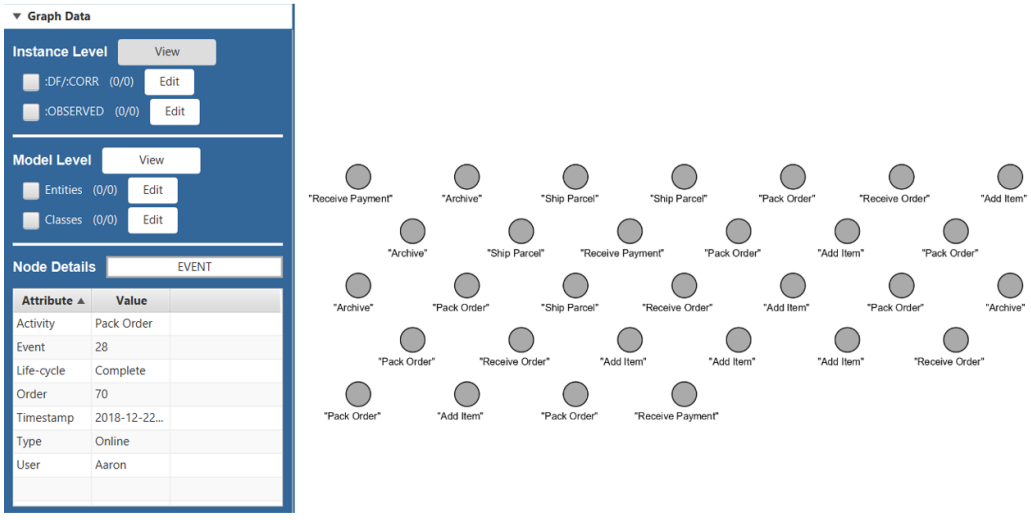


Figure 4.10: Visualizing the graph data. Users can interact with the Graph Data panel to display nodes in the Graph panel and see the node properties on the table at the bottom. The Log label shows the log currently selected.

retrieve the Event nodes from the database and display them. In addition, clicking on a node on the Graph panel will populate the table at the bottom of the "Graph Data" panel with all of the existing properties of the selected node. In Figure 4.10, we can see the 30 Event nodes corresponding to the 30 events from Table 4.2, with the table in the Graph Data panel displaying the properties stored inside the Event node corresponding to event 28.

The functionality of the buttons referring to the Entity and Class nodes and their respective relations is discussed in sections 4.4, 4.5, and 4.7.

Preventing false connections between Event Nodes

Now that it is possible to import data into the database through the user interface, it is important to make sure queries are reliable, matching exclusively the nodes belonging to the correct log. At first glance, it might seem like this has already been taken care of with the :HAS relation, connecting the Event nodes to exactly one Log node, a constraint that is mentioned in section 4.4 of [1]. However, there is a case where this would not hold, caused by the potential import of new event data under the same Log ID.

As stated earlier, the ID of the log is determined by the CSV file name, and the query that creates the relation between Events and its Log uses that condition to identify the nodes to link. If the user tries to upload the same event log twice, or if a different event log is imported using the same file name, the query will basically return a cross product between the two Log nodes and the Event nodes belonging to both logs files, creating false connections between the data.

To prevent this, a new Cypher query was written to define a constraint in the ID attribute of the Log node, preventing the creation of a second Log node with the same ID value, thus preventing the issue.

```
CREATE CONSTRAINT UniqueLogs
IF NOT EXISTS ON (l:Log)
ASSERT l.ID IS UNIQUE;
```

In addition, as a design decision, we decided to allow the user to toggle the activation of this constraint.

```
DROP CONSTRAINT UniqueLogs IF EXISTS;
```

Regarding the user interface, users can interact with this functionality through the menu bar on top of the application, as shown in Figure 4.11.

Deleting Event Data

Finally, we considered it was necessary to provide a way inside the user interface for the users to delete the existing data. Two distinct functionalities

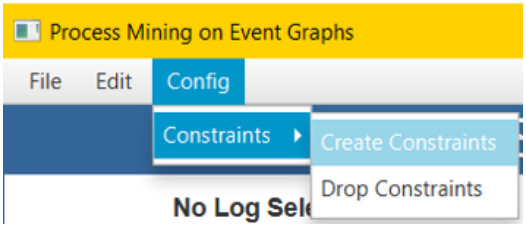


Figure 4.11: Database constraints can be enabled or disabled by the user at any point.

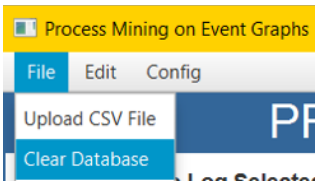


Figure 4.12: Users are able to clear the database by clicking a button on the user interface.

were implemented for this purpose, the first one allows the user to clear the full database, removing all existing nodes and relations, and the second one allows the users to delete a specific event log that may no longer be needed.

Clearing the Database To clear all the existing nodes from the database, we defined a query that deleted the nodes with periodic executions to avoid memory errors during the process.

```
CALL apoc.periodic.iterate(
  "MATCH (n) RETURN n",
  "DETACH DELETE n",
  {batchSize:500})
YIELD batches, total
RETURN batches, total
```

To allow the users to interact with this functionality, we added a menu option in the tool, as shown in Figure 4.12.

Deleting a Log To delete a log from the database, we first defined a query that found and deleted all the Event nodes linked to the Log currently selected by the user. Then, we defined a second query that deletes the Log node itself.

```
CALL apoc.periodic.iterate(\n" +
  "MATCH (l:Log{ID:'filename.csv'})-[:L_E]->(e) RETURN e",
  "DETACH DELETE e",
    {batchSize:500})
```

```
YIELD batches, total
RETURN batches, total

MATCH (l:Log{ID:'Orders.csv'})
DETACH DELETE l
```

To allow the users to interact with this functionality, we added a button called "Delete Log" in the "Logs" panel, which can be observed in Figure 4.9.

Now that we have described all functionalities from the Event layer, we can look at the first functionality implemented for the Entity and Behavior Layer, the creation of entity identifiers.

4.4 Creating Entities

In this section, we start describing how the Entity and Behavior Layer was implemented by focusing on how the tool addresses the *Enriching logs* activity through the creation of the entity identifiers. First, we describe how the entity identifiers were defined in [1]. Then, we describe the need to adapt the queries presented in [1] for our tool. Finally, we describe how we use these queries to define the entity identifiers through the user interface.

4.4.1 Defining the Entities

Once the Event Layer has been implemented, addressing the *Importing Data* activity and other functionalities required for that layer, we can move on to the implementation of the second layer, the Entity and Behavior Layer, which addresses the *Creating views* and *Enriching logs* activities by defining entity identifiers and event classes for the event data. To do this, we must start by describing the entity identifiers and their representation in the data model from [1].

Entity identifiers are used to correlate events based on a common attribute. In one-dimensional event logs, this is commonly known as the case identifier, which allows us to identify which events belong to the same execution of the process registered in the log. In [1], these entity identifiers are implemented in the form of Entity nodes, where each node represents a specific entity of the process, such as Order 40 or Order 50 in Table 4.2.

Entity nodes have 3 properties. Property *EntityType* describes the type of the entity, property *ID* refers to the entity identifier, and property *uID* stores the combination of the EntityType and ID values to have a unique value for this Entity node in the entire graph.

To materialize the correlation between events and entities in the graph database, [1] introduces the :CORR relation, connecting Event nodes to the Entity nodes. The :CORR relations allow us to correlate any event to any number of entities of different types.

[1] also defines queries to create the Entity nodes and define the :CORR relations between them and the Event nodes. The first query creates the Entity nodes based on the set of all entities that occur in the data, assigning the name of the event property used to define the entity as the EntityType and the value of the event property as the ID, with their combination stored as the uID property. The second query correlates an entity to all the events where the Entity node ID matches with the Event node property.

In addition to the creation of entities, the data model proposed in [1] also allows us to model relations and interactions between entities. The purpose of this is to provide a way for users to analyze the interaction between events correlated to different entities, providing another way to enrich the event log.

To analyze this interaction, the data model introduces the derived entities, which represent the relationship between to entities. Similar to the regular entities, these derived entities are represented through Entity nodes, and are connected to the Event nodes through the :CORR relations. The properties of the derived Entity nodes are the same as those for the regular entities, with the difference that the values for the EntityType, ID, and uID are a combination of the properties from the Entity nodes used to create it.

The data model also defines a connection between these two nodes and the derived Entity node in the form of the :REL relation, which has the property *Type*. For the :REL connection between the two original Entity nodes, the Type property has the combined EntityType properties as its value, while the :REL connection between the original Entity nodes and the derived Entity node has "Reified" as its value.

There are 3 queries defined in [1] to create and connect the derived Entity nodes. The first query creates the :REL relation between the original entities based on the properties of the events correlated to them, if an event correlated to the first entity has a reference to the second entity in the form of a node property, then the :REL relation is created between the two Entities. The second query creates the derived Entity node, sets its properties and connects it to the corresponding Entity nodes through the :REL relations. Finally, the third query correlates the derived Entity nodes with the Event nodes through the :CORR relation.

In order for us to start with the implementation of the Entity and Behavior Layer, we must be able to create the Entity nodes as it is proposed in the data model from [1], which will help us address the *Enriching logs* activity.

4.4.2 Considerations for the Creation of Entities

The queries already defined in [1] to create and connect the entities and derived entities provide a guideline for us to generate these nodes inside our tool, helping us address the *Enriching logs* activity inside the Entity and Behavior Layer. However, there are three aspects not considered by the original queries that must be addressed in our current implementation.

The first aspect is related to the usability of the tool. We consider that the tool can be used to store multiple event logs, which are not necessarily related, so it is important that we distinguish between the nodes that belong to different logs to avoid creating false connections in the database. This is not considered in the original queries during the creation of the Entity nodes, which assume that every Event node in the database belongs to the same log and is something we need to update in the queries.

The second aspect is related to our discussion regarding independent dimensions in Section 4.2. If we work with the scenario S3, we need to consider the dimension to which an event is related to before making the connection between events and entities in order to avoid correlating events to an entity that refers to a different dimension. Therefore, we also need to include the dimensionality in the queries that create and connect the Entity nodes.

For the third aspect, also caused by the scenario S3, we must adapt the queries that create the derived entities since they do not consider that, with the independent dimensions, we could have a case where there seems to be no match between different entities. We can see an example in Table 4.2; if we consider "Delivery" and "Order" as independent dimensions, then no event related exclusively to the "Order" dimension has a value defined for the "Delivery" attribute, which could cause no derived entities to be identified by the original queries defined in [1].

In addition to these three aspects, we must also define a way to allow the users to use these queries to create the entities and visualize the results through the user interface.

4.4.3 Creating Entities through the User Interface

Creating Entities

To allow users to create Entity nodes from the user interface, we decided to add a new button to the user interface inside the *Entities* panel called "New Entity", which can be observed in Figure 4.13.

Clicking on the "New Entity" button will show a new window displaying all the existing event attributes stored as node properties inside the Event nodes, as shown in Figure 4.14. The numbers shown in parenthesis next to

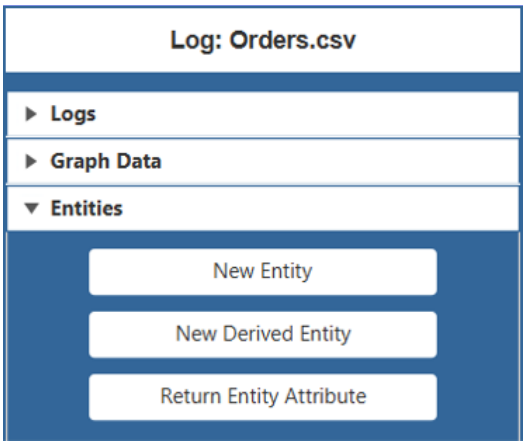


Figure 4.13: Entities panel of the user interface.

the attribute name represent the amount of distinct values for that attribute, letting the user know in advance how many Entity nodes would be created. Clicking on the "Create Entity" button will execute the Cypher queries that create the Entity nodes and the :CORR relations with the Event nodes.

Continuing with our running example, once the event data has been uploaded, specifying the "Order" and "Delivery" attributes as independent dimensions, we can create the Entity nodes to define the entity identifiers for our event log, which will be the "Order" and "Delivery" dimensions. First, on the window of Figure 4.14, we select the "Order" attribute and click on "Create Entity". Then, we open that window again and now select the "Delivery" attribute to create the second entity type.

The adapted queries that create and connect the Entity nodes are shown next. The first query we execute creates the Entity nodes. In line 1, we specify the log for which we will create the Entity nodes, filtering out Event nodes that belong to a different log from the start. In line 3, we can see that the events related to a dimension with a higher priority are filtered out before creating the :CORR relations. In our running example, we would see the "Delivery" dimension as the parameter inside the collection when we create the "Order" entity type. Finally, in line 6, we added an additional parameter to specify the log to which the Entity nodes belong to, since there is no direct connection between the Log and Entity nodes in the data model. This will help us retrieve them in the next query without bringing additional Entity nodes from other event logs.

```
1 MATCH (l:Log{ID:'filename.csv'})--(e:Event)
2 WITH e, keys(e) AS properties
3 WHERE NOT ANY(x IN properties WHERE x IN ["Dimension1", ..., "
    DimensionN"])
```

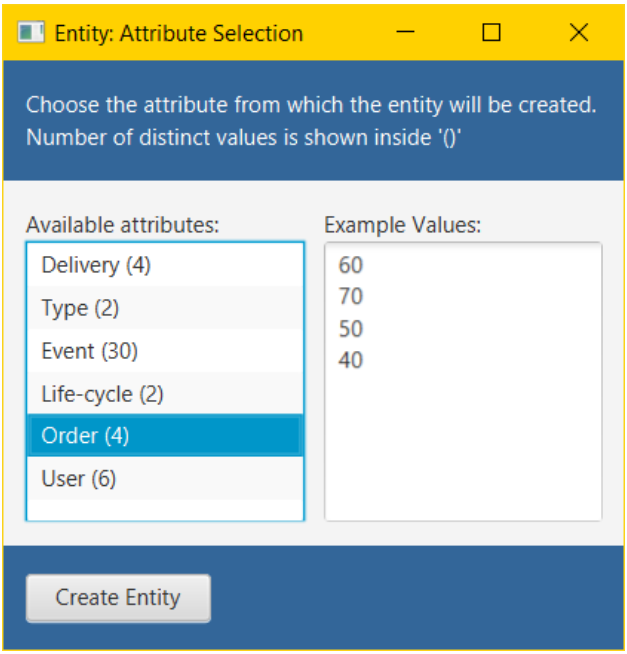


Figure 4.14: Entity selection window.

```
4 WITH e WHERE EXISTS(e.‘EntityType‘)
5 WITH DISTINCT e.‘EntityType‘ AS id
6 CREATE (en:Entity {ID:id, uID:("EntityType"+toString(id)),
    EntityType:"EntityType", log:"filename.csv"})
```

The second query correlates the Event nodes with the recently created Entity nodes. In line 1 we can see again how we match exclusively for the Events from a specific log. In line 3, we again filter the events based on the dimensionality. Finally, in line 6, we retrieve the Entity nodes from the specific log we are working on.

```
1 MATCH (l:Log{ID:'filename.csv'})--(e:Event)
2 WITH e, keys(e) AS properties
3 WHERE NOT ANY(x IN properties WHERE x IN ["Dimension1", ..., "
    DimensionN"])
4 WITH e WHERE EXISTS(e.‘EntityType‘)
5 MATCH (n:Entity {EntityType: "EntityType"})
6 WHERE e.‘EntityType‘ = n.ID AND n.log = "filename.csv"
7 CREATE (e)-[:CORR]->(n)
```

Creating Derived Entities

To allow the user to create derived entities, we decided to add a ”New Derived Entity” button on the Entities panel, as shown in Figure 4.13. Clicking on

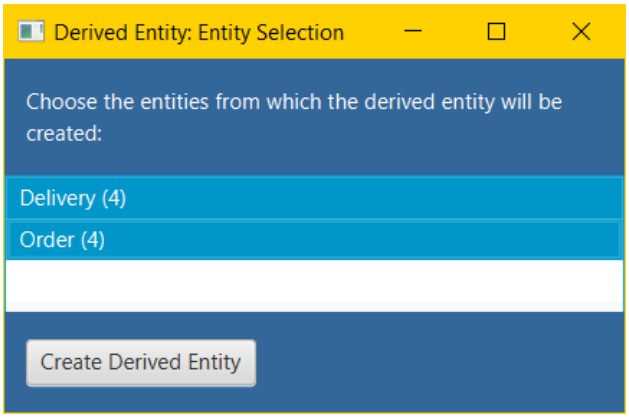


Figure 4.15: Two entities must be selected to create a derived entity.

this button will show a new window, shown in Figure 4.15, where the users can select which two entities will be used as input to create the derived entity.

In our running example, we have already created two entities, one based on the "Order" attribute, and another one based on the "Delivery" attribute, which is why we can see those entities as available options in Figure 4.15. As our next step, we create a derived Entity based on the "Delivery" and "Order" entities.

The adapted queries that create and connect the derived Entity nodes are shown next. The first query we execute is the one that creates the derived Entity nodes. In line 2, we identify those Event nodes that have both entities defined as properties, then, with the values of those properties, in line 6 we find their corresponding Entity nodes. Using the properties of these Entity nodes, we can define the properties of the derived entity in line 8. Also in line 8, we can see that we needed to add the entity ID of the original entities as additional properties to help us make the correlation with the Event nodes later on. Using our running example as reference, the properties of the derived Entity node between order 40 and delivery 432 would be: EntityType:'DeliveryOrder', uID:'DeliveryOrder_432_40', DeliveryID:'432', OrderID:'40', log:'Orders.csv'.

```
1 MATCH (l:Log{ID:'filename.csv'})--(e:Event)
2 WHERE EXISTS(e.`EntityType1`) AND EXISTS(e.`EntityType2`)
3 WITH DISTINCT e.`EntityType1` AS nID1, e.`EntityType2` AS nID2
4 MATCH (l:Log{ID:'filename.csv'})--(:Event)--(n1:Entity)
5 MATCH (l)--(:Event)--(n2:Entity)
6 WHERE n1.uID = 'EntityType1'+nID1 AND n2.uID = 'EntityType2'+nID2
7 WITH DISTINCT n1.ID as n1_id, n2.ID as n2_id
8 CREATE (:Entity{EntityType:"DerivedEntityType", uID:'
    DerivedEntityType'+'_'+toString(n1_id)+'_'+toString(n2_id),
```

```
    Entity1ID:n1_id, Entity2ID:n2_id, log:"filename.csv"})
```

After we create the derived Entity nodes, we execute the queries that correlate them with the Event nodes. We run this query twice, one for each original entity that compose the derived entity. In line 2, we find the events related to one of the original entities, and in line 4 we find the appropriate derived Entity node through the *EntityID* property to make the connection with the events in line 5.

```
1 MATCH (l:Log)--(e1:Event)-[:CORR]->(n1:Entity)
2 WHERE l.ID = "filename.csv" AND n1.EntityType="EntityType"
3 MATCH (derived:Entity)
4 WHERE derived.EntityType = "DerivedEntityType" AND n1.ID =
    derived.EntityID AND derived.log = "filename.csv"
5 MERGE (e1)-[:CORR]->(derived)
```

Finally, we create the :REL connections between the original Entity nodes and the derived Entity node. In line 1, we obtain all the derived Entity nodes. Then, in line 4, we find the original Entity nodes based on their IDs. Then, in order to obtain the same result for the direction of the :REL relations every time, we store both original entities in an array in line 5 so we can sort them alphabetically in line 6. Once that is done, we create the :REL connections in lines 8, 9, and 10.

```
1 MATCH (l:Log{ID:'filename.csv'})--(:Event)--(en:Entity{
    EntityType:'DerivedEntityType'})
2 WITH DISTINCT en
3 MATCH (en1:Entity)--(:Event)--(en)--(:Event)--(en2:Entity)
4 WHERE en1.ID = en.Entity1ID AND en2.ID = en.Entity2ID AND en1.
    EntityType <> en2.EntityType
5 WITH DISTINCT en AS dEnt, [en1,en2] AS ents
6 WITH DISTINCT dEnt, apoc.coll.sortNodes(ents, 'uID') AS
    sortedEnts
7 WITH dEnt,sortedEnts[0] AS ent1,sortedEnts[1] AS ent2
8 MERGE (dEnt)-[:REL{Type:'Reified'}]->(ent1)
9 MERGE (dEnt)-[:REL{Type:'Reified'}]->(ent2)
10 MERGE (ent1)-[:REL{Type:'DerivedEntityType'}]->(ent2)
```

Visualizing Entities

As a last step, we must also define a way for the users to visualize the created Entity nodes, providing them with feedback on the results.

Since the Event and Entity nodes are connected between them, we decided to put the visualization functions for the Entity nodes in the same panel already containing the visualization options for the Event nodes. This is why the button to visualize the Entities that have been created is placed on the same panel already introduced in Figure 4.10.

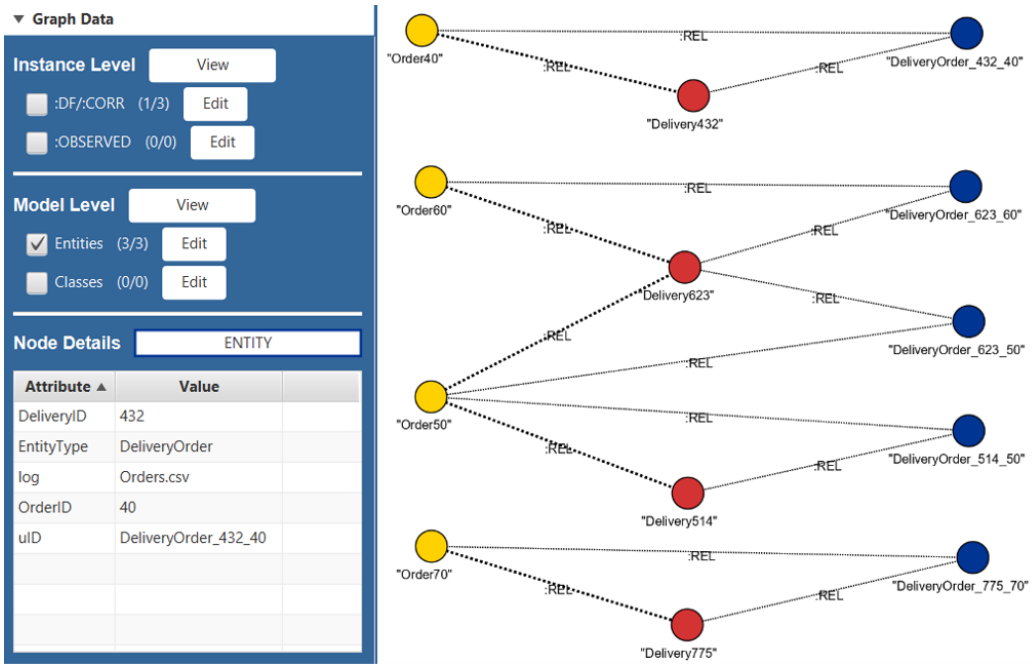


Figure 4.16: Entity nodes visualization.

Figure 4.16 shows the result of clicking on the "View" button next to the "Model Level" label. The existing Entity nodes are displayed on the Graph panel, and, similar to the Event nodes, clicking on a node will populate the table with the node properties. To improve the visualization, we use colors to distinguish between different Entity Types.

Figure 4.16 shows the entities created for our running example. The yellow Entity nodes on the left represent the "Order" entities, the red Entity nodes in the middle represent the "Delivery" entities, and the blue Entity nodes on the right represent the derived "DeliveryOrder" events. As we can see, these nodes are connected between them through the :REL relations.

As we can see at the top-left of Figure 4.16, the :CORR relations that correlate the events and entities can also be visualized using the Instance Level "View" button, but before looking into this, we must first discuss another result from the creation of the Entity nodes, which is the connection between events by directly-follows (:DF) relations.

4.5 Directly-Follows Relations

In this section, we continue describing the implementation of the Entity and Behavior Layer, detailing how we address the *Creating views* activity by

using the Entity nodes to connect the Event nodes through directly-follows relations. First, we define what the directly-follows relations represent in the data model. Then, we describe how we can visualize these relations in the user interface.

4.5.1 Defining the Directly-Follows Relations

The directly-follows relations create a temporal ordering of events based on the perspective of a specific entity, allowing us to analyze the behavior recorded in a sequential event log.

In the data model proposed in [1], this temporal ordering of events is represented through the :DF relations. The events are first grouped by entity and then ordered based on their timestamp, after which the :DF relations can be created between adjacent nodes in each grouping.

The data model allows us to correlate events to multiple entities, which means that events may have multiple :DF relations. For each new entity that is defined, a new perspective of the event data is created through the new :DF relations, providing a way for us to define different views of the same event log. This is how the Entity and Behavior layer is able to address the *Creating Views* activity.

Each :DF relation goes forward in time and is specific to exactly one entity type. To distinguish between the multiple :DF relations of an event, the data model defines the *EntityType* as its property, allowing us to query for event paths based on their common entity identifier.

[1] defines one query to create the :DF relations between event. This query retrieves all events correlated to a specific entity, orders them by timestamp and ID (to break ties between identical timestamps), and creates the :DF relations between the nodes.

In order for us to continue with the implementation of the Entity and Behavior Layer and address the *Creating Views* activity, we need to provide a way to define the :DF in our tool. In this case, other than having to specify the log to retrieve the correct set of Event and Entity nodes, we do not need to adapt the query that creates the :DF relations, we can use the query as originally defined. The only consideration that needs to be addressed for our tool is to determine how this query can be executed by the user and how the results can be visualized in the user interface.

4.5.2 Creating and Visualizing the :DF Relations

Creating the :DF relations

Since the :DF relations depend on the entity identifiers, we decided to execute the query that creates them right after the Entity node creation, which is triggered when the user clicks on the "Create Entity" button shown in Figure 4.14.

The query that creates the :DF relations is shown below. As mentioned earlier, no significant changes had to be made on the original query from [1]. In line 2, we specify the log that has the events to be ordered and the entity type that will be used to group the events. In line 3 we order the events based on their timestamp and ID, and in line 4 we group the events per Entity node. Finally, in line 7, each event is connected with the next in line through the :DF relations.

```
1 MATCH (l:Log)--(e:Event)--(n:Entity)
2 WHERE l.ID = "filename.csv" AND n.EntityType="EntityType"
3 WITH n, e as nodes ORDER BY e.Timestamp, ID(e)
4 WITH n, COLLECT(DISTINCT(nodes)) as nodeList
5 UNWIND RANGE(0,SIZE(nodeList)-2) AS i
6 WITH n, nodeList[i] AS first, nodeList[i+1] AS second
7 MERGE (first)-[:DF{EntityType:"EntityType"}]->(second)
```

Visualizing the :DF relations

For the visualization of the results in the user interface, we decided to use the Instance Level "View" button, first shown in Figure 4.10, to allow the users to visualize the :DF relations. The reason for this is that both the :DF and :CORR relations directly involve the nodes at the Instance Level (i.e. the Event nodes), so their visualizations go hand in hand. Once an Entity Type has been defined, if the ":DF/:CORR" box is checked when the user clicks on the Instance Level "View" button, the tool will not only retrieve the Event nodes, but also the :DF relations, Entity nodes and some :CORR relations as follows. Not all the :CORR relations between Event and Entity nodes are retrieved because the :DF relations already build a path between the events that belong to the same entity, so showing the connection between the Entity node and one Event node is enough to identify which events belong to each entity.

Figure 4.17 shows an example of the visualization of the :DF relations connecting the Event nodes. This view shows the :DF relations for the "Order" and "Delivery" entity types of our running example, where both of them were specified as dimensions when the log was imported. The blue

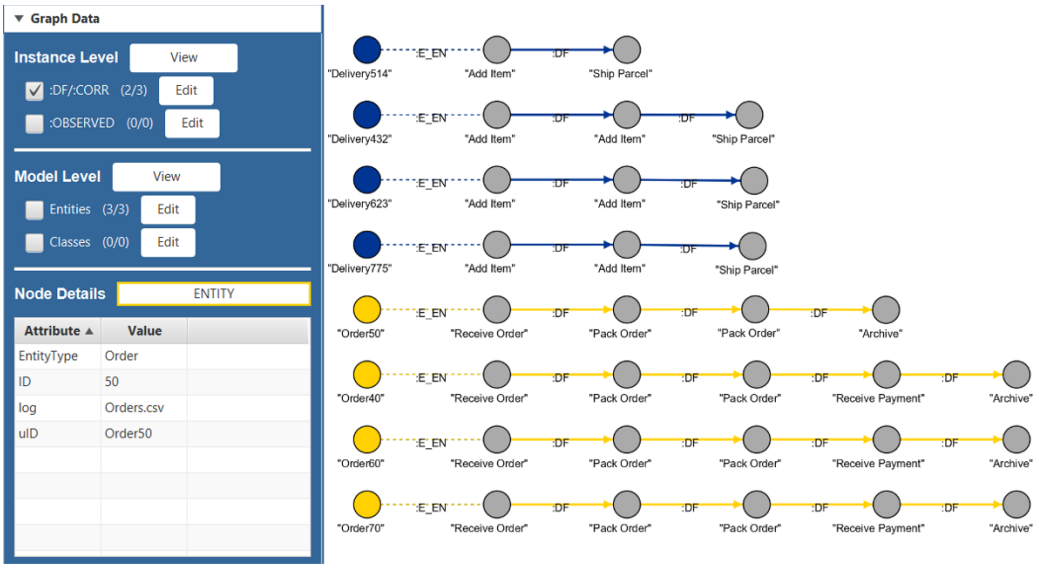


Figure 4.17: Event nodes connected by :DF relations of the Order and Delivery Entity Types.

nodes represent the "Delivery" Entity nodes and the yellow nodes represent the "Order" Entity nodes, all of them connected by the :CORR relation to the first Event node in their respective :DF paths.

For the initial placement of the nodes in the Graph panel, we adapted the Sugiyama algorithm, a procedure used to define the drawing layout for hierarchical graphs [5]. The details on our implementation of the Sugiyama algorithm, which is based on the technique described in [29], can be found in Appendix A.

To allow the user to decide what node types and relations to visualize in the Graph panel, we implemented an "Edit" button, which shows a new window to the users where they can select which Entity Type to use for the visualization. This window, shown in Figure 4.18, displays the available entity types to display the ordered nodes based on our running example, where three entity types have been defined, one based on the "Order" attribute, another based on the "Delivery" attribute, and a third one created as a derived entity between these two attributes. Figure 4.17 already shows the visualization when both the "Order" and "Delivery" Entity Types are selected in the window shown in Figure 4.18, but if we change the selection to display only the "DeliveryOrder" derived entity, the result is the one shown in Figure 4.19.

Figure 4.19 shows how the derived entity defined in our running example connects the Event nodes that were initially correlated exclusively to the

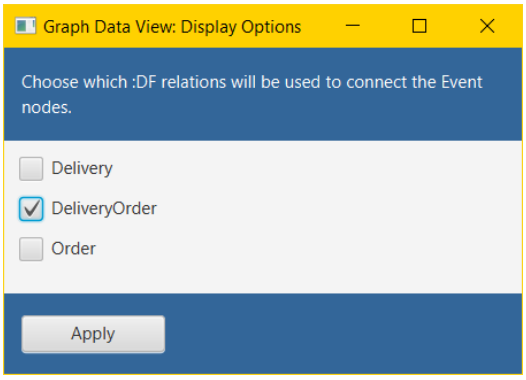


Figure 4.18: Display options window. Users can select which entity will be used to visualize the connections between Event nodes.

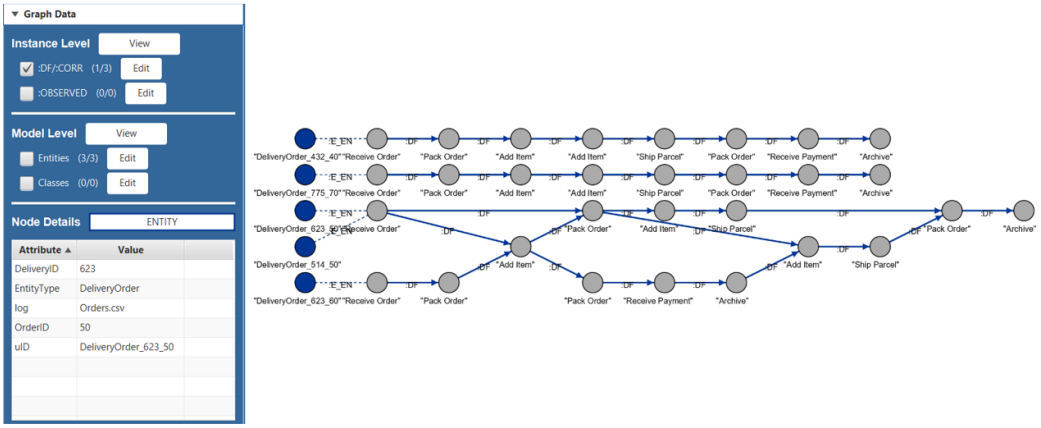


Figure 4.19: Event nodes connected by :DF relations of the DeliveryOrder Entity Type with the Delivery and Order attributes specified as independent dimensions.

"Order" and "Delivery" entities. We can see for example how the :DF paths of the derived entity connects a "Pack Order" event (event 10 from Table 4.2), which is originally correlated to the "Order" entity, to a couple of "Add Item" events (events 11 and 13 from Table 4.2), which were originally correlated to the "Delivery" entity. The Event nodes were connected this way because of our definition of "Order" and "Delivery" as independent dimensions, which falls under the scenario S3 defined in 4.2.

In contrast, had we not defined the "Delivery" and "Order" attributes as independent dimensions during the data import and defined the derived entity under scenario S2 instead, the resulting connections between events would be different, as it can be observed in Figure 4.20. We can see how the "Pack Order" event is no longer connected to two "Add Item" events. Determining which behavior is correct depends on the domain knowledge

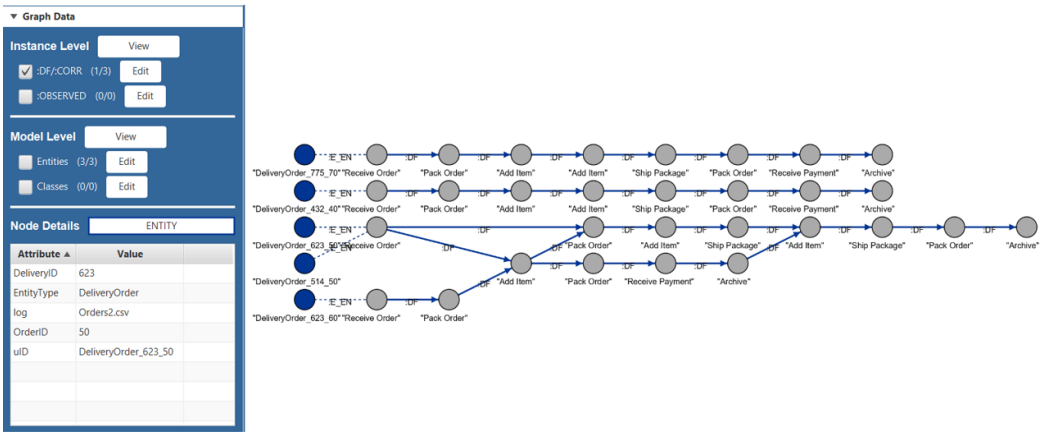


Figure 4.20: Event nodes connected by :DF relations of the DeliveryOrder Entity Type with no independent dimensions specified.

from the users regarding the process being evaluated.

After looking at how events are connected between them based on the entities defined for the log, we can discuss how we can take advantage of these relations to reduce redundancy in the data stored for each event by defining entity type attributes.

4.6 Entity Type Attributes

In this section, we define the Entity Type Attributes, which describe attributes whose value does not change for the Event nodes connected in the :DF path of an entity. First, we describe how these attributes are represented in one-dimensional event logs in the form of case attributes. Then, we describe the challenge of implementing this concept in the graph data model. Finally, we present our solution to define the Entity Type Attributes.

4.6.1 Case Attributes

In one-dimensional event logs, we say that all the events correlated to the same entity belong to the same *case* [1]. If all the events within a case share the same value for a specific attribute, we can call that attribute a *case attribute*, but if the attribute changes along the case, it is simply an event attribute. Then, if the same attribute is a case attribute for every case in the event log, we can call such attribute a *global case attribute*. Thus, when an attribute is called a global case attribute, we can assume that every event inside a case has the same value for that particular attribute, even if each

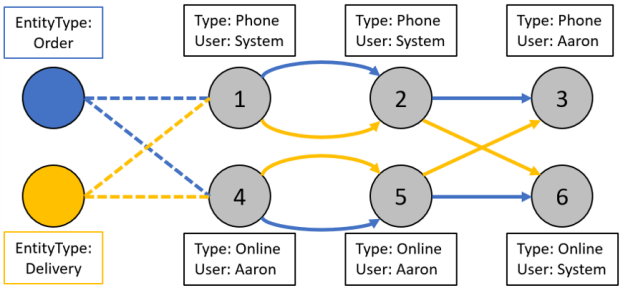


Figure 4.21: Depending on the :DF path, entity type attributes may change.

case has a different value.

As an example, we can look at Table 4.2. If we define the "Order" attribute as the case identifier, we can consider the "Type" attribute as a global case attribute, since all the events in every case share the same value. The "Type" attribute for all events from orders 40 and 50 is "Phone" while the "Type" attribute for all events from orders 60 and 70 is "Online".

