UTRECHT UNIVERSITY

Thesis

Artifact Centric Dynamic Software Architecture Reconstruction

P.S. (Pratik) Kushwaha
p.s.kushwaha@students.uu.nl / kushwahapratik28@gmail.com
DEPARTMENT OF INFORMATION AND COMPUTING SCIENCE
UTRECHT UNIVERSITY
UTRECHT, THE NETHERLANDS

Masters Student, Business Informatics
Utrecht University
Student Number: ***6113567***

First Supervisor:
Dr. ir. J.M.E.M. (Jan Martijn) van der Werf
j.m.e.m.vanderwerf@uu.nl
Utrecht University

Second Supervisor:
Dr. ir. XiXi Lu
x.lu@uu.nl
Utrecht University

Abstract

Introduction: Understanding interactions within software architecture has been an important domain for researchers and business stakeholders. Implementation of a software's architecture can help understand these interactions in detail. However, a software architecture representation showcasing interactions of complex many-to-many type requires a definitive approach.

Method: In this thesis, firstly, we showcase the existing understanding of interactions, and how they are modelled when representing a software system. We further propose an interaction discovery and modelling algorithm to model interactions in a software application dynamically. We follow an artifact-centric process modelling approach and define an extended Petri net modelling framework, where the interaction nodes between artifact communications are modelled using choreographies.

Results: Our solution design presents steps for interaction discovery, where we first capture the dynamic software event logs, consisting of artifacts and their interactions. We define further steps to model dynamic interaction nodes using choreographies and create an algorithm to generalize an artifact's lifecycles. We model these artifacts using our extended Petri nets formalism, and together they represent the technical architecture of the system. We employ a software architecture modelling approach which segregates the architecture at various abstraction levels. The most detailed level represents the artifact-centric view of the dynamic software system, showcasing complex interactions and their respective cardinalities.

Discussion: These results suggest that it is feasible to extract generalizable, objectively transparent architectural model of a software system by dynamically executing and generating lifecycles of each software artifact. In this process, we uncover their true interactions and can model them using an extended version of existing modelling formalisms. The software architecture reconstruction represents redefining the architecture at various abstraction levels, where the most detailed level represents the artifact-centric dynamic software interactions.

Keywords: *Artifact-Centric, Interaction Discovery, Software Architecture, Interaction Modelling*

Preface

> I would rather have questions that can't be answered than answers that can't be questioned.
>
> *Richard Phillips Feynman, American theoretical physicist*

In August 2019, I began my journey on this thesis project for my Masters. Constructing software architectures to ease the differences between the technical and business stakeholders always remained an interesting domain for me. Choosing a topic where I was able to shed some light in this domain was an important pivotal point during the initial phases. It helped me to gain more knowledge about software interactions and understand where the knowledge gap exists. Over the months, this project challenged me to carry out important decisions that proved vital to my research project. I am glad to have learned from the difficulties and opportunities in this subject area, and contributed in tackling the research gap, if ever so slightly.

This study would not have been possible without a number of important people, who guided and helped me throughout my research phase. Firstly, my supervisor Jan Martijn van der Werf, for your never-ending guidance and support, challenging me and helping me learn and understand things. Secondly, my supervisor XiXi Lu, for your guidance and perspective on various topics of my study, throughout the research process. Last but not least, I would like to thank my girlfriend, friends and family, who encouraged me to excel this year. Without them, none of this would have been possible.

Pratik Kushwaha

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software systems are represented by the overall architecture of the implemented system. Software architecture is defined as the structure or structures of the system, which comprises software elements, the externally visible properties of those elements, and the relationships among them [8]. Improvement of software systems over time has also led to more complex software systems, thus increasing the overall software architectures complexity. The complexity of a software system increases even more with time, as often the intended architecture of the software ends up disagreeing with the implemented software [9]. The intended architecture is referred by the documented architecture of the system, which was designed to meet its intended business and quality goals. Software architects and researchers have recognized this problem and over the years, invested their energy in designing and developing various software architecture reconstruction [9] approaches. Most of these approaches result in static views of the system gathered from the system's source code [10], or via dynamic approaches of software execution data [6]. Understanding the dynamic views of the implemented architecture of a software system often requires discovering the underlying process of the overall system.

Data from relational databases, such as that of ERP systems, have been used to mine their schema and perform artifact centric process analysis to generate a life-cycle model of the software, thus reconstructing its architecture [2]. Artifact-centric models describe the software processes by defining artifacts, which are smaller business entities having their own life-cycle and information model. These artifacts models collaborate and communicate, and the overall behavior represents the entire software's architecture [11]. ERP systems can be decomposed to provide an artifact-centric description using the system's database schema. A similar artifact-centric process mining approach can be also applied using software operation data. Software operation data describes the observed behavior of a software system, which can help in discovering the main elements of the software architecture and the behavior of these elements. This data can be fetched during software execution by logging the right information about the system's operation. The understanding of this dynamic view of the system and the runtime behavior of software elements is gaining interest in software architects and researchers. The difficulty lies in discovering the right business process-oriented views or artifacts, and their interactions in the software system. To understand the software system and its structure, software execution data have been used that records the execution data of a system. Application of process mining techniques [12] to analyze software execution data using event logs led to multiple studies [6, 13, 14, 15, 4], under the research area of Software Process Mining

[5]. These studies show how system execution data enriches the software architecture by helping discover the overall module communication and visualize the internal behavior of modules. However, they do not rely on an artifact-centric approach to identify software processes within the architecture. When we consider real-life examples of software systems, where several instances of a business artifact can be invoked during execution having their own life-cycle, the artifact-centric approach seems to represent the software processes more naturally [11].

The dynamic reconstruction of a software architecture requires details that can be fetched from the software system during the execution of a specific operation. What can be fetched from the system, how can we fetch it, what is the structure of the information fetched, and how can this data be utilized to construct a software architecture reconstruction approach? Software systems produce all kinds of information that can help us answer these questions. Event logs can be be mined to extract information for an artifact-centric setting so that the recorded executions contain events of artifact life-cycles and their interactions [11]. In addition, the event logs generated usually have a hierarchical structure where every level could define a process model [14, 4]. When dealing with this dynamic approach, researchers have often limited their scope in terms of the interactions of the architectural elements covering mainly one-to-one relations between them [4, 16, 17, 18]. In our thesis, we focus on not just the discovery of artifact elements and their interactions, but also on the type of interaction that emerges dynamically having many-to-many relations, and the visualization of the overall architectural model depicting the dynamic behavior.

When it comes to interactions of these elements, researchers such as Decker [19] have defined them by distinguishing it into elementary interactions and complex interactions. Elementary interactions are standard message exchanges such as changing notification settings for a given application, or an acknowledgement flags sent to a specific component of an application. Complex interactions are defined as a set of elementary interactions that are invoked for a given functional use case. Consider a use case where a notification component updates multiple users simultaneously at runtime by refreshing their notification feed based on relevant activities performed by other users. This is one such complex interaction that involves a notification registry or database sending updates to the notification component of an application, which further classifies the notification content and instantly updates the notification feed of the user relevant to that notification content. We will employ a similar example, described in detail in the upcoming sections of this chapter, to answer our research questions. Simple elementary interactions are covered by multiple researchers, and formal notations and techniques exist; however, discovering complex many-to-many interactions and modeling them correctly remains a significant gap in the literature. This gap is highlighted by our analysis of state-of-the-art interaction modeling techniques and formalisms in Chapter 4.

1.1 Problem Statement

As part of this research, the intention is to bridge the above-discussed gap by providing a better understanding of how to analyze dynamic information from the system and discover detailed interaction models from software operation data.

The main problem statement for this research thesis can be summarized as,

The discovery of an architectural model from the dynamic perspective of a software system using an artifact centric approach lacks thorough research. The available studies do not cover the complex interactions (many-to-many relations) between identified elements or artifacts.

In response to this problem, our research proposes the following,

Dynamic reconstruction of a software architecture using artifact centric process mining, and visualization of the architecture covering many-to-many interactions between artifacts.

We hypothesize that for representing complex interactions between artifacts within software architecture, we would need to create an interaction discovery algorithm along with a modeling notation for representing them, and would be discussed throughout our literature review section in Chapter 4. We will also define our research questions in Chapter 2 that tackles our problem statement.
On an extensive level, the research aims to achieve the following tasks via this thesis,

- Analyze state-of-the-art approaches for modeling interactions. Showcase the need for a dynamic approach covering one-to-many or many-to-many type of interactions between artifacts.
- Design and develop an interaction discovery algorithm to capture such type of communication between artifacts. Apply an existing conformance check [12] approach with sample software systems using the developed algorithm.
- Application of the developed interaction discovery and visualization approach within the software architecture reconstruction process.

1.2 Running Example

As a running example in this thesis, a notification content management systems example is considered. To make it simple, the example is designed to showcase the basic modules of a notification content management system. This can be compared to a notification feed that a particular user sees in their software application, which helps them track activities on various data points in the software system. The example application consists of users who register to the application and get updated as and when another user performs any activity relevant to the logged-in user. A notification content database is defined to capture activities and is fed as a trigger point for the creation of the notification feed. The application is kept minimal to showcase the overall functionality of the notification system. It makes use of the observer pattern design for achieving the complex communication protocols between various modules of the system. Loggers are created in the application to trace messages or events as and when they are executed during a runtime scenario.

The application consists of three modules, namely, the user module and the main application module. There is also a notification module which works closely with the main application module. Whenever a new notification is received, for example, when a particular user likes a post, this information is captured in the notification module, which informs the main application. The user module is where the users of the application are registered. When a user wants to register for a notification, it sends a request to the main application, and the main application, in turn, notifies the user with its relevant notification. The user will continue to receive relevant notification updates in this manner until the user is deregistered. The application is modeled in such a way that multiple users can be registered with the main application, and whenever a notification is received by the main application via the notification module, the users will get notified. The manual behavior of users registering and deregistering is performed randomly to create a more dynamic scenario. This entire scenario is implemented using the observer pattern, which is designed to deal with complex communication scenarios in a notification system. An AspectJ logger for logging essential communication calls and events is implemented in the application, and logs are generated in a rolling file using Log4j2. The log files generated contain individual call events [6] before and after a critical event is initiated in the application and needs to be modified further to understand their interactions with correct timestamps.

1.3 Research Motivation

Researchers identifying the critical business elements have conducted numerous studies, often termed as artifacts, based on the data in their relational databases. Recent studies took another approach to identify these business entities by looking at the source code of the implemented software architecture. This approach gained more support due to the increase in software coding standards and practices, which were more structured and well designed to guarantee software efficiency. It depicted major business elements that were broken down by the development team in coordination with a domain expert resulting in the currently implemented software development architecture. Leveraging this development architecture setup is exactly where the researchers focused their aim. Various studies were performed to identify the critical software components from the development model, and this was done with the help of software event logs. Fetching and analyzing the event data using multiple identification techniques and algorithms were part of a significant amount of research. However, this is not the complete picture. Interactions between these components would depict how these business elements were communicating. However, when it comes to identifying the interactions between the components, only a limited number of studies focus on this domain. Few studies which did try to analyze or identify this communication cover simple interactions with interaction cardinality of one-to-one. When it comes to identifying and modeling a complex set of interactions, especially for one-to-many or many-to-many type of interaction, the current techniques lack depth.

This research aims to look at existing methods that are used to describe the interactions between the software component and the reasons why there is a need for a modeling standard for complex software interactions. The primary purpose is to reconstruct the software architecture better, which identifies not only the right business entities but also

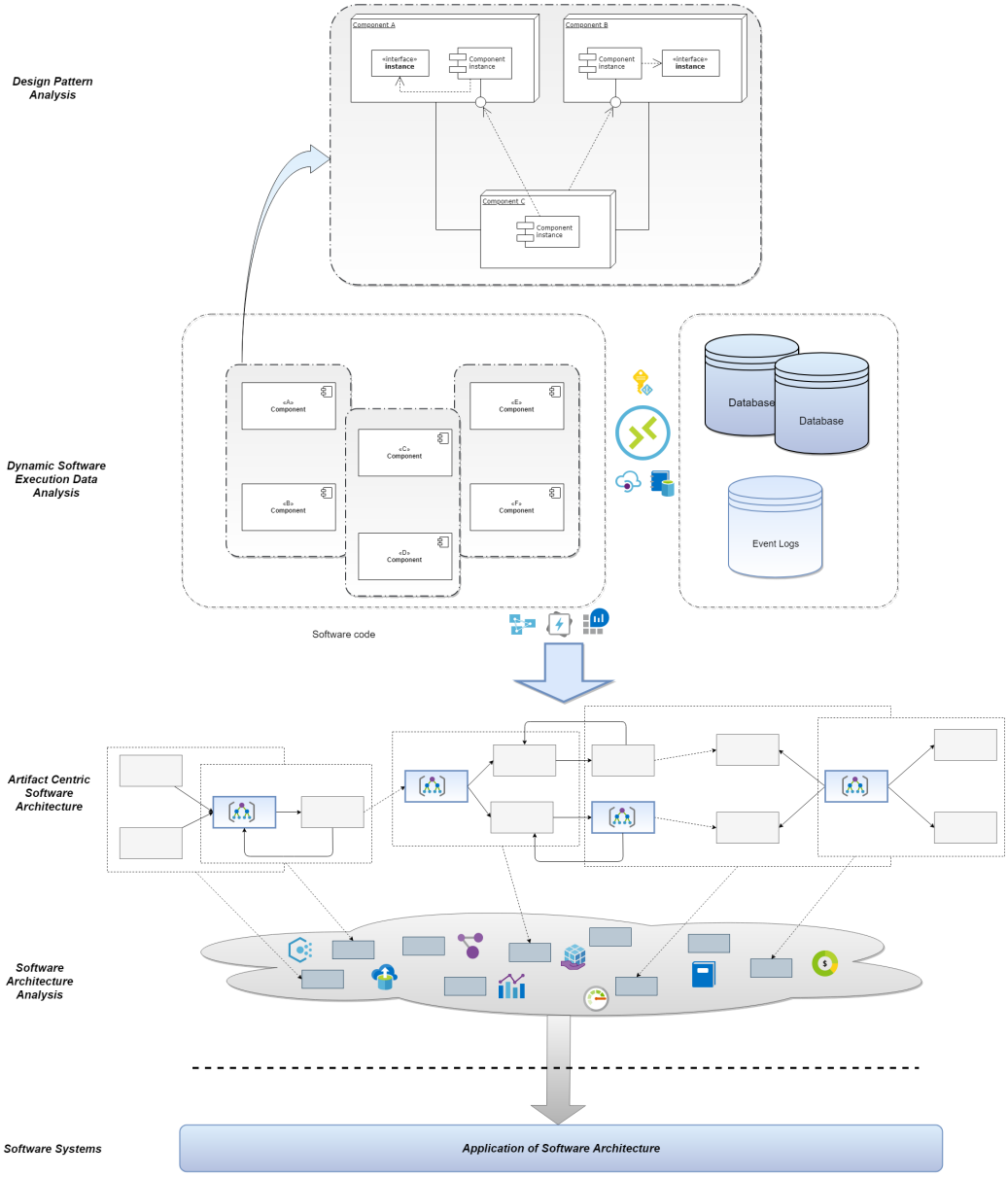


Figure 1.1: Thesis motivation and application

their complex interactions. The need for this is showcased with the help of a running example that implements the observer pattern for a notification component. The running example is discussed in detail in further sections. Figure 1.1 gives an overview of the leading research goal of the thesis and its application motivation. The first layer depicts the code design pattern and architecture in place, in this case, an intricate communication pattern (observer pattern). The understanding of the runtime behavior of the first layer lies with the understanding of the dynamic software execution data via event logs. The research relies on leveraging the advantages of artifact centric modeling approach and further, the applications of the software architecture in real-world software systems. Thus, the association of various business tasks with these set of artifacts models can be performed by software stakeholders and domain experts, leading to generation of crucial business workflows.

Chapter 2

Research Approach

This section defines the primary research approach covering the research questions, the relevant research approach, and the research method was chosen for the achieve those tasks. We end with a section for our research motivation and the overall thesis outline.

2.1 Research Questions

Based on the described problem statement, we were able to define the research approach, as mentioned in the earlier section. The research objectives are postulated in the form of research questions to test our hypothesis and validate our results. Thus, we define our main research questions for this research thesis as follows,

Research Question

***RQ**: How to reconstruct a software architecture that can visualize the dynamic behavior of complex software system interactions?*

In order to answer the main research question (RQ), we propose a set of sub-questions (SQs) that decomposes the main research question. We come up with four critical sub-research questions mentioned below that concludes our main research question for this thesis. The sub-research questions are as follows,

SQ1: What are the state-of-the-art approaches to best describe a dynamic software architecture reconstruction?

*To answer this sub-question, we look at various interaction discovery and modeling approaches. We showcase how the existing interaction modeling language falls short to model certain communication behaviors. We further try to analyze the shortcomings and define the key areas of interaction discovery and communication that needs to be catered as part of this research. This is a literature question and will be answered via thorough **detailed literature research** of state-of-the-art interaction modeling languages.*

SQ2: How to visualize the software execution data in a software architecture having one-to-many or many-to-many type of interaction between artifacts?

*We present an **interaction modeling language** which is defined with the help of existing state-of-the-art modeling languages by extending them to our research goals. This visualization of software execution data to depict the communication patterns between various artifacts of a process model will further help in redefining the software archictecture of the system.*

SQ3: How to extract an interaction model from software execution data using an artifact centric approach?

*Answering this sub-question involves creation of an **interaction discovery algorithm**, and further an interaction modeling language. We use our running example which represents a scenario of complex interactions for multiple instances of a user's notification content. We make use of the software execution data and highlight the communication as a basis to create our interaction discovery algorithm.*

SQ4: How can the proposed techniques be applied in real-life complex software systems?

*After creating our approach of defining complex interactions, the application of such formalisms in real life software systems or similar example applications will validate our approach. Validation is performed using a **controlled experiment on our running example**. The results will be summarized after detailed assessment and interpretations of our algorithm and visualization techniques on such applications.*

2.2 Research Phases

The research phases for this thesis is chosen as per the design science research (DSR) approach [20]. The design science approach is targeted for improvement problems, that aims to iteratively design and demonstrate a solution for the problem context. The necessary design science process and its relevance with the research scope are shown in the figure 2.1.

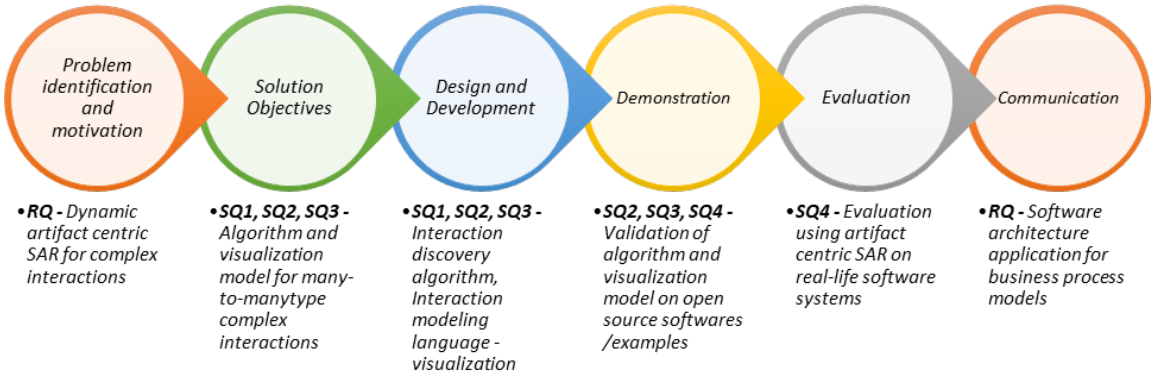


Figure 2.1: Design Science Research approach and thesis relevance

Chapter 3

Preliminaries

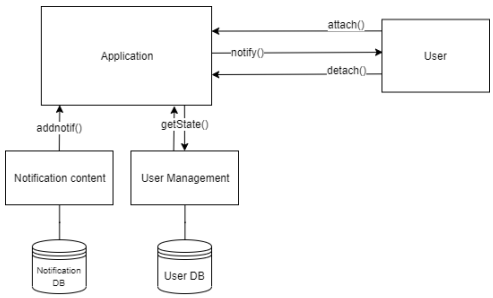
This chapter consists of the preliminaries for this research, where we give an overview of key concepts and their relevance to our research. The running example is referred throughout the chapter, wherever applicable, to show highlight our research use cases.

3.1 Software Architecture

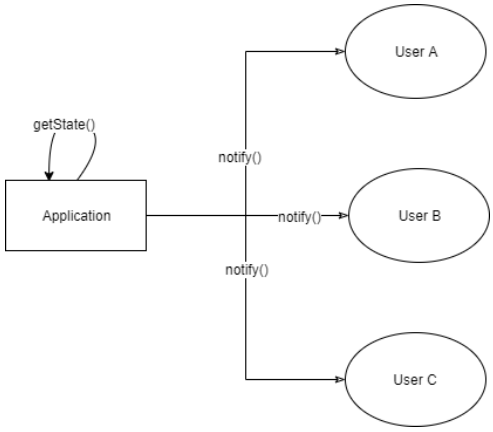
We start by defining a few key concepts which are widely used in the software architecture community and which would be referred to in the latter part of this thesis. We know that a software architecture defines the system's structure which comprises of software elements, its properties, and the relationships among them [8]. We will be looking at software architecture from the perspective of static and dynamic structures of the system. The static structures within a software architecture define a systems internal design-time elements and their arrangement. The dynamic structure defines the systems runtime elements and their interactions. The primary sources of a software architecture's information are found in the static view of a software fetched from the source code and structural connections among classes. The dynamic information of the overall system is obtained using traces, logs, and events [14]. The information captured using both static and dynamic sources often overlap. Static analysis of a software architecture is usually faster (more efficient) than dynamic analysis, but often less precise [14], and when it comes to an understanding and modeling the lower level interactions correctly, we hypothesize that precision would be important. This would be discussed further during our interaction discovery model generation step. We make use of software execution data in our running example, as our basis to understand and model the elements and interaction in the software architecture.

3.2 Software systems

In this thesis, we look at software systems and their architecture and implementation elements to understand the overall software architecture and its modeling scenarios. Before we can use these elements in our approach, it is necessary to look at the meta-model of a software system.



(a) High level architecture of the notification application



(b) Multiple user scenario for notifying via the application

Figure 3.1: High level view of running example

3.2.1 Components and interfaces Class and methods

The structure of a software code comprises various components, and these components interact with each other via interfaces in an object-oriented software system. An individual software component can be defined as a software package, a web service, a web resource, or a module that encapsulates a set of related functions or data. In our thesis, we define components as classes or group of classes, which are instantiated multiple times during the a software execution via multiple component instances[5]. A specific software application can thus be structured and represented by logically divided components, each having a set of operations that it performs. When a component has to perform any service for the rest of the system, it makes use of an interface that specifies the services that other components can utilize. The interface can be seen as a signature of the component, and any stakeholder using the interface does not need to know about the inner workings of the component before using it. Similar to components instances, an interface can also have multiple instances based on the interface definition.

Components and interfaces are defined as the architecture-level elements of a software system. When we look at the elements of the software system from an implementation point of view, an object-oriented software system consists of a group of classes. A class is defined as an extensible program code-template for creating objects, providing initial values for variables, and implementations of behaviors represented as methods. A class may have multiple methods and can be instantiated multiple times, relating to multiple objects. When a method is invoked, it is called as a method call. Similar to how components defined as classes or groups of classes that can be instantiated multiple times, based on the functional aspects of the component, interfaces can also be defined. These interfaces are invoked to be used by other components as per the functional need of the application. In our running example, the main components are the user, application and notification components which together forms the notification feed module as shown in the Figure 3.1. The interfaces in our running example would be the communication interfaces between various classes within a given component, as well as the inter component communications across the application.

3.2.2 Software Execution Data - Event logs, Trace, Case

Software execution data can be perceived in the form of logs. Event logs can be defined to contain data related to the execution of a single process, which is referred to as a case for the events within it. Every event can be looked as an activity carried out in the software execution process. Additionally, it can also contain the resource and a timestamp of the event. A resource is anything that carries out the execution, i.e., it could be a user or a module within the software. A trace is defined as a finite sequence of events in which the event occurs only once. Trace can be viewed as an ordered list of events pertaining to a specific attribute of the event in a case, that is a process instance [14]. In our example of notification content feed, the basic event log is generated using AspectJ logger embedded within the application, as shown in the figure, 3.2.

```
2019-10-30 13:50:42.386; [main]; notification.User; void notification.User.checknotif(ArrayList)_ID_16863697101; void notification.User.checknotif(ArrayList)
2019-10-30 13:50:42.387; [main]; notification.User; void notification.User.update(Application)_ID_16863697101; void notification.User.update(Application)
2019-10-30 13:50:42.387; [main]; notification.Application; void notification.Application.Notify()_ID_16863697101; void notification.Application.Notify()
2019-10-30 13:50:44.402; [main]; Enter notification.Application; void notification.Application.Deattach(Observer)_ID_16863697101; void notification.Application.Deattach(Observer)
2019-10-30 13:50:44.403; [main]; notification.Application; void notification.Application.Deattach(Observer)_ID_16863697101; void notification.Application.Deattach(Observer)
2019-10-30 13:50:44.403; [main]; notification.Application; void notification.Application.addnotif(Notification, Observer)_ID_16863697101; void notification.Application.addnotif(Notification, Observer)
2019-10-30 13:50:44.404; [main]; Enter notification.Application; void notification.Application.addnotif(Notification, Observer)_ID_16863697102; void notification.Application.addnotif(Notification, Observer)
2019-10-30 13:50:44.405; [main]; Enter notification.Application; void notification.Application.Attach(Observer)_ID_16863697102; void notification.Application.Attach(Observer)
2019-10-30 13:50:44.405; [main]; notification.Application; void notification.Application.Attach(Observer)_ID_16863697102; void notification.Application.Attach(Observer)
2019-10-30 13:50:44.406; [main]; Enter notification.Application; void notification.Application.Notify()_ID_16863697102; void notification.Application.Notify()
```

Figure 3.2: Snippet of the generated basic event log via AspectJ logger

3.2.3 XES and ProM

XES (eXtensible Event Stream) standard is an XML-based standard used for representing event logs. It provides a general format that is acknowledged by the scientific community for capturing event data. It was created by the IEEE task force for process mining and is used in different tools and application for the same purpose [21]. It is also extended to a format for capturing XES software execution data called XES-Software. This was done to understand the behavioral model discovery of software execution data. XES software extension consists of method call-related attributes which are the callee class, callee object, caller method, caller class, return value along with the attributes related to the method such as input parameter type and return type [22]. It can also be associated with software telemetry extensions to support the data capturing the basic performance profile related resources such as CPU usage, thread usage, and memory usage [14].

