Department of Information
and Computing Sciences

Faculty of Science

Interaction Oriented Architecture:

A choreography-based design method for component-based software systems

Masters Thesis

M. Klijs BSc - 5863872

Master of Business Informatics

Supervisors:

dr. ir. J.M.E.M. van der Werf
Utrecht University

Dr. J. Gulden
Utrecht University

August 11, 2021

Abstract

Software is often defined and documented in a Software Architecture. This architecture has static and dynamic structures. A part of the dynamic structure of architecture defines the interactions between software components. Current methods of modeling the architecture of a system lack the ability to model both the static and the dynamic structure in one model. Furthermore, most models require a modeler to model every interaction in a separate model. It is therefore not possible to simulate the system as a whole. In this thesis, we introduce INORA - Interaction Oriented Architecture. As part of INORA, we introduce the Interaction Model and Protocols. These models can automatically be composed into one system to allow for simulation. Additionally, we introduce tool support for modeling an Interaction Oriented Architecture in this thesis.

Abstract

Software wordt vaak gedefinieerd en gedocumenteerd in een Software Architectuur. Deze architectuur heeft statische en een dynamische structuren. Een deel van de dynamische structuur van een architectuur definieert de interacties tussen softwarecomponenten. De bestaande methoden om de architectuur van een systeem te modelleren, missen de mogelijkheid om zowel de statische als de dynamische structuur in één model te modelleren. Bovendien vereisen de meeste modellen een modelleur om elke interactie in een apart model te modelleren. Het is daarom niet mogelijk om het systeem als geheel te simuleren. In deze thesis introduceren we INORA - Interaction Oriented Architecture. Als onderdeel van INORA introduceren we het interactiemodel (Interaction Model) en protocollen (Protocols). Deze modellen kunnen automatisch worden samengesteld tot één systeem om simulatie mogelijk te maken. Daarnaast introduceren we in deze thesis een tool voor het modelleren van een Interaction Oriented Architecture.

Acknowledgements

I started my Bachelor's in 2015 at Utrecht University, where I followed the course "Informatiesystemen" teched by Jan Martijn van der Werf during my first year. Due to the dedication of Jan Martijn to the topic, I became very interested in the topic of architecture and processes. During the remainder of my Bachelor's program, I have been a student assistant every year for this course. This kept me interested in the topic. I finished my Bachelor's degree under the supervision of Jan Martijn. In my Bachelor thesis, I have looked at improving critical-thinking skills in process modeling. I really enjoyed the process of writing this thesis and the support by Jan Martijn. The thesis introduced me to the topic of thinking about modeling.

During my Masters's degree, I have followed the course "Software Architecture" by Jan Martijn. This further introduced me to the field of Software Architecture. During the course, Jan Martijn tossed the idea of a modeling language that allowed for the simulation of a whole software system using interactions. We have discussed the topic (outside the course) extensively and eventually we discussed devoting my thesis to this topic.

I would like to thank Jan Martijn for his passionate teaching during both my Bachelor and Master - which introduced me to the topic of this Master Thesis. Furthermore, I would like to thank him for all the time that he has spent working on INORA with me and the feedback that he has provided to the text and contents of this thesis. Without his help, INORA would not have been what it is now.

Furthermore, I would like to thank Jens Gulden for his tips on the technical tools for creating the tool support for INORA. Without his help, the tool would have not have been as extensive as it is now.

Mitchell Klijs
August 11, 2021

Contents

1 Introduction **1**
1.1 Problem statement . . . 1
1.2 Research questions . . . 1
1.3 Contributions . . . 2
1.4 Methods . . . 2
1.5 Running example . . . 2
1.6 Outline . . . 3

2 Software Architecture **4**
2.1 What is software architecture? . . . 4
2.2 Software architectural views . . . 5
2.3 Concurrency viewpoint . . . 6
2.4 Conclusion . . . 8

3 Component-based software design **9**
3.1 What is component-based software design? . . . 9
3.2 Service Oriented Architecture . . . 11
3.3 Modeling components . . . 13
3.4 Conclusion . . . 14

4 Existing notations to model interactions **15**
4.1 Capturing interactions using scenarios . . . 15
4.2 Classifying component interaction modeling techniques . . . 15
4.3 Existing methods to model component interaction . . . 18
4.4 Modeling consistency between choreographies . . . 22
4.5 Conclusion . . . 22

5 INORA: Interaction Oriented Architecture **23**
5.1 The Interaction Model . . . 24
5.2 Protocols . . . 31
5.3 Running example . . . 34
5.4 Conclusion . . . 36

6 The semantics of the Interaction Oriented Architecture **37**
6.1 Petri nets . . . 37
6.2 Workflow net . . . 38
6.3 Analyze INORA . . . 38
6.4 Converting the Interaction Model to the Container net . . . 41
6.5 Converting the Protocols to Protocol nets . . . 43
6.6 Compose the system . . . 50
6.7 Running example . . . 53
6.8 Analysis of the composed system . . . 61
6.9 Conclusion . . . 61

