Visualizing Software Architecture using Dynamic Data

Master Thesis

Eileen Riedel
2288419
e.riedel@students.uu.nl

First supervisor: Dr. ir. Jan Martijn van der Werf
Second supervisor: Dr. Nishant Saurabh
External Supervisor: Robbert Rosseboom

Universiteit Utrecht

Business Informatics
Department of Information and Computer Science

July 2024

Abstract

This thesis explored how software architecture can be visualized using dynamic data. To address this, literature reviews and experiments were conducted to determine the necessary data requirements. Subsequently, various visualizations were evaluated by stakeholders to identify the most effective approach for representing software architecture. Finally, the Interaction Network Visualization was developed to provide a comprehensive and effective visualization of software architecture.

The thesis identified two distinct methods for visualizing software architecture using dynamic data. Tracing data can be visualized using process mining techniques, particularly Directly Follows Graphs, as they are well-suited for identifying and visualizing sequences. For visualizing data that offers more hierarchical and structural information about the software architecture, graph-based visualizations, such as the Interaction Network Visualization, are more appropriate. These visualizations provide greater flexibility in identifying and visualizing the overall structure of a software architecture.

Contents

Chapter 1

Introduction

Software architecture is seen as an important aid to software development and maintenance. However, it is frequently observed that software systems within corporate environments are, if at all, inadequately documented [1]. This lack of documentation, regardless of the cause, can lead to various issues in regards of maintenance or security. With the introduction of process mining, which gathers information about the process as they take place, several frameworks and solutions have been developed [2].

In [3] the Integrated Component Identification Framework was proposed, which identifies a set of components for a given software system using execution data in the process mining framework ProM [4] to construct information about a running system. While the framework is able to identify sets of components, it cannot reconstruct the overall architecture. Furthermore does the framework not allow for the evaluation of the software architecture, including the identification of bottlenecks or measuring quality attributes.

In [5], a ProM plugin that uses dynamic inputs was developed, to create a hierarchical view that shows the interactions of different elements within a system. The developed plugin uses logs of object interactions to analyze and transform them into a hierarchical interaction model, which depicts container interaction. While the developer involvement is minimal, the plugin is limited to instrumenting Java code.

While comprehensive documentation is the initial step in the right direction, it still leaves it up to the reader to identify issues or bottlenecks related to quality attributes within the software system. Although the Integrated Component Identification framework provides a starting point towards an automated solution, it focuses more on a process event perspective rather

than a software architecture perspective. Other challenges are the discovery of component interactions via interfaces, and reconstructing the overall software architecture [3]. The hierarchical interaction model identifies container interactions for java systems, leaving other systems unaddressed [5].

As this shows, several process mining solutions have been developed, to address the lack of software architecture documentations. Frameworks to identify components already exist, but lack the overall reconstruction and evaluation of the software architecture [3]. Other plugins [5] show a more hierarchical view, but are limited in their system compatibility. While existing frameworks and plugins provide a first step into documenting software architecture, challenges still persist.

To address these challenges, a framework can be introduced which generates a visualization from a software architecture perspective. In Addition, parameters for analyzing quality attributes could be incorporated to address the above mentioned challenges. Furthermore should event logs from different systems be able to be analyzed by the framework, making it possible to include multiple different systems in the component identification.

Problem statement

The objective of this thesis is to use dynamic data from existing software architectures to visualize them. By doing so, this research aims to offer an approach to documenting, understanding, and analyzing software systems within corporate environments. For this, dynamic data will be used, as it is a form of data present within every system. The focal point lies in identification and visualization of software architecture components, leading to the following Research Question (RQ):

`RQ: How can we visualize software architecture using dynamic data?`

Chapter 2

Research Approach

The following sections outline the research questions addressed in this thesis and the methods employed to answer them. Additionally, a running example and case study will be explained, which serve as experimental frameworks for testing the proposed solutions. Finally, an overview of the tools utilized throughout the thesis will be provided.

2.1 Research Questions

To address the problem statement described in Chapter 1 the following research question (RQ) has been formulated:

`RQ: How can we visualize software architecture using dynamic data?`

To answer the research question, several sub-questions (SQs) have been defined.

The effectiveness of visualizing software architecture depends on selecting the right event data. Therefore, the first step is to define what data is relevant. It ensures the visualization includes only the information needed to accurately represent the architecture, preventing misinterpretations. Focusing on relevant data makes the visualization easier to understand and navigate.
This results in the following sub-question:

`SQ1: Which event data is relevant for visualizing software architecture?`

Different visualization techniques offer varying degrees of clarity and effectiveness in communicating complex software architectures. Identifying the most effective technique ensures that the intended message is communicated accurately to diverse stakeholders. Therefore, the following sub-question needs to be answered:

SQ2: Which visualization techniques are considered the most effective for representing software architecture?

Complex software systems with dynamic behavior can be difficult to understand. Visualizations can help identify bottlenecks, support quality assurance, or facilitate communication between stakeholders. This makes it crucial to understand how dynamic data can be utilized for visual representations, leading to the following sub-question:

SQ3: How can dynamic data from existing software architectures be used for visual representations to aid in understanding complex software systems?

2.2 Research Methods

The research of this thesis will follow the principles of design science as described in [6]. After the problem statement in Chapter 1, background information on the research will be covered. This will be followed by an analysis of the required data, which will result in answering SQ1. For this, the data will be processed to extract the necessary information, which will then be visualized using process mining techniques to verify if the extracted data meets the requirements to visualize software architecture. Afterwards, literature will be analyzed to identify visualization techniques for software architecture. The results of the analysis will be analyzed with experiments and stakeholder interviews, resulting in answering SQ2. Based on these results, experiments will be conducted to identify how software architecture can be visualized using dynamic data, answering SQ3 in the process of it. The results will again be evaluated with a case study.

2.3 Running Example

For a running example of this thesis, the group project[1] of group 7[2] of the course "Cloud and Edge Computing" at the Utrecht University from 2023 will be used.

The project is about a research lab where different scientists conduct experiments. The research facility contains multiple labs where scientists can run experiments for their research, some of which require monitoring of the temperature throughout the experiment. For this, a Temperature Observability Microservice is used, which provides historic trace of temperature values, and notifies scientists when temperature values are not within a predefined range of the experiment. For each experiment there is an arbitrary amount of sensors monitoring the temperature. The average temperature of all sensors of an experiment result in the physical space's temperature, which is defined for each experiment by the scientist when configuring the experiment. Each experiment is carried out for as long as the scientist decides to.

The Temperature Observability Microservice keeps track of multiple experiments running concurrently. Its architecture, including the Kafka topic and the notification service, are illustrated in Figure 2.1.
The consumer service reads from the Kafka topic, where the experiment data are being produced. Each message is taken in one by one, deserialized, and send to the appropriate URL path of the API. Once the API received all measurements of an experiment from the consumer, it aggregates the measurements into a single mean value after which it will check whether the scientist should be notified or not. If a scientist needs to be notified, the API sends a message to the notification service, which will then notify the scientist. Moreover, the API is responsible for the connection to the PostgreSQL database for the persistent storage. Scientists can request from the API historic temperatures of an experiment, and when a temperature was out or range.
The individual components will log either the data they are sending, the data they are receiving, or both. The arrows indicate the data flow between the components. Event logs containing the necessary data will be retrieved from the individual components.

[1] https://github.com/landaudiogo/cc-assignment-2023.git
[2] https://github.com/EC-labs/cc2023-g7.git

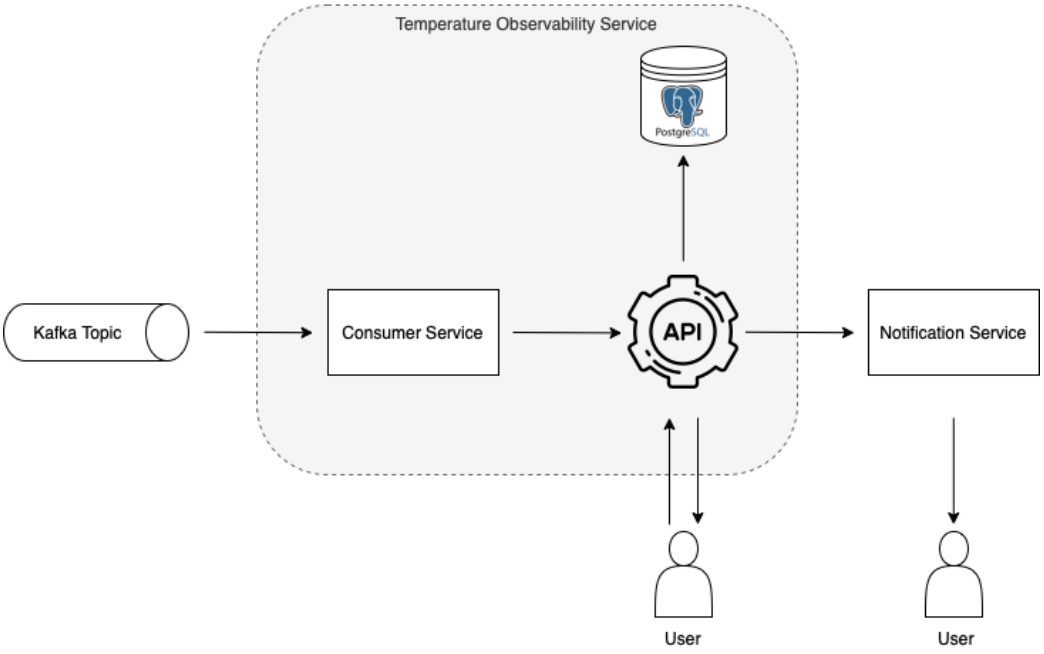


Figure 2.1. *Architecture of the running example.*

2.4 Case Study

To include real-life complexity of software architectures within companies, a case study within the company Bol will be concluded as well.
Bol (formerly known as bol.com) is a web-shop in the Netherlands, offering products in various categories since 1999. As of January 1st 2024, Bol has 13 million active customers in the Netherlands and Belgium, a product range of 38 million items and 50,000 sales partners [7].
The company made two different datasets available to use within this thesis - one including data from dynamic tracing, and one including application connecting data. Furthermore will stakeholder interviews be conducted in regards of visualizations.

Connections Dataset

The dataset including application connection data shows data about applications that are allowed to connect. This means it contains connections that not necessarily have been made, but could be made if wanted. As seen

in Figure 2.2, the relevant columns are named *application_shortname*, *depend_application_shortname*, *opex*, *productGroupShortName*, *productAreaShortName*, and *productDomainShortName*.
The *application_shortname* makes calls to the *depend_application_shortname*. *opex* is a Bol specific name referring to the team name and, like *productGroupShortName*, *productAreaShortName*, and *productDomainShortName*, refers of the *application_shortname*.
The dataset contains 26,535 lines of data and did not have to be cleaned.

application_shortname	depend_application_shortname	opex	productGroupShortName	productAreaShortName	productDomainShortName
sup-brands	brand-registry	teambrandnew	brand-experience	partner-experience-ecommerce	e-commerce
wh-outbound	tetris	team5p	outbound	warehousing	logistic-services
wh-outbound	tetris	team5p	outbound	warehousing	logistic-services
wh-outbound	tetris	team5p	outbound	warehousing	logistic-services
gitlab	auditpe	teamcicd	engineering-tooling	technical-platform	tech-enabling
nps	avatar	team42dt			
sec-approvals	tonto	teamsecops	security_insights	security	tech-enabling
wh-outbound	cer	team5p	outbound	warehousing	logistic-services

Figure 2.2. *Extract of the Connections Dataset.*

From the column names it can be derived, that the dataset provides information about the structure of the software architecture at Bol. The *application_shortname* and the *depend_application_shortname* columns provide information about the interaction of applications within Bol, while *opex*, *productGroupShortName*, *productAreaShortName*, and *productDomainShortName* provide more hierarchical information.

Trace Dataset

The dataset including data from dynamic tracing includes data about the web-shop, as this is currently the only area where dynamic tracing is applied within Bol. It starts tracing actions done by customers once they enter the web-shop page of Bol.com, such as accessing their user profile, looking for products, or writing reviews.

As seen in Figure 2.3, the columns of the file are named *spanId*, *parentSpanId*, *http_url*, *istio_namespace*, *istio_canonical_service*, *traceId*, and *startTime*. The values within the *http_url* column have been changed in order to obtain privacy for Bol.

The *parentSpanId* refers to the *spanId* which came before the *spanId* in terms of sequence, as shown in Figure 2.4. The *http_url* contains the destination URL of the *spanId*, and the *istio_namespace* contains the destination application of the *spanId*. The *traceId* identifies the *spanIds* belonging to-

	spanId	parentSpanId	http_url	istio_namespace	istio_canonical_service	traceId	startTime
2	ca8b4dc7d6b9b537	1b17bd54a9c9ce1f	http://service:path	wspc	wspc-deployment	0023745ae48f07b5beaf68213c151457	2024-01-15 10:51:45.480087 UTC
3	c9dbd2d8d7fbf8fe	2c54bf5baa83384a	http://service:path	harvey	harvey	0023745ae48f07b5beaf68213c151457	2024-01-15 10:51:45.504558 UTC
4	ac843fafb9b13e5d	d4fea51abb38a9e5	http://service:path	harvey	harvey	0023745ae48f07b5beaf68213c151457	2024-01-15 10:51:45.500815 UTC
5	6c1c7b1c2d30ee97	269c1930d195766d	http://service:path	wspc	wspc-deployment	0023745ae48f07b5beaf68213c151457	2024-01-15 10:51:45.47259 UTC
6	d80c6efc0a4bc2ff	ca8b4dc7d6b9b537	http://service:path	basket	basket-deployment	0023745ae48f07b5beaf68213c151457	2024-01-15 10:51:45.480386 UTC
7	a913ff5c6743acd5	e0314678be191b21	http://service:path	harvey	harvey	0023745ae48f07b5beaf68213c151457	2024-01-15 10:51:45.500003 UTC
8	38f0ea8db7dd5d98	6ede28db2dc000ed	http://service:path	escherserve	escherserve	0023745ae48f07b5beaf68213c151457	2024-01-15 10:51:45.516569 UTC

Figure 2.3. *Extract of the Trace Dataset.*

gether.

Lines of data with empty fields in the *http_url* column have to be removed, as they are considered incomplete. However, empty fields in the *istio_namespace* column are not considered incomplete and therefore do not have to be removed. The original file contains 378,425 lines of data. However, about two-third of the file had to be removed during the data cleaning process due to incomplete data, leaving a total of 119,271 lines of data.

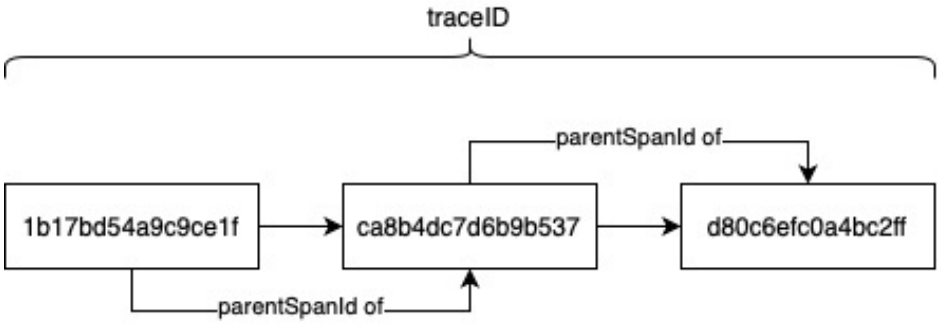


Figure 2.4. *Example of the spanId and parentSpanId connection.*

Stakeholder Interviews

The stakeholder interviews will be conducted regarding visualizations of software architecture. The intention with this is to receive opinions about the visualizations in regards of usability and displayed information, as well as improvement for their work.

The stakeholders interviewed are employees of Bol. Most of them are IT architects, while few are managers. All of the interviewed stakeholders are familiar with the data which will be used to create the visualizations. While some had a good understanding of a bigger amount of the data, at least everyone had a specific part which they were extremely familiar with due to working with it on a regular basis. Each visualization was shown

to the stakeholders and briefly explained. They were then asked about the following topics:

- Their general opinion of the visualization.
- What they like or don't like about the visualization.
- What information they can obtain from the visualization.
- What other information they would like to see in the visualization.
- If they can think of use cases when they would use this visualization for work related topics.

The outcomes of the interviews will be used to evaluate the visualizations based on qualitative feedback from stakeholders regarding perceived error rates, efficiency improvements, and overall satisfaction. The visualizations should positively impact the stakeholders' work, making their tasks less error-prone and less time-consuming.

2.5 Tools for Process Mining and Visualization

This chapter will give an overview of the tools and libraries used throughout this thesis. The usage of each depends on the task to be achieved. Each tool and library comes with its own advantages and disadvantages, which will be addressed in their respective chapter where they are used.

2.5.1 PM4Py

To incorporate process mining techniques, the Python library PM4Py has been used. Its function like `pm4py.read.read_xes()` reads files in XES format, which is the standard format for event data, into a pandas data frame. This allows for easy processing of the data for further process mining techniques. Furthermore, does it allow for process discovery, where it creates directly-follows graphs or petri-nets. The idea behind using PM4Py is to use the incorporated process mining techniques in order to identify processes, and to apply process mining statistics.

2.5.2 D3.js

It was decided to use D3.js[3], which is a free, open-source JavaScript library for visualizing data using dynamic, data-driven graphics. An advantage of using D3.js is that it already provides ready-to use solutions where only the own dataset needs to be provided. It consists of 30 discrete libraries which can be combined as needed [8].
Furthermore does it provide a gallery with different examples, divided into subcategories. This makes it easier to test different types of visualizations to be able to quickly determine what visualization works best.

[3]https://d3js.org

Chapter 3

Background

In this background chapter, the context of this research will be covered. It starts by giving an introduction to software architecture and its concepts, as well as its evaluation methods. It continues to give an overview of visualization techniques for software systems. Lastly, the topics of dynamic tracing, process mining, and architecture mining will be explained.

3.1 Software Architecture

In [9] software architecture is defined as a "set of structures needed to reason about the system, which compromise software elements, relations among them, and properties of both". According to this definition, architecture is seen as a set of software structures, where a structure is a set of elements and its relations. In other words, the architecture describes which high-level components a software system consists of, which responsibilities these components have towards other components in the system, and how these components are organized and interact [9, 10].

Software architecture is considered a key asset for any organization that builds complex software-intensive systems [10]. Reason for that is modern systems becoming more complex, which makes it difficult to grasp the system all at once [9]. Therefore, [11] focuses on three fundamental concepts of software architecture: stakeholders, viewpoints, and perspectives.

Stakeholders

A stakeholder is considered everyone who is affected by the system. Their needs is the reason why the system is created in the first place, making

meeting their need the main goal. It is therefore important to know how to work with stakeholders: identify them, understand their concerns, and balance often conflicting needs. The goal should be to design an architecture that addresses requirements as effectively as possible [11].