We make use of the XES software extension format for our thesis. It consists of attributes as shown below with the data they refer [23],

- Class class name of the called method
- Object instance name of the called method
- Parameter Type parameter type of called method
- Parameter Value parameter value of the called method
- Return Type the return type of called method
- Return Value return value of the called method
- Caller method method name of the caller method

- Caller Class class name of the caller method
- Caller Method method name of the caller method

We also make use of ProM6 throughout our thesis, which relies on XES format as an exchange for mining processes. It is a generic open-source framework for implementing process mining tools or algorithms in the form of plugins. There are various plugins embedded in ProM6 which is used to mine process structures and interactions out from a given log [21]. In our running example, we use ProM tool by converting the generated AspectJ logs into XES format as shown in Figure 3.3. The log file generated contains the timestamp, thread ID, class name (caller method), method name (callee method), message. During the implementation of AspectJ logger for the application, the event log content had to be modified to capture every interaction for a given observer/user attached to the application. We defined the log structure so that every observer has a unique ID, which can then be viewed as an individual case for the notification content update event. This unique ID is accompanied by the callee method, which is the method called for an operation. We also segregated the entry and exit logs for every operation such that the entry and exit timestamp can be extracted. The extraction is performed by the python converter, which also converts the log file into a ProM readable .CSV format. As part of a scenario, we took six different instances of notification content, which is pushed via a notification module to the application. In order to make the behavior slightly asynchronous, we attached and detached observers randomly. Hence, as we can see in the figure, in one scenario of the event log, the first observer gets attached before the addnotify() method is called.

```
CaseID;Start Time;End Time;Thread;Caller;Callee;Message
ID_16863697101;2019-10-30 13:50:42.380;2019-10-30 13:50:42.381; [main]; notification.Application; void notification.Application.Attach(Observer)_ID_16863697101; void notification.Application.Attach(Observer)
ID_16863697101;2019-10-30 13:50:42.384;2019-10-30 13:50:42.385; [main]; notification.Application; ArrayList notification.Application.getState()_ID_16863697101; ArrayList notification.Application.getState()
ID_16863697101;2019-10-30 13:50:42.386;2019-10-30 13:50:42.386; [main]; notification.User; void notification.User.checknotif(ArrayList)_ID_16863697101; void notification.User.checknotif(ArrayList)
ID_16863697101;2019-10-30 13:50:42.383;2019-10-30 13:50:42.387; [main]; notification.User; void notification.User.update(Application)_ID_16863697101; void notification.User.update(Application)
ID_16863697101;2019-10-30 13:50:42.383;2019-10-30 13:50:42.387; [main]; notification.Application; void notification.Application.Notify()_ID_16863697101; void notification.Application.Notify()
ID_16863697101;2019-10-30 13:50:44.402;2019-10-30 13:50:44.403; [main]; notification.Application; void notification.Application.Deattach(Observer)_ID_16863697101; void notification.Application.Deattach(Obser
ID_16863697101;2019-10-30 13:50:42.370;2019-10-30 13:50:44.403; [main]; notification.Application; void notification.Application.addnotif(Notification, Observer)_ID_16863697101; void notification.Application.
ID_16863697102;2019-10-30 13:50:44.405;2019-10-30 13:50:44.405; [main]; notification.Application; void notification.Application.Attach(Observer)_ID_16863697102; void notification.Application.Attach(Observer)
ID_16863697102;2019-10-30 13:50:44.407;2019-10-30 13:50:44.407; [main]; notification.Application; ArrayList notification.Application.getState()_ID_16863697102; ArrayList notification.Application.getState()
```

Figure 3.3: Snippet of the generated and modified event log via AspectJ logger

We further push the converted file into ProM and analyze the log structure. This log example is further in the research used to understand and analyze interactions between various software entities and to create a model capturing such interactions. The figure shown in 3.4 shows the statechart of the application generated via the event log. The figure 3.5 shows the causal graph of interaction between components mined using an alpha miner in ProM. The dependency graph of components is also generated 3.6 using the data-aware heuristic miner plugin, which showcases the dependency model of each component and its dependency weight. We will make use of these graphs in further sections to better understand the components and their interactions.

Several pieces of literature mentioned under the related literature section make use of ProM to implement various process mining and modeling techniques. These discovered processes and their interactions are visualized using various modeling languages discussed in the upcoming sections.

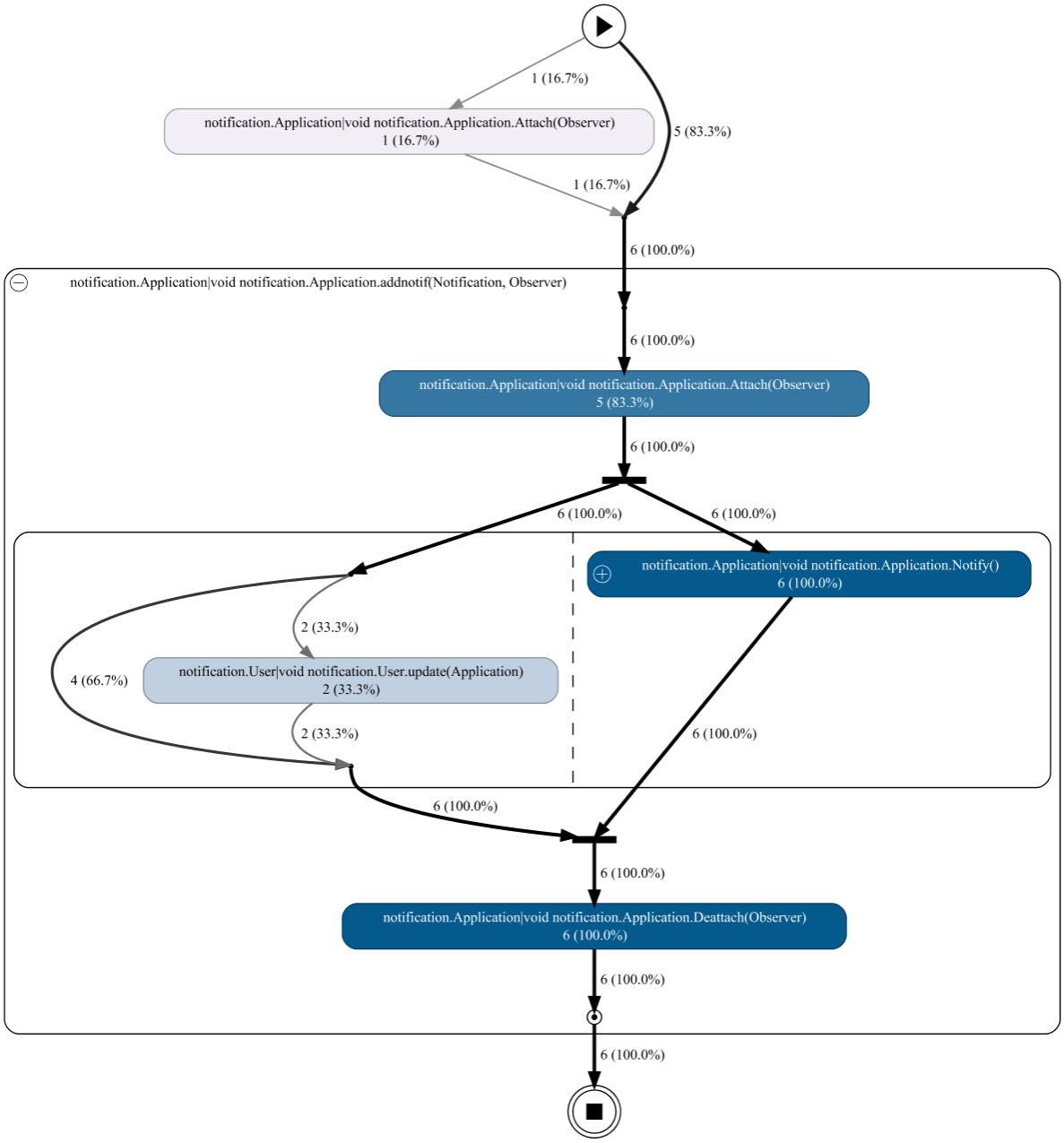


Figure 3.4: State chart of the application from the generated log via ProM

3.3 Process Mining

Process Mining is a domain that aims to discover, monitor, and improve the real-life process by extracting process-related insights from such execution data, or event logs [12]. The main parts of the process mining techniques are defined as shown in the Figure 3.7: (1) process discovery, which is used to find or discover process models to describe the behaviour of the runtime , (2) conformance checking, which verifies the recorded behaviour with the provided process model, and (3) enhancement, which tries to combine the process model with the results of the conformance [1]. To define the process models, artifact centric process modeling proves to be a better model as artifacts describe the business entities by both an information model and a lifecycle. It defines a process in terms of

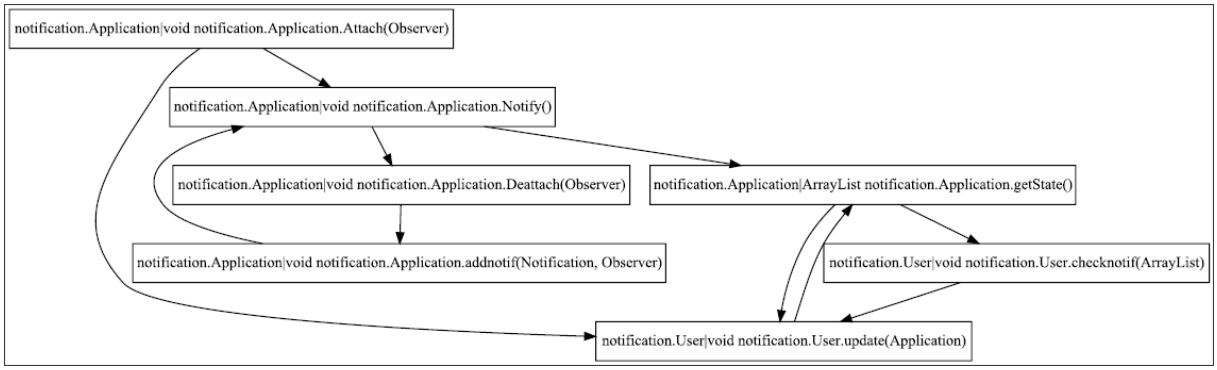


Figure 3.5: Causal Graph using alpha miner - ProM

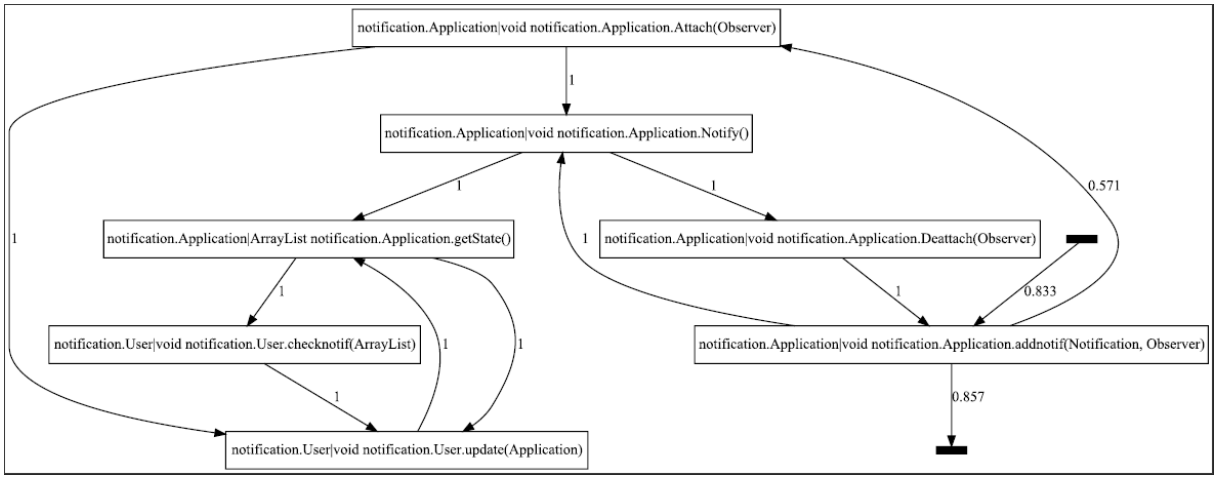


Figure 3.6: Dependency graph using data-aware heuristic alpha miner ProM

multiple collaborative artifacts, which is analyzed with its lifecycle and interactions with each other [3]. The elements of software architecture are chosen based on the main functional aspects of the application in design. We make use of process mining in our running example by first extracting event logs for the notification component in our application, which forms the process discovery step. The event logs are generated to discover the main elements of the software code and their interactions. The main elements of the software in our running example are the main application component working closely with the notification component, and the user component. These three components can represent the three significant artifacts of the application where the main component is the core application, which is used by users via the user component and works closely with the notification component to manage the notification feed for the users. We further analyze the recorded event logs for conformance and enhancement of the model.

3.4 Modeling the Software Architecture

3.4.1 Views and Viewpoints

We will first look at the definition of architectural views and viewpoints. The software architectural view is defined as a representation of one or more structural aspects of an

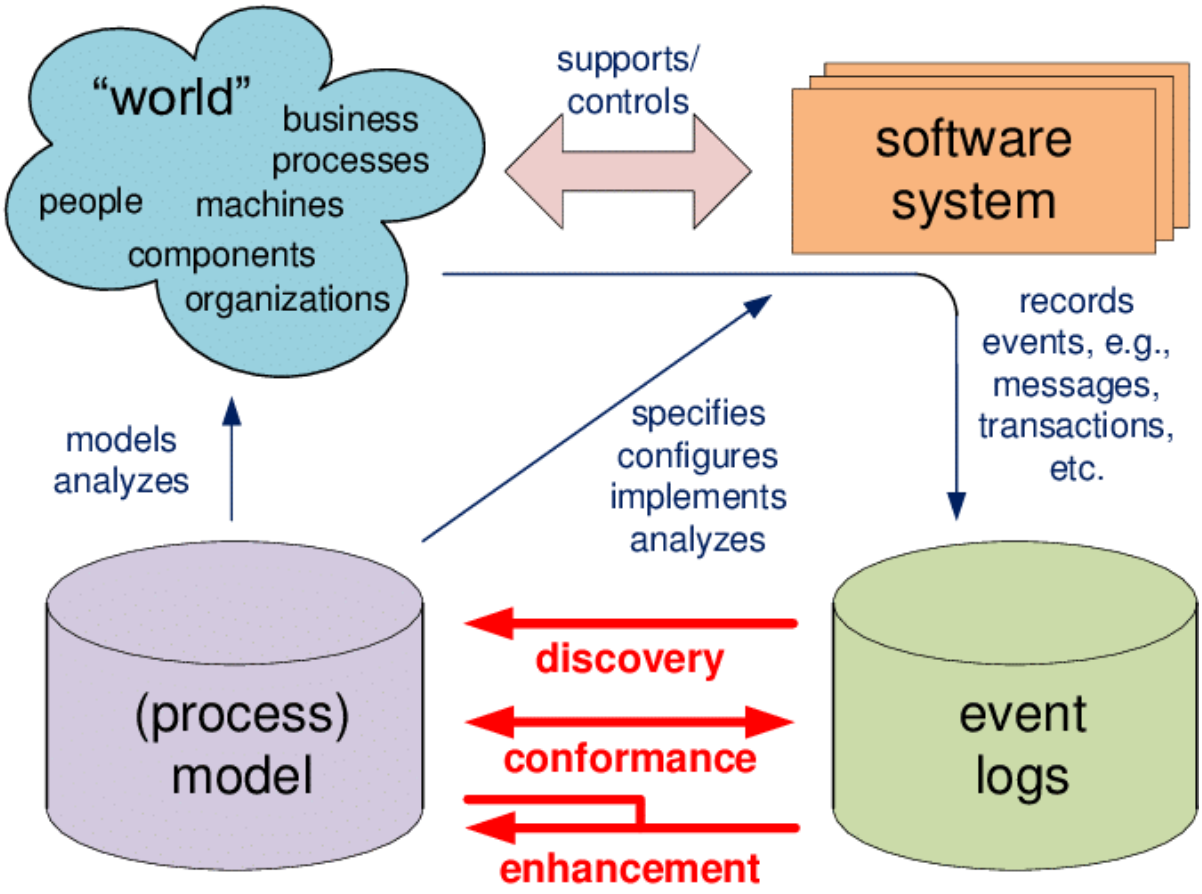


Figure 3.7: Process mining types: Discovery, Conformance, Enhancement [1]

architecture that illustrates how the architecture addresses one or more concerns held by one or more of its stakeholders [24]. A viewpoint is a collection of patterns, templates, and conventions for constructing one type of view. It defines the stakeholders whose concerns are reflected in the viewpoint and the guidelines, principles, and template models for constructing their views. There are six core viewpoints namely, the Functional, Information, and Concurrency viewpoints that characterize the fundamental organization of the system, the Development viewpoint which exists to support the systems construction, and the Deployment and Operational viewpoints which characterizes the system once in its live environment.

The functional viewpoint gives a nice overview of the overall functionality of the application, but it has poorly defined interfaces. When the viewpoint needs more information about the system processes, using the information viewpoint can be handy. The system-level concurrency models consist of processes and inter-process communication which can be depicted using a UML statechart. Achieving consistency across these views can help define the architectural description and is a vital characteristic without which the system will not work correctly, will not achieve its design goals, and may even be impossible to build [24]. Each view that we employ consists of one or more models that are used to represent the view. A model is defined as an abstract or simplified representation of some aspects of architecture. The purpose of a model is to communicate those aspects of the system to one or more stakeholders. Thus, models act as a medium of communication,

helps us understand and analyze the situations in a software system, and also helps us align our processes, teams, and deliverables.

3.4.2 Petri nets and BPMN Choreography

When a process is visualized to depict the structural and behavioral aspects of software architecture, researchers have relied on numerous modeling techniques and formalisms such as Petri nets, Process trees, Business Process Modeling Notations (BPMN), Unified Modeling Language (UML) activity diagrams, etc. The choice of a modeling language is based on what needs to be expressed and at what level of the software. Also, representing concurrency in software is an essential decision while visualizing a process. Petri nets are often used the most as they are widely applied in process mining techniques for concurrency, and because most of the other notations can be converted into a Petri net format. However, Petri nets showcase complex visualizations and are often very large, and thus we do need to rely on other high-level formalisms, which can be useful for stakeholders performing high-level analysis. For this thesis, we will use Petri nets along with other formalisms such as other interaction modeling techniques mentioned below to represent the complex one-to-many type of communication between identified process artifacts.

Petri nets are a directed bipartite graph with two types of nodes within it called places and transitions. Petri nets have various properties that can be applied as per the requirement from the process model. The properties, such as reachability, liveness, and boundedness, which represents a Petri Net, are also represented mathematically. There are several extensions to the Petri Nets, out of which few have backward-compatibility options. Some of the Petri net extensions are colored Petri nets, hierarchical Petri nets, dualistic Petri nets (DPNs), and stochastic Petri nets [25]. Workflow Nets is also a type of Petri Net which is used to model and analyze workflow systems. Workflow nets adhere to the soundness property of a Petri net, especially the block-structure Petri nets represented by process trees, as shown in the paper by Cong [14]. On the other hand, BPMN Choreography diagrams are less formal than Petri nets and can be used to model interactions by using message exchange instances [6]. Every message instance between software class components can thus be modeled for high-level analysis using choreography diagrams. A choreography diagram is more relevant for a business stakeholder trying to understand the current software architecture concerning functional and behavioral aspects of the system involving crucial business elements.

In the case of our running example, the business stakeholder might be interested to see how the notification component addresses an individual users notification feed during runtime. Analysis of various non-functional attributes, such as security, performance, scalability, availability, and resilience. Understanding the concurrency viewpoint of the notification module of the application would thus describe the concurrency structure of the system. It would map functional elements to concurrency units to identify the parts of the system that can execute concurrently and shows how this is coordinated and controlled [24]. Using UML notations to depict concurrent and composite states seems to be a good visualization of the functional artifacts. Additionally, adding the mentioned non-functional attributes to the notations would make the validation of the resulting architecture much beneficial.

A Petri net of the running example log is showcased in figure 3.8 along with the sequence diagram depicting choreography of the interaction between various application

components for the same scenario in figure 3.9. All these models are generated using the ProM 6.0 tool. We also generated a sequence diagram to cover each observer/user interaction, as shown in figure 3.10. This diagram gives us a detailed interaction sequence for every observer getting notified via the application. We will be using all these models further to tackle our research questions.

3.5 Observer Pattern and related design patterns

Design patterns for software are used to define the software template structure categorized as creational, structural, or behavioral patterns [26]. Another further classification based on the concurrent behavior of the complex software components gave rise to architectural design patterns that can be applied on an architectural level on the software, as in the case of Model-View-Controller or Observer design patterns. For the current research thesis, we look at the Observer pattern which was designed to solve the one-to-many dependency problem between software objects, which were tightly coupled. We know that tightly coupled objects are hard to implement, change, test, or reuse as they consist of interaction calls involving multiple different objects [26]. The observer design pattern comes to rescue this case by defining a design pattern that involves one-to-many dependency communication and hence serves the right tool for analyzing complex interactions in software. This runtime behavior of tightly coupled objects interactions is the main inspiration for this research idea.

The Observer pattern is defined as [26] : One or more observers are interested in the state of a subject and register their interest with the subject by attaching themselves. When something changes in our subject that the observer may be interested in, a notify message is sent, which calls the update method in each observer. When the observer is no longer interested in the subjects state, they can simply detach themselves. An observer patterns interaction behavior can be stated as one-to-many type interaction where multiple observers are registered on the subject or the application. When an update occurs in the application module or its sub-module at runtime, the observer pattern makes sure that the registered observer or user. Figure 3.11 shows the observer pattern implemented for our running example. We can see the one-to-many type of interaction between the subject and observer interface which results in the complex interaction we require for our research investigation. The Observer pattern defines a subject class that (a) maintains a list of observer references; (b) sends notifications to its registered observers on state changes, and (c) provides interfaces for registering and unregistering observer objects. On the other hand, the observer interface defines an update interface for objects that should be notified when the subject changes by invoking the notify method, which implements a sequential call of the update methods on all registered observer objects. In short, the overall observer pattern in our example is identified by the following,

- Observer class interface to update interface objects
- A subject class interface which sends notifications to interested observer objects
- getState method to fetch the attached observers for notifying
- addnotif method to fetch the notification content to be sent