7 Implementation **62**
7.1 Implementation framework . . . 62
7.2 Data model . . . 62
7.3 Tool . . . 66
7.4 Install INORA Tool . . . 75
7.5 Conclusion . . . 76

8 Conclusions **77**
8.1 Implications . . . 77
8.2 Answers to research questions . . . 77
8.3 Limitations & Suggestions for Future Research . . . 78

References **I**

1 Introduction

Developing software is complex. When a software system is developed, one does not simply start programming the system. The structure of the system is planned out beforehand. This planning is documented in a software architecture. Software architecture uses a *divide & conquer* approach to split up a whole software system into manageable parts. This approach is often called a component-based approach.

Bass et al. (2003) define software architecture as: "The set of structures needed to reason about the system, which comprises software elements, relations among them and properties of both". The architecture has *static* and *dynamic structures* (Allen et al., 1998; Eide et al., 2002). Static aspects document modular decomposition. The dynamic structure of a system documents how the single components are connected and interact. One specific part of the dynamic structure is concerned with the *interactions* between components. Interactions document and model how different components in a system communicate with each other and the environment (Bass et al., 2003). Components often communicate with many other components. Additionally, most components are involved in multiple interactions. Interaction models can, therefore, become large and complex. Complex models are not preferable, as they are hard to reason and communicate about (Rozanski & Woods, 2012). A multitude of smaller, less complex, models is therefore often created to represent the whole system. Each model represents one type of interaction or only one possible path of interaction.

With multiple models, it is often hard to validate the interactions of an entire system. One model can be (automatically) validated, but it is not possible to (automatically) combine the different models to validate if the set of models is valid. We define the interactions of an entire system as *complex interactions*: interactions that cannot be (clearly) expressed in one model.

1.1 Problem statement

The problem this thesis aims to solve can be classified as a design science problem. We look into methods to model complex interactions in such a way that helps software architects maintain consistency in the interaction design. We, therefore, adopt Wieringa's template for the problem statement this research will focus on (Wieringa, 2014):

> *This thesis aims to improve* component-based software design *by* creating a systemic approach to design and analyze complex interactions *that* helps software architects to maintain internal consistency in the interaction design *in order to* improve the quality of the designed architecture.

1.2 Research questions

To address our objective, we formulate the following research question:

> RQ1 *What is a systematic approach to design and analyze complex interactions between components?*

The main research question is divided into multiple subquestions (SQs).

> SQ1 *What are current methods to model interaction?*
>
> As the first step, we look into various existing approaches to model interactions. These methods have a scientific basis or are used in practice. We identify why these methods are not sufficient to model complex interactions.
>
> SQ2 *What are the concepts and relations required to design complex interactions between components?*
>
> Next, to create an approach that can be used to design complex interactions between components, it is important to fully understand which concepts and relations are required. Furthermore, we look into how the concepts and relations can be used to analyze the whole system of complex interactions.
>
> SQ3 *What are tools and techniques to support the proposed approach?*
>
> When we have established the theory of an approach to model complex interactions, we implement this approach in a tool. This tool adds support for modeling complex interactions.

1.3 Contributions

This research is relevant for both science and practice. In SQ2 we create an approach and support this approach with a mathematical foundation. We believe that this approach will, in the future, improve the ability to analyze complex interactions.

Furthermore, we believe that our approach can be used in practice to design and analyze complex interactions. With the tool that we create in SQ3, we think that we provide users with a ready-to-use tool that supports their modeling. We believe that the tool will be able to highlight mistakes in a model in order to improve the quality of the designed architecture. The tool can also be used in education - to teach students how to model complex interactions.

1.4 Methods

This thesis follows the design cycle by Wieringa (2014). The design cycle is a subset of the engineering cycle by Wieringa. The engineering cycle consists of the following steps:

1. *Problem investigation* - before designing a treatment, we need to better understand the problem. The goal of the *Problem Investigation* stage is to identify the problem and goals.
2. *Treatment design* - In the *Treatment design* stage, available treatments are identified and a treatment is created.
3. *Treatment validation* - The created treatment is validated to determine if it solves the identified problem.
4. *Treatment implementation* - The treatment is implemented in a real-life situation.
5. *Treatment evaluation / problem investigation* - After the implementation, the treatment is evaluated in the original context. If needed, this also serves as the problem investigation for the next cycle.

In design science, the designed treatment is not implemented in a real-life situation. In the design science cycle, only the first three steps are therefore executed. The last two phases are out of the scope of this thesis. To answer our research questions, we apply various research methods (illustrated in Table 1).

Research method	**SQ1**	**SQ2**	**SQ3**
Design-phase	1	2	2 & 3
Literature research	✓		
Technique construction		✓	✓

Table 1: Research methods used to answer the subquestions (SQs)

1.5 Running example

In this paper, we use a running example to illustrate different concepts in software architecture and to evaluate the feasibility of our approach.