Viewpoints

An architectural view is a description of one aspect of a systems architecture. It can be seen as the application of the principle of separation of concerns, as considering a systems architecture through a number of distinct views can help to understand, define, and communicate a complex architecture, without overwhelming its readers. While a view is a representation of one or more structural aspects of an architecture, a viewpoint provides a framework for constructing a view. Collectively, all views together describe the whole system [11].

Perspectives

Nonfunctional factors, such as performance, security, or availability, are called quality properties (or quality attributes). Their goal is to describe how the system provides its services and is therefore crucial for an architecture. To address a particular quality property, architectural perspectives are used. Perspectives are a complementary concept to viewpoints and help structure the architecture definition process by separating concerns [11].

3.2 Evaluation Methods

Software architecture plays a significant role in achieving system-wide quality attributes. Therefore, methods for evaluating the quality attributes of a system are important, since they can assess whether or not an architecture will lead to the desired quality attributes [12]. Software architecture evaluations can reduce the possibility of risks and verify quality requirements on one hand, and manage and understand existing systems on the other. Furthermore can they determine the attributes characteristics, or identify potential risks in architecture design. Evaluation methods can be performed at various stages of the software development process. They can be performed during the early design stages, or they can be used to evaluate existing systems before they undergo future maintenance or enhancements [10, 13].

Evaluation methods can be divided in four main categories: experience-

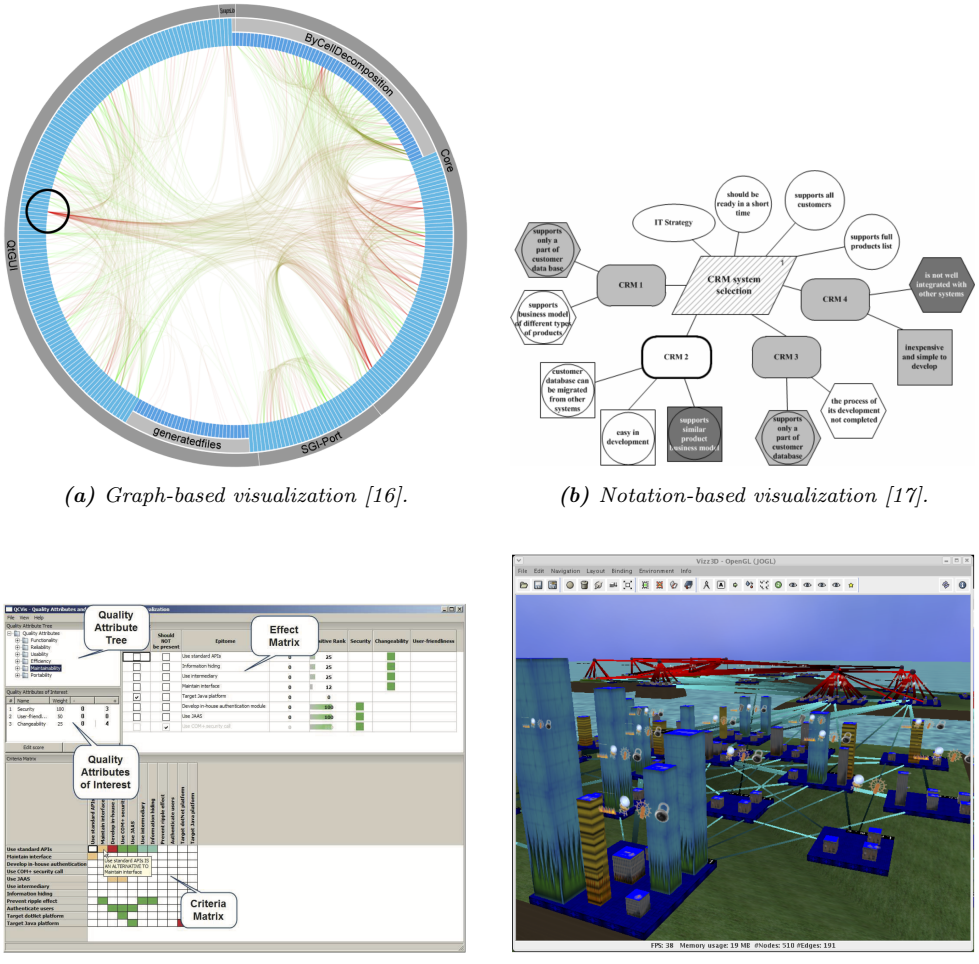
based, simulation-based, mathematical modelling, and scenario-based. Experience-based evaluations are based on previous experience and domain knowledge to say if a software architecture will be good enough. Simulation-based evaluations create a high-level implementation of some or all of the components in the software architecture. The results of the simulation are then used to evaluate quality requirements. Mathematical modelling evaluations use mathematical proofs and methods to mainly evaluate operational qualities. They can be combined with simulation-based evaluations to create more accurate results. Scenario-based evaluations create a scenario profile for a particular quality attribute to step through the software architecture. The consequences of the scenario are then documented [10, 13].

3.3 Visualization Techniques

Over the years, software systems have become larger and more complex. This has lead to an increased interest in visualization techniques (VTs) as they are used to communicate and understand software architectures of large scale complex systems. Software architecture visualization is used to help stakeholders (e.g. architects, developers, testers, project managers) of a system to not only reason and understand the designed architecture, but also to manage and evolve software intensive systems [14]. Visualization in computer graphics is used to enhance information understanding by communicating information which may otherwise not be easy to describe and understand in other formats, e.g. textual. This is achieved by creating images, diagrams, or animations [14]. While software visualization is defined as the visual representation of artifacts related to software and its development process [15], software architecture visualization is defined as the visual representation of architectural models and some or all architectural design decisions about the models [14].

In a systematic literature review (SLR) in [14], it was found that visualization techniques can be classified into four types: graph-based, notation-based, matrix-based, and metaphor-based. Examples for each visualization type can be found in Figure 3.1. Graph-based visualization techniques make use of nodes and links to represent the structural relationships between architectural elements. Their emphasis is more on the overall properties of a structure than on the types of nodes. Notation-based visualization uses a combination of the modeling techniques SysML (systems modeling language), UML (unified modeling language) and specific notation-based visu-

alization. It tries to represent the different relationships between elements in a structure, as well as the role of it. To achieve this, various notations are provided. Matrix-based visualizations provide a complementary representation of a graph when a graph is large or dense. Metaphor-based visualization uses contexts from the physical world (e.g. cities) to visualize entities and their relationships. This makes the visualization process rather intuitive and effective [14].



(a) Graph-based visualization [16]. *(b) Notation-based visualization [17].*

(c) Matrix-based visualization [18]. *(d) Metaphor-based visualization [19].*

Figure 3.1. *Examples of four visualization techniques used in software architecture.*

In [20], architecting activities were classified, which are various activities performed by architects for different purposes during the architecting process. Those activities are used towards the construction of the architecture of a software-intensive system. General architecting activities include

architecture recovery, architecture description, architecture understanding, change impact analysis, and architecture reuse. Specific architecting activities are architectural analysis, architectural synthesis, architectural evaluation, architectural implementation, and architectural evolution [14].

According to the SLR, architecture recovery was the most popular architecting activity that was being supported by various visualization techniques. It is followed by architectural evolution, architectural evaluation, change impact analysis, architectural analysis and architectural synthesis. It is also stated that graph-based visualization techniques are mostly employed in architectural recovery and architectural evolution, while there is no visualization technique that can support all architectural activities [14].

The (level of) tool support is an important concern of stakeholders when it comes to software architecture visualization techniques. It can be considered an indicator of the limitations and can indicate whether the reported techniques are theoretical or practical. According to the SLR, visualization techniques mainly use various kind of tools (92.%), followed by semi-automatic tools (47.1%), automatic tools (41.5%) and manual support (11.3%) [14].

According to [14], the domains graphics software, distributed system, and information management system have gained the most attention when it comes to applying architecture visualization techniques to support architecting activities. Altogether, visualization techniques have been validated in 16 domains with the most popular architecture visualization techniques used in industry being graph-based visualization technique (38), matrix visualization technique (9), notation visualization technique (7), and metaphor visualization technique (6).

3.4 Dynamic Tracing

Dynamic tracing is an observation technology, enabling dynamic instrumentation of unmodified kernel and user software. It allows for analyzing the behavior and performance of a software system during its execution, both in development and production. The provided observability is achieved through the use of instrumentation points called probes, and therefor not requiring code modifications. By instrumenting the code at runtime it can collect real-time information about various aspects of the system including function calls, variable values, or resource utilization. This way it enables insights into how

the program behaves in different scenarios without the need for pre-defined static instrumentation [21].

Its capabilities are vast, allowing troubleshooting of any software function, detailed observation of devices and core resources, and analysis of the entire software stack. Moreover, it can be utilized in production environments without the need for restarting or modifying applications or operating systems [21].

Dynamic tracing inserts instrumentation points, called probes. These probes can trace execution flow through code, collect relevant data, and provide insights into kernel behavior, application performance, and network activity. If a probe has been enabled and the code where the probe has been inserted executes, the probe will fire. Once the actions assigned to the probe have been taken, the code resumes executing normally. Dynamically generated probes alter code only when they are in use, ensuring minimal impact on performance when disabled [21].

A well-known tool for dynamic tracing is DTrace, which was first introduced in 2005. Previous to DTrace, existing tools consisted of several limitations, such as adding performance burden to the running system, they required special recompiled versions of the software to function, several different tools were needed to give a complete view of the system behavior, they had limited available instrumentation points and data, and they required significant postprocessing to create meaningful information from the gathered data. DTrace addresses these limitations by offering a comprehensive observability across the entire software stack, making it able to understand software systems [21].

3.5 Process Mining

Process mining emerged as a new research field over the past decade due to the growth of event data and the maturing of process mining techniques. Its overarching goal is to focus on analysing processes using event data. Other goals include to discover, monitor, and improve real processes in a variety of application domains. The main drive of process mining is the increasing amount of recorded events, which are needed to improve and support business processes in competitive and rapidly changing environments [22, 23].

Process mining is a field between computational intelligence and data mining on one hand, and process modeling and analysis on the other. While

it is seen to be related to data mining (mostly data-centric), process mining is process-centric. It is also truly intelligent (learning from historic data) and fact based (based on event data and not on opinion) [23]. While classical data mining techniques don't focus on business process models and often only focus on a specific step, process mining focuses on the end-to-end processes [22]. This makes it an important bridge between data mining and business process modeling and analysis [23].

There are three types of process mining which event logs can be used for: discovery, conformance, and enhancements. A discovery technique takes event logs and produces a model without using any prior information. A conformance technique, on the other hand, compares an existing model with an event log of the same process. This is done to check if reality, as recorded in the log, conforms to the existing model. An enhancement technique is used to extend or improve an existing process model [22].

Tool support

Different tool support for process mining exists, both software tools with a graphical user interface, and programming libraries. Some of these will be introduced in the following paragraphs.

ProM is a plugable generic open-source process mining framework for implementing process mining algorithms in a standard environment. Since its introduction it has allowed many developers in different countries to contribute their research in the form of plug-ins [24, 25].

Process Mining for Python (PM4Py) is a process mining library which was introduced in [26], providing integration with state-of-the-art data science libraries, e.g., pandas, numpy, scipy, and scikit-learn. It aims to provide a solution which does not rely on a graphical user interface and facilitates a usage in a large-scale experimental setting. Furthermore does it provide a wider support for algorithmic customization [26].

eXtensible Event Stream (XES) is an XML-based standard for event logs. It stores event logs from many different information systems directly to provide a generally-acknowledged format for the interchange of event log data. It has been adopted by the IEEE Task Force on process mining as the default format for event logs [23, 25, 27].

```
<log>
  <extension name="Lifecycle" prefix="lifecycle"
      uri="http://www.xes-standard.org/lifecycle.xesext" />
  <extension name="Time" prefix="time"
      uri="http:///www.xes-standard.org/time.xesext" />
  <extension name="Concept" prefix="concept"
      uri="http:///www.xes-standard.org/concept.xesext" />
  <extension name="Semantic" prefix="semantic"
      uri="http:///www.xes-standard.org/semantic.xesext" />
  <extension name="Organizational" prefix="org"
      uri="http:///www.xes-standard.org/org.xesext" />
  <extension name="Order" prefix="order"
      uri="http://my.company.com/xes/order.xesext" />
  <global scope="trace">
    <string key="concept:name" value="unknown" />
  </global>
  <global scope="event">
    <string key="concept:name" value="unknown" />
    <string key="lifecycle:transition" value="unknown" />
    <string key="org:resource" value="unknown" />
  </global>
  <classifier name="Activity classifier" keys="concept:name lifecycle:transition" />
  <string key="concept:name" value="Example log" />
  <trace>
    <string key="concept:name" value="Order 1" />
    <float key="order:totalValue" value="2142.38" />
    <event>
      <string key="concept:name" value="Create" />
      <string key="lifecycle:transition" value="complete" />
      <string key="org:resource" value="Wil" />
      <date key="time:timestamp" value="2009-01-03T15:30:00.000+01:00" />
      <float key="order:currentValue" value="2142.38" />
      <string key="details" value="Order creation details">
        <string key="requestedBy" value="Eric" />
        <string key="supplier" value="Fluxi Inc." />
        <date key="expectedDelivery" value="2009-01-12T12:00:00.000+01:00" />
      </string>
    </event>
  </trace>
</log>
```

Figure 3.2. *XES example log [25].*

Figure 3.2 shows a XES log example. XES uses log, trace, and event elements to define the structure of the document, meaning those elements do not contain any information themselves. The log element contains all event information that is related to one specific process. The trace element describes the execution of one specific instance, or case, of the logged process. The event element represents atomic granules of activity that have been observed during the execution of a process [27].
To store any data in XES format, attributes are used. Every attribute has a string based key, a known type, and a value of that type. Possible types are string, date, integer, float, and boolean. Attributes can have attributes themselves, which allows to provide more specific information.
Extensions are used to define precise semantics of an attribute. There are standard extensions and user-defined extensions. Standard extensions include concept extensions, lifecycle extensions, organizational extension, time extension, and semantic extensions.
To make events comparable to other events, event classifiers are used, which can be specified in the log element.
The global element is used to specify in the log that certain attributes have well-defined values for every trace and/or event [25].

3.6 Architecture Mining

In [28] the Architecture Mining Framework was introduced as a way to use runtime software execution data to assess and improve the quality of software architecture. Architecture Mining addresses the collection, analysis, and interpretation of software operation data to foster architecture evaluation and evolution. Its aim is to close the loop between the intended software architecture and the realized software architecture by monitoring and analyzing the realized system [28, 29].

The five main activities of architecture mining as proposed by [28] can be seen in the gray area of Figure 3.3: *Architecture Reconstruction*, *Evolution Analyzer*, *Architecture Conformance*, *Runtime Analyzer*, and *Architecture Improvement Recommender*.

The architecture mining framework seen in Figure 3.3 does not imply an order between the different phases, only dependencies are shown. The black arrows indicate the dependencies between different elements, while the red arrows indicate where software execution data can be utilized.

In the framework, the intended architecture is shown separate from

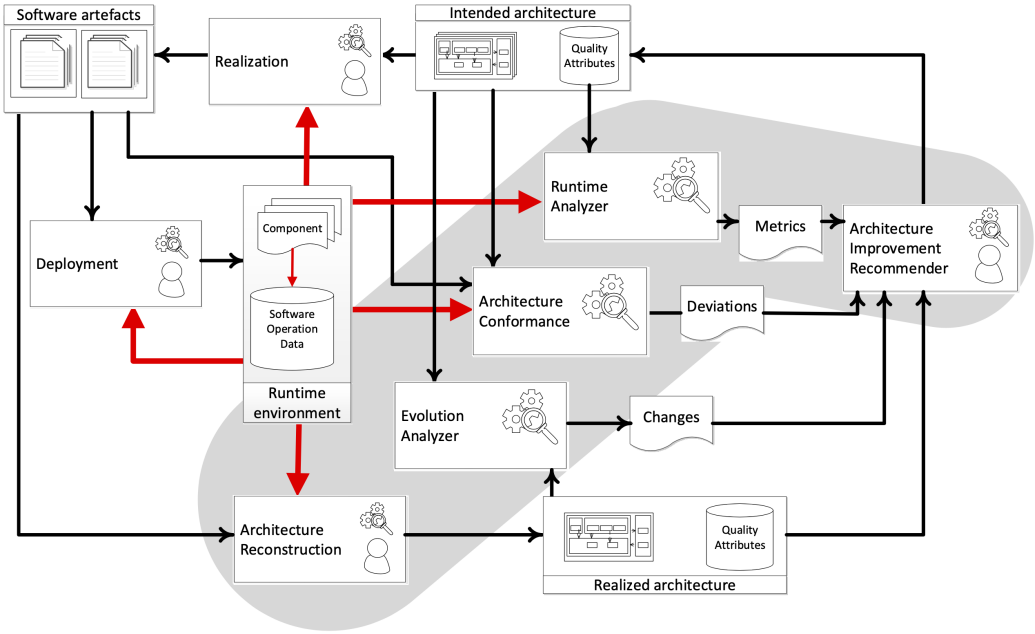


Figure 3.3. *Architecture Mining Framework [28].*

the realized architecture due to architectural erosion, which is the realization of software typically tending to drift away from the intended architecture. Both, the intended and the realized architecture, consist of a set of structures, represented by architectural views, and quality attributes [28].

The intended software is realized to create software artefacts, which are then deployed. This results in a operational software system, creating software operation data, which is the main input for architecture mining. Software operation data is then used for three activities: *Architecture Reconstruction*, *Runtime Analyzer*, and *Architecture Conformance*. The *Architecture Reconstruction* does a reverse engineering by combining the realized software artefacts and the dynamic information captured in software operation data. This activity results in the *Realized Architecture*. *Architecture Conformance*, together with the *Intended Architecture*, validates whether an architecture conforms to the intended architecture, resulting in a set of deviations. The *Runtime Analyzer* uses software operation data to analyze to which degree the quality attributes in the intended architecture are satisfied. This analysis results in metrics. The *Evaluation Analyzer* compares the intended and the realized architecture and gives insights in how they drifted apart. This results in a set of changes. The *Architecture Improvement Recommender* then uses the realized architecture, the metrics, deviations, and

changes as input and results in an improved intended architecture [28, 29].

In previous works, [5] developed an approach that uses dynamic inputs to create a hierarchical view that shows interactions of different elements within a system under study. The approach uses software execution data as an input, which consists of logs including method calls with their callers and callees registered. A ProM plugin was developed, which can process these logs and use them to visualize the system under study as a hierarchical interaction model. The model allows users to abstract parts of the system and select interactions between software elements for process mining. Evaluations of the plugins showed that the approach created accurate models without developer effort [5].

In [30], an approach was taken to rely on the software operation data generated by the system to gain new insights for software architects. The Architectural Intelligence Framework, or ArchitectureCity, was developed, which uses the analogy of cities to visualize the runtime of software. Buildings represent individual architectural elements and are grouped in districts based on different clustering techniques. Streets depict the traffic between the different districts. The framework is based on software operation data by the system and employs architecture mining to extract and enhance operational data to support the software architect. This way, it is able to capture dynamic aspects of running systems to construct relevant architectural views and perspectives for stakeholders by analyzing the logging data the system produces [30].