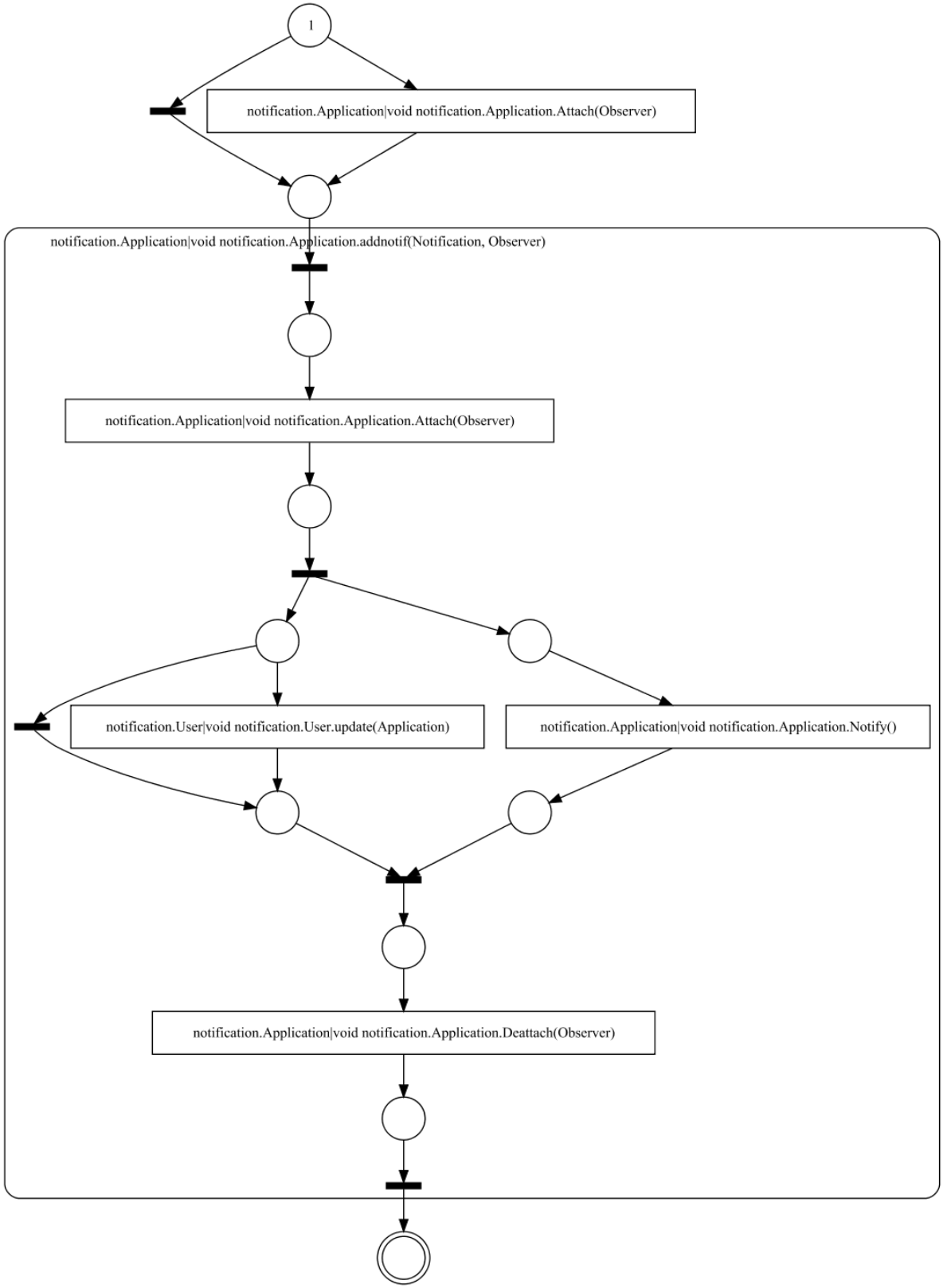


Figure 3.8: Application Interaction Petri Net using ProM

- Notify method to notify observers on subjects state change

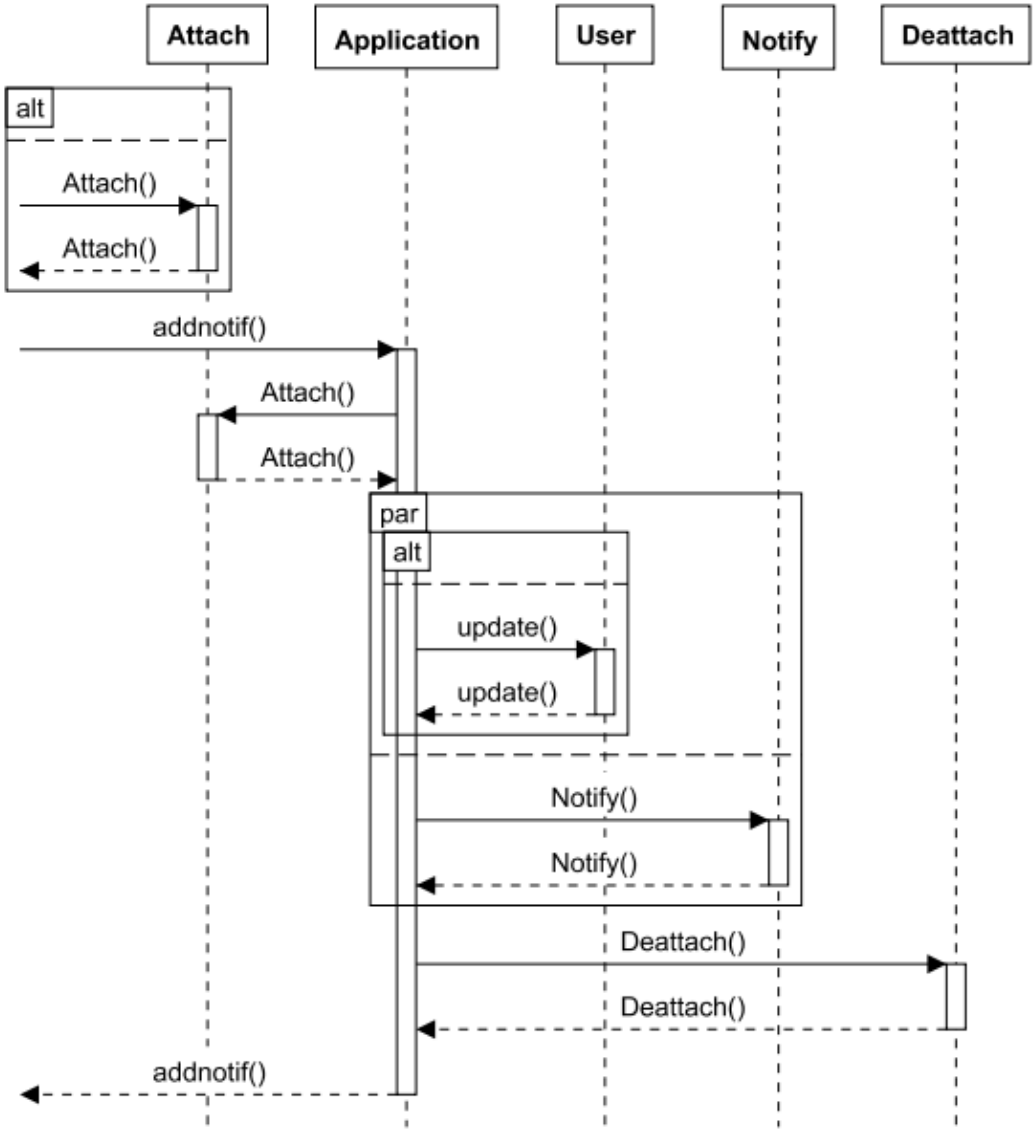


Figure 3.9: Sequence diagram for the application notification module using ProM

- Update method implemented by observer objects called by the notify method
- Register method for adding observers to the set of subjects active objects
- Deregister method for removing observers from the set of subjects active objects

There are similar design patterns that were made to tackle the similar one-to-many or many-to-many type of scenarios as we can see in an observer pattern. A Publisher-Subscriber pattern is similar to an observer pattern with a significant difference in how the observers communicate with the subject [26]. The observers, or in this case, the listeners, are connected to the main subject or publisher via an event bus, which acts as the message broker. The event bus filters the messages and showcases the overall design, where the components are more loosely coupled. The observer pattern is synchronous as compared to the publisher/subscriber pattern asynchronous implementation. However, it can be

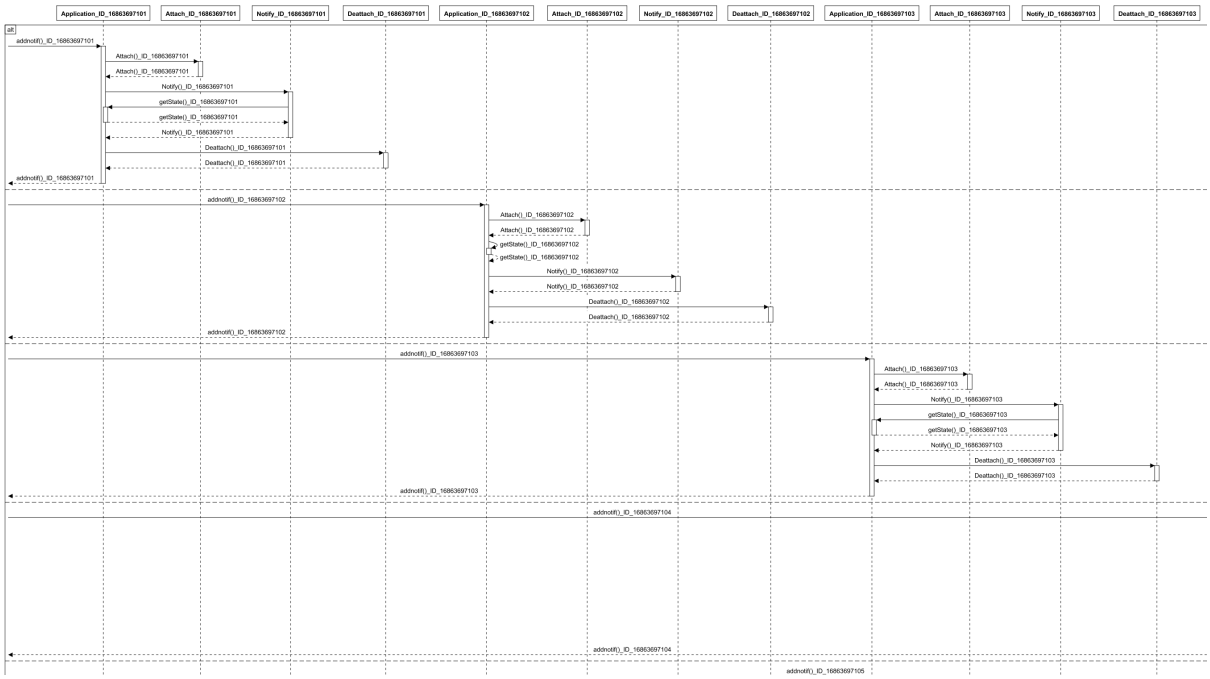


Figure 3.10: Sequence diagram for multiple instance IDs using ProM (partial view)

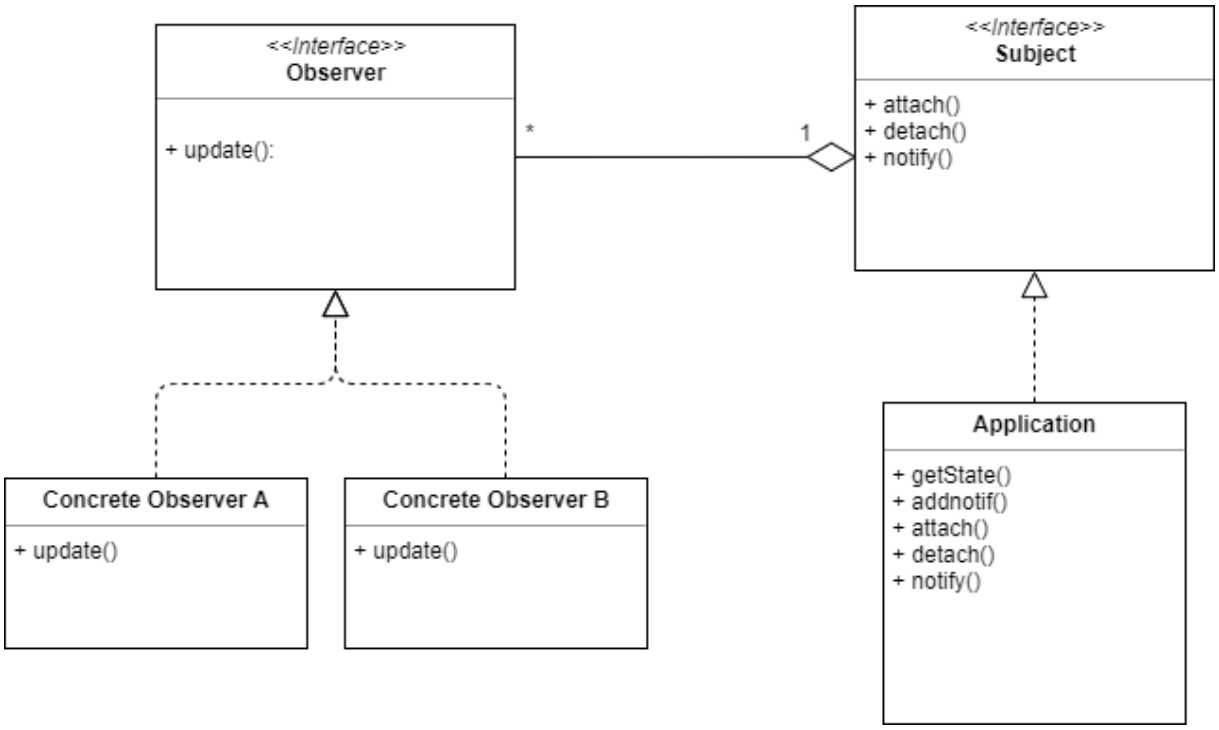


Figure 3.11: Observer pattern in our running example

easily argued that the overall static interaction model emerging from these two patterns would be very similar as the communication between components, though asynchronous in publisher/subscriber pattern, remains the same.

Chapter 4

Related Literature

The basic definition of Software Architecture relies on the definition of the various elements and their interactions. The identification of such elements is performed using the process mining steps, where process discovery has been an exciting research topic during the last few years. Researchers have looked at numerous algorithms and devised modeling techniques for the same, one of which is artifact-centric process model. Artifact-centric models describe the entire business processes by defining artifacts which are smaller business entities having their own lifecycle and information model [11]. These artifacts collaborate and, together with their interactions, represent the overall process model of the organization.

4.1 Artifact-Centric Process Models and Interactions

When we look at the artifact-centric view of business processes, the two most important things that the definition requires are the identification of right artifacts, and the interaction model within and among those artifacts. Nooijen [2] was one of the first who tried to automate the artifact identification from a given set of a relational data source. It is called the Xtract approach, which mainly has two phases, one for artifact schema identification and other for artifact lifecycle discovery. The relational data source schema is used and filtered to generate an artifact schema which further uses existing process discovery techniques to generate the lifecycles of these artifacts. However, the paper does not address any interactions between the various artifacts identified. The convergence and divergence issues related to the event logs are also not addressed. Discovering lifecycles of artifacts has been an interesting research approach, such as one by Popova [27], which uses the Guard-Stage-Milestone (GSM) notation to represent relevant identified artifacts. Again, the interactions between artifacts were not on the agenda of their main research questions.

Lu [3] in her research thesis, tried to automate the discovery of an artifact-centric model for a relational data source and used the Xtract tool and added key manual artifact identification steps within it. The resulting work is an extension of the Xtract tool and a categorization method for interactions at various artifact levels, also implemented as an InteractionMiner plugin in the ProM tool. However, it was realized that manual intervention for artifact identification and interactions relied highly on the expertise of the domain. Following this topic, in her paper [28], a reconstruction of high-level end-to-end business processes using transactional data is performed on two case studies and

helps identify unusual flows in execution. To counter convergence and divergence issues of event logs, it treats one-to-many or many-to-many relations as an interaction between two artifacts, which is one-to-one.

Relying on the relational database for identifying the right artifacts has been a significant interest in the work of earlier researchers. In relational database systems such as ERP systems, data related to a specific business process is usually stored across multiple tables, and tables are connected through key relationships. As pointed out by van der Aalst [29], there are no explicit references related to events and processes instances. Hence there lies a huge challenge in translating the existing raw data from the structure of the table to event log structures. The process mining approach for visualizing software behavior often results in models which are UML diagrams or functional architecture models or other ad-hoc and non-standard formalisms. A metamodel approach to defining the artifact-centric approach to event log extraction was defined by Ana Pajic [30]. An ontological metamodel of artifact centric approaches by employing process mining techniques on event logs on ERP systems was created as part of this research. They conduct a detailed analysis of the artifact-centric approach concepts and describe its constructs by the ontological metamodel. The underlying logical and semantically rich structure of the approach is presented through the model definition. Kaats[4] created a functional architecture model to show how can processes in a collection of (inter-)operating systems be appropriately identified, analyzed, and utilized. It relies on discovering communication between various functional modules using a behavioral matrix. The researchers generate event logs as per modules and features, further creating features and reference nets that are used to generate a functional architecture of a software system. There are also other formal notations and formalisms such as Petri Nets or Finite State Machines (FSMs) used to represent certain aspects of these ad-hoc models. However, we see that majority of the architecture models are represented by semi-formal or informal modeling approaches as they are easier to comprehend for both business and technical stakeholders. Another framework introduced by Lei J. [17] proposes a scalable artifact centric distributed framework ArtiREST based on the REST architectural style. When it comes to cloud-based systems and artifact centric modeling of such system, Calvo's paper [16] provides a different insight by introducing new steps in the artifact centric process modeling and focuses on improving the transformation of raw data into event logs. An API extractor is created by the author for cloud systems to facilitate artifact extraction and is implemented in the third version of the Xtract tool, Xtract V3.

All of the above-mentioned artifact centric modeling approaches rely on the relationships among identified database artifacts generated via event logs. Hence, discovered interactions showcase the dependent communication between artifacts and sometimes their cardinalities, but fails to give detailed interaction design of the communicating artifacts. Studies which do focus on resolving interactions models cover directly or indirectly only one-to-one type of communications.

4.2 Software Execution Data and Process Modeling with interactions

Further studies relied on software design and code data to get the right software architecture, which represents the system. Thus, the dynamic aspects of the system can be looked at via the software execution data, whereas the static information can be viewed by understanding the software architecture structure. Ideally, the implemented architecture of the software is intended to represent the right software architecture of the system. However, various technical architecture decisions get altered during the design and implementation phase, which may or may not cover the intended architecture of the system and can be identified by performing conformance checks. This remodeling of the intended architecture based on the implemented architecture, with the help of software code and execution data, was the basis of numerous research questions.

One such paper which takes the dynamic aspect into notion is by Rooijmans [31]. The paper attempts to capture dynamic aspects of running systems via XES format event logs to construct relevant architectural views and perspectives for stakeholders. It implements an Architectural Intelligence Mining (AIM) framework, which has a parser, an analyzer, and a visualizer to conduct the process mining steps for Software Architecture Reconstruction (SAR). It uses the structure provided by Ducasse [9] which divides the field along five axes namely, goals, processes, inputs, techniques, and outputs, which will also be used in this research to position the SAR approach. For visualization, the paper uses CodeCity, which is a city-like representation of code. It uses classes or methods as buildings to visualize the software system. However, the visualizer faces performance issues when it comes to lower-level abstractions or using other features in CodeCity to represent other metrics of the software system. It is also unable to capture interactions between various elements in the visualizer. Another ProM plugin was created by Jong [6] using which users can create a Hierarchical Interaction Model (HIM) from a log and extract interactions of selected elements. The HIM is based on functional architecture model and shows only the occurring interactions. It does not cover any further details on modeling various interaction scenarios. An instrumentation code, AJPOLog, is also created by the author, which can be used to visualize the run-time structure and behavior of a system without developer effort or knowledge of the source code. Conformance checking is performed on the implemented architecture with the use of the different architectural violations delivered by this research approach. We will refer this paper to implement our running example and automatically generate logs that highlight our use case.

Another research by Liu and Aalst [5] aims to exploit the software execution data to discover behavioral models for each software component. It first identifies component instances and constructs software event logs for each component from the raw software execution data. It then implements a corresponding hierarchical software event log using calling relations among methods. It represents the behavioral model of the components using a Petri net with nested transitions. It also implements the process mining step as an extension in the open-source toolkit ProM. Another research is performed by Eck and Aalst [18]. The research provides ways of visualizing and quantifying interactions among different artifacts via a CSM Miner implemented in ProM. It acts as an interactive artifact-centric process discovery tool focussing on interactions. Although the evaluation is limited and provides only an indication of the usefulness of the approach in practice, it

does establish an unusual analysis step by looking at the measures of interestingness in artifact interactions. The approach is also limited as it only captures one-to-one relations between artifacts. In the paper by Fahland [32], it is pointed out that further research is required on how to handle this interaction complexity, which motivates this research.

An architectural model discovery and design pattern detection approach is presented by Cong Liu [14]. It offers a detailed and complete outlook on the existing component discovery approaches and what are their required inputs and techniques. It further goes on to discover the component behavioral model by using object interaction graphs. This leads to a hierarchical software event log construction capturing the components and their behaviors. It also evaluates the discovery models on four different real-life software systems. Interface identification for discovered components are also done by four different approaches which groups methods differently, that is, (1) with all the methods of the component, (2) with all methods of the same class, (3) with all methods called by the same component, and (4) with all methods called by the same method of another component. When interfaces are identified by grouping methods that are called by the same method of another component, more meaningful interfaces are discovered. It leads to duplicated method problem and is solved in this paper by merging similar interfaces based on a specified user threshold, thus improving flexibility. The visualization of SAR is done using multi-instance Petri nets (MNPs), depicting multiple views for architectural models. The research is also extended to design pattern detection framework along with quality metrics. The Design Pattern Detection Tool is also applied to the Observer pattern, which is relevant to this thesis, to discover and validate it. It successfully discovers the primary candidates, which are the subject, observers, and the notify method sending the notification. The tool, however, fails to discover the update, register, and deregister methods of the Observer pattern as they are hidden within the hierarchy of the class components. Tool support is provided for all features of the research via the ProM toolkit and systematically evaluates both synthetic and real-life software execution data. This research would be vital in aligning our researchs interaction discovery process.

We rely on software static structure for identifying artifacts and software execution data for generating the runtime structure of the interacting artifacts. Interaction discovery approaches, such as the ones discussed above, would be extended to more complex many-to-many interactions to align our use case. Use of software execution data to generate event logs suitable for ProM tool analysis is helpful in detailing the interaction dependencies for better discovery and design.

4.3 Understanding interactions between artifacts

For accurately understanding interactions, we looked at the research by Lorenzoli [15], which generates a model displaying relations between data values and component interactions. The researchers come up with an algorithm called GK-Tail which is used to automatically generate extended finite state machines (EFSMs) using interaction traces. Another research that generates an interaction model is by Lohmann [37]. This paper introduces the concept of agents and locations that adds location information to the global interaction model by defining location-aware artifacts. Adding location information and data values information seems to add more value to a given interaction model. Visual-

Table 4.1: Overview of relevant research studies with their findings

Ref.	**Key Findings**	**Modelling approach**
[3].	Tackle data divergence and convergence problem	Artifact centric process model with extended interactions
[2]	Artifact centric model for ERP systems	Artifact centric model
[27]	Guard-Stage-Milestone type artifact centric modelling	Petri Nets, GSM model
[4]	FAM using process discovery	FAM
[17]	Distributed artifact- centric BPM framework based on REST principles.	ArtiREST
[30]	Ontological metamodel of artifact centric approaches	UML Metamodel
[31]	Architecture model based on architecture mining	UML, Architecture City
[16]	Artifact centric model for cloud systems	-
[33]	Runtime architecture model using artifact centric process mining	Artifact centric model, FAM, social network analysis
[34]	Artifact centric process mining via novel log format and modelling language	Object-Centric Behavioral Constraint (OCBC) model
[14]	Architecture model based on software execution data	Class and object interaction graph, connection model
[5]	PROM plugin to create a process model using component behavior discovery	Component identification graph, behavioral model, interaction graph
[18]	PROM plugin for artifact centric process model and interactions	Process model using sojourn time per trace
[35]	Artifact centric model - UML based	UML class, activity, state machines, BAUML FRAMEWORK
[7]	Artifact centric process model using software execution data	Artifact centric model, Proclets, GSM
[36]	Process model using recursion aware process modelling technique	Process tree
[19]	Process model using an interaction modelling approach	Choreography
[6]	PROM plugin to create a process model using Hierarchical interaction model(HIM)	Petri nets, BPMN, FSM, MSD

ization of such interaction models was done using EFSMs or Petri Nets. It also relies on discovering one-to-one type of interaction within the model. A concrete choreography modeling language called iBPMN is introduced in paper by Decker [19]. To decompose the interactions and design a choreography, it states that realization of commercial transactions through complex interactions and further refining them via more detailed interactions on message exchanges is the key. The language, however, does not address data flow as data visibility, and conflicting views on data add to the problem. We would try to adopt these solutions and issues as part of our research when we implement our interaction discovery algorithm and visualization technique.

In the paper by Fahland and Aalst [11], a simple model is presented for designing interactions in many-to-many type of artifact communication. The researcher shows how data specification in the artifact centric model leads to the discovery of complex many-to-many interactions. It defines the artifact choreography using proclets as a formal model [38], that defines each artifact as a Petri net and defines channels between proclet ports for interaction choreography. Process behaviors having many-to-many interactions was the principal motive of research in the paper by Fahland [7]. The research is based on the foundational concept of dynamic unbounded synchronization, which generalizes the artifact centric choreographies proposed by Lohmann [37] and extends it to many-to-many type of complex interactions. It generates case-identity tokens to validates an instance of the artifact interaction and uses the proclets to model it. We apply the same concept on our running example to showcase its applicability in figure 4.2 in form of the generalized interaction between application and multiple users. We also showcase synchronicity of the same by taking an interaction instance of two users getting notified by the application in figure 4.3. Overall, the research studies that motivated our research approach and goals are shown in the figure 4.1.

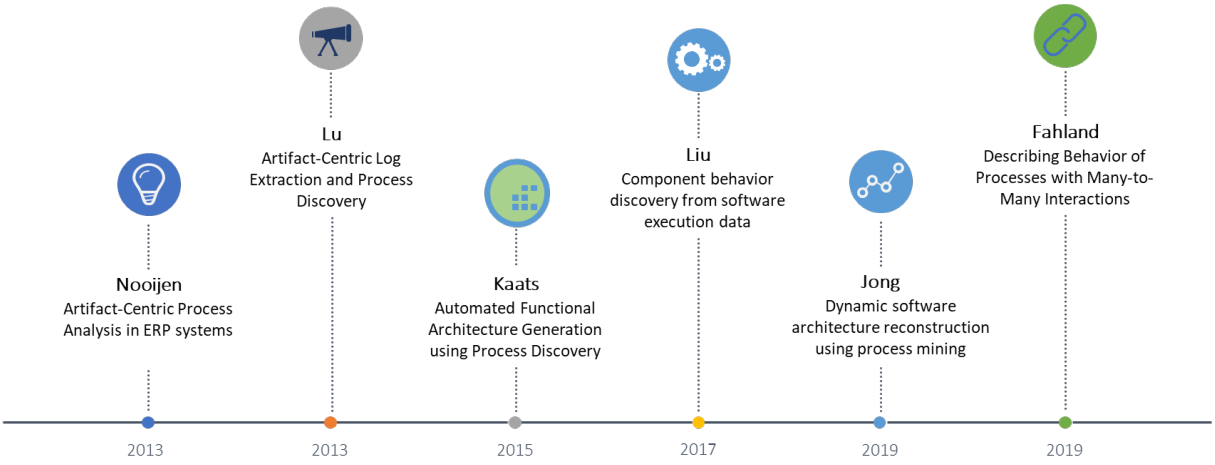


Figure 4.1: Relevant research studies motivating our research goal [2, 3, 4, 5, 6, 7]

A process coordination pattern (PCPs) catalog is provided by Rosado [39], which defines seven types of many-to-many interactions for processes, which are simple succession, concurrent succession, choice, coexistence, synchronization, selective synchronization, and reassignment. It will be used as guidance for evaluating the degree to which existing approaches capture more complex process interactions, such as in the case of our research thesis.