The running example concerns a system of different banks working together to provide Automated Teller Machines (ATMs). This example is inspired by Geldmaat[1] - a cooperation between Dutch banks to provide one ATM service. The cooperation exists because people in The Netherlands use less and less cash (and thus fewer ATMs). Geldmaat provides different banks with a sustainable way to keep running ATMs and to ensure that cash remains available, accessible, affordable, and safe for everyone[2]. It started in 2019 and has since then replaced all ATMs of three major Dutch banks. The company allows other banks to join. It is, however, also possible for customers of not-joined banks to use the ATMs. In our example, we use the main concepts as an inspiration for this real-life example. The architecture and models that are portrait in this thesis, however, are in no way verified or documentation of how the real-life Geldmaat-system works.

[1] https://www.geldmaat.nl/
[2] https://www.geldmaat.nl/vragen/veelgestelde-vragen/728886

In our running example, we distinguish the following concepts:

- The **Bank authority** - these are the multiple banks (the three banks that are currently using the Geldmaat system, banks that join in the future, but also not-joint banks that are supported by the system).
- The **ATM** - multiple (types of) ATMs that allow a customer to withdraw money from their bank.
- The **Customer** - the end-customer of a bank that uses the ATMs.

Our running example can be classified as a many-to-many problem; there are many banks and many ATMs that should all be able to communicate with each other. With many-to-many issues, a central entity is often added to abstract the many-to-many interactions. In this way, the ATMs do not have to communicate with all supported bank authorities directly. This prevents the task of updating all individual ATMs when a new bank authority needs to be supported or one of the existing bank authorities changes their systems. Furthermore, this design is also more secure, because the different bank authorities are not responsible for validating the ATMs. In our example, we call this central entity: *System*.

The context of our example is illustrated in Figure 1 (a so-called context diagram, read more in Chapter 2). The ghosted boxes depict a multitude of entities (many banks and many ATMs). The arrows depict the interactions between the actors. For example, the *Customer* uses the *ATM*, the *ATM* communicates with the *System*.

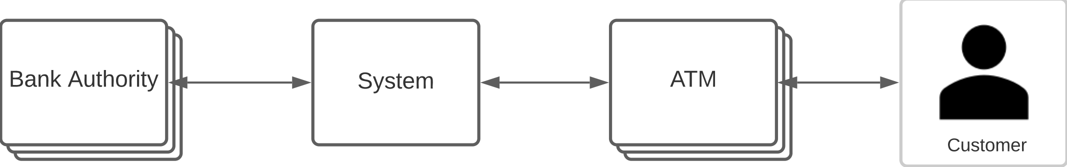


Figure 1: Context diagram of the ATM running example.

1.6 Outline

The rest of this thesis is structured as follows. Chapter 2 explains Software Architecture in depth. It will shortly touch upon the software architectural process and explain software architectural views. In Chapter 3, we dive into the software design pattern of component-based design. We also discuss Service Oriented Architecture and how to model the different responsibilities of components.

In Chapter 4 we aim to answer SQ1. It lays out current techniques for modeling interactions between components in software. In Chapter 5, we introduce INORA: the Interaction Oriented Architecture. This method aims to create an approach to design complex interactions. We will discuss all concepts that are used in this method. Additionally, we will create an Interaction Model and Protocols for our running example. In Chapter 6, we present the semantics of INORA. We introduce a method to (automatically) compose a system that is modeled in INORA. This composed system can be used to analyze the system. We also show how to use this method on our running example. After Chapter 5 and Chapter 6, we can answer SQ2. Next, in Chapter 7 we answer SQ3 by introducing tool support for INORA.

We conclude this thesis in Chapter 8, where we summarize the findings of this thesis, discuss the limitations of this research, and provide some pointers for future research.

2 Software Architecture

Software architecture is the connecting factor between requirements and the implementation of a system. It is used to control the complexity of creating software. The process of creating a software architecture is often displayed in the form of the Three Peaks Model (Woods & Rozanski, 2010), which stems from the Twin Peaks Model (Cleland-Huang et al., 2013). The Three Peaks Model is depicted in Figure 2.