The analysis of event logs can benefit from identifying these type of attributes since events can be simplified by no longer having to specify the case attributes individually, which is something we want to replicate in our tool.

4.6.2 Challenge of Replicating Case Attributes

Replicating the Case Attributes into the graph data model is not a straightforward task. While a one-dimensional event log can define the global case attributes from the start once the case identifier is defined, the same does not apply in our case, since the graph data model under which the tool is built allows us to work with multi-dimensional event logs by defining multiple entities, providing different perspectives of the data. For each perspective, the potential case attributes, or *Entity Type Attributes* in our case, may change, presenting a challenge for its replication in our tool.

We can look at Figure 4.21 to see how entity type attributes may change depending on the :DF path that we decide to follow. Following the paths from "Entity Type:Order" (blue), we can see that for every path, the value for the "Type" attribute remains the same. For events 1, 2, and 3, the value is always "Phone" and for events 4, 5, and 6, the value is always "Online". In this case, the "User" attribute is not an entity type attribute, since not all of the nodes for every entity path have the same value. A simple comparison between the values of the "User" attribute for events 2 (User:System) and 3 (User:Aaron) discard the possibility of this being an entity type attribute.

This, however, changes if we follow the paths from "Entity Type: Deliv-

ery" (yellow), where now the "User" attribute can be considered an entity type attribute since, for every path of the entity type, the value for the "User" attribute remains the same, "System" for events 1, 2, and 6, and "Aaron" for events 4, 5, and 3, and in contrast, the "Type" attribute becomes simply an event attribute.

Therefore, we must consider these different perspectives of the data to successfully replicate the case attributes in our tool as entity type attributes.

4.6.3 Implementing the Entity Type Attributes

To replicate the case attributes in our tool, we implemented a function to identify the attributes that appear in all the :DF paths of an entity type and whose value does not change for all the events correlated to the same Entity node, which represent the potential entity type attributes.

Using both a Cypher query and Java code, the function is able to identify entity type attributes based on a given entity type. First, the Cypher query retrieves the data from the Event node properties, providing a list of all the event attributes and a collection of their distinct values that appear in every :DF path of the entity type provided as an argument. Then, the Java code iterates over these values to determine if the attribute has 1 distinct value for every :DF path and also checks if that value appears in every Event node in the path. If the Java code finds more than 1 distinct value in any path, or finds out that not every Event node in any path contains the attribute as a property, that attribute is discarded as a potential entity type attribute. The details on the Cypher query and the activities performed in Java can be found in Appendix B.

We decided to execute this function right after a new entity type is created, which is done by clicking on the "Create Entity" button shown in Figure 4.14 or by clicking on the "Create Derived Entity" button shown in Figure 4.15. If our function finds at least one potential entity type attribute, a new window is shown to the user.

This window, shown in Figure 4.22, allows the user to select whether they want to move the attributes into the Entity node and remove the attribute from the Event nodes, or if they want to keep the attributes as they are. This is because, while removing an the attribute from the Event nodes reduces redundancy in the database, the user will no longer be able to select that attribute to define a new entity, so we leave this decision to the user.

Even when the users decide to send an attribute to the Entity nodes, they might need the attribute later on a subsequent iteration of the process mining analysis, which is why we decided to implement a button that allows them to return the attribute from the Entity node to the Event nodes. The

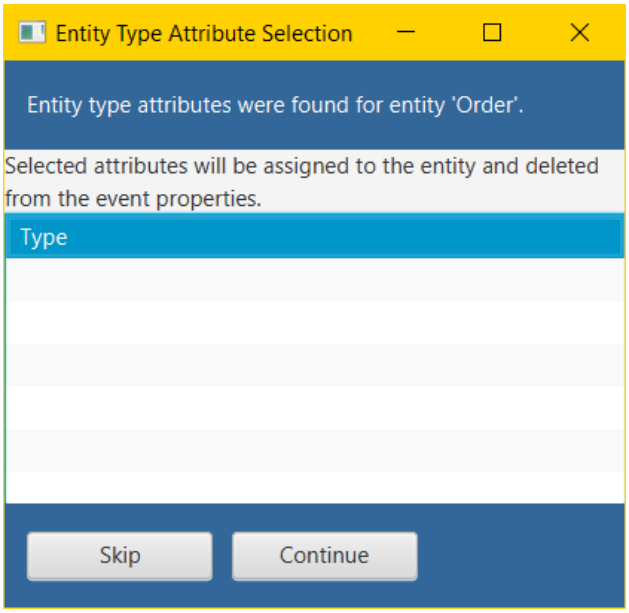


Figure 4.22: After an Entity is created, users can decide if the attribute is assigned to the entity and deleted from the Event nodes or keep the node properties intact.

"Return Entity Attribute" button was added to the "Entities" panel of the user interface, as shown in Figure 4.13.

Once the Entity nodes have been defined and the :DF paths have been created, we can look at the second way in which the Entity and Behavior Layer addresses the *Enriching logs* activity through the creation of the Class nodes.

4.7 Creating Classes

In this section, we focus on the second way in which the tool addresses the *Enriching logs* activity through the creation of event classes. First, we describe how the event classes are represented in the data model from [1]. Then, we discuss the considerations we need to account for to create the Class nodes in our tool. Finally, we describe how we can create and visualize the Class nodes through the user interface.

4.7.1 Defining the Event Classes

In Section 4.3, we mentioned that any event log that was meant to be imported should contain, among other things, one column defining the Activity attribute for the event. This mandatory attribute represents the first way in

which we can classify the events in the log, but [1] describes a second way in which events can be classified. This second way is presented in the form of event classes, represented in the data model by Class nodes. In other words, a Class node represents a set of events with the same characteristics, which could mean a set of events with the same Activity or the same combination of different attributes.

Class nodes are defined by a unique *ID* and *Type* properties. The Type property can be based on the Activity attribute or any other combination of attributes.

In addition to Class nodes, the data model from [1] also presents two new types of relations, one between Event and Class nodes, the :OBSERVES relations, and a second one between Class nodes, the :DF_C relations. The :OBSERVES relations associate the Class nodes with the events that match their defining attributes, while the :DF_C relations connect the Class nodes based on the aggregated :DF relations of events for a specific entity type. This entity type is included as a the property *EntityType* in the :DF_C relations, which means that these relations can be created only after at least one Entity Type has been defined in the database.

There are 3 existing queries in [1] that provide a way to create and connect the Class nodes. The first query queries for all distinct values of particular event attributes and creates a new Class node for each retrieved value. The second query links the Class nodes to those Event nodes that match on the defining event attributes. The third and final query creates the :DF_C relations between Class nodes by aggregating the :DF relations between events that have the attributes that define the classes. The :DF relations are aggregated only if they connect events correlated to the same entity, and classes are only connected if they are from the same type.

In order for us to complete the implementation of the Entity and Behavior layer and fully address the *Enriching logs* activity, we need to provide a way to create and connect the Class nodes in our tool.

4.7.2 Considerations for the Class creation

The queries provided in [1] can help us to create and connect the Class nodes, but we also need to add a match to the query to make sure we retrieve the Event and Entity nodes from the correct log, since we must consider that the database may contain multiple, sometimes unrelated, event logs.

As a second consideration, we must account for a scenario where, during the process mining project, the user may want to define additional :DF_C connections for already existing Class nodes. In this case, we should prevent the creation of duplicate nodes in the database, so this must be addressed in

the queries as well.

A third consideration is related to the implementation of the *Process discovery* activity from the next layer, the Model Layer. The process discovery activity creates a process model that describes the behavior of the process being evaluated based on the connections between the events registered on the event log. Since the Class nodes already describe the connections between activities or other attributes of the Event nodes, we can consider them as our inputs for the process discovery. However, matching Event nodes with Class nodes composed of more than one attribute could increase the complexity of the query, which is why we must consider adding an additional property to the nodes describing the composed attribute to simplify the queries later on.

In addition to these three considerations, we must also determine how these queries can be executed by the user and how the results can be visualized in the user interface.

4.7.3 Creating and Visualizing the Class Nodes

Creating the Class nodes

To execute the queries that create and connect the Class nodes, we need two inputs from the user. The first one is the event attribute that will be used to define the class type, and the second one is the entity type defining the :DF relations that will be used as a reference to create the :DF_C relations.

To allow the user to select these inputs through the user interface, we decided to add a "New Class" button in the Class panel, as shown in Figure 4.23. This button shows a new window asking the user to select the attribute(s) to be used to define the Class type. Since the class type can be defined by a combination of different attributes, this window allows the user to select more than one attribute, as shown in Figure 4.24.

For our running example, we will define a Class type based on the "Activity" and "Life-cycle" attributes from Table 4.2. This will help us distinguish between the activities that define the start of the packing (Activity:"Pack Order", Life-cycle:"Start") from those that define the end of the packing (Activity:"Pack Order", Life-cycle:"Complete").

After the user chooses the attribute(s) to define the class, clicking on "Continue" will show a second window, shown in Figure 4.25, where the user can select the Entity Type that will be used to aggregate the :DF relations and create the :DF_C classes. For our running example, we will choose the derived entity type, "DeliveryOrder", to make the :DF aggregation.

Once the Entity Type has been selected, clicking on the "Create Class" button will execute the queries that create the Class nodes and the :OB-

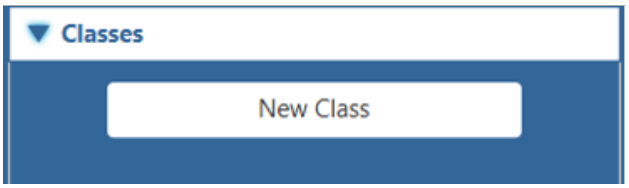


Figure 4.23: Class panel for the user interface.

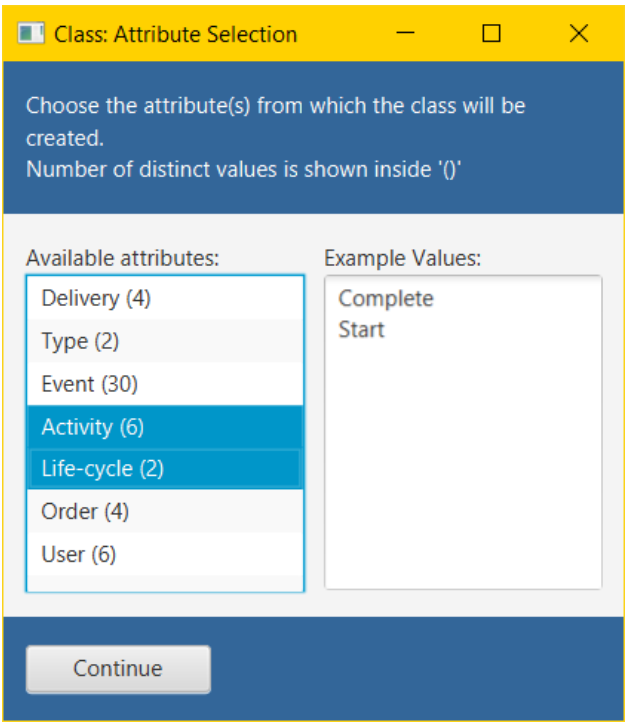


Figure 4.24: Class type selection window.

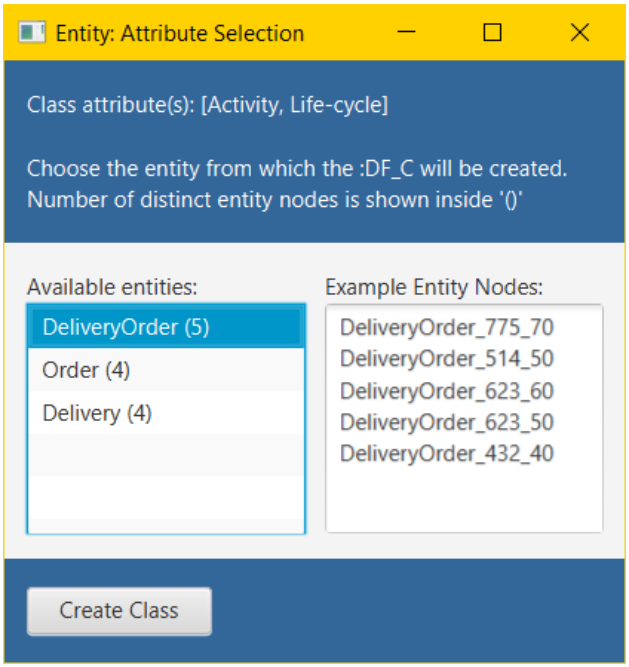


Figure 4.25: Entity selection window to create the :DF_C relations.

SERVES and :DF_C relations.

The queries that create and connect the Class nodes are shown next. The first query creates the Class nodes. In line 1, we make sure we use only the events from a specific log. In line 2, we obtain the values from all the attributes that will make up the class ID. Then, in line 3, we search the database in case the Class nodes already exist to prevent the creation of duplicate nodes. The existing Class nodes are filtered out in line 4. Finally, in line 5 we create the nodes, where we define add an additional *Log* property to be able to retrieve the correct nodes for the subsequent queries.

```
1 MATCH (l:Log{ID:"filename.csv"})--(e:Event)
2 WITH DISTINCT e.`Attribute1` AS `ValueAttribute1`, ..., e.`
    AttributeN` AS `ValueAttributeN`
3 OPTIONAL MATCH (:Log{ID:"filename.csv"})--(:Event)--(c:Class{
    ID: `Attribute1`+"+"...+`AttributeN`})
4 WITH c, `ValueAttribute1`, ..., `ValueAttributeN` WHERE c IS NULL
5 CREATE (:Class {`Attribute1`:`ValueAttribute1`, ..., `
    AttributeN`:`ValueAttributeN`, Type:"Attribute1+...+AttributeN",
    ID: `ValueAttribute1`+"+"...+`ValueAttributeN`, Log:"filename.
    csv"})
```

The second query connects the Class nodes with the Event nodes. In line 1, we retrieve the Event nodes from the log. In line 2, we retrieve the recently created Class nodes. In line 3 we match the Class nodes with the

Event nodes that contain the class type as attributes and finally we create the :OBSERVES relation in line 4.

```
1 MATCH (l:Log{ID:"filename.csv"})--(e:Event)
2 MATCH (c:Class) WHERE c.Log = "filename.csv" AND c.Type = "
    ClassType"
3 AND e.`Attribute1` = c.`Attribute1` ... AND e.`AttributeN` = c.`
    AttributeN`
4 MERGE (e)-[:OBSERVED]->(c)
```

Then, in the third query, we create the :DF_C relations. In line 1, we find the Events connected by the :DF relation that are also connected to the Class nodes. In line 2, we make sure that the events are correlated to the same Entity node and in line 3 we make sure they are connected to the same Log node. In line 5 we aggregate by counting the :DF relations to finally create the :DF_C relations in line 6.

```
1 MATCH (c1:Class)<-[:OBSERVED]-(e1:Event)-[df:DF]->(e2:Event)
    -[:OBSERVED]->(c2:Class)
2 MATCH (e1)-[:CORR]->(n)<-[:CORR]-(e2)
3 MATCH (e1)--(l:Log)--(e2) WHERE l.ID = "filename.csv"
4 AND n.EntityType = "EntityType" AND df.EntityType = "EntityType"
     AND c1.Type = "ClassType" AND c2.Type="ClassType"
5 WITH n.EntityType AS EType, c1, COUNT(df) AS df_freq, c2
6 MERGE (c1)-[rel2:DF_C{EntityType:EType}]->(c2) ON CREATE SET
    rel2.count=df_freq
```

Finally, in case the Class type is composed of more than 1 event attribute, we run a final query to set the combined attribute as a new event property. In line 2 we filter for the Class nodes recently created and we set the event attribute in line 4. The definition of this property can be helpful if this Class type is used as an input for the implementation of the *Process Discovery* activity at a later stage.

```
1 MATCH (l:Log{ID:"filename.csv"})--(e:Event)--(c:Class)
2 WHERE c.Type = "ClassType1+...+ClassTypeN"
3 WITH DISTINCT e,c
4 SET e.`ClassType1+...+ClassTypeN` = c.ID
```

Visualizing the Class nodes

Now that the Class nodes exist in the database, we need to define a way to visualize them in the user interface. Since the Event and Class nodes are connected, we decided to put the visualization functions for the Class nodes in the Graph Data panel, first shown in Figure 4.10. By doing this, all the different types of nodes introduced in the data model from [1] can be selected for visualization in the same panel.

To visualize the connection between Event and Class nodes, users need to check the :OBSERVES box and click on the Instance Level "View" button. The result can be observed in Figure 4.26. We should also mention that we took the design decision to display a maximum of 3 Event nodes for each Class node in order to maintain the readability of the Graph panel.

Then, to visualize the :DF_C relations between Class nodes, users need to check the "Classes" box and click on the Model Level "View" button. The result can be observed in Figure 4.27. For this visualization, we decided to add two gray, "virtual" nodes with the intention of showing the EntityType of the :DF_C relations that connect the Class nodes.

For our running example, Figure 4.27 shows the result of creating the Class nodes based on the combination of the "Activity" and "Life-cycle" attributes of the Event nodes. The connections between the Class nodes are defined based on the aggregated :DF relations of the derived entity "DeliveryOrder". We can see how the "Receive Order+Complete" Class node is followed by the "Add Item+Complete" and the "Pack Order+Start" activities, which represent the connections between the "Receive Order", "Add Item" and "Pack Order" Event nodes from Figure 4.19.

The implementation of the queries that create the Class nodes and its relations not only completes the functions needed to address the *Enriching logs* activity, but also marks the final functionality needed to complete the implementation of the Entity and Behavior Layer and the data model presented in [1]. The next step is to extend this data model by implementing the next layer in the scope defined on Section 3.3, the Model Layer, which addresses the *Process discovery* activity.

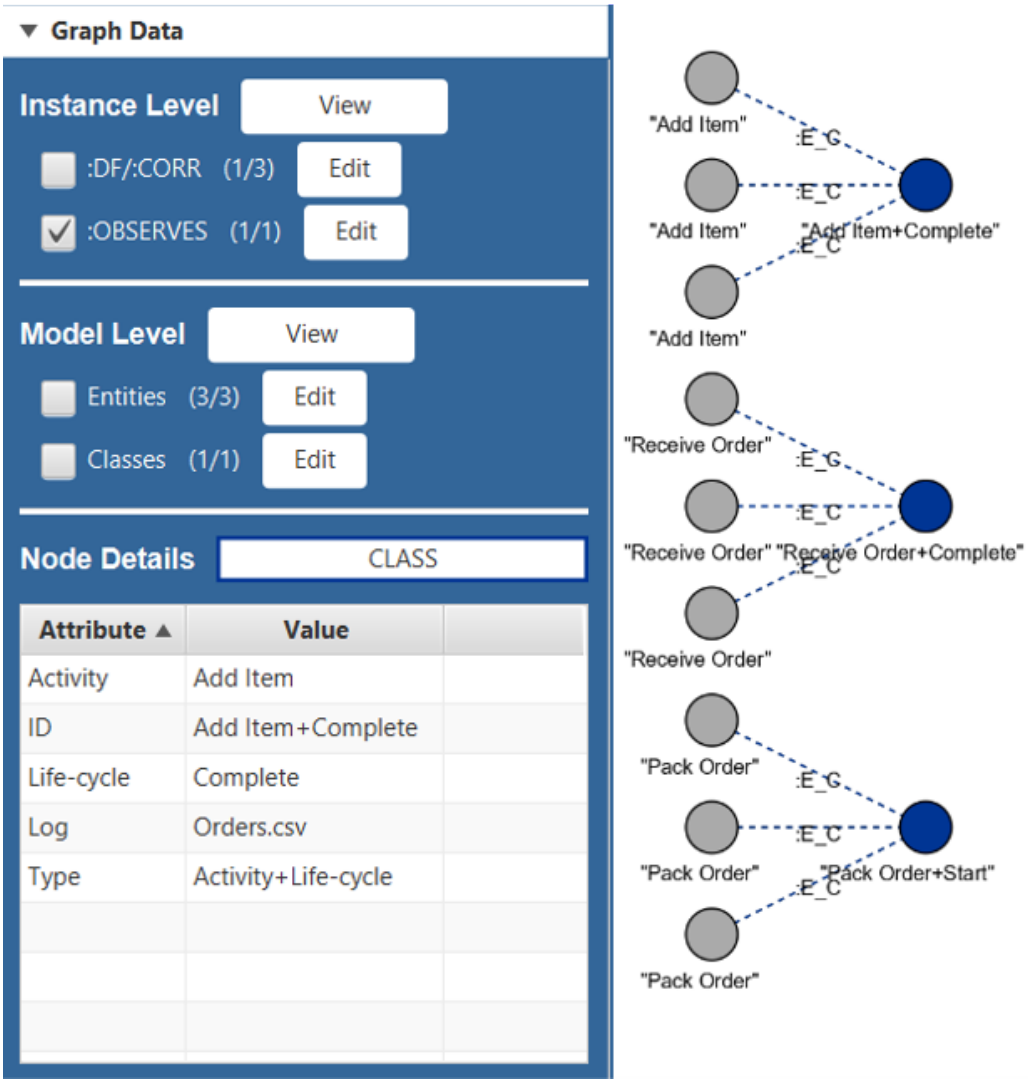


Figure 4.26: Graph panel showing the :E_C connections between Event and Class nodes.

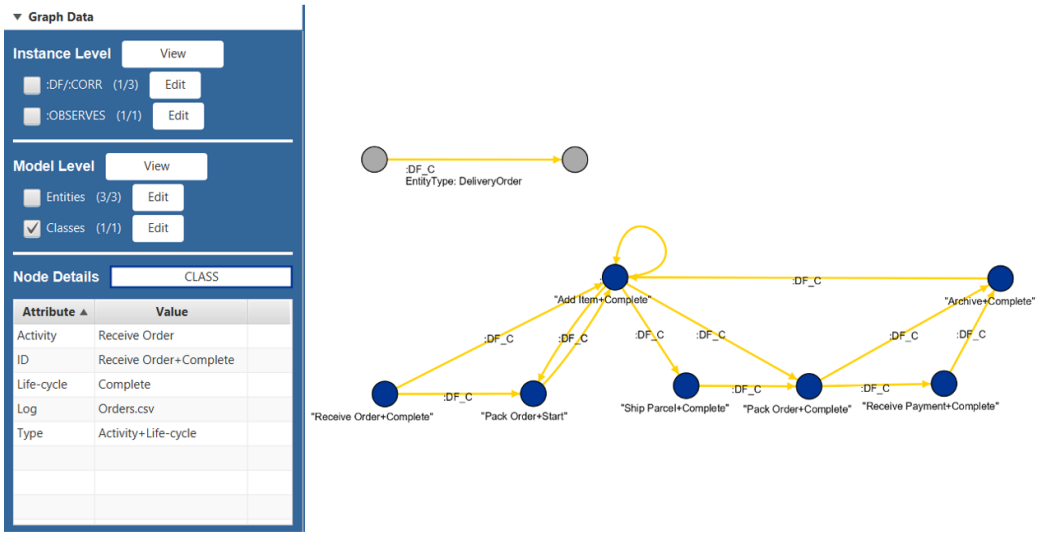


Figure 4.27: Graph panel showing the :DF_C connections between Class nodes.

Chapter 5

Extending the User Interface to Execute Process Discovery

In Chapter 4, we define how the Event and Entity and Behavior Layers were implemented in the tool, addressing the *Importing Data*, *Creating views* and *Enriching logs* activities associated with process mining projects. In this chapter, we discuss the implementation of the Model Layer, addressing the *Process discovery* activity. In Section 5.1, we describe how the outputs of the process discovery activities can be connected with the rest of the data model. Then, in Section 5.2, we describe our implementation of a process discovery algorithm, the Heuristic Miner.

5.1 Connecting the Process Models with the Data Model

In this section, we describe how the data model from [1] was extended to account for the creation of process models. First, we describe how the outputs of process discovery are represented. Then, we discuss four key aspects to consider for the representation of process models in the graph data model. Finally, we present our extension to the data model to consider the storage of process models in the database.

5.1.1 Process Discovery

Process discovery refers to the process mining task whose objective is to create a process model based on the information from an event log that is able to capture the behavior seen in the log [6]. Process discovery algorithms

attempt to map an event log onto a process model that is representative of the behavior of the log [6].

The process models that result from the process discovery activities can be represented through a variety of modeling languages, but we can identify the most common ones with help from the research made in [7]. In this study, they review and compare several process discovery methods to provide a classified inventory and a benchmark to allow researchers to compare new methods against the existing ones.

In [7], they identify the modeling languages used by the most common groups of discovery algorithms to describe process models. The model languages mentioned in that research include Petri nets [16], process trees [17], causal nets [18], state machines [19], BPMN models [20] and declarative models [21]. One common characteristic of these modeling languages is that they use the nodes and edges of the graph structure to represent the models.

Therefore, to implement the Model Layer in our tool and address the *Process discovery* activity, we first need to determine how the modeling language that represents the discovered models can be included in the graph data model. Next, we discuss four key aspects that need to be accounted for to achieve this.

5.1.2 Key Aspects to Extend the Data Model

Before implementing any process discovery algorithm, we must describe how the data model from [1] can be extended to account for the creation of process models. To define a way to make the extension, we consider four key aspects, the first two related to the representation of the process models, and the last two related to the possible connections with the existing data model.

First Aspect The first aspect we need to consider is related to the modeling languages identified in [7]. As mentioned earlier, the one thing these modeling languages have in common is that all of them are different representations of graph structures, which fits directly with the database used for our tool. This means that, as long as we work with algorithms that use any of those model languages, we should be able to store them in the database through nodes and edges.

Second Aspect The second aspect is related to the characteristics of these modeling languages. Petri nets, process trees and other languages use the nodes and edges of the graph structure to fulfill different purposes in the representation of models based on the discovery algorithm used. For exam-

ple, a node in a Petri net could represent a *place* between *transitions* for the α-Algorithm, while a node in a process tree could represent a *cut* for the Inductive Miner. These differences between algorithms and its representations must be considered while defining the way in which process models can be represented in the graph database.

Third Aspect The third aspect is related to the factors that influence the process model obtained as a result. These factors are the event log used as an input and the discovery algorithm used to create it. Changing the event data, the discovery algorithm, or the algorithm parameters will result in a new, different process model. Therefore, we must also define a way to connect the discovered process models with the log and to describe the algorithm and the parameters used to generate them.

Fourth Aspect The fourth and final aspect considered is related to the graph database. So far, we have described how we can import or create Log, Event, Entity and Class nodes in the database, and our next step is to create process models and store them in the same database using different types of nodes and relations. The fact that we are planning to use the same space to store Events and Models provides us with a very clear opportunity to connect the process models with the event data. This connection could help us address challenge C10, one of the current challenges described in the Process Mining Manifesto, which mentions, among other things, that it is necessary to improve usability for the non-experts by linking the event data with process models to provide valuable interactions with end-users [3]. Therefore, we must also consider how we can include these connections during the process discovery.

5.1.3 Connecting the Process Models

Based on the previous four aspects, we define the extension for the data model proposed on [1]. The updated data model can be observed in Figure 5.1, which shows three new node types, the *Algorithm*, *Model*, and *Model_node*, and five new relations, the *:MAPS*, *:PRODUCES*, *CONTAINS*, *REPRESENTS*, and *MODEL_EDGE*, which help us describe and connect process models in the graph database. These node types and their relations are described next.

Algorithm The purpose of the Algorithm node is to provide a connection between the Log and the Process Model and to store the algorithm param-

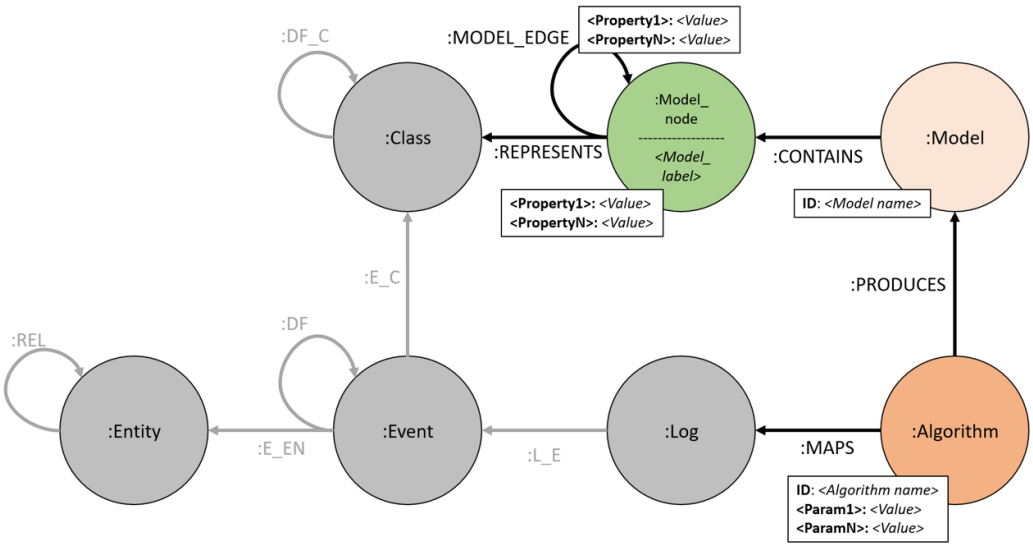


Figure 5.1: Extended data model to include process models.

eters defined by the user to discover the process model. Every time a new process model is created, a new Algorithm node is created as well. This was done to let the users know that the process model was created from within the tool and the log used to generate it is still inside the database. Since the process discovery algorithms map an event log, the Algorithm node is connected to the Log node through a :MAPS relation. A Log node may be connected to several Algorithm nodes, but every Algorithm node is connected to only one Log node. The only defined property for this node is the "ID", containing the name of the algorithm used for the process discovery. The parameters needed to execute the algorithm (if any), can be added as additional attributes to the node. Finally, since the discovery algorithms produce process models, the Algorithm node is also connected with the Model node through a :PRODUCES relation, which is a *1:1* connection.

Model_node The Model_node nodes represent the process model. As we discussed earlier, these nodes can fulfill different purposes depending on the graph structure it is representing. This node is also used to establish the connection with the event data. A second label can be defined for this node to provide more details on the type of graph structure it is representing (e.g. :Model_node/:ProcessTree_node). The properties of this node depend on the model language it is representing. For example, we could have a property describing the activity name for one model, and a different property describing the root node for another. To define the graph structure that connects

the nodes, we introduce the :MODEL_EDGE relations, whose properties, similar to those of the node, depend on the modeling language. Finally, we can connect these nodes with the already existing Class nodes through the :REPRESENTS relations to connect specific model activities with the event data. This connection allows us to make queries that go one step further and correlate the model activities with specific events.

Model The Model node represents the collection of model nodes that compose a process model. The purpose of this node is to help us retrieve all the Model_node nodes that make up the process model. The connection between the Model and Model_nodes is done through the :CONTAINS relations. In addition, these nodes provide us with a simple way to identify all the models currently in the database. The Model nodes have one property, the "ID", which is a custom name to distinguish one model from the next; an additional constraint was added to the database to make sure no two models share the same name.

Now that we have described the extensions to the data model providing a guideline to store process models in the graph database, we can look at the process discovery algorithm implemented for this tool that allows us to obtain these process models.

5.2 Implementing the Heuristic Miner algorithm

In this section, we describe how the Heuristic Miner, a process discovery algorithm, was implemented in the tool. First, we describe why this algorithm was implemented. Then, we provide more details on one of its existing implementations, the Flexible Heuristic Miner. Then, we define the scope of our implementation and how the outputs of the algorithm can be represented in the data model. Then, we describe the queries built to execute the different steps of the algorithm. Finally, we describe how the user can interact with the user interface to execute the algorithm and visualize the results.

After defining how a process model can be represented in the graph database, we can address the *Process discovery* activity of the Model Layer by implementing a process discovery algorithm that generates a process model based on an event log.

We chose to implement the Heuristic Miner algorithm because, in contrast with other algorithms that can also be represented with a graph structure, the premise of the Heuristic Miner allows us to take the most advantage of

the data already generated in the data model up to this point.

As mentioned in [6], heuristic mining algorithms "take frequencies of events and sequences into account when constructing a process model. The basic idea is that infrequent paths should not be incorporated into the model" [6]. In other words, the premise of these algorithms is to build a process model by removing the infrequent relations between activities.

From our current data model, we already have a type of node that describes the existing relations between activities in the form of Class nodes and its :DF_C relations. As we saw in Section 4.7, the :DF_C relations thatb-connect the Class nodes are based on the aggregated :DF relations of events for a specific Entity Type, so every path between two Event nodes of the same Entity Type is represented by a path between two Class nodes. If we were to add the frequency with which these paths occur in the Event nodes, we could already observe which are the most frequent and infrequent paths in the event log, providing the ideal input to execute a heuristic miner algorithm.

5.2.1 Flexible Heuristic Miner

The Flexible Heuristic Miner (FHM) [8] is an updated version of the Heuristic Miner algorithm originally described in [10]. The FHM is "a heuristics-driven, control-flow mining algorithm" [8] which is also built under the premise of removing infrequent relations between activities to discover a process model.

To use the FHM to discover a process model based on an event log, the log must be analyzed to find tasks that follow other tasks and determine if there is a dependency relation between them. To analyze these relations, the FHM builds a Dependency Graph (DG), and in order to build this graph, the FHM defines the following types of relations:

1. $a >_W b$ (Direct successor). We say that b is a direct successor of a if there is a trace in log W where b directly follows a.

2. $a >>_W b$ (Length-Two Loop). We say that a and b make up a length-two loop if there is a trace in log W where a is directly followed by b and then b is directly followed by a, and $a \neq b$.

In addition to these relations, it also defines three dependency measures to indicate how certain we are that there is a dependency relation between two tasks. Let us say that $|a >_W b|$ defines the frequency with which $a >_W b$ occurs and $|a >>_W b|$ defines the frequency with which $a >>_W b$ occurs, then, we can define these three measures as follows:

1. (Absolute) Dependency.

$$a \Rightarrow_W b = \left(\frac{|a >_W b| - |b >_W a|}{|a >_W b| + |b >_W a| + 1} \right) if(a \neq b)$$

2. Length-1 Loop (L1L) Dependency.

$$a \Rightarrow_W a = \left(\frac{|a >_W a|}{|a >_W a| + 1} \right)$$

3. Length-2 Loop (L2L) Dependency.

$$a \Rightarrow^2_W b = \left(\frac{|a >>_W b| - |b >>_W a|}{|a >>_W b| + |b >>_W a| + 1} \right)$$

After defining these relations, the FHM defines three steps to execute the algorithm: (1) Mining the dependency graph, (2) Mining of the splits/joins, and (3) Mining long-distance dependencies, which we briefly describe next.

Step 1. Mining the dependency graph The dependency graph is a graph structure that represents the causal relations between the tasks of the event log, specifying their dependency measures. We can see an example in Figure 5.2. This figure shows how the dependency graph would look like assuming we have the log shown in the top-left. We can see how each node in the DG represents a task of the event log and each edge between nodes represents the causal relation between them. Also, we can see how each edge defines the frequency (Freq) and dependency (Dep) between the two connecting tasks. In the case of the self-loop of task E, we use the L1L dependency measure instead of the normal dependency.

To mine the dependency graph, the FHM defines 4 thresholds to determine if a dependency relation is accepted or not. The first three thresholds are the *Dependency threshold*, the *Length-One Loop threshold*, and the *Length-Two Loop threshold*, which are based on the measures of the same name and whose default value is 0.9. The fourth threshold is the *Relative to Best threshold*, which indicates that we will also accept relations that have either a dependency measure above the *dependency threshold* or their difference with respect to the best dependency measure is lower than the *relative-to-best threshold*. The FHM does not define a default value for this threshold, but for one of their examples they use 0.05, while ProM leaves it at 0.

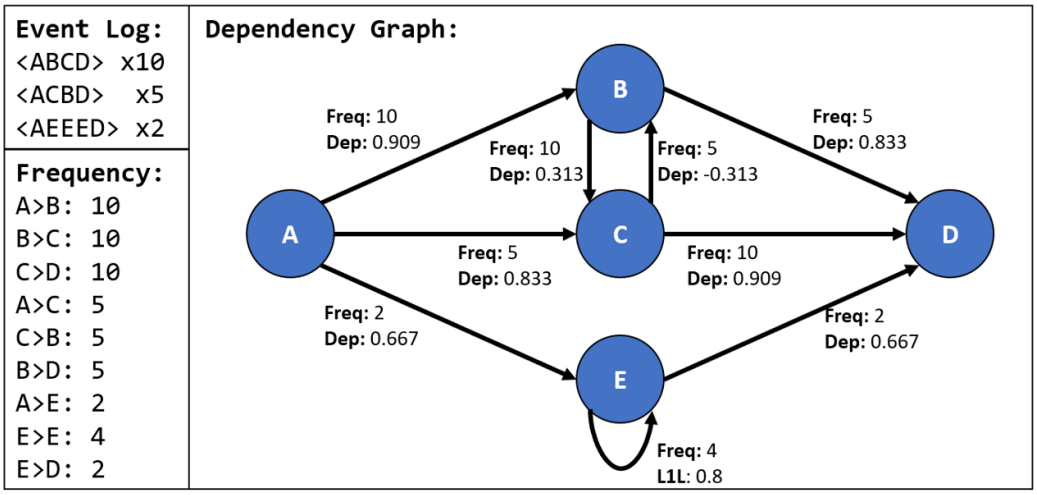


Figure 5.2: Dependency Graph built based on the Event Log shown in the top-left.

These thresholds are used in the formal definition of the mining of the DG, where 12 steps are defined to determine which relations are accepted and which infrequent relations are removed from the DG. We will provide additional details regarding the formal definition in Section 5.2.5 when we implement it for our tool.