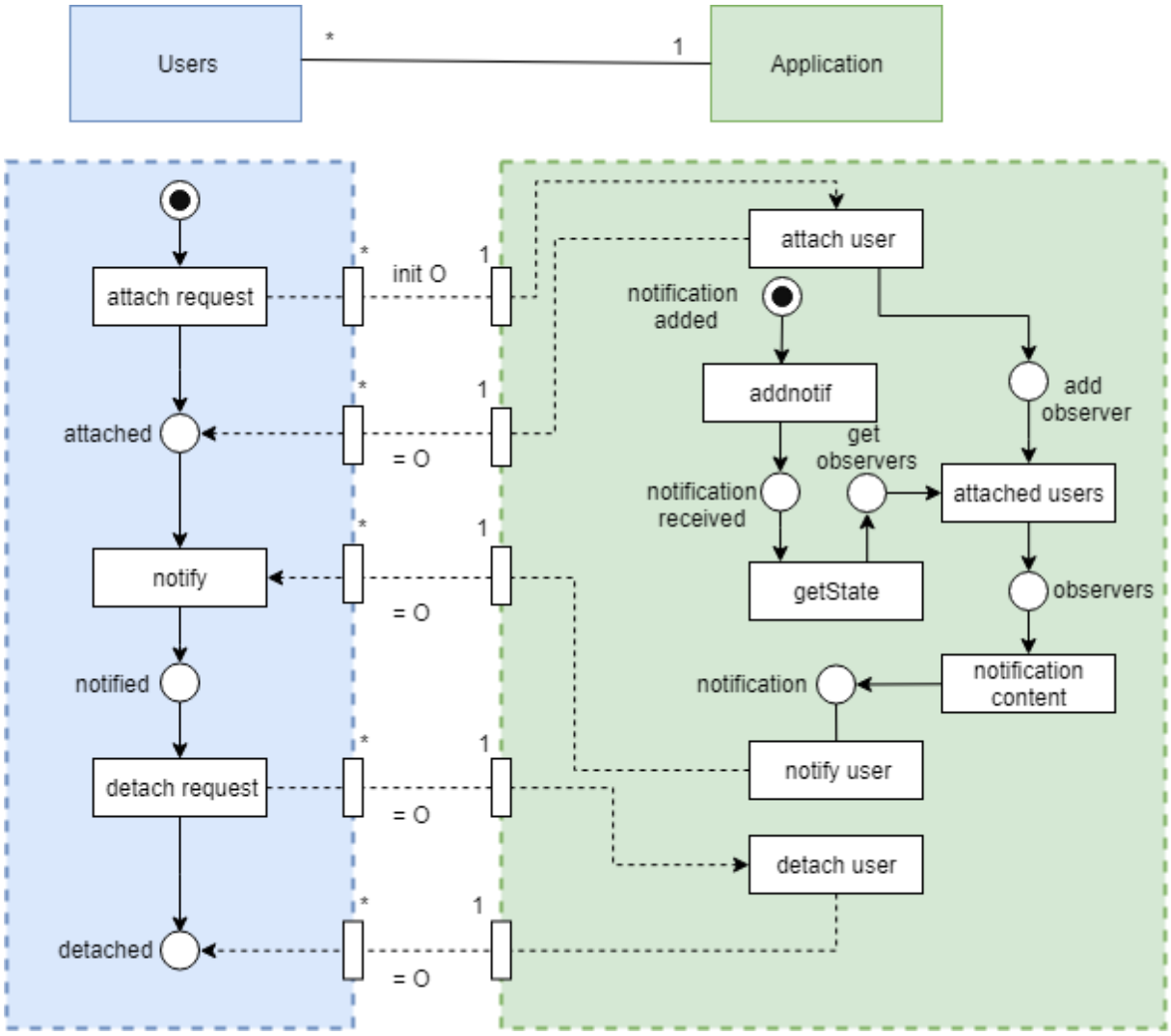


Figure 4.2: Many-to-one Interaction via the extended proclet system

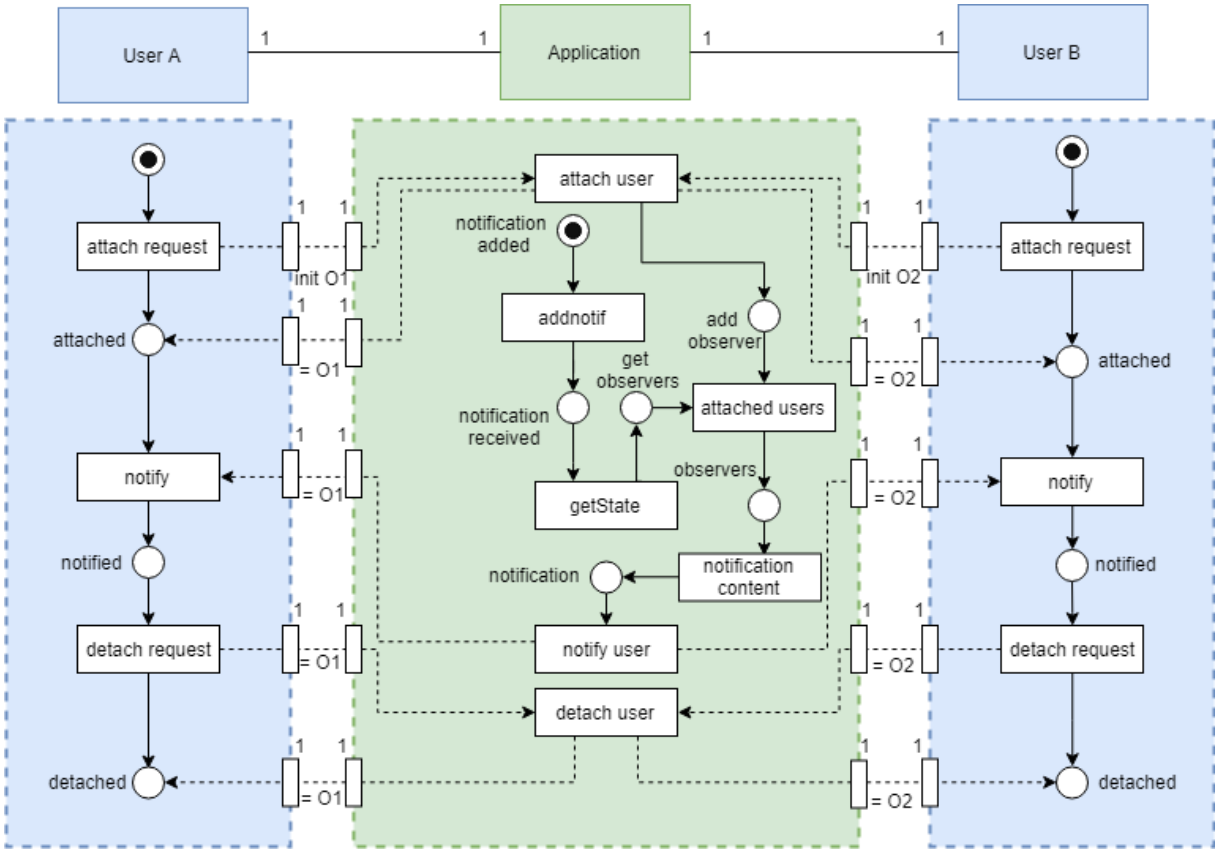


Figure 4.3: Two synchronous users instance of the interaction model using extended proclets system

Chapter 5

Solution Design

In this chapter, we describe our implementation process to tackle our research questions. We first look at the problem statement and further define our implementation steps.

5.1 Overview

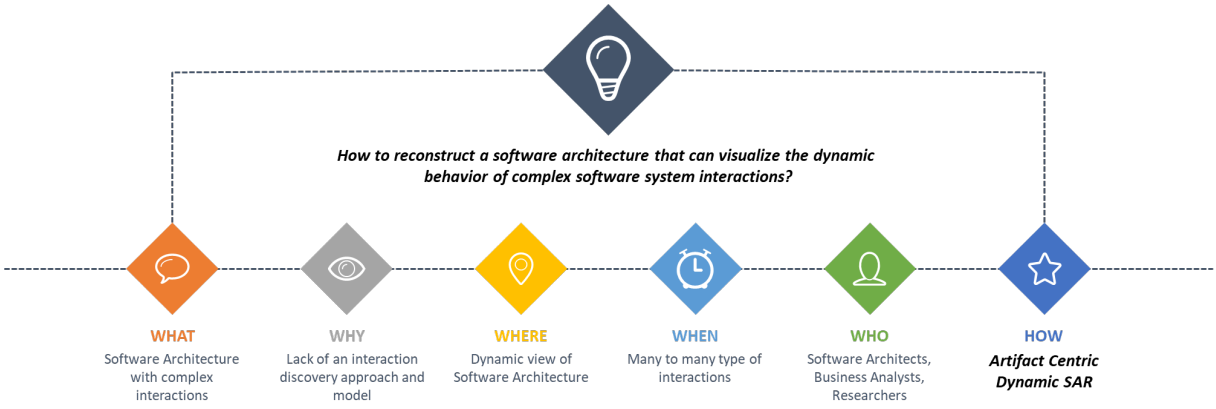


Figure 5.1: Overview of our problem statement

The above figure 5.1 gives a high-level overview of the problem statement. The current techniques and approaches with their shortcomings are highlighted in the previous section. We leverage research provided [14, 6] to dynamically define an event log which can capture information during runtime. For exploring an artifact-centric view of a system, additionally, we also identify artifacts involved in the overall software architecture in the event log. All these event log definitions are part of the first step of our implementation. Further, we try to understand the presence of complex interactions in the software system via a dynamic interaction matrix. Identifying one-to-many, many-to-one, or many-to-many communications within artifacts is the primary purpose of the next step.

Given the presence of complex interaction identified from the previous step, we capture multiple traces of the systems dynamic behaviour and create projections of the event log as per application sessions to leverage inter-session artifact interactions. We further introduce the next step to generate the artifact lifecycle model for each artifact by clustering session-specific events capturing the incoming and outgoing event messages. This step tries to add more clarity to interactions between artifacts that portray anomalous

behaviour. Further, the overall artifact lifecycle model is constructed, combining all the identified artifact lifecycles, which shows the overall dynamic behaviour of the system. This is then abstracted to highlight the interaction model in the artifact lifecycle model using the above two steps. Finally, we generate a software architecture from the artifact-centric process model. The detailed steps are as mentioned in detail in the following section.

5.2 Interaction Discovery Algorithm

This section gives the detailed steps involved in the interaction discovery approach, along with the modelling techniques used. We first start by defining how we can record the dynamic information from the system. Depending on the programming language used for the software application under observation, objects such as method or functions can be used to trace the software behaviour. One can record the entry and exit timestamps when an object instance passes through a method or a function. This information, along with a few other details, can help generate a dynamic overview of an application.

5.2.1 Event Log Generation

The first challenge is to correctly generate an event log that captures interaction messages between the various part of the software. This interaction must be captured in such a way that it gives the object instance information along with its source location of the artifact.

To capture system's artifacts, we define them such that every class within the source code acts as an artifact. Firstly, let ar denote the unique artifacts within the system such that,

$$Ar = \{ar_1, ar_2, ar_3, ..., ar_n\}, \text{ containing set of unique artifacts.}$$

Let m denote the unique event interaction messages captured in our event log such that,

$$M = \{m_1, m_2, m_3, ..., m_n\}, \text{ containing set of unique event interaction messages between artifact instances.}$$

Finally, log L contains an event interaction, of tuple E, having the set of event log entries, $e_1, e_2, e_3, ..., e_n$ with event messages M, and artifact Ar from a running application in chronological order such that,

$$E = (Ar \times M \times Ar), \text{ for every event interaction } e,$$
$$\&$$
$$L = \{E^*\}$$

We can refer to each event entry, e by Π where π_1 denote the artifact Ar with respect to caller artifact(source of the interaction event), π_2 denote the event message M, and π_3 denote the artifact Ar with respect to callee artifact (method being called).

Within the event log, temporal and location information is stored by capturing the timestamp (start and end) of the event, the Caller or the source method for the event call, and the Callee or the destination method or the actual method which implements a given task. This source and destination information are captured in the form of artifacts and its instance information.

For capturing the dynamic information when executing multiple runs of the system, we capture the instance information in terms of $object_{ID}$ of an event. This $object_{ID}$, for instance, can simply refer to a unique user session, and the interaction that occurs between the user and the application maintains this $object_{ID}$ throughout the user artifacts lifecycle. The reason we do this is to capture the complex multi-instance scenarios for an object from the same artifact with other artifacts. We know that such interactions are the subjects of our study and creating session-based scenarios for the system can help us understand them clearly. For example, in case of an order to invoice scenario, two separate purchase orders when placed with a supplier can ships the materials together in a single receipt. For understanding the use case of the receiving process, the interaction model between multiple order artifact instances and the corresponding receipt artifacts needs to be understood for modelling interactions capturing a successful receiving process use case. Hence, in such a case of an ERP process, order artifact can log the unique instance information, which will be its unique order ID.

In essence, we generate our event log L with the following information,

- **Start Date:** Captures the start date and time for the event
- **End Date:** Captures the end date and time for the event

  We capture the timestamps in the standard *DD:MM:YYYY:hh:mm:sss* format where,
  YYYY = four-digit year
  MM = two-digit month (01=January, etc.)
  DD = two-digit day of month (01 through 31)
  hh = two digits of hour (00 through 23) (am/pm NOT allowed)
  mm = two digits of minute (00 through 59)
  ss = two digits of second (00 through 59)
  s = one or more digits representing a decimal fraction of a second
  Using the entry and the exit timestamps of an event call, we list the events occurring chronologically, where each event entry is a tuple of type E.

- **Events** (M)**:** The unique event messages in our running example mentioned below are captured in our event log as event messages
- **Caller and Callee Artifacts** (Ar)**:** The artifacts source and destination information are stored here as an extension of it's artifact's instances. The actual method from the artifact involved is captured, along with the instance information as $object_{ID}$. Since the method information gets covered via event messages column, we wouldn't need to refer to this field for method related information, but only the $object_{ID}$ information. $Object_{ID}$ is captured in the event log by manually injecting an auto-generated unique session ID along suffixed with the artifact information. That is, the $object_{ID}$ is unique for each interaction and its life cycle. The interactions where one session interacts with another session via the application, we

capture the session details in their caller/callee information using this unique ID. The total number of $object_{ID}s$ to be used can be varied across various scenarios of the software application.

We also must define a successful trace which can help filter the unnecessary information from our event log. This way the unsuccessful traces can be removed from the event log and the remaining traces capture the intended dynamic behaviour of the system resulting in the event log L. Every software architecture must represent the overall use case of the application that helps in the understanding of the software system. We capture these use cases explicitly by solely capturing successful traces of events within our event log L. Multiple instances of these use cases are captured as the unique $object_{ID}s$, that we store for every interaction event.

5.2.2 Dynamic Interaction Behaviour Matrix

In the previous step, we saw how we generate an event log by capturing the applications dynamic information. The next step is to identify the dynamic behaviour rightly in a particular interaction event. This is achieved by the construction of an interaction behaviour matrix. We create an artifact events matrix capturing the total number of instance calls for every interaction between caller and callees within the event log L. Let Φ denote the total number of instance calls for such an interaction.

We define Φ as the total number of unique $object_{ID}s$ involved for a specific interaction between π_1 and π_3 for an event e in the event log. A matrix, I is generated with caller artifacts (π_1) against callee artifacts (π_3) with their interactions showing their respective Φ values.

Creating this matrix helps in understanding whether one needs to capture the one-to-many, many-to-one or, many-to-many type from the event log or not. We show many to many interaction types occurring between artifacts for every interaction by comparing the Φ values with the number of sessions, or simply the total number of unique $object_{ID}s$ used. The presence of complex behaviour can be known between artifacts interactions if their Φ values are greater than the total number of unique $object_{ID}s$ for a given event log L. We can generate a high-level idea of the interaction type involved during execution, especially if the event log captures the complex many-to-many type of interaction. If the Φ values in the above matrix did not indicate any such complex behaviour, one could choose to design the interaction model simply by taking the standard one-to-one type of communication projection in the next step.

Next, we can identify our interaction nodes. We do this by checking the interactions in the matrix having values of $\Phi \neq null$. For every such interaction, we create interaction nodes C_k, where every node is an interaction between any two unique artifacts, Ar. The various event messages involved in those interactions between the two artifacts are assigned to the respective node. That is, each interaction node, C_k, we will have a pair of unique artifacts interacting with a set of event messages that are used in communication between the two artifacts, representing the node.

5.2.3 Artifact Projection Log

To get to the projection of specific session instances, we project the event log generated to capture simply the interactions that occur during the lifecycle of the artifact object

session. The various events that are involved in the interaction between two artifacts and their order of occurrence, for a given session can help in the definition of artifacts lifecycle.

The direct projection set event log P is defined as,

$$P \in L$$

where the event tuples E within the log, are segregated as per every unique session instance as,

$$E_i = (Ar \times M \times Ar), \text{ where } i \text{ is the unique } object_{ID}.$$

i.e., events traces occurring during the interaction of the specific artifact session instance ID, listed chronologically.

This projection filters the activities performed by the artifacts during a specific session and is obtained by executing a single session at a time. Thus, the projection event log is a subset of the event log L, but captures the information that occurs for a specific object instance during its interaction lifecycle. This is achieved by simply capturing every session in sequential order in isolation from the other sessions, such that the other sessions log information is not part of the projected event log P_i.

Multiple projections are created for an artifact with unique event sequences based on the number of unique $object_{ID}s$ used. This will further help us while projecting the interaction model from the artifacts lifecycle model. Two projections can also eventually result in the same event set, and to avoid duplicity, we further consider only the unique projection logs. We use these artifact session logs to show the abstracted artifact interaction events flow. Also based on some critical observation, we can identify certain events from the logs to be sequential or parallel; if the said event follows a specific sequence or can occur anytime during the execution.

We further abstract the object instances to showcase the actual artifact / artifact entity involved in the interaction. We can ignore the artifact $object_{ID}$ to create a generic artifact interaction set, and their corresponding interactions from the generalized projection log. We can use this set to create a high-level container model between artifacts. This model simply consists of the artifacts, Ar, with the interaction flows from one artifact to another. These interactions are denoted by the $nodes : C_1, C_2, C_3, C_4, C_5, ...$, and so on, as from the last step. Our goal is to highlight further these interactions with respect to multiple instance scenarios, parallel to the lifecycle of the artifacts involved in such interactions.

5.2.4 Artifact Lifecycle Model

In the previous step we saw how for an artifact's session instance i, P_i projection shows the artifact lifecycle with respect to the artifact directly initiated outgoing interactions and events that lead to certain incoming interactions. Using the above projection event logs, we can construct our artifact lifecycle for each artifact separately by taking any one of the projections P from the previous step and clustering the events to capture a log Pr_i^j such that,

$$Pr_i^j \subset P_i, \text{ containing event tuples consisting of an artifact, } ar_j$$

The projection P_i shows the artifact lifecycle with respect to a single session instance and hence, the lifecycle generated only showcases the artifact lifecycle with respect to other artifacts. Thus, we can see how a single artifact session interacts with cardinalities one-to-one and one-to-many behaviour for a single object session. We can use this information to generate the interaction protocol model for each identified communication node, along with the cardinalities. However, we simply identify the one-to-one and one-to-many interactions here as Pr_i^j would not rightly capture any many-to-many interactions at this level as a single object session is considered.

We can use any of the projection sets belonging to a specific session. The reason we can use any of the session projection sets here to define the node protocols is that the various sessions only showcase the differences when it comes to the artifact's lifecycle, and not the interaction protocol itself. Thus, we use this projection set to generate the nodes interaction protocol for each unique pair of artifacts from a projection set, as long as we cover all artifact interaction messages. These interaction protocols are the one-to-one type of interactions between artifacts. We can use the following steps for designing the node protocol choreographies,

Interaction Node Choreographies Algorithm

- For each π_1 and π_3 value pair for an event e, within a chosen session projection set for an artifact ar_j, Pr_i^j
  - For every unique pair of artifacts for an interaction node C_k, apply the standard discovery algorithm on the event tuple e, where activities are drawn as choreographies.
  - Design each choreography element using the standard BPMN 2.0 notation, in the sequence they occur, where for every event, π_2 is the choreography task, and π_1 and π_3 are the initiating and non-initiating participants respectively.
  - The choreography diagram must only model the flow of the interactions and not multiplicities. Gateways can be used to model synchronicity of the flows in case of conditional interaction flows.
  - Generate choreography diagram similarly for all available unique pairs representing an interaction node.
- Move through various artifacts ar_j and through various session projections, Pr_i^j, until all nodes C_k are modelled.

Further, to design the artifact's lifecycle, we can create a generalized projection set for every artifact. We use the following algorithm steps for designing a generalized artifact's lifecycle projection set,

Generalized Artifact Projection Set Algorithm

- For an artifact ar_j, consider a base projection log by choosing any of the above set of logs from Pr_i^j, say the first set, Pr_1^j .

- Mark all interaction edges with an initial flag status as *unchecked* in the base projection log.
- Create an empty generalized set, Pr_g^j to capture the lifecycle of an artifact ar_j.
- For every interaction within the base log, Pr_1^j, find the next occurring interaction with a different event and store them both as the first two edges in a temporary generalized set.
- If the edge sequence occurs in all the projection sets, Pr_i^j, for an artifact ar_j check,
  - If the interactions always follow each other in every Pr_i^j, then add it to the finalized generalized set, Pr_g^j. Update both the interaction event status flag to *checked*.
  - If the interaction does not follow each other in every Pr_i^j, skip and choose the next two interactions from the base projection set, Pr_1^j.
  - Repeat the above two steps until the last interaction in the base log is reached.
  - Mark *unckecked* status interactions as a parallel event as it occurs randomly within the artifact's lifecycle, with transitions event point as m_p, going through the first and last occurrence of the parallel interaction event considering all projections.
- The generalized set depicts the corresponding artifact's lifecycle, and model it accordingly using Petri nets formalism [38].
- Further, model the high-level interaction flows and its cardinalities with respect to the artifact's lifecycle and its interactions using the node protocols such that,
  - If an event occurs once in any of the projection sets, Pr_i^j, model an simple directional arc from the callee artifact event, through the choreography event interaction, and to the caller artifact, with the cardinality of the callee artifact as one-to-one.
  - If an event occurs multiple times in any of the projection sets, Pr_i^j, model an simple directional arc from the callee artifact event, through the choreography event interaction, and to the caller artifact, with the cardinality of the callee artifact as one-to-many.