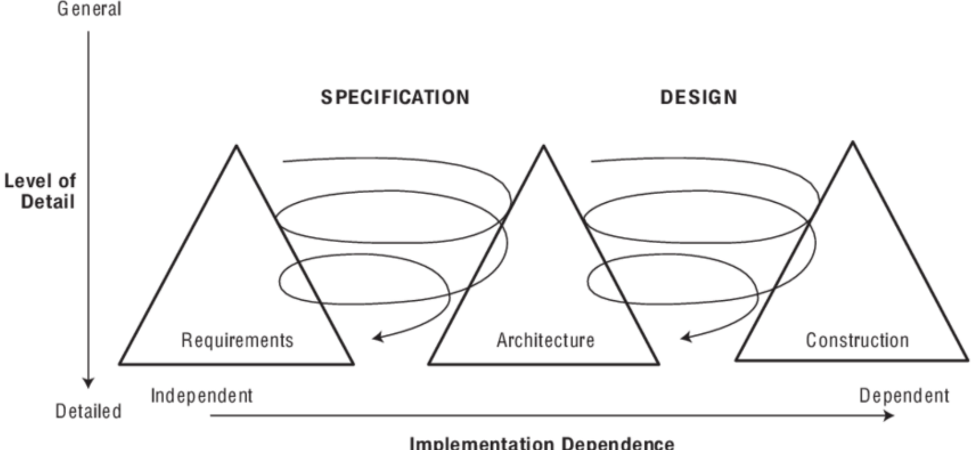


Figure 2: Three Peaks model (Woods & Rozanski, 2010)

The x-axis in the model represents the degree of dependence on the implementation. The model starts with the *requirements*; these are most often completely independent of the implementation. In the process called *specification*, the requirements are translated into an *architecture*. Next, the architecture is *constructed* in a process called *design*. The construction is, of course, implementation-dependent. The swirling arrows depict that both the specification and design processes are intertwined and cannot and should not be considered in isolation.

The y-axis in the Three Peaks Model represents the level of detail. The three triangles (peaks) start very general; as the level of detail increases, the amount of elaboration increases. This is depicted by the triangles that start small when the level of detail is general and become wider as the level of detail increases.

In this thesis, we focus on the creation of a software architecture. In the Three Peaks model, this is called specification. This chapter defines the field of software architecture. First, we elaborate on what software architecture is and how it is documented. Next, the concept of software architectural views and viewpoints is explained. We conclude this chapter by explaining the concurrency viewpoint more in-depth, as it is the most relevant viewpoint to this thesis.

2.1 What is software architecture?

Every software system has an architecture, even if it is not explicitly documented or discussed during development (Bass et al., 2003). There are, however, multiple reasons to have an explicit software architecture and define and document it during development. Some of these reasons include:

- A software system must satisfy many *business concerns*. As illustrated by the Three Peaks model earlier in this chapter, all requirements are considered by creating an architecture. This process ensures that all requirements are incorporated in the system design and prevents certain requirements from being underexposed or others from being overexposed.
- A (large) software system is often divided into multiple smaller *components*. Every component has it's own *responsibility* and *scope*. Such a system is called a *component-based software system* (Crnkovic, 2001) (more in Chapter 3). In software development, components are often created by separate teams. In order to coordinate that every separate component works when combined into one system, a software architecture is designed and maintained (Rozanski & Woods, 2012).

A formal definition of software architecture is formulated by Bass et al. (2003):

> **Definition 2.1** (Software architecture according to Bass et al. (2003))**.** The software architecture of a system is the set of structures needed to reason about the system, which comprises software elements, relations among them, and properties of both.

This definition focuses on the *documentation* of the design of a software architecture. So, a software architecture consists of a set of models (*products*) that describe the system. Often a software architecture is created before developing a system. It is, however, important to note that an architecture is not a deliverable that is finished when a project is finished. It is a (set of) document(s) that constantly changes when the system is adapted and extended.

Every decision that is made will impact other decisions as well. Sometimes, it is necessary to make a trade-off between multiple requirements. It is therefore almost always impossible to satisfy every requirement fully. Software architecture helps to create a clear picture of the potential design of a system. By creating multiple models of multiple options, the architecture can be used to reason about the system and discuss decisions in the design. By discussing multiple options, a *trade-off analysis* between all requirements is made. It is important to document these decisions (Jansen & Bosch, 2005) in a *design document*. By documenting the design decisions, we prevent that decisions can only be implicitly derived from the system or corresponding models. It captures the thinking process that leads to a certain decision. By documenting the whole decision process, and not only the outcome, knowledge vaporization is also prevented.

In this thesis, we focus on a *product-based approach* to software architecture. Therefore, we use the definition by Bass et al. (2003).

2.2 Software architectural views

When designing an architecture for a software system, many aspects need to be taken into account. Examples include the grouping of responsibilities and functionalities, the way information is stored, and the required physical hardware used to run the system.

Modeling all these aspects in one model is nearly impossible and will, at best, create a very complex model that fails to communicate the core concepts of the architecture. To represent a system in a manageable and comprehensible way, multiple models are often created and different types of models are used. These models are categorized into *viewpoints*. Viewpoints provide 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 its views (Group et al., 1999; Rozanski & Woods, 2012). The software architecture as a whole provides a *holistic view* of the system.

By using an viewpoint-approach, different concerns in the architecture are separated into multiple separate models. This allows an architect to focus on one concern in the architecture at a time. Additionally, every viewpoint defines a couple of common *pitfalls*: mistakes that are commonly made and should be taken into account when creating the models.