In addition, the FHM uses the *all-tasks-connected* heuristic, which ensures that each non-initial task must have at least one other task as its cause. This means that every task that appears in the event log will remain in the final DG even if their causes are initially not accepted, because this heuristic will make sure that at least one cause relation is kept in the end.

Step 2. Mining of the splits/joins The mining of the split and join points of the DG refers to the identification of the input and output bindings for each task. These bindings help us identify the type of connection between a task and its neighbors. For example, a dependency graph may show that the outputs of task A are tasks B and C, but these connections may refer to an AND-split, where A is always followed by B and C, an OR-split, where A is followed by either B or C, or both.

The mining of the splits/joins relies on both the fully connected Dependency Graph obtained in the previous step and the traces of the event log. For example, if we have A as the cause relation for B and C in the DG, and the log traces indicate that task A is the nearest candidate that appears before B and C, we determine that their relations indicate an AND-split. In this step, we determine the input and output bindings of each task to find

the splits and joins of the graph.

Step 3. Mining long-distance relationships In this final step, the FHM identifies the dependencies that are not represented yet in the DG. The long-distance dependencies "indicate cases where task X depends indirectly on another task Y to be executed. That means that, in a split or join point, the choice may depend on choices made in other parts of the process model." [8].

Once the long-distance relationships are identified and represented in the DG, the execution of the FHM is complete, with the resulting DG representing the discovered process model.

Extensions to the Heuristic Miner

There are 2 relevant extensions presented for the heuristic miner algorithms that we should briefly discuss.

Additional Thresholds In [6], as part of the description of the heuristic miner algorithms, two additional thresholds are defined to remove the infrequent relations from the dependency graph. The first one is the *positive observations threshold* or *frequency threshold* for short, which helps us remove relations whose absolute frequency is lower than the threshold value. There is no default value for this threshold provided in [6], but ProM sets the default value at 0.1. The second threshold is the *bindings threshold*, which helps us remove those relations that represent an input or output binding with low frequency with respect to the binding with the highest frequency for that task. For example, if task A has two output bindings, B, with a frequency of 10, and the AND-split of BC with a frequency of 4, and the bindings threshold is set at 0.5, then the BC binding would be removed since $4/10 = 0.4$, which is lower than the threshold value. Again, there is no default value for this threshold in [6], but ProM sets it at 0.1.

Accepted-task-connected Heuristic [9], in their implementation of the heuristic miner, defines the *accepted-task-connected* heuristic, which impacts the way in which dependency relations are brought back for those tasks that ended with no cause or effect after the mining of the DG. This heuristic allows us to connect tasks with missing inputs or outputs by finding their best neighboring tasks (i.e. the neighboring tasks whose relation with the target task has the highest dependency measure). As opposed to the *all-tasks-connected* heuristic, a cause is found only for those activities that remain in

the DG, so those tasks that ended without neither a cause nor an effect are initially discarded, but they can be brought back if they represent the best neighbor for one of the remaining activities. Given this scenario, the process is repeated until all the activities in the dependency graph have a cause and an effect.

5.2.2 Defining the scope

Now that we have seen how the Flexible Heuristic Miner works and some of its relevant extensions, we must define the scope for our implementation in order to test if it is possible to generate a process model in our tool based on this algorithm.

To address the *Process discovery* activity of the Model Layer, we will use the FHM as our main reference. For the first step of the FHM, we will not only consider the 12 steps of their formal definition for the mining of the dependency graph, but we will also include an additional step to remove the infrequent relations based on the *frequency threshold*.

Then, instead of implementing the *all-tasks-connected* heuristic, we will make sure every task/activity is connected through the *accepted-task-connected*. This way, we can still have a resulting model where all the activities from the final DG are connected but we can still exclude those with infrequent input and output relations that may just be representing noise in the process. This also means that, in order for us to provide a connection for the activities that have either no input or output, we must keep a reference of all the original relations to find the best one and bring it back to the DG.

Then, once the remaining activities of the DG are set, we can create the :REPRESENTS edges between the Model_node nodes and the Class nodes, so our implementation must consider that as well at this stage of the algorithm.

Finally, we will implement the second step of the FHM to find the splits and joins through the identification of the input and output bindings of each activity. In addition, we will include the *bindings threshold* to remove the infrequent bindings from the final DG.

The third step of the FHM will be excluded from our implementation. This step, which mines the long-distance relations, is meant to identify connections between activities that are not yet represented in the event log [8], so we consider that this step is not essential to test the feasibility of being able to generate a process model that describes the behavior of the event log.

In addition to the implementation of this algorithm, we must also find a way to allow the users to set its parameters, execute it, and visualize the discovered process models through the user interface.

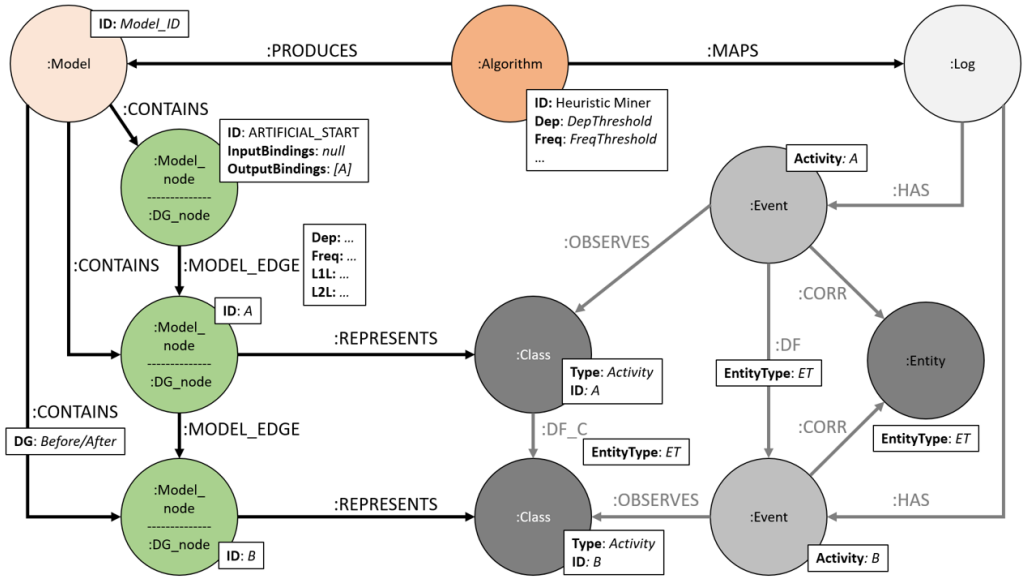


Figure 5.3: Example of how the discovered dependency graph connects with the rest of the graph data model.

Next, we present our implementation in 7 steps: (1) Defining the DG in the data model, (2) Setting up the initial DG, (3) Mining the dependency graph, (4) Connecting the activities, (5) Connecting the process model with the Class nodes, (6) Mining of the splits/joins, and (7) Creating and visualizing the process model.

5.2.3 Defining the Dependency Graph in the Data Model

Before describing our implementation of the algorithm, we must describe how we plan to represent the Dependency Graph, the main output of the FHM, using the extension to the graph data model presented in Section 5.1.3.

Figure 5.3 shows an example of how a process model discovered through our implementation of the Heuristic Miner algorithm is connected with the rest of the graph data model. At the top, in the middle, we can see the Algorithm node, where we specify the name of the algorithm ("Heuristic Miner") and all the thresholds (Dependency, Frequency, L1L, L2L, Relative-to-best, and Bindings) used as parameters. In this node we also store as properties the *Class* and the *DF* relations that will be used to define the model activities and their dependency relations respectively. Then, at the top-left, we can see the Model node, which contains only the ID as its property.

Figure 5.3 also shows how the Model node connects with every Model_node in the DG through a :CONTAINS relation. Since one of the steps in our

implementation requires us to bring back relations that were considered infrequent initially, we decided that it would be important to create two DG during runtime. The first DG will contain the initial DG, before any relation has been removed, and the second DG will contain the final DG with the remaining activities and relations. To distinguish between these two DGs, we define a runtime property for the :CONTAINS attribute in the form of the *DG* property. Every :CONTAINS relation between the Model node and the nodes that make up the initial DG will have "Before" as their property value, and every :CONTAINS relation between the Model node and the nodes that make up the final DG will have "After" as their property value. Once the initial DG is not needed anymore, it is deleted from the database.

Then, for the Model_node nodes in the left side of Figure 5.3, we define their second, more specific label, as *DG_node*, to specify that these particular nodes represent the tasks/activities of a Dependency Graph. The DG_nodes that represents the start and end of the process have "ARTIFICIAL_START" and "ARTIFICIAL_END" as their ID property. In addition to the ID that represents the activity, we add two more properties to all the DG_nodes in the form of the *InputBindings* and *OutputBindings*. These properties store the collection of bindings for every task. Every OR-split/join is represented by a new item in the collection and every AND-split/join is represented by an "|" character inside the item. For example, if activity A has an OR-split of two output bindings, B and an AND-split between C and D, the value for this property would be "OutputBindings: [B, C|D]".

Then, Figure 5.3 also shows the :MODEL_EDGE relations between the DG_nodes that represent the dependency relations of the DG. These relations have several properties that represent their dependency measures. These properties are the frequency ($|a >_W b|$), dependency ($|a \Rightarrow_W b|$), L1L dependency ($a \Rightarrow_W a$) and L2L dependency ($a \Rightarrow^2_W b$).

Finally, we can also see in Figure 5.3 that the DG_nodes are connected to the Class nodes that represent the same activity through the :REPRESENTS relation, creating an indirect connection between the process model activities and its corresponding Event nodes.

Now that we have defined the planned structure for the DG, we can start to look at the queries that generate this structure.

5.2.4 Setting Up the Initial Dependency Graph

The first queries that we execute generate the Algorithm, Model, and DG_node nodes together with their relations between them and to other nodes in the database (i.e. the Log and Class nodes).

Create Algorithm and Model nodes

The first query we execute is shown below. This query creates the Algorithm and Model nodes. The Algorithm node is connected to the Log node and the Model node, and is used to store the parameters to run the model.

```
1 MATCH (l:Log{ID:"filename.csv"})
2 MERGE (l)<-[:MAPS]-(:Algorithm{ID:"Heuristic Miner",Class:"
    ClassType",DF:"EntityType",Freq:FreqThreshold,Dep:DepThreshold,
    L1L:L1LThreshold,L2L:L2LThreshold,Rel:RelThreshold,Bind:
    BindThreshold})-[:PRODUCES]->(:Model{Algorithm:"Heuristic
    Miner",Log:"filename.csv",ID:"ModelID"})
```

Create DG_node nodes

Then, we create the DG_nodes using two queries. The first query creates the DG_nodes based on the class type passed as argument. In line 2, we retrieve the corresponding Class nodes. In line 3, we create and connect the DG_nodes with the Model node, specifying in the :CONTAINS relation that these nodes represent the DG *before* the mining. In line 4, we connect the DG_nodes with their corresponding Class node. Then, the second query creates the artificial Start and End DG_nodes. In lines 8 and 9, we create and connect the Start and End DG_nodes respectively. To easily distinguish these nodes from the rest in the queries, we added the properties *isStart* and *isEnd* respectively.

```
1 //Query 1
2 MATCH (m:Model{ID:"ModelID"})--(:Algorithm)--(:Log)--(:Event)
    --(c:Class{Type:"ClassType"})
3 MERGE (m)-[:CONTAINS{DG:"Before"}]->(dg:DG_node:Model_node{ID
    :c.ID})
4 MERGE (dg)-[:REPRESENTS]->(c)
5
6 //Query 2
7 MATCH (m:Model{ID:"ModelID"})
8 MERGE (m)-[:CONTAINS{DG:"Before"}]->(:DG_node:Model_node{ID:"
    ARTIFICIAL_START",isStart:True})
9 MERGE (m)-[:CONTAINS{DG:"Before"}]->(:DG_node:Model_node{ID:"
    ARTIFICIAL_END",isEnd:True})
```

Create :MODEL_EDGE relations

After the DG_nodes have been created, we create the :MODEL_EDGE relations between them. The first query shown below creates the :MODEL_EDGE relations between model activities. In lines 2 and 3 we match two DG_nodes

with their corresponding Class nodes. In line 4 we make a match to keep only those pairs of DG_nodes whose classes are connected by a :DF_C relation of the *EntityType* passed as argument. Since we also need to compute the frequency with which these relations occur in the event log, in line 5 we find all the :DF relations between events related to the Class nodes. In line 6, we count these :DF relations. Finally, in line 7 we connect the DG_nodes through a :MODEL_EDGE relation, setting the frequency of this relation as a property. Then, the second query shown below creates the :MODEL_EDGE relations between the artificial Start node and the rest of the DG_nodes. In line 10, we match the DG_nodes with their Event nodes. In line 11, we find those events that do not have an ingoing :DF relation, which represent the start activities. To also register the frequency for this relation, in line 12 we count the events with no ingoing :DF, grouped by activity. Finally, in line 13 we retrieve the DG_node that represents the artificial Start and in line 14 we connect it with the DG_nodes that represent starting activities, setting the frequency as a property of their :MODEL_EDGE relation.

```
1  //Query 1
2  MATCH (m:Model{ID:"ModelID"})-->(dg:DG_node)-->(c:Class)
3  MATCH (m)-->(dg2:DG_node)-->(c2:Class)
4  MATCH (c)-[:DF_C{EntityType:"EntityType"}]->(c2)
5  MATCH (c)<--(:Event)-[df:DF{EntityType:"EntityType"}]->(:Event)
       -->(c2)
6  WITH DISTINCT dg AS From, dg2 AS To, COUNT(df) AS f
7  MERGE (From)-[:MODEL_EDGE{Freq:f}]->(To)
8
9  //Query 2
10 MATCH (m:Model{ID:"ModelID"})--(dg:DG_node)--(:Class)--(e:
       Event)
11 WHERE NOT EXISTS (()-[:DF{EntityType:"EntityType"}]->(e))
12 WITH m, dg AS startActivity, COUNT(e) AS f
13 MATCH (m)--(st:DG_node{isStart:True})
14 MERGE (st)-[:MODEL_EDGE{Freq:f}]->(startActivity)
```

A third query is also executed to generate the :MODEL_EDGE relations between the model activities and the artificial End node. This query is similar to the second query shown above (see Appendix C).

Calculate Dependency Measures

Now that the DG_nodes are connected, we can compute their dependency measures (absolute, L1L and L2L dependencies) using the equations described in Section 5.2.1. We define one query per measure.

The first query calculates the absolute dependency. In line 1, we retrieve the dependency relation *me* between two model activities. In line 2, we check

if there is also an inverted dependency relation between the same activities. If there is no such relation, we use the COALESCE function in line 3 to set the *fB* frequency value as 0 instead of null. Then, in line 4 we compute the dependency with the (Absolute) Dependency equation from Section 5.2.1 to finally set the computed value in line 6.

```
1 MATCH (m:Model{ID:"ModelID"})--(a:DG_node)-[me:MODEL_EDGE]->(b
    :DG_node)
2 OPTIONAL MATCH (b)-[me2:MODEL_EDGE]->(a)
3 WITH DISTINCT a, b, me, me.Freq AS fA, COALESCE(me2.Freq,0)
    AS fB
4 WITH a,b,ABS(ROUND(((fA-fB)*1.0/(fA+fB+1)),3)) AS dep
5 MATCH (a)-[me:MODEL_EDGE]->(b)
6 SET me.Dep = dep
```

Then, the second query calculates the L1L dependency. In line 1, we find DG_nodes that have a self-referencing :MODEL_EDGE, and for those that do, we set the L1L property using the L1L dependency equation.

```
1 MATCH (m:Model{ID:"ModelID"})--(a:DG_node)-[me:MODEL_EDGE]-(a)
2 SET me.L1L = ROUND((me.Freq*1.0)/(me.Freq+1),3)
```

Finally, the third query calculates the L2L dependency. In line 1, we find those DG_nodes that have an "A->B->A" pattern in the DG. Then, to compute the frequency with which $a >>_W b$ occurs, we find the events e1, e2, and e3 that have a path that also describes the "A->B->A" pattern. In lines 5 and 6, we find the events related to the classes that represents activities "A" and "B" respectively. In line 7, we check if these events describe the pattern through the :DF relations of the entity type passed as argument. In line 8, we count the frequency with which this pattern occurs in the event log. Then, in lines 9-12 we use a similar approach, this time to find the pattern "B->A->B" so we can find the frequency with which $b >>_W a$ occurs. Now that we have the required parameters, we can use the L2L dependency equation in line 13 to calculate the dependency measure. Finally, this value is set as a property for the :MODEL_EDGE relation in line 15.

```
1 MATCH (m:Model{ID:"ModelID"})--(dg:DG_node)-[:MODEL_EDGE]->(
    dg2:DG_node)-[:MODEL_EDGE]->(dg)
2 WITH DISTINCT m, dg, dg2
3 MATCH (m)--(dg)--(c:Class)
4 MATCH (m)--(dg2)--(c2:Class)
5 MATCH (e1:Event)--(c)--(e3:Event)
6 MATCH (c2)--(e2:Event)
7 OPTIONAL MATCH (e1)-[df:DF{EntityType:"EntityType"}]->(e2)-[:DF
    {EntityType:"EntityType"}]->(e3)
8 WITH m,dg,dg2,c,c2,COUNT(df) AS l2lFreqA
9 MATCH (e4:Event)--(c2)--(e6:Event)
10 MATCH (c)--(e5:Event)
```

```
OPTIONAL MATCH (e4)-[df2:DF{EntityType:"EntityType"}]->(e5)-[:
    DF{EntityType:"EntityType"}]->(e6)
WITH m,dg,dg2,l2lFreqA,COUNT(df2) AS l2lFreqB
WITH m,dg,dg2,ROUND(((l2lFreqA+l2lFreqB)*1.0)/(l2lFreqA+
    l2lFreqB+1),3) AS l2lDep
MATCH (m)--(dg)-[me:MODEL_EDGE]->(dg2)
SET me.L2L = l2lDep
```

Now that the initial DG and the dependency measures have been stored in the database, we can describe the query that executes the first step of the FHM, the mining of the DG.

5.2.5 Mining the Dependency Graph

[8] provides the formal definition for the mining of the DG. This definition consists of 12 steps indicating how to obtain the dependency graph. The definition of the 12 steps, as presented in [8], is:

Dependency Graph (DG)-algorithm *Let W be an event log over a finite set of tasks T, W^+ an extension of W including start/end tasks, σ_a the (absolute) Dependency Threshold (default 0.9), σ_{L1L} the Length-one-loops Threshold (default 0.9), σ_{L2L} the Length-two-loops Threshold (default 0.9), and σ_r the Relative-to-best Threshold (default 0.05). $DG(W^+)$ (i.e., the dependency graph for W^+) is defined as follows.*

1. $T = \{t | \exists_{\sigma \in W^+}[t \in \sigma]\}$ *(the set of tasks appearing in the log),*
2. $C_1 = \{(a, a) \in T \times T | a \Rightarrow_W a \geq \sigma_{L1L}\}$ *(length-one loops),*
3. $C_2 = \{(a, b) \in T \times T | (a, a) \notin C_1 \wedge (b, b) \notin C_1 \wedge a \Rightarrow_{2W} b \geq \sigma_{L2L}\}$ *(length-two loops),*
4. $C_{out} = \{(a, b) \in T \times T | b \neq End \wedge a \neq b \wedge \forall_{y \in T}[a \Rightarrow_W b \geq a \Rightarrow_W y]\}$ *(for each task, the strongest follower),*
5. $C_{in} = \{(a, b) \in T \times T | a \neq Start \wedge a \neq b \wedge \forall_{x \in T}[a \Rightarrow_W b \geq x \Rightarrow_W b]\}$ *(for each task, the strongest cause),*
6. $C'_{out} = \{(a, x) \in C_{out} | (a \Rightarrow_W x) < \sigma_a \wedge \exists_{(b,y) \in C_{out}}[(a, b) \in C_2 \wedge ((b \Rightarrow_W y) - (a \Rightarrow_W x)) > \sigma_r]\}$ *(the weak outgoing-connections for a length-two loop),*
7. $C_{out} = C_{out} - C'_{out}$ *(remove the weak connections),*

8. $C'_{in} = \{(x,a) \in C_{in} | (x \Rightarrow_W a) < \sigma_a \wedge \exists_{(y,b)\in C_{in}}[(a,b) \in C_2 \wedge ((y \Rightarrow_W b) - (x \Rightarrow_W a)) > \sigma_r]\}$ *(the weak incoming-connections for a length-two loop),*

9. $C_{in} = C_{in} - C'_{in}$ *(remove the weak connections),*

10. $C''_{out} = \{(a,b) \in T \times T | (a \Rightarrow_W b) \geq \sigma_a \vee \exists_{(a,c)\in C_{out}}[((a \Rightarrow_W c) - (a \Rightarrow_W b)) < \sigma_r]\}$,

11. $C''_{in} = \{(b,a) \in T \times T | (b \Rightarrow_W a) \geq \sigma_a \vee \exists_{(b,c)\in C_{in}}[((b \Rightarrow_W c) - (b \Rightarrow_W a)) < \sigma_r]\}$,

12. $DG = C_1 \cup C_2 \cup C''_{out} \cup C''_{in}$.

As mentioned in Section 5.2.2, the Cypher query that mines the DG will not only execute these 12 steps, but it will also execute one additional step to remove relations based on the *frequency threshold*.

The 13 total steps are executed in one large query, whose main structure can be observed below. The query does not need to explicitly execute Step 1 of the mining of the DG since the set of tasks appearing in the log is already stored in the DG_nodes of the initial DG. The other 12 steps are chained together with the use of the *WITH* command to pass the intermediate results from one step to the next (the full query is included in Appendix C).

In line 4 of the query below, we can see how we store the result from Step 2 in variable *C1*. Then, we can see an example of how we deal with null results in line 5, where we check whether there are any values stored in *C1*, if not, we define an empty collection for the variable. A similar process is executed for the results of Step 3 in lines 10 and 11. Then, in lines 16 and 21 we store the results of Steps 4 and 5 respectively, and we can see that we keep chaining the new variables that will be used in upcoming steps. Step 7 asks us to remove the weak connections found in Step 6 (line 26) from the results stored in *Cout*, which is why in line 30 we use an *apoc* function to subtract these connections. We execute a similar process in Steps 8 and 9 to remove the weak connections from *Cin* (lines 35 and 39). Then, for Step 10, we need to find connections based on two conditions. The results from the first condition are stored in *Cout2_1* (line 44) and the results of the second condition are stored in *Cout2_2* (line 46). Finally, these results are combined in one variable, *Cout2* (line 48). We execute a similar process for Step 11 in lines 53, 55, and 57. For Step 12, we combine the results in variable *dgEdges* (line 61), completing the 12 steps defined in the FHM. Then, for the extra step, we store the connections below the frequency threshold in *Cfreq* (line 66) and remove them from the connections stored in *dgEdges* (line 67).

Finally, the remaining connections are used to create the DG_nodes that make up the second DG, which are connected to the Model node through :CONTAINS relations that specify the *"DG:After"* property (lines 71-76).

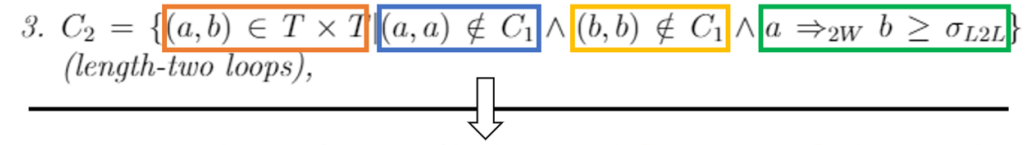
```
1  ...
2      // FHM Step 2
3      // (Pairs with a Length-1 loop connection above the L1L threshold are
       stored in C1)
4  WITH COLLECT(DISTINCT [a.ID,a.ID]) AS C1
5  WITH CASE WHEN C1[0][0] IS NULL THEN [] ELSE C1 END AS C1
6  ..
7
8      // FHM Step 3
9      // (Pairs with a Length-2 loop connection above the L2L threshold are
       stored in C2)
10 WITH C1,COLLECT(DISTINCT [a.ID,b.ID]) AS C2
11 WITH C1, CASE WHEN C2[0][0] IS NULL THEN [] ELSE C2 END AS C2
12 ...
13
14     // FHM Step 4
15     // (The pairings of each task with its strongest follower are stored in
       Cout)
16 WITH C1,C2, COLLECT(strPairs) AS Cout
17 ...
18
19     // FHM Step 5
20     // (The pairings of each task with its strongest cause are stored in Cin
       )
21 WITH C1,C2,Cout, COLLECT(strPairs) AS Cin
22 ...
23
24     // FHM Step 6
25     // (Pairs representing weak outgoing-connections for a length-two loop
       are stored in Cout1)
26 WITH C1,C2,Cout,Cin,CASE WHEN Cout1[0][0] IS NULL THEN [] ELSE Cout1 END AS
       Cout1
27
28     // FHM Step 7
29     // (Pairs representing weak connections are removed)
30 WITH C1,C2,Cin,apoc.coll.subtract(Cout, Cout1) as Cout
31 ...
32
33     // FHM Step 8
34     // (Pairs representing weak incoming-connections for a length-two loop
       are stored in Cin1)
35 WITH C1,C2,Cout,Cin,CASE WHEN Cin1[0][0] IS NULL THEN [] ELSE Cin1 END AS
       Cin1
36
37     // FHM Step 9
38     // (Pairs representing weak connections are removed)
39 WITH C1,C2,Cout,apoc.coll.subtract(Cin, Cin1) as Cin
40 ...
41
42     // FHM Step 10
43     // (Pairs representing outgoing-connections above the absolute
       dependency threshold, stored Cout2_1, together with pairs representing
       outgoing-connections under the relative-to-best threshold, stored in
       Cout2_2, are stored in Cout2)
44 WITH C1,C2,Cin,Cout,COLLECT(DISTINCT [a.ID,b.ID]) AS Cout2_1
45 ...
46 WITH C1,C2,Cin,Cout2_1,COLLECT(DISTINCT [a.ID,b.ID]) AS Cout2_2
```

```
47 ...
48 WITH C1,C2,Cin,Cout2_1+Cout2_2 AS Cout2
49 ...
50
51     // FHM Step 11
52     // (Pairs representing incoming-connections above the absolute
       dependency threshold, stored Cin2_1, together with pairs representing
       incoming-connections under the relative-to-best threshold, stored in
       Cin2_2, are stored in Cin2)
53 WITH C1,C2,Cout2,Cin,COLLECT(DISTINCT [b.ID,a.ID]) AS Cin2_1
54 ...
55 WITH C1,C2,Cout2,Cin2_1,COLLECT(DISTINCT [b.ID,a.ID]) AS Cin2_2
56 ...
57 WITH C1,C2,Cout2,Cin2_1+Cin2_2 AS Cin2
58
59     // FHM Step 12
60     // (The connections stored in C1, C2, Cout2, and Cin2 are stored in
       dgEdges, representing the final connections of the DG based on Step 1 of
        the FHM)
61 WITH C1+C2+Cout2+Cin2 AS dgEdges
62 ...
63
64     // Step 13
65     // (Connections below the frequency threshold, stored in Cfreq, are
       removed)
66 WITH dgEdges,COLLECT(DISTINCT [a.ID,b.ID]) AS Cfreq
67 WITH apoc.coll.subtract(dgEdges, Cfreq) as dgEdges
68 ...
69
70     // (Every connection stored in dgEdges is stored in the database)
71 UNWIND dgEdges AS dgEdge
72 WITH dgEdge
73 MATCH (m:Model{ID:"ModelID"})
74 MERGE (m)-[:CONTAINS{DG:"After"}]->(a:DG_node:Model_node{ID:dgEdge[0]})
75 MERGE (m)-[:CONTAINS{DG:"After"}]->(b:DG_node:Model_node{ID:dgEdge[1]})
76 MERGE (a)-[:MODEL_EDGE]->(b)
```

Figures 5.4 and 5.5 show two examples that describe how the steps from the formal definition were translated into Cypher queries.

Figure 5.4 shows our translation of Step 3 in the formal definition, where the algorithm requires us to obtain all the pairs of activities involved in a length-2 loop. The formal definition can be observed at the top of Figure 5.4. To make the translation of this step, we need to make sure that neither activity in the pair is already involved in a length-1 loop (stored in variable *C1*) and that the length-2 dependency value for the connection between activities is above the L2L threshold. The Cypher query adapted based on these requirements can be observed at the bottom of Figure 5.4. In line 1, we first retrieve the pairs of connected activities. Then, in line 2, we check for the conditions specified in the formal definition. Since we could have an event log with no activities involved in a length-2 loop, we make the match optional. If there are no matches, causing us to store null values in the *C2* variable in line 3, we set an empty collection as its value in line 4.

Then, Figure 5.5 shows our translation of Step 10 in the formal definition,



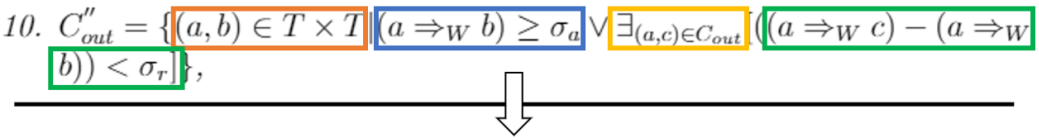

```
1 OPTIONAL MATCH (m:Model{ID:"HM_1"})--(a:DG_node)-[me:MODEL_EDGE]->(b:DG_node)
2 WHERE NOT [a.ID,a.ID] IN C1 AND NOT [b.ID,b.ID] IN C1 AND me.L2L >= L2LThreshold
3 WITH C1,COLLECT(DISTINCT [a.ID,b.ID]) AS C2
4 WITH C1, CASE WHEN C2[0][0] IS NULL THEN [] ELSE C2 END AS C2
```

Figure 5.4: Cypher query to obtain the result from Step 3 in the FHM formal definition.

where we need to find all those pairs of activities *[a,b]* whose connection meets either one of two conditions: (1) their dependency value is above the threshold or (2) their connection has a dependency value inside the *Relative-to-best* threshold when compared against the strongest outgoing connection of *a*. In line 1, we retrieve the pairs of connected activities to check the first condition. In line 2 we do the check by identifying those pairs whose connection has a dependency value above the Dependency threshold passed as argument. In line 3 we store these pairs in a variable *Cout2_1*. In line 4, we retrieve the *[a,b]* and *[a,c]* pairs of activities needed to check the second condition. In line 5, we first check that the pair *[a,c]* is stored in *Cout* (which stores the strongest outgoing connections for each activity) and then we check if the difference between the dependency measures of *[a,c]* and *[a,b]* are below the relative-to-best threshold passed as argument. In line 6, we store these second set of pairs in variable *Cout2_2*. In line 7, we check if *Cout2_2* has null values so they can be replaced by an empty collection. Finally, in line 8, we combine the results from both conditions to obtain the final result of Step 10 in variable *Cout2*.

After the 12 steps from the formal definition of the DG were implemented in the query following a similar approach, we added the final step that removes the connections whose frequency is below the *frequency threshold*.

The section of the Heuristic Miner query that removes the connections based on their frequency is shown below. In line 1, we have the final connections, resulting from the execution of the 12 steps of the FHM, stored in variable *dgEdges*. In lines 2 and 3, we make a path from the Model node to the Entity nodes to find the amount of existing Entity nodes for the entity type used to create the model. In line 4, we store this number in variable *numEntities*. In line 5, we retrieve the pairs of nodes connected through a :MODEL_EDGE relation. In line 6, we filter to keep those pairs whose frequency measure (stored in the :MODEL_EDGE relation), divided by the total amount of entity nodes, is lower than the frequency threshold. In line




```
1 MATCH (m:Model{ID:"HM_1"})--(a:DG_node)-[me:MODEL_EDGE]->(b:DG_node)
2 WHERE me.Dep >= DepThreshold
3 WITH C1,C2,Cin,Cout,COLLECT(DISTINCT [a.ID,b.ID]) AS Cout2_1
4 OPTIONAL MATCH (m:Model{ID:"HM_1"})--(a:DG_node)-[me:MODEL_EDGE]->(b:DG_node),
        (a)-[me2:MODEL_EDGE]->(c:DG_node)
5 WHERE [a.ID,c.ID] IN Cout AND ABS(me2.Dep - me.Dep) < RelThreshold
6 WITH C1,C2,Cin,Cout2_1,COLLECT(DISTINCT [a.ID,b.ID]) AS Cout2_2
7 WITH C1,C2,Cin,Cout2_1,CASE WHEN Cout2_2[0][0] IS NULL THEN []
        ELSE Cout2_2 END AS Cout2_2
8 WITH C1,C2,Cin,Cout2_1+Cout2_2 AS Cout2
```

Figure 5.5: Cypher query to obtain the result from Step 10 in the FHM formal definition.

7 we store the pairs below the threshold in variable *Cfreq*. Finally, in line 8, we subtract those infrequent pairs from the remaining relations of the DG.

```
1 WITH COLLECT(DISTINCT dgEdge) AS dgEdges
2 MATCH (m:Model{ID:"ModelID"})--(:DG_node)--(:Class)--(:Event)
    --(n:Entity)
3 WHERE n.EntityType = "EntityType"
4 WITH dgEdges,COUNT(DISTINCT n) AS numEntities
5 OPTIONAL MATCH (m:Model{ID:"ModelID"})--(a:DG_node)-[me:
    MODEL_EDGE]->(b:DG_node)
6 WHERE [a.ID,b.ID] IN dgEdges AND (me.Freq*1.0/numEntities) <
    FreqThreshold
7 WITH dgEdges,COLLECT(DISTINCT [a.ID,b.ID]) AS Cfreq
8 WITH apoc.coll.subtract(dgEdges, Cfreq) as dgEdges
```

As mentioned earlier, the remaining pairs stored in variable *dgEdges* are used to create the second DG, whose DG_nodes are connected to the Model node through the :CONTAINS relations, now with the *"DG:After"* property.

Once this second DG is created, the first step of the FHM is complete. In the next step of our implementation, we make sure all the remaining activities in the DG are connected to at least one input and one output using the *accepted-task-connected* heuristic.

5.2.6 Connecting all activities

After generating the dependency graph by removing the most infrequent relations, we make sure that all the remaining activities have at least one input and one output by implementing the *accepted-task-connected* heuristic.

To apply the heuristic for the model activities with no outputs, we first

run the query below to check if there is at least one activity with this situation. First, we find the DG_nodes that only have an outgoing :MODEL_EDGE relation to themselves. In line 2, we retrieve the DG_nodes. In line 3, we count the number of outgoing :MODEL_EDGE relations for each node. In lines 4 and 5, we check whether the DG_nodes have only one outgoing connection pointing to themselves. In line 6, we define the UNION to join these results with a second query. In the second query, we find the DG_nodes that do not have any outgoing :MODEL_EDGE relation. In line 7, we retrieve the nodes, and in line 8 we filter to keep those that do not have any outgoing :MODEL_EDGE relation and are not the artificial End node. Finally, based on the joint result (stored in variable *a* in line 9), we return the number of nodes in line 10.

```
1  CALL{
2      MATCH (m:Model{ID:"ModelID"})-[:CONTAINS{DG:'After'}]-(a:
       DG_node)-[me:MODEL_EDGE]->()
3      WITH a,COUNT(me) AS numEdges
4      MATCH (a:DG_node)-[me:MODEL_EDGE]->(b)
5      WHERE a = b AND numEdges = 1 RETURN a
6      UNION
7      MATCH (m:Model{ID:"ModelID"})-[:CONTAINS{DG:'After'}]-(a:
       DG_node)
8    WHERE NOT (a)-[:MODEL_EDGE]->() AND a.isEnd IS NULL RETURN
       a
9  } WITH a
10 RETURN COUNT(a) AS numNodes
```

The number of nodes is read in Java and, if the value is higher than 0, we execute the query below, which creates the outgoing connections. In line 5, variable *a* has the collection of DG_nodes with no outputs. In line 6, We retrieve the :MODEL_EDGE relations from the original DG, which we can identify through the *"DG:Before"* property of the :CONTAINS relations that connect the original DG with the Model node. In line 7, we match the nodes with the missing outputs (stored in *a*), with the nodes from the original DG (stored in *b*). In line 8, we store the ID of the target nodes (stored in *c*) in variable *cID*, and the dependency measures of their relation with *b* in variable *depVal*. Then, in line 9, for each node *a* with missing outputs, we use an *apoc* function to find the *cID* with the highest dependency value. The *cID*s that represent the activity with the highest dependency relation are stored in a collection in variable *items* (line 10). Then, we retrieve the cID from the collection and rename them as *out* (line 11). Then, in line 13 we check if a DG_node from the second DG (identified through the *"DG:After"* property of the :CONTAINS relation) contains a DG_node with the same ID as the one with the best dependency relation. In line 15, we check if the second DG

does not contain such node, in which case we create a new DG_node with setting its ID property with the value from variable *out* (line 16) and then we connect this new node to the Model node (line 17). In case the DG does contain the node, we simply create the :MODEL_EDGE relation between the nodes (line 18).

```
1   // First, we find the model activities whose only output is themselves
2   // Then, we find the model activities, different from the artificial End,
      that have no outputs.
3   // The DG_nodes that represent these activities are stored in variable "a"
4  ...
5  WITH a
6  MATCH (m:Model{ID:"ModelID"})-[:CONTAINS{DG:'Before'}]-(b:DG_node)-[me:
       MODEL_EDGE]->(c:DG_node)
7  WHERE b.ID = a.ID AND b <> c
8  WITH m,a,c.ID AS cID,me.Dep AS depVal
9  WITH m,a,apoc.agg.maxItems(cID,depVal) AS strFollowers
10 WITH m,a,strFollowers.items AS items
11 UNWIND items AS out
12 WITH m,a,out
13 OPTIONAL MATCH (m)-[:CONTAINS{DG:'After'}]-(b:DG_node{ID:out})
14 CALL apoc.do.when(
15   b IS NULL,
16     "MERGE (a)-[:MODEL_EDGE]->(c:DG_node:Model_node{ID:out})
17     MERGE (m)-[:CONTAINS{DG:'After'}]->(c)",
18     "MERGE (a)-[:MODEL_EDGE]->(b)",
19     {m:m,a:a,b:b,out:out}
20 ) YIELD value RETURN 1
```

Since the previous query can add new nodes to the DG, it is possible that new nodes are missing an output, which is why after this query is executed, we execute the first query to check once again the number of nodes that have no outputs. If there are still nodes with this situation, we execute again the second query, and we remain in that loop (coded in Java) until all the activities in the second DG have at least one output.