We model the generalized projection set, Pr_g^j, for each artifact ar_j by applying the generalized algorithm. The individual artifact lifecycle is extended with the node choreographies by joining them using the directional arcs, as specified in the algorithm. The generic representation shown in figure 5.2 shows how an artifact connects to its choreography nodes. The interaction nodes are shown as $C_1, C_2, C_3, ..$ and so on for each interaction message for an artifact, say ar_i. Each interaction node choreography connects with the

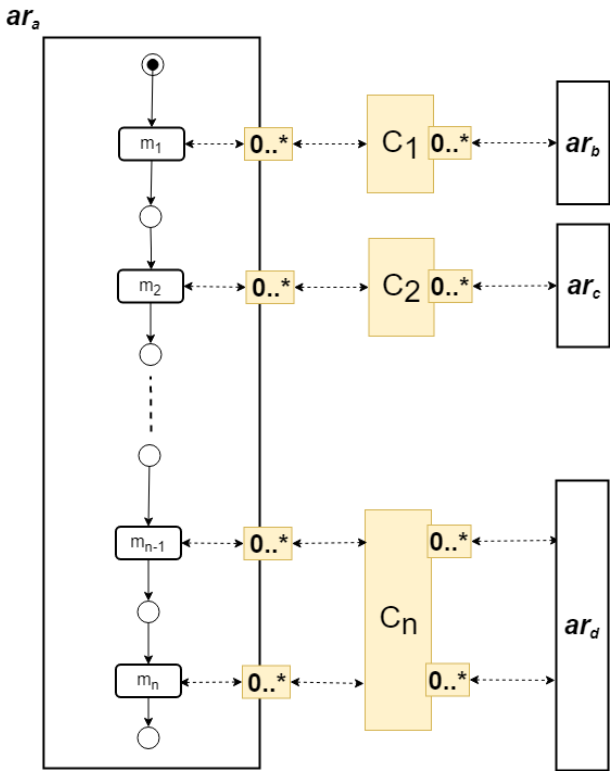


Figure 5.2: Generalized Artifact Lifecycle Model - Generic View

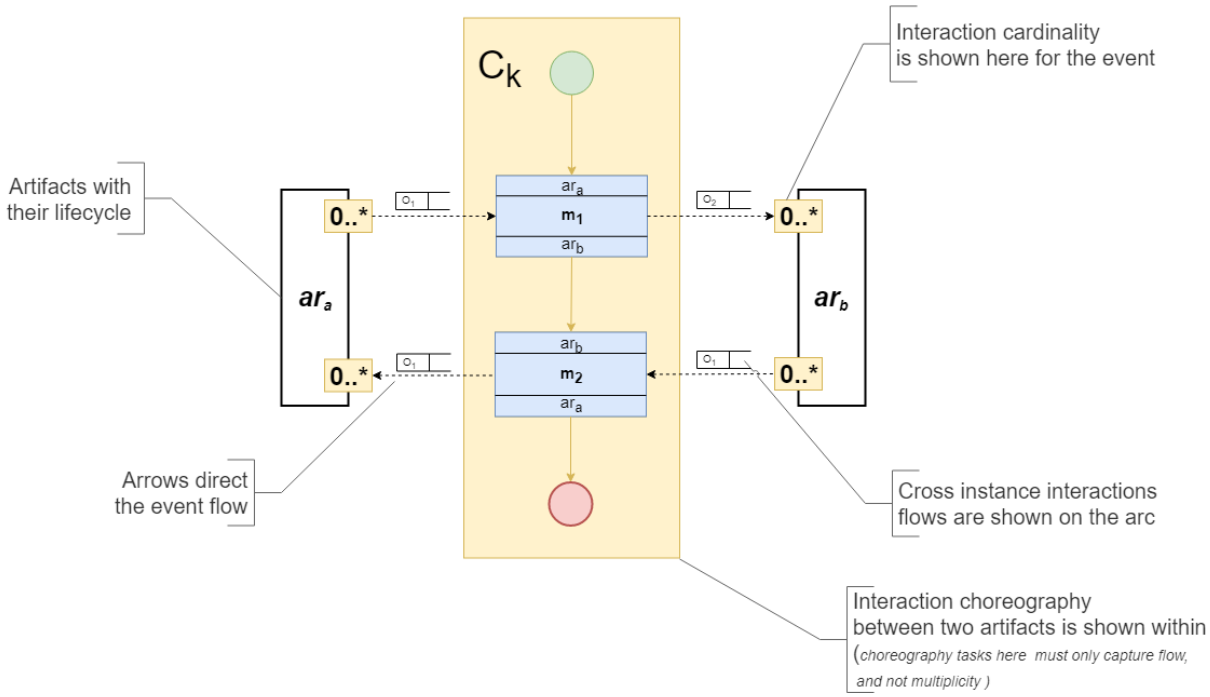


Figure 5.3: Node Choreography representation with Artifact Lifecycle Model - Generic View

artifact's interaction as shown in figure 5.3. It shows how each event within an artifacts lifecycle can have multiple node choreographies associated with multiple artifacts. The

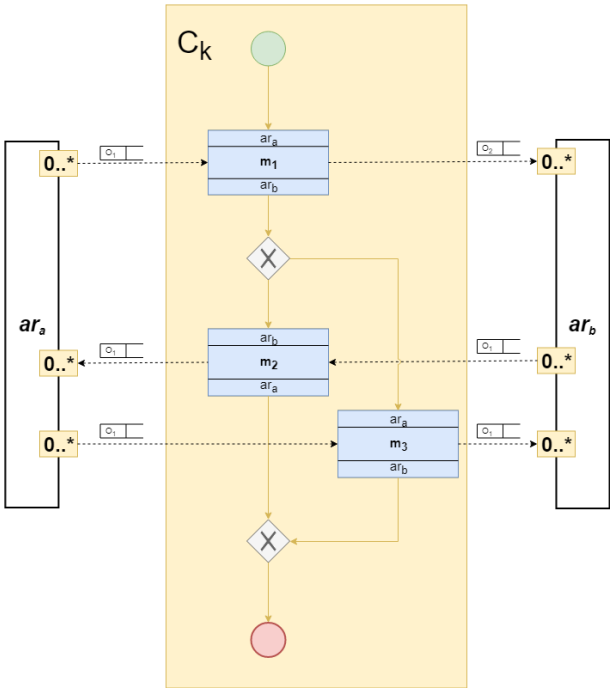


Figure 5.4: Node Choreography representation with Artifact Lifecycle Model - with choreography gateways

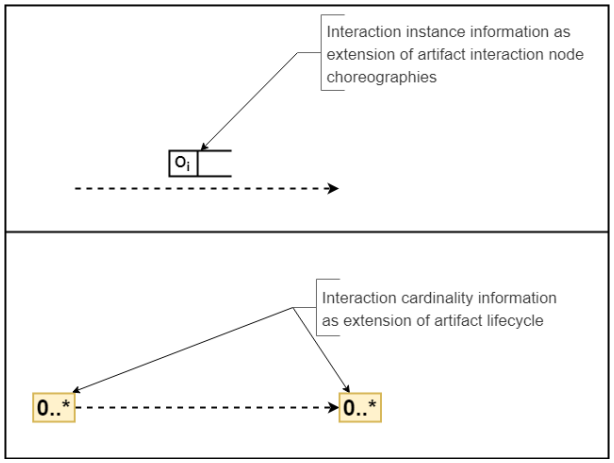


Figure 5.5: Notations on connecting arcs

arcs connecting the artifact lifecycle's events with the corresponding interaction events also have notations for showing cardinalities and instance interaction information, as shown in figure 5.5. It shows how two artifacts and their interactions for two separate events (as choreography tasks) with their flow. The cardinal multiplicities are shown for an interaction at the place where the corresponding artifacts are connected. The figure 5.4 shows another node choreography where the choreography tasks have a certain conditional gateway in between interactions fetched from different sessions.

Once all the artifact lifecycles are modelled using the generalized projection set, the artifacts can be combined to get the overall artifact view. The overall high-level interaction model of the system for a specific use case was generated using the individual artifact lifecycle obtained earlier. We then extend our high-level model showcasing the

lifecycles of each artifacts and with interactions between two artifacts denoted by the node choreographies. The cardinalities for the interactions generated can also be showcased in the overall model by a simple extension of cardinalities from the high-level contextual model and projection logs from the previous step, to a high-level artifact centric model. The interaction cardinalities for the artifact used for a specific $object_{ID}s$ instance, can be showcased with the right values. One-to-many behaviour is highlighted if an event occurs multiple times for the considered single session. However, the interaction type does not give details related to the lifecycle of the interaction at this level completely. The various interaction dependencies that occur for a many-to-many scenario still remains uncovered. Hence, we capture it in the next step to specifically target the artifact centric projections covering many-to-many interactions, revealing its dependencies and overall process.

5.2.5 Overall Artifact Model

In this step, we take the artifact model and projection logs we generated with simple one-to-one interactions and extend it to many-to-many interactions with the help of the event log interactions we observed for a single artifact covering the entire generated event log L during multiple sessions.

Now to get the interaction model in this case with respect to each artifact, we classify the log into artifact specific tables as we did for a single object session such that,

$$Pr^j \subset L, \text{ containing event tuples consisting of an artifact, } ar_j$$

Now for each projection set Pr^j, we check the number of times the artifact ar_j interacts with another artifact for an event tuple with the same interaction event message M. We associate the interaction cardinalities as follows,

- If an event message M occurs multiple times for a pair of unique interacting artifacts, and multiple times for a unique instance $object_{ID}$, we associate the interaction as many-to-many type.
- If an event message M occurs multiple times for a pair of unique interacting artifacts, and once for a unique instance $object_{ID}$, we associate the interaction as many-to-one type.
- If an event message M occurs once for a pair of unique interacting artifacts, and once for a unique instance $object_{ID}$, we associate the interaction as one-to-one type.

We can now update the generated overall artifact model from the previous step with the right interaction cardinalities. This model now represents the overall interaction model of the system with respect to various artifacts within the software system. Viewing the interaction choreographies with respect to an artifact helps in understanding the interaction lifecycle with respect to the lifecycle of the chosen artifact. The view can thus be toggled with the same logic to cater interaction view of any of the artifacts by applying the above-extended projection log sets.

To realise the multi-instance interactions, we check each event entry for cross instance communication in the above projection set. If the instance ID for the session events differs for caller and callee artifacts, the interaction node can be marked with an initialization code accordingly. That is, for an event tuple e, if the $object_{ID}(\pi_1) \neq object_{ID}(\pi_3)$, the

corresponding interaction node can be marked with initialization nodes $O1, O2$, denoting multi-instance interaction. Other nodes without such complex multi-instance interactions can be denoted just as $O1$. Note that, use of initialization nodes in the model should only be when there exists a cross instance interaction between two artifacts.

5.2.6 Constructing Software Architecture

In order to construct the software architecture using the above-constructed artifacts view and the interaction centric view, we leverage the C4 notation for abstracting the architecture at various levels [40]. The C4 notation model was created as a way to guide and help software development teams describe and communicate software architecture. It considers the static structures of a software system in terms of containers, components and code and users interacting with the system. We use the first three levels of the modelling notation to showcase the static containers and components, which in our case are artifacts. The code level is expanded to include the code level interactions between artifacts, along with its instance-specific details modelled together with the lifecycle of a chosen artifact.

The context view shows an overview of the system with respect to the use case. This view can be further instantiated to container level showing the high-level overview of the system, designed as per the identified artifacts involved. We can also view the actual components or in our case artifacts involved at the component level of the architecture as the model generated in our artifact projection set definitions in Step 3. The container and component levels of abstraction differ in the functional and technical level of artifact definitions. An artifact identified at the container level can further contain multiple artifacts within it, determined at the component level of abstraction. Finally, to showcase the code level view by displaying the overall artifact model identified in the previous steps with their interaction denoted by node choreographies generated in the earlier stages of our solution. The modelling approach relies on the overview first, where a user can then zoom in to display details-on-demand.

Software Architecture can be viewed by displaying one artifacts lifecycle at a time to minimize model visual complexity. The resulting model can also show the node protocols only when zoomed in to minimize the code view size.

Chapter 6

Case Study: Notification Application

In this chapter, we take our running example, the notification application, and our implement our solution design to show its feasibility and use. The steps mentioned in the previous chapter, 5 are applied to our example case study.

6.1 Event Log

In our running example, we generate our event log E with the following information,

- **Start Date:** Captures the start date and time for the event
- **End Date:** Captures the end date and time for the event

  We capture the timestamps in the standard *DD:MM:YYYY:hh:mm:sss* format where,
  YYYY = four-digit year
  MM = two-digit month (01=January, etc.)
  DD = two-digit day of month (01 through 31)
  hh = two digits of hour (00 through 23) (am/pm NOT allowed)
  mm = two digits of minute (00 through 59)
  ss = two digits of second (00 through 59)
  s = one or more digits representing a decimal fraction of a second

- **Events** (M)**:** The unique event messages in our running example mentioned below are captured in our event log as event messages
  - m_1: attach_req
  - m_2: attach_res
  - m_3: add_notif
  - m_4: get_observers
  - m_5: notify
  - m_6: detach_req
  - m_7: detach_res
  - m_8: activity

That is, our event messages set $M = \{m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8\}$.

- **Caller and Callee Artifacts (Ar):** The main artifacts of our running example are listed below. We do not capture this information in our event log directly. However, the source and destination information is an extension of these artifact's instances, where we also include the actual method from the artifact involved, along with the $object_{ID}$. Since the method information gets covered via event messages column, we wouldn't need to refer to this field for method related information.

  - ar_1: User - User module of the application
  - ar_2: Application_Notification - contains the main content and notify method
  - ar_3: Application_User_Mananger - contains methods for attaching, detach and getting list of attached users
  - ar_4: Application_Activity_Feed - contains methods for adding a notification content and checking for new notifications for notifying users

  That is, our artifacts set $Ar = \{ar_1, ar_2, ar_3, ar_4\}$.

  $Object_{ID}$ is captured in the event log by manually injecting an auto-generated unique user session ID along suffixed with the artifact information. That is, the $object_{ID}$ is unique for each users interaction and its life cycle. The interactions where one user session interacts with another user session via the application, capture the user session details in their caller/callee information using this unique ID. The reason we do this is to capture the complex multi-instance scenarios for an object from the same user artifact with other artifacts.

In our case, a successful trace is considered as an attached user getting notified by the application (with available notification activity content) before it detaches. We ignore the other unsuccessful traces, and our event log only captures the successful traces of the application in execution.
For simplicity, let's assume the $Object_{ID} = \_i$, where i = 1,2,3,, and so on, denoting the user session instance for the run. In our running example, the $object_{ID}$ is merged with the caller/callee artifacts information itself. However, we showcase the $object_{ID}$ in a separate column for clarity of understanding. The table 6.1 shows the event log entries for three user sessions interacting with the application, chronologically. We will be using this table as our main event log L to implement the solution design proposed in the previous chapter.

6.2 Dynamic Interaction Behaviour Matrix

We design the matrix, I where we capture the total calls for each event, Φ, between callers and callee artifacts for this step. In the running example sample event log above, we capture three different sessions, each having a unique $object_{ID}$s. From the event log, we calculate the Φ values for each interaction between artifacts to generate the following table, 6.2.

We see that the interaction between user and activity feed artifacts (ar_1 and ar_4), activity feed and the notification artifacts (ar_4 and ar_2), and user and the notification

Table 6.1: Event log sequence (Example of 3 users interacting with the application: event ID, callers, callees and message)

Event_ID (E)	**Caller**		**Callee**		**Message** (M)
	Class (Ar)	$Object_{ID}$	**Class** (Ar)	$Object_{ID}$	
e_1	ar_1	1	ar_3	1	m_1
e_2	ar_3	1	ar_1	1	m_2
e_3	ar_1	1	ar_4	2	m_8
e_4	ar_4	2	ar_2	2	m_3
e_5	ar_1	1	ar_4	3	m_8
e_6	ar_4	3	ar_2	3	m_3
e_7	ar_1	2	ar_3	2	m_1
e_8	ar_3	2	ar_1	2	m_2
e_9	ar_2	2	ar_3	2	m_4
e_{10}	ar_2	2	ar_1	2	m_5
e_{11}	ar_1	2	ar_4	1	m_8
e_{12}	ar_4	1	ar_2	1	m_3
e_{13}	ar_2	1	ar_3	1	m_4
e_{14}	ar_2	1	ar_1	1	m_5
e_{15}	ar_1	1	ar_3	1	m_6
e_{16}	ar_3	1	ar_1	1	m_7
e_{17}	ar_1	2	ar_3	2	m_6
e_{18}	ar_3	2	ar_1	2	m_7
e_{19}	ar_1	2	ar_4	3	m_8
e_{20}	ar_4	3	ar_2	3	m_3
e_{21}	ar_1	3	ar_3	3	m_1
e_{22}	ar_3	3	ar_1	3	m_2
e_{23}	ar_2	3	ar_3	3	m_4
e_{24}	ar_2	3	ar_1	3	m_5
e_{25}	ar_2	3	ar_3	3	m_4
e_{26}	ar_2	3	ar_1	3	m_5
e_{27}	ar_1	3	ar_3	3	m_6
e_{28}	ar_3	3	ar_1	3	m_7

Table 6.2: Interaction Behavior Matrix for 3 users

Interaction Behavior Matrix		***Callee***							
		$ar_1 : m_2$	$ar_1 : m_5$	$ar_1 : m_7$	$ar_2 : m_3$	$ar_3 : m_1$	$ar_3 : m_4$	$ar_3 : m_6$	$ar_4 : m_8$
Caller	$ar_1 : m_1$	0	0	0	0	***3***	0	0	0
	$ar_1 : m_8$	0	0	0	0	0	0	0	***4***
	$ar_1 : m_6$	0	0	0	0	0	0	***3***	0
	$ar_2 : m_4$	0	0	0	0	0	***3***	0	0
	$ar_2 : m_5$	0	***4***	0	0	0	0	0	0
	$ar_3 : m_2$	***3***	0	0	0	0	0	0	0
	$ar_3 : m_7$	0	0	***3***	0	0	0	0	0
	$ar_4 : m_3$	0	0	0	***4***	0	0	0	0

artifacts (ar_1 and ar_2) are many-to-many type as the Φ is more than the number of artifact objects instances (3). For example, a user can perform multiple activities, and multiple notifications are triggered for notification relevant activity. We can also see how communication between user and user manager artifacts (ar_1 and ar_3) are of the one-to-many type where many users attach and detach using the user manager artifact just once.

We can then generate the communication protocol between two artifacts referred as simply interaction nodes. This is done by mapping the events for each interaction in the high-level model, to a node between any two artifacts within the system. In our example,

- C_1: m8
- C_2: m1, m2, m6, m7
- C_3: m5
- C_4: m4
- C_5: m3

6.3 Artifact Projection Log

We apply the artifact projection logs step for our notification application generated log. This is done by running and capturing the unique user sessions in isolation, one at a time. We encounter the following unique projection log sets for every user artifact instance.

Projection log P_1 shows interactions when a user instance with $object_{ID} = 1$ interacts with the application. The logs show events in chronological order in the table 6.3.

Table 6.3: Projection log, P_1 when instance with object_ID = 1 interacts with the application

Event_ID (E)	**Caller**		**Callee**		**Message** (M)
	Class (Ar)	$Object_{ID}$	**Class** (Ar)	$Object_{ID}$	
e_1	ar_1	1	ar_3	1	m_1
e_2	ar_3	1	ar_1	1	m_2
e_3	ar_1	1	ar_4	2	m_8
e_4	ar_4	2	ar_2	2	m_3
e_5	ar_1	1	ar_4	3	m_8
e_6	ar_4	3	ar_2	3	m_3
e_{11}	ar_1	2	ar_4	1	m_8
e_{12}	ar_4	1	ar_2	1	m_3
e_{13}	ar_2	1	ar_3	1	m_4
e_{14}	ar_2	1	ar_1	1	m_5
e_{15}	ar_1	1	ar_3	1	m_6
e_{16}	ar_3	1	ar_1	1	m_7

Similarly, we can project the event logs to create projection sets for two more sessions, as shown in table 6.4 and table 6.5. The behaviour of these sessions is manually controlled

in our running application such that the overall trace of the event log and each projection set follow the same user session behaviours and perform the same use case.

Table 6.4: Projection log, P_2 when instance with object_ID = 2 interacts with the application

Event_ID (E)	**Caller**		**Callee**		**Message** (M)
	Class (Ar)	$Object_{ID}$	**Class** (Ar)	$Object_{ID}$	
e_3	ar_1	1	ar_4	2	m_8
e_4	ar_4	2	ar_2	2	m_3
e_7	ar_1	2	ar_3	2	m_1
e_8	ar_3	2	ar_1	2	m_2
e_9	ar_2	2	ar_3	2	m_4
e_{10}	ar_2	2	ar_1	2	m_5
e_{11}	ar_1	2	ar_4	1	m_8
e_{12}	ar_4	1	ar_2	1	m_3
e_{17}	ar_1	2	ar_3	2	m_6
e_{18}	ar_3	2	ar_1	2	m_7
e_{19}	ar_1	2	ar_4	3	m_8
e_{20}	ar_4	3	ar_2	3	m_3

Table 6.5: Projection log, P_3 when instance with object_ID = 3 interacts with the application

Event_ID (E)	**Caller**		**Callee**		**Message** (M)
	Class (Ar)	$Object_{ID}$	**Class** (Ar)	$Object_{ID}$	
e_5	ar_1	1	ar_4	3	m_8
e_6	ar_4	3	ar_2	3	m_3
e_{19}	ar_1	2	ar_4	3	m_8
e_{20}	ar_4	3	ar_2	3	m_3
e_{21}	ar_1	3	ar_3	3	m_1
e_{22}	ar_3	3	ar_1	3	m_2
e_{23}	ar_2	3	ar_3	3	m_4
e_{24}	ar_2	3	ar_1	3	m_5
e_{25}	ar_2	3	ar_3	3	m_4
e_{26}	ar_2	3	ar_1	3	m_5
e_{27}	ar_1	3	ar_3	3	m_6
e_{28}	ar_3	3	ar_1	3	m_7

We can use this trace to create a high-level communication model between artifacts as shown in figure 6.1. The model shows the main artifacts of the notification system, along with the interaction flows from one artifact to another. These interactions are denoted by nodes: C_1, C_2, C_3, C_4, C_5 in our running example. Our goal is to highlight further these interactions for multiple instance scenarios, parallel to the lifecycle of the artifacts involved in such interactions. The generalized artifact interaction abstraction set in our example is,

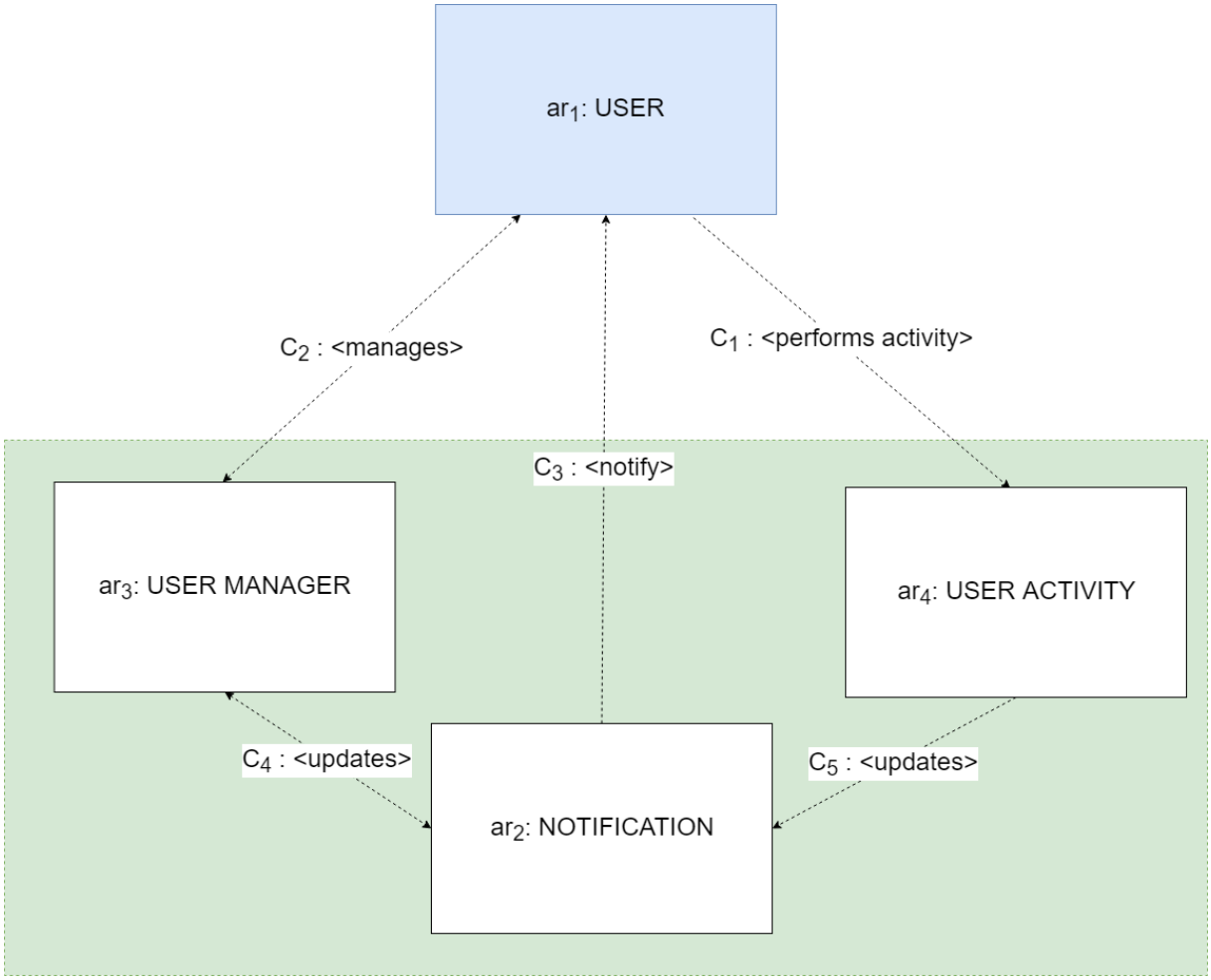


Figure 6.1: High level model of the application

$$(ar_1 \rightarrow ar_3), (ar_3 \rightarrow ar_1), (ar_1 \rightarrow ar_4), (ar_4 \rightarrow ar_2),$$
$$(ar_2 \rightarrow ar_3), (ar_2 \rightarrow ar_1), (ar_1 \rightarrow ar_3), (ar_3 \rightarrow ar_1)$$

6.4 Artifact Lifecycle Model

In this step, the session-specific projection logs generated are used to create artifact specific session projection sets. The sets are used to generate all the interaction node choreographies. Using the generalized artifact projection set algorithm, we then generate artifact lifecycles using our proposed modelling approach.

From P_1 (projection set for object instance 1), we can generate Pr_1^1 for ar_1 by classifying the interactions where only ar_1 is involved. That is, $Pr_1^1 \in$P$_1$, where Caller/Callee $= ar_1$ as shown in table 6.6.

We first need to generate the node interaction protocol for interactions between the identified artifacts. We apply the interaction node choreographies algorithm on our pro-

Table 6.6: Projection log Pr_1^1 for interactions with artifact ar_1

Event_ID (E)	**Caller**		**Callee**		**Message** (M)
	Class (Ar)	$Object_{ID}$	**Class** (Ar)	$Object_{ID}$	
e_1	ar_1	1	ar_3	1	m_1
e_2	ar_3	1	ar_1	1	m_2
e_3	ar_1	1	ar_4	2	m_8
e_5	ar_1	1	ar_4	3	m_8
e_{11}	ar_1	2	ar_4	1	m_8
e_{14}	ar_2	1	ar_1	1	m_5
e_{15}	ar_1	1	ar_3	1	m_6
e_{16}	ar_3	1	ar_1	1	m_7

Table 6.7: Projection log Pr_2^1 for interactions with artifact ar_1

Event_ID (E)	**Caller**		**Callee**		**Message** (M)
	Class (Ar)	$Object_{ID}$	**Class** (Ar)	$Object_{ID}$	
e_3	ar_1	1	ar_4	2	m_8
e_7	ar_1	2	ar_3	2	m_1
e_8	ar_3	2	ar_1	2	m_2
e_{10}	ar_2	2	ar_1	2	m_5
e_{11}	ar_1	2	ar_4	1	m_8
e_{17}	ar_1	2	ar_3	2	m_6
e_{18}	ar_3	2	ar_1	2	m_7
e_{19}	ar_1	2	ar_4	3	m_8

Table 6.8: Projection log Pr_3^1 for interactions with artifact ar_1

Event_ID (E)	**Caller**		**Callee**		**Message** (M)
	Class (Ar)	$Object_{ID}$	**Class** (Ar)	$Object_{ID}$	
e_5	ar_1	1	ar_4	3	m_8
e_{19}	ar_1	2	ar_4	3	m_8
e_{21}	ar_1	3	ar_3	3	m_1
e_{22}	ar_3	3	ar_1	3	m_2
e_{24}	ar_2	3	ar_1	3	m_5
e_{26}	ar_2	3	ar_1	3	m_5
e_{27}	ar_1	3	ar_3	3	m_6
e_{28}	ar_3	3	ar_1	3	m_7

jection sets to identify all interaction node protocols. The figure, 6.2 shows the interaction node models for 'C_1', 'C_2' and 'C_3' created from the projection set Pr_1^1. We further create the node interaction models for 'C_4' and 'C_5' using the projection set Pr_1^2, as shown in the figure. We can stop designing the choreographies here as all the interaction node protocols have been modelled. Nodes C_1, C_2, and C_3 are involved in the interaction of artifact ar_1 and other artifacts, whereas C_4 and C_5 are interactions within the application artifacts excluding the user artifact. We can also say that the interaction model of nodes C_1, C_2, and C_3 lies within any of the session projection sets, P_i for ar_1, represented by

the generalized projection set from Pr_i^1.