3.7 Conclusion

This chapter has provided a comprehensive overview of key concepts and topics essential for understanding the context of the research. It started with a definition of software architecture, emphasizing its significance in handling complex software systems. The chapter then delved into evaluation methods for software architecture, emphasizing its role in assessing system wide quality attributes, reducing the possibility of risks and verifying quality requirements throughout the software development process.

Visualization techniques for software systems were introduced, highlighting their increase in interest, as they are used to communicate and understand software architectures of large scale complex systems. The systematic literature review on visualization techniques revealed four main types: graph-

based, notation-based, matrix-based, and metaphor-based. Architecting activities were introduced as well, were graph-based techniques were widely employed.

The chapter then introduced dynamic tracing, an observing technology which enables real-time insights into the behavior and performance of software systems without code modifications. Its allowance for analyzing a software system during its execution provides a potential source for extracting event logs, which can be then used in process mining.

Process mining was explored as a field dedicated to analyzing processes using event data. The field emerged due to the increasing amount of recorded events, which are needed to improve and support business processes in competitive and rapidly changing environments. With process mining being a field between data mining, and process modeling and analysis, it provides a technique useful for visualizing and analyzing software architectures.

Lastly, the Architecture Mining Framework was introduced as an approach to use software execution data to assess and enhance the quality of software architecture. Its five main activities were outlined, showing the framework's capability to provide a bridge between intended and realized software architectures, as well as emphasizing the loop between it. The chapter concluded by referencing specific works that applied architecture mining concepts for architectural intelligence and hierarchical interaction modeling based on software operation data.

Chapter 4

Scope of the Thesis

The scope of this thesis includes two principle activities: *Architecture Reconstruction* and *Runtime Analyzer* of architecture mining. As described in Chapter 1, documentation of software architectures is notably scare. Consequently, this thesis assumes that there is insufficient documentation for a given software architecture available. This may include a complete absence of documentation, incomplete documentation, outdated documentation, lack of access to static information, or other scenarios. Therefore, architecture reconstruction needs to be performed in order to retrieve the existing architecture of the system, establishing the basis for any further action. This approach involves the use of dynamic data, i.e. event logs, which will be used for reverse engineering, as they can be captured within a running system.

Once the software architecture has been reconstructed, the runtime analyzer will evaluate to which extend quality attributes are satisfied. Given the absence of documentation of the software architecture, there may be a corresponding lack of documentation concerning the quality attributes or their criteria for satisfaction. Therefore, measures to evaluate quality attributes or identify bottlenecks may have to be derived from the data. This can be achieved by using timestamp information of events, which can indicate when a system is particularly loaded or stressed.

The activity of *Architecture Conformance* will not be part of this thesis, given that it is assumed that there is insufficient documentation for a given software architecture available. While the architecture reconstruction and the runtime analyzer result in outcomes which can be used for the *Architecture Improvement Recommender*, this activity will be out of scope of this thesis.

Chapter 5

Data Processing for Software Architecture Visualization

To start visualizing software architecture, data needs to be collected, manipulated, and transformed in order to retrieve meaningful information. Therefore, this chapter will address the topics of data processing to obtain software architecture visualizations.

5.1 Data Requirements

It first needs to be determined what data is needed to achieve software architecture visualizations. For this, it is relevant to know what the goals and aspects of the software architecture that will be visualized are. As explained in Chapter 4, the activities that will be focused on in this thesis are *Architecture Reconstruction* and *Runtime Analyzer*. Therefore, data that will achieve these two activities will be considered relevant.

The main source for software execution data that will be used are event logs from components of the software architecture. Ideally, each log should contain a case ID, an activity name, timestamp(s), and component identifier(s). However, the required information is not always given in one log file. In these cases, event logs should be able to be linked to additional datasets containing the required data.
The case ID and activity name are necessary to be able to apply process mining techniques to the event logs [22]. Timestamps can be used to identify sequences and time sensitive performance metrics. This can include event timestamps, which record the exact time at which a particular event took place within a system, system timestamps, which provide a reference for the

current time within the system and are often used for coordinating and sequencing events, or both. The component identifiers are required to identify the involvement and connection of different components with particular interactions. For an ideal component interaction identification, both the sender and receiver component identifier should be included.
The above mentioned case ID and activity name are requirements for process mining and are therefore defined to suite event logs of processes. Since the data used throughout this thesis is software execution data, and not necessarily process event data in terms of process mining, the definitions for case ID and activity name need to be changed to match the software execution data.
In terms of software architecture, the case ID is the identification of a specific instance or occurrence of a process, such as a transaction ID or an instance ID. The activity name can be a specific action within a process, such as a event name, method name, or action label.

Figure 5.1 shows an example of event logs which were obtained from the consumer service of the running example, which was introduced in Chapter 2.3. It includes date and time of the log generation within the consumer service, the receiving endpoint (i.e., the IP address hosting the consumer service), the data source (i.e., the bootstrap server of the Kafka topic), the activity name, and the content of the message received. The contents of the application_data can be split into their own columns, allowing for the experiment ID to be used as a case ID.

```
date       time     endpoint              source                activity_name        {application_data}
2024-03-11 11:35:06 endpoint: 172.18.0.5 , source: kafka-1:19092 , data: experiment_configured {'experiment': 'f7bef6e0-67e8-446d-9fd9-30c86fe6bace', 'researcher': 'd.landa
2024-03-11 11:35:08 endpoint: 172.18.0.5 , source: kafka-1:19092 , data: stabilization_started {'experiment': 'f7bef6e0-67e8-446d-9fd9-30c86fe6bace', 'timestamp': 171015330
2024-03-11 11:35:08 endpoint: 172.18.0.5 , source: kafka-1:19092 , data: sensor_temperature_measured {'experiment': 'f7bef6e0-67e8-446d-9fd9-30c86fe6bace', 'sensor': 'b94fa
2024-03-11 11:35:08 endpoint: 172.18.0.5 , source: kafka-1:19092 , data: sensor_temperature_measured {'experiment': 'f7bef6e0-67e8-446d-9fd9-30c86fe6bace', 'sensor': '67514
2024-03-11 11:35:08 endpoint: 172.18.0.5 , source: kafka-1:19092 , data: sensor_temperature_measured {'experiment': 'f7bef6e0-67e8-446d-9fd9-30c86fe6bace', 'sensor': '67514
2024-03-11 11:35:08 endpoint: 172.18.0.5 , source: kafka-1:19092 , data: sensor_temperature_measured {'experiment': 'f7bef6e0-67e8-446d-9fd9-30c86fe6bace', 'sensor': 'b94fa
2024-03-11 11:35:08 endpoint: 172.18.0.5 , source: kafka-1:19092 , data: experiment_started {'experiment': 'f7bef6e0-67e8-446d-9fd9-30c86fe6bace', 'timestamp': 1710153308.4
2024-03-11 11:35:08 endpoint: 172.18.0.5 , source: kafka-1:19092 , data: sensor_temperature_measured {'experiment': 'f7bef6e0-67e8-446d-9fd9-30c86fe6bace', 'sensor': '67514
2024-03-11 11:35:08 endpoint: 172.18.0.5 , source: kafka-1:19092 , data: sensor_temperature_measured {'experiment': 'f7bef6e0-67e8-446d-9fd9-30c86fe6bace', 'sensor': 'b94fa
2024-03-11 11:35:08 endpoint: 172.18.0.5 , source: kafka-1:19092 , data: sensor_temperature_measured {'experiment': 'f7bef6e0-67e8-446d-9fd9-30c86fe6bace', 'sensor': 'b94fa
2024-03-11 11:35:08 endpoint: 172.18.0.5 , source: kafka-1:19092 , data: sensor_temperature_measured {'experiment': 'f7bef6e0-67e8-446d-9fd9-30c86fe6bace', 'sensor': '67514
2024-03-11 11:35:08 endpoint: 172.18.0.5 , source: kafka-1:19092 , data: sensor_temperature_measured {'experiment': 'f7bef6e0-67e8-446d-9fd9-30c86fe6bace', 'sensor': 'b94fa
2024-03-11 11:35:08 endpoint: 172.18.0.5 , source: kafka-1:19092 , data: sensor_temperature_measured {'experiment': 'f7bef6e0-67e8-446d-9fd9-30c86fe6bace', 'sensor': '67514
2024-03-11 11:35:08 endpoint: 172.18.0.5 , source: kafka-1:19092 , data: sensor_temperature_measured {'experiment': 'f7bef6e0-67e8-446d-9fd9-30c86fe6bace', 'sensor': 'b94fa
2024-03-11 11:35:08 endpoint: 172.18.0.5 , source: kafka-1:19092 , data: sensor_temperature_measured {'experiment': 'f7bef6e0-67e8-446d-9fd9-30c86fe6bace', 'sensor': '67514
2024-03-11 11:35:08 endpoint: 172.18.0.5 , source: kafka-1:19092 , data: sensor_temperature_measured {'experiment': 'f7bef6e0-67e8-446d-9fd9-30c86fe6bace', 'sensor': 'b94fa
2024-03-11 11:35:08 endpoint: 172.18.0.5 , source: kafka-1:19092 , data: sensor_temperature_measured {'experiment': 'f7bef6e0-67e8-446d-9fd9-30c86fe6bace', 'sensor': '67514
```

Figure 5.1. *Example of event logs of the consumer service.*

Figure 5.2 shows an example of event logs which were obtained from the API of the running example. Similar to the consumer event logs, the API event logs contain date and time of the log generation. Additionally, certain logs include details such as the source (i.e., the IP address of the consumer sending the data), the endpoint (i.e., the API URL where the data was received), and the experiment ID of application data. Meanwhile, other

logs contain the response code generated by the API. The experiment ID can be used as a case ID here as well", while the message that is being sent (e.g., "Information received, "200 OK") can be used as activity name.

```
2024-03-11 11:35:06 [INFO]: Information received. API URL: http://fastapi:80/config/, from: 172.18.0.5, with info: f7bef6e0-67e8-446d-9fd9-30c86fe6bace
2024-03-11 11:35:06 INFO:     172.18.0.5:56010 - "POST /config/ HTTP/1.1" 200 OK
2024-03-11 11:35:08 [INFO]: Information received. API URL: http://fastapi:80/notification/stabilization/, From: 172.18.0.5, With Info: f7bef6e0-67e8-446d-9fd9-30c86fe6bace
2024-03-11 11:35:08 INFO:     172.18.0.5:56020 - "POST /notification/stabilization/ HTTP/1.1" 200 OK
2024-03-11 11:35:08 INFO:     172.18.0.5:56032 - "POST /newtemp/ HTTP/1.1" 200 OK
2024-03-11 11:35:08 [INFO]: Information received. API URL: http://fastapi:80/notification/started/, From: 172.18.0.5, With Info: f7bef6e0-67e8-446d-9fd9-30c86fe6bace
2024-03-11 11:35:08 INFO:     172.18.0.5:56070 - "POST /notification/started/ HTTP/1.1" 200 OK
2024-03-11 11:35:08 INFO:     172.18.0.5:56078 - "POST /newtemp/ HTTP/1.1" 200 OK
2024-03-11 11:35:10 [INFO]: Information received. API URL: http://fastapi:80/notification/ended/, From: 172.18.0.5, With Info: f7bef6e0-67e8-446d-9fd9-30c86fe6bace
2024-03-11 11:35:10 [INFO]: Experiment stored in Database: testdb, Server: postgres, Port: 5432. Experiment ID: f7bef6e0-67e8-446d-9fd9-30c86fe6bace
2024-03-11 11:35:10 [INFO]: Measurements stored in Database: testdb, Server: postgres, Port: 5432. Experiment ID: f7bef6e0-67e8-446d-9fd9-30c86fe6bace
2024-03-11 11:35:10 INFO:     172.18.0.5:48580 - "POST /notification/ended/ HTTP/1.1" 200 OK
```

Figure 5.2. *Example of event logs of the API.*

5.2 Data Transformation

Once the data requirements have been made clear, the next step is to collect the data and convert it into the desired format and structure. The data collection depends on the company that wants to visualize software architecture. It can be available in files right away, or needs to be retrieved from a database or a different source.
Once the data is collected, it can be converted into the desired format and structure. Since process mining will be part of this thesis, converting the data into XES format is the most ideal choice. As mentioned in Chapter 3.5, XES is an XML-based standard for event logs and has been adopted as the default format for event logs in process mining.

Also mentioned in Chapter 3.5 is ProM, a plugable generic open-source process mining framework for implementing process mining algorithms. Additionally to implementing process mining algorithms, it also provides several plug-ins which allow to convert different data formats into XES, such as the *Convert CSV to XES* package by F. Mannhardt, N. Tax, and D.M.M. Schunselaar.

The data of the running example comes in CSV format. Therefore, the *Convert CSV to XES* package has been used to convert it to XES format, as seen in Figure 5.3. In this example, the *date* was selected as case column, and the *record name* was selected as event column. Generally, the case column and the event column have to be selected for every data file individually, depending on the data they contain. When deciding on which column best to use for this selection, it is advised to follow the guidelines in Chapter 3.5.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!-- This file has been generated with the OpenXES library. It conforms -->
<!-- to the XML serialization of the XES standard for log storage and -->
<!-- management. -->
<!-- XES standard version: 1.0 -->
<!-- OpenXES library version: 1.0RC7 -->
<!-- OpenXES is available from http://www.openxes.org/ -->
<log xes.version="1.0" xes.features="nested-attributes" openxes.version="1.0RC7">
    <extension name="Time" prefix="time" uri="http://www.xes-standard.org/time.xesext"/>
    <extension name="Lifecycle" prefix="lifecycle" uri="http://www.xes-standard.org/lifecycle.xesext"/>
    <extension name="Concept" prefix="concept" uri="http://www.xes-standard.org/concept.xesext"/>
    <classifier name="Event Name" keys="concept:name"/>
    <classifier name="(Event Name AND Lifecycle transition)" keys="concept:name lifecycle:transition"/>
    <string key="concept:name" value="XES Event Log"/>
    <trace>
        <string key="concept:name" value="10.03.2024"/>
        <event>
            <string key="endpoint" value="172.18.0.4"/>
            <string key="researcher" value="d.landau@uu.nl"/>
            <string key="sensors" value="[19d400ca-c20b-4b50-b688-6ff39ebe696d; e0294f33-3e2a-416c-ae67-3fd9e4f8db51]"/>
            <string key="concept:name" value="experiment_configured"/>
            <string key="experiment_id" value="c5f5242c-3b0a-42ae-855c-de60259c1e10"/>
            <string key="source" value="kafka-1:19092"/>
            <string key="lifecycle:transition" value="complete"/>
            <float key="upper_threshold" value="26.5"/>
            <date key="time:timestamp" value="1970-01-01T21:26:32+01:00"/>
            <float key="lower_threshold" value="25.5"/>
        </event>
        <event>
            <string key="endpoint" value="172.18.0.4"/>
            <string key="concept:name" value="stabilization_started"/>
            <string key="experiment_id" value="c5f5242c-3b0a-42ae-855c-de60259c1e10"/>
            <string key="source" value="kafka-1:19092"/>
            <string key="lifecycle:transition" value="complete"/>
            <date key="time:timestamp" value="1970-01-01T21:26:34+01:00"/>
            <float key="timestamp" value="1.7101023945040119E9"/>
        </event>
        <event>
            <string key="endpoint" value="172.18.0.4"/>
            <string key="concept:name" value="sensor_temperature_measured"/>
            <string key="experiment_id" value="c5f5242c-3b0a-42ae-855c-de60259c1e10"/>
            <float key="temperature" value="21.111278533935547"/>
            <string key="measurement_hash" value="nftg34wWyV3qppbU"/>
            <string key="sensor" value="19d400ca-c20b-4b50-b688-6ff39ebe696d"/>
            <string key="source" value="kafka-1:19092"/>
            <string key="lifecycle:transition" value="complete"/>
            <string key="measurement_id" value="0685a766-8966-46bf-bbb7-33cf7e79119f"/>
            <date key="time:timestamp" value="1970-01-01T21:26:34+01:00"/>
            <float key="timestamp" value="1.7101023945169919E9"/>
        </event>
```

Figure 5.3. *Example of the converted XES file of the consumer log data from the running example.*

In a next step, the XES files of the consumer log and the API log were combined, to create one file. This was done to be able to create a singular DFG for the combined datasets, as explained in 5.3. PM4Py does not provide a function that allows the combination of two or more XES files. However, since the `pm4py.read.read_xes()` function reads the XES file into a pandas data frame, pandas could be used to merge multiple files using `pd.concat`.

Lastly, to focus more on the components of the software architecture, the sources and endpoints will be converted to activity names. For this, the source and endpoint combination of the loaded data frame will be extracted

and combined to chains. As an example, if source $\{A\}$ has endpoint $\{B\}$, and in another row $\{B\}$ is the source and $\{C\}$ the endpoint, then the chain $\{A, B, C\}$ will be formed. Then, for each value in the chain, a new row will be added to the data frame, where the value of source or endpoint will be assigned the value of the activity name. The source and endpoint columns will be removed and consecutive numbering for each chain will be generated as the case column. Once this is done, the data frame can be saved as a XES file again. This step is done to shift from a process event perspective to a software architecture perspective by looking at the connections of the software architecture components of the process.

5.3 Data Visualization

Once the data is available in XES format, it can be analysed. For this, the PM4Py library was used. To start with something simple, the start and end activities of the consumer log were determined, using the `pm4py.get_start_activities(xes_event_log)` and the `pm4py.get_end_activities(xes_event_log)` functions. As a result it gives the names of the start activity *experiment_configured* and the end activity *experiment_terminated*.

In a next step, a separate Directly Follows Graph (DFG) of the consumer log and the fastapi log was generated, using the `pm4py.discover_dfg(log)` function. As seen in Figure 5.4 and Figure 5.5, the DFG contains a start and end point, the activities of the event log, the direction of the activities, and the amounts each activity was performed.

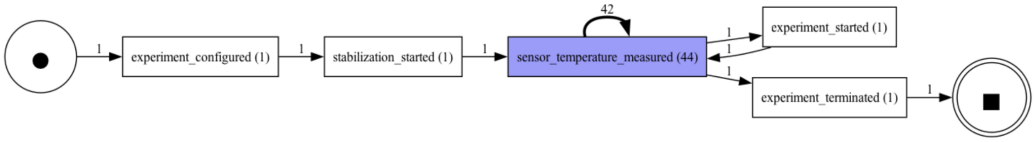


Figure 5.4. *Directly Follows Graph of the consumer log using PM4Py.*
consumer

After, a DFG was created for the combined XES log, which was created in Chapter 5.2. For this, the `ignore_index` needed to be set to `True`, in order to treat the dataset as one, and not as two (or multiple) separate ones. If this were not done, the `pm4py.discover_dfg(log)` function would treat it as different processes and generate separate DFGs like in Figure 5.4 and 5.5, simply in one picture.

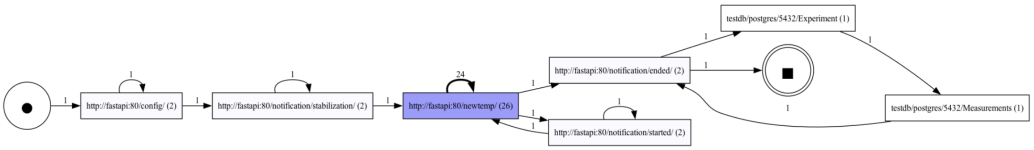


Figure 5.5. *Directly Follows Graph of the fastapi log using PM4Py.*

By setting the `ignore_index` to `True`, it combines the two individual DFGs and creates one DFG with one start and one end point, as seen in Figure 5.6.

Figure 5.6. *Directly Follows Graph of the fastapi log and the consumer log combined using PM4Py.*

The DFGs in Figure 5.4 and 5.5 show the process within each component, and the DFG in Figure 5.6 shows the complete process. However, while the naming of some activities gives some indication about the component, it does not distinguish (graphically) between the components involved, i.e., the consumer, the fastapi, and the database. As clear from the created DFGs, the graphs are created using the activity name of the datasets. In order to therefore create a DFG using the names of the components used, i.e., source and endpoint, the data needs to be changed accordingly.

As explained in Chapter 5.2, the source and endpoint of the combined data frame were converted into activity names. This is required to be able to create a DFG of the components. Figure 5.7 shows the DFG which is a result of the newly generated data frame. It can be seen that it represents the structure of the software architecture of the running example.

5.4 Case Study

The above explained steps will be applied to the datasets received from Bol, to see whether they can be applied to bigger and more complex data from real-life scenarios. By applying the steps to the trace dataset, it will also be evaluated whether the explained process can be applied to tracing data. The overall intention is to see whether the described process can be

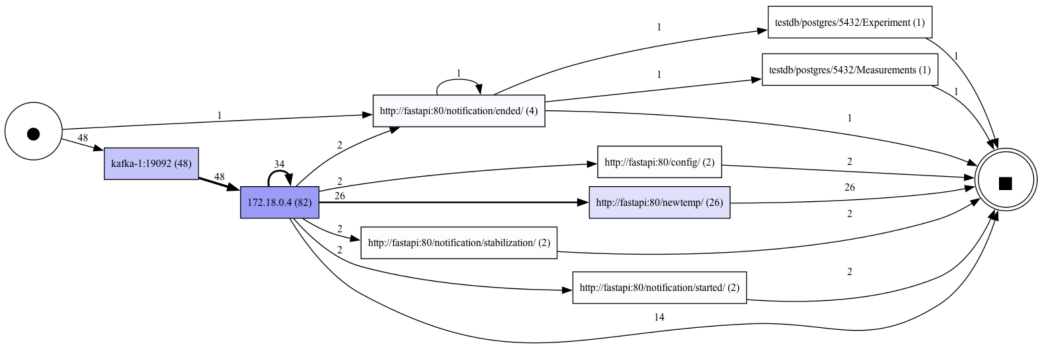


Figure 5.7. *DFG with components as activity names.*

applied to real-life datasets and what their results will be.

5.4.1 Connection Dataset

The connection dataset was explained in Chapter 2.4 as a dataset containing structural information about the software architecture at Bol.