To add the missing inputs, we created a similar pair of queries to those presented in this Section, both of which can be observed in Appendix C.

After executing these queries, ensuring that all the activities in the final DG have at least one input and one output, the initial DG, connected to the Model node through the :CONTAINS relations that have the *"DG:Before"* property, can be deleted.

Now that the final set of model activities has been identified, we can make the final connections between the model activity nodes and the Class nodes.

5.2.7 Connecting the Model with the Class nodes

After the mining of the DG, where we generated a second DG with the final set of activities and dependency relations, we delete the original DG generated as an input for the execution of the algorithm. However, deleting these

nodes also means that we have deleted the initial :REPRESENTS relations made between the DG_nodes and the Class nodes. Therefore, it is important to reestablish them with the final DG, so we can keep the link between the process model and the events.

The query that creates this connection is shown below. In line 1, we retrieve the Class nodes used to generate the DG. In line 2, we retrieve the final set of DG_nodes. Finally, in line 3 we create the :REPRESENTS relations between them.

```
1 MATCH (m:Model{ID:"ModelID"})--(:Algorithm)--(:Log)--(:Event)
    --(c:Class{Type:"ClassType"})
2 MATCH (m)-[:CONTAINS{DG:"After"}]->(dg:DG_node{ID:c.ID})
3 MERGE (dg)-[:REPRESENTS]->(c)
```

Now that we have connected all the activities that make up the final DG, we can execute the second step of the FHM, where we identify the split and join points for each model activity.

5.2.8 Mining of the splits/joins

The mining of the split and join points of the DG refers to the identification of the input and output bindings for each activity. This will help us identify the type of dependency relation between an activity and its neighbors.

From Section 5.2.1, we can recall that the mining of the splits and joins depends on both the fully connected DG, which in our case is represented by the DG_nodes created in the previous step, and the traces of the event log, which in our case are represented by the Event nodes connected between them by the :DF relations of a specific entity type. Therefore, we must use these two inputs to identify the bindings of each activity and store them in the DG_nodes as described in Figure 5.3. We should also mention that the mining of the splits and joins must make sure that every remaining dependency relation in the DG is represented in at least one output binding, so we must also ensure that every :MODEL_EDGE relation is still considered independently of the bindings threshold.

To find the input and output bindings that define the splits and joins of the DG and store them in the database, we built two Cypher queries, one to find the input bindings and one to find the output bindings.

The main steps followed to define the final output bindings for all the activities in the DG are shown next with the most relevant parts of the Cypher query. The full query can be observed in Appendix C.

The steps to find and store the output bindings of the DG are the following:

1. First we make a union to obtain the output bindings of the model activities considering three scenarios: (1) the regular output bindings of model activities, (2) the output bindings of the model activities connected to the artificial End node. (3) the output bindings of the artificial Start node. We need to consider these three scenarios separately because the artificial nodes are not connected to the Class nodes, so the query to find the bindings differs. The steps to find the output bindings in the first scenario are the following:

   (a) For each activity in the DG, represented by event i in the log, find all the events j that appear later on in the :DF paths (lines 3-6). We store these events in the variable *eventsCaused* (line 7). In line 7, we also use variable *classType* to store the class type of the Class nodes connected with the model activities.

   (b) Then, remove those events j whose cause is not i. We determine that the cause is not i if an intermediate event k exists between i and j in the :DF path and the DG contains a :MODEL_EDGE relation between the model activities of k and j (lines 8-10). These events are stored in variable *eventsOtherCause* (line 11) and are subtracted from the collection in *eventsCaused* (line 12). The remaining events are stored in variable *eventList*.

   (c) Then, for every event i, group the events (stored in *eventList*) based on their property that was used to define the class type (lines 13-14). Each grouping of these properties represents an output binding and is stored in variable *oB*. In line 14, we add a variable n with value of 1 to help us count the frequency of each binding in the next step.

   (d) Finally, use the class type property of every event i (stored in variable *mActivity*), to group the output bindings and calculate the total frequency with which they occur in the event log (line 15). This frequency is stored in variable *bindFreq*.

   (e) We follow a similar process for the second and third scenarios, which can be observed in the full query in Appendix C.

2. Now that we have identified the existing output bindings for each model activity, in line 26 we identify the highest frequency among its bindings. This value is stored in variable *freqMax* and is used to compare the frequency of the bindings agains the threshold in a subsequent step. Also in line 26, we use variable *bindingsDetails* to store the collection of output bindings and their frequencies for each model activity.

3. Next, for each model activity, we collect all its outputs in the DG (lines 28-30) in variable *allOutputs*. We do this to have a reference of all the outputs and make sure that all of them are represented in at least one binding.

4. Then, for each model activity we check whether they have one or more output bindings (lines 31-32).

5. Since every model activity must have at least one binding (except for the End node), if an activity has one binding, it is kept automatically (lines 34-35). The binding is stored in variable *outputBindings*. In case the binding is composed of more than one activity, the activities are joined with the help of a '|' character (e.g. "B" and "C" are stored as "B|C".

6. If an activity has more than one binding, we make a union of two results: (1) For every output of every model activity (stored in *allOutputs*), we retrieve the activity binding with the highest frequency that contains this output. This way, we can make sure that every output is represented in the final bindings even if its best binding is below the frequency threshold. (2) For each model activity, we retrieve the output bindings above the dependency threshold. The steps to obtain the second result are the following:

   (a) For every output binding of each model activity (which we retrieve in lines 43-45), we check whether their frequency, divided by the highest frequency of the activity's bindings (stored in *freqMax*), is above the bindings threshold (line 46). The frequency of each binding is stored in the second slot of variable *bindDetails* and the threshold is a parameter of the query.

   (b) Since this result is not final (it is only part of the union), we store the bindings above the threshold in a new variable called *oBindings*.

7. The union of the results for activities that have more than one binding is now stored in variable *oBindings* in line 50. To remove duplicates between the two results, we unwind the collections and collect the bindings again to remove the duplicates using the DISTINCT function (lines 51-52). The distinct bindings for each activity are stored in variable *outputBindings*.

8. Finally, the output bindings for each activity are stored as a collection in property *OutputBindings* (lines 56-58). Each item in the collection

represents an OR-split, and each '|' character in each item represents an AND-split.

```
CALL{
  // Retrieve output bindings of every model activity
  MATCH (m:Model{ID:"ModelID"})-[:CONTAINS]-(a:DG_node)-[:MODEL_EDGE]->(b:DG_node)
  MATCH (a)--(cl:Class)--(i:Event)
  MATCH (b)--(:Class)--(j:Event)
  MATCH (i)-[:DF*{EntityType:"EntityType"}]->(j)
  WITH DISTINCT m, cl.Type AS classType, i, COLLECT(j) AS eventsCaused
  MATCH (m)-[:CONTAINS]-(c:DG_node)-[:MODEL_EDGE]->(b:DG_node)
  OPTIONAL MATCH (i)-[:DF*{EntityType:"DeliveryOrder"}]->(k:Event)-[:DF*{EntityType:"DeliveryOrder"}]->(j:Event)
  WHERE EXISTS((c)--(:Class)--(k)) AND EXISTS((b)--(:Class)--(j)) AND j IN eventsCaused
  WITH i, classType, eventsCaused, COLLECT(DISTINCT j) AS eventsOtherCause
  WITH i, classType, apoc.coll.subtract(eventsCaused, eventsOtherCause) AS eventList
  UNWIND eventList AS event
  WITH DISTINCT i, classType, apoc.coll.sort(COLLECT(DISTINCT event[classType])) AS oB, 1 AS n
  RETURN DISTINCT i[classType] AS mActivity, oB, SUM(n) AS bindFreq ORDER BY bindFreq DESC
  UNION
  ...
  // Retrieve output bindings of model activities connected to the artificial End node.
  ...
  UNION
  ...
  // Retrieve output bindings of artificial Start activity.
  ...
}
// For every mActivity, obtain the frequency with the highest value from its output bindings and store the bindings with its frequency in bindingsDetails.
WITH mActivity, apoc.agg.maxItems(mActivity, bindFreq).value AS freqMax, COLLECT([oB, bindFreq]) AS bindingsDetails
// Collect all the output activities of each mActivity.
MATCH (m:Model{ID:"ModelID"})-[:CONTAINS]-(a:DG_node)-[:MODEL_EDGE]->(b:DG_node)
WHERE a.ID = mActivity
WITH DISTINCT mActivity, bindingsDetails, freqMax, COLLECT(b.ID) AS allOutputs
CALL apoc.when(
  SIZE(bindingsDetails) = 1,
  // If a model activity has one binding, it is kept without checking against the threhsold.
    "UNWIND bindingsDetails AS bindDetails
    RETURN mActivity AS mAct,COLLECT(apoc.text.join(bindDetails[0],'|')) AS outputBindings",
  // If the model has more than one output binding, we make a UNION of two results:
    "CALL{
    ...
    // (1) For each output of every mActivity, we retrieve the binding with the highest frequecy that includes such output.
    ...
    UNION
    // (2) For each mActivity, we retrieve its bindings above the threshold.
```

```
43        WITH mActivity, bindingsDetails, freqMax
44          UNWIND bindingsDetails AS bindDetails
45          WITH mActivity,freqMax,bindDetails
46          WHERE (bindDetails[1]*1.0)/freqMax >= BindingsThreshold
47          RETURN mActivity AS mAct,COLLECT(DISTINCT apoc.text.join(bindDetails
      [0],'|')) AS oBindings
48        }
49    // The distinct output bindings are stored in the variable outputBindings.
50      WITH mAct,oBindings
51      UNWIND oBindings AS oBind
52      RETURN mAct, COLLECT(DISTINCT oBind) AS outputBindings",
53      {mActivity:mActivity,bindingsDetails:bindingsDetails,freqMax:freqMax,
      allOutputs:allOutputs}
54 )YIELD value
55 // The output bindings are set as properties for the DG_node nodes.
56 WITH value.mAct AS mActivity, value.outputBindings AS outputBindings
57 MATCH (m:Model{ID:"ModelID"})-[:CONTAINS]-(dg:DG_node{ID:mActivity})
58 SET dg.OutputBindings = outputBindings
```

The query to identify the input bindings that define the OR-joins and AND-joins, whose main structure is the same as the one presented above, can also be observed in Appendix C.

Once the input and output bindings have been identified for every activity in the DG, the mining of the splits and joins is complete, which also marks the completion of our implementation for the Heuristic Miner algorithm.

For our running example with the events from the *Orders.csv* log, we used the following parameters to discover a process model using our implementation of the Heuristic Miner algorithm:

- *Class*: "Activity+Life-cycle"
- *Entity Type (:DF)*: "DeliveryOrder"
- *Frequency Threshold*: 0.1
- *Dependency Threshold*: 0.6
- *L1L Threshold*: 0.9
- *L2L Threshold*: 0.9
- *Bindings Threshold*: 0.1
- *Relative-to-best Threshold*: 0.0

Figure 5.6 shows a subset of the nodes that remain in the database after the execution of the algorithm based on the parameters described above for our running example. The algorithm parameters are stored in the Algorithm node as properties. As a design decision, we generate a default name for

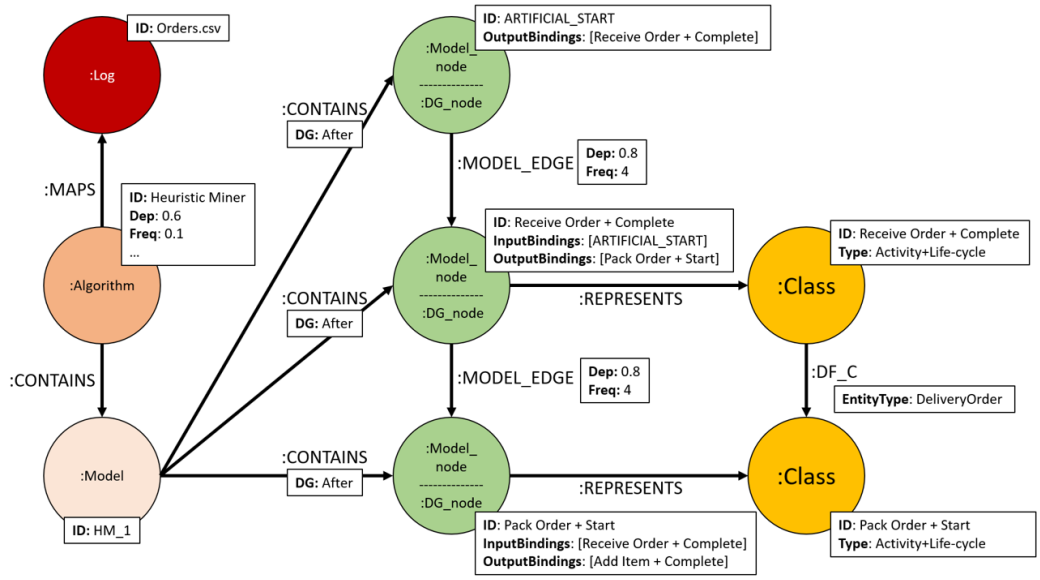


Figure 5.6: Example of the nodes and relations created in the database after the Heuristic Miner algorithm is executed.

every model that is generated. For this model, generated with the Heuristic Miner algorithm, the ID is "HM_1" (the next model would receive the name "HM_2"). The :CONTAINS relations between the Model node and the DG_nodes remain with the "*DG: After*" property, helping us identify that this DG does represent the final result. Another thing to note is that the DG_node that represents the artificial Start is neither connected to the Class nodes nor it has the InputBindings property, as opposed to the DG_nodes that do represent a model activity.

Once the process model, represented by the DG, has been created in the database, we need to provide a way for the users to execute this algorithm and visualize the results. The functionalities implemented in the user interface for this purpose are described next.

5.2.9 Visualizing the Process Model

In this section, we describe the different functionalities implemented in the user interface to allow users to execute the Heuristic Miner algorithm, visualize the existing models in the database, and visualize the Dependency Graphs that represent the process models.

To allow users to execute the Heuristic Miner algorithm, we decided to add a "Generate Model" button in the Algorithms panel, as shown in Figure 5.7. Since we are building the tool thinking that possible extensions can be

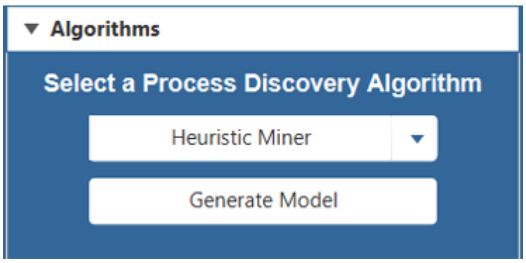


Figure 5.7: Algorithms Panel.

added in the future, we added a dropdown menu on top of the button that will allow the users to select between different process discovery algorithms to generate a process model.

Once the log has been selected in the Logs panel and the Heuristic Miner algorithm has been selected in the dropdown menu, clicking on the "Generate Model" button will display a new window, shown in Figure 5.8, where the user can specify the parameters needed by the queries to generate the process model.

Since the data model allows us to create several views for an event log, the first parameters that need to be defined by the user are the Class type and Entity Type that will be used to define the model activities and the path that connects them; this information is retrieved from the existing Class and Entity nods in the database. Then, to allow the users to define the thresholds, we created sliders, similar to the user interface from ProM. We consider that the thresholds that have a bigger impact on the discovery of the process model are the Frequency and Dependency thresholds, which is why we put them together as part of the basic configuration. The other parameters needed for the queries are included inside the *Advanced Configuration* section of the window. Figure 5.8 shows the default values for every threshold. Once the user clicks on the "Generate Model" button, the queries defined from Section 5.2.4 until Section 5.2.8 are executed, generating the process model.

To allow the users to check the existing process models in the database, we created a table in the Models panel of the user interface, shown in Figure 5.9. The table displays the list of models based on the existing Model nodes in the database. The table at the bottom can be populated with additional details of the model by clicking on the "View Model Details" button, showing properties such as the algorithm or the parameters used to generate the model.

Then, to allow the users to visualize a process model, we added the "Show Model" button, which will display the model currently selected on the Graph panel, as shown in Figure 5.10. If the user clicks on one of the nodes in the

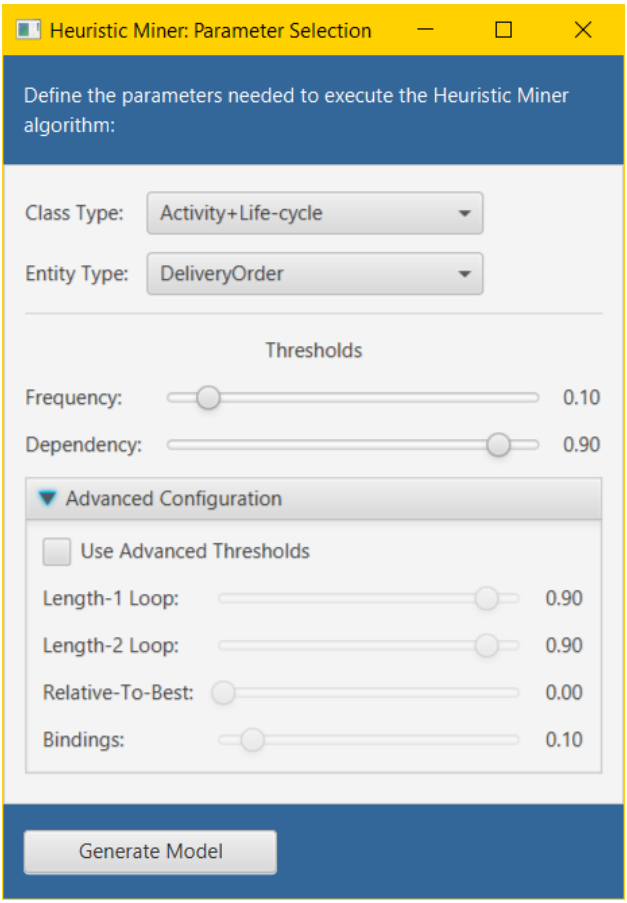


Figure 5.8: Parameter selection window for the Heuristic Miner algorithm.

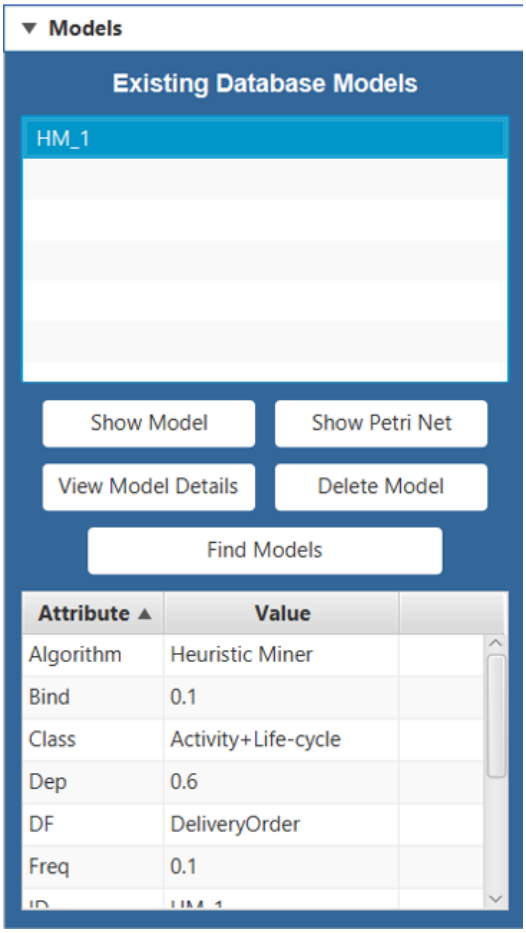


Figure 5.9: Models panel of the user interface.

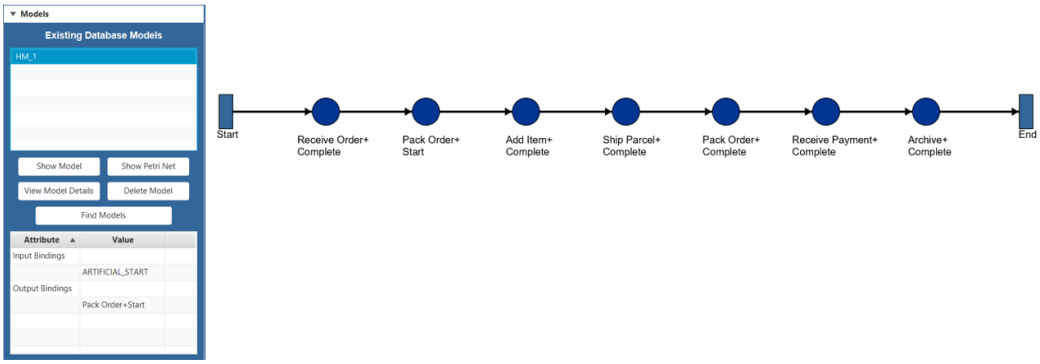


Figure 5.10: Clicking on the "Show Model" button will display the process model on the Graph panel.

graph, the table at the bottom of the Models panel will be populated with the that activity's input and output bindings so the user can see where the graph contains its splits and joins.

Finally, to allow users to delete models from the database, we added the "Delete Model" button, which will run a Cypher query that deletes the Algorithm, Model, and Model_node nodes, and their relations, from the database.

We should also mention that we took the design decision to not delete the process models when a log is deleted from the database to allow the users to remove event logs from the database while keeping the process models for future reference. Other than the Event, Entity, Class, and Log nodes, the only node related to the models that is also deleted is the Algorithm node. The Algorithm node stores the parameters defined through the user interface to generate the model based on the event log, so if the event log is deleted from the database, the tool can no longer ensure that the same parameters will produce the same process model, given that the tool cannot confirm whether a newly imported event log that has the same name as the original one will also contain the same data, which is why the Algorithm node is deleted together with the event log. On the other hand, the models can still be retrieved from the database using the Model node as reference even if they are not connected to an Algorithm node.

Now that the user interface provides a way for users to generate and visualize a process model based on an event log through the Heuristic Miner algorithm, we have completed the implementation for the Model Layer of the architecture shown in Figure 3.1, addressing the *Process discovery* activity. The next step is to add additional functionalities in the tool that help the user analyze the process models that have been generated. We do this by implementing the next layer of the architecture, the Model Analysis layer, which addresses the *Diagnose* activity.

Chapter 6

Performing Process Mining Analysis with the User Interface

In Chapter 5, we define how the Model Layer was implemented in the tool, addressing the *Process discovery* activity. In this chapter, we discuss the implementation of the Model Analysis Layer, addressing the *Diagnose* activity. First, we identify two tasks that we must be able to execute to address this activity. Then, we describe our implementation of the first task, *Model comparison*. Finally, we describe our implementation of the second task, *Model Querying*.

6.1 Identifying the Requirements for Model Analysis

In this Section, we describe the requirements we have identified for our tool to implement the Model Analysis Layer, addressing the *Diagnose* activity. First, we discuss the need to provide a model comparison functionality to help analysts identify the models that provide the best value for the diagnosis. Then, given that our tool is capable of storing several process models in the database, we discuss the need to be able to query for them and find the most relevant models for the analysis.

The next step after process models are generated is to analyse how these models represent the current state of the process so potential improvements and changes can be identified, addressing the project's goals set initially. As mentioned in the PM^2 methodology, two important components to make a successful analysis are the correct interpretation of the results and being

able to distinguish unusual results from the expected ones [2]. Although these components depend on the user's expertise in understanding process models and the domain knowledge to identify anomalies in the results, a good representation of the results can help in that regard as well.

The tool already presents a first contribution to provide a good representation of the results by providing the users with an interactive view of the process model generated through a discovery algorithm, where they can obtain additional details on the model activities by clicking on them in the user interface, as shown in Section 5.2.9. However, more often than not, more than one process model will be generated during the analysis. This might be done to obtain different perspectives of the data or to find interesting behavior in the process that may appear through the use of different discovery algorithms or algorithm parameters. Therefore, it becomes necessary to compare these models so the process analysts can have a way to select those that provide the best value for the diagnosis.

There are already some existing methods in which two models can be compared. For example, the quality of the models can be compared based on their results for the four quality dimensions mentioned in [11], allowing us to determine which model best represents the behavior of the event log. We can also determine whether two models are equivalent or not based on observed behavior, as proposed in [12]. However, even with these methods available, there are still challenges being reported with respect to the existing tool support for model comparison.

During the discussion of their results, the authors of the PM^2 methodology mention the difficulties found while comparing process models. At one point during their analysis, it was necessary to compare between several process models, but the manual comparison required from significant effort and they found limited tool support on this regard. Later on, they also mention that in ProM "it is currently time-consuming to switch between different views or filter applications on the same data and to compare results" [2].

The graph data model presents us with a good opportunity to contribute in this regard and provide a way to compare different process models. Since the data for every model is stored in the same space, we should be able to retrieve the nodes and edges that conform the models and find a way to compare them in the user interface.

However, we should also consider the fact that for some process mining projects, the analysis may cause multiple models to be generated and stored in the database, as we can see from the research made in [22]. In this research, they report that large organizations may deal with collections of hundreds or thousands of business process models, which is why they must rely on repository technology to store the models.

The model repositories provide the necessary infrastructure to store a collection of process models, and are meant to support different management techniques that handle process model collections. The first management technique mentioned in [22] is *querying*. In this context, querying refers to finding a specific process model within a large collection. As mentioned in [22], querying for process models in a repository has multiple uses, such as finding models that do not comply with standards, finding models that can be used as a template to define new ones, or finding models that have specific activities or relations between activities.

Since our tool allows us to store multiple models, it becomes relevant for us to also consider the implementation of a management technique such as querying to allow users to find which models they would like to analyse or compare

In summary, we consider that to complete the implementation of the Model Analysis Layer and address the *Diagnose* activity, we must be able to execute these two tasks inside the tool, the *Model Comparison* and the *Model Querying*. The discussion on how we can include the comparison between process models is presented next.

6.2 Model Comparison

In this section, we describe our implementation for model comparison on top of the graph database. First, we discuss an additional implementation needed for the process models. Then, we describe how users can perform model comparison through the user interface.

The challenges related with the model comparison mentioned in [2] help us realize that tool support is needed during the analysis of the results when the project requires analysts to make comparisons between different process models. We consider that the first and most intuitive way to compare models is through visual inspection, such as finding the differences between the models' graphical structures or identifying which activities appear in one or both models. This consideration helps us define our first requirement for the model comparison: (**R1**) to provide a visual inspection of two process models. Since our user interface allows us to display any set of nodes and edges from the database, including the model nodes, we could simply retrieve the components of two models and display them in the Graph panel so the users can do the visual inspection of the results.

However, even when this simple approach works for our current implementation, it might not provide the best solution considering that different discovery algorithms can be implemented for the tool moving forward. Right

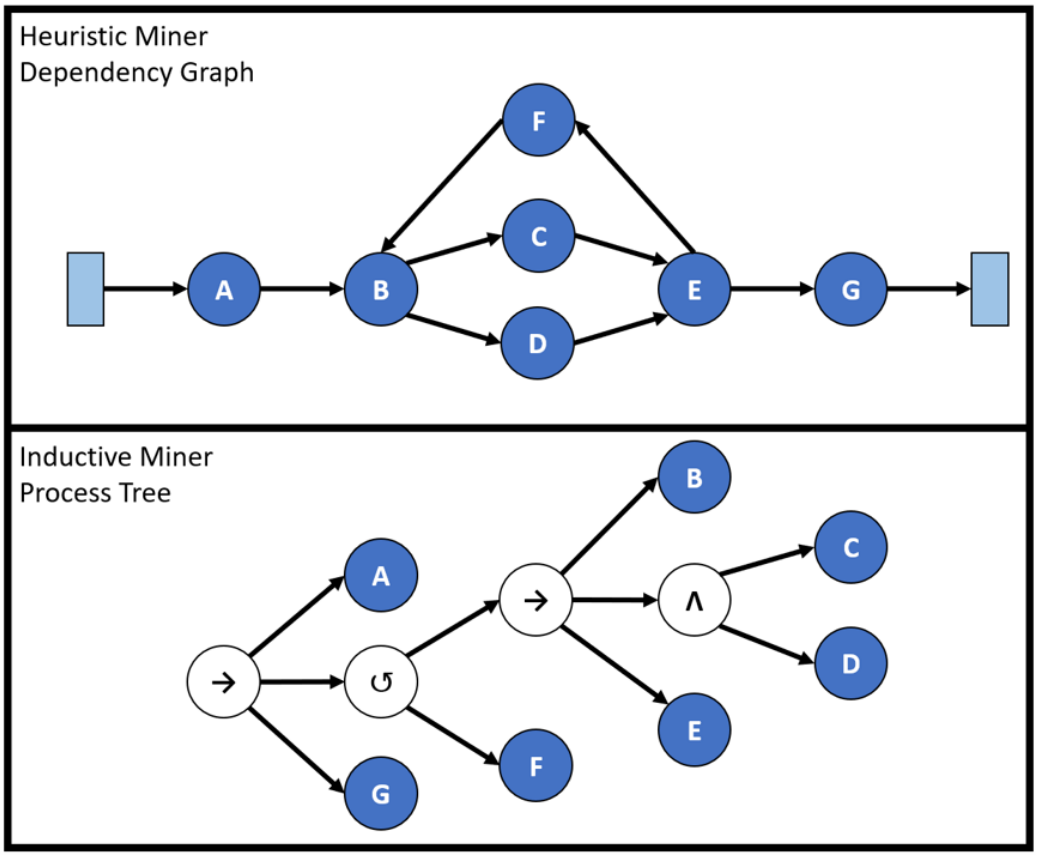


Figure 6.1: Comparison between the process models generated by the Heuristic Miner (dependency graph) and the Inductive Miner (process tree).

now, we have implemented one process discovery algorithm whose process models are always generated in the same modeling language, making the comparison between them straightforward, but when a different algorithm is implemented and whose output is not a dependency graph, the comparison will prove more difficult, lessening the impact of our potential contribution. Figure 6.1 shows an example of this; both models represent the same behavior, however, this is not apparent from the visual inspection, making the side-to-side comparison ineffective.

Therefore, we must define a prior requirement for our implementation of the model comparison: (**R0**) to provide the users with a common ground to make the comparison between process models. We consider that Petri nets can provide a solution for this problem. Petri nets are one of the most common ways in which process models are represented, and, even if it is not the default model language used by some algorithms to express their results (such as the Heuristic Miner), it is possible to translate these languages into

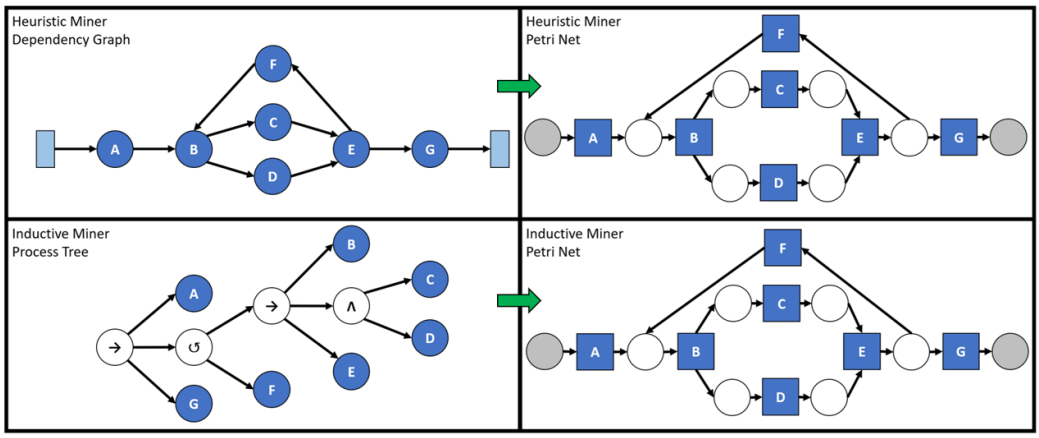


Figure 6.2: Comparison between the process models generated by the Heuristic Miner and the Inductive Miner including their Petri net representations.

a Petri net. In Figure 6.2, we have a follow-up to the models shown in Figure 6.1; this time, we have added their Petri net representations, where it is now evident that both models describe the same behavior.

Seeing how this translation can help us make the model comparison more effective, the next step in our implementation is to define how we can include Petri nets in the data model and address requirement **R0**.

6.2.1 Representing Petri Nets in the Data Model

To add Petri nets into the data model, we need to find a way to represent its different components and then connect them with the rest of the graph. A Petri net is, at its core, a graph structure, where the edges connect nodes that represent either places or transitions. As we know, places represent states, such as the start of the process, and transitions represent state changes, which in our case refer to the activities of the process model.

Considering this, we can define our next extension to the data model to represent Petri nets. This extension can be observed in Figure 6.3.

In our proposed extension, we introduce a new label for the Model_nodes called *:PetriNet*. As we defined in Section 5.1.3, the Model_nodes are used to represent the process model, with its second label allowing us to provide more details on the type of graph structure it is representing, which is what we are doing with this new label. The PetriNet nodes represent the different types of places and transitions that may be used in a Petri net, and the way to distinguish between them is through the *type* attribute. There are four possible values that can be assigned for the *type* attribute in a PetriNet node: *p* for places, *s_e* for the start and end places, *t* for transitions, and *tau* for

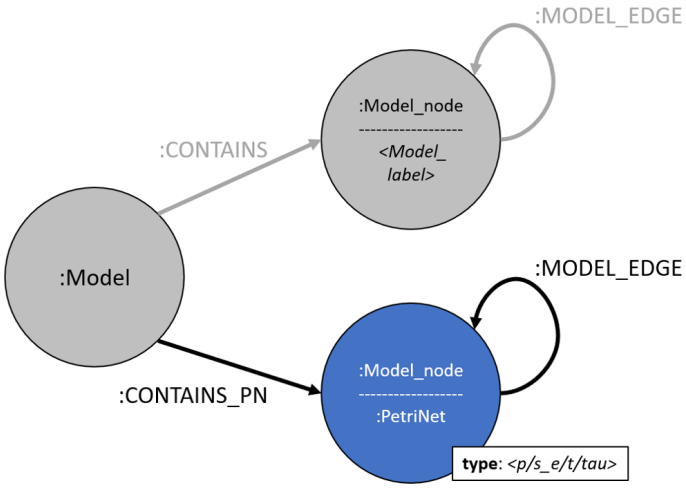


Figure 6.3: Our second extension to the model. PetriNet nodes connect with the rest of the graph through the Model nodes.

tau transitions. PetriNet nodes that represent transitions have an additional property t to define the name of the activities of the model. We decided to use the same node label to represent all the components of a Petri net to simplify the queries that search for a match inside this part of the graph, avoiding the need to build queries that need to distinguish between different node labels in the graph path.

Following the definition of our first extension to the data model (shown in Figure 5.1), the PetriNet nodes are connected between them by :MODEL_EDGE relations. However, in contrast to that extension, the PetriNet nodes are not connected to the Model node through a :CONTAINS relation. Since both graph structures are connected to the same Model node, we need to distinguish between the edges that connect the Model node to the different types of graphs, and we can think of two ways to do this: (1) Define a property for the :CONTAINS edge that specifies the type of graph the Model node is pointing to, or (2) Define a new relationship type for the new graph structure. We decided to implement the second option because it prevents us from having to query over the properties of each relation of the Model node to find those that connect to the graph we need. This is why the connection between the Model node and the PetriNet nodes is done through a new type of relation, called :CONTAINS_PN.

Figure 6.4 shows an example of a Petri net represented inside the data model, where we can see how the different components of the Petri net are identified through the node properties.

Now that we have established how the data model can be extended to

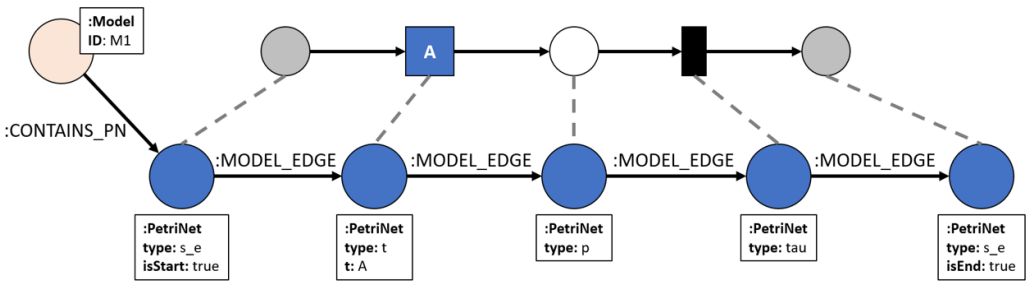


Figure 6.4: Example showing how a Petri Net is represented in the data model.

include Petri nets, addressing **R0**, we can focus on how the dependency graphs that result from the execution of the Heuristic Miner algorithm can be translated into Petri nets.