Different stakeholders have different interests in different viewpoints. Some viewpoints are very informative to the end-user of the system, for example, while other viewpoints mainly focus on development-specific aspects that are not directly interesting for the end-user.

Figure 3 provides an overview of commonly used viewpoints. Central in this figure is the software design. Every viewpoint is briefly described below, as defined by Rozanski and Woods (2012):

- **Context viewpoint -** The context viewpoint describes the relationships, dependencies, and interactions between the system and its environment (the people, systems, and external entities with which it interacts).
- **Functional viewpoint -** The functional viewpoint describes the system's functional elements, their responsibilities, interfaces, and primary interactions. It drives the shape of other system structures.

– **Information viewpoint -** Describes the way that the architecture stores, manipulates, manages, and distributes information. It displays a complete, but high-level, view of the static data structure and information flow.

– **Concurrency viewpoint -** The concurrency viewpoint describes the concurrency structure of the system and maps functional elements to concurrency units to clearly identify the parts of the system that can execute concurrently and how this is coordinated and controlled. This entails the creation of models that show the process and thread structures that the system will use and the interprocess communication mechanisms used to coordinate their operation.

– **Development viewpoint -** Describes the architecture that supports the software development process. Development views communicate the aspects of the architecture of interest to the stakeholders involved in building, testing, maintaining, and enhancing the system.

– **Deployment viewpoint -** Describes the environment into which the system will be deployed, including capturing the dependencies the system has on its runtime environment. This view captures the hardware environment that the system needs, the technical environment requirements for each element, and the mapping of the software elements to the runtime environment that will execute them.

– **Operational viewpoint -** Describes how the system will be operated, administered, and supported when it is running in its production environment. The aim of the Operational viewpoint is to identify system-wide strategies for addressing the operational concerns of the system's stakeholders and to identify solutions that address these.

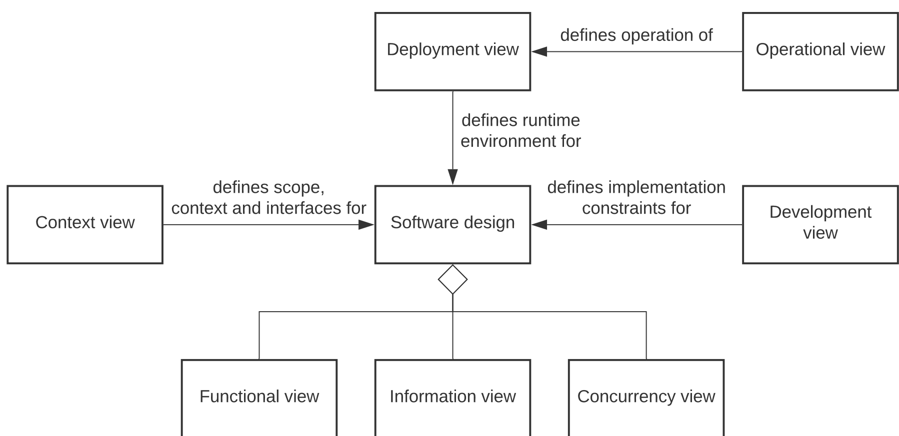


Figure 3: Relationships between software architectural views (Rozanski & Woods, 2012)

Every viewpoint has its pitfalls, but there are also some general pitfalls that architects need to be aware of when using a viewpoint approach:

– By creating a multitude of models describing the system (which the viewpoint approach promotes), the risk of inconsistency between different models is being introduced. If, for example, one model is altered this can impact other models as well. Therefore, it is important to constantly check if the alterations to one model require changes in other models.

– Furthermore, the (sometimes) large number of models can make it much more difficult to comprehend the whole system as a whole, as one needs to study a larger number of models to have an overview of the entire system.

2.3 Concurrency viewpoint

This thesis focuses on interactions between software components. Models that capture interactions between components can be categorized under the concurrency viewpoint. The concurrency viewpoint has been introduced in the previous section, but because this viewpoint is important to this thesis, this section provides a more in-depth look at the viewpoint.

The concurrency viewpoint covers multiple concerns. For this thesis, the following concerns (as proposed by (Rozanski & Woods, 2012)) are relevant and explained below: Interprocess Communication, Synchronization and Integrity, Task Failure, and Reentrancy.