```
<trace>
    <string key="concept:name" value="1" />
    <event>
        <string key="opex" value="team42dt" />
        <boolean key="isFinancial" value="false" />
        <string key="concept:name" value="nps" />
        <string key="gcpProjectId" value="bolcom-pro-nps-dc3" />
        <string key="piiLevel" value="2" />
        <string key="lifecycle:transition" value="complete" />
        <string key="connectionType" value="Synchronous" />
        <string key="depends_on_shortname" value="avatar" />
        <float key="productAreaShortName" value="nan" />
        <float key="productAreaName" value="nan" />
        <float key="productDomainShortName" value="nan" />
        <float key="productDomainName" value="nan" />
        <float key="productGroupShortName" value="nan" />
    </event>
    <event>
        <string key="opex" value="team42dt" />
        <boolean key="isFinancial" value="false" />
        <string key="concept:name" value="avatar" />
        <string key="gcpProjectId" value="bolcom-pro-nps-dc3" />
        <string key="piiLevel" value="2" />
        <string key="lifecycle:transition" value="complete" />
        <string key="connectionType" value="Synchronous" />
        <string key="depends_on_shortname" value="avatar" />
        <float key="productAreaShortName" value="nan" />
        <float key="productAreaName" value="nan" />
        <float key="productDomainShortName" value="nan" />
        <float key="productDomainName" value="nan" />
        <float key="productGroupShortName" value="nan" />
    </event>
    <event>
        <string key="opex" value="teamaccord" />
        <boolean key="isFinancial" value="false" />
        <string key="concept:name" value="avatar" />
        <string key="gcpProjectId" value="bolcom-pro-avatar-d55" />
        <string key="piiLevel" value="1" />
        <string key="lifecycle:transition" value="complete" />
        <string key="connectionType" value="Synchronous" />
        <string key="depends_on_shortname" value="cmt" />
        <string key="productAreaShortName" value="customer-service" />
        <string key="productAreaName" value="Customer Service" />
        <string key="productDomainShortName" value="foundation" />
        <string key="productDomainName" value="Foundation Products" />
        <string key="productGroupShortName" value="service_by_people" />
    </event>
```

Figure 5.8. *Excerpt of the transformed XES file of the Connection dataset.*

Data Requirements

First, the columns and the data of the connection dataset mentioned in Chapter 2.4 are compared with the data requirements in Chapter 5.1. The *opex* values can be used as a case ID, the *application_shortname* and *depend_application_shortname* are component identifiers. Both activity name and timestamp are missing from the dataset. This was to be expected, as the connection dataset isn't technically an event log containing information about processes, but rather about structural connections.

However, in Chapter 5.1 the columns source and endpoint, which are component identifiers, were transformed into activity names. This was done to focus on the components of the software architecture of the

running example by extracting the connections of the components and turning them into activity names in order to be able to visualize them using DFGs. This logic can be applied to the connections dataset as well. Therefore, the only data requirement that is missing is the timestamp.

Data Transformation

The datasets have been provided in CSV format and, in a first step, converted to XES using the *Convert CSV to XES* package in ProM. The *productGroupName* was selected as case column as it can be used as an unique identifier for its events, and the *application_shortname* as the event column as it indicates a specific action within the process.

In a next step, the XES file was read into a data frame using PM4Py, in order to convert the *application_shortname* and *depend_application_shortname* into activity names. An excerpt of the result can be seen in Figure 5.8. The file has a total of 962,203 lines of data and consists of 20,002 traces.

Data Visualization

The newly generated XES file was used to generate a DFG using PM4Py. On a first attempt, the code had been running for over an hour and still hadn't produced a DFG. It was therefore decided to run the code with a smaller file, consisting of 5,199 lines of code and 42 traces, which generated a DFG seen in Figure 5.9 within a couple of seconds. Generating DFGs based on bigger files than the smaller file are possible, however, their DFGs become unreadable due to their size and the zoom in function not being sufficient enough. Similar results have been achieved using ProM.

Evaluation

Different generated DFGs were shown to a group of stakeholders. The amount of lines, a missing filtering option, and a missing visual distinction between teams and their respective applications made it difficult to derive information from the DFGs, according to the stakeholders.

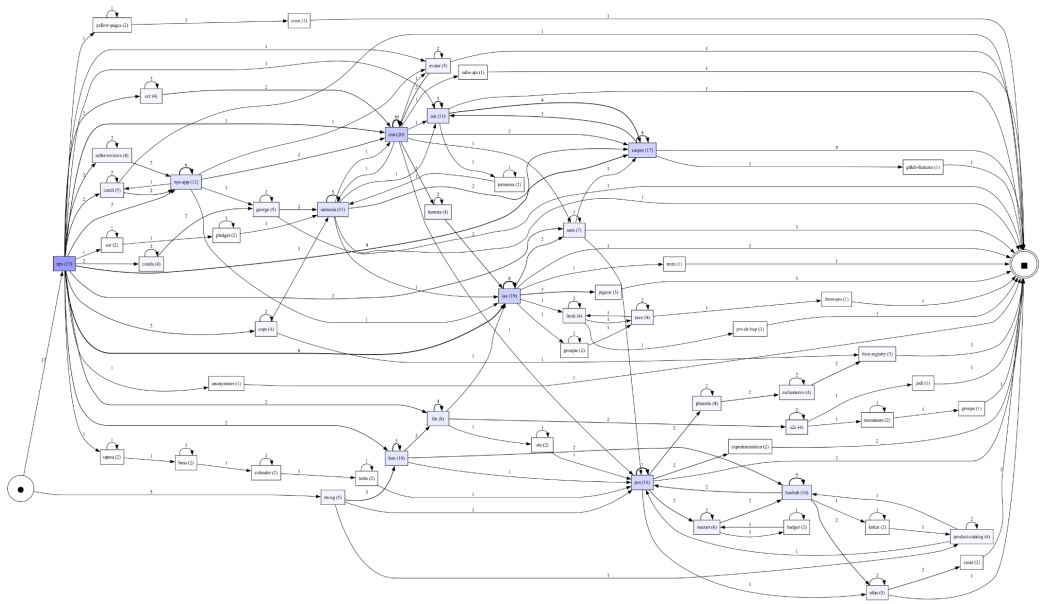


Figure 5.9. *DFG of a smaller Connection Dataset including 42 traces.*

5.4.2 Trace Dataset

The trace dataset includes data from dynamic tracing as mentioned in Chapter 2.4.

Data Requirements

Comparing the columns of the trace dataset with the data requirements in Chapter 5.1 shows that the dataset fulfills all requirements. The column *traceId* is the case ID, *http_url* is the activity name, *istio_namespace* is the component identifier, and *startTime* is the timestamp. If the connection between the applications want to be analyzed instead of the connection between the http URLs, then the *istio_namespace* can be converted into activity names similar as done with the connection dataset.

Data Transformation

The trace dataset has been provided in CSV format as well, and has been converted to XES using the *Convert CSV to XES* package in ProM. The *traceId* was selected as the case column and the *http_url* as the event column. The generated XES file can directly be used to generate a DFG with the *http_url* as activity names. However, Bol was more interested in

the connection between the different applications, instead of the http URLs. Therefore, the XES file had to be changed accordingly.

Figure 5.10. *Excerpt of the final trace XES file using application names as activity names.*

The originally generated XES file is read into a data frame using PM4Py, in order to extract the values for *spanId*, *parentSpanId*, *istio_namespace*, *startTime*, and *traceId*.
Sequences are generated using the *spanId* and the *parentSpanId*. For this, the combinations of *spanId* and *parentSpanId* are derived from each row, including their application name (istio_namespace). Once all combinations are extracted, the sequences are build by taking the first combination and extending it by the combination where the *parentSpanId* matches the *spanId*. Figure 5.11 shows an example for this process.

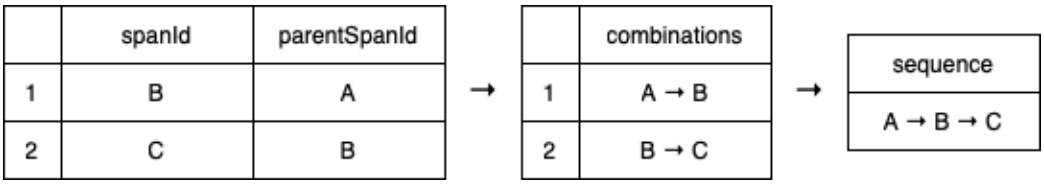


Figure 5.11. *Example of the sequence creation for the trace dataset.*

Once the sequences are created, the values of the *spanId* and the *parentSpanId* are replaced by their respective application value. Since the *istio_namespace* of a row always refers to the *spanId* of the same row, there are cases where the *parentSpanId* does not have a *istio_namespace* value. In these cases the application value is replaced with *NaN*. The final data frame is then converted back into a XES file using PM4Py, with the traceId being the case ID and the application name being the activity name. Figure 5.10 shows an excerpt of the final XES file.

Data Visualization

The generated XES file was used to generate a DFG using PM4Py. Since the individual traces have been created separately from each other, there does not exist a connection between them. Therefore, it is not useful

to create a DFG containing more than one traceID. During the DFG creation of the connection dataset it was identified that PM4Py was not able to process larger datasets in a reasonable time. Since the trace dataset is larger than the connection dataset, combined with the fact that generating a DFG for multiple trace IDs is not logical, the DFG seen in Figure 5.12 was created using only one trace ID. The XES file for this trace ID consisted of 922 lines.

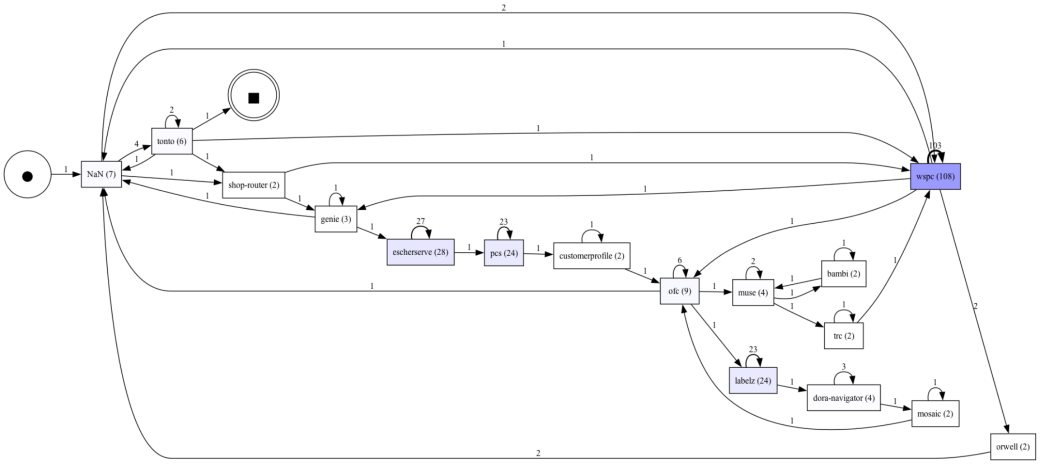


Figure 5.12. *DFG of one Trace ID of the trace dataset.*

Evaluation

Different generated DFGs were shown to a group of stakeholders. Since only one trace ID was shown at a time, the DFGs were more clear compared to the DFGs generated for the connection dataset. While a visual distinction between the team membership of applications was still missing, it was considered more of a "nice to have" instead of a "must have", as it was for the connection dataset. The possibility to load a larger dataset and then selectively choose a trace ID for visualization was requested.
Overall speaking, the DFGs for the traces was met with approval and interest by the stakeholders, as they were quickly able to derive information from it.

5.5 Conclusion

This Chapter addressed the topics of data processing in order to obtain meaningful information for software architecture visualizations. For this, data requirements and transformation for software execution data was discussed, and data visualizations for software architectures using process min-

ing were analyzed.

In regards of data requirements and data transformations, the assumptions made in the beginning were mostly confirmed by the case study. Data that provides information about the structure and interaction of a software architecture, like the connection dataset, might most likely not include an activity name. However, the data transformation and visualization showed that the names of the component identifiers can be used as activity names, indicating that activity names in the original data are not required. The reason for this is that they provide the same purpose in a DFG in process mining as an activity name would.
On the other hand, tracing data, like the trace dataset, tend to include an activity name. While their activity name can be used in a DFG in process mining like it is intended to, the names of component identifiers can also be used as activity names. This largely depends on the data available and the information that wants to be retrieved from it.

Furthermore did the missing timestamps in the connection dataset not cause any issues, as the order of applications could be recreated using chains. It can therefore be said, that data to visualize software architecture does not necessarily require an activity name or timestamps. However, it does require component identifiers, which can be transformed into activity names for process mining. Combining the two datasets could also be an option to use the timestamps of the trace dataset in the connection dataset. However, this is dependent on the datasets itself. Since the connection dataset contains information of connections that could be made if wanted, but not information about actual connections made like the trace dataset does, combining these two datasets would fall under the category of Architecture Conformance, which is out of the scope for this thesis.

Process mining tools can be used to visualize software architecture using DFGs up until a certain level. The visualizations are mainly limited by processing power, graphical options, and filtering. PM4Py and ProM both reached their limit when visualizing the complete connection dataset, containing about 120,000 lines of code. Commercial process mining tools like Celonis or Disco might be able to process such big files, but were not tested within this thesis.

While the DFG does show the connection between individual application, it cannot visually distinguish between groups of them, such as applica-

tions belonging to the same team or domain. Especially with large, complex visualizations, this feature could help with a quicker distinguishing of applications belonging together.

The DFGs are always structured in a block-like structure, with the start point on top (or left side) and the end point at the bottom (or right side) with the activity names arranged between them. This prevents the visualization from potentially showing the structure of the software architecture. The start and end point of the DFG can also be misleading, as a software architecture might not necessarily have a start and end point.

Additionally, both PM4Py and ProM lack essential filtering options. While PM4Py did not have any dynamic filtering options at all, ProM was not able to filter based on team name or domain name. Such filtering options would be beneficial for focusing on specific parts of the visualization, which is particularly necessary for large files, as seen in the case study. Viewing the complete visualization of large datasets can be overwhelming and hinder the extraction of useful information.
Another functionality missing is the view of different hierarchical views. Especially for datasets like the connection dataset, a switch between different hierarchies (i.e., from team to area to domain) could visualize more of the information that is already provided.
Furthermore became DFGs created with PM4Py unreadable from a certain size onwards.

The findings suggest that, while the data sources are already sufficient for visualizing software architecture, PM4Py and ProM, or process mining in general, are not ideal to visualize them due to the above named shortcomings. When it comes to trace data, however, the DFGs generated were able to provided sufficient visualizations within the case study.

The next Chapter will therefore explore other options of visualizations for software architecture. The focus here will lie on the capability to visualize larger datasets, filtering options, and visual distinguishing options.

Chapter 6

Testing Visualization Techniques for Software Architecture

Complex software systems with dynamic behavior can be difficult to understand. Visualizations can help identify bottlenecks, support quality assurance, or facilitate communication between stakeholders. This makes it crucial to understand how dynamic data can be utilized for visual representations. With different types of visualizations being available, it needs to be determined which ones are most suitable for visualizing software architecture.

The following Chapter will address topics regarding the purpose, and realization of visualizations in the context of software architecture.

6.1 Visualization Purpose

As mentioned in Chapter 3.3, different visualizations are used depending on the architecting activities, and the stakeholders addressed [20]. Therefore, it needs to be determined what purpose the architecting activity wants to achieve, and who the stakeholders are.

6.1.1 Architecting Activity

As mentioned in Chapter 1 and 4, this thesis aims to bridge the gap of missing, outdated, or incomplete software architectures. Furthermore, as mentioned in Chapter 4, the scope of this thesis is limited to the activities of *Architecture Reconstruction* and *Runtime Analyzer* of the Architecture

Mining Framework. Together with the general architecting activities defined in [20], mentioned in Chapter 3.3, the architecting activities that want to be achieved with the software architecture visualizations in this thesis are *Architecture Recovery*, and *Architecture Impact Analysis*.
Architecture Recovery is defined as uncovering architecture design based on existing implementation and documentation of the system. *Architecture Impact Analysis* identifies the architectural elements affected by a change scenario. The outcome helps architects to understand the dependencies between the changed parts and the affected parts of an architecture [20].

In the findings of the SLR in [14], the visualization technique most used to support *Architecture Recovery* is graph-based visualization, accounting for more than half of the chosen visualization techniques. For *Architecture Impact Analysis*, the most used visualization technique is notation-based visualization, occurring one time more than graph-based visualization.
To combine both architecting activities in one visualization, the graph-based visualization is therefore the most common visualization technique used to achieve this.

6.1.2 Stakeholders

Stakeholders are considered those who will use the visualization to obtain information from them in order to fulfill the above mentioned architecting activities. It can also be of interest to those who need to understand different aspects of software systems during the software development process as it can help to reduce the cost of software evolution [15]. Therefore, stakeholders are skilled and knowledgeable people within the domain of software architecture, or a related field, such as architects, developers, testers, and project managers [14].

6.2 Graph-Based Visualizations

From the D3.js gallery, which includes several examples, visualization examples from the subcategories *Hierarchies* and *Networks* were selected. The selection was made based on the data requirements for the D3.js visualizations and the data available of the case study. The focus here was to find visualizations that were able to reflect the connections between the components, and therefore the software architecture. Furthermore should the visualizations be able to reflect the structure of the software architecture in a way that lets stakeholders look at individual parts if needed. To incorpo-

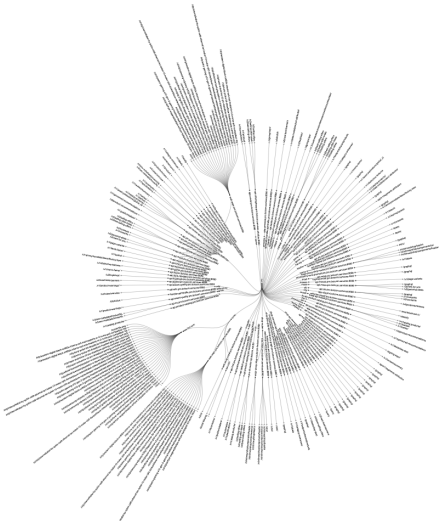
rate process mining techniques, the data structure used for the visualizations should be extendable accordingly and the visualizations themselves should be able to visualize process mining techniques. These visualizations were then shown to different stakeholders as part of a case study. By doing so, stakeholders could voice their opinions and give insights into whether the created visualizations gave them insights into the software architecture or not.

The visualization examples selected were:

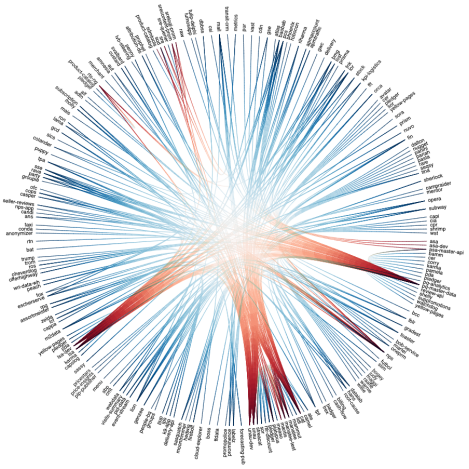
- Hierarchies: Tidy Tree, Radial Tidy Tree, Cluster Dendrogram, Radial Dendrogram
- Networks: Mobile Patent Suits, Arch Diagram, Hierarchical Edge Bundling

The Tidy Tree, the Cluster Dendrogram, and the Arch Diagram were discarded first from the options, as they display the data in a row like structure. This results in having to scroll down on a page to be able to see the complete data displayed, making it difficult to get an overall picture of the data visualized and the structure of the software architecture that comes with it. The Radial Dendrogram was discarded as well, since it does not display nodes on the same level, which could lead to a misinterpretation of the visualization and its data.

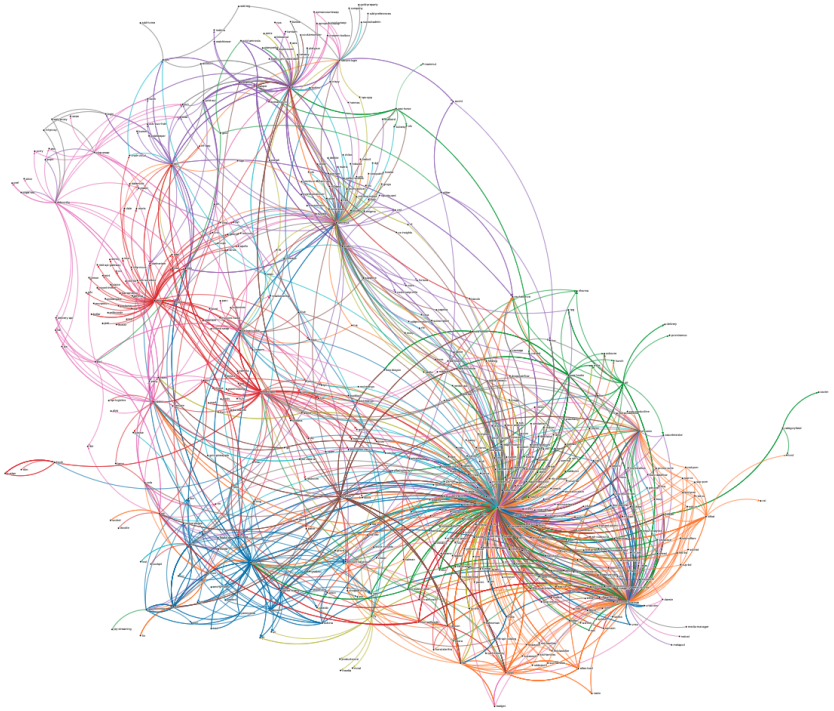
Figure 6.1 shows an overview of the final three visualizations which were selected. The Mobile Patent Suits and the Hierarchical Edge Bundling visualizations have been created using the connections dataset, while the Radial Tidy Tree visualization has been created using the trace dataset. Only the datasets from the case study have been used, as they are larger and more complex than the dataset form the running example, resulting in more realistic visualizations. However, for the first visualizations the dataset was shortened. Reason for this was that the complete dataset lead to crowded visualizations, which didn't allow for any valuable insights anymore. Since an option to filter the visualization was not yet available, the dataset had to be reduced to around 10% of its original size. This, however, did not have an influence on the original message of the visualization and was still larger than the dataset of the running example.



(a) Radial Tidy Tree Visualization.



(b) Hierarchical Edge Bundling Visualization.



(c) Mobile Patent Suits Visualization.

Figure 6.1. *Examples of the selected D3.js Visualizations.*

6.2.1 Mobile Patent Suits

The Mobile Patent Suits visualization shows the connection between different nodes by linking them together with arrows. Each arrow is represented in a different color, indicating the group the node belongs to.

Data Requirements

The Mobile Patent Suits visualization requires three different information as data input in a CSV file: source, target, and type. Source and target are the nodes, with the arrow of the link between them pointing to the target node. Type is the group each node belongs to, with different colours for the links used as a way of distinguishing between them.

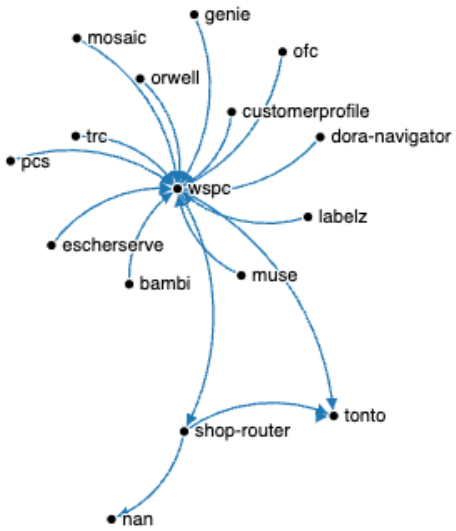


Figure 6.2. *Example of a trace visualized using the Mobile Patent Suits Visualization.*

To generate the data input for the Mobile Patent Suits visualization, the XES file of the connection dataset is converted into a CSV file. For this, for every event the *application_shortname* value is added as the source, the *depend_application_shortname* value is added as the target, and the *opex* value is added as the type. The connection dataset was not used for this visualization, as it is not suitable for this type of visualization. Reason for this is that the sequence within a trace is a key information, which cannot be visualized adequately, as seen in Figure 6.2.

Visualization

Figure 6.1c shows the Mobile Patent Suits visualization using the connection dataset. Every node represents an application, each color stands for a different team. The way the visualization creates the links between different nodes allows for a more unstructured visualization, allowing for a good overview of the structure of the software architecture.

Evaluation

The Mobile Patent Suits visualization was shown first during the stakeholder interview, as it gives a good overview of the connections between different applications. This was also something all stakeholders confirmed. While a good overview of the software architecture and its structure was something all stakeholders thought of as useful, they also all wanted it to display or contain more information. The requested information was usually about the individual nodes itself, wanting to add metrics or descriptions. Another frequent request was to be able to select a node and get to a different hierarchical level from there. This was reasoned with stakeholders wanting to use this visualization for a specific use case and diving deeper into its connections by switching to a different hierarchy.
While IT Architects can see themselves using this visualization for specific use cases, managers tend to also want to use it for exploration.
The coloring of the links based on team affiliation was perceived as positive and useful, making it easier to grasp certain information from the visualization. Requests for a filtering option to be able to use the complete dataset, different hierarchical views, more descriptive information about the nodes, and a short description regarding the meaning of the directions of the arrows were made by all stakeholders.

6.2.2 Hierarchical Edge Bundling

The Hierarchical Edge Bundling visualization is used to show dependencies among various modules in a software system. It helps in understanding how different parts of the system are interconnected, which modules are critical based on their size, and which modules have the most dependencies.

Data Requirements

The visualization uses a JSON file as data input. As seen in Figure 6.3, it consists of objects containing the properties *name*, *size*, and *imports*. The *name* includes the name and the hierarchical path of the object. In the example shown it is structured as *organization.team.application*, where application is the name of the object, while organization and team are the hierarchical path. The *size* represents the amount of imports of this object, while the *imports* lists the names of the objects which are imported to the related object.