6.2.2 Translating Dependency Graphs into Petri Nets

To translate the dependency graphs generated during the execution of the Heuristic Miner algorithm, we implemented additional Cypher queries that are executed right after the execution of the Heuristic Miner algorithm ends.

These queries take advantage of the input and output bindings defined for each activity in the dependency graph to create the PetriNet nodes. In our implementation, we defined two main queries, one that translates the output bindings into places and transitions, and another that does the same for the input bindings. We can look at Figure 6.5 for an example of how the queries use the output bindings to create the Petri net. The full set of queries generated to translate the dependency graph into a Petri net can be observed in Appendix D.

In this example, activity A has two output bindings, one towards activities B and C, and another one towards activity D. Multiple output bindings represent an OR-split. The first step is to translate this OR-split, and to do this we create the PetriNet node for transition A and connect it to an output place (place 1 in Figure 6.5). Then, as a second step, for each output binding, we create and connect a tau transition to the newly created place (tau transitions 2 and 3 are created and connected to place 1). In the third and final step, for each activity in every output binding, we connect a place to the corresponding tau transition. In the example of Figure 6.5, the first output binding has two activities, so we create two places (places 4 and 5), and the second output binding has one activity, so we create one place (place 6). Multiple activities in one output binding represent an AND-split.

After doing this translation for the output bindings of all the activities in

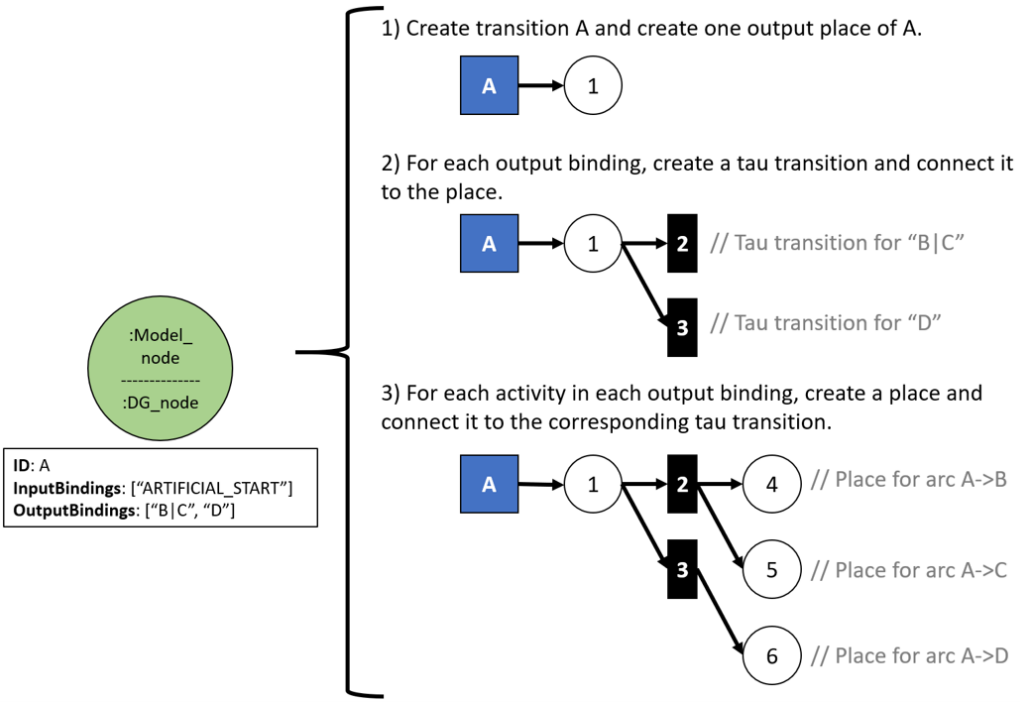


Figure 6.5: Steps followed to translate the output bindings of activity A into a Petri Net.

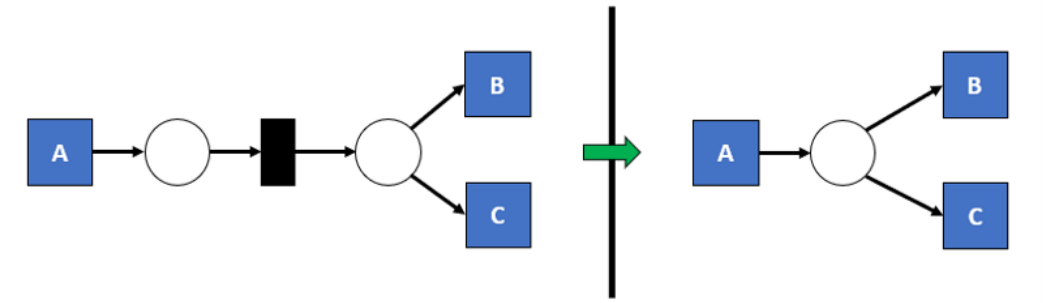


Figure 6.6: The Petri Nets generated can be simplified by identifying and removing unnecessary tau transitions.

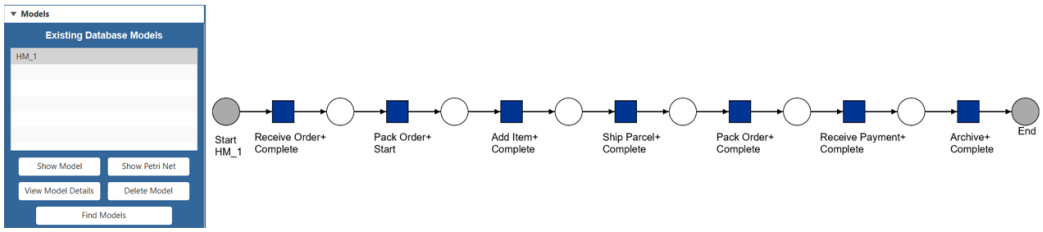


Figure 6.7: Clicking on the "Show Petri Net" button will display the corresponding Petri net on the Graph panel.

the dependency graph, we run a similar query for the input bindings, merging the inputs with the PetriNet nodes already created for the outputs. Once this is done, we have completed the translation from the dependency graph into a Petri net.

However, to reduce the amount of tau transitions generated during the translation, we implemented another pair of queries that simplify the Petri net. These queries can also be observed in Appendix D. An example of this simplification can be observed in Figure 6.6, where we simplify the Petri net by removing the redundant tau transition and place nodes.

After defining how a Petri net can be created and simplified, we must provide a way for the user to visualize it in the user interface. To do this, we added a "Show Petri Net" button in the Models panel, as shown in Figure 5.9. Clicking on this button will execute a simple query that matches the corresponding Model node with the PetriNet nodes based on the :CONTAINS_PN relation. The results from this query are processed to display the Petri net on the Graph panel, as shown in Figure 6.7.

Figure 6.7 shows the resulting Petri net representation of our running example. As we can see, the names of the transitions represent the ID of the activities from the dependency graph first shown in Figure 5.10.

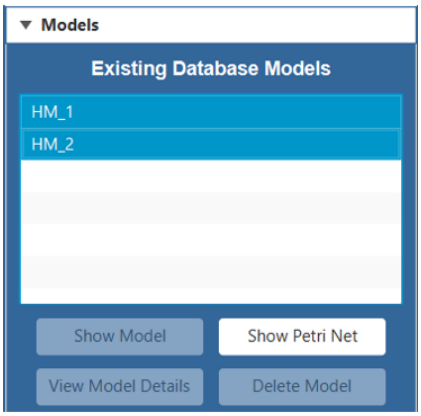


Figure 6.8: When 2 models are selected, only the "Show Petri Net" button remains enabled.

Now that we have defined how Petri nets can be visualized in the user interface, we can use them to implement the model comparison functionality in the tool and address requirement **R1**.

6.2.3 Comparing Models

After defining the common ground to make the comparison between process models in the form of Petri nets, we can describe our implementation that allows users to visualize a side-by-side comparison between two Petri nets.

To do the comparison, users first need to select the two models to compare from the list of existing models in the Models panel. As shown in Figure 6.8, the buttons that affect one model are disabled while the "Show Petri Net" button remains enabled. The second model shown in the figure, "HM_2", was created as an example to showcase the model comparison functionality. This model was created manually from a modified version of the "Orders.csv" log, where the name of the "Ship Parcel" activity was changed to "Ship Package" and an additional "Add Item+Complete" transition was added to the Petri net.

Clicking on the "Show Petri Net" button while the two models are selected will show a new visualization of the Petri nets in the Graph panel, as shown in Figure 6.9. This visualization shows the two selected models side by side. A "virtual" connection is drawn between Petri net transitions that have the same name. Clicking on one of these transitions will highlight it together with its virtual connections and the transitions at the other end (as seen in Figure 6.9 for the "Add Item+Complete" transition), providing a way for users to analyse the similarities between the models. On the other hand,

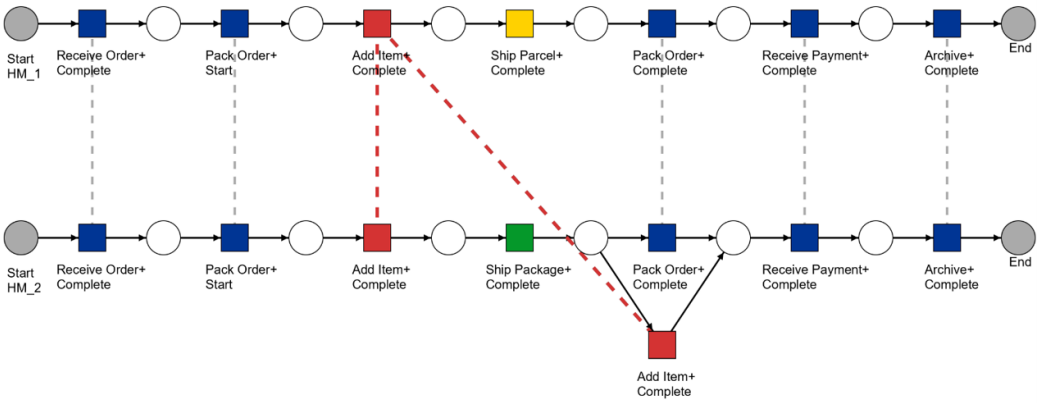


Figure 6.9: The Model comparison functionality displays two models side by side and highlights the selected transitions and missing/additional transitions between models.

those transitions from the top log that are not found in the bottom log and vice versa are colored yellow and green respectively, to make them stand out from the rest (such as transitions "Ship Parcel+Complete" and "Ship Package+Complete" in Figure 6.9).

Now that we have defined how two models can be compared in the user interface based on their Petri net representation, addressing requirement **R1**, we look into the second task we defined for the Model Analysis layer, *Model Querying*, which will help us find models stored inside the database.

6.3 Model Querying

In this section, we describe our implementation to allow users to find models inside the database. First, we describe our approach for model querying, which is based on the behavior patterns of the process models. Then, we describe how this approach was implemented in the tool.

6.3.1 Defining our approach for Model Querying

As we mentioned in Section 6.1, it is possible that during a process mining project several process models are created in order to try to find the best answers for the research questions defined at the beginning of the project. To deal with large collections of models, the research from [22] mentions model querying as one of the management techniques used to handle these collections in model repositories. This is why we believe that being able to query process models is an essential activity during the analysis of the

process, especially when our tool already provides a way for users to compare between process models.

We consider that the model comparison functionality for the tool becomes more effective when we have already queried for process models that share some characteristics, making their differences more apparent.

In order for us to add this functionality into the tool, we need to define the way in which the model querying will be implemented. For example, in the survey research made in [13], they focus on 5 types of queries that can be used for model querying:

1. *Exact query based on graph structure.* For this type of query, process models are considered as graph data and models are retrieved based on their match with a given subgraph.

2. *Similarity query based on graph structure.* For these queries, instead of expecting an exact match, a threshold is defined to also retrieve models that are similar to the subgraph provided as an input.

3. *Exact query based on behavior semantics.* These queries look for models based on a given set of behavior requirements, for example, finding models where activity "A" precedes activity "B" or activity "C" can be executed in parallel with activity "D".

4. *Similarity query based on behavior semantics.* Similar to the previous type of queries, these queries look for models based on behavior. In this case, models are retrieved if their traces are the same as the one of the given condition model.

5. *Query based on operation semantics.* In contrast with the previous types of queries, these are based on operation semantics. For these queries to work, process models must be tagged with its operational information based on a defined ontology.

For our tool, we decided to implement the model querying functionality based on the third type of queries described in [13], the *Exact query based on behavior semantics*. This is because Cypher allows us to define queries that look for a match based on behavior, not only on specific graph structures. This allows us to provide more flexibility to the users than what we could provide by having to ask them to build a Petri net subgraph to be used for the model querying.

To define what kind of behavior users can query in the Petri nets stored in the database, we can use the basic workflow patterns described in [14]

as reference. As part of the fundamentals of behavior inclusion to query process models, they define three workflow patterns: *sequence*, *parallel split*, and *exclusive choice*.

1. *Sequence.* This pattern returns a match for those models that execute the transitions in the same order as that provided in the input trace. This can be satisfied if such transitions are on a path in the model.

2. *Parallel Split.* This pattern returns a match if the transitions provided as input can be enabled concurrently in the process. This means that, between these transitions, the ordering in which they can be fired should not be restricted in the matching model.

3. *Exclusive Choice.* This pattern returns a match when the process splits into several paths and only the path of one of the transitions provided as input can be active.

Now that we have defined these basic patterns, we can describe the querying patterns that we included in our implementation.

6.3.2 Querying Patterns

For our tool, we define 6 different patterns. First, based on the sequence pattern, we implemented a *Directly-Follows* and an *Eventually-Follows* pattern. Then, we added the *Parallel Split* and *Exclusive Choice* patterns based on the basic patterns of the same name. Finally, we added two more patterns that we consider can provide value during the model querying, the *Starts with* and *Ends with* patterns. The Cypher queries built for each pattern are described next.

Directly-Follows Pattern The Cypher query for the Directly-Follows pattern is shown below. To find the matching models, we first need to specify the transitions that must be in the path (lines 1-3), then, we need to make sure that any node in between is not another transition, which we can confirm by checking whether the "t" property exists (lines 5-7), since this property is exclusive of transitions.

```
1 MATCH (m)-[:CONTAINS_PN]->(s:PetriNet)-[:MODEL_EDGE*]->(p:
    PetriNet)
2 MATCH path = (p)-[:MODEL_EDGE*]->(q)
3 WHERE p.t = "TransitionA" AND q.t = "TransitionB"
4 WITH DISTINCT m.ID AS modelID, nodes(path) AS path
5 WITH modelID, tail(path) AS path
```

```
WITH modelID, apoc.coll.remove(path, size(path)-1) AS path
WHERE ALL(n IN path WHERE n.t IS NULL)
RETURN DISTINCT modelID
ORDER BY modelID
```

Eventually-Follows Pattern This query is similar to the one built to find Directly-Follows patterns, but in this case there is no need to check what is in between transitions A and B, as long as they are on the same path along the Petri net the model will be considered a match.

```
MATCH (m)-[:CONTAINS_PN]->(s:PetriNet)-[:MODEL_EDGE*]->(p:
    PetriNet)
MATCH (p)-[:MODEL_EDGE*]->(q)
WHERE p.t = "TransitionA" AND q.t = "TransitionB"
RETURN DISTINCT m.ID AS modelID
ORDER BY modelID
```

Parallel Split Pattern For this query, we have to find the "AND" splits and joins in the Petri net. We consider an "AND" split is found when a transition has more than one outgoing edge, and we consider an "AND" join is found when a transition has more than one ingoing edge (lines 1-5). Then, we try to find two distinct paths in the Petri net between a parallel split and a parallel join that contain A and B (lines 7-8). If B is not inside the path for A and vice versa, we have found a model that contains a parallel split for A and B (line 9).

```
MATCH (m)-[:CONTAINS_PN]->(s:PetriNet)-[:MODEL_EDGE*]->(ps:
    PetriNet)-[e:MODEL_EDGE]->()
MATCH (m)-[:CONTAINS_PN]->(s:PetriNet)-[:MODEL_EDGE*]->()-[f:
    MODEL_EDGE]->(pj:PetriNet)
WHERE (ps.type = "tau" OR ps.type = "t") AND (pj.type = "tau"
     OR pj.type = "t")
WITH m.ID AS modelID, ps, COUNT(DISTINCT e) AS numSplit, pj,
    COUNT(DISTINCT f) AS numJoin
WHERE numSplit > 1 AND numJoin > 1
WITH DISTINCT modelID, ps, pj
MATCH path = (ps)-[:MODEL_EDGE*]->(p)-[:MODEL_EDGE*]->(pj)
MATCH path2 = (ps)-[:MODEL_EDGE*]->(q)-[:MODEL_EDGE*]->(pj)
WHERE p.t = "TransitionA" AND q.t = "TransitionB" AND NOT q IN
    nodes(path) AND NOT p IN nodes(path2)
RETURN DISTINCT modelID
ORDER BY modelID
```

Exclusive Choice Pattern This query is similar to the one built for the Parallel split patterns, but the difference is where we look for the splits and

joins. Instead of looking for transitions with more than one outgoing/ingoing edge, we look for *places* that have these edges (line 3). Then, the query follows the same steps as the previous one to determine which models are a match.

```
1 MATCH (m)-[:CONTAINS_PN]->(:PetriNet)-[:MODEL_EDGE*0..]->(xs:
    PetriNet)-[e:MODEL_EDGE]->()
2 MATCH (m)-[:CONTAINS_PN]->(:PetriNet)-[:MODEL_EDGE*0..]->()-[
    f:MODEL_EDGE]->(xj:PetriNet)
3 WHERE (xs.type = "p" OR xs.type = "s_e") AND (xj.type = "p"
    OR xj.type = "s_e")
4 WITH m.ID AS modelID, xs, COUNT(DISTINCT e) AS numSplit, xj,
    COUNT(DISTINCT f) AS numJoin
5 WHERE numSplit > 1 AND numJoin > 1
6 WITH DISTINCT modelID, xs, xj
7 MATCH path = (xs)-[:MODEL_EDGE*]->(p)-[:MODEL_EDGE*]->(xj)
8 MATCH path2 = (xs)-[:MODEL_EDGE*]->(q)-[:MODEL_EDGE*]->(xj)
9 WHERE p.t = "TransitionA" AND q.t = "TransitionB" AND NOT q IN
    nodes(path) AND NOT p IN nodes(path2)
10 RETURN DISTINCT modelID
11 ORDER BY modelID
```

Starts With Pattern For this query, we just need to determine if the transition passed as parameter is connected to the starting place of the Petri net (lines 1-2) to consider a match for the model.

```
1 MATCH (m)-[:CONTAINS_PN]->(s:PetriNet)-[:MODEL_EDGE]->(p:
    PetriNet)
2 WHERE s.isStart IS NOT NULL AND p.t = "TransitionA"
3 RETURN DISTINCT m.ID AS modelID
4 ORDER BY modelID
```

Ends With Pattern Similar to the previous query, we just need to determine if the transition passed as parameter is connected to the end place of the Petri net (lines 1-2) to consider a match for the model.

```
1 MATCH (m)-[:CONTAINS_PN]->(:PetriNet)-[:MODEL_EDGE*]->(p:
    PetriNet)-[:MODEL_EDGE]->(e:PetriNet)
2 WHERE e.isEnd IS NOT NULL AND p.t = "TransitionA"
3 RETURN DISTINCT m.ID AS modelID
4 ORDER BY modelID
```

Now that we have defined the queries for the patterns that we can use to find models in the database, we can look at how these queries were added into the user interface.

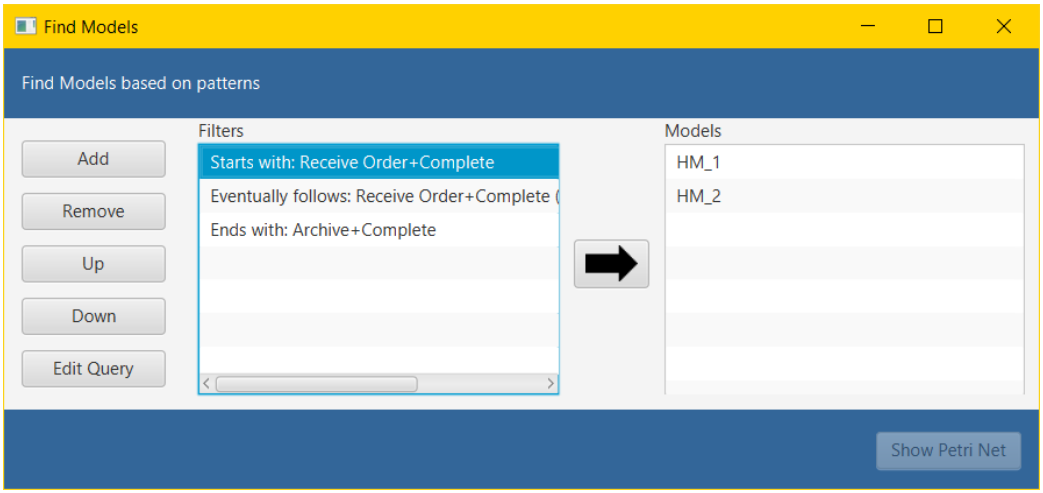


Figure 6.10: Several pattern can be defined to find specific models stored in the database.

6.3.3 Querying models through the user interface

To allow users to find models inside the database through the pattern queries, we implemented a "Find Models" button in the Models panel, as shown in Figure 5.9. Clicking on this button will display the window shown in Figure 6.10.

We considered that the tool should allow users to define their search through one or more patterns so they can obtain more specific results depending on their needs. This is why we included a list of "Filters", where the result of the first pattern of the list will be used as input for the next pattern in line and so on.

To add a search pattern, the user can click on the "Add" button, which will display the window shown in Figure 6.11. Inside this window, the user can select one of the six patterns defined previously and specify the transition(s) they want to search for in the Petri nets that represent the process models. Each pattern is accompanied by an image and a description to help users understand how the pattern looks like in the graph structure of a Petri net. On this window, clicking on the "Add Filter" button will add a new entry to the "Filters" list in Figure 6.10.

Back in the main window (Figure 6.10), users may choose to remove patterns with the "Remove" button or they can switch the order in which they filter from the list of models with the "Up" and "Down" buttons.

To allow more experienced users to define custom queries outside of the predefined patterns, we added an "Edit Query" button that allows users to modify the queries for the patterns already selected. We can see an example

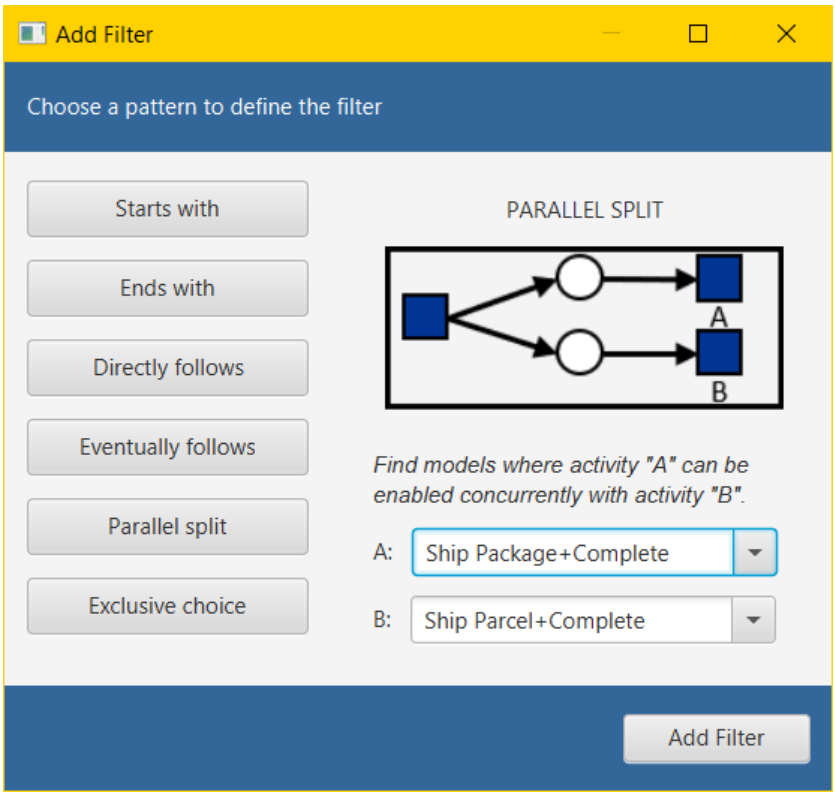


Figure 6.11: Users can choose between 6 different patterns to define a filter for the model querying.

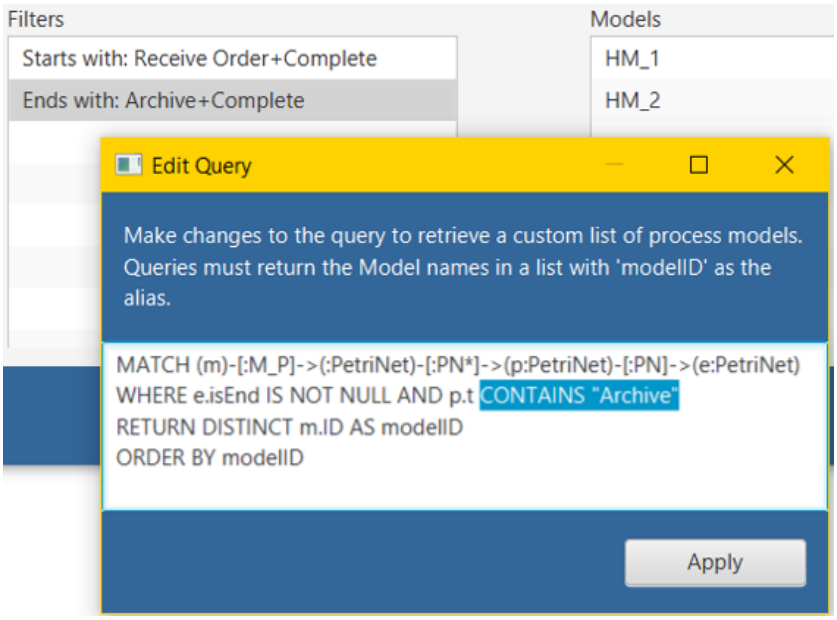


Figure 6.12: Users can modify the pattern queries to customize their results.

in Figure 6.12, where we edit the original "Ends with" query to find process models that end with a transition whose name *contains* the word "Archive", when the original query was built to find an exact match with the name "Archive+Complete". On this window, clicking on the "Apply" button will save the changes and update the list to let the user know that the pattern query has been modified, as shown in Figure 6.13. These queries are executed with a "read-only" session to prevent unexpected changes into the data model caused by the custom queries.

Once the pattern filters are set, the user can start the search by clicking on the "arrow" icon shown in the middle of Figure 6.10. We implement the search with the help of Java code, where we create a loop that stops either when the query for the last filter on the list has been executed or one of the intermediate queries returns no matching models.

After the search is complete, the list of models on the right side of Figure 6.10 is updated with the results, but at any point the user can choose one or two models from this list to display their Petri nets in the Graph panel, similar to the functionalities described for the "Show Petri Net" button in Sections 6.2.2 and 6.2.3.

This functionality marks the end of our implementation for the Model Analysis Layer, where the model comparison and model querying tasks help us to address the *Diagnose* activity. Now that the implementation of the tool is complete, we can move on to its evaluation.

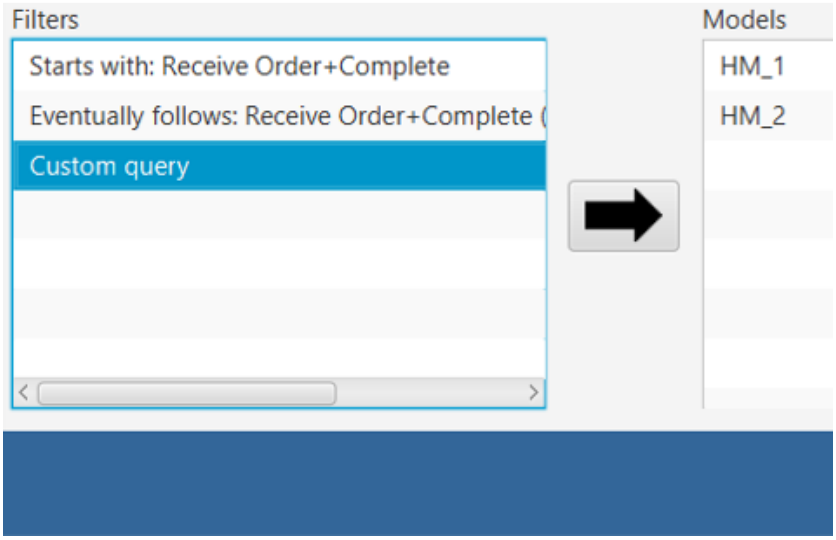


Figure 6.13: After a query has been modified by the user, the name of the filter is updated to "Custom query" to let the users know which pattern queries have been updated.

Chapter 7

Evaluation

In this chapter, we present 4 experiments designed to help us determine if our implementation can help us answer the research questions set initially in Section 1.3. As we recall, the main objective of our project is to determine if it is possible to build upon the graph-based data model presented in [1] to obtain a viable alternative to execute the activities that provide significant results in a process mining project. For each experiment, we provide the details about its setup and execution, after which we present and discuss the results. All the experiments were run on a 2 Core Intel i7-6500U @ 2.50 GHz Windows machine with 8 GB RAM.

7.1 Experiment 1

7.1.1 Setup

The objective of the first experiment is to help us answer research questions RQ1 and RQ2:

RQ1. *Is it possible to build on top of the data model proposed in [1] to execute process mining-related activities in a graph database?*

RQ2. *Can we store process models in the graph database? If so, how would the execution of process mining change in this environment?*

Our implementation described in the previous chapters already seems to provide a positive answer for these questions, since we have defined how we can build on top of the data model from [1] and extend it to execute and store the results from distinct process mining activities, such as the creation of views, process discovery or model comparison.

However, it is important to test these functionalities with a real dataset to check if the activities are not only executed successfully, but also fulfill their overall purpose by providing significant results for a process mining project.

In addition, we should also compare the execution of these activities against other process mining tools to check whether we produce the same results and to identify if their execution in this environment presents changes in process mining.

The experiment is designed to simulate the need for an exploratory analysis on the BPI Challenge (BPIC) 2017 log [15]. This log records cases of a loan application process at a Dutch financial Institute and in total it contains 1,202,267 events pertaining to 31,509 loan applications. For these applications, a total of 42,995 offers were created.

Let us assume that, for a process mining project related to the handling these credit applications, we want to do an exploratory analysis and obtain additional insights on the activities related to the extension of an *Offer* inside an *Application*. In other words, we would like to know how the Offer activities interact between them and how they interact within the process of an Application. For this experiment, we will work under the assumption that all the activities registered in the log that start with either *"A_"* or *"W_"* represent the Application events, while the activities that start with *"O_"* represent the Order events.

Starting from a random subset of the BPIC17 event data stored in a CSV file (consisting of 765 events pertaining to 20 different applications) [1], we use both our tool and ProM to generate two process models, one including only the Offer-related activities, the "Offers" model, and another including both the Offer-related and Application-related activities, the "Applications" model, whose visualizations will allow us to obtain the insights requested.

To showcase some of the differences between the models generated by our tool and ProM, for the model that includes both the Offer and Application-related activities we will adjust one of the parameters of the Heuristic Miner, the *Length-2 Loop* threshold, changing the default parameter of 0.9 and setting it to 0.7. In both cases, we keep track of the amount of interactions with the application, such as clicking on a button, moving a slider or typing a file name, and the time we take to complete each step.

We should note that our tool and ProM have some differences in the way they handle and present their results. These differences must be accounted for during the experiment because those functionalities add value to the final

[1]The Python script used to generate the subsets for all the experiments is available at https://github.com/vhernandezs/event-graph-process-mining. The name of this file is "BPIC17_Sample_20cases.csv". This script is based on those provided in the GitHub repository shared in [1].

Objective	Activity (ProM)	Activity (Graph Tool)
Importing event log	Import event data	Import event data
Creating "Offer" view	Create the "OfferID" XES file	Define the "Offer" Entity Type
	Save the "OfferID" XES file	Using "Offer" Entity Type, define the "Activity+Lifecycle" Class
Creating "Offer" model	Create the "OfferID" model	Create the "Offer" model
	Save the "OfferID" model	Visualize the "Offer" model
Creating "Application" view	Create the "Application" XES file	Define the "Application" Entity Type
	Save the "Application" XES file	Create the Derived Entity
		Using the derived Entity Type, define the "Activity+Lifecycle" Class
Creating "Application" model	Create the "Application" model	Create the "Application" model
	Save the "Application" model	Visualize the "Application" model
Comparing models		Compare the models

Table 7.1: Activities executed in each tool for experiment 1.

outcome, so we consider that we must add steps in each execution to best emulate what the other tool does automatically.

For the experiment, we include two additional steps in our tool and two additional steps in ProM. The first step added in our tool is the visualization of the discovered model. ProM already provides the visualization automatically after the process model has been discovered, so we need to account for that. The second step added in our tool is the model comparison. This functionality provides us with a better way to analyze the results from the task, so we consider important to add it as well. Then, the steps added in ProM are the export of the event log and the process model. Since our tool automatically stores the results in the database and the results in ProM are lost once the application is closed, we need to account for that functionality in ProM.

7.1.2 Execution

Table 7.1 shows the activities executed in each tool to obtain the process models that will help us understand how the Offer activities interact between them and how they interact with the Application activities. This table includes an additional column called "Objective" that helps us group the activities based on the main goal they help to achieve, providing a direct comparison between their executions. This way, we can identify for example that to create the "Offer" view, in ProM we need to create and save the XES file based on the imported CSV file, but in our tool we need to define the "Offer" entity type and the "Activity+lifecycle" class. The details on how each step was executed can be observed in Appendix E.1.

After executing the steps shown in Table 7.1 and having generated the process models in both tools, we can proceed to show the results from the

Objective	Time (s)		Interactions	
	Graph	ProM	Graph	ProM
Importing event log	20	8	13	4
Creating "Offer" view	27	29	14	19
Creating "Offer" model	36	26	8	11
Creating "Application" view	52	24	15	18
Creating "Application" model	698	30	11	14
Comparing models	8	-	2	-
Total	**841**	**117**	**63**	**66**

Table 7.2: Comparison between execution times (in seconds) and interactions with each tool.

first experiment.

7.1.3 Results

We divide the presentation and discussion of the results in two parts, first, we discuss the usability of our tool compared against ProM and then we discuss the differences between the resulting models.

Usability

We can already see in Table 7.1 that our tool does need to execute more activities overall than ProM (11 against 9). Even when we include the steps taken in ProM to save the log files and models, the two-step process to define a view on the data for our tool (define entities and classes) does impact on the final result together with the extra step we take to compare models, which is something ProM does not provide.

Then, we can see the comparison between execution times and interactions in Table 7.2. The times and interactions were also added by objective to obtain a direct comparison between the tools.

The difference between execution times favors ProM, where the process from start to finish took around 2 minutes while the same process for our tool took around 14 minutes, with the creation of the "Application" model as the biggest difference-maker. Regarding the interactions, the numbers are very close to each other.

At first glance, it may seem like our tool does not provide any significant advantage over ProM, but we should mention that the model comparison we executed in 8 seconds and with two clicks would take a considerable effort to replicate using the results from ProM, where switching between views

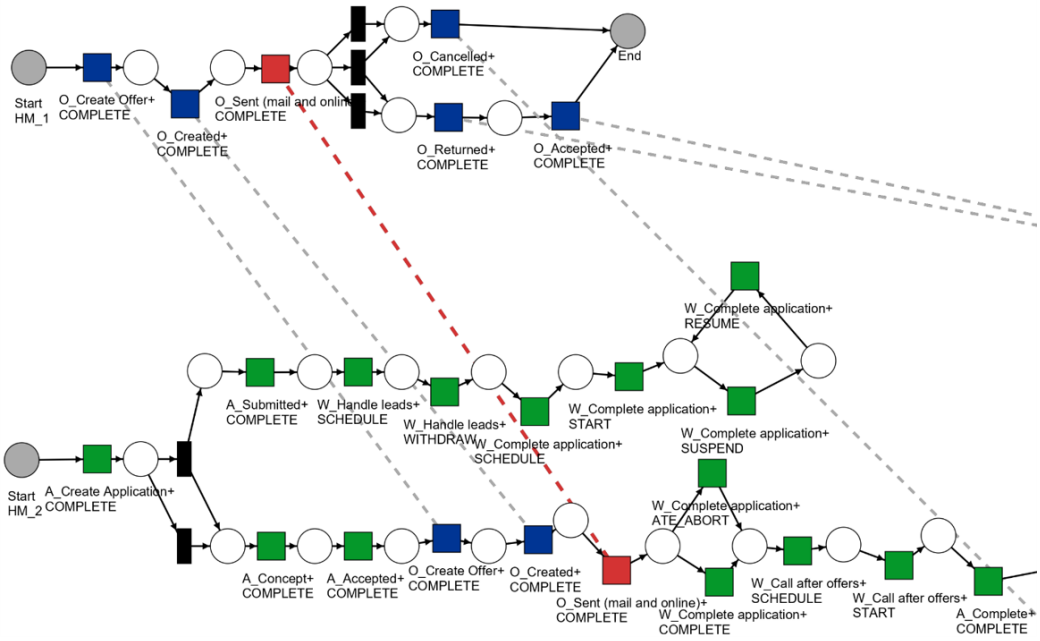


Figure 7.1: Petri net comparison of the "Offer" and "Application" process models discovered.

or exporting the images of the models to compare them side by side would increase the complexity of the analysis. In addition, our model comparison, shown in Figure 7.1, does provide additional value to the analysis. As we can see, the visualization helps us to clearly identify and track where the Offer-related activities interact within an Application.