- **Interprocess Communication** - Interprocess communication refers to communication between different processes. Communication within one process is quite straightforward - the process shares a common address space in memory and can communicate via this shared space. Communication between processes is more complex. The processes are not synchronized and have no direct influence on each other. There are a number of communication mechanisms that can be used to communicate between processes. These methods can be very low-level (sharing space in memory or piping output to inputs of processes for example) or more high-level (like some sort of messaging system between the processes). The higher level of communication is most relevant for this thesis.
- **Synchronization and Integrity** - Processes can work independently from each other until they need some piece of information from another process. At that point, the process needs to wait for the other process. This is called synchronization. If for whatever reason, a process does not wait for another process to provide a piece of information, integrity issues can occur. This can, for example, happen when a multi-threaded process is reading/writing to/from a shared variable. In a more high-level process structure (as used in this thesis), this issue of integrity is far less likely. The issue of synchronization, however, is a very applicable concern.
- **Task Failure** - It is inevitable that sometimes a process/task fails. When running over multiple processes, this issue can be complex. Tasks rely on each other - so when one task fails, another might not be able to continue and finish. A concurrency design must take into account that one process might not be available or not answer in a timely manner.
- **Reentrancy** - Reentrancy refers to the ability of a software element to operate correctly when used by multiple processes at a time.

Rozanski and Woods (2012) also define some common pitfalls in concurrency design. In this thesis, we discuss the following pitfalls: modeling the wrong concurrency, modeling the concurrency wrongly, excessive complexity, resource contention, deadlock, livelock, and race conditions.

- **Modeling the wrong concurrency** - In a system, there is a lot of concurrency going on. When modeling concurrency, it is important to model the correct concurrency. Often, in the general scope of a system, it is not important to model the internal concurrency of one specific process. Only the overall concurrency structure is important to a software architecture.
- **Modeling the concurrency wrongly** - When modeling concurrency, it is important to really model the concurrency and not accidentally create a state model for example. It is also possible to model a concurrency that is not possible to execute: a trigger might never occur or a certain combination of triggers is not possible.
- **Excessive Complexity** - Simplicity should always be an aim when designing a system. This is also the case for concurrency. Simple designs are easier to create, analyze, build, deliver and support.
- **Resource Contention** - Resource contention refers to one process being more heavily used than others - causing contention in one process. It is, however, almost impossible to have no resource contention. It is (realistically) not possible to have an even load over every process.
- **Deadlock** - A deadlock occurs when their processes are waiting on each other. For example, Process A is waiting for Process B, but Process B is waiting for Process A.
- **Livelock** - Livelocks are similar to deadlocks. Livelocks also entail a process that is stuck. The difference is that livelocked processes do change state, but never exit a loop.
- **Race conditions** - A race condition can be problematic when two (or more) processes attempt to perform the same action concurrently. Both processes will, for example, read a piece of state. One of the processes will reach a point in the program that alters the state, while the other process still has the old state in memory.

2.4 Conclusion

This chapter introduced the field of software architecture. The process of creating a software architecture converts a set of requirements into an architecture. Different viewpoints are used to structurally think about the system while creating the architecture. All the decisions that have been made using these viewpoints can be captured in different models. The interactions between software components are modeled in the concurrency viewpoint.

We have mentioned that the creation of multiple models introduces a high risk of inconsistency. Furthermore, we have mentioned four relevant concerns in the design of the concurrency of a system. Additionally, we elucidated multiple common pitfalls in concurrency design.

We have shortly mentioned component-based design. In the next chapter, Chapter 3, we further elaborate on this type of design.

3 Component-based software design

In the previous chapter, we have introduced Software Architecture. We have briefly mentioned component-based design as a method to split a system into manageable parts. This allows every component to have its own responsibility and scope. In this chapter, we elaborate on component-based design as a pattern in software architecture.

Components in software can be used to hide *implementation details* (Chaudron & de Jong, 2000). Components are *black-boxes* of small actions that can be composed together to perform a larger action. This idea was already mentioned by McIlroy (1968) at the first conference on Software Engineering in October 1968 in Garmisch (Germany). He argued that there should be a standard set of routines that can be used interchangeably. The routines would be classified by properties. Examples of those are precision, robustness, time-, and space-performance and size limits. This promotes re-using code and not rebuilding a "standard" routine when a new system is constructed.

In software engineering, the software pattern of component-based design embraces the ideas of McIlroy. This chapter explains what component-based software design is. Furthermore, we introduce *Service Oriented Architecture* (SOA), a commonly used method to create a component-based design. We mention some *communication protocols* that are used in SOA and mention some *implementations* of SOA. We conclude this chapter by introducing some techniques for modeling components.

3.1 What is component-based software design?

We have shortly defined a software component as a standard set of routines that can be used interchangeably. One could think of a software component as a "re-usable piece of code". This is, however, a definition on a very low level. A procedure or method could then be classified as a component: it is, after all, re-usable.

In software engineering, however, this is often thought to be too low-level (Jifeng et al., 2005). Component-based software consists of bigger components. Chaudron and de Jong (2000) have attempted to propose axioms that postulate basic assumptions about software components based on earlier literature. Based on these axioms, they have deduced corollaries that qualify components and the composition mechanism. In their definition, they use the term *composition* to express the composition of a number of components into a larger configuration. Everything outside a component is called the *environment*.