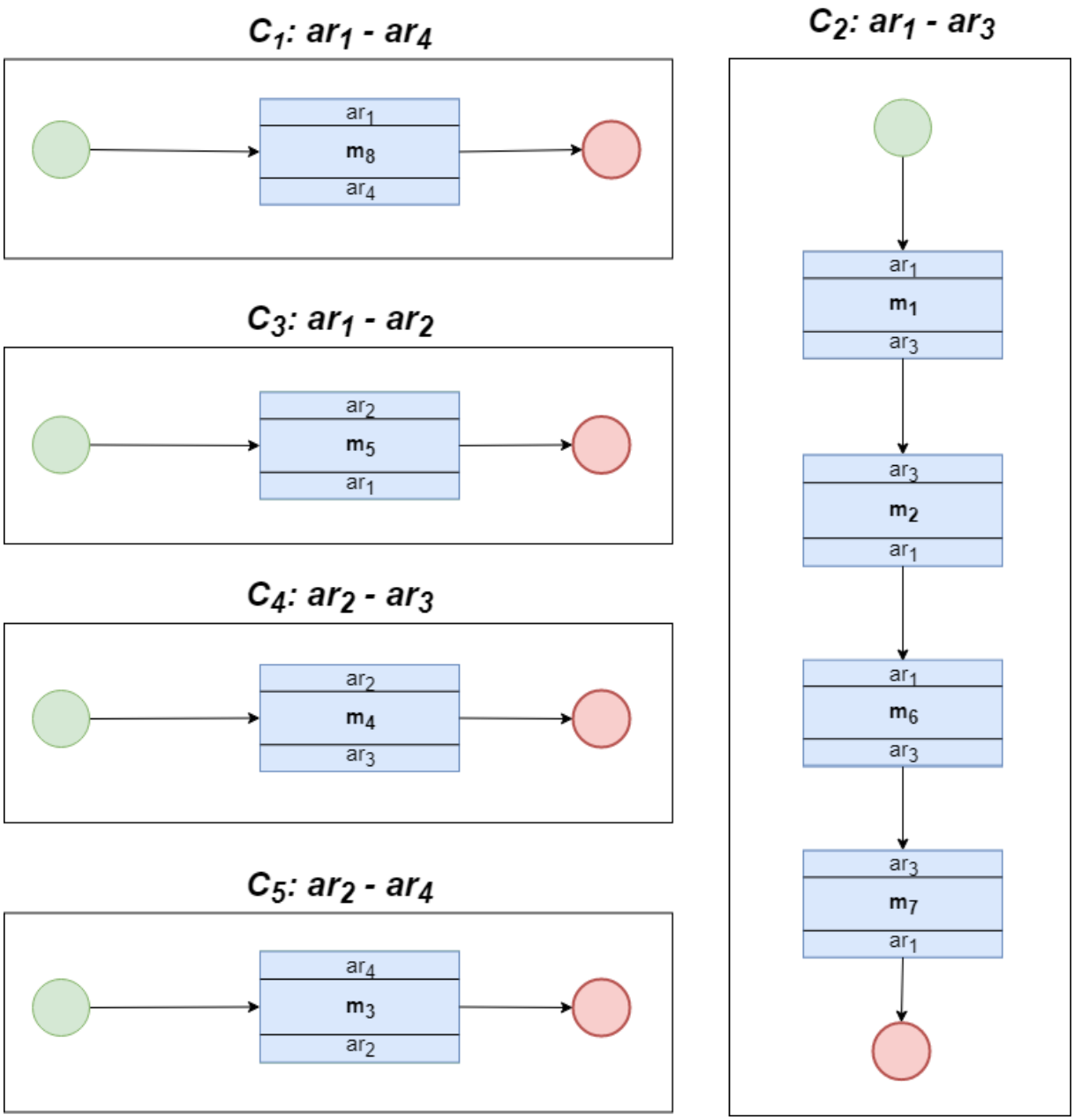


Figure 6.2: Node protocol choreographies for our running example

From the above tables 6.6, 6.7, and 6.8, we can create the generic artifact lifecycle for ar_1 by applying the generalized artifact projection set algorithm on the projection sets. In our running example, when we generalize the projection set for artifact ar_1, the events $m_1 \rightarrow m_2 \rightarrow m_5 \rightarrow m_6 \rightarrow m_7$, always occur in a sequence. The event m_8 occurs randomly for each session of user artifact, and it is modelled as a parallel event with transition events m_p. The events m_5 and m_8 is one-to-many type, while the rest are one-to-one type. The figure 6.3 shows the resulting model, designed using our modelling approach mentioned in the previous chapter.

Similarly, we can use the projection logs for each artifact, create their generalized set and model it. We show the first session tables for ar_2, ar_3, and ar_4, in the tables, 6.4,

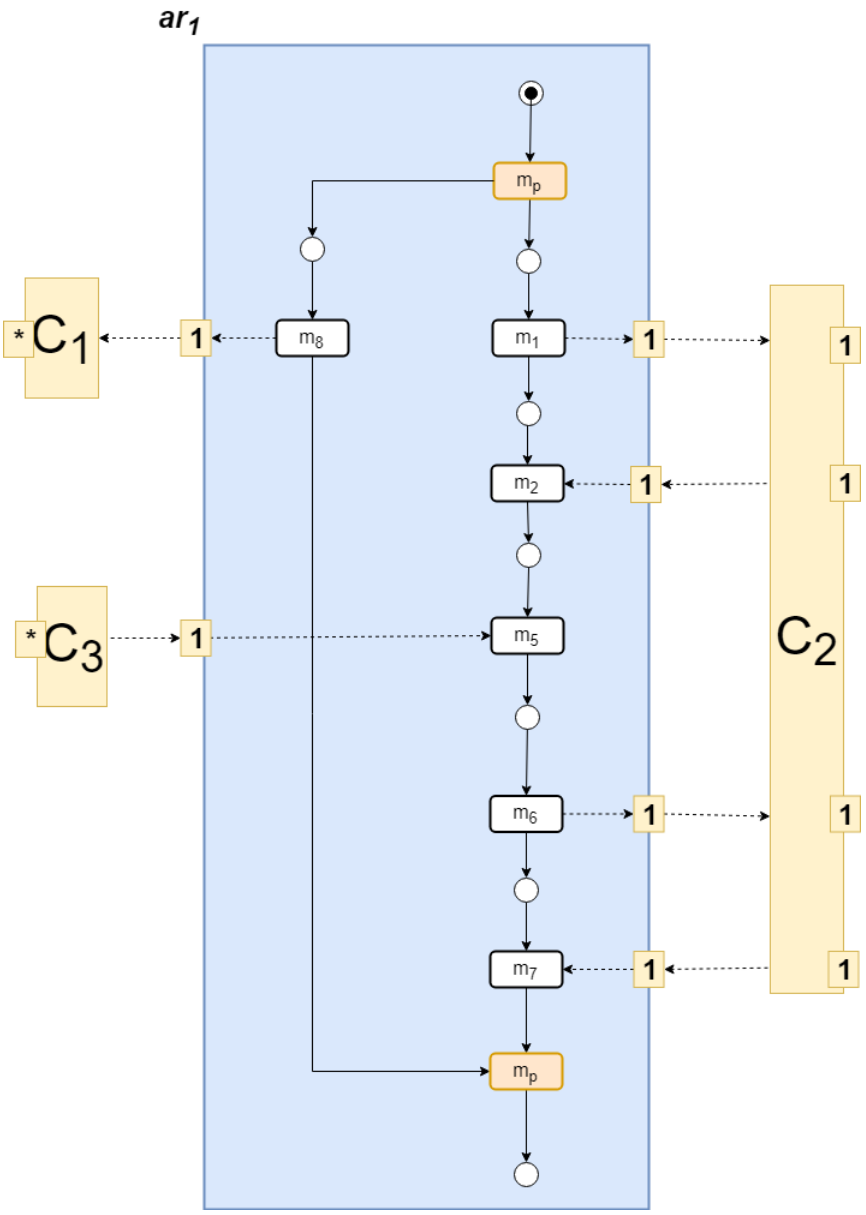


Figure 6.3: Artifact lifecycle for ar_1: User

6.10, and 6.11 respectively. We apply the same steps to generate a model out from their generalized set across various session projection logs for every artifact. The resulting model can be merged, as shown in the next step, to show the new view of the overall model of the running example, covering one-to-one and one-to-many interactions with respect to the user artifact. As the sessions IDs are set and captured in the event logs as per the $object_{ID}$ values originating from artifact ar_1 in our running example.

The cardinalities for the interactions generated can also be showcased in the overall model by a simple extension of cardinalities from the high-level contextual model and projection logs from the previous step, to a high-level artifact centric model as shown below in the figure 6.4.

In our running example, the user artifact initiates multiple sessions during its interaction with the application artifacts. Hence, the right interaction type for user artifact

Table 6.9: Projection log Pr_1^2 for interactions with artifact ar_2

Event_ID (E)	**Caller**		**Callee**		**Message** (M)
	Class (Ar)	$Object_{ID}$	**Class** (Ar)	$Object_{ID}$	
e_4	ar_4	2	ar_2	2	m_3
e_6	ar_4	3	ar_2	3	m_3
e_{12}	ar_4	1	ar_2	1	m_3
e_{13}	ar_2	1	ar_3	1	m_4
e_{14}	ar_2	1	ar_1	1	m_5

Table 6.10: Projection log Pr_1^3 for interactions with artifact ar_3

Event_ID (E)	**Caller**		**Callee**		**Message** (M)
	Class (Ar)	$Object_{ID}$	**Class** (Ar)	$Object_{ID}$	
e_1	ar_1	1	ar_3	1	m_1
e_2	ar_3	1	ar_1	1	m_2
e_{13}	ar_2	1	ar_3	1	m_4
e_{15}	ar_1	1	ar_3	1	m_6
e_{16}	ar_3	1	ar_1	1	m_7

Table 6.11: Projection log Pr_1^4 for interactions with artifact ar_4

Event_ID (E)	**Caller**		**Callee**		**Message** (M)
	Class (Ar)	$Object_{ID}$	**Class** (Ar)	$Object_{ID}$	
e_3	ar_1	1	ar_4	2	m_8
e_4	ar_4	2	ar_2	2	m_3
e_5	ar_1	1	ar_4	3	m_8
e_6	ar_4	3	ar_2	3	m_3
e_{11}	ar_1	2	ar_4	1	m_8
e_{12}	ar_4	1	ar_2	1	m_3

can be showcased for all outgoing interactions as many-to-one. Many-to-many behaviour is highlighted if an event occurs multiple times for a single user session similarly. However, the interaction type does not give details related to the lifecycle of the interaction at this level completely. The many-to-many interaction dependencies that occur remains uncovered and is captured in the next step.

6.5 Overall Artifact Model

For ar_1, we get the following entries in the extended projection sets created from the event log L, showcasing interactions between ar_1 and other artifacts for multiple object sessions. We identify ar_1 interactions with respect to the other artifacts and establish the true interaction cardinalities for such a case.

From the table 6.12 we can conclude the following cardinalities for interactions between ar_1 and other artifacts as follows,

- ar_1 and ar_3 is many-to-one: all events occurs once for each instance, i.e., multiple

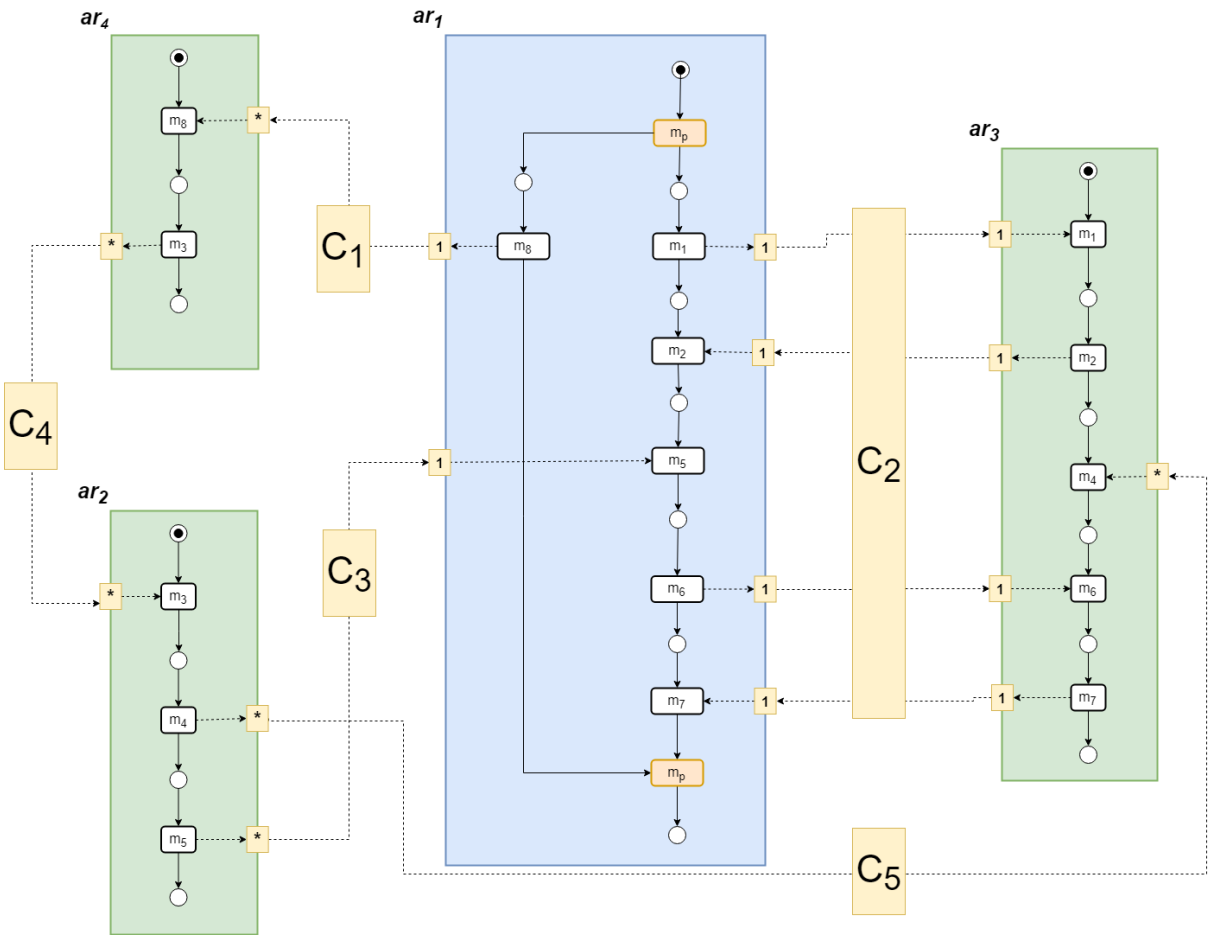


Figure 6.4: Overall Artifact Model - Partial View

users can attach and detach happens once.

- ar_1 and ar_4 is many-to-many: event event occurs multiple times for each instance, i.e., multiple users can perform multiple activities.
- ar_1 and ar_2 is many-to-many: event occurs multiple times for each instance, i.e., multiple users can get notified multiple times.

Similarly, we can classify the overall event log L, with respect to ar_2, ar_3 and ar_4 for our running example.

Interaction type description for ar_2 (refer table 6.13):

- ar_2 and ar_4 is many-to-many: event occurs multiple times for each instance, i.e., multiple instances of notifications are sent.
- ar_2 and ar_3 is many-to-many: event occurs multiple times for each instance, i.e., notify module asks for observers list multiple times.
- ar_2 and ar_1 is many-to-many: event occurs multiple times for each instance, i.e., multiple users can get notified multiple times.

Interaction type description for ar_3 (refer table 6.14):

Table 6.12: Extended projection log, Pr_1 for interactions with artifact ar_1

Event_ID (E)	**Caller**		**Callee**		**Message** (M)
	Class (Ar)	$Object_{ID}$	**Class** (Ar)	$Object_{ID}$	
e_1	ar_1	1	ar_3	1	m_1
e_2	ar_3	1	ar_1	1	m_2
e_3	ar_1	1	ar_4	2	m_8
e_5	ar_1	1	ar_4	3	m_8
e_7	ar_1	2	ar_3	2	m_1
e_8	ar_3	2	ar_1	2	m_2
e_{10}	ar_2	2	ar_1	2	m_5
e_{11}	ar_1	2	ar_4	1	m_8
e_{14}	ar_2	1	ar_1	1	m_5
e_{15}	ar_1	1	ar_3	1	m_6
e_{16}	ar_3	1	ar_1	1	m_7
e_{17}	ar_1	2	ar_3	2	m_6
e_{18}	ar_3	2	ar_1	2	m_7
e_{19}	ar_1	2	ar_4	3	m_8
e_{21}	ar_1	3	ar_3	3	m_1
e_{22}	ar_3	3	ar_1	3	m_2
e_{24}	ar_2	3	ar_1	3	m_5
e_{26}	ar_2	3	ar_1	3	m_5
e_{27}	ar_1	3	ar_3	3	m_6
e_{28}	ar_3	3	ar_1	3	m_7

Table 6.13: Extended projection log, Pr_2 for interactions with artifact ar_2

Event_ID (E)	**Caller**		**Callee**		**Message** (M)
	Class (Ar)	$Object_{ID}$	**Class** (Ar)	$Object_{ID}$	
e_4	ar_4	2	ar_2	2	m_3
e_6	ar_4	3	ar_2	3	m_3
e_9	ar_2	2	ar_3	2	m_4
e_{10}	ar_2	2	ar_1	2	m_5
e_{12}	ar_4	1	ar_2	1	m_3
e_{13}	ar_2	1	ar_3	1	m_4
e_{14}	ar_2	1	ar_1	1	m_5
e_{20}	ar_4	3	ar_2	3	m_3
e_{23}	ar_2	3	ar_3	3	m_4
e_{24}	ar_2	3	ar_1	3	m_5
e_{25}	ar_2	3	ar_3	3	m_4
e_{26}	ar_2	3	ar_1	3	m_5

- ar_3 and ar_1 is one-to-many: all events occurs once for each instance, i.e., multiple users can attach or detach once.

Interaction type description for ar_4 (refer table 6.15):

- ar_4 and ar_1 is many-to-many: event occurs multiple times for each instance, i.e.,

Table 6.14: Extended projection log, Pr_3 for interactions with artifact ar_3

Event_ID (E)	**Caller**		**Callee**		**Message** (M)
	Class (Ar)	$Object_{ID}$	**Class** (Ar)	$Object_{ID}$	
e_1	ar_1	1	ar_3	1	m_1
e_2	ar_3	1	ar_1	1	m_2
e_7	ar_1	2	ar_3	2	m_1
e_8	ar_3	2	ar_1	2	m_2
e_{15}	ar_1	1	ar_3	1	m_6
e_{16}	ar_3	1	ar_1	1	m_7
e_{17}	ar_1	2	ar_3	2	m_6
e_{18}	ar_3	2	ar_1	2	m_7
e_{21}	ar_1	3	ar_3	3	m_1
e_{22}	ar_3	3	ar_1	3	m_2
e_{27}	ar_1	3	ar_3	3	m_6
e_{28}	ar_3	3	ar_1	3	m_7

Table 6.15: Extended projection log, Pr_4 for interactions with artifact ar_4

Event_ID (E)	**Caller**		**Callee**		**Message** (M)
	Class (Ar)	$Object_{ID}$	**Class** (Ar)	$Object_{ID}$	
e_3	ar_1	1	ar_4	2	m_8
e_4	ar_4	2	ar_2	2	m_3
e_5	ar_1	1	ar_4	3	m_8
e_6	ar_4	3	ar_2	3	m_3
e_{11}	ar_1	2	ar_4	1	m_8
e_{12}	ar_4	1	ar_2	1	m_3
e_{19}	ar_1	2	ar_4	3	m_8
e_{20}	ar_4	3	ar_2	3	m_3

multiple activities are performed by multiple users:

- ar_4 and ar_2 is many-to-many: event occurs multiple times for each instance, i.e., multiple notifications are sent by activity module for different instances.

We also realise multi-instance interactions when artifact ar_1 interacts with ar_4. Hence, we mark the corresponding interaction node, 'C_1' to showcase the cross instance communication. Applying the above projection sets interaction cardinalities and multi-instance interactions on the artifact lifecycle model obtained in the previous step, leads to the overall artifact model between user and application, as shown in figure 6.5.

The overall model of our running example captures a successful trace for the notification to be received by the user covering the actual interaction that occurs due to the multi-instance user scenario, including notification independent interactions and artifact specific interactions within the overall application. For our notification use case in our running example, the user artifact was chosen as the artifact for the interaction analysis as it covers the primary scenario of users interacting with the notification application, and where the complex interactions exist.

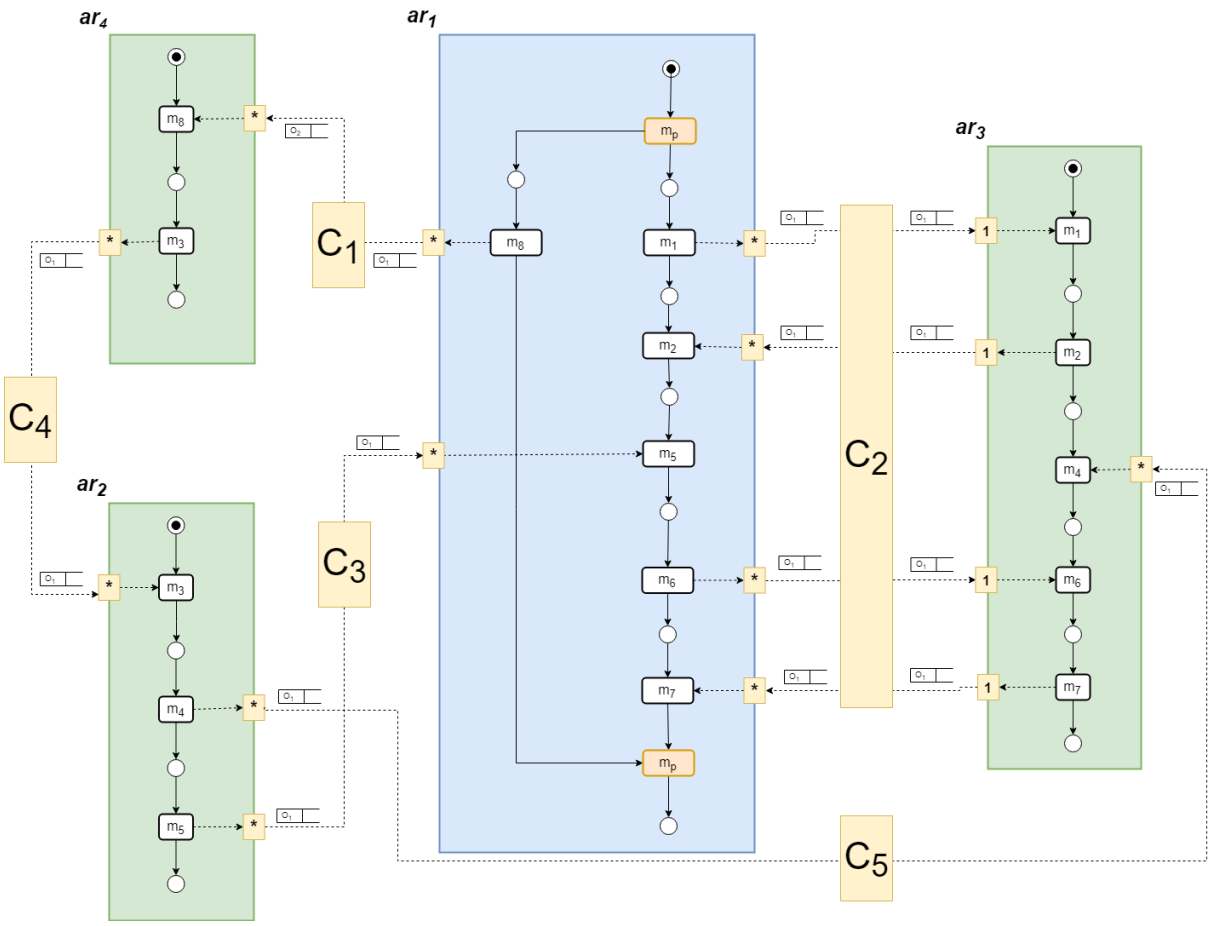


Figure 6.5: Overall Artifact Centric Model

6.6 Software Architecture

In the case of our running example, the first level shows the context level view of the application where we see a user simply being able to access and use the application. The containers are the various logical entities of the application, which are the artifacts or containers consisting of a group of artifacts. The components are directly associated with the artifacts identified using our approach. Our notification running example have the same containers as the components, or rather artifacts. Finally, the code level view shows the artifact view along with the overall artifact centric view, as generated in the previous step. The figure 6.6 shows the entire software architecture view of the notification example and highlights the interaction view between the user and the application artifacts.