Another thing to mention is that with our tool we do not need to create and store different views of the event data as we require in ProM with the XES and model files. This advantage becomes more apparent when the process mining project becomes iterative, where, after the initial analysis, more views of the event data are required to obtain different insights. In that case, the saved XES files serve no purpose and the CSV file must be pre-processed once again. In contrast, our tool allows the users to go back into the Entities or Classes panel of the user interface at any time to define new ways to connect and interpret the data.

Resulting Process Models

Figures 7.2 and 7.3 show the models generated by ProM and our tool respectively for the Offer-related activities, which describe the same behavior, providing us with an evidence that our implementation for the Heuristic Miner algorithm was successfully implemented. Another thing we should

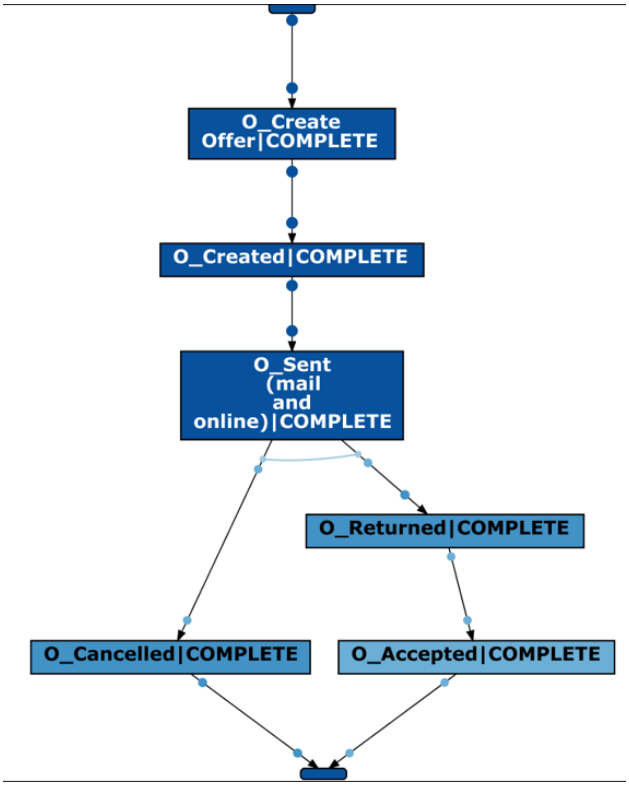


Figure 7.2: "Offers" process model discovered by the Heuristic Miner algorithm in ProM.

mention with respect to these models is that the visualization for the discovered model in ProM also showed us the Application-related activities, even if they were not connected to the start and end of the process, as shown in Figure 7.4.

Then, the models generated by ProM and our tool that include both the Offers and Applications can be observed at the top and bottom of Figure 7.5 respectively. For this second pair of process models, to improve its readability, the figure only displays the first part of the models, but this is enough for us to discuss about their similarities and differences.

As we can see on the right-hand branch of the ProM model and the top branch of the Graph tool model (both enclosed in a green box in Figure 7.5), the same behavior is described in both models where only the Application activities are involved, but there are a couple of differences with respect to the left-hand branch of the model generated by ProM and the bottom branch generated by our tool (both enclosed in an orange box in Figure 7.5). These 2 differences are described next.

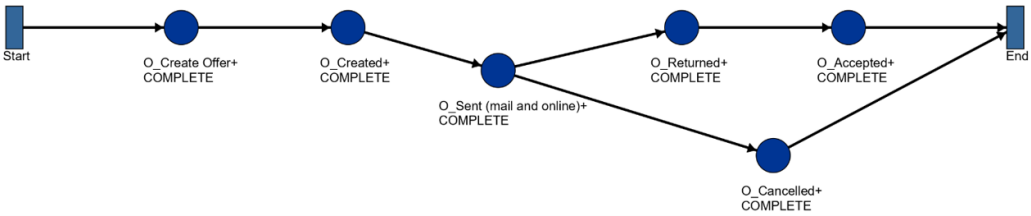


Figure 7.3: "Offers" process model discovered by the Heuristic Miner algorithm in our tool.

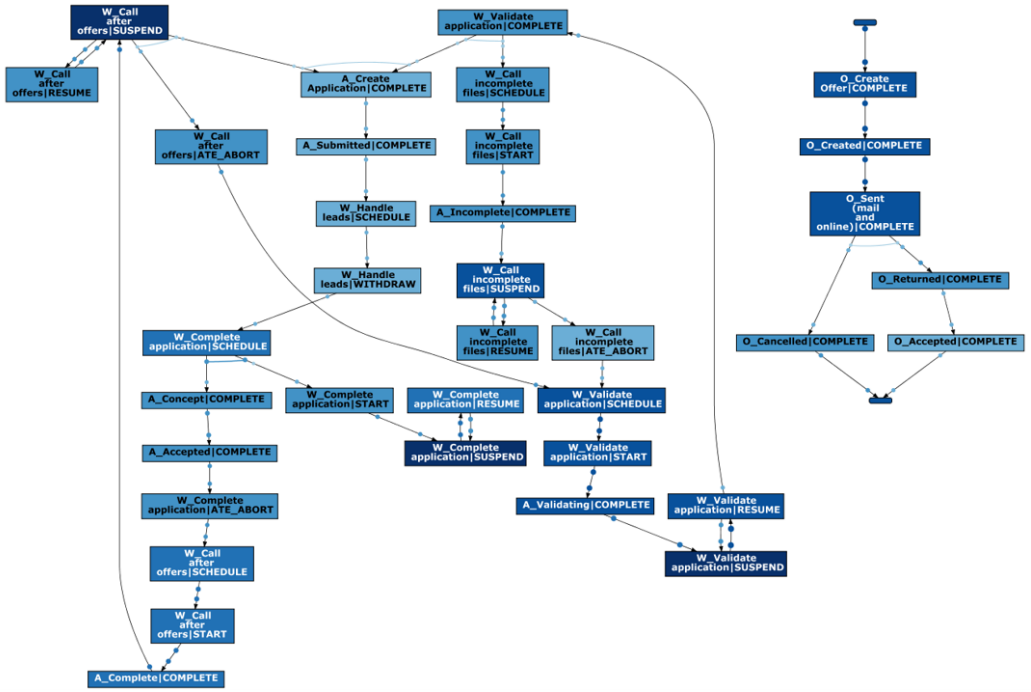


Figure 7.4: Full visualization of the "Offers" process model discovered by the Heuristic Miner algorithm in ProM.

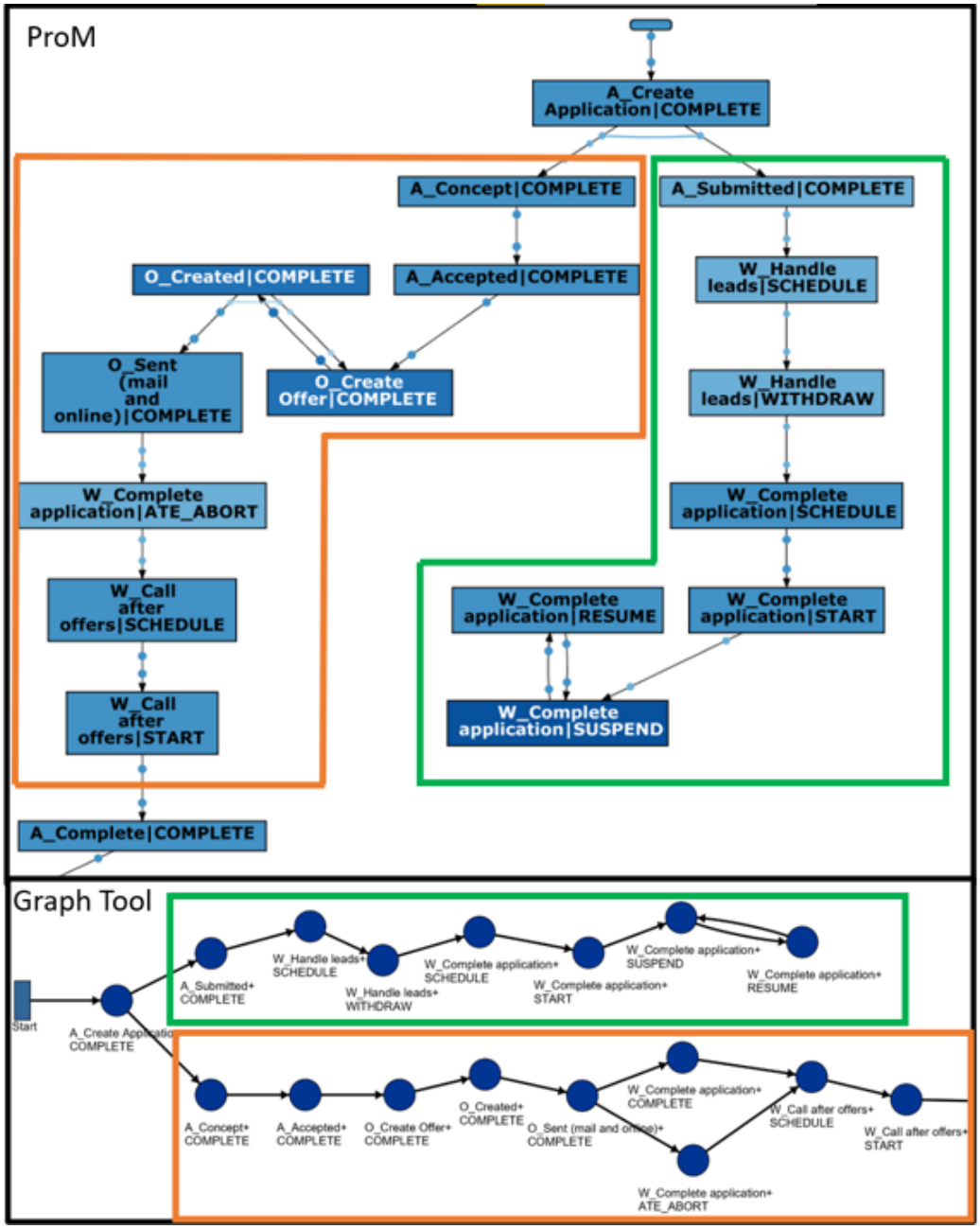


Figure 7.5: Top: "Application" process model discovered by the Heuristic Miner algorithm in ProM. Bottom: "Application" process model discovered by the Heuristic Miner algorithm in our tool.

idx	case	OfferID	Activity	lifecycle	timestamp
142	Application_232032104	Offer_1923271850	O_Create Offer	COMPLETE	2016-03-30T09:57:22.695+0100
143	Application_232032104	Offer_1923271850	O_Created	COMPLETE	2016-03-30T09:57:24.234+0100
144	Application_232032104	Offer_2146383945	O_Create Offer	COMPLETE	2016-03-30T09:58:09.174+0100
145	Application_232032104	Offer_2146383945	O_Created	COMPLETE	2016-03-30T09:58:10.715+0100
146	Application_232032104	Offer_221454371	O_Create Offer	COMPLETE	2016-03-30T09:58:35.041+0100
147	Application_232032104	Offer_221454371	O_Created	COMPLETE	2016-03-30T09:58:36.893+0100
148	Application_232032104	Offer_1614027	O_Create Offer	COMPLETE	2016-03-30T09:58:58.056+0100
149	Application_232032104	Offer_1614027	O_Created	COMPLETE	2016-03-30T09:58:59.540+0100

Figure 7.6: Offer-related activities inside an Application from the BPIC17 dataset.

Difference 1 The first difference can be found in the connections between the "O_Create Offer+COMPLETE" and the "O_Created+COMPLETE" activities. We are able to see this difference due to the value set for the Lenght-2 Loop parameter (L2L=0.7). In the ProM model, it is implied that we can move back and forth between these two activities in the process before moving on to the "O_Sent (mail and online)+COMPLETE" activity. This behavior is not represented in the graph tool model, where the process can only move forward at that point. This difference is caused by the independent dimensions we defined for the log during the import of event data in our tool, which help us address the case where we have events that could be associated to multiple entities and we need to choose one, avoiding the inclusion of unseen behavior in the model. We can confirm this by taking a closer look at some of the events of the "BPIC17_Sample_20cases.csv" log file. A simplified version of this file can be observed in Figure 7.6.

In Figure 7.6 we can see that Application_232032104 contains 8 activities related to 4 different offers. Looking at the events for this application without considering the offers as a second, separate entity (as ProM does), we can see why the process model discovered includes a connection from "O_Created+COMPLETE" to "O_Create Offer+COMPLETE", since it seems to happen 4 times only in this application. However, if we consider the Offer as a separate entity, we can see that no single offer moves back to the "O_Created+COMPLETE" activity, and that behavior is correctly identified by the model generated by our tool.

Difference 2 The second difference between the model generated by ProM and our tool is in the inclusion of the "W_Complete application+COMPLETE" in our model as an intermediate activity between the "O_Sent (mail and online)+COMPLETE" and the "W_Call after offers+SCHEDULE" activities. Taking a closer look at some of the events from the "BPIC17_Sample

idx	case	OfferID	Activity	lifecycle	timestamp
150	Application_232032104	Offer_1614027	O_Sent (mail and online)	COMPLETE	2016-03-30T09:59:34.256+0100
151	Application_232032104	Offer_221454371	O_Sent (mail and online)	COMPLETE	2016-03-30T09:59:34.259+0100
152	Application_232032104	Offer_2146383945	O_Sent (mail and online)	COMPLETE	2016-03-30T09:59:34.262+0100
153	Application_232032104	Offer_1923271850	O_Sent (mail and online)	COMPLETE	2016-03-30T09:59:34.264+0100
154	Application_232032104		W_Complete application	COMPLETE	2016-03-30T09:59:34.291+0100

Figure 7.7: Second set of Offer-related activities inside an Application from the BPIC17 dataset.

_20cases.csv" log in Figure 7.7, we can see why this is happening.

We can see that for Application_232032104 there are four "O_Sent (mail and online)+COMPLETE" activities taking place in a row, followed by a "W_Complete application+COMPLETE" activity. Again, if we do not consider the Offers as a second, separate entity, it might seem that the Offer-related activities are in a loop before finally moving on to the "W_Complete application+COMPLETE" activity. However, since we specified in our tool that the "OfferID" attribute is a separate entity, we can see that these four Offer activities belong to different offers, and they converge in the "W_Complete application+COMPLETE" activity, which marks a difference on the frequency with which this connection happens, going from one to four.

Overall, the results from this experiment help us realize that, even if the execution of our tool does not present a significant improvement in terms of performance and usability when compared to ProM, the process mining-related activities implemented in our tool on top of the graph-based data model not only provide significant results in a real-case scenario, but also provide additional, valuable insights for the process in the form of the model comparison visualization and the accurate description of behavior through our handling of multi-dimensional event data.

7.2 Experiment 2

7.2.1 Setup

The objective of the second experiment is to help us add on the answer for the research question RQ1:

RQ1. Is it possible to build on top of the data model proposed in [1] to execute process mining-related activities in a graph database?

Experiment 1 already helped us answer this question from a usability perspective, where we confirmed that with our tool it is possible to execute

#	Subset Name	# of Cases	# of Events
1	BPIC17_Sample_20cases.csv	20	765
2	BPIC17_Sample_40cases.csv	40	1596
3	BPIC17_Sample_80cases.csv	80	2883
4	BPIC17_Sample_160cases.csv	160	5808
5	BPIC17_Sample_320cases.csv	320	12297

Table 7.3: BPIC17 Subsets details

distinct process mining-related activities on top of a graph database. However, we can further explore this question by analyzing how the performance of the tool scales when we try to obtain results based on larger event logs.

Similar to what we did for the previous experiment, we should also compare its performance against other process mining tools to to put the results into perspective.

This experiment is designed to register the time it takes for the tool to complete a series of tasks. Once again, we will be using the BPIC17 dataset as our reference. From the BPIC17 dataset, we generated five random subsets of events, whose details can be observed in Table 7.3.

For every log, we use both our tool and ProM to generate and visualize one process model. This means that we need to import the data, define the Entity Type/Class and Case Attribute respectively, and generate the model using the Heuristic Miner algorithm. Similar to the previous experiment, we add additional steps in ProM and our tool to emulate the functionalities that each of them makes automatically (e.g. model visualization in ProM and data storage in our tool).

7.2.2 Execution

Table 7.4 shows the activities executed in each tool to obtain the process models for each log in Table 7.3 and observe the performance. The amount of activities performed is less than those executed for the previous experiment because the focus now is the execution times of the activities, not their functionality. We should mention that for every step executed in each tool, we count the time taken from the first interaction until the result is shown on screen. The details on how each step was executed can be observed in Appendix E.2.

After executing the steps shown in Table 7.4 and having generated a process model for each log in Table 7.3, we can proceed to show the results from the second experiment.

Objective	Activity (ProM)	Activity (Graph Tool)
Importing event log	Import event data	Import event data
Creating "Application" view	Create the "Application" XES file	Define the "Application" Entity Type
	Save the "Application" XES file	Using "Application" Entity Type, define the "Activity+Lifecycle" Class
Creating "Application" model	Create the "Application" model	Create the "Application" model
	Save the "Application" model	Visualize the "Application" model

Table 7.4: Activities executed in each tool for experiment 2.

Objective	Time (s)									
	20 cases		40 cases		80 cases		160 cases		320 cases	
	Graph	ProM	Graph	ProM	Graph	ProM	Graph	ProM	Graph	ProM
Importing event log	9	12	10	12	9	18	24	14	14	18
Creating "Application" view	20	14	20	14	19	16	36	15	27	20
Creating "Application" model	69	20	378	22	1368	28	5066	38	19760	80
Total	**98**	**46**	**408**	**48**	**1396**	**62**	**5126**	**67**	**19801**	**118**

Figure 7.8: Execution times for logs containing different number of cases.

7.2.3 Results

Figure 7.8 shows the execution times for each objective for the different number of cases included in the logs. Then, Figure 7.9 shows a plot where we can see how the execution times (in minutes) scale up for each tool. In this second figure we can see that, even when the number of cases increases from 20 to 320, the execution time for ProM only takes around 2 times longer, while for our tool the time increases by approximately 4 times just from the scale up from 20 to 40 cases. This can also be observed on the scale up from 160 to 320 cases.

Based on the breakdown of the execution times shown in Figure 7.8, which indicates that the creation of the process model is what takes most of the time, we assume that the complexity of the queries built to execute the Heuristic Miner algorithm, especially those that compute the input and output bindings (which need to query through all the events connected by :DF paths), are the main reason why this activity takes longer. We should also mention that the biggest log used for this experiment, which consists of 320 cases, represents ∼1% of the total number of cases included in the BPIC17 dataset (31,509 cases in total).

From the results of this second experiment we can say that there is a significant increment in the execution times for our tool with larger datasets, which becomes even more apparent when compared against ProM. Other than revising the process discovery queries to find ways to optimize their performance, we believe that the tool could also benefit from the implementation of the third layer defined in Figure 3.1, the View Layer, whose activities

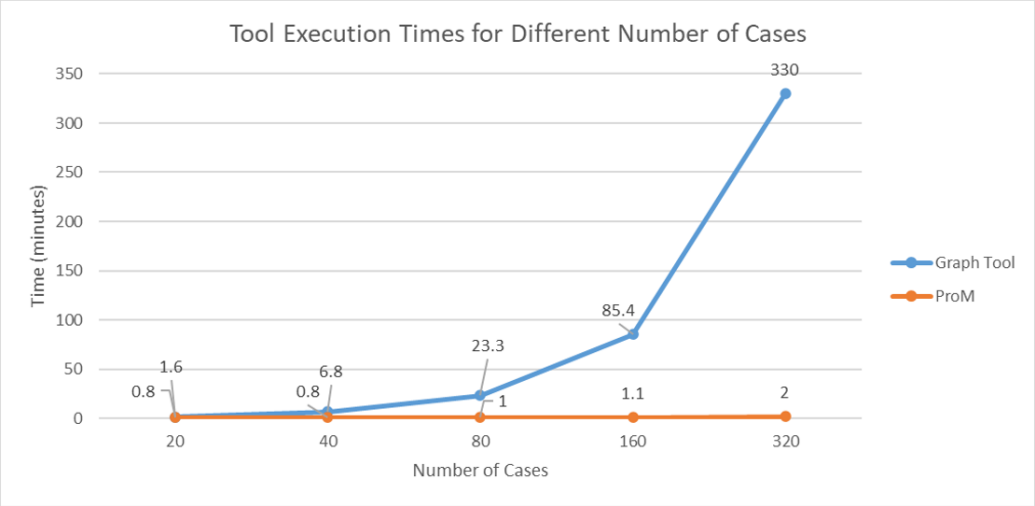


Figure 7.9: Plot showing how the execution times (in minutes) scale up for each tool.

include the filtering and aggregation of events, both of which help in reducing the complexity of the log and potentially help diminish the impact of the long execution times when discovering a process model.

7.3 Experiment 3

7.3.1 Setup

The objective of the third experiment is to help us answer research question RQ3:

RQ3. Is there an added benefit on the joint storage of different process mining components in the graph database?

In Section 5.1, we defined a new relation called :REPRESENTS that connects the activities from a process model with the event classes. We were able to do this given that both components are stored in the same graph database, represented by different types of nodes and relations. In that section we established that this connection could help us to potentially address challenge C10 of the Process Mining Manifesto [3], which mentions that it is necessary to improve usability for non-experts by linking the event data with process models to provide valuable interactions with end-users.

To determine if we can provide valuable interactions through the :REPRESENTS relation, the experiment is designed to check how the connection

between models and events can be used to obtain additional insights on the process being evaluated.

As we discussed in Section 3.1, another activity that should be considered during the *Mining and Analysis* stage of a process mining project is the *Conformance Checking*, where we can detect inconsistencies between a process model and the event log used to discover it. This activity is not only mentioned in the PM^2 methodology [2], but also in the Process Mining Manifesto [3], highlighting its importance. Although building a conformance checking functionality is outside the scope of the current implementation, we can define an example query that shows how the connection between models and events provides a guideline for a future possible implementation of conformance-checking activities.

In summary, the third experiment is about defining a query that shows how the connection between Model and Class nodes can provide value to end-users by allowing us to see how the behavior described by a process model represents the event data, and thus providing a path for a future conformance checking implementation.

7.3.2 Execution

Since Conformance Checking is used to check whether a model correctly represents the activities recorded in the event log, we consider that a first step towards this can be the identification of those activities that appear in the event log but are not represented in the model; with this information, we can already obtain a first answer on how many traces (or entity type paths in our case) cannot be fully replayed in the generated model.

The query shown below helps us compute which activities were excluded from a given model and, for those activities, count the number of distinct entity type paths in which they appear. In line 1, we specify the name of the model with the *ModelID* passed as argument and we make use of the :REPRESENTS relation to obtain the Class type used to generate the model in line 2. Now that we now the Class type, in line 3 we traverse the graph from the other end of the process model to retrieve all the existing Class nodes for that type. If we retrieved the Class nodes using the :REPRESENTS relations, we would miss the existing Class nodes that are not represented in the model. Then, in line 4, we filter these Class nodes to keep only those that do not have a connection with the process model. Then, with the remaining Class nodes stored in variable "classNode" in line 5, in line 6 we traverse the graph to reach the Entity nodes and retrieve those whose :DF relations were used to connect the events (we identify the correct set of Entity nodes through the *EntityType* argument). Then, in line 7, we return the IDs of the

Class nodes that represent the model activities together with the number of traces (Entity Type paths) where these activities appear. We order the results by number of appearances first and by Class ID second in line 8.

```
1 MATCH (m:Model{ID:ModelID})--(dg:DG_node)-[:REPRESENTS]->(c:
    Class)
2 WITH DISTINCT m AS ProcessModel, c.Type AS ClassType
3 MATCH (ProcessModel)--(:Algorithm)--(:Log)--(:Event)--(c:
    Class{Type:ClassType})
4 WHERE NOT EXISTS((ProcessModel)--(:DG_node)-[:M_C]->(c))
5 WITH c AS classNode
6 MATCH (classNode)--(:Event)--(en:Entity{EntityType:EntityType})
7 RETURN classNode.ID AS ClassActivity, COUNT(DISTINCT en) AS
    EntityPathAppearances
8 ORDER BY EntityPathAppearances DESC, ClassActivity
```

To execute the query, we need to define the model ID and the Entity Type. The model we use as an input is the "Application" model, whose ID is "HM_2" and was originally generated as part of the experiment described in Section 7.1. This model describes the behavior of the BPIC17 sample dataset for the Application and Offer activities. The inputs used to generate this model for that experiment were the 765 Event nodes that make up 20 applications of the BPIC17 dataset, 27 derived Entity nodes, whose entity type is "OfferIDcase" and describe the relation between the Application and Offer events, and 50 Class nodes, which describe the model activities through the "Activity+Lifecycle" event class.

7.3.3 Results

Using (ModelID="HM_2") and (EntityType="OfferIDcase") as the parameters for the query shown in Section 7.3.2, we obtain the results shown in Table 7.5.

In Table 7.5 we can see that 11 out of the 50 distinct Class activities were excluded from the model, with "W_Handle leads+COMPLETE" and "W_Handle leads+START" appearing in 4 out of 27 entity type paths. This means that, in the best case scenario, the process model can replay 85.2% (23 out of 27) of the traces (entity paths).

The result of this experiment shows us that the connection between models and events (through the Class nodes) can represent value to the users by providing insights on how well the model is representing an event log.

In addition, this connection could in theory be visualized in the tool. The user interface allows us to display any type of nodes and relations, so we can assume that an additional functionality can be implemented where, based on a given model ID, the nodes that represent the model and the Class

ClassActivity	EntityPathAppearances
W_Handle leads+COMPLETE	4
W_Handle leads+START	4
A_Denied+COMPLETE	2
O_Refused+COMPLETE	2
O_Sent (online only)+COMPLETE	1
W_Call after offers+COMPLETE	1
W_Call after offers+WITHDRAW	1
W_Call incomplete files+COMPLETE	1
W_Complete application+WITHDRAW	1
W_Handle leads+RESUME	1
W_Handle leads+SUSPEND	1

Table 7.5: Excluded activities from the "Application" model.

or Event nodes correlated to it can be retrieved and displayed in the Graph panel, similar to what is already done for the model comparison functionality described in Section 6.2.3.

7.4 Experiment 4

7.4.1 Setup

The objective of the fourth experiment is to help us add on the answer for research question RQ3:

RQ3. Is there an added benefit on the joint storage of different process mining components in the graph database?

In our tool, we were able to implement several distinct functionalities thanks to the storage of different process mining components as nodes and relations of a graph. Therefore, we should also check if the sum of all these functionalities represents any additional benefit, and to do this, we can refer to the Process Mining Manifesto [3] and check if our tool helps address some of the challenges related to process mining.

As mentioned in Section 2.1.3, the manifesto lists 11 challenges that need to be addressed to ensure that this discipline can continue providing value for organizations. For this experiment, we first list all the main functionalities implemented for our tool and then we try to identify which of them help address one or more of these challenges, allowing us to determine how our

Graph Tool Functionalities	
Upload CSV	Define Entity Type Attributes
Define Dimensions	Create Classes
Clear Database	Discover Models: Heuristic Miner
Set/Clear Constraints	Display Available Models
Display Available Logs	Show Model
Delete Log	Show Petri Net
View Log Details	View Model Details
View Instance Level Nodes	Delete Model
View Model Level Nodes	Model Comparison
View Node Details	Find Models
Display Nodes with Sugiyama Layout	Storage of Process Mining Data
Create Entities/Derived Entities	

Table 7.6: List of the 23 functionalities implemented for our tool.

tool provides additional benefits by contributing to the discipline of process mining.

7.4.2 Execution

After doing an inspection through our implementation and the different ways in which we can interact with the tool through the user interface, we identified 23 different functionalities that were implemented. The list of these functionalities can be observed in Table 7.6.

Based on the description of the challenges provided in the Manifesto, we identified those functionalities that helped addressing one or more of them.

7.4.3 Results

Table 7.7 shows the functionalities that address at least one challenge, with a total of 4 challenges addressed by our implementation. From the 23 functionalities identified, 17 of them are included in this table. The functionalities excluded are *Upload CSV*, *Clear Database*, *Set/Clear Constraints*, *Delete Log*, *Define Entity Type Attributes*, and *Delete Model*. We already described in the previous sections why these 6 functionalities are required for the correct operation of the tool, but while they address a need for our implementation, we did not identify any way in which they address one of the challenges. The way in which each challenge is addressed is described next.

	Functionality	C1	C9	C10	C11
F1	Define Dimensions	X		X	
F2	Display Available Logs			X	
F3	View Log Details			X	
F4	View Instance Level Nodes		X	X	X
F5	View Model Level Nodes		X	X	X
F6	View Node Details		X	X	
F7	Display Nodes with Sugiyama Layout				X
F8	Create Entities/Derived Entities	X			
F9	Create Classes	X			
F10	Discover Models: Heuristic Miner			X	
F11	Display Available Models			X	
F12	Show Model		X		X
F13	Show Petri Net				X
F14	View Model Details		X	X	
F15	Model Comparison		X		X
F16	Find Models			X	
F17	Storage of Process Mining Data	X		X	

Table 7.7: Process Mining Manifesto challenges addressed by tool functionalities.

Challenge C1 The first challenge addressed by the tool is C1, *Finding, Merging and Cleaning Event Data*. More specifically, the tool addresses two of the hurdles described in this challenge. The first hurdle talks about the need to add context to the events by merging event data with context data. Functionality F1 handles this by allowing the user to specify the existing dimensions in the event data, providing a guideline to obtain more accurate representations of the data. The second hurdle refers to the need to turn event data into "process centric" rather than "object centric" [3], often times needing merging and preprocessing to make the change, but with functionalities F8 and F9 building on the concepts of the data model proposed in [1], the user is able to define the connections between the data, creating entities and classes that best describe the process by relating events based on the selected attribute. Since the details of the dimensions, entities, and classes are stored in the graph database, we can also consider that functionality F17 is also helping to address these hurdles.

Challenge C9 The second challenge addressed by the tool is C9, *Combining Process Mining With Other Types of Analysis*, specifically when this challenge describes the importance of combining process mining with *visual*

analytics, which "combines automated analysis with interactive visualizations for a better understanding of large and complex data sets" [3]. Functionalities F4, F5, F6, F12, F14, and F15 all address this since they use interactive visualizations to provide a better understanding of the different steps of process mining. The usage of interactive visualizations is the following: If the Event, Entity and/or Class nodes that are displayed in F4 and F5 are clicked, the table in F6 will be populated with their properties; clicking on a model node displayed by F12 will populate the table of F14 with the input and output bindings of that node, and finally the comparison of two Petri nets provided by F15 enables a new interaction where a click on an activity of the first Petri net will highlight the activity or activities with the same name in the second Petri net.

Challenge C10 The third challenge addressed by the tool is C10, *Improving Usability for Non-Experts*, which refers to the creation of user-friendly interfaces and the linking of event data with process models to provide valuable interactions with end-users [3]. Given that a GUI was developed, it is normal to expect that this is the challenge covered by most features. The design of the tool selection windows in F1, F4, F5, F10, and F16 help the user understand what to do to interact with the tool, while the tables in F2, F3, F6, F11, and F14 display relevant information about the logs, process models, or nodes currently displayed in the graph panel of the user interface. Then, in order to provide the event data and process model interactions for the end-users, the tool stores in the database the connection between Model and Class nodes after generating a process model, therefore F17 also helps to address this challenge.

Challenge C11 Finally, the tool also addresses challenge C11, *Improving Understandability for Non-Experts*, which, among other things, mentions that results should be presented using a suitable representation [3]. The visualizations in F4, F5, F12, F13, and F15 take advantage of F7, which adapts the Sugiyama algorithm to display the nodes.

The results from this experiment help us determine that several functionalities implemented for the tool not only provide the necessary steps to execute a process mining project, but also make a contribution in the process mining field by addressing 4 of the challenges described by the Process Mining Manifesto.

7.5 Discussion

In this section, we further discuss how the experiments helped us answer the three research questions proposed initially.

RQ1. Is it possible to build on top of the data model proposed in [1] to execute process mining-related activities in a graph database?

Experiment 1 helped us realize that, with a tool built on top of a graph database, using as a basis the data model proposed in [1], we can execute the basic steps of a process mining project that require an interaction with a tool and obtain significant results that provide value for the analysis of the process being evaluated. In addition, this experiment showed us that we can not only replicate what ProM does, but thanks to the flexibility provided by the data model to specify the dimensionality of the data, we can obtain results that more accurately represent the process.

However, experiment 2 then showed us that there is still room for improvement for the tool with respect to its usability. While our implementation does provide several benefits such as the storage of all the process mining-related components in one place or the definition of data dimensionality to obtain more accurate results, its performance with larger datasets is something that could be improved to make this tool a more viable option for any kind of process mining project.

We should also mention that these results were obtained almost exclusively with the usage of the graph database through Cypher queries with minimal support from Java, which, other than for the implementation of the user interface, was only used in a couple of occasions to deal with loops that cannot be handled easily by Cypher (e.g. the iterations in the Heuristic Miner to connect the activities with missing inputs or outputs, as described in Section 5.2.6).

RQ2. Can we store process models in the graph database? If so, how would the execution of process mining change in this environment?

Experiment 1 also showed us that we can not only store process models inside the graph database, but we can also execute process discovery algorithms to create them. Additionally, this experiment also showed us that to obtain these results, the way in which process mining is executed changes when we use our tool, since the outputs from previous stages differ in the graph-based environment.

In regular process mining projects, where we usually have to define the case identifier for the log, which remains constant for the duration of the analysis, now we can define multiple entities, derived entities and classes that provide different perspectives of the same event log and allow us to discover new process models, enriching the analysis.

The ease with which our tool allows us to define new entity identifiers and event classes to discover new process models makes the overall execution of a process mining project a more iterative proposition, where we can jump back from the model analysis to the definition of views at any point and define new ways to connect the data without losing the information generated previously, given that everything remains stored in the graph database.

RQ3. Is there an added benefit on the joint storage of different process mining components in the graph database?

Experiment 3 helped us realize that storing the models together with the event data can provide valuable insights on the process under evaluation. The connection between models and events was not only useful to define the queries that execute the Heuristic Miner algorithm, but it also provides a path to expand on our current implementation and be able to address more process mining-related activities such as conformance checking.

Then, experiment 4 showed us the benefits of developing a tool under a data model that integrates several different process mining components. The functionalities provided by our tool, made possible by the representation of events, entities, classes and models exclusively through nodes and relations, not only provide value by allowing us to define views or discover models which can later be queried or compared, but it is the fact that all of these activities can be done in one place that increases its positive impact. Even now we can already identify that, in a future update, the tool could go one step further and be configured to show how the models and event data relate to each other in the same visualization, providing yet another valuable interaction for process mining projects. As we saw in the results of experiment 4, we were able to address multiple process mining challenges with several tool functionalities thanks to this integrated approach.

Chapter 8

Conclusion

In this thesis, we discuss the implementation of a tool that is able to build on top of a graph-based data model to execute several process mining-related activities, including the data import into the graph database, the creation and enrichment of event logs, process discovery, and process diagnosis. To evaluate the tool, we executed several experiments to test its usability, performance and contributions to the discipline of process mining.

To the best of our knowledge, there is no existing process mining tool that takes advantage of the benefits of the graph-based data models to handle multi-dimensional data and execute process mining-related activities. This is why we aimed to determine the feasibility of using one of these data models to obtain a viable alternative to execute process mining.

For our implementation, we referred to the PM^2 methodology [2] to identify the main activities of a process mining project that are executed with the help of a tool to determine what a tool built on top of a graph database should be able to execute. Based on these activities, we defined a 5-layer architecture that provided the guideline for the implementation of our tool. The 5 layers are: (1) Event layer, (2) Entity and Behavior Layer, (3) View Layer, (4) Model layer, and (5) Model Analysis layer.

From the main list of activities, we identified the most essential that provide valuable insights into the process under evaluation and confirm the feasibility of the graph-based approach, which helped delimit the scope of our tool. As a result, the activities from the third layer were excluded from our implementation, whose objective was to execute activities that create data subsets or reduce the complexity of the log. As part of our scope, we considered the implementation of an intuitive user interface to simplify the execution of the activities.

Using as a basis the graph-based data model and queries presented by Esser and Fahland in [1], we implemented the first two layers of the tool,

where we defined the way to import the data and define and enrich views of the event data through entities and classes.

Then, we presented our extension to the data model from [1] by defining three new node types in the form of the Algorithm, Model and Model_node. These nodes help us represent process models in the graph database and connect them with the rest of the data model. Using this extended data model, we implemented a process discovery algorithm, the Heuristic Miner, where we define queries that were able to successfully generate a process model that describes the behavior registered in an event log, completing our implementation of the Model layer.

Then, to implement the Model Analysis layer and allow users to perform diagnosis on the discovered models, we implemented two more functionalities in the graph-based tool, the model comparison and the model querying. For the model comparison, we identified the need to translate the original output of the Heuristic Miner, the dependency graph, into a Petri net, to allow the comparison of any pair of process models that can be represented by Petri nets, instead of considering exclusively dependency graphs. Then, for the model querying, we define 6 search patterns to query for models already stored in the graph database, providing users with an alternative to find the models to analyze.