```
[
    {
        "name": "bol.team42dt.avatar",
        "size": 2,
        "imports": [
            "bol.team42dt.nps"
        ]
    },
    {
        "name": "bol.team42dt.nps",
        "size": 8,
        "imports": [
            "bol.teamcustomeranalytics.cat",
            "bol.teamasa.pq-master-data",
            "bol.teamcustomeranalytics.unau-dev",
            "bol.teamcustomeranalytics.unau"
        ]
    },
    {
        "name": "bol.team42dt.cer",
        "size": 8,
        "imports": [
            "bol.team42dt.nps",
            "bol.teamasa.pq-master-data"
        ]
    },
```


Figure 6.3. *Example of the data structure for the Hierarchical Edge Bundling visualization.*

To generate the required JSON file, the XES file of the connection dataset is read into a data frame using PM4Py. For each row in the data frame, the values *opex*, *application_shortname*, and *depend_application_shortname* are extracted in order to construct the *names* in the format of *bol.opex_value.- application_shortname* and *bol.opex_value.- depend_application_shortname*.
The *application_shortname* is added to the *imports* of the *depend_application_shortname* if they appear in the same row in the data frame. Additionally, a count is maintained for the imports of each *depend_application_shortname*.
Since each *application_shortname* needs to be an object as well in order to generate the links between *application_shortname* and *depend_application_shortname* in the visualization, it needs to be initialized if necessary.

Visualization

The Hierarchical Edge Bundling visualization displays the data in a circle, as seen in Figure 6.1b. This allows for a good overview of the software architecture and its interconnections. Each name around the circle represents an application, grouped together by their teams. Small spaces visually separate the teams from another. The connections between the applications is represented by blue and red lines, where blue represents the source it is coming from and red the target it is going to.

Evaluation

The Hierarchical Edge Bundling visualization uses the connection dataset like the Mobile Patent Suits visualization. Therefore, stakeholders pointed out the similarity of the overall information shown and compared the two visualizations with each other. While both, the Hierarchical Edge Bundling

and the Mobile Patent Suits visualization, show connections between applications, the Hierarchical Edge Bundling focuses more on the overview of the data flow. The two different colors and the thickness of lines allows for a quick identification of hot spots and general traffic between applications and teams, which all interviewees liked. While the overall visualization and its perceived information was liked by all stakeholders, they all preferred the Mobile Patent Suits visualization over the Hierarchical Edge Bundling visualization. When asked for the reason, stakeholders named structural representation of the software architecture and personal needs for information. Some voiced the idea to incorporate the thickness of lines into the Mobile Patent Suits visualization, as this was the feature most favored about the Hierarchical Edge Bundling visualization.

6.2.3 Radial Tidy Tree