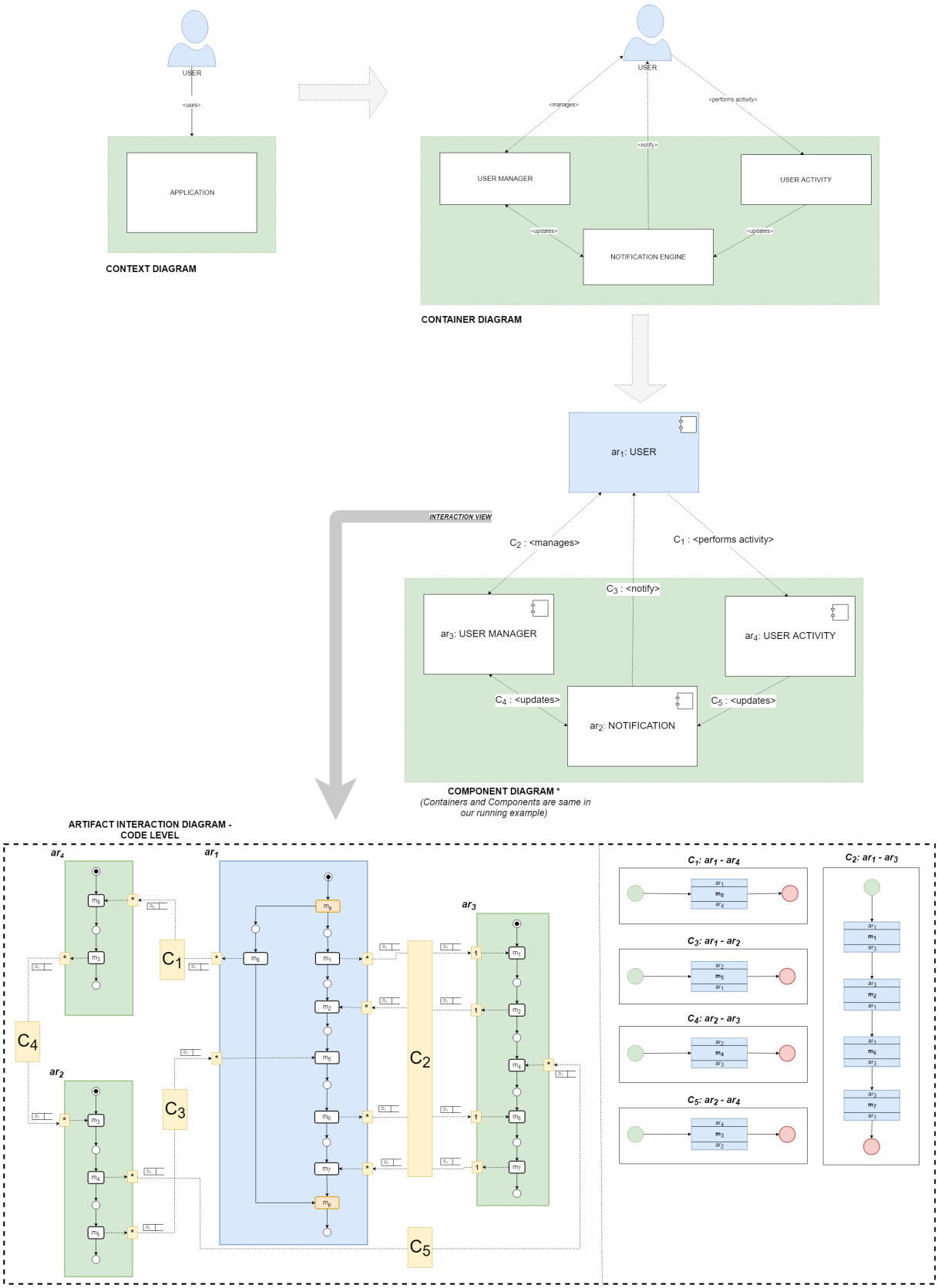


Figure 6.6: Artifact Centric Dynamic Software Architecture Representation of our running example using C4 notation

Chapter 7

Validation and Analysis

This chapter consists of the validation performed for this research, and the analysis of the resulting outputs from the implementation steps performed. The running example is referred for the extensive validation of the implementation approach for the interaction algorithm and the interaction modelling approach.

For internally validating our interaction algorithm and modelling approach, we extended our running example to attach and detach users throughout their session lifetime randomly. The activity content was also randomly pushed to the notification module to depict a real-life scenario of user interaction with such an application. We applied our implementation steps defined in chapter 5 for such cases multiple times and modelled the resulting artifact-centric interaction views. There was no change recorded in the resulting model using the defined interaction algorithm. The resulting artifact centric diagram also remained unchanged.

In terms of the modelling approach chosen, the extended Petri net based diagrams constructed for each identified artifact was compared with the findings of the ProM designed application proclets view showcased in chapter 3 in figure 3.8. Using our approach, with the identification of artifacts and the artifact-centric view, we could showcase more details of the resulting software system, as shown in figure 6.5. The cardinalities of interaction between artifacts are identified for many-to-many interactions as well. The node protocol choreography diagram, shown in 6.2, also showcases the communications occurring for a given artifact concerning other artifacts with which it interacts.

7.1 Approach Analysis

This section details the outputs of our implementation steps, which are the interaction algorithm and modelling approach, along with the final representation of the overall software architecture. The analysis is performed by comparing the approaches with the existing state-of-art techniques, including artifact-centric methods and standard modelling notations such as BPMN 2.0 and Extended Proclets. We applied a few of these approaches in chapter 3 using existing modelling and mining tools, which adds to the approach analysis performed.

7.1.1 Interaction Discovery and Modelling Algorithm

The implementation steps for our interaction algorithm can be summarized by the following steps:

Solution Design

1. Event Log Generation (Refer section 5.2.1)
   - Create event log L having event tuples as, $L = \{E^*\}$, where $E = (Ar \times M \times Ar)$, for every event interaction M between caller and callee artifacts, Ar, with timestamps.
   - $Object_{ID}$ is captured in the event log by manually injecting an auto-generated unique session ID with the caller and callee information.
   - Event log consists only of successful traces such that the dynamic information captures the intended use-case of the software system.
2. Dynamic Behaviour Matrix (Refer section 5.2.2)
   - Construct an interaction behaviour matrix, I, to capture the total number of unique instance $object_{ID}s$ calls (Φ), for a specific interaction between π_1 and π_3 for an event e in the event log, L.
   - Compare Φ with the total number of artifact instance $object_{ID}s$ used for the event log, to identify the presence of simple one-to-one or complex many-to-many interactions.
   - Create a high-level view of interacting artifacts by capturing an abstracted set of artifact communication and its events as they occur in the event log to identify interaction nodes as C_k.
3. Artifact Projection Log (Refer section 5.2.3)
   - Create projection sets from event log, $P_i = (Ar \times M \times Ar)$, having event interactions between artifact instances for a unique session $object_{ID}$, i, listed chronologically.
   - Create a high level container model showing various artifacts, Ar and their nodes of communication.
4. Artifact Lifecycle Model (Refer section 5.2.4)
   - Create artifact specific projection sets, Pr_i^j, such that, $Pr_i^j \subset P_i$, containing event tuples consisting of an artifact, ar_j.
   - Generate node protocol choreographies using the above projection sets using the node choreography algorithm
   - Using the generalized clustered set modelling algorithm to generate Pr_j^g, for every artifact ar_j create artifact specific models to showcase their lifecycle using the specified extended Petri nets formalism.
   - Merge all the individual artifact models from the above step, to showcase the partial interaction view of the overall artifact model.

5. Overall Artifact Model (Refer section 5.2.5)
    - Extend the projection sets Pr^i_j, such that, $Pr^j \subset L$, containing event tuples consisting of an artifact, ar_j.
    - Realise the identified cardinalities with respect to the multi-user scenario, depicting many-to-many interactions if any.
    - Realise the interaction nodes which have multi-instance communication between artifacts across user sessions by use of initialization notations in the model, as specified.
    - Extend the overall artifact model from the above step to showcase the newly identified interactions cardinalities.
6. Constructing Software Architecture (Refer section 5.2.6)
    - Abstract the software system at the abstraction levels, such as context, containers, and components view, using the C4 notation.
    - Extend the code view of the C4 notation to showcase the artifact centric overall artifact model obtained in the previous step.

The above steps help in designing the overall model and represent a given software's architecture. The definition of event logs involves the definition of artifacts and their interaction events. The artifacts, when described with respect to the software's code, are simply classes for an object-oriented software system or similar logical components of any given software system. The event log also captures object session IDs that helps in identifying a unique artifact's session. The choice of artifact here is based on the use-case the system is trying to achieve. For an application where a user interacts with the application, it seems that the best way is to define object sessions in terms of user artifact's instances, as interactions between the user and the application are of utmost importance. The temporal information in the event log gives the chronological sequence that helps in defining the artifact's lifecycles. The main difference in the definition of our event log, and event logs used until now in the literature, is the addition of artifact IDs and session IDs. The need is reasoned with the idea of creating an artifact specific model along with capturing complex interactions. The use of session IDs can help break down these complex interactions with respect to a particular key artifact instance, which in our running example is the user.

The dynamic behaviour matrix is a good abstraction for understanding the presence of complex many-to-many communications between artifacts and for each of its interaction event. Further creating projection logs for each session, breaks down the event logs to showcase session-specific logs, which can be further broken down as per each artifact. We can now observe the various lifecycles of an individual artifact, with respect to multiple sessions. Thus, the artifact lifecycle is revealed in this process when we generalize the projection sets for every session. We can go ahead and model these artifacts, and together they would represent the overall artifact-centric view of the system with basic one-to-one type interaction overview. We propose the steps mentioned in the implementation step under chapter 5, for creating one-to-one node protocol choreographies, and then creating the generalized artifact specific projection sets using our generalized clustering algorithm.

Still, the interactions remain unspecified in terms of their actual complexities, as only one-to-one and one-to-many interactions are covered via these projection logs. For this reason, we extend our projection sets for each artifact with respect to the entire event log, consisting of all sessions. It helps in uncovering the many-to-many type of communications between artifacts, by comparing the extended projection sets for every artifact and its interaction with other artifacts. Multiple interaction events originating from the same artifact can also showcase cross-session interactions in the overall artifact model, which are modelled along with the interaction cardinalities for each event at the node.

The need to generate session-specific projection sets to showcase simple one-to-one communications between artifacts is due to the need of logically breaking down the main event for each artifact and study the behaviour of a specific artifact's instance. This means the generated projection set can depict the interaction concerning the chosen artifact's single instance session. If there exists a cross instance communication for that artifact, it gets captured in the respective projection set. If there is no such cross instance communication, the projection sets can still unveil the basic one-to-one or one-to-many type of communication from it. Further extending these projection sets for all sessions, leads to the identification of many-to-many type interactions between artifacts, which are captured and modelled in the resulting interaction centric modelling view.

7.1.2 Process Deliverable Diagram

The corresponding process deliverable diagram for the solution design is shown in the figure, 7.1. The Process Deliverable diagram (PDD) shows the activities to be performed on the left-hand side, whereas the right-hand side shows the concepts involved in the process and their relationships. The activities shown are the steps taken in this research implementation solution design, which consists of further sub-activities as shown. The concepts related to each step is linked to the corresponding activity or sub-activity.

We start our solution design by the Event Log Generation activity. The first sub-activity within is the identification of log entities, where we define concepts such as, ARTIFACTS and EVENT MESSAGES. Every artifact can be associated with one or more OBJECT_IDs, and an EVENT MESSAGE can occur between two ARTIFACTS. All these concepts are part of a SOFTWARE SYSTEM [24]. Next, we generate our EVENT LOG having tuples consisting of ARTIFACTS and EVENT MESSAGES. Note that, the ARTIFACTS when captured here are along with their OBJECT_ID information.

The EVENT LOG is used in the next activity to generate the INTERACTION BEHAVIOUR MATRIX and identify the interaction nodes and the presence of complex interactions. The matrix is captured using Step 2 mentioned in our solution design in Chapter 5, and the interaction nodes are captured via the concept INTERACTION NODES. The next activity contains sub-activities of creation of projection sets and the creation of a high-level model of all interacting artifacts. The projection sets are created from EVENT LOG, as defined by the Step 3 mentioned in the Chapter 5, and is represented by concept PROJECTION SETS. These sets, along with ARTIFACTS and INTERACTION NODES define the high-level model in the concept MODEL.

The next activity is the construction of artifact specific projection logs, as shown in our solution design Step 4 in Chapter 5. This is shown in the PDD via the concept ARTIFACT PROJECTION SETS created using the PROJECTION SETS, as specified in the solution design. These artifact specific projection sets are then used to gener-

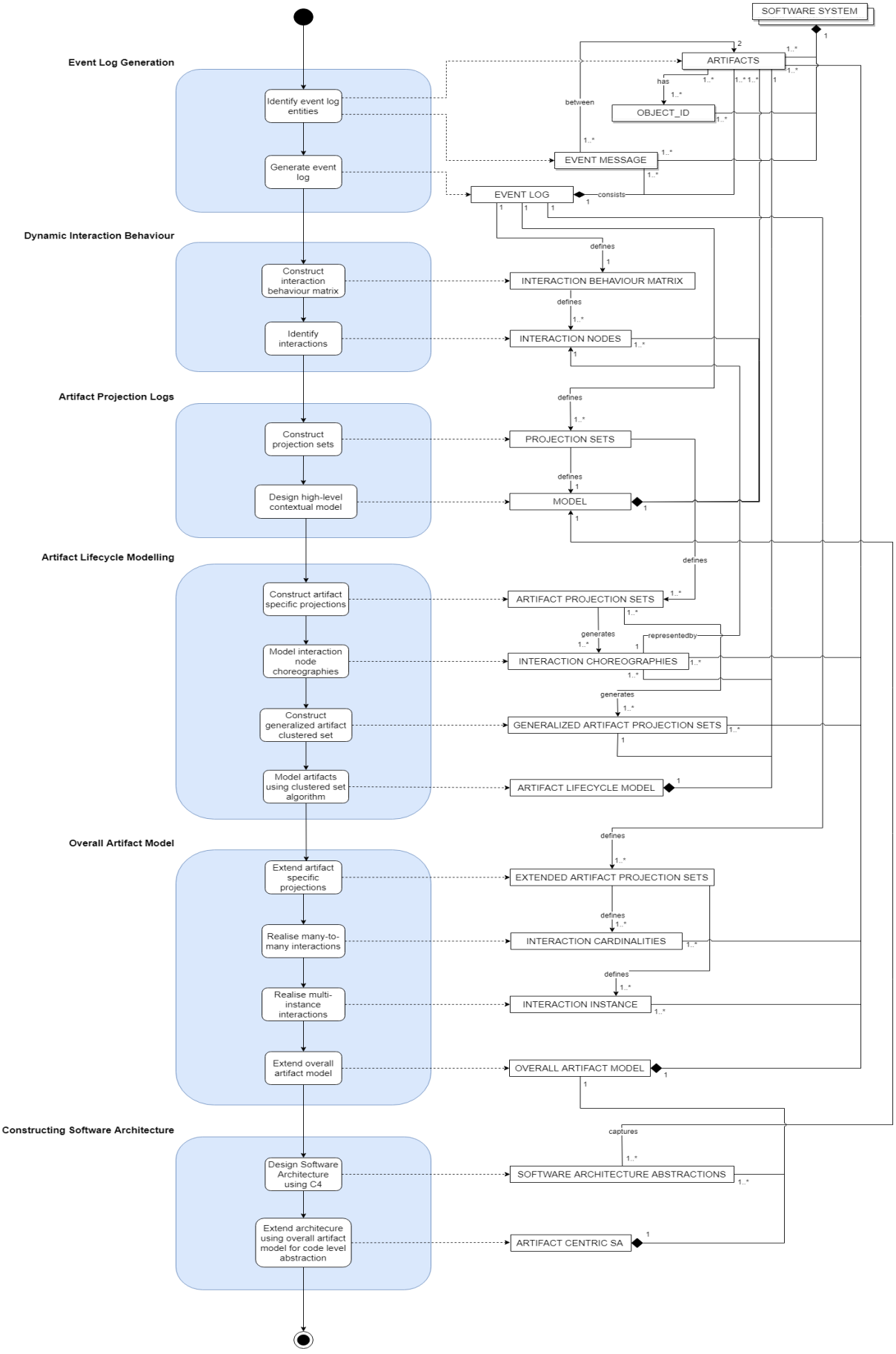


Figure 7.1: Process Deliverable Diagram for the Interaction Discovery and Modelling Algorithm

ate INTERACTION CHOREOGRAPHIES using the node choreography algorithm, and GENERALIZED ARTIFACT PROJECTION SETS using the generalized artifact projection algorithm of our solution design from Chapter 5. The next sub-activity showcases the generation of artifact specific models via the concept ARTIFACT LIFECYCLE MODEL, defined using ARTIFACTS, the GENERALIZED ARTIFACT PROJECTION SETS, and INTERACTION CHOREOGRAPHIES.

The overall artifact model activity results in an overall artifact model with their interaction cardinalities. We have sub-activities such as extending the artifact specific projections shown by the concept EXTENDED ARTIFACT SPECIFIC PROJECTIONS, generated from EVENT LOG, using the steps mentioned in our solution design. We further realise interaction multiplicities in the next concept INTERACTION CARDINALITIES. We also define a concept to capture the multi-instance interactions arcs as INTERACTION INSTANCE. The final sub-activity here is to generate an overall artifact model, shown by the concept OVERALL ARTIFACT MODEL, modelled using ARTIFACTS, GENERALIZED ARTIFACT PROJECTION SETS, INTERACTION CHOREOGRAPHIES, INTERACTION CARDINALITIES, and INTERACTION INSTANCE concepts.

The final activity of our solution design is the extension of the overall artifact model with respect to various abstraction levels of software architecture. First sub-activity here is to define software architecture abstractions using C4 approach and defined using the MODEL concept. This is shown by the concept of SOFTWARE ARCHITECTURE ABSTRACTIONS. We then extend the C4 model to showcase the code level abstractions using the OVERALL ARTIFACT MODEL, shown by the concept ARTIFACT CENTRIC SA.

The final software architecture model can be thus captured within a database model, with the following concept entities with the mentioned elements:

- ARTIFACTS: Artifacts within the SOFTWARE SYSTEM
  - Artifact_ID: Unique artifact identifier
  - Artifact_Value
- EVENT MESSAGES: Event messages within the SOFTWARE SYSTEM
  - Event_Message_ID: Unique event message identifier
  - Event_Message_Value
- INTERACTION NODES: Captures the group of EVENT MESSAGES between any two ARTIFACTS
  - Node_ID: Unique interaction node identifier
  - Callee Artifact: Artifact_ID
  - Caller Artifact: Artifact_ID
  - Node_Events: Set of event messages, i.e. Event_Message_IDs
- INTERACTION CHOREOGRAPHIES: Captures the flow of EVENT MESSAGES between any two ARTIFACTS for an INTERACTION NODE
  - Node_ID

  - Choreography: Set of Event_Message_IDs

- GENERALIZED ARTIFACT PROJECTION SETS: Captures the flow of EVENT MESSAGES within a specific ARTIFACT
  - Artifact_ID
  - Lifecycle: Set of Event_Message_IDs
- INTERACTION CARDINALITIES: Captures the incoming and outgoing multiplicities for every EVENT MESSAGE of a choreography task, in INTERACTION CHOREOGRAPHIES, for an INTERACTION NODE.
  - Node_ID
  - Event_Message_ID
  - Artifact_ID
  - Caller_Artifact_Value
  - Callee_Artifact_Value
- INTERACTION INSTANCE: Captures the cross-instance object information on each incoming and outgoing arc for every EVENT MESSAGE of a choreography task, in INTERACTION CHOREOGRAPHIES, for an INTERACTION NODE.
  - Node_ID
  - Event_Message_ID
  - Artifact_ID
  - Caller_Artifact_Value
  - Callee_Artifact_Value
- SOFTWARE ARCHITECTURE ABSTRACTIONS: Showcases the various level of architecture represented by ARTIFACTS and INTERACTION NODES
  - Level_ID: Unique abstraction level ID (4 levels using C4 approach)
  - Artifact_ID
  - Node_ID

Applying the above database model on our running example shown in Chapter 6, we created our final software architecture with the help of the following concept entities with their values:

- Artifacts: Ar (Refer section 6.1)
- Event Messages: M (Refer section 6.1)
- Interaction Nodes: C_1, C_2, C_3, C_4, C_5 (Refer section 6.2)
- Interaction Choreographies: Refer diagram 6.2 (Refer section 6.4)
- Generalized Artifact Projection Sets: Pr_j^g for each artifact ar_j (Refer section 6.4)

- Interaction Cardinalities: (Refer section 6.5)
- Interaction Instance: (Refer section 6.5)
- Software Architecture Abstractions: C4 levels where Container level abstraction is same as Component level (Refer section 6.6)

7.1.3 Interaction Modelling

We perform analysis and discuss our extended Petri nets formalism with the choreography diagram covering interaction nodes in this section. We compare the modelling notations with existing state-of-art modelling approaches for representing complex behaviours. We refer the models created from our generated event log using existing ProM extensions in Chapter 3.

Extended Petri Nets with interaction choreograhies

The extended Petri nets design referred to in this thesis is influenced by the research performed by Aalst [11] to understand many-to-many behaviour within the system by designing artifacts lifecycle as Petri net based proclets. Further research on many-to-many interactions[7], relies on these extended proclets and adds token identities to showcase synchronization of processes. We try to similarly extend the proclet design to support initialization identities for multi-instance dynamic interactions. We create projection logs for covering one-to-many and many-to-many interactions.

The interaction centric view designed using the extended proclets design can be simplified to show only the interaction nodes using the choreography diagram. We extend the choreography notation to our Petri nets formalism to include the interaction details of our extended Petri nets design. The standard modelling view is shown in the figure, 5.2. We make use of directional arcs with notations (refer figure 5.5) to connect the choreographies with the Petri net lifecycles (refer figures 5.3 and 5.4). The choreography based interaction view does not include the artifact's lifecycle view but showcases the interaction lifecycle from one artifact to another artifact. Comparing the modelling approach to the existing modelling algorithm tools in ProM, we see how the artifact interactions in our approach extensively cover the communication type, one-to-one as well as many-to-many. The existing approaches are applied in the chapter 3 in the figure 3.8. The dependency and causal graphs generated via ProM, as shown in the figures 3.6 and 3.5 respectively, tells us how the various caller and callee artifacts interacts but don't showcase the type of interaction. We combine the formalism of Petri nets notation for artifact lifecycle, along with the flexible modelling of choreographies for interaction nodes, to generate the overview of a software application. We then introduce the cardinalities in the model to showing the type of interactions, which if can be modelled via ProM would generate a far better software architecture overview.

The running example is used to show the extended Petri nets notation, where each artifact is modelled with respect to the overall notification use-case, as shown in figure 6.3. The interaction node labels are modelled with the choreography diagram, as shown in figure 6.2. Together, along with the interaction cardinalities, we see the overall artifact model of the notification application in the figure, 6.5.

7.1.4 Software Architecture Representation

The representation of software architecture in our thesis relies on the abstraction levels of the software systems at various logical stages. We refer the C4 notation for our abstraction levels definition, where the highest level of abstraction is the context diagram, followed by the container diagram, then the component diagram and finally the code diagram. We use the standard notations used to represent a C4 based software architecture view for the first three abstraction levels. For the code view, we make use of our extended Petri nets design to show both the artifact lifecycle and its interaction choreography view.

In our running example, the context level diagram is shown by depicting the interaction between the two main entities of the use-case, a user getting notified by an application, as shown in figure 6.6. We further zoom-in to see the container diagram, which is same as our component diagram, as the logical containers in case of our running example are the actual components itself. These components are the identified artifacts as well. We see how the user interacts with the three containers/components/artifacts in our example. To see the interaction view, we further zoom-in from our component diagram to view only the node choreography design. The extended choreography diagram can also be projected on-demand with the overall artifacts lifecycle view at the code level view to understanding the interaction design, as shown in figure 6.6.

The use of C4 modelling approach breaks down the architecture at the abstraction levels that helps in a better understanding of the system. The code level being the most technical view of the system, use of our modelling approach of extended Petri Nets with node choreographies, simplifies the code view. Since artifact centric process mining approach is used in our algorithm, the lifecycle of each software entity within the application can be understood with ease. Use of choreographies generates a simplistic view of the interaction flow between any two such entities. The C4 approach is relatively new among researchers but seems to have gained increasing interest among the business stakeholders. Application of our approach can thus seem useful for representing software architectures among such stakeholders, without losing the synchronicity with the actual implemented architecture of the software system.

7.2 Limitations and Discussion

In this section, we discuss our solution design approach with respect to the internal and external shortcomings from our validation, discuss alternatives, and possible enhancements with respect to those shortcomings.

7.2.1 Internal Validity

We considered how our solution design bias was minimized earlier. However, we do establish certain internal threats to the solution, which is discussed in detail under this section. We make an assumption of assigning artifacts to technical architectural elements, such as class or package hierarchies. While this assumption works perfectly in demonstration of our interaction algorithm, it does create a need for abstractions in the languages used for constructing a software system. For our running example notification application, we used Java as the programming language, with classes abstracted as artifacts. We do realise that the languages might not be consistent, especially for interface systems where multiple languages could be used in the overall software application.

We then define our event logs with event tuples containing artifacts and their interactions. The artifact information is stored with respect to the caller and callee artifacts, as defined, and assigns $object_{ID}s$ to these artifacts. The choice of which artifact to be chosen has been reasoned with the use case the software application serves. However, such decisions require the need of a business stakeholder defining the importance of an artifact with a technical stakeholder instrumenting the software application. These unique $object_{ID}s$ helps us in generating session-specific logs, which further helps us generalize the artifact projection logs while defining an artifact's lifecycle. For our running example, we chose three different sessions $object_{ID}s$ to showcase our solution design. We also were able to capture cross instance interactions by injecting the $object_{ID}s$ with our artifacts caller and callee information, thus providing us with a log of artifacts with their unique source and destination $object_{ID}s$, and the interaction message which transpires between artifacts during runtime. The feasibility of injecting unique session $object_{ID}$ can incur a problem when a system doesn't allow a user to inject such IDs within the event log. This could be due to arising consistency problems when a certain read/write lock is modified for an object instance. Thus, we recommend using a dynamic run on a test system which duplicates the software behaviour exactly the same but wouldn't lead to any problems on the implemented running software application.