To evaluate the tool, we defined 4 experiments to test the usability, performance and contributions of our tool. The results from these experiments helped us confirm that it is not only possible to execute process mining on top of a graph database, but there are some added benefits by doing so. We were able to confirm that with the graph data model we are able to discover process models that correctly describe the behavior of event logs that contain multi-dimensional data, as evidenced by the differences between the models generated by our tool and those generated by the process mining tool ProM. The joint storage of distinct process mining components proved to be an additional advantage of our approach, since it allowed us to create connections and implement functionalities that may not be available in other environments. These include the connection between process models and the event data, which provide another way to analyze how the models represent the behavior of the event log, and the usage of the database as a model repository, which allows us to provide model querying and model comparison as additional tools for users in their analysis.

Overall, the tool also has an impact on the regular approach to process mining, since the sum of these functionalities integrated in one tool make for a more iterative approach, where the steps from the definition of views to the comparison of discovered process models can be easily replayed through the user interface, allowing for more flexibility in the analysis. In addition,

the implementation of the user interface invites for users with less or no experience in the usage of graphs for process mining to work with the tool and explore the possibilities it provides without the need to define complex Cypher queries from the start [1].

8.1 Limitations and Future Work

The current execution time of the tool to discover a process model through our implementation of the Heuristic Miner algorithm may present an important limitation of our tool in practice. Further improvements on the performance of the algorithm queries are required, where perhaps the directly follows connections between event classes could be further exploited to avoid the extensive search in the event data that is required for one of the steps of the algorithm.

Another alternative that could have a positive impact on the performance of the algorithm is the implementation of filtering and aggregation activities on the event data. These activities, which are part third layer in the architecture described in Figure 3.1 (View Layer), were excluded from our scope since they were not considered essential to test the feasibility of working with process mining in a graph database, but they could prove valuable to reduce the complexity of the log, and therefore improve the performance of the algorithm. However, it remains an open question how other process discovery algorithms would perform using our proposed graph-based data model as a basis.

In this regard, we should say that our implementation does consider that further functionalities can be added in the future other than the already mentioned *View Layer*. First, the extended data model was designed with the intention of describing not only the dependency graphs generated by the Heuristic Miner algorithm, but also any modeling language that can be represented through a graph structure. Then, the representation of Petri nets established in our data model invites for more discovery algorithm to be added into the graph environment with the certainty that they can be compared with other process models through our user interface.

Finally, future work can also include the development of additional process mining-related activities such as process enhancement or conformance checking, with a suggested start to approach the latter already provided as part of the evaluation done in this thesis.

[1] The full implementation of the work done for this thesis, together with the data subsets used for its evaluation, is available at the following GitHub repository: https://github.com/vhernandezs/event-graph-process-mining

Bibliography

[1] Esser, S., & Fahland, D. (2021). Multi-Dimensional Event Data in Graph Databases. *Journal on Data Semantics*, 10, 109–141. https://doi.org/10.1007/s13740-021-00122-1

[2] Eck, van, M. L., Lu, X., Leemans, S. J. J., & van der Aalst, W. M. P. (2015). PM2 : a Process Mining Project Methodology. In J. Zdravkovic, M. Kirikova, & P. Johannesson (Eds.), *Advanced Information Systems Engineering : 27th International Conference, CAiSE 2015, Stockholm, Sweden, June 8-12, 2015. Proceedings* (pp. 297-313). (Lecture Notes in Computer Science; Vol. 9097). Springer. https://doi.org/10.1007/978-3-319-19069-3_19

[3] van der Aalst W. et al. (2012) Process Mining Manifesto. In: Daniel F., Barkaoui K., Dustdar S. (eds) *Business Process Management Workshops. BPM 2011. Lecture Notes in Business Information Processing, vol 99*. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-28108-2_19

[4] Emamjome F., Andrews R., ter Hofstede A.H.M. (2019) A Case Study Lens on Process Mining in Practice. In: Panetto H., Debruyne C., Hepp M., Lewis D., Ardagna C., Meersman R. (eds) *On the Move to Meaningful Internet Systems: OTM 2019 Conferences. OTM 2019. Lecture Notes in Computer Science, vol 11877*. Springer, Cham. https://doi.org/10.1007/978-3-030-33246-4_8

[5] Nikolov N.S. (2016) Sugiyama Algorithm. In: Kao MY. (eds) *Encyclopedia of Algorithms*. Springer, New York, NY. https://doi.org/10.1007/978-1-4939-2864-4_649

[6] Van der Aalst, W.M. (2016). *Process mining: data science in action*. (2nd ed.). Springer. https://doi.org/10.1007/978-3-662-49851-4

[7] Augusto, A., Conforti, R., Dumas, M., La Rosa, M., Maggi, F., Marrella, A., Mecella, M. & Soo, A. (2017). Automated Discovery of Process

Models from Event Logs: Review and Benchmark. *IEEE Transactions on Knowledge and Data Engineering*. PP. 10.1109/TKDE.2018.2841877.

[8] Weijters, A. & Ribeiro, J. (2011). Flexible Heuristics Miner (FHM). *Journal of Applied Physiology - J APPL PHYSIOL*. 310-317. 10.1109/CIDM.2011.5949453.

[9] Mannhardt F., de Leoni M., Reijers H.A., van der Aalst W.M.P. (2017) Data-Driven Process Discovery - Revealing Conditional Infrequent Behavior from Event Logs. In: Dubois E., Pohl K. (eds) *Advanced Information Systems Engineering. CAiSE 2017. Lecture Notes in Computer Science, vol 10253.* Springer, Cham. https://doi.org/10.1007/978-3-319-59536-8_34

[10] Weijters, A. J. M. M., Aalst, van der, W. M. P., & Alves De Medeiros, A. K. (2006). *Process mining with the HeuristicsMiner algorithm*. (BETA publicatie : working papers; Vol. 166). Technische Universiteit Eindhoven.

[11] Buijs, J. C. A. M., Dongen, van, B. F., & Aalst, van der, W. M. P. (2014). Quality dimensions in process discovery : the importance of fitness, precision, generalization and simplicity. *International Journal of Cooperative Information Systems*, 23(1), 1440001/1-39. https://doi.org/10.1142/S0218843014400012

[12] Aalst, van der, W. M. P., Alves De Medeiros, A. K., & Weijters, A. J. M. M. (2006). Process equivalence : comparing two process models based on observed behavior. In S. Dustdar, J. L. Fiadeiro, & A. Sheth (Eds.), *Business Process Management (Proceedings 4th International Conference, BPM 2006, Vienna, Austria, September 5-7, 2006)* (pp. 129-144). (Lecture Notes in Computer Science; Vol. 4102). Springer. https://doi.org/10.1007/11841760_10

[13] Wang, J., Jin, T., Wong, R.K. et al. Querying business process model repositories. *World Wide Web* 17, 427–454 (2014). https://doi.org/10.1007/s11280-013-0210-z

[14] Kunze, M., Weidlich, M. & Weske, M. Querying process models by behavior inclusion. *Softw Syst Model* 14, 1105–1125 (2015). https://doi.org/10.1007/s10270-013-0389-6

[15] van Dongen, B.: BPI Challenge 2017. Dataset. https://doi.org/10.4121/uuid: 5f3067df-f10b-45da-b98b-86ae4c7a310b

[16] Desel, J. and Esparza, J. Free Choice Petri Nets,volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1995

[17] Buijs, J. C. A. M., Dongen, van, B. F., & Aalst, van der, W. M. P. (2012). A genetic algorithm for discovering process trees. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2012, Brisbane, Australia, June 10-15, 2012)* (pp. 1-8). Institute of Electrical and Electronics Engineers. https://doi.org/10.1109/CEC.2012.6256458

[18] van der Aalst W., Adriansyah A., van Dongen B. (2011) Causal Nets: A Modeling Language Tailored towards Process Discovery. In: *Katoen JP., König B. (eds) CONCUR 2011 – Concurrency Theory. CONCUR 2011.* Lecture Notes in Computer Science, vol 6901. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-23217-6_3

[19] van Eck, M. L., Sidorova, N., & van der Aalst, W. M. P. (2016). Discovering and Exploring State-based Models for Multi-perspective Processes. In M. La Rosa, P. Loos, & O. Pastor (Eds.), *Business Process Management : 14th International Conference, BPM 2016, Rio de Janeiro, Brazil, September 18–22, 2016. Proceedings* (pp. 142-157). (Lecture notes in computer science; Vol. 9850). Springer. https://doi.org/10.1007/978-3-319-45348-4_9

[20] Conforti R., Dumas M., García-Bañuelos L., La Rosa M. (2014) Beyond Tasks and Gateways: Discovering BPMN Models with Subprocesses, Boundary Events and Activity Markers. In: *Sadiq S., Soffer P., Völzer H. (eds) Business Process Management. BPM 2014.* Lecture Notes in Computer Science, vol 8659. Springer, Cham. https://doi.org/10.1007/978-3-319-10172-9_7

[21] Maggi, F. M., Mooij, A. J., & Aalst, van der, W. M. P. (2011). User-guided discovery of declarative process models. In N. Chawla, I. King, & A. Sperduti (Eds.), *Proceedings of the IEEE Symposium on Computational Intelligence and Data Mining (CIDM 2011, Paris, France, April 11-15, 2011)* (pp. 192-199). Institute of Electrical and Electronics Engineers. https://doi.org/10.1109/CIDM.2011.5949297

[22] Dijkman, R. M., La Rosa, M., & Reijers, H. A. (2012). Managing large collections of business process models - current techniques and challenges. *Computers in Industry, 63*(2), 91-97. https://doi.org/10.1016/j.compind.2011.12.003

[23] Robinson, I., Webber, J., and Eifrem, E. *Graph databases*. O'Reilly Media, Inc., 2nd edition, 2015.

[24] Mahmud, S. M. H., Hossin, M., Jahan, H. Noori, S., Bhuiyan, T. Csv-annotate: Generate annotated tables from csv file. In *2018 International Conference on Artificial Intelligence and Big Data (ICAIBD)*, pages 71-75, 2018.

[25] GraphStream. https://graphstream-project.org/. Accessed on 2021-08-05.

[26] Dijkman, R., Gao, J., Syamsiyah, A., van Dongen, B., Grefen, P., & ter Hofstede, A. (2020). Enabling efficient process mining on large data sets: realizing an in-database process mining operator. *Distributed and Parallel Databases*, 38(1), 227-253. https://doi.org/10.1007/s10619-019-07270-1

[27] Romero, M. & Rodriguez, A. (2007). *A Graph-Oriented Model and Query Language for Events*. 4802. 358-367. 10.1007/978-3-540-76292-8_42.

[28] *Event log extraction from SAP ECC 6.0*. Piessens, D. A. M. (Author). 30 Apr 2011

[29] Gansner, E., Koutsofios, E., North, S. & Vo, K.-P. (1993). A Technique for Drawing Directed Graphs. Software Engineering, *IEEE Transactions on. 19*. 214 - 230. 10.1109/32.221135.

[30] *Process Mining on Event Graph Databases: Exploratory Case Study on an ITIL Incident Analysis*. Türkyılmaz, P. (Author). 28 Aug 2020

List of Figures

List of Tables

Appendix A

Sugiyama Algorithm

In this appendix we describe our implementation of the Sugiyama algorithm, which is used to display the nodes retrieved from the database in the user interface. First, we describe the main steps of our implementation. Then, we proceed to show the Java code.

Our implementation of the Sugiyama algorithm is based on [29]. In this approach, they propose a technique consisting of 4 steps to draw directed graphs: (1) *Rank*, where the nodes are assigned in layers, (2) *Order*, to order the nodes within each layer to avoid edge crossings, (3) *Position*, to set the layout coordinates of the nodes, and (4) *Drawing edges*, to define the control points for edges.

For our implementation, we excluded the last step and we included a step prior to the *Rank* to remove the cycles and make the graph acyclic, which is a prerequisite to execute the Sugiyama algorithm.

The Java code for the functions that represent the four main steps of our implementation is shown below. The full implementation can be found in the GitHub repository available at https://github.com/vhernandezs/event-graph-process-mining.

1. Remove Cycles

```
1  private void removeCycles(){
2    List<Node> nodes = graph.nodes().collect(Collectors.toList());
3
4    for(Node n : nodes){
5      dfsRemove(n);
6    }
7  }
8
9  private void dfsRemove(Node node){
10   if(marked.contains(node.getId())) return;
11
12   marked.add(node.getId());
```

```
13    stack.add(node.getId());
14
15    List<Edge> outgoingEdges =
        node.leavingEdges().collect(Collectors.toList());
16    for(Edge e : outgoingEdges){
17      if(stack.contains(e.getTargetNode().getId())){
18        if(removeCycles) allEdges.remove(e);
19        else changeEdgeDirection(e);
20      }else if(!marked.contains(e.getTargetNode().getId())){
21        dfsRemove(e.getTargetNode());
22      }
23    }
24
25    stack.remove(node.getId());
26  }
```

2. Rank

```
1   private void assignLayers(){
2     List<Edge> edges = new ArrayList<>(allEdges);
3     List<Node> nodes = graph.nodes().collect(Collectors.toList());
4
5     List<Node> start = getVerticesWithoutIncomingEdges(edges, nodes);
6     while(start.size() > 0){
7       layers.add(start);
8       List<Node> finalStart = start;
9       edges = edges.stream().filter(e ->
        !finalStart.contains(e.getSourceNode())).collect(Collectors.toList());
10      nodes.removeAll(start);
11      start = getVerticesWithoutIncomingEdges(edges, nodes);
12    }
13  }
14
15  private static List<Node> getVerticesWithoutIncomingEdges(List<Edge> edges,
      List<Node> nodes){
16    List<Node> targets = edges.stream().map(Edge::getTargetNode).
17          distinct().collect(Collectors.toList());
18
19    return nodes.stream().filter(n ->
        !targets.contains(n)).collect(Collectors.toList());
20  }
```

3. Order

```
1   private void orderVertices(){
2     createVirtualVerticesAndEdges();
3
4     for(int i = 0; i < 4; i++){ //Paper "A Technique for Drawing Directed
        Graphs" indicates i < 24
5       median(i);
6       transpose();
7     }
8   }
9
10  private void createVirtualVerticesAndEdges(){
11    int virtualIndex = 0;
12    int virtualEdgeIndex = 0;
13
14    for(int i = 0; i < layers.size()-1; i++){
15      List<Node> currentLayer = layers.get(i);
16      List<Node> nextLayer = layers.get(i+1);
```

```
17      for(Node n : currentLayer){
18        List<Edge> outgoingMulti = allEdges.stream()
19            .filter(e->e.getSourceNode() == n)
20            .filter(e->
    Math.abs(getLayerNumber(e.getTargetNode())-getLayerNumber(n)) > 1)
21            .collect(Collectors.toList());
22        List<Edge> incomingMulti = allEdges.stream()
23            .filter(e->e.getSourceNode() == n)
24            .filter(e->
    Math.abs(getLayerNumber(e.getSourceNode())-getLayerNumber(n)) > 1)
25            .collect(Collectors.toList());
26        for(Edge e : outgoingMulti){
27          graph.addNode("v_" + virtualIndex);
28          Node virtualNode = graph.getNode("v_" + virtualIndex);
29          nextLayer.add(virtualNode);
30          virtualIndex++;
31
32          allEdges.remove(e);
33
    graph.addEdge("ve_"+virtualEdgeIndex,e.getSourceNode(),virtualNode,true);
34          allEdges.add(graph.getEdge("ve_"+virtualEdgeIndex));
35          virtualEdgeIndex++;
36
    graph.addEdge("ve_"+virtualEdgeIndex,virtualNode,e.getTargetNode(),true);
37          allEdges.add(graph.getEdge("ve_"+virtualEdgeIndex));
38          virtualEdgeIndex++;
39        }
40        for(Edge e : incomingMulti){
41          graph.addNode("v_" + virtualIndex);
42          Node virtualNode = graph.getNode("v_" + virtualIndex);
43          nextLayer.add(virtualNode);
44          virtualIndex++;
45
46          allEdges.remove(e);
47
    graph.addEdge("ve_"+virtualEdgeIndex,virtualNode,e.getTargetNode(),true);
48          allEdges.add(graph.getEdge("ve_"+virtualEdgeIndex));
49          virtualEdgeIndex++;
50
    graph.addEdge("ve_"+virtualEdgeIndex,e.getSourceNode(),virtualNode,true);
51          allEdges.add(graph.getEdge("ve_"+virtualEdgeIndex));
52          virtualEdgeIndex++;
53        }
54      }
55    }
56  }
57
58  private void median(int i){
59    if(i%2 == 0){
60      for(int j = 1; j < layers.size(); j++){
61        Map<String,Double> median = new HashMap<>();
62        for(Node n : layers.get(j)){
63          median.put(n.getId(),getMedianValue(n,j-1, 1));
64        }
65        sortLayer(layers.get(j),median);
66      }
67    }else{
68      for(int j = layers.size()-2; j >= 0; j--){
69        Map<String,Double> median = new HashMap<>();
70        for(Node n : layers.get(j)){
71          median.put(n.getId(),getMedianValue(n,j+1, 2));
72        }
```

```
      sortLayer(layers.get(j),median);
    }
  }
}

private void transpose(){
  boolean improved = true;
  while(improved){
    improved = false;
    for(int i = 0; i < layers.size()-1; i++){
      for(int j = 0; j < layers.get(i).size()-2; j++){
        Node v = layers.get(i).get(j);
        Node w = layers.get(i).get(j+1);
        if(swapImproves(i+1,v,w)){
          improved = true;
          Collections.swap(layers.get(i),j,j+1);
        }
      }
    }
  }
}
```

4. Position

```
private void assignPositions(){
  //After the layers have been ordered, medianPosition() is used to obtain the best Y coordinate for each node
  calculateInitialCoordinates();

  for(int i = 0; i < 2; i++){ //Paper "A Technique for Drawing Directed Graphs" indicates i < 8
    calculateCoordinates(i); // Look at previous/following layer for position reference
  }
}

private void calculateInitialCoordinates(){
  int yCoord = 0;
  // Get initial Y position for all nodes in every layer. Initial position is the same as their placement in the list
  for (List<Node> layer : layers) {
    for (Node n : layer) {
      nodeYPositions.put(n.getId(), yCoord);
      yCoord--;
    }
    yCoord = 0;
  }
}

private void calculateCoordinates(int dir){
  if(dir%2 == 0) {
    for (int j = 1; j < layers.size(); j++) {
      List<Integer> layerPositions = new ArrayList<>();
      for (Node n : layers.get(j)) {
        // Calculate median position for the node based on its adjacent nodes on the left.
        // Example: If the node is connected to 3 nodes in positions 4,8,10 (respectively), the added value will be 8.
        layerPositions.add(getMedianPos(n, j - 1, 0));
      }
      calculateLayerCoordinates(j,layerPositions);
```

```
32      }
33    }else{
34      for(int j = layers.size()-2; j >= 0; j--){
35        List<Integer> layerPositions = new ArrayList<>();
36        for (Node n : layers.get(j)) {
37          // Calculate median position for the node based on its adjacent
      nodes on the right.
38          layerPositions.add(getMedianPos(n, j + 1, 1));
39        }
40        calculateLayerCoordinates(j,layerPositions);
41      }
42    }
43  }
```

Appendix B

Entity Type Attributes

In this appendix we describe the details on the detection of possible Entity Type attributes, which are described in Section 4.6. First, we describe the Cypher query used to retrieve the attribute data from the Event nodes. Then, we describe the Java code that helps us identify the candidates. Finally, we describe the Cypher queries that allow us to set the Entity Type Attributes as properties of the Entity node and delete them from the Event nodes.

The query used to retrieve the attribute data from the database is shown below. First, we retrieve all the distinct properties assigned to the nodes of a log (lines 1-5). Then, we filter out the properties that refer to the ID, Timestamp, Activity, or Entity Type (lines 6-7). The first two properties are filtered out because those properties cannot be the same throughout all the Event nodes of an Entity Type path, and the last two properties are filtered out because they should not be removed from the Event nodes. Then, in line 8, for each property and each entity ID, we define three variables: (1) *numEvents*, which contains the number of events per entity ID, (2) *numExistingValues*, which contains the number of events in the path that contain that property, and (3) *distinctValues*, which contains the number of distinct values for that property throughout the entity type path. Finally, in line 9, we collect these variables per property, so they can be analyzed in the Java code.

```
1 MATCH (l:Log)--(e:Event)
2 WHERE l.ID = "filename.csv"
3 WITH DISTINCT keys(e) AS keys
4 UNWIND keys AS Properties
5 WITH DISTINCT(Properties) as p
6 MATCH (l:Log)--(e:Event)--(en:Entity{EntityType:"EntityType"})
7 WHERE l.ID = "filename.csv" AND NOT p IN ['ID', 'Timestamp', '
    Activity', 'EntityType']
8 WITH p AS property, en.ID AS entityID, COUNT(e) AS numEvents,
```

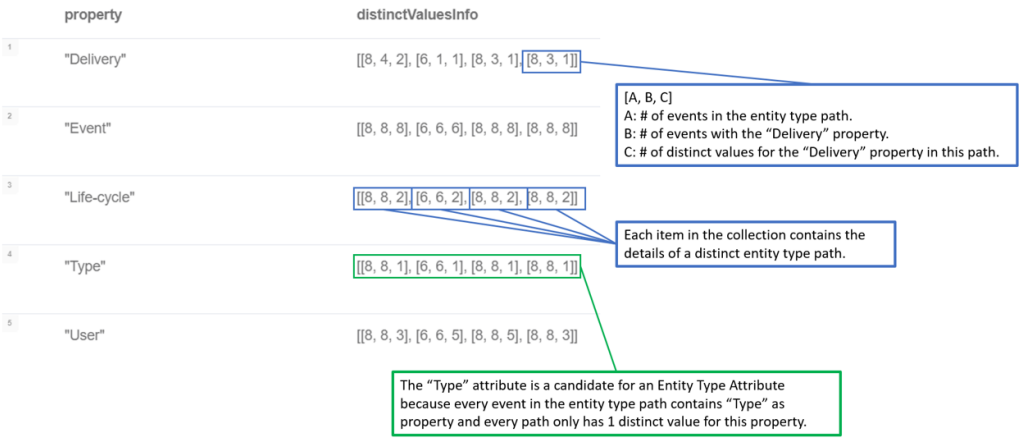


Figure B.1: Example result from executing the query that retrieves the data needed to find the Entity Type Attributes.

```
    COUNT(e[p]) AS numExistingValues, COUNT(DISTINCT(e[p]))
    AS distinctValues
RETURN property, COLLECT([numEvents,numExistingValues,
    distinctValues]) AS distinctValuesInfo
```

An example from the result of running this query can be observed in Figure B.1, which details the information returned by the query.

The Java code that analyzes the output from the previous query is shown below. Using the data retrieved from the query, we can check for two conditions to discard an attribute as an Entity Type Attribute (line 10): (1) Not every event in any path has the attribute defined as a property or (2) The number of distinct values for a given property in any path is different from 1.

```
1  List<String> entityTypeAttributes = new ArrayList<>();
2  boolean isEntityTypeCandidate = true;
3
4  //Variable 'queryAnswer' contains the result from the query
5  for(Record r : queryAnswer){
6    List<Object> attributeInfo = r.get("distinctValuesInfo").asList();
7    for(Object o : attributeInfo){
8      List<Long> entityInfo = new ArrayList<>((Collection<Long>) o);
9      // If the number of events on the :DF path does not match with the
       number of values or the number of distinct values is different from 1,
       it means there are events in the :DF that do not have that attribute or
       that have more than one distinct value, so it is discarded as a
       possible entity type attribute.
10     if(!entityInfo.get(0).equals(entityInfo.get(1)) || entityInfo.get(2) !=
       1){
11       isEntityTypeCandidate = false;
12       break;
13     }
14   }
15   if(isEntityTypeCandidate)
```

```
    entityTypeAttributes.add(r.get("property").asString());
  isEntityTypeCandidate = true;
}

return entityTypeAttributes;
```

The Entity Type attributes are displayed to the user as we saw in Figure 4.22. In that window, if the user clicks on "Continue", two more queries will be run.

Both queries are shown below. In the first query, we retrieve the attribute from the Event nodes (line 3) and set it as a property of the Entity node (line 4). Then, in the second query, we delete the property from the Event nodes (lines 7-9).

```
1 MATCH (l:Log)--(e:Event)--(en:Entity)
2 WHERE l.ID = "filename.csv" AND en.EntityType = "EntityType"
3 WITH DISTINCT en, e.`Attribute` AS property
4 SET en.`Attribute` = property
5
6
7 MATCH (l:Log)--(e:Event)--(en:Entity)
8 WHERE l.ID = "filename.csv" AND en.EntityType = "EntityType"
9 SET e.`Attribute` = null
```

Then, if the user decides to return the Entity Type attributes to the Event nodes through the "Return Entity Attribute" shown in Figure 4.13, the queries shown below are executed. The first query sets the attribute as a property of the Event nodes (lines 1-3). Then, the second query deletes the attribute from the Entity nodes (lines 6-8).

```
1 MATCH (l:Log)--(e:Event)--(en:Entity)
2 WHERE l.ID = "filename.csv" AND EXISTS(en.Attribute)
3 SET e.Attribute = en.Attribute
4
5
6 MATCH (l:Log)--(:Event)--(en:Entity)
7 WHERE l.ID = "filename.csv"
8 SET en.Attribute = null
```

Appendix C

Heuristic Miner Cypher Queries

In this appendix we show the full set of queries used in our implementation of the Heuristic Miner algorithm, described in Section 5.2.

Initial Setup

1. Create Algorithm and Model nodes.

```
MATCH (l:Log{ID:"filename.csv"})
MERGE (l)<-[:MAPS]-(:Algorithm{ID:"Heuristic Miner",Class:"
    ClassType",DF:"EntityType",Freq:FreqThreshold,Dep:DepThreshold,
    L1L:L1LThreshold,L2L:L2LThreshold,Rel:RelThreshold,Bind:
    BindThreshold})-[:PRODUCES]->(:Model{Algorithm:"Heuristic
    Miner",Log:"filename.csv",ID:"ModelID"})
```

2. Create DG_nodes and connect them to the Class nodes.

```
MATCH (m:Model{ID:"ModelID"})--(:Algorithm)--(:Log)--(:Event)
    --(c:Class{Type:"ClassType"})
MERGE (m)-[:CONTAINS{DG:"Before"}]->(dg:DG_node:Model_node{ID
    :c.ID})
MERGE (dg)-[:REPRESENTS]->(c)
```

3. Create Artificial Start and End nodes.

```
MATCH (m:Model{ID:"ModelID"})
MERGE (m)-[:CONTAINS{DG:"Before"}]->(:DG_node:Model_node{ID:"
    ARTIFICIAL_START",isStart:True})
MERGE (m)-[:CONTAINS{DG:"Before"}]->(:DG_node:Model_node{ID:"
    ARTIFICIAL_END",isEnd:True})
```

4. Create :MODEL_EDGE relations.

```
MATCH (m:Model{ID:"ModelID"})-->(dg:DG_node)-->(c:Class)
MATCH (m)-->(dg2:DG_node)-->(c2:Class)
```

```
MATCH (c)-[:DF_C{EntityType:"EntityType"}]->(c2)
MATCH (c)<--(:Event)-[df:DF{EntityType:"EntityType"}]->(:Event)
   -->(c2)
WITH DISTINCT dg AS From, dg2 AS To, COUNT(df) AS f
MERGE (From)-[:MODEL_EDGE{Freq:f}]->(To)
```

5. Create :MODEL_EDGE relations for the Artificial Start Node.

```
MATCH (m:Model{ID:"ModelID"})--(dg:DG_node)--(:Class)--(e:
   Event)
WHERE NOT EXISTS (()-[:DF{EntityType:"EntityType"}]->(e))
WITH m, dg AS startActivity, COUNT(e) AS f
MATCH (m)--(st:DG_node{isStart:True})
MERGE (st)-[:MODEL_EDGE{Freq:f}]->(startActivity)
```

6. Create :MODEL_EDGE relations for the Artificial End Node.

```
MATCH (m:Model{ID:"ModelID"})--(dg:DG_node)--(:Class)--(e:
   Event)
WHERE NOT EXISTS ((e)-[:DF{EntityType:"EntityType"}]->())
WITH m, dg AS endActivity, COUNT(e) AS f
MATCH (m)--(end:DG_node{isEnd:True})
MERGE (endActivity)-[:MODEL_EDGE{Freq:f}]->(end)
```

7. Calculate Dependency measure.

```
MATCH (m:Model{ID:"ModelID"})--(a:DG_node)-[me:MODEL_EDGE]->(b
   :DG_node)
OPTIONAL MATCH (b)-[me2:MODEL_EDGE]->(a)
WITH DISTINCT a, b, me, me.Freq AS fA, COALESCE(me2.Freq,0)
   AS fB
WITH a,b,ABS(ROUND(((fA-fB)*1.0/(fA+fB+1)),3)) AS dep
MATCH (a)-[me:MODEL_EDGE]->(b)
SET me.Dep = dep
```

8. Calculate L1L Dependency measure.

```
MATCH (m:Model{ID:"ModelID"})--(a:DG_node)-[me:MODEL_EDGE]-(a)
SET me.L1L = ROUND((me.Freq*1.0)/(me.Freq+1),3)
```

9. Calculate L2L Dependency measure.

```
MATCH (m:Model{ID:"ModelID"})--(dg:DG_node)-[:MODEL_EDGE]->(
   dg2:DG_node)-[:MODEL_EDGE]->(dg)
WITH DISTINCT m, dg, dg2
MATCH (m)--(dg)--(c:Class)
MATCH (m)--(dg2)--(c2:Class)
MATCH (e1:Event)--(c)--(e3:Event)
MATCH (c2)--(e2:Event)
OPTIONAL MATCH (e1)-[df:DF{EntityType:"EntityType"}]->(e2)-[:DF
   {EntityType:"EntityType"}]->(e3)
WITH m,dg,dg2,c,c2,COUNT(df) AS l2lFreqA
MATCH (e4:Event)--(c2)--(e6:Event)
```

```
MATCH (c)--(e5:Event)
OPTIONAL MATCH (e4)-[df2:DF{EntityType:"EntityType"}]->(e5)-[:DF{EntityType:"EntityType"}]->(e6)
WITH m,dg,dg2,l2lFreqA,COUNT(df2) AS l2lFreqB
WITH m,dg,dg2,ROUND(((l2lFreqA+l2lFreqB)*1.0)/(l2lFreqA+l2lFreqB+1),3) AS l2lDep
MATCH (m)--(dg)-[me:MODEL_EDGE]->(dg2)
SET me.L2L = l2lDep
```

Flexible Heuristic Miner

10. The 12 steps of Flexible Heuristic Miner plus one additional step to remove edges below the *frequency threshold*.

```
// ------------------------------------
// Flexible Heuristic Miner Step 2
// ------------------------------------
MATCH (m:Model{ID:"ModelID"})--(a:DG_node)-[me:MODEL_EDGE]->(a)
WHERE me.L1L >= L1L_Threshold
WITH COLLECT(DISTINCT [a.ID,a.ID]) AS C1
WITH CASE WHEN C1[0][0] IS NULL THEN [] ELSE C1 END AS C1
// ------------------------------------
// Flexible Heuristic Miner Step 3
// ------------------------------------
OPTIONAL MATCH (m:Model{ID:"ModelID"})--(a:DG_node)-[me:MODEL_EDGE]->(b:DG_node)
WHERE me.L2L >= L2L_Threshold AND NOT [a.ID,a.ID] IN C1 AND NOT [b.ID,b.ID] IN C1
WITH C1,COLLECT(DISTINCT [a.ID,b.ID]) AS C2
WITH C1, CASE WHEN C2[0][0] IS NULL THEN [] ELSE C2 END AS C2
// ------------------------------------
// Flexible Heuristic Miner Step 4
// ------------------------------------
MATCH (m:Model{ID:"ModelID"})--(a:DG_node)-[me:MODEL_EDGE]->(b:DG_node)
WHERE a <> b AND b.isEnd IS NULL
WITH C1,C2, a.ID AS act, [a.ID,b.ID] AS pair, me.Dep AS depVal
WITH C1,C2, act,apoc.agg.maxItems(pair,depVal) AS strFollowers
UNWIND strFollowers.items AS strPairs
WITH C1,C2, COLLECT(strPairs) AS Cout
// ------------------------------------
// Flexible Heuristic Miner Step 5
// ------------------------------------
MATCH (m:Model{ID:"ModelID"})--(a:DG_node)-[me:MODEL_EDGE]->(b:DG_node)
WHERE a <> b AND a.isStart IS NULL
WITH C1,C2,Cout, b.ID AS act, [a.ID,b.ID] AS pair, me.Dep AS depVal
WITH C1,C2,Cout, act,apoc.agg.maxItems(pair,depVal) AS strCauses
UNWIND strCauses.items AS strPairs
WITH C1,C2,Cout, COLLECT(strPairs) AS Cin
// ------------------------------------
// Flexible Heuristic Miner Step 6
// ------------------------------------
OPTIONAL MATCH (m:Model{ID:"ModelID"})--(a:DG_node)-[me:MODEL_EDGE]->(x:DG_node),
(m)--(b:DG_node)-[me2:MODEL_EDGE]->(y:DG_node)
WHERE [a.ID,x.ID] IN Cout AND me.Dep < Dep_Threshold AND [b.ID,y.ID] IN Cout AND [a.ID,b.ID] IN C2 AND (me2.Dep - me.Dep)>Rel_To_Best_Threshold
WITH C1,C2,Cout,Cin,COLLECT(DISTINCT [a.ID,x.ID]) AS Cout1
```

```
WITH C1,C2,Cout,Cin,CASE WHEN Cout1[0][0] IS NULL THEN [] ELSE Cout1 END AS Cout1
// ------------------------------------------
// Flexible Heuristic Miner Step 7
// ------------------------------------------
WITH C1,C2,Cin,apoc.coll.subtract(Cout, Cout1) as Cout
// ------------------------------------------
// Flexible Heuristic Miner Step 8
// ------------------------------------------
OPTIONAL MATCH (m:Model{ID:"ModelID"})--(x:DG_node)-[me:MODEL_EDGE]->(a:DG_node),
(m)--(y:DG_node)-[me2:MODEL_EDGE]->(b:DG_node)
WHERE [x.ID,a.ID] IN Cin AND me.Dep < Dep_Threshold AND [y.ID,b.ID] IN Cin AND [a.ID,b.ID] IN C2 AND (me2.Dep - me.Dep)>Rel_To_Best_Threshold
WITH C1,C2,Cout,Cin,COLLECT(DISTINCT [x.ID,a.ID]) AS Cin1
WITH C1,C2,Cout,Cin,CASE WHEN Cin1[0][0] IS NULL THEN [] ELSE Cin1 END AS Cin1
// ------------------------------------------
// Flexible Heuristic Miner Step 9
// ------------------------------------------
WITH C1,C2,Cout,apoc.coll.subtract(Cin, Cin1) as Cin
// ------------------------------------------
// Flexible Heuristic Miner Step 10
// ------------------------------------------
MATCH (m:Model{ID:"ModelID"})--(a:DG_node)-[me:MODEL_EDGE]->(b:DG_node)
WHERE me.Dep >= Dep_Threshold
WITH C1,C2,Cin,Cout,COLLECT(DISTINCT [a.ID,b.ID]) AS Cout2_1
OPTIONAL MATCH (m:Model{ID:"ModelID"})--(a:DG_node)-[me:MODEL_EDGE]->(b:DG_node),(a)-[me2:MODEL_EDGE]->(c:DG_node)
WHERE [a.ID,c.ID] IN Cout AND ABS(me2.Dep - me.Dep)<Rel_To_Best_Threshold
WITH C1,C2,Cin,Cout2_1,COLLECT(DISTINCT [a.ID,b.ID]) AS Cout2_2
WITH C1,C2,Cin,Cout2_1,CASE WHEN Cout2_2[0][0] IS NULL THEN [] ELSE Cout2_2 END AS Cout2_2
WITH C1,C2,Cin,Cout2_1+Cout2_2 AS Cout2
// ------------------------------------------
// Flexible Heuristic Miner Step 11
// ------------------------------------------
MATCH (m:Model{ID:"ModelID"})--(b:DG_node)-[me:MODEL_EDGE]->(a:DG_node)
WHERE me.Dep >= Dep_Threshold
WITH C1,C2,Cout2,Cin,COLLECT(DISTINCT [b.ID,a.ID]) AS Cin2_1
OPTIONAL MATCH (m:Model{ID:"ModelID"})--(b:DG_node)-[me:MODEL_EDGE]->(a:DG_node),(c:DG_node)-[me2:MODEL_EDGE]->(a)
WHERE [c.ID,a.ID] IN Cin AND ABS(me2.Dep - me.Dep)<Rel_To_Best_Threshold
WITH C1,C2,Cout2,Cin2_1,COLLECT(DISTINCT [b.ID,a.ID]) AS Cin2_2
WITH C1,C2,Cout2,Cin2_1,CASE WHEN Cin2_2[0][0] IS NULL THEN [] ELSE Cin2_2 END AS Cin2_2
WITH C1,C2,Cout2,Cin2_1+Cin2_2 AS Cin2
// ------------------------------------------
// Flexible Heuristic Miner Step 12
// ------------------------------------------
WITH C1+C2+Cout2+Cin2 AS dgEdges
// ------------------------------------------
// Step 13. Remove edges below Frequency threshold
// ------------------------------------------
UNWIND dgEdges AS dgEdge
WITH COLLECT(DISTINCT dgEdge) AS dgEdges
MATCH (m:Model{ID:"ModelID"})--(:DG_node)--(:Class)--(:Event)--(n:Entity)
WHERE n.EntityType = "EntityType"
WITH dgEdges,COUNT(DISTINCT n) AS numEntities
OPTIONAL MATCH (m:Model{ID:"ModelID"})--(a:DG_node)-[me:MODEL_EDGE]->(b:DG_node)
WHERE [a.ID,b.ID] IN dgEdges AND (me.Freq*1.0/numEntities) < Freq_Threshold
```

```
WITH dgEdges,COLLECT(DISTINCT [a.ID,b.ID]) AS Cfreq
WITH apoc.coll.subtract(dgEdges, Cfreq) as dgEdges
// ------------------------------------
// Create new DG
// ------------------------------------
UNWIND dgEdges AS dgEdge
WITH dgEdge
MATCH (m:Model{ID:"ModelID"})
MERGE (m)-[:CONTAINS{DG:"After"}]->(a:DG_node:Model_node{ID:dgEdge[0]})
MERGE (m)-[:CONTAINS{DG:"After"}]->(b:DG_node:Model_node{ID:dgEdge[1]})
MERGE (a)-[:MODEL_EDGE]->(b)
```