Definition 3.1 (Software component as defined by Chaudron and de Jong (2000))**.**
A software component is defined by the following axioms:

- A1 A component is capable of performing a task in isolation; i.e. without being composed with other components.
- A2 Components may be developed independently from each other.
- A3 The purpose of composition is to enable cooperation between the constituent components.

The following corollaries can be deduced:

- C1 A component is capable of acquiring input from its environment and/or of presenting output to its environment.
- C2 A component should be independent of its environment
- C3 The addition or removal of a component should not require modification of other components in the composition.
- C4 Timeliness of output of a component should be independent of timeliness of input.
- C5 The functioning of a component should be independent of its location in a composition.

C6	The change of location of a component should not require modifications to other components in the composition.
C7	A component should be a unit of fault-containment.

According to Chaudron and de Jong (2000) C1 follows from the fact that performing some task (A1) would be ineffective if there was no way to observe the effect of the component. Furthermore, the corollary can be inferred from A3: in order to achieve cooperation between different components, there must be some mechanism to facilitate interaction (defined as the input and output). We can define C2, because of A1 as well. When a component would require input from the environment, it could not perform a task in isolation. C3 up till and including C7 follow from C2. C3: if the addition or removal of a component would require the modification of other components, the component is not independent of its environment. C4: a component should be able to produce output even if the input is not as expected. C5 and C6: the component would not be independent if it or other components are dependent on the location of the component. C7: the component may not expect the input to be error-free (because it is independent of its environment as defined in C2).

Z. Liu and Joseph (1999) and Sommerville (2001) argue that in practice some of these properties do not always have to apply. For example, Barroca and Hall (2000) mention that a software component can rarely run in complete isolation because the component itself can also require input from other components. It is, thus, dependent on these components. The composition of those components can however be independent of the environment.

Another (more concise) definition that is often used is the definition by Szyperski et al. (2002). He defines software components as follows:

> **Definition 3.2** (Software component as defined by Szyperski et al. (2002))**.** Software components are a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties

In this thesis we will define software components as follows, inspired by the definitions of Chaudron and de Jong (2000) and Szyperski et al. (2002):

> **Definition 3.3** (Software component)**.** A software component implements a coherent *set of functionalities*. These functionalities should only be available to the environment via *explicitly defined interfaces*. The component should be as *independent* as possible. It may use other components, but all dependencies should be known and the dependencies may only be used via the components' interfaces.

3.1.1 Interfaces

Although the exact definition of software components has not reached a consensus, as illustrated in the previous section, both definitions that we present in this thesis mention the importance of interfaces. Interfaces are descriptions of what is needed for the component to be used in building and maintaining software systems (Jifeng et al., 2005).

Van der Werf (2011) defines four main types of interfaces, depicted in Figure 4 and explained below. We exemplify each interface with an example based on the context of our running example as introduced in Chapter 1.

- **Human interface** - The human interface defines the interface with which a user interacts with the component. This could, for example, be a GUI (Graphical User Interface) or a CLI (Command Line Interface).

  In our ATM example, this could be the (physical) interface with which the customer interacts with the ATM. This is thus the display, the pin pad, the card reader, etc.

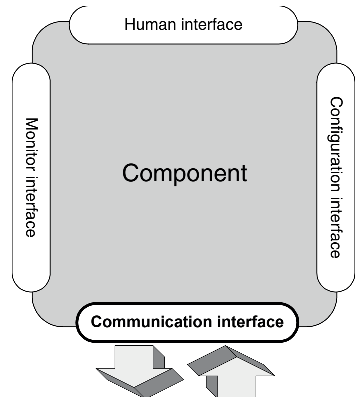


Figure 4: Component interfaces (van der Werf, 2011)

- **Configuration interface** - The configuration interface allow for setting parameters during the deployment of a component. Parameters could, for example, be the maximum allowed connections (how many human interfaces may be connected at once).

  In our example this could, for example, be the maximum cash every customer may withdraw at once at that specific ATM. An ATM at a very populated spot may have a different maximum than an ATM at a more quiet spot in order to provide all customers with cash.

- **Communication interface** - The communication interface defines how other components can communicate with the component. This interface is often called the API (Application Programming Interface). In this thesis, this interface is the most important because we look at the commuication between the components.

  In our example, this interface allows for communication between the ATM and the System.

- **Monitor interface** - The monitor interfaces allow monitoring of the usage of a component. The interface allows the logging of events (like errors and warnings) generated by the component.

  In our example, it could log, for example, when the ATM is used or how much cash is remaining in the machine. This can then be reported back to the System. The System can use this information to perform actions.

3.2 Service Oriented Architecture

Service oriented computing is a paradigm in which application components are a network of services that are loosely coupled to create an application (Papazoglou et al., 2007). *Service oriented architecture* (SOA) is a logical way / pattern of designing such systems and providing *published* and *discoverable interfaces*.

In a Service Oriented Architecture, services are offered and described by *service providers*. Services are consumed by *service consumers*. They need to be able to understand and use the services without any detailed knowledge of their implementation (Bass et al., 2003).