```
{
    "name": "bol",
    "children": [
    {
        "name": "zli",
        "children": [
            {
                "name": "service1",
                "children": [
                    {
                        "name": "path1"
                    },
                    {
                        "name": "path2"
                    },
                    {
                        "name": "path3"
                    },
                    {
                        "name": "path4"
                    },
                    {
                        "name": "path5"
                    },
                    {
                        "name": "path6"
                    }
                ]
            }
        ]
    },
```

Figure 6.4. *Example of the data structure for the Radial Tidy Tree visualization.*

The Radial Tidy Tree visualization constructs hierarchical node-link diagrams, placing nodes of the same hierarchical level on the same level in the visualization.

Data Requirements

The Radial Tidy Tree uses a JSON file as data input, with a data structure that represents a hierarchical organization of nodes, where each node has a name and may have a list of child nodes. As seen in Figure 6.4, the file begins with a root node, which in the case of this example is the company name. This root node has a name and multiple children. The child nodes represent the first hierarchy level, which in this example is the application. Each child node can have none, one, or multiple child nodes itself, representing different hierarchical levels. In the shown example, the second hierarchical level consists of the services of each application. The third, and last, hierar-

chical level represents the path of the service.

In order to create the required JSON file, the *http_url* values of the trace dataset had to be split into *Scheme*, *Netloc*, *Path*, and *Params* in order to create a hierarchical structure. The new XES file was then read into a data frame using PM4Py, and the values for *istio_namespace*, *Netloc*, and *Path* were extracted. To create the hierarchical structure, the *istio_namespace* is added to a list if it isn't already in there. It then checks if *Netloc* already exists under the current namespace. If not, it creates a new netloc entry. The same is done for the *Path* under the current *Netloc* entry.

Visualization

Just like the Hierarchical Edge Bundling visualization, the Radial Tidy Tree visualization displays the data in a circle, as seen in Figure 6.1a. By arranging the nodes like this, the visualization allows for a better overview than compared to a list like arrangement. The root node is centered in the middle, with the nodes of each hierarchy level visualized on a different level around it.

Evaluation

The Radial Tidy Tree caused the most controversy between stakeholders. While very few liked the depth of the visualization, the remaining stakeholders could not get any use out of it. Those who did not like the visualization found it too detailed, and couldn't see a use case for it. Those who did like the visualization appreciated the depth and detail it visualized.
The reason for the two different opinions mainly could be find in the position of the stakeholders, and therefore the work they do. Managers and Tech leads liked the visualization as it gave them insights into the software architecture, which they stated reflected the quality of outcome of certain decisions that had been made. An example was that the decision had been made to bundle APIs within the company. However, the visualization showed that this had not been realized everywhere and could be improved in other cases. The visualization also gave them inspiration of new topics to look into that would improve the overall software architecture within the company. Architects, on the other hand, had little to no interest in the grain of detail, as it didn't contribute to the decisions they had to make.

6.3 Conclusion

This Chapter aimed to find visualizations most suitable for visualizing software architecture. To be able to answer this question, the purpose of the visualization has been identified. Furthermore have different visualizations been created and evaluated by stakeholders.

The visualization purpose depends on the architecting activity that wants to be achieved, and the stakeholders requirements for the visualizations. Within the scope of this thesis, the *Architecture Recovery* and *Architecture Impact Analysis* are the identified architecting activities that want to be achieved with the software architecture visualizations. People within the domain of software architecture, or a related field, were identified as the stakeholders.

The evaluation of the generated visualizations reveled that all interviewed stakeholders preferred the Mobile Patent Suits visualization. The reasons for this are a quick overview of connections between applications, the distinguishing between teams, and identification of the structure of the software architecture. Additionally, this was the visualization stakeholders thought would most answer questions in regards of decisions that had to be made for their work. Furthermore is the dataset used for this visualization easily extendable, allowing to enrich the visualization with more information. The node and link structure also reminds of a DFG, which means similar graphical representations of information (e.g., thickness or color of the lines based on occurrence or time) could be incorporated into it.

However, while the explored visualizations were capable of loading the complete datasets of the case study, they lack capabilities to visualize large datasets in a sufficient manner. This meant that only part of the datasets could be loaded and visualized at a time, defeating the point of visualizing a complete software architecture. Additionally were hierarchical structures, which are given in both datasets, not always able to be visualized in a desired manner. Furthermore did the explored visualizations not include any metrics that could be used for an *Architecture Impact Analysis*.

These findings suggest that, while visualizations like the Mobile Patent Suits spark interest among stakeholder, the information they convey is still limited. The above mentioned shortcomings highlight the need for more advanced and flexible visualization tools in the field of software architecture.

The tested visualisations, while helpful, fall short in handling large datasets and accurately representing hierarchical structures. To address these limitations, future tools should incorporate the ability to visualize complete datasets and integrate relevant metrics for activities such as *Architecture Impact Analysis*. Additionally, the tools must be adaptable to the specific needs of different architecting activities and stakeholder requirements. By enhancing these aspects, visualization tools can provide more comprehensive and actionable insights.

The next Chapter will therefore address the above mentioned shortcomings within the visualizations, in order to improve them and be able to derive more information from them.

Chapter 7

Software Architecture Visualization

As concluded in Section 6, the Mobile Patent Suits visualization provides a good base structure for the visualization of software architecture with datasets like the connection dataset. However, it is missing certain attributes and functionalities to sufficiently visualize software architecture and to be able to derive useful information from it. Therefore, the following Chapter will explain and discuss the attributes and functionalities that were added to the Mobile Patent Suits visualization to achieve the desired goals.
Since the main feature of the Mobile Patent Suits Visualization is to show interactions between different nodes, the visualization will be called Interaction Network Visualization from now on.

7.1 Data Extension

The original data input file of the Interaction Network Visualization only included information about source, target, and type. In order to be able to derive more information from the final visualization, more information needs to be made available in the data input file. Therefore, the information about Group, Area, and Domain from the connection dataset were added. This allows to view the software architecture from different hierarchies, while also providing information for potential calculation of metrics. Furthermore has a description of the applications been added, which could be derived from an additional file.

7.2 Data Filtering and Graphical Representation of Group Distinctions

As previously mentioned, the ability to effectively visualize large datasets is currently insufficiently handled by the Mobile Patent Suits Visualization. Large datasets can also overwhelm the end-user's ability to interpret the visualizations. Therefore, incorporating filtering options into data visualization tools is essential. In order to achieve this, a checkbox selection for the *type* values has been added. Figure 7.1 shows an example of the checkbox selection of the connection dataset, where the team names were used.

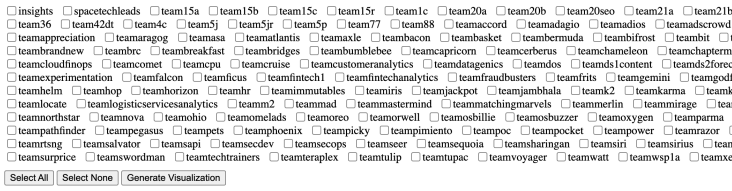


Figure 7.1. *Checkbox filtering for the team names.*

The data is filtered based on the value in the *type* column. This means, that by selecting a *type*, the visualization generated will include nodes of all the values in the *source* column and their respective values in the *target* column, that match the selected value in the *type* column. The links between the nodes will be included in the visualization as well, with the arrow starting at the node of the *source* and pointing to the node of the *target*. For the connection dataset, this means the visualization will show all calls that are made within the selected team, but also calls made from the selected team to applications from other teams, as well as calls made from other teams to applications of the selected team.

In order to include calls made from application of other teams to applications of the selected team, the filtering logic needs to be extended. For this, the values of the *source* where the *type* value matches the selected *type* are temporarily put into a separate list. The values from this list are then compared with the entries from the *target* column. If there is a match, value of the *source* will be added as a node to the visualization, with a link to the *target* value, which should already be in the visualization. This ensures that the final visualization will not only include nodes and links of the selected team and the applications it makes calls to, but also nodes of other teams that make a selection to the selected team. This generates a more complete

view of the interaction between teams and applications, as desired by the interviewed stakeholders.

As previously mentioned, the visualization includes nodes representing different types, i.e., teams, areas, groups, or domains. To facilitate easy differentiation between the types, the links associated with each type are color-coded uniquely, as seen in Figure 7.2 for the connection dataset.

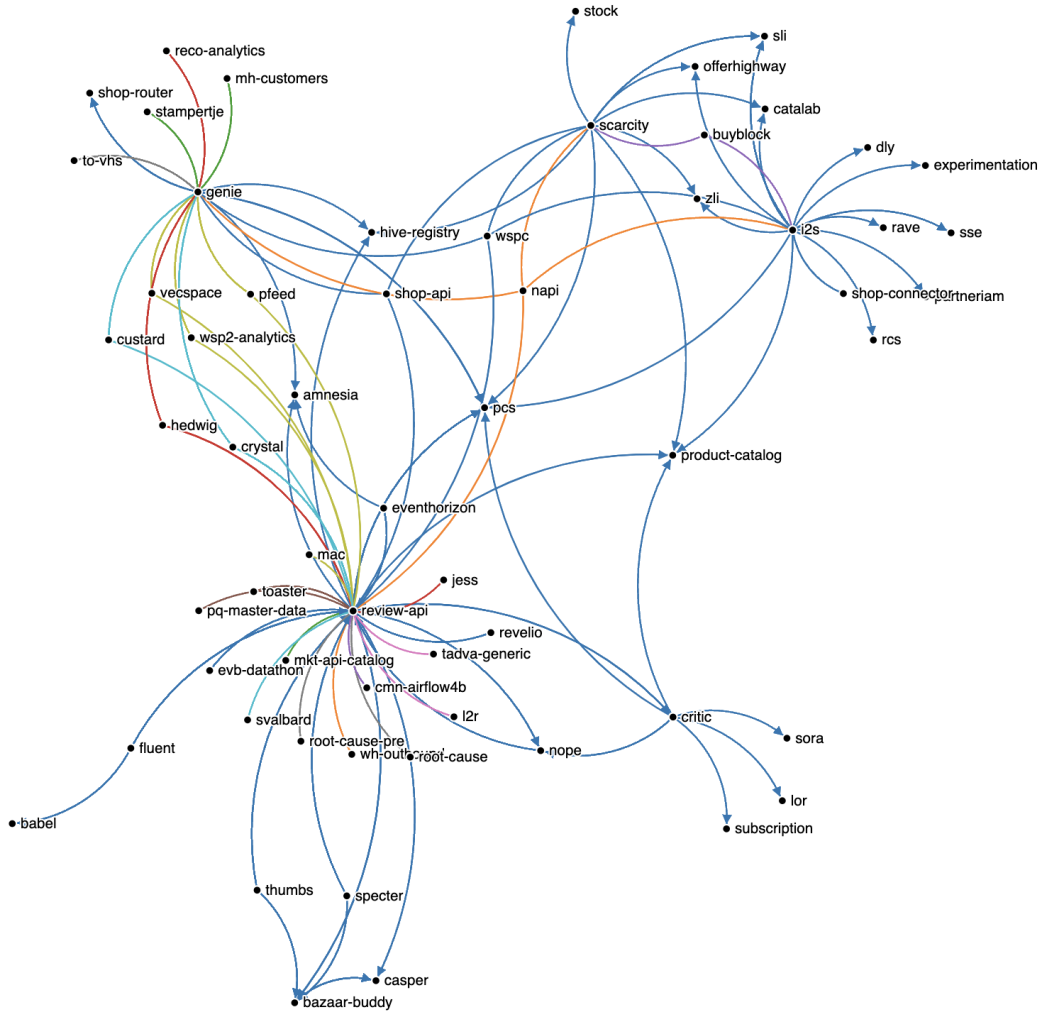


Figure 7.2. *Team phoenix and its nodes, with team phoenix represented by blue links.*

7.3 Hierarchical Views

The example explained in Section 7.2 used team names for filtering, as it represents the highest hierarchy available in the dataset. However, the same logic can be applied to different hierarchical levels, such as Group, Area, and Domain. To achieve this, the *application_shortname* values and their respective *opex* values are stored as pairs in a dictionary. The values from the *application_shortname* column are then compared with the *application_shortname* values in the dictionary, and if a match is found, they are replaced by their corresponding *opex* values. This process is repeated for the *depend_application_shortname* column. The result of this visualization can be seen in Figure 7.3, where it is filtered for groups and teams are represented as nodes.
This process can be repeated for all other hierarchies as well.

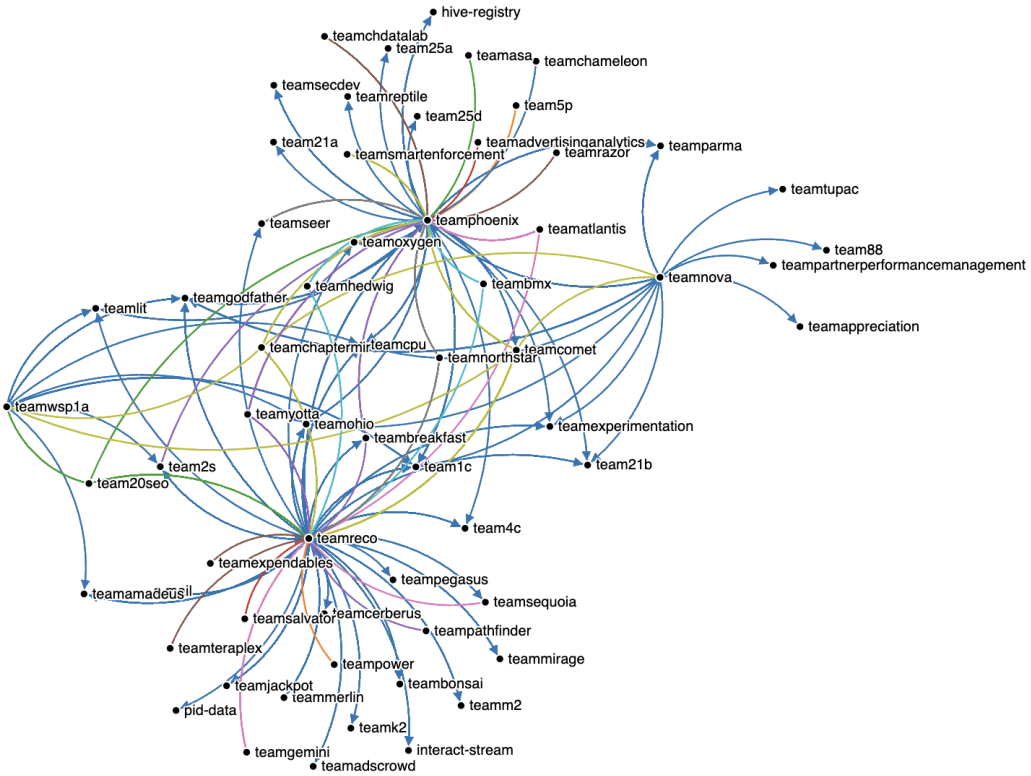


Figure 7.3. *Group "Decide to Buy" and its respective teams represented as nodes with blue links.*

7.4 Metrics and Information

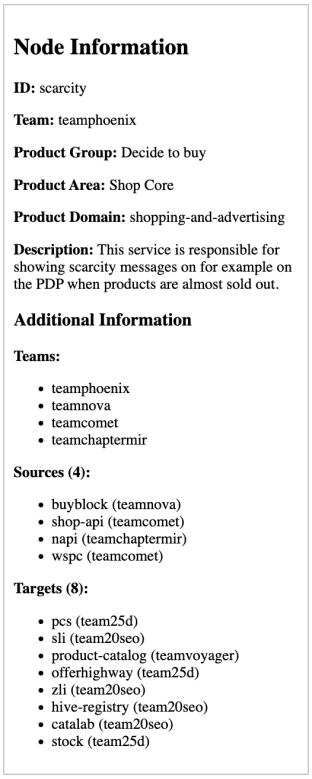


Figure 7.4. *Example of the information box for application scarcity.*

Above the visualization, metrics about the selected team(s) were displayed. These metrics included the count of distinct teams, areas, groups, and domains with which the team(s) interact. During interviews, stakeholders highlighted that this information is particularly valuable for them when making changes to the software architecture, as it helped them to estimate a potential impact. It also gives them insights into the cross-dependency of teams, which they stated that they want to reduce due to risk and cost aspects. They also stated that this task currently had to be done manually and therefore was quite time-consuming.

To enhance the visualization and provide more details about each node, an interactive information box was added, as seen in Figure 7.4. When a node is clicked, this box appears, displaying additional information such as the team, group, area, and domain to which the node belongs. It also includes a description of the node, if available, along with information about the teams and applications that the node interacts with, either by making calls to or receiving calls from them.

7.5 Evaluation

The final visualizations were presented to the stakeholders, who expressed great enthusiasm and appreciation for the work. They were particularly impressed with the filtering capabilities, the graphical distinction between groups, and the metrics provided. The filtering allows the stakeholders to to get a quick overview of the applications within their own team, but also the connections that are made or coming from outside the team. This is also applicable to different hierarchical views and, together with the color distinction between groups, helps stakeholders to quickly estimate the extend of a potential change. Since this task currently has to be done manu-

ally by talking to different team members or looking through different textual documentations, the visualization drastically reduces the time consumption needed for this task. The addition of an information box about each node, displayed on the side, was also well-received, as the list of names of the interacting nodes provides a more detailed information of the visualization.
The stakeholders were pleased with the differences compared to the earlier visualizations they were shown, noting that the final version was much more helpful and allowed them to derive more useful information. They identified several practical use cases for the visualizations and suggested additional metrics and features that could enhance their utility further. Additional features mentioned were the size of the message flows within calls, the count of the calls made, and the comparison of calls allowed to make and calls actually made. Stakeholders also mentioned additional datasets that they thought of which could be added to the visualization in order to enrich it and improve their work even more.
Overall, the feedback was rated positive and successful, as the visualization provided an automated documentation of the software architecture which improved the work of the stakeholders by providing new insights or reducing the time needed for tasks that currently are done manually.

7.6 Conclusion

This Chapter focused on addressing the attributes and functionalities identified as missing in the previous Chapter, which explored visualizations for software architectures. The goal of this Chapter is to experiment with the capabilities of the Interaction Network Visualization, aiming to sufficiently represent software architecture and extract meaningful information from it.

Various functionalities were explored to determine their feasibility and contribution to the desired end product. As a result, features such as filtering, graphical representation of group distinctions, hierarchies, metrics, and additional information were incorporated into the Interaction Network Visualization to enhance its effectiveness. The final visualization was then evaluated by stakeholders, who expressed satisfaction with the results. They also provided suggestions for further enhancements to address additional use cases and improve the visualization even more.
Process mining techniques were not implemented due to the complexity of combining the PM4Py library with JavaScript. However, such an implementation would increase the possibilities to analyze the data available, such as throughput times, occurrences of connections, most used connections, and so

on. The connection dataset already provides a good amount of data to apply process mining techniques to it, but replacing it with a dynamic dataset to add the currently missing timestamps would be more beneficial.

One limitation to consider is that the visualizations are influenced by the structure of the dataset used. While this thesis aimed to create universally applicable results, it cannot be denied that the specific structure of the provided dataset had an impact on the outcomes. Consequently, this limitation implies that the conclusions may not be directly transferable to other data structures or sets.

In conclusion, this Chapter successfully demonstrated the potential for enhancing data visualizations through thoughtful integration of advanced functionalities, paving the way for future improvements and applications. Future work could focus on integrating these suggestions to further refine the visualization and expand its applicability.

Chapter 8

Conclusion, Limitations, Discussion, and Future Work

This thesis explored the question of how software architecture can be visualized using dynamic data. To address this, literature reviews and experiments were conducted to determine the necessary data requirements. Subsequently, various visualizations were evaluated by stakeholders to identify the most effective approach for representing software architecture.

This Chapter aims to draw conclusions from the project by providing concise answers to the posed research questions, acknowledge the limitations encountered, and suggest potential directions for future work to further advance the field of software architecture visualization.

8.1 Conclusion

In order to derive conclusions surrounding the research question, the sub-questions that were posed are addressed first.

`SQ1: Which event data is relevant for visualizing software architecture?`

With the first sub-question, the topic of what kind of data is needed for effectively visualizing software architecture was addressed in Chapter 5. To answer this question, experiments were conducted with the available data to determine what kinds of graphs could be generated using process mining, and what specific information was necessary to achieve useful results. Given that

process mining was a key aspect of this thesis, it was evident that the data requirements for process mining should be integrated into the overall data requirements. By converting component information into activity names, Directed Flow Graphs (DFGs) were successfully generated to visualize both software architecture and trace data. The results demonstrated that having activity names in the original data is not mandatory, as component identifiers can be converted into activity names. This implies that detailed information about the components is essential for visualizing software architecture.

To determine the sequence of each activity or component, either a timestamp or the order of the components (i.e., source and endpoint information) is necessary. Typically, trace data includes a traceID, serving the role of a case ID in process mining. For software architecture component data, a case ID can be generated.

SQ2: Which visualization techniques are considered the most effective for representing software architecture?

To answer sub-question 2, literature was reviewed to identify the most effective visualization techniques for representing software architecture, based on the suitability of its visualization purposes, *Architecture Recovery* and *Architecture Impact Analysis*. Given that graph-based visualizations are most commonly used for these purposes, three different graph-based visualizations were selected for evaluation.

To assess their effectiveness, three example visualizations using the datasets provided within the case study were presented to stakeholders. During this process, stakeholders evaluated each visualization technique and provided feedback on aspects such as clarity, usefulness, and the ability to convey information. Among the visualizations presented, the Mobile Patent Suits visualization, which visualized structural data, was favored by stakeholders. They appreciated its dynamic nature and the comprehensive information it displayed, which aligned with their needs and expectations. Consequently, based on the findings from the case study, the Mobile Patent Suits visualization emerged as the most effective technique for visualizing software architecture in this context. Comparing the results from Chapter 5 and Chapter 6, it was also discovered that DFGs are the most suitable way of visualizing tracing data, as they are capable of accurately visualizing sequence data, while the other visualizations weren't.

SQ3: How can dynamic data from existing software architectures be used for visual representations to aid in understanding complex software systems?

To answer sub-question 3, the Mobile Patent Suits visualization, which was determined most effective for visualizing software architecture, was extended to create the Interaction Network Visualization. The Interaction Network Visualization included additional data attributes, filtering options, and color-coded links to distinguish between different types. Hierarchical views were implemented to allow for different levels of detail, and interactive information boxes provided detailed node-specific information. Through stakeholder feedback it was evaluated that these enhancements aid in understanding complex software systems by providing clearer insights and actionable metrics, highlighting the practical benefits and potential for further improvements.
Process mining techniques were not applied to the Interaction Network Visualization due to the complexity of incorporating the PM4Py library with JavaScript, which resulted in a decreased availability of data analyzing functionalities.

RQ: How can we visualize software architecture using dynamic data?

By combining the insights gained from the previous research questions, the main research question can be answered: how to (best) visualize software architecture depends on the data. When employing process mining techniques for visualization, it is essential to transform the available data appropriately. This involves converting each component into an activity name and accurately representing the sequence of connections. Directed Flow Graphs (DFGs) generated through process mining can provide initial insights into smaller datasets or less complex software architectures, and are especially useful for tracing data. Furthermore can process mining techniques easily be applied to DFGs due to the inherent structure and properties of them that align well with the goals and methods of process mining. For more comprehensive analysis and detailed information, graph-based visualizations are more suitable. This is especially applicable for dynamic data that provides structural information of a software architecture. These visualizations excel in dynamically representing each component as a node with links indicating their connections, offering a clearer and more detailed depiction of the software architecture. Process mining techniques are more complex to apply to them, as there currently is no ready-to-use solution available.

8.2 Limitations

One significant limitation of this study is the use of datasets from only one company. This singular focus means that the data structure and visualizations observed may not be representative of other organizations. As a result, the findings and conclusions drawn from these datasets may not be universally applicable. This is especially true for the connection dataset, as different companies may have different structures and patterns, which could lead to varying results when the same visualizations and analyses are applied. Therefore, caution should be exercised when generalizing these results to other contexts, and further research involving diverse datasets from multiple companies is recommended to validate these findings.

8.3 Discussion and Future Work

The motivation behind this project was to visualize software architecture using dynamic data, in order to be able to derive information from it. Process mining played a central aspect in the processing of the data and the creation of the visualizations in this. As the experiments and results in this thesis have shown, there are two different outcomes based on the two different datasets that were analyzed within the case study.
The connection dataset contains structural information about the overall software architecture. However, process mining techniques are not yet suitable for visualizing this type of data. The primary limitations include the inability to effectively represent different hierarchical levels, challenges in processing large datasets, and the lack of visual distinction between groups or components. These shortcomings hinder the ability to gain clear insights into the complex structure of the software architecture using current process mining visualization methods.
Conversely, the trace dataset containing data gathered from dynamic tracing proved to be more suitable for process mining. This dataset enabled the quick generation of useful Directed Flow Graphs (DFGs), which stakeholders found very interesting. This area presents significant opportunities for future work, as there is difference of definition between traces in the context of software architecture or company data are, and traces in process mining. Additionally, the potential use cases of trace data within companies warrant further research. Improving process mining for trace data is another promising topic. While current methods are quite effective at deriving information, there is room for experimentation with features such as loading large datasets and selectively displaying individual traces, comparing traces with

each other, and more. These enhancements could further refine the process mining capabilities and provide deeper insights into software architecture.

Future work in the domain of software architecture visualizations using process mining should focus on enhancing the ability to visualize different hierarchical views and enable seamless switching between them. Currently, process mining techniques lack this capability, which limits their effectiveness in representing complex software architectures. Research in this direction could significantly improve the utility of process mining for software architecture data, providing more comprehensive and multi-layered insights into the structure and dynamics of software systems. Developing methods to better integrate hierarchical visualizations would facilitate a deeper understanding of architectural components and their interactions, ultimately leading to more robust and actionable analyses.

Another promising area for future work is enhancing process mining for tracing data. While visualizations for both the connection dataset and the trace dataset were deemed helpful, the DFGs generated from tracing data sparked greater interest and inspired more use cases among stakeholders. While the previously mentioned improvements would already benefit tracing data, it furthermore provides promising opportunities in all three types of process mining: discovery, conformance, and enhancement.
For discovery, tracing data can be rapidly implemented across the entire software architecture to uncover actual data flows and interactions between components. When happy flows are available, tracing data can be utilized for conformance checking to ensure the system operates as intended. Additionally, process mining could be used to identify and apply the right metrics to the discovered software architecture, providing suggestions for enhancements to improve the system. These advancements in process mining could significantly optimize the understanding and management of complex software systems.

Bibliography

[1] de Souza, S. C. B., Anquetil, N. & de Oliveira, K. M. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, 68–75 (2005).

[2] Van der Aalst, W. M. & Weijters, A. J. Process mining: a research agenda. *Computers in industry* **53**, 231–244 (2004).

[3] Liu, C., van Dongen, B. F., Assy, N. & van der Aalst, W. M. A general framework to identify software components from execution data. In *ENASE*, 234–241 (2019).

[4] van Dongen, B. F., de Medeiros, A. K. A., Verbeek, H. M. W., Weijters, A. J. M. M. & van der Aalst, W. M. P. The prom framework: A new era in process mining tool support. In Ciardo, G. & Darondeau, P. (eds.) *Applications and Theory of Petri Nets 2005*, 444–454 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2005).

[5] de Jong, T. *From Package to Process: dynamic software architecture reconstruction using process mining.* Master's thesis (2019).

[6] Hevner, A. R., March, S. T., Park, J. & Ram, S. Design science in information systems research. *MIS Q.* **28**, 75–105 (2004). URL `http://misq.org/design-science-in-information-systems-research.html`.

[7] bol.com. Our story (2024). URL `https://pers.bol.com/en/our-story/`.

[8] D3.js. What is d3? (2024). URL `https://d3js.org/what-is-d3`.

[9] Bass, L., Clements, P. & Kazman, R. *Software Architecture in Practice: Software Architect Practice_c3* (Addison-Wesley, 2012).

[10] Mattsson, M., Grahn, H. & Mårtensson, F. Software architecture evaluation methods for performance, maintainability, testability, and portability. In *Second International Conference on the Quality of Software Architectures*, 18 (2006).

[11] Rozanski, N. & Woods, E. *Software systems architecture: working with stakeholders using viewpoints and perspectives* (Addison-Wesley, 2012).

[12] Babar, M. A., Zhu, L. & Jeffery, R. A framework for classifying and comparing software architecture evaluation methods. In *2004 Australian Software Engineering Conference. Proceedings.*, 309–318 (IEEE, 2004).

[13] Patidar, A. & Suman, U. A survey on software architecture evaluation methods. In *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, 967–972 (IEEE, 2015).

[14] Shahin, M., Liang, P. & Babar, M. A. A systematic review of software architecture visualization techniques. *Journal of Systems and Software* **94**, 161–185 (2014).

[15] Diehl, S. *Software visualization: visualizing the structure, behaviour, and evolution of software* (Springer Science & Business Media, 2007).

[16] Beck, M., Trümper, J. & Döllner, J. A visual analysis and design tool for planning software reengineerings. In *2011 6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, 1–8 (IEEE, 2011).

[17] Zalewski, A., Kijas, S. & Sokołowska, D. Capturing architecture evolution with maps of architectural decisions 2.0. In *Software Architecture: 5th European Conference, ECSA 2011, Essen, Germany, September 13-16, 2011. Proceedings 5*, 83–96 (Springer, 2011).

[18] de Boer, R. C., Lago, P., Telea, A. & van Vliet, H. Ontology-driven visualization of architectural design decisions. In *2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*, 51–60 (IEEE, 2009).

[19] Panas, T., Epperly, T., Quinlan, D., Saebjornsen, A. & Vuduc, R. Communicating software architecture using a unified single-view visualization. In *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, 217–228 (IEEE, 2007).

[20] Li, Z., Liang, P. & Avgeriou, P. Application of knowledge-based approaches in software architecture: A systematic mapping study. *Information and Software technology* **55**, 777–794 (2013).

[21] Gregg, B. & Mauro, J. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD* (Prentice Hall Professional, 2011).

[22] Van Der Aalst, W. Process mining: Overview and opportunities. *ACM Transactions on Management Information Systems (TMIS)* **3**, 1–17 (2012).

[23] Van Der Aalst, W. *et al.* Process mining manifesto. In *Business Process Management Workshops: BPM 2011 International Workshops, Clermont-Ferrand, France, August 29, 2011, Revised Selected Papers, Part I 9*, 169–194 (Springer, 2012).

[24] Verbeek, H., Buijs, J., Van Dongen, B. & van der Aalst, W. M. Prom 6: The process mining toolkit. *Proc. of BPM Demonstration Track* **615**, 34–39 (2010).

[25] Verbeek, H., Buijs, J. C., Van Dongen, B. F. & Van Der Aalst, W. M. Xes, xesame, and prom 6. In *Information Systems Evolution: CAiSE Forum 2010, Hammamet, Tunisia, June 7-9, 2010, Selected Extended Papers 22*, 60–75 (Springer, 2011).

[26] Berti, A., Van Zelst, S. J. & van der Aalst, W. Process mining for python (pm4py): bridging the gap between process-and data science. *arXiv preprint arXiv:1905.06169* (2019).

[27] Günther, C. W. & Verbeek, E. Standard definition. *Fluxicon Process Laboratories, XES Version* **1** (2012).

[28] van der Werf, J. M. E. *et al.* Architectural intelligence: A framework and application to e-learning. In *RADAR+ EMISA@ CAiSE*, 95–102 (2017).

[29] de Jong, T. & van der Werf, J. M. E. Process-mining based dynamic software architecture reconstruction. In *Proceedings of the 13th European Conference on Software Architecture-Volume 2*, 217–224 (2019).

[30] Rooimans, R. *Architecture Mining with ArchitectureCity*. Master's thesis (2017).

Appendix A

Code Snippets