11. Set the specific properties for the DG_nodes that represent the Artificial Start and End nodes and were generated after the execution of the FHM.

```
//Create Start Properties
OPTIONAL MATCH (m:Model{ID:"ModelID"})-[:CONTAINS{DG:"After"}]-(a)
WHERE a.ID = "ARTIFICIAL_START"
CALL apoc.do.when(
  a IS NULL,
    "MATCH (m:Model{ID:'ModelID'}) MERGE (m)-[:CONTAINS{DG:'After'}]->(:
    DG_node:Model_node{ID:'ARTIFICIAL_START',isStart:true})",
    "SET a.isStart = true",
    {a:a}
)YIELD value RETURN 1

//Create End Properties
OPTIONAL MATCH (m:Model{ID:"ModelID"})-[:CONTAINS{DG:"After"}]-(a)
WHERE a.ID = "ARTIFICIAL_END"
CALL apoc.do.when(
  a IS NULL,
    "MATCH (m:Model{ID:'ModelID'}) MERGE (m)-[:CONTAINS{DG:'After'}]->(:
    DG_node:Model_node{ID:'ARTIFICIAL_END',isEnd:true})",
    "SET a.isEnd = true",
    {a:a}
)YIELD value RETURN 1
```

Add Missing Inputs and Outputs

12. Obtain number of nodes with missing outputs.

```
CALL{
    MATCH (m:Model{ID:"ModelID"})-[:CONTAINS{DG:'After'}]-(a:
    DG_node)-[me:MODEL_EDGE]->()
    WITH a,COUNT(me) AS numEdges
    MATCH (a:DG_node)-[me:MODEL_EDGE]->(b)
    WHERE a = b AND numEdges = 1 RETURN a
    UNION
    MATCH (m:Model{ID:"ModelID"})-[:CONTAINS{DG:'After'}]-(a:
    DG_node)
  WHERE NOT (a)-[:MODEL_EDGE]->() AND a.isEnd IS NULL RETURN
    a
} WITH a
RETURN COUNT(a) AS numNodes
```

13. Create missing outgoing edges.

```
CALL{
    MATCH (m:Model{ID:"ModelID"})-[:CONTAINS{DG:'After'}]-(a:DG_node)-[me:
    MODEL_EDGE]->()
    WITH a,COUNT(me) AS numEdges
    MATCH (a)-[me:MODEL_EDGE]->(b:DG_node)
    WHERE a = b AND numEdges = 1 RETURN a
    UNION
    MATCH (m:Model{ID:"ModelID"})-[:CONTAINS{DG:'After'}]-(a:DG_node)
  WHERE NOT (a)-[:MODEL_EDGE]->() AND a.isEnd IS NULL RETURN a
} WITH a
MATCH (m:Model{ID:"ModelID"})-[:CONTAINS{DG:'Before'}]-(b:DG_node)-[me:
    MODEL_EDGE]->(c:DG_node)
WHERE b.ID = a.ID AND b <> c
WITH m,a,c.ID AS cID,me.Dep AS depVal
WITH m,a,apoc.agg.maxItems(cID,depVal) AS strFollowers
WITH m,a,strFollowers.items AS items
UNWIND items AS out
WITH m,a,out
OPTIONAL MATCH (m)-[:CONTAINS{DG:'After'}]-(b:DG_node{ID:out})
CALL apoc.do.when(
  b IS NULL,
    "MERGE (a)-[:MODEL_EDGE]->(c:DG_node:Model_node{ID:out})
    MERGE (m)-[:CONTAINS{DG:'After'}]->(c)",
    "MERGE (a)-[:MODEL_EDGE]->(b)",
    {m:m,a:a,b:b,out:out}
) YIELD value RETURN 1
```

14. Obtain number of nodes with missing inputs.

```
CALL{
    MATCH (m:Model{ID:"ModelID"})-[:CONTAINS{DG:'After'}]-(a:
    DG_node)-[me:MODEL_EDGE]->()
    WITH a,COUNT(me) AS numEdges
    MATCH (a:DG_node)-[me:MODEL_EDGE]->(b)
    WHERE a = b AND numEdges = 1 RETURN a
    UNION
    MATCH (m:Model{ID:"ModelID"})-[:CONTAINS{DG:'After'}]-(a:
    DG_node)
  WHERE NOT (a)-[:MODEL_EDGE]->() AND a.isEnd IS NULL RETURN
    a
} WITH a
RETURN COUNT(a) AS numNodes
```

15. Create missing ingoing edges.

```
CALL{
  MATCH (m:Model{ID:"ModelID"})-[:CONTAINS{DG:'After'}]-(:DG_node)-[me:
    MODEL_EDGE]->(a)
  WITH a,COUNT(me) AS numEdges
  MATCH (b:DG_node)-[me:MODEL_EDGE]->(a)
  WHERE a = b AND numEdges = 1 RETURN a
  UNION
  MATCH (m:Model{ID:"ModelID"})-[:CONTAINS{DG:'After'}]-(a:DG_node)
  WHERE NOT ()-[:MODEL_EDGE]->(a) AND a.isStart IS NULL RETURN a
} WITH a
MATCH (m:Model{ID:"ModelID"})-[:CONTAINS{DG:'Before'}]-(b:DG_node)-[me:
    MODEL_EDGE]->(c:DG_node)
```

```
WHERE c.ID = a.ID
WITH m,a,b.ID AS bID,me.Dep AS depVal
WITH m,a,apoc.agg.maxItems(bID,depVal) AS strCauses
WITH m,a,strCauses.items AS items
UNWIND items AS in
WITH m,a,in
OPTIONAL MATCH (m)-[:CONTAINS{DG:'After'}]->(b:DG_node{ID:in})
CALL apoc.do.when(
  b IS NULL,
  "MERGE (c:DG_node:Model_node{ID:in})-[:MODEL_EDGE]->(a)
  MERGE (m)-[:CONTAINS{DG:'After'}]->(c)",
  "MERGE (b)-[:MODEL_EDGE]->(a)",
  {m:m,a:a,b:b,in:in}
) YIELD value RETURN 1
```

Connect Model nodes to Class nodes

16. Create :REPRESENTS relations.

```
MATCH (m:Model{ID:"ModelID"})--(:Algorithm)--(:Log)--(:Event)
    --(c:Class{Type:"ClassType"})
MATCH (m)-[:CONTAINS{DG:"After"}]->(dg:DG_node{ID:c.ID})
MERGE (dg)-[:REPRESENTS]->(c)
```

Cleanup

17. Copy :MODEL_EDGE properties from initial dependency graph to the dependency graph created after the FHM was executed.

```
MATCH (m:Model{ID:"ModelID"})-[:CONTAINS{DG:"Before"}]->(b1)-[
    meB:MODEL_EDGE]->(b2)
MATCH (m)-[:CONTAINS{DG:"After"}]->(a1)-[meA:MODEL_EDGE]->(a2
    )
WHERE b1.ID = a1.ID and b2.ID = a2.ID
SET meA = meB
```

18. Delete initial DG.

```
MATCH (m:Model{ID:"ModelID"})-[:CONTAINS{DG:"Before"}]->(n)
DETACH DELETE n
```

Bindings

19. Define output bindings.

```
CALL{
  MATCH (m:Model{ID:"ModelID"})-[:CONTAINS]->(a:DG_node)-[:MODEL_EDGE]->(b:
    DG_node)
  MATCH (a)--(cl:Class)--(i:Event)
  MATCH (b)--(:Class)--(j:Event)
  MATCH (i)-[:DF*{EntityType:"EntityType"}]->(j)
  WITH DISTINCT m,cl.Type AS classType,i,COLLECT(j) AS eventsCaused
  MATCH (m)-[:CONTAINS]->(c:DG_node)-[:MODEL_EDGE]->(b:DG_node)
```

```
8   OPTIONAL MATCH (i)-[:DF*{EntityType:"EntityType"}]->(k:Event)-[:DF*{
      EntityType:"EntityType"}]->(j:Event)
9   WHERE EXISTS((c)--(:Class)--(k)) AND EXISTS((b)--(:Class)--(j)) AND j IN
      eventsCaused
10  WITH i,classType,eventsCaused,COLLECT(DISTINCT j) AS eventsOtherCause
11  WITH i,classType,apoc.coll.subtract(eventsCaused, eventsOtherCause) AS
      eventList
12  UNWIND eventList AS event
13  WITH DISTINCT i,classType,apoc.coll.sort(COLLECT(DISTINCT event[classType
      ])) AS oB,1 AS n
14  RETURN DISTINCT i[classType] AS mActivity, oB, SUM(n) AS bindFreq ORDER BY
      bindFreq DESC
15  UNION
16  MATCH (m:Model{ID:"ModelID"})-[:CONTAINS]-(a:DG_node)-[:MODEL_EDGE]->(b:
      DG_node)
17  MATCH (a)--(cl:Class)--(i:Event)
18  WHERE b.isEnd AND NOT EXISTS((i)-[:DF{EntityType:"EntityType"}]->())
19  WITH DISTINCT i,cl.Type AS classType,[b.ID] AS oB, 1 AS n
20  RETURN DISTINCT i[classType] AS mActivity, oB, SUM(n) AS bindFreq ORDER BY
      bindFreq DESC
21  UNION
22  MATCH (m:Model{ID:"ModelID"})-[:CONTAINS]-(a:DG_node)-[:MODEL_EDGE]->(b:
      DG_node)
23  MATCH (b)--(cl:Class)--(i:Event)--(en:Entity{EntityType:"EntityType"})
24  WHERE a.isStart
25  WITH DISTINCT m,cl.Type AS classType,a AS start,en.uID AS enUID,COLLECT(i)
      AS eventsCaused
26  MATCH (m)-[:CONTAINS]-(c:DG_node)-[:MODEL_EDGE]->(b:DG_node)
27  OPTIONAL MATCH (k:Event)-[:DF*{EntityType:"EntityType"}]->(j:Event)
28  WHERE EXISTS((c)--(:Class)--(k)) AND EXISTS((b)--(:Class)--(j)) AND j IN
      eventsCaused
29  WITH start,enUID,classType,eventsCaused,COLLECT(DISTINCT j) AS
      eventsOtherCause
30  WITH start,enUID,classType,apoc.coll.subtract(eventsCaused,
      eventsOtherCause) AS eventList
31  UNWIND eventList AS event
32  WITH DISTINCT start,enUID,apoc.coll.sort(COLLECT(DISTINCT event[classType
      ])) AS oB,1 AS n
33  RETURN DISTINCT start.ID as mActivity, oB, SUM(n) AS bindFreq ORDER BY
      bindFreq DESC
34 }
35 WITH mActivity,apoc.agg.maxItems(mActivity, bindFreq).value AS freqMax,
     COLLECT([oB,bindFreq]) AS bindingsDetails
36 MATCH (m:Model{ID:"ModelID"})-[:CONTAINS]-(a:DG_node)-[:MODEL_EDGE]->(b:
     DG_node)
37 WHERE a.ID = mActivity
38 WITH DISTINCT mActivity, bindingsDetails, freqMax, COLLECT(b.ID) AS
     allOutputs
39 CALL apoc.when(
40   SIZE(bindingsDetails) = 1,
41     "UNWIND bindingsDetails AS bindDetails
42     RETURN mActivity AS mAct,COLLECT(apoc.text.join(bindDetails[0],'|')) AS
     outputBindings",
43     "CALL{
44       WITH mActivity, bindingsDetails, allOutputs
45         UNWIND allOutputs AS o
46         WITH mActivity, bindingsDetails, o
47         UNWIND bindingsDetails AS bindDetails
48         WITH mActivity, o, bindDetails
49         WHERE o IN bindDetails[0]
50         WITH mActivity,o,apoc.agg.maxItems(bindDetails[0],bindDetails[1]).
     items AS bindActsMaxFreq
```

```
51          RETURN mActivity AS mAct,COLLECT(DISTINCT apoc.text.join(
        bindActsMaxFreq[0],'|')) AS oBindings
52        UNION
53          WITH mActivity, bindingsDetails, freqMax
54            UNWIND bindingsDetails AS bindDetails
55            WITH mActivity,freqMax,bindDetails
56            WHERE (bindDetails[1]*1.0)/freqMax >= BindingsThreshold
57            RETURN mActivity AS mAct,COLLECT(DISTINCT apoc.text.join(bindDetails
        [0],'|')) AS oBindings
58        }
59        WITH mAct,oBindings
60        UNWIND oBindings AS oBind
61        RETURN mAct, COLLECT(DISTINCT oBind) AS outputBindings",
62        {mActivity:mActivity,bindingsDetails:bindingsDetails,freqMax:freqMax,
        allOutputs:allOutputs}
63 )YIELD value
64 WITH value.mAct AS mActivity, value.outputBindings AS outputBindings
65 MATCH (m:Model{ID:"ModelID"})-[:CONTAINS]-(dg:DG_node{ID:mActivity})
66 SET dg.OutputBindings = outputBindings
```

20. Define input bindings.

```
1  CALL{
2    MATCH (m:Model{ID:"ModelID"})-[:CONTAINS]-(a:DG_node)-[:MODEL_EDGE]->(b:
       DG_node)
3    MATCH (a)--(cl:Class)--(i:Event)
4    MATCH (b)--(:Class)--(j:Event)
5    MATCH (i)-[:DF*{EntityType:"EntityType"}]->(j)
6    WITH DISTINCT m,cl.Type AS classType,j,COLLECT(i) AS causeEvents
7    MATCH (m)-[:CONTAINS]-(c:DG_node)-[:MODEL_EDGE]->(b:DG_node)
8    OPTIONAL MATCH (i:Event)-[:DF*{EntityType:"EntityType"}]->(k:Event)-[:DF*{
       EntityType:"EntityType"}]->(j)
9    WHERE EXISTS((b)--(:Class)--(k)) AND EXISTS((c)--(:Class)--(i)) AND i IN
       causeEvents
10   WITH j,classType,causeEvents,COLLECT(DISTINCT i) AS eventsOtherCause
11   WITH j,classType,apoc.coll.subtract(causeEvents, eventsOtherCause) AS
       eventList
12   UNWIND eventList AS event
13   WITH DISTINCT j,classType,apoc.coll.sort(COLLECT(DISTINCT event[classType
       ])) AS iB,1 AS n
14   RETURN DISTINCT j[classType] AS mActivity, iB, SUM(n) AS bindFreq ORDER BY
        bindFreq DESC
15   UNION
16   MATCH (m:Model{ID:"ModelID"})-[:CONTAINS]-(a:DG_node)-[:MODEL_EDGE]->(b:
       DG_node)
17   MATCH (b)--(cl:Class)--(i:Event)
18   WHERE a.isStart AND NOT EXISTS(()-[:DF{EntityType:"EntityType"}]->(i))
19   WITH DISTINCT i,cl.Type AS classType,[a.ID] AS iB, 1 AS n
20   RETURN DISTINCT i[classType] AS mActivity, iB, SUM(n) AS bindFreq ORDER BY
        bindFreq DESC
21   UNION
22   MATCH (m:Model{ID:"ModelID"})-[:CONTAINS]-(a:DG_node)-[:MODEL_EDGE]->(b:
       DG_node)
23   MATCH (a)--(cl:Class)--(i:Event)--(en:Entity{EntityType:"EntityType"})
24   WHERE b.isEnd
25   WITH DISTINCT m,cl.Type AS classType,b AS end,en.uID AS enUID,COLLECT(i)
       AS causeEvents
26   MATCH (m)-[:CONTAINS]-(c:DG_node)-[:MODEL_EDGE]->(b:DG_node)
27   OPTIONAL MATCH (i:Event)-[:DF*{EntityType:"EntityType"}]->(k:Event)
28   WHERE EXISTS((c)--(:Class)--(i)) AND EXISTS((b)--(:Class)--(k)) AND i IN
       causeEvents
29   WITH end,enUID,classType,causeEvents,COLLECT(DISTINCT i) AS
```

```
    eventsOtherCause
  WITH end,enUID,classType,apoc.coll.subtract(causeEvents, eventsOtherCause)
     AS eventList
  UNWIND eventList AS event
  WITH DISTINCT end,enUID,apoc.coll.sort(COLLECT(DISTINCT event[classType]))
     AS iB,1 AS n
  RETURN DISTINCT end.ID as mActivity, iB, SUM(n) AS bindFreq ORDER BY
    bindFreq DESC
}
WITH mActivity,apoc.agg.maxItems(mActivity, bindFreq).value AS freqMax,
    COLLECT([iB,bindFreq]) AS bindingsDetails
MATCH (m:Model{ID:"ModelID"})-[:CONTAINS]-(a:DG_node)-[:MODEL_EDGE]->(b:
    DG_node)
WHERE b.ID = mActivity
WITH DISTINCT mActivity, bindingsDetails, freqMax, COLLECT(a.ID) AS
    allInputs
CALL apoc.when(
  SIZE(bindingsDetails) = 1,
    "UNWIND bindingsDetails AS bindDetails
    RETURN mActivity AS mAct,COLLECT(apoc.text.join(bindDetails[0],'|')) AS
    inputBindings",
    "CALL{
      WITH mActivity, bindingsDetails, allInputs
        UNWIND allInputs AS in
        WITH mActivity, bindingsDetails, in
        UNWIND bindingsDetails AS bindDetails
        WITH mActivity, in, bindDetails
        WHERE in IN bindDetails[0]
        WITH mActivity,in,apoc.agg.maxItems(bindDetails[0],bindDetails[1]).
    items AS bindActsMaxFreq
        RETURN mActivity AS mAct,COLLECT(DISTINCT apoc.text.join(
    bindActsMaxFreq[0],'|')) AS iBindings
    UNION
      WITH mActivity, bindingsDetails, freqMax
        UNWIND bindingsDetails AS bindDetails
        WITH mActivity,freqMax,bindDetails
        WHERE (bindDetails[1]*1.0)/freqMax >= BindingsThreshold
        RETURN mActivity AS mAct,COLLECT(DISTINCT apoc.text.join(bindDetails
    [0],'|')) AS iBindings
    }
    WITH mAct,iBindings
    UNWIND iBindings AS iBind
    RETURN mAct, COLLECT(DISTINCT iBind) AS inputBindings",
    {mActivity:mActivity,bindingsDetails:bindingsDetails,freqMax:freqMax,
    allInputs:allInputs}
)YIELD value
WITH value.mAct AS mActivity, value.inputBindings AS inputBindings
MATCH (m:Model{ID:"ModelID"})-[:CONTAINS]-(dg:DG_node{ID:mActivity})
SET dg.InputBindings = inputBindings
```

Appendix D

Petri Net Cypher Queries

In this appendix we show the full set of queries used in our implementation to translate the Dependency Graph into a Petri Net, as described in Section 6.2.1.

Create Petri Net elements based on Output Bindings

```
MATCH (m:Model{ID:"ModelID"})--(dg:DG_node)
WITH dg.ID AS mAct, dg.OutputBindings AS maOB
UNWIND maOB AS mActs
WITH mAct,mActs AS Output
WITH mAct, COLLECT(SPLIT(Output,"|")) AS Output
CALL apoc.do.when(
  SIZE(Output) > 1,
    "MERGE (p:PetriNet:Model_node{type:'t',t:mAct,model:'ModelID'})-[:
    MODEL_EDGE]->(o:PetriNet:Model_node{type:'p',model:'ModelID'})
  WITH Output,mAct,o
  UNWIND Output as a
  CREATE (o)-[:MODEL_EDGE]->(t:PetriNet:Model_node{type:'tau',model:'
    ModelID'})
  WITH t,a,mAct
  UNWIND a AS ma
  OPTIONAL MATCH (n:PetriNet{model:'ModelID'}) WHERE n.in = mAct AND n.out =
    ma
  CALL apoc.do.when(
    n IS NULL,
    \"CREATE (t)-[:MODEL_EDGE]->(:PetriNet:Model_node{type:\\'p\\',in:mAct,
    out:ma,model:\\'ModelID\\'})\",
    \"CREATE (t)-[:MODEL_EDGE]->(n)\",
    {n:n,t:t,mAct:mAct,ma:ma})
  YIELD value RETURN 1",
  "MERGE (p:PetriNet:Model_node{type:'t',t:mAct,model:'ModelID'})
    WITH p,Output,mAct
    UNWIND Output as a
    WITH p,a,mAct
    UNWIND a AS ma
    OPTIONAL MATCH (n:PetriNet{model:'ModelID'}) WHERE n.in = mAct AND n.out
    = ma
    CALL apoc.do.when(
        n IS NULL,
```

```
        \"CREATE (p)-[:MODEL_EDGE]->(:PetriNet:Model_node{type:\\'p\\',in:
    mAct,out:ma,model:\\'ModelID\\'})\",
        \"CREATE (p)-[:MODEL_EDGE]->(n)\",
        {n:n,p:p,mAct:mAct,ma:ma})
    YIELD value RETURN 1",
    {mAct:mAct,Output:Output})
YIELD value RETURN 1
```

Create Petri Net elements based on Input Bindings

```
CREATE (:PetriNet:Model_node{model:"ModelID",t:"ARTIFICIAL_END",type:"t"})
WITH 1 AS ignore
MATCH (m:Model{ID:"ModelID"})--(dg:DG_node)
WITH dg.ID AS mAct, dg.InputBindings AS maIB
UNWIND maIB AS mActs
WITH mAct,mActs AS Input
WITH mAct, COLLECT(SPLIT(Input,"|")) AS Input
CALL apoc.do.when(
  SIZE(Input) > 1,
    "MATCH (n:PetriNet{type:'t',t:mAct,model:'ModelID'})
    MERGE (o:PetriNet:Model_node{type:'p',model:'ModelID'})-[:MODEL_EDGE]->(n
    )
  WITH Input,mAct,o
  UNWIND Input as a
  CREATE (t:PetriNet:Model_node{type:'tau',model:'ModelID'})-[:MODEL_EDGE]->(
    o)
    WITH t,a,mAct
    UNWIND a AS ma
  OPTIONAL MATCH (n:PetriNet{model:'ModelID'}) WHERE n.in = ma AND n.out =
    mAct
    CALL apoc.do.when(
        n IS NULL,
        \"CREATE (:PetriNet:Model_node{type:\\'p\\',in:ma,out:mAct,model:\\'
    ModelID\\'})-[:MODEL_EDGE]->(t)\",
        \"CREATE (n)-[:MODEL_EDGE]->(t)\",
        {n:n,t:t,mAct:mAct,ma:ma})
    YIELD value RETURN 1",
    "MATCH (p:PetriNet{type:'t',t:mAct,model:'ModelID'})
    WITH p,Input,mAct
    UNWIND Input as a
    WITH p,a,mAct
    UNWIND a AS ma
    OPTIONAL MATCH (n:PetriNet{model:'ModelID'}) WHERE n.in = ma AND n.out =
    mAct
    CALL apoc.do.when(
        n IS NULL,
        \"CREATE (:PetriNet:Model_node{type:\\'p\\',in:ma,out:mAct,model:\\'
    ModelID\\'})-[:MODEL_EDGE]->(p)\",
        \"CREATE (n)-[:MODEL_EDGE]->(p)\",
        {n:n,p:p,mAct:mAct,ma:ma})
    YIELD value RETURN 1",
    {mAct:mAct,Input:Input})
YIELD value RETURN 1
```

Identify Start and End Places

Since the Start and End places of the Petri net are initially treated as additional transitions, we need to change their properties to specify them as places.

```
MATCH (m:Model{ID:"ModelID"}),(s:PetriNet{t:"ARTIFICIAL_START
    "}),(e:PetriNet{t:"ARTIFICIAL_END"})
CREATE (m)-[:CONTAINS_PN]->(s)
SET s.type = "s_e",s.isStart=true,s.t=null
SET e.type = "s_e",e.isEnd=true,e.t=null
```

Once they have been identified, we need to remove the extra place between the Start place and the first transition.

```
MATCH (m:Model{ID:"ModelID"})-[:CONTAINS_PN]->(s:PetriNet)-->(
    p:PetriNet)-->(n)
MERGE (s)-[:MODEL_EDGE]->(n)
DETACH DELETE p
```

Then, we remove the extra place between the last transition and the End place.

```
MATCH (n)-->(p:PetriNet)-->(e:PetriNet{model:"ModelID",isEnd:
    true})
MERGE (n)-[:MODEL_EDGE]->(e)
DETACH DELETE p
```

Petri Net Simplification

First, we find cases where the Petri Net has the following pattern: *(Place1 -> Tau -> Place2)*. If the Tau transition only has one input and one output and Place2 only has 1 input, the Petri Net can be simplified by connecting the outputs of Place2 directly to Place1 and removing Tau and Place2.

```
MATCH (m:Model{ID:"ModelID"})-[:CONTAINS_PN]->(s:PetriNet)
    -[*]->(t{type:'tau'})
WITH DISTINCT t
MATCH ()-[a:MODEL_EDGE]->(t)
WITH t,COUNT(DISTINCT a) AS inTau
MATCH (t)-[b:MODEL_EDGE]->()
WITH t,inTau,COUNT(DISTINCT b) AS outTau
WHERE inTau = 1 AND outTau = 1
MATCH (t)-->(p2)
MATCH ()-[c:MODEL_EDGE]->(p2)
WITH p2,t,COUNT(c) AS inP2
WHERE inP2 = 1
MATCH (p1)-->(t)-->(p2)
MATCH (p2)-->(outP2)
MERGE (p1)-[:MODEL_EDGE]->(outP2)
DETACH DELETE t,p2
```

Then, we again find cases where the Petri Net has the following pattern: *(Place1 -> Tau -> Place2)*. This time, if the Tau transition only has one input and one output and Place1 only has 1 output, the Petri Net can be

simplified by connecting the inputs of Place1 directly to Place2 and removing Place1 and Tau.

```
MATCH (m:Model{ID:"ModelID"})-[:CONTAINS_PN]->(s:PetriNet)
   -[*]->(t{type:'tau'})
WITH DISTINCT t
MATCH ()-[a:MODEL_EDGE]->(t)
WITH t,COUNT(DISTINCT a) AS inTau
MATCH (t)-[b:MODEL_EDGE]->()
WITH t,inTau,COUNT(DISTINCT b) AS outTau
WHERE inTau = 1 AND outTau = 1
MATCH (p1)-->(t)
MATCH (p1)-[c:MODEL_EDGE]->()
WITH p1,t,COUNT(c) AS outP1
WHERE outP1 = 1
MATCH (p1)-->(t)-->(p2)
MATCH (inP1)-->(p1)
MERGE (inP1)-[:MODEL_EDGE]->(p2)
DETACH DELETE p1,t
```

Appendix E

Evaluation Details

E.1 Execution of Experiment 1

In this appendix we describe the details on the execution of the first experiment for the evaluation of the tool, described in Section 7.1. First, we describe the steps taken in ProM. Then, we describe the steps taken in our tool. A thing to note is that the "case" column in the input CSV files refers to the Applications and the "OfferID" column refers to the Offers.

E.1.1 ProM

1. *Import event data.* To import the event data into the ProM workspace, we click on the "Import" button, select the first CSV file, named "BPIC17_Sample _20cases.csv", and click on "Open". In the popup window, we leave the default import plugin and click "Ok".

2. *Create the "Offer" XES file.* To do this, we first click on the "Use resource" (Play) button. Then, on the "Actions" view, we select the "Convert CSV to XES" action and click on "Start". On the "Configure CSV Parser Settings" window, we click on "Next". We want to obtain a model for the Offer-related activities, and we know from domain knowledge that the best way to describe the process for the BPIC17 is through the combined attributes of "Activity" and "lifecycle", so, on the "Configure Conversion from CSV to XES" window, we need to make two changes. First, we select "OfferID" as the case column. and then we select "Activity" and "lifecycle" as the event columns. Once the case and event columns are set, we click on "Next" and in the next window we click on "Finish".

3. *Save the "Offer" XES file.* We go back to the "Workspace" view and select the XES event log. Then, we click on "Export to disk". Finally, we assign a name to the log file and click on "Save".

4. *Create the "Offer" model.* In the "Workspace" view, we select the recently generated XES event log, and click on the "Use resource" button. In the "Actions" view, we search for "Heuristic Miner" to filter the actions. Then, we select the "Interactive Data-aware Heuristic Miner (iDHM)" action and click on "Start". Then, on the popup window, we click "Continue".

5. *Save the "Offer" model.* In the Heuristic Miner view, we click on "Export model" and click "Ok" on the popup window. Then, we go back to the "Workspace" view, select the Causal net that was just generated and click on "Export to disk". We assign a name to the file and click on "Save".

6. *Create the "Application" XES file.* To do this, we select the original CSV file imported into ProM and click on the "Use resource" (Play) button. Then, on the "Actions" view, we select the "Convert CSV to XES" action and click on "Start". On the "Configure CSV Parser Settings" window, we click on "Next". This time, we leave the default value for the case columns and just select "Activity" and "lifecycle" as the event columns. Once the case and event columns are set, we click on "Next" and in the next window we click on "Finish".

7. *Save the "Application" XES file.* We go back to the "Workspace" view and select the XES event log. Then, we click on "Export to disk". Finally, we assign a name to the log file and click on "Save".

8. *Create the "Application" model.* In the "Workspace" view, we select the recently generated XES event log, and click on the "Use resource" button. In the "Actions" view, we search for "Heuristic Miner" to filter the actions. Then, we select the "Interactive Data-aware Heuristic Miner (iDHM)" action and click on "Start". Then, on the popup window, we click "Continue". After the discovered process model is shown on screen, we click on the "Expert Options" button to display additional thresholds. For the "L2 Loop" threshold, we set the value at 0.7.

9. *Save the "Application" model.* In the Heuristic Miner view, we click on "Export model" and click "Ok" on the popup window. Then, we go back to the "Workspace" view, select the Causal net that was just

generated and click on "Export to disk". We assign a name to the file and click on "Save".

E.1.2 Graph Tool

1. *Import event data.* To do this, we use the "Upload CSV File" option from the tool to select the CSV file, named "BPIC17_Sample _20cases.csv". On the "Select Activity Attribute", we select "Activity" and click on "Continue". Then, on the "Select Event Attributes" window, we select all the remaining attributes and click on "Next". Then, in the "Select Log Dimensions" window, we first add "OfferID" and then we add "case" as dimensions. Then, we click on "Finish".

2. *Define the "Offer" Entity Type.* In this model we want to visualize how the Offer-related activities interact between them. To do this, we expand the "Logs" panel and select the recently uploaded log, whose name is the same as the one for the CSV file, "BPIC17_Sample _20cases.csv". After the log name appears in the Log Label, we expand the "Entities" panel and click on "New Entity". On the "Entity: Attribute Selection" window, we select "OfferID" and click on "Create Entity".

3. *Using "Offer" Entity Type, define the "Activity+Lifecycle" Class.* To do this, we expand the "Classes" panel and click on "New Class". We know from domain knowledge that the best way to describe the process for the BPIC17 is through the combined attributes of "Activity" and "lifecycle", so, in the "Class: Attribute Selection" window, we select both the "Activity" and "lifecycle" attributes, and then we click on "Continue". Then, on the "Entity: Attribute Selection" window, we select the Entity Type previously defined, the "OfferID", and click on "Create Class".

4. *Create the "Offer" model.* We expand the "Algorithms" panel, select "Heuristic Miner" from the dropdown menu and click on "Generate Model". To generate the process model, we will use the default attributes, so, on the "Heuristic Miner: Parameter Selection" window, we can click directly on "Generate Model".

5. *Visualize the "Offer" model.* We expand the "Models" panel, select the "HM_1" model from the list and click on "Show Model". Now we can see the discovered process model containing exclusively the activities related to an Offer.

6. *Define the "Application" Entity Type.* In this model, we want to visualize how the Offer-related activities and Application-related activities interact between them. We already have defined the Entity Type for the Offers through the OfferID attribute, so the next step is to define the Entity Type for the Applications. To do this, we go back to the "Entities" panel and click on "New Entity". On the "Entity: Attribute Selection" window, we select "case" and click on "Create Entity".

7. *Create the Derived Entity.* The derived entity will help us to connect the events from Offers and Applications through a new Entity Type. To do this, we click on the "New Derived Entity" button in the "Entities" panel. Then, on the "Derived Entity: Entity Selection" window, we select the two entities that we have created, "OfferID" and "case", and click on "Create Derived Entity".

8. *Using the derived Entity Type, define the "Activity+Lifecycle" Class.* To define the Class, we go back to the "Classes" panel and click on "New Class". Similar to what we did for the previous class, in the "Class: Attribute Selection" window, we select both the "Activity" and "lifecycle" attributes, and then click on "Continue". Then, on the "Entity: Attribute Selection" window, we select the Entity Type previously defined, the "OfferIDcase", and click on "Create Class".

9. *Create the "Application" model.* We click on "Generate Model" in the "Algorithms" panel. In the "Heuristic Miner: Parameter Selection" window, first we can select "OfferIDcase" as the Entity Type. Then, we expand the "Advanced Configuration" menu, check on the "Use Advanced Thresholds" option and move the slider for the Length-2 Loop threshold to 0.7. Finally we click on "Generate Model".

10. *Visualize the "Application" model.* We can visualize this model by selecting the "HM_2" model from the list on the "Models" panel and clicking on "Show Model". We can look for the activities that start with "O_" to analyze where these activities interact with the Application-related activities, but we can execute one additional step in the tool to obtain a better way to identify these activities.

11. *Compare the models.* We can select both models and then click on the "Show Petri Net" button to see the comparison between the models in the Graph panel. We can click on any transition from the "HM_1" (Offers) model to see where it appears in the "HM_2" (Offers+Applications) model.

E.2 Execution of Experiment 2

In this section we describe the details on the execution of the second experiment for the evaluation of the tool, described in Section 7.2. First, we describe the steps taken in ProM. Then, we describe the steps taken in our tool.

E.2.1 ProM

1. *Import the event data into the ProM Workspace.* To do this, we click on the "Import" button, select the first CSV file, named "BPIC17_Sample _20cases.csv", and click on "Open". In the popup window, we leave the default import plugin and click "Ok".

2. *Create the XES file.* To do this, we first click on the "Use resource" (Play) button. Then, on the "Actions" view, we select the "Convert CSV to XES" action and click on "Start". On the "Configure CSV Parser Settings" window, we click on "Next". Then, on the "Configure Conversion from CSV to XES" window, the selected event columns must be "Activity" and "lifecycle". Once these two columns are selected, we click on "Next" and in the next window we click on "Finish".

3. *Save the log to disk.* We go back to the "Workspace" view and select the XES event log. Then, we click on "Export to disk". Finally, we assign a name to the log file and click on "Save".

4. *Create the model.* In the "Workspace" view, we select the recently generated XES event log, and click on the "Use resource" button. In the "Actions" view, we search for "Heuristic Miner" to filter the actions. Then, we select the "Interactive Data-aware Heuristic Miner (iDHM)" action and click on "Start". Then, on the popup window, we click "Continue".

5. *Save the model to disk.* In the Heuristic Miner view, we click on "Export model" and click "Ok" on the popup window. Then, we go back to the "Workspace" view, select the Causal net that was just generated and click on "Export to disk". We assign a name to the file and click on "Save".

6. Repeat steps 1-5 for the 4 remaining CSV files included in Table 7.3.

E.2.2 Graph Tool

1. *Import the event data into the database.* To do this, we use the "Upload CSV File" option from the tool to select the first CSV file, named "BPIC17_Sample_20cases.csv". On the "Select Activity Attribute", we select "Activity" and click on "Continue". Then, on the "Select Event Attributes" window, we select all the remaining attributes and click on "Finish".

2. *Define the Entity Type for the model.* To do this, we expand the "Logs" panel and select the recently uploaded log, whose name is the same as the one for the CSV file, "BPIC17_Sample_20cases.csv". After the log name appears in the Log Label, we expand the "Entities" panel and click on "New Entity". On the "Entity: Attribute Selection" window, we select "case" and click on "Create Entity".

3. *Define the Class.* To do this, we expand the "Classes" panel and click on "New Class". In the "Class: Attribute Selection" window, we select both the "Activity" and "lifecycle" attributes, and then we click on "Continue". Then, on the "Entity: Attribute Selection" window, we select the Entity Type previously defined, the "case", and click on "Create Class".

4. *Create the model.* We expand the "Algorithms" panel, select "Heuristic Miner" from the dropdown menu and click on "Generate Model". To generate the process model, we will use the default attributes, so, on the "Heuristic Miner: Parameter Selection" window, we can click directly on "Generate Model".

5. *Visualize the model.* We expand the "Models" panel, select the "HM_1" model from the list and click on "Show Model".

6. Repeat steps 1-5 for the 4 remaining CSV files included in Table 7.3.