Next, we see how our modelling approach for artifact lifecycle design bias was minimized and what limitations it possesses. We make use of standard Petri net based notations for representing an artifact, and BPMN choreographies for interactions nodes, along with the use of cardinalities to depict the one-to-one or many-to-many behaviour of interaction nodes between artifacts. These standard formalism are often used separately to depict a software application. Use of these notations together may lead to certain inconsistencies with respect to modelling but can be easily tackled by adjusting a few parameters. The multiplicity of interactions and their cross-instance behaviour are shown using directional arrows with certain known notations. Use of colours, as in our running example, can add more clarity and can be chosen while designing the application. We recommend the use of legends while showcasing the overall artifact model, adding more clarity on the use of various placeholders. The C4 method of abstracted view of

the software system can reduce the visual load on the screen, increasing usability. There still remains the case where the application of used notations do not represent a specific element of a given system and would require additional notation attributes to be added to our modelling approach. The existing notations for Petri nets and choreographies, if applicable, can be used too, given they do not clash with the modelling requirements mentioned in our solution design. However, this again would need verification for multiple types of software systems.

7.2.2 External Validity

In this section, we look at the threats and how it is reduced in the generalization of our solution design. To depict a real-life scenario of many-to-many interactions, we constructed a running notification application. We further create user and application interaction scenarios and dynamically capture the log of events, which defines a successful use case of the application. We leverage the fact that since the use case involves multiple users interacting with an application, we can generate an event log by injecting each user session's information, refer figure 3.1, along with the basic interaction events captured within the event log.

When it comes to real-life application and threats, we tried to replicate the application of our solution design on the running notification application using randomized scenarios. Randomly attaching and detaching the users, and randomly creating notification content, we were able to close a few of the external threats. We see how the application of our solution design fits from the perspective of user interaction with an application. When we try to apply the same logic on more business-oriented systems, say an ERP system, we would have to face the challenge of defining artifacts which is to be used for the multi-instance behaviour of the system. Again, a business stakeholder can shed light in this area, but additional impacts of such a business use case need to be studied in depth. Furthermore, there exists a situation where multiple artifacts can be used to inject their instance IDs in the event log, given all these artifact generate unique object instances which, when interacts, needs to be modelled. Further studies and experiments on various type of systems can add more information here.

The modelling approach used on the basis of abstraction levels of context, containers, components, and code minimizes the real-life usability problems. One can choose to view the architecture at the level of abstraction required, with the code level adding the most insight in the interactions within various elements of the software system, denoted by artifacts. The artifact lifecycles and their interaction nodes at the code level can be viewed on-demand as well. Use of Petri nets for modelling artifact lifecycles and showing interactions as choreographies are part of standard approaches. However, modelling them together such that the multiplicities of interactions between artifacts are modelled separately using interaction cardinalities notations, helps in showing the complex many-to-many interactions at both ends of the interaction. The notations used for cross-instance details the interactions even further, but could be subject to change based on the further designing improvements. The code-level abstraction thus adds clarity on such complex interactions. Application of such design abstractions to represent a software architecture can minimize the external usability threats arising in real-world software systems.

7.2.3 Alternatives

For our event log generation, we create an injecting AspectJ logger to capture the dynamic interaction within the system. During the initial phase, we tried leveraging AJPOlog AspectJ logging tool [6]. However, we couldn't get it to work for our application due to certain changes in the logging packages configurations. If a generalized logging mechanism can be generated to cater majority of the type of software systems consisting of logical programming language abstractions, such as our running example, the processing time of investigating and generating a system's dynamic view can be significantly reduced.

Another alternative step could be performed during the discovery of node choreographies. We make use of an existing discovery algorithm within our approach. The existing algorithm is used to generate process tress as their discovery process, while we use it for the discovery of interaction events for a particular interaction node. However, we generate the model manually using the same logic. One can choose to simply modify the attributes of the discovery modelling algorithm to cater to this need. Further, the generalized projection sets created for each artifact can be extended to consider even the unsuccessful traces, which possibly could showcase the limitations over the identified interactions. These limitations can be of an advantage to check system technical or logical breakdowns while analysing a software's architecture. Realisation of multi-instance interactions on the corresponding nodes can be validated with more scenarios, and use of modelling alternatives to depict such a notation can be performed on the existing modelling formalism. We further discuss improvements on the existing interaction discovery and modelling approach in the next chapter under future work section.

Chapter 8

Conclusion and Outlook

This section presents our conclusions for our research questions and its sub-questions. We further elaborate on the possible tasks that can be performed on the existing solution design, as part of future work and research prospects.

8.1 Conclusion

Research Question

***RQ**: How to reconstruct a software architecture that can visualize the dynamic behaviour of complex software system interactions?*

In order to answer the main research question (RQ), we proposed a set of sub-questions (SQs) that decomposes the main research question. We first answer these four critical sub-research questions mentioned below that concludes our main research question for this thesis. The sub-research questions are answered as follows,

SQ1: What are the state-of-the-art approaches to best describe a dynamic software architecture reconstruction?

This sub-question is answered using the detailed related literature review conducted in Chapter 4. We highlight the main artifact-centric approaches and other process modelling approaches using software execution data. We further looked at the various interaction-centric research that helped in the understanding of complex many-to-many interactions. The relevant state-of-the-art approaches are shown in the table, 4.1 and figure, 4.1.

SQ2: How to visualize the software execution data in a software architecture having one-to-many or many-to-many type of interaction between artifacts?

This sub-question is answered using the interaction modelling approach showcased in the Chapter 5 and Chapter 6. We define the interaction discovery algorithm steps which define a node choreography modelling algorithm, as well as an artifact lifecycle modelling approach based on extended Petri Nets. The generic notation is

showcased in figure 5.2. Finally, we represent the overall software architecture with the help of C4 abstraction levels, as shown in the software architecture generated for our running example in figure 6.6. The abstraction levels are defined to portray the high-level containers and components of an application represented by artifacts, and the code level abstraction view is designed to show the extended Petri nets formalism for all those artifacts, along with their interaction choreographies. Note, all the models generated are manually designed using the draw.io tool [41].

SQ3: How to extract an interaction model from software execution data using an artifact centric approach?

This sub-question is answered using the interaction discovery algorithm showcased in the Chapter 5 and Chapter 6. We define the interaction discovery algorithm steps, along with an algorithm to generate one-to-one type interaction node choreographies. We also define a generalized artifact projection set algorithm which models each artifact within the software system one at a time. We further merge these individual artifacts to show the overall artifacts lifecycle. Realisation of many-to-many interactions and multi-instance communication within an event tuple of our log is performed. The overall process deliverable diagram is shown in figure 7.1 represents the solution design proposed for our interaction discovery algorithm, thus extracting the interaction model from a software execution data using artifact-centric approach.

SQ4: How can the proposed techniques be applied in real-life complex software systems?

This sub-question is answered using the implementation of our solution design as covered in Chapter 6. We make use of a detailed running example representing a notification application constructed using Java. Randomized behaviour of interactions is implemented within the code to create multiple real-life scenarios, which are then generalized to represent the entities of software architecture, in terms of artifacts. Using our interaction discovery approach for extracting artifact lifecycles and their interactions, we successfully generate the software architecture of the application at various abstraction levels. The proposed techniques for the running notification application implementation of our solution design can thus be applied in real-life complex software systems.

The above sub-questions together answer our main research question. We were able to generate and reconstruct a software architecture to showcase the complex interactions, at various degrees of abstraction. The code level abstraction using our interaction discovery algorithm approach visualizes the dynamic behaviour of the overall software system using artifact-centric process mining, revealing its architecture.

8.2 Future Work

This section covers the probable prospects of this research thesis, focusing on the interaction discovery and modelling algorithm proposed.

8.2.1 Event Logging

We mentioned the shortcomings with respect to the generation of event logs in the previous chapter. These limitations, such as the intervention by a business stakeholder to define the artifacts responsible for the creation of unique object identifiers. These session IDs generated are injected separately in the event log with the caller and callee information. The problems of clashing identifiers could result in contingencies while understanding the multiplicity of interactions. Future work could be focused on solving these problems by creating unique artifact specific session IDs, which are auto-generated and injected in the event log during runtime.

The generation of multiple sessions of event logs and further creation of projection sets to represent each artifact specific interactions can also be automated. These steps would reduce the processing time quite significantly. Further tests would need to be conducted to check the feasibility of such auto-generated projection sets.

8.2.2 Auto-generation of generalized artifact projection sets

The generalization algorithm for artifact specific projection set takes all possible traces of events and tries to generate an artifact's true lifecycle. We proposed the usage of parallel silent notation as transition events for modelling scenarios having parallel or random interactions. Future work would require the auto-generation of this generalized set to automatically create artifact lifecycle using our modelling notation of extended Petri Nets. The node choreography algorithm can also be automated and modelled together in this step. Nevertheless, one would require further conformance checks of these auto-generated sets to validate the process.

8.2.3 Visualization

The visualization notation chosen in this research leverages the existing Petri nets and choreography diagram notations. The existing usage of these notations are covered in Chapter 3 and Chapter 4. Our approach relies on these visualization techniques as they represent the formal behaviour of a given system very well. By combining the notations as suggested by our approach, we can take advantage of both the notations. Further, we showcase the software architecture at four different abstraction levels. Further research must be conducted to check the feasibility of the combination of all these modelling formalism, and any clashing notations would need to be dealt with. We did not come across any such clashing notations, but thorough research can shed more knowledge in this domain. Use of semi-structured and structured notations together also increases the usability of the resulting software architecture.

8.2.4 Mining algorithms for complex interactions

As per our literature research, we know that there aren't any mining algorithms that can deduce the many-to-many type interactions for a software application. Potential prospect for research would be to create a ProM plugin using our interaction discovery and modelling approach. The process deliverable diagram, shown in figure 7.1, can be used to formulate an extension that can auto-generate event logs and apply our interaction discovery process. The event log we created for our running application can be leveraged here for creating a generic instrumentation code for injecting the right information during the generation of event logs in XES format. The existing models are created using JGraphs tool draw.io, which also provide packages for ProM extension construction. It might be interesting to see how the auto-generated models work for different types of software systems and their architecture, having complex many-to-many interactions.

Bibliography

[1] W. M. P. van der Aalst, *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer Publishing Company, Incorporated, 1st ed., 2011.

[2] E. H. J. Nooijen, "Artifact-Centric Process Analysis Process discovery in ERP systems," no. 257593, 2013.

[3] X. Lu, "Artifact-Centric Log Extraction and Process Discovery," no. September, pp. 1–116, 2013.

[4] E. J. Kaats, "Automated Functional Architecture Generation using Process Discovery," 2015.

[5] C. Liu, B. Van Dongen, N. Assy, and W. M. Van Der Aalst, "Component behavior discovery from software execution data," *2016 IEEE Symposium Series on Computational Intelligence, SSCI 2016*, 2017.

[6] T. D. Jong, *Dynamic software architecture reconstruction using process mining*. PhD thesis, Utrecht University, 2019.

[7] D. Fahland, *Describing Behavior of Processes with Many-to-Many Interactions*, vol. 11522 LNCS. Springer International Publishing, 2019.

[8] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998.

[9] S. Ducasse, D. Pollet, S. Ducasse, D. Pollet, S. Architecture, and D. Pollet, "Software Architecture Reconstruction : A Process-Oriented Taxonomy To cite this version : Software Architecture Reconstruction : a Process-Oriented Taxonomy," 2010.

[10] S. Kebir, A. D. Seriai, S. Chardigny, and A. Chaoui, "Quality-centric approach for software component identification from object-oriented code," *Proceedings of the 2012 Joint Working Conference on Software Architecture and 6th European Conference on Software Architecture, WICSA/ECSA 2012*, pp. 181–190, 2012.

[11] W. M. P. van der Aalst, B. F. van Dongen, M. de Leoni, and D. Fahland, "Many-to-Many: Some Observations on Interactions in Artifact Choreographies," *CEUR Workshop Proceedings*, vol. 705, pp. 9–15, 2011.

[12] W. van der Aalst, *Process Mining: Data Science in Action*. Springer Publishing Company, Incorporated, 2nd ed., 2016.

[13] M. Leemans and W. M. P. van der Aalst, "Process mining in software systems: Discovering real-life business transactions and process models from distributed systems," in *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pp. 44–53, 2015.

[14] Cong Liu, *Software data analytics : architectural model discovery and design pattern detection*. PhD thesis, 2019.

[15] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," *Proceedings - International Conference on Software Engineering*, no. January, pp. 501–510, 2008.

[16] H. Santana Calvo, *Artifact-centric log extraction for cloud systems*. PhD thesis, Eindhoven University of Technology, 2017.

[17] G. L. Z. L. Lei J., Bai R., *Towards a Scalable Framework for Artifact-Centric Business Process Management Systems*, vol. 10042 LNCS. 2016.

[18] M. L. Van Eck, N. Sidorova, and W. M. Van Der Aalst, "Guided interaction exploration in artifact-centric process models," *Proceedings - 2017 IEEE 19th Conference on Business Informatics, CBI 2017*, vol. 1, no. 2017, pp. 109–118, 2017.

[19] G. Decker and M. Weske, "Interaction-centric modeling of process choreographies," *Information Systems*, vol. 36, no. 2, pp. 292–312, 2011.

[20] Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee, "A Design Science Research Methodology for Information Systems Research," *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45–77, 2007.

[21] H. M. W. Verbeek, J. C. A. M. Buijs, B. F. Dongen, van, and W. M. P. Aalst, van der, *XES, XESame, and ProM6*. 2011.

[22] M. Leemans and C. Liu, "XES Software Event Extension," 2017.

[23] C. Günther, "XES Standard Definition," 2009.

[24] N. Rozanski and E. Woods, *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, 2005.

[25] S. Haddad and L. Pomello, "Application and Theory of Petri Nets," *LNCS 7347*, 2012.

[26] J. H. R. J. R. Gamma, Erich; Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[27] V. Popova, D. Fahland, and M. Dumas, "Artifact Lifecycle Discovery," no. May 2014, 2013.

[28] X. Lu, M. Nagelkerke, D. Van De Wiel, and D. Fahland, "Discovering Interacting Artifacts from ERP Systems," *IEEE Transactions on Services Computing*, vol. 8, no. 6, pp. 861–873, 2015.

[29] W. M. P. van der Aalst, "Extracting Event Data from Databases to Unleash Process Mining," in *BPM - Driving Innovation in a Digital World* (J. vom Brocke and T. Schmiedel, eds.), pp. 105–128, Cham: Springer International Publishing, 2015.

[30] A. Pajić and D. Bečejski-Vujaklija, "Metamodel of the artifact-centric approach to event log extraction from ERP systems," *International Journal of Decision Support System Technology*, vol. 8, no. 2, pp. 18–28, 2016.

[31] R. M. Rooijmans, "Architecture Mining with ArchitectureCity," no. May, 2017.

[32] D. Fahland, *Artifact-Centric Process Mining*. 2019.

[33] R. van Langerak, J. M. E. M. van der Werf, and S. Brinkkemper, "Uncovering the Runtime Enterprise Architecture of a Large Distributed Organisation," *Advanced Information Systems Engineering*, vol. 10253, pp. 247–263, 2017.

[34] G. Li and R. M. De Carvalho, "Dealing with artifact-centric systems: A process mining approach," in *CEUR Workshop Proceedings*, vol. 2097, pp. 80–84, 2018.

[35] D. Calvanese, M. Montali, M. Estañol, and E. Teniente, "Verifiable UML artifact-centric business process models," *CIKM 2014 - Proceedings of the 2014 ACM International Conference on Information and Knowledge Management*, no. September 2016, pp. 1289–1298, 2014.

[36] M. Leemans, W. M. Van Der Aalst, and M. G. Van Den Brand, "Recursion aware modeling and discovery for hierarchical software event log analysis," *25th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2018 - Proceedings*, vol. 2018-March, pp. 185–196, 2018.

[37] N. Lohmann and K. Wolf, "Artifact-centric choreographies," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6470 LNCS, pp. 32–46, 2010.

[38] W. M. P. VAN DER AALST, P. BARTHELMESS, C. A. ELLIS, and J. WAINER, "PROCLETS: A FRAMEWORK FOR LIGHTWEIGHT INTERACTING WORKFLOW PROCESSES," *International Journal of Cooperative Information Systems*, vol. 10, no. 04, pp. 443–481, 2001.

[39] M. S. Rosado, *Discovery and Evaluation of Coordina- tion Patterns for Business Processes in many-to-many Relationships*. PhD thesis, Ulm University — 89069 Ulm — Germany, 2019.

[40] S. Brown, "The C4 model for visualising software architecture - Context, Containers, Components and Code."

[41] JGraph, "Draw.IO tool - Create and share diagrams."

Appendices

Running Example Code

The running example represents a notification application. The example code files, along with the generated event log and .csv converted files are uploaded on the GitHub page with the following link:

`https://github.com/kushwahapratik28/masters.git`

Additional ProM Results

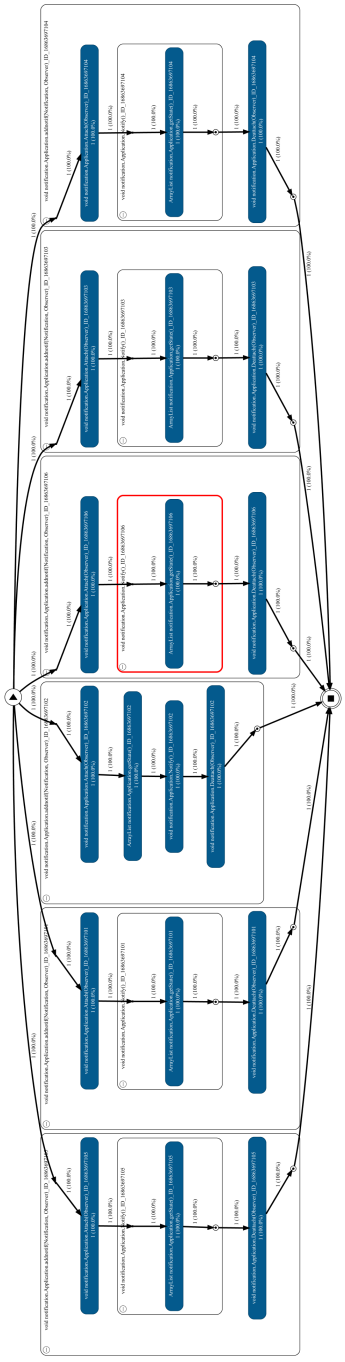


Figure 1: Statechart - Multiple instance IDs

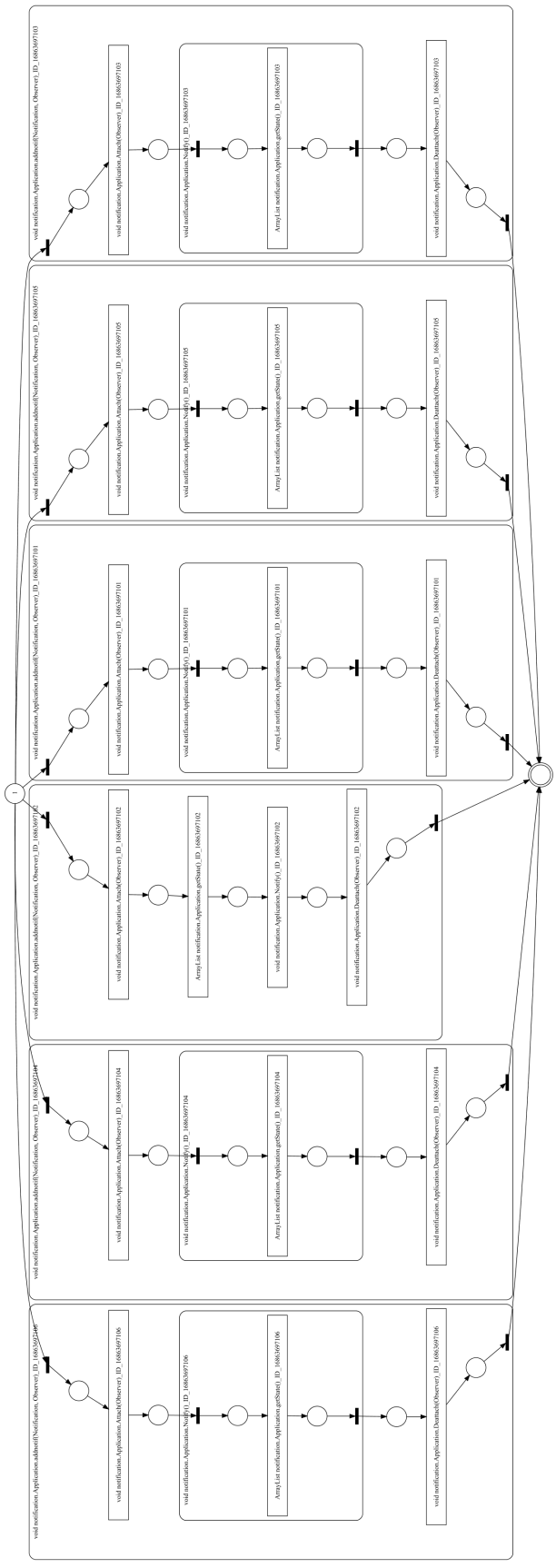


Figure 2: Petri Net - Multiple instance IDs

Research Planning

This sub-section contains the central plan as well as weekly plans for the thesis project. The planning is split by creating multiple milestones that cover the whole period of the project. Figure 3 shows the main plan divided into six significant milestones and then the final phase.

The execution phase weekly is shown in table 1 and runs from August 2019 to January/February 2020, covering the full period of research. The implemented activities column shows the performed task concerning major milestones and displays the actual progress of our research. The activities corresponding to week are planned with an extra buffer week, but if the research is synchronized with the intended plan, the buffer weeks would not be required.

Note that the overall planning chart is the planned approach designed during the initialization phase and is subject to change. The weekly execution phase covers such changes and updates as the thesis evolves. Any significant deviations or challenges faced are also highlighted via this table.

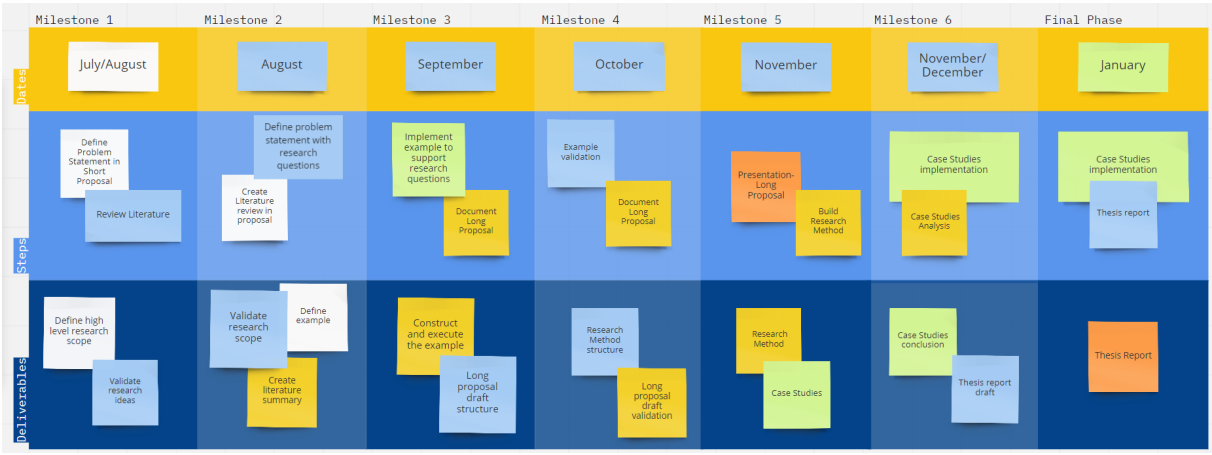


Figure 3: Overall Planning Chart

Table 1: Weekly Execution Plan

Week	Date	Planned Activities	Implemented Activities: Major Milestones
29	17-07-2019	Initiation of the research idea and problem statement	Initiated the research idea with supervisor
30	24-07-2019	Systematic Literature Review (SLR)	Initiated SLR
31	01-08-2019	SLR	
32	08-08-2019	SLR	
33	12-08-2019	Research Question and Example definition	Defined high-level research questions
34	19-08-2019	SLR, Create Long Proposal structure	
35	26-08-2019	Example initiation, SLR	Defined sample example in Java Notification module
36	02-09-2019	Example design, SLR	
37	09-09-2019	Example design, Long proposal structure, SLR	
38	16-09-2019	Example implementation, Long proposal structure, SLR	Additional literature reviewed in SLR
39	23-09-2019	Example implementation, Long proposal draft, SLR	Implemented AspectJ logger and Python converter for sample example
40	30-09-2019	Example implementation, Long proposal draft, SLR	
41	07-10-2019	Example implementation, Long proposal draft, SLR	Defined research approach
42	14-10-2019	Example validation, Long proposal validation	Submitted first draft version of the long proposal to supervisor
43	21-10-2019	Example validation, Long proposal validation	
44	28-10-2019	Build Research Method, Long proposal validation	Initiated research implementation using the sample example
45	04-11-2019	Build Research Method, Long proposal validation	Submitted second draft version of the long proposal to supervisor
46	11-11-2019	Research Implementation, Long proposal validation	Submitted third draft version of the long proposal to supervisor
47	18-11-2019	Research Implementation, Long Proposal presentation	
48	25-11-2019	Research Implementation and Analysis	
49	02-12-2019	Research Implementation and Analysis	Finalized Proposal and Sent for review
50	09-12-2019	Research Implementation and Analysis	
51	16-12-2019	Research Implementation and Analysis - Validations	Long Proposal Presentation
52	23-12-2019	Research Implementation and Analysis - Conclusions	Research Implementation and Analysis
1	30-12-2019	Thesis Report	Thesis Report
2	06-01-2020	Thesis Report	
3	13-01-2020	Thesis Report	
4	20-01-2020	Thesis Report, Thesis Defense	
5	27-01-2020	Thesis Report, Thesis Defense	
6	03-02-2020	Thesis Report, Thesis Defense	Research Implementation and Analysis - Validation
7	10-02-2020	Thesis Report, Thesis Defense	
8	17-02-2020	(Buffer)	Research Implementation and Analysis - Conclusions
9	24-02-2020	(Buffer)	Thesis Defense

UTRECHT UNIVERSITY
Universiteit Utrecht
Heidelberglaan 8
3584 CS Utrecht
Nederland
tel. (030) 253 35 50
https://www.uu.nl/