```
1  import pm4py
2
3  def import_xes(xes_file_path):
4      xes_event_log = pm4py.read_xes(xes_file_path)
5      start_activities = pm4py.get_start_activities(
       xes_event_log)
6      end_activities = pm4py.get_end_activities(xes_event_log)
7      print("Start activities: {}\nEnd activities: {}".format(
       start_activities, end_activities))
8
9  if __name__ == "__main__":
10     import_xes("/Users/Path/XES_event_log.xes")
```

Listing A.1. *Getting the Start and End Activity with PM4Py.*

```
1  import pm4py
2
3  if __name__ == "__main__":
4      # read xes file into dataframe
5      log = pm4py.read_xes(os.path.join("/Users/Path/
       XES_event_log.xes"))
6      # discover dfg
7      dfg, start_activities, end_activities = pm4py.
       discover_dfg(log)
8      # show dfg
9      pm4py.view_dfg(dfg, start_activities, end_activities)
```

Listing A.2. *Discover Directly Follows Graph with PM4Py*

```
1  import pm4py
2  import os
3  import pandas as pd
4
5  if __name__ == "__main__":
```

```
    # read first xes file into log (DataFrame)
    log1 = pm4py.convert_to_dataframe(pm4py.read_xes(os.path.
    join("/Users/Path/XES_event_log_one.xes")))

    # read second xes file into log (DataFrame)
    log2 = pm4py.convert_to_dataframe(pm4py.read_xes(os.path.
    join("/Users/Path/XES_event_log_two.xes")))

    # concatenate logs using pandas with continuous index
    combined_log_df = pd.concat([log1, log2], ignore_index=
    True)

    # convert the concatenated DataFrame back to a pm4py
    event log
    combined_log = pm4py.convert_to_event_log(combined_log_df
    )

    # discover dfg
    dfg, start_activities, end_activities = pm4py.
    discover_dfg(combined_log)

    # show dfg
    pm4py.view_dfg(dfg, start_activities, end_activities)
```

Listing A.3. *Getting the Start and End Activity with PM4Py.*

```
import pm4py
import os
import pandas as pd

# Read the XES file into a DataFrame
log = pm4py.convert_to_dataframe(pm4py.read_xes(os.path.join(
    "/Users/Path/XES_event_log.xes")))

# Initialize a list to store the new rows
new_rows = []

# Initialize a dictionary to track chains of events
chains = {}
case_concept_counter = 1

# Function to add a row to the list with a given
    case_concept_name
def add_row(row, case_concept_name, is_source=True):
    new_row = row.copy()
    new_row['concept:name'] = new_row['source'] if is_source
    else new_row['endpoint']
    new_row['case:concept:name'] = str(case_concept_name)
    new_rows.append(new_row)

```

```
# Process each row to form chains
for index, row in log.iterrows():
    source = row['source']
    endpoint = row['endpoint']

    # Check if the source is already in any chain
    found_chain = False
    for key in list(chains.keys()):
        if chains[key][-1] == source:
            # Append the endpoint to the existing chain
            chains[key].append(endpoint)
            found_chain = True
            # Add rows for source and endpoint
            add_row(row, key, is_source=True)
            add_row(row, key, is_source=False)
            break

    # If no existing chain was found, create a new chain
    if not found_chain:
        chains[case_concept_counter] = [source, endpoint]
        # Add rows for source and endpoint
        add_row(row, case_concept_counter, is_source=True)
        add_row(row, case_concept_counter, is_source=False)
        case_concept_counter += 1

# Convert the list of new rows to a DataFrame
transformed_log = pd.DataFrame(new_rows)

# Drop the 'source' and 'endpoint' columns
transformed_log.drop(columns=['source', 'endpoint'], inplace=
    True)

# Save the new dataframe to a XES file
pm4py.write_xes(transformed_log, "/Users/Path/
    XES_event_log_transformed", case_id_key='case:concept:name
    ')
```

Listing A.4. *Generating a new pandas data frame with source and endpoint values as activity names.*

```
import pm4py
import pandas as pd
import os
import numpy as np
from pm4py.objects.log.obj import EventLog, Trace, Event

# Read the XES file into a DataFrame
log = pm4py.convert_to_dataframe(pm4py.read_xes(os.path.join(
    "/Users/Path/XES_event_log.xes")))

```

```
# Function to build sequences from the chains
def build_sequences(chains):
    sequences = []
    while chains:
        current_chain = chains.pop(0)
        current_sequence = [current_chain[0], current_chain[1]]
        applications_sequence = [current_chain[2], current_chain[2]]
        timestamps_sequence = [current_chain[3], current_chain[3]]

        while True:
            found = False
            for i, (src, end, app, timestamp) in enumerate(chains):
                if current_sequence[-1] == src:
                    current_sequence.append(end)
                    applications_sequence.append(app)
                    timestamps_sequence.append(timestamp)
                    chains.pop(i)
                    found = True
                    break
            if not found:
                break

        sequences.append((current_sequence, applications_sequence, timestamps_sequence))
    return sequences

# Function to replace sequence values with application values or 'NaN' based on preceding values
def replace_with_application(sequences):
    # Find all unique sequence values that have preceding values
    preceding_values = set()
    for seq, apps, timestamps in sequences:
        for i in range(1, len(seq)):
            preceding_values.add(seq[i])

    replaced_sequences = []
    for seq, apps, timestamps in sequences:
        replaced_sequence = []
        for i in range(len(seq)):
            if i == 0 and seq[i] not in preceding_values:
                replaced_sequence.append(np.nan)
            else:
                replaced_sequence.append(apps[i] if apps[i] is not None else np.nan)
```

```
51          replaced_sequences.append((replaced_sequence,
    timestamps))
52
53      return replaced_sequences
54
55  # Initialize an empty list to store all sequences
56  all_sequences = []
57
58  # Group the log by 'case:concept:name' and process each group
59  for trace_id, group in log.groupby('case:concept:name'):
60      # Initialize a list to store unique (source, endpoint,
    application, timestamp) tuples for the current traceId
61      chains = []
62
63      # Process each row in the group to collect unique (source
    , endpoint, application, timestamp) tuples
64      for index, row in group.iterrows():
65          source = row['source']
66          endpoint = row['endpoint']
67          application = row['istio_namespace']
68          timestamp = row['time:timestamp']
69
70          if (source, endpoint, application, timestamp) not in
    chains:
71              chains.append((source, endpoint, application,
    timestamp))
72
73      # Build the sequences from the chains
74      sequences = build_sequences(chains)
75
76      # Replace sequence values with application values or 'NaN
    '
77      replaced_sequences = replace_with_application(sequences)
78
79      # Append the replaced sequences to the list of all
    sequences with the traceId
80      for seq, timestamps in replaced_sequences:
81          all_sequences.append({'traceId': trace_id, 'sequence'
    : seq, 'timestamps': timestamps})
82
83  # Convert the sequences to a DataFrame for better
    visualization
84  sequences_df = pd.DataFrame(all_sequences)
85
86  # Convert the sequences DataFrame back to XES format
87  event_log = EventLog()
88  for trace_id, group in sequences_df.groupby('traceId'):
89      trace = Trace()
90      trace.attributes["concept:name"] = trace_id
```

```
    for i, row in group.iterrows():
        for j, (activity, timestamp) in enumerate(zip(row['sequence'], row['timestamps'])):
            event = Event()
            event["concept:name"] = activity if not pd.isna(activity) else 'NaN'
            event["time:timestamp"] = timestamp
            # event["case:concept:name"] = trace_id
            trace.append(event)
    event_log.append(trace)

# Export the event log to an XES file
pm4py.write_xes(event_log, "/Users/Path/XES_event_log_activity_names.xes")
```

Listing A.5. *Generating a new XES file with application names as activity name for a trace.*

```
import os
import pm4py
import csv

# Directory containing XES files
xes_directory = "/Users/Path"

# Open a CSV file in write mode
with open('mobile_patent_suits.csv', 'w', newline='') as csv_file:
    # Define field names for the CSV file
    fieldnames = ['source', 'target', 'type']
    # Create a CSV writer object
    writer = csv.DictWriter(csv_file, fieldnames=fieldnames)
    # Write the header row
    writer.writeheader()

    unique_combination = set()

    # Read XES files
    for xes_file in os.listdir(xes_directory):
        if xes_file.endswith(".xes"):
            # Read XES file into dataframe
            log = pm4py.read_xes(os.path.join(xes_directory, xes_file))

            # Process each row in the log
            for index, row in log.iterrows():
                # Extract values from the columns
                opex_value = row['opex']
                source = row['source'] # application_shortname
```

```
                endpoint = row['endpoint'] # depend_application_shortname

                combination = (source, endpoint, opex_value)

                if combination not in unique_combination:
                    unique_combination.add(combination)

                    # Write the row to the CSV file
                    writer.writerow({'source': source, 'target': endpoint, 'type': opex_value})
```

Listing A.6. *Creating the CSV file for the Mobile Patent Suits Visualization from the XES file of the connection dataset.*

```
import os
import pm4py
import json

# Directory containing XES files
xes_directory = "/Users/Path"

# Define a dictionary to hold the hierarchical data
data = {}

# Read XES files
for xes_file in os.listdir(xes_directory):
    if xes_file.endswith(".xes"):
        # Read XES file into dataframe
        log = pm4py.read_xes(os.path.join(xes_directory, xes_file))

        # Process each row in the log
        for index, row in log.iterrows():
            # Extract values from the columns
            opex_value = row['opex']
            application = f"bol.{opex_value}.{row['application_shortname']}"
            depend_application = f"bol.{opex_value}.{row['depend_application_shortname']}"

            # Update data dictionary for depend_application
            if depend_application not in data:
                data[depend_application] = {"size": 0, "imports": []}

            # Increment size for depend_application
            data[depend_application]["size"] += 1

```

```
                # Add application to imports of
    depend_application if not already present
            if application not in data[depend_application]["
    imports"]:
                data[depend_application]["imports"].append(
    application)

                # Update data dictionary for application if
    not already present
            if application not in data:
                data[application] = {"size": 0, "imports":
    []}

# Convert dictionary to list of dictionaries
result = [{"name": namespace, "size": data[namespace]["size"
    ], "imports": data[namespace]["imports"]} for namespace in
     data]

# Save the data to a JSON file
with open('hierarhical_edge_bundling.json', 'w') as json_file
    :
    json.dump(result, json_file, indent=4)
```

Listing A.7. *Creating the JSON file for the Hierarchical Edge Bundling Visualization from the XES file of the connection dataset.*

```
import os
import pm4py
import json
import pandas as pd

# Directory containing XES files
xes_directory = "/Users/Path"

# Define a list to hold the hierarchical data
data = []

# Read XES files
for xes_file in os.listdir(xes_directory):
    if xes_file.endswith(".xes"):
        # Read XES file into dataframe
        log = pm4py.read_xes(os.path.join(xes_directory,
    xes_file))

        # Process each row in the log
        for index, row in log.iterrows():
            istio_namespace = row['istio_namespace']
            netloc = row['Netloc']
            path = row['Path']

```

```
            # Skip rows where any required field is empty or NaN
            if not istio_namespace or pd.isna(istio_namespace) or not netloc or pd.isna(netloc) or not path or pd.isna(path):
                continue

            # Check if istio_namespace already exists in data
            namespace_exists = False
            for namespace in data:
                if namespace["name"] == istio_namespace:
                    namespace_exists = True
                    namespace_entry = namespace
                    break

            if not namespace_exists:
                # Create a new namespace entry
                namespace_entry = {"name": istio_namespace, "children": []}
                data.append(namespace_entry)

            # Check if netloc already exists under the namespace
            netloc_exists = False
            for nl in namespace_entry["children"]:
                if nl["name"] == netloc:
                    netloc_exists = True
                    netloc_entry = nl
                    break

            if not netloc_exists:
                # Create a new netloc entry
                netloc_entry = {"name": netloc, "children": []}
                namespace_entry["children"].append(netloc_entry)

            # Check if path already exists under the netloc
            path_exists = False
            for p in netloc_entry["children"]:
                if p["name"] == path:
                    path_exists = True
                    break

            if not path_exists:
                # Create a new path entry
                path_entry = {"name": path}
                netloc_entry["children"].append(path_entry)

```

```
# Save the data to a JSON file
with open('radial_tidy_tree.json', 'w') as json_file:
    json.dump(data, json_file, indent=4)
```

Listing A.8. *Creating the JSON file for the Radial Tidy Tree Visualization from the XES file of the connection dataset.*

```
import pandas as pd

# Read the CSV file into a DataFrame
df = pd.read_csv('mobile_patent_suits.csv')

# Duplicate the target column
df['original_target'] = df['target']

# Create a dictionary to map source to type
source_to_type = df.set_index('source')['type'].to_dict()

# Function to replace target with type if source matches target
def replace_target(row):
    if row['target'] in source_to_type:
        return source_to_type[row['target']]
    return row['target']

# Apply the function to the DataFrame
df['target'] = df.apply(replace_target, axis=1)

# Function to replace source with type
def replace_source(row):
    if row['source'] in source_to_type:
        return source_to_type[row['source']]
    return row['source']

# Apply the function to the DataFrame to replace source values
df['source'] = df.apply(replace_source, axis=1)

# Write the modified DataFrame back to the CSV file
df.to_csv('mobile_patent_suits_GroupHierarchy.csv', index=False)
```

Listing A.9. *Creating the CSV file for the Mobile Patent Suits Visualization for the Group Hierarchy.*

```
import define1 from "./a33468b95d0b15b0@817.js";

function _1(md){return(
md`
<div style="font-size: 26px; font-weight: bold; margin-top: 30px">Interaction Network Visualization</div>
```

```
6
7  <div style="font-size: 16px;margin-top: 20px">This view shows
       teams and the calls made between their applications. The
       arrow points to the application the call is made to.</div>
8
9  <div style="font-size: 16px;margin-top: 20px">Each selected
       team is represented by a different color.</div>
10
11 <div id="top-stats-container" style="display: flex; flex-wrap
       : wrap; margin-bottom: 20px;"></div>
12 <div id="distinct-nodes-info" style="font-size: 16px; margin-
       top: 20px; display: none;"></div>
13
14 <div id="checkbox-container" style="display: flex; flex-wrap:
        wrap; margin-bottom: 10px;"></div>
15 <button id="select-all-button">Select All</button>
16 <button id="select-none-button">Select None</button>
17 <button id="generate-button">Generate Visualization</button>
18 <div id="node-info" style="border: 1px solid grey; padding:
       10px; margin-top: 60px; display: none;position:absolute;
       right:50px;top:550px;">
19   <h2>Node Information</h2>
20   <p id="node-id"></p>
21   <p id="node-additional-info" style="max-width: 300px; word-
       wrap: break-word;"></p>
22   <h3>Additional Information</h3>
23   <p id="node-stats"></p>
24 </div>
25 `
26 )}
27
28 function _2(Swatches, chart) {
29   return Swatches(chart.scales.color)
30 }
31
32 function _chart(suits, d3, location, drag, linkArc,
       invalidation) {
33   const width = 2428;
34   const height = 1400;
35   const types = Array.from(new Set(suits.map(d => d.type))).
       sort();
36   let nodes = [];
37   let links = [];
38
39   let color = d3.scaleOrdinal();
40
41   let simulation = d3.forceSimulation(nodes)
42     .force("link", d3.forceLink(links).id(d => d.id))
43     .force("charge", d3.forceManyBody().strength(-100))
```

```
    .force("x", d3.forceX())
    .force("y", d3.forceY());

  const svg = d3.create("svg")
    .attr("viewBox", [-width / 2, -height / 2, width, height
   ])
    .attr("width", width)
    .attr("height", height)
    .attr("style", "max-width: 100%; height: auto; font: 14px
    sans-serif;"); //outline: thin solid black;

  // Add zoom functionality
  const zoom = d3.zoom()
    .scaleExtent([0.1, 10])
    .on("zoom", (event) => {
      g.attr("transform", event.transform);
    });

  svg.call(zoom);

  const g = svg.append("g");

  // Add checkboxes to the container
  const checkboxContainer = document.getElementById('checkbox
   -container');
  types.forEach(type => {
    const label = document.createElement('label');
    label.style.display = 'flex';
    label.style.alignItems = 'center';
    label.style.marginRight = '10px';
    const checkbox = document.createElement('input');
    checkbox.type = 'checkbox';
    checkbox.value = type;
    checkbox.checked = false; // Set checkboxes to be
   unchecked initially
    label.appendChild(checkbox);
    label.appendChild(document.createTextNode(type));
    checkboxContainer.appendChild(label);
  });

  // Get node stats for nodes in selected visualization
  function calculateNodeStats(nodes, links) {
    const nodeStats = {};

    nodes.forEach(node => {
      nodeStats[node.id] = {
        types: new Set(),
        sources: new Set(),
        targets: new Set(),
```

```
89          sourceNames: [],
90          targetNames: [],
91          productGroup: "",
92          productArea: "",
93          productDomain: "",
94          Description: "",
95          additionalTypes: new Set(),
96          additionalGroups: new Set(),
97          additionalAreas: new Set(),
98          additionalDomains: new Set()
99        };
100     });
101
102     // Updates the nodeStats object with details based on the
        links (type name, source name, target name)
103     links.forEach(link => {
104       nodeStats[link.source].types.add(link.type);
105       nodeStats[link.target].types.add(link.type);
106       nodeStats[link.source].targets.add(link.target);
107       nodeStats[link.target].sources.add(link.source);
108       nodeStats[link.source].targetNames.push(link.target);
109       nodeStats[link.target].sourceNames.push(link.source);
110     });
111
112     // Searches for an entry in the suits dataset that has
      its source equal to the id of the current node
113     nodes.forEach(node => {
114       const nodeData = suits.find(suit => suit.source ===
      node.id);
115       if (nodeData) {
116         nodeStats[node.id].type = nodeData.type;
117         nodeStats[node.id].productGroup = nodeData.
      productGroup;
118         nodeStats[node.id].productArea = nodeData.productArea
      ;
119         nodeStats[node.id].productDomain = nodeData.
      productDomain;
120         nodeStats[node.id].Description = nodeData.Description
      ;
121       }
122     });
123
124     // Adds team (type), source, and target list of displayed
       visualization, i.e., only type, source and target which
      are visualized will be listed here
125     Object.keys(nodeStats).forEach(key => {
126       nodeStats[key].typeCount = nodeStats[key].types.size;
127       nodeStats[key].sourceCount = nodeStats[key].sources.
      size;
```

```
128         nodeStats[key].targetCount = nodeStats[key].targets.
    size;
129         nodeStats[key].typeList = Array.from(nodeStats[key].
    types).join(", ");
130 
131         // Ensure sourceList includes only distinct entries
132         const uniqueSourceNames = Array.from(new Set(nodeStats[
    key].sourceNames));
133         nodeStats[key].sourceList = uniqueSourceNames.join(", "
    );
134 
135         // Ensure targetList includes only distinct entries
136         const uniqueTargetNames = Array.from(new Set(nodeStats[
    key].targetNames));
137         nodeStats[key].targetList = uniqueTargetNames.join(", "
    );
138       });
139 
140       return nodeStats;
141   }
142 
143   // Calculate full stats for each node based on the complete
      dataset
144   const allNodes = Array.from(new Set(suits.flatMap(l => [l.
    source, l.target]))).map(id => ({ id }));
145   const allStats = calculateNodeStats(allNodes, suits);
146 
147   function displayTopStats(stats, selectedTypes,
    additionalStats) {
148       const topStatsContainer = document.getElementById('top-
    stats-container');
149       topStatsContainer.innerHTML = '';
150 
151       // Calculate and display the total number of distinct
    nodes per selected type
152       const nodeCountByType = selectedTypes.reduce((acc, type)
    => {
153         acc[type] = new Set(Object.entries(stats).filter(([_,
    stat]) => stat.types.has(type)).map(([id, _]) => id)).size
    ;
154         return acc;
155       }, {});
156 
157       // Display the total count of distinct nodes per type
158       const distinctNodesInfo = document.getElementById('
    distinct-nodes-info');
159       distinctNodesInfo.style.display = 'block';
160       distinctNodesInfo.innerHTML = '<h3>Total Nodes Selected
    Team makes a call to</h3>';
```

```
    const ulDistinctNodes = document.createElement('ul');
    selectedTypes.forEach(type => {
      const li = document.createElement('li');
      li.textContent = `${type}: ${nodeCountByType[type]}`;
      ulDistinctNodes.appendChild(li);
    });
    distinctNodesInfo.appendChild(ulDistinctNodes);

    // Display additional stats
    const additionalStatsContainer = document.createElement('div');
    additionalStatsContainer.innerHTML = '<h3>Additional Stats</h3>';
    additionalStatsContainer.innerHTML += `
      <p>Distinct Teams: ${additionalStats.types.size}</p>
      <p>Distinct Groups: ${additionalStats.groups.size}</p>
      <p>Distinct Areas: ${additionalStats.areas.size}</p>
      <p>Distinct Domains: ${additionalStats.domains.size}</p>
    `;
    topStatsContainer.appendChild(additionalStatsContainer);
  }

  function calculateAdditionalStats(links, allStats) {
    const additionalStats = {
      types: new Set(),
      groups: new Set(),
      areas: new Set(),
      domains: new Set()
    };

    links.forEach(link => {
      if (allStats[link.source]) {
        additionalStats.types.add(allStats[link.source].type);
        additionalStats.groups.add(allStats[link.source].productGroup);
        additionalStats.areas.add(allStats[link.source].productArea);
        additionalStats.domains.add(allStats[link.source].productDomain);
      }
      if (allStats[link.target]) {
        additionalStats.types.add(allStats[link.target].type);
        additionalStats.groups.add(allStats[link.target].productGroup);
        additionalStats.areas.add(allStats[link.target].productArea);
```

```
200         additionalStats.domains.add(allStats[link.target].
    productDomain);
201       }
202     });
203
204     return additionalStats;
205   }
206
207   // Function to update the chart based on selected
    checkboxes
208   function updateChart() {
209     // Select checked types
210     const selectedTypes = Array.from(checkboxContainer.
    querySelectorAll('input:checked')).map(input => input.
    value);
211     // Filters links based on selected types
212     let filteredLinks = suits.filter(d => selectedTypes.
    includes(d.type));
213     // Identifies nodes (source and target) of selected types
214     const filteredNodesSet = new Set(filteredLinks.flatMap(l
    => [l.source, l.target]));
215
216     // Create a set of sources from the filtered links
217     const filteredSourcesSet = new Set(filteredLinks.map(l =>
     l.source));
218
219     // Add additional source-target combinations based on the
     new condition
220     // Filters the suits dataset again to find additional
    links where the target node is already in the
    filteredSourcesSet
221     const additionalLinks = suits.filter(link =>
    filteredSourcesSet.has(link.target));
222     // Concatenates additionalLinks to the existing
    filteredLinks
223     filteredLinks = filteredLinks.concat(additionalLinks);
224     additionalLinks.forEach(link => {
225       filteredNodesSet.add(link.source);
226       filteredNodesSet.add(link.target);
227     });
228
229     const filteredNodes = Array.from(filteredNodesSet).map(id
    => ({ id }));
230
231     const nodeStats = calculateNodeStats(filteredNodes,
    filteredLinks);
232     const additionalStats = calculateAdditionalStats(
    filteredLinks, allStats);
233
```

```
234     // Update the color scale based on selected types
235     color = d3.scaleOrdinal(selectedTypes, d3.
        schemeCategory10);
236 
237     // Stop the previous simulation
238     simulation.stop();
239 
240     // Clear previous elements
241     g.selectAll("*").remove();
242 
243     // Create a new simulation with the filtered nodes and
        links
244     simulation = d3.forceSimulation(filteredNodes)
245       .force("link", d3.forceLink(filteredLinks).id(d => d.id
        ))
246       .force("charge", d3.forceManyBody().strength(-1200))
247       .force("x", d3.forceX())
248       .force("y", d3.forceY());
249 
250     // Redefine markers
251     g.append("defs").selectAll("marker")
252       .data(selectedTypes)
253       .join("marker")
254       .attr("id", d => `arrow-${d}`)
255       .attr("viewBox", "0 -5 10 10")
256       .attr("refX", 15)
257       .attr("refY", -0.5)
258       .attr("markerWidth", 6)
259       .attr("markerHeight", 6)
260       .attr("orient", "auto")
261       .append("path")
262       .attr("fill", color)
263       .attr("d", "M0,-5L10,0L0,5");
264 
265     // Update the links
266     const link = g.append("g")
267       .attr("fill", "none")
268       .attr("stroke-width", 1.5)
269       .selectAll("path")
270       .data(filteredLinks)
271       .join("path")
272       .attr("stroke", d => color(d.type))
273       .attr("marker-end", d => `url(${new URL(`#arrow-${d.
        type}`, location)})`);
274 
275     // Update the nodes
276     const node = g.append("g")
277       .attr("fill", "currentColor")
278       .attr("stroke-linecap", "round")
```

```
279         .attr("stroke-linejoin", "round")
280         .selectAll("g")
281         .data(filteredNodes)
282         .join("g")
283         .call(drag(simulation))
284         .on("mouseover", (event, d) => {
285           const stats = nodeStats[d.id];
286           d3.select(event.currentTarget)
287             .append("title")
288             .text(`ID: ${d.id}\nTeam: ${stats.typeList}\nTeam
    Count: ${stats.typeCount}\nSources: ${stats.sourceCount}\
    nTargets: ${stats.targetCount}`);
289         })
290         .on("mouseout", (event, d) => {
291           d3.select(event.currentTarget).select("title").remove
    ();
292         })
293         // add box with node info on right side
294         .on("click", (event, d) => {
295           const stats = nodeStats[d.id];
296 
297           const nodeInfo = document.getElementById('node-info')
    ;
298           nodeInfo.style.display = 'block';
299           document.getElementById('node-id').innerHTML = `<
    strong>ID:</strong> ${d.id}`;
300           document.getElementById('node-stats').innerHTML = `
301             <p><strong>Teams:</strong></p>
302             <ul>${stats.typeList.split(', ').map(type => `<li>$
    {type}</li>`).join('')}</ul>
303             <p><strong>Sources (${stats.sourceCount}):</strong
    ></p>
304             <ul>${stats.sourceList.split(', ').map(source => {
305               const sourceType = suits.find(suit => suit.source
     === source)?.type;
306               return `<li>${source} (${sourceType})</li>`;
307             }).join('')}</ul>
308             <p><strong>Targets (${stats.targetCount}):</strong
    ></p>
309             <ul>${stats.targetList.split(', ').map(target => {
310               const targetType = suits.find(suit => suit.target
     === target)?.type;
311               return `<li>${target} (${targetType})</li>`;
312             }).join('')}</ul>
313           `;
314           document.getElementById('node-additional-info').
    innerHTML = `
315             <p><strong>Team:</strong> ${stats.type}</p>
316             <p><strong>Product Group:</strong> ${stats.
```

```
    productGroup}</p>
317         <p><strong>Product Area:</strong> ${stats.
    productArea}</p>
318         <p><strong>Product Domain:</strong> ${stats.
    productDomain}</p>
319         <p><strong>Description:</strong> ${stats.
    Description}</p>
320       `;
321     });
322
323     node.append("circle")
324       .attr("stroke", "white")
325       .attr("stroke-width", 1.5)
326       .attr("r", 4);
327
328     node.append("text")
329       .attr("x", 8)
330       .attr("y", "0.31em")
331       .text(d => d.id)
332       .clone(true).lower()
333       .attr("fill", "none")
334       .attr("stroke", "white")
335       .attr("stroke-width", 3);
336
337     simulation.on("tick", () => {
338       link.attr("d", linkArc);
339       node.attr("transform", d => `translate(${d.x},${d.y})`)
    ;
340     });
341
342     // Update top stats
343     displayTopStats(nodeStats, selectedTypes, additionalStats
    );
344   }
345
346   // Event listeners for the select all and select none
     buttons
347   document.getElementById('select-all-button').
     addEventListener('click', () => {
348     checkboxContainer.querySelectorAll('input[type="checkbox
    "]').forEach(checkbox => {
349       checkbox.checked = true;
350     });
351     updateChart(); // Trigger chart update
352   });
353
354   document.getElementById('select-none-button').
     addEventListener('click', () => {
355     checkboxContainer.querySelectorAll('input[type="checkbox
```

```
    "]').forEach(checkbox => {
      checkbox.checked = false;
    });
    updateChart(); // Trigger chart update
  });

  document.getElementById('generate-button').addEventListener
    ('click', updateChart);

  return Object.assign(svg.node(), { scales: { color } });
}

function _suits(FileAttachment) {
  return FileAttachment("suits.csv").csv();
}

function _linkArc() {
  return function linkArc(d) {
    const r = Math.hypot(d.target.x - d.source.x, d.target.y
    - d.source.y);
    return `
    M${d.source.x},${d.source.y}
    A${r},${r} 0 0,1 ${d.target.x},${d.target.y}
  `;
  };
}

// drag function
function _drag(d3) {
  return simulation => {

    function dragstarted(event, d) {
      if (!event.active) simulation.alphaTarget(0.3).restart
    ();
      d.fx = d.x;
      d.fy = d.y;
    }

    function dragged(event, d) {
      d.fx = event.x;
      d.fy = event.y;
    }

    function dragended(event, d) {
      if (!event.active) simulation.alphaTarget(0);
      d.fx = null;
      d.fy = null;
    }

```

```
    return d3.drag()
      .on("start", dragstarted)
      .on("drag", dragged)
      .on("end", dragended);
  };
}

export default function define(runtime, observer) {
  const main = runtime.module();
  function toString() { return this.url; }
  const fileAttachments = new Map([
    ["suits.csv", { url: new URL("./files/63
    c4d2f34c05d62a116fc16daf04215d82790c6bd036ce5783f7d002c5d83f704798ae8d61da5
    .csv", import.meta.url), mimeType: "text/csv", toString }]
  ]);
  main.builtin("FileAttachment", runtime.fileAttachments(name
    => fileAttachments.get(name)));
  main.variable(observer()).define(["md"], _1);
  main.variable(observer()).define(["Swatches", "chart"], _2)
    ;
  main.variable(observer("chart")).define("chart", ["suits",
    "d3", "location", "drag", "linkArc", "invalidation"],
    _chart);
  main.variable(observer("suits")).define("suits", ["
    FileAttachment"], _suits);
  main.variable(observer("linkArc")).define("linkArc",
    _linkArc);
  main.variable(observer("drag")).define("drag", ["d3"],
    _drag);
  const child1 = runtime.module(define1);
  main.import("Swatches", child1);
  return main;
};
```

Listing A.10. *Code for the Interaction Network Visualization.*