There is no formal definition of SOA or requirements that an architecture should exhibit in order to classify as SOA (Abuosba & El-Sheikh, 2008). Abuosba and El-Sheikh (2008), however, provide the following characteristics that should be exhibited by services in SOA:

- All the services in a particular system are *autonomous* and *self-sufficient*. Services can be *dynamically located* and invoked during runtime.
- Services support all modes of operation and are *interoperable*. Services are distributed and can be accessed over *a network*.
- Services should possess the tendency to expand and be automated.

According to Bass et al. (2003), each service runs as it's own *independent process* and has it's own *state*. The independent services communicate with each other over some sort of *protocol*. Every service can depend on other services, but every service should be able to be swapped out with another service that uses

the same communication protocol. Because every service runs completely independently, every service can be written in a different programming language, can use different hardware, can store information differently, etc.

There is a multitude of patterns that implement the SOA principle. Each of these patterns has a number of required elements (Bass et al., 2003): service providers, service consumers, a message bus, registry of services, and an orchestration server:

- **Service provider** - a service provider provides one or more services through a published interface. A service provider may also be a consumer of other services.
- **Service consumer** - a service consumer invokes services (of a service provider) directly or through an intermediary service.
- **Message bus** - a message bus is an intermediary element that can route and transform messages between service providers and service consumers.
- **Registry** - service providers register themselves to a registry of services. Consumers can use this registry to discover instances of services at runtime.
- **Orchestrator** - the orchestrator coordinates the interactions between service consumers and providers.

An example of an architecture pattern that implements SOA is called the *registry* pattern. The registry pattern can be depicted as in Figure 5. In the registry pattern, providers register themselves with the registry when they are instantiated. Clients can query the registry at runtime for the communication details of a particular service. The registry provides the details to the client and the client can then communicate directly with the service provider. In the registry pattern, there is no orchestrator. Every client communicates directly with the providers.

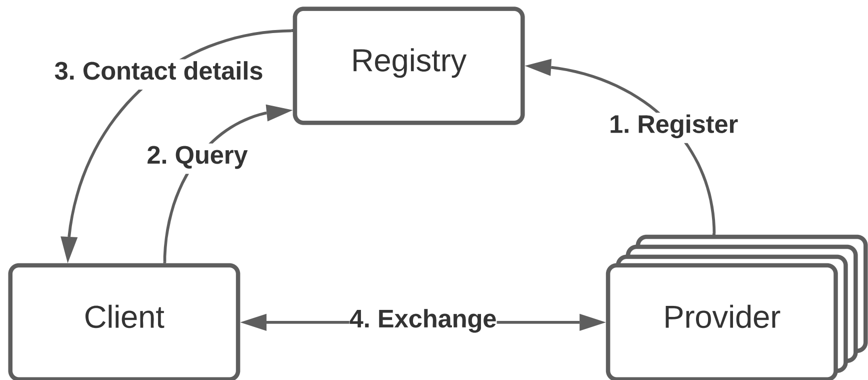


Figure 5: Illustration of the registry pattern for Service Oriented Architecture

3.2.1 Communication protocols

There are a number of communication protocols and patterns that are used in SOA. The basic types that are used are *SOAP*, *REST* and *asynchronous messaging* (Bass et al., 2003).

- **SOAP** - Simple Object Access Protocol (SOAP) is a standard protocol for web service communication. Clients and providers communicate via *XML*[3] requests and responses that are transmitted over *HTTP*[4].
- **REST** - Representational State Transfer (REST) are HTTP requests using the basic HTTP commands (GET, POST, UPDATE, DELETE). These requests (send by the client) tell the service provider to get, create, update or delete resources.

[3]More info: https://nl.wikipedia.org/wiki/Extensible_Markup_Language
[4]More info: https://nl.wikipedia.org/wiki/Hypertext_Transfer_Protocol

- **Asynchronous messaging -** In an asynchronous messaging system, participants send information but do not wait for an acknowledgment or reply. An example of asynchronous messaging is messaging over an enterprise bus.

3.2.2 Implementations

CORBA (Common Object Request Broker Architecture) is one of the first attempts at a general reusable approach to SOA. It is a vendor-independent architecture that allows applications to communicate over the network (Schmidt & Kuhns, 2000). It provides high-level interfaces that allow communication between all different types of hardware and operating systems. Applications use the Object Request Broker (ORB) as middleware to route a request from one application (client) to another application (server). CORBA has, however, some issues. One example is that (standard) CORBA doesn't allow for runtime location of services (Flissi et al., 2005; Sheikh, 2012).

Web Services are an implementation of SOA based on web technologies. A Web Service is defined by W3C[5] as "a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. A Web service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols". Web Services use SOAP as a message exchange. WSDL (Web Service Description Language) is used for describing the capabilities and requirements of a Web Service. UDDI (Universal, Description, Discover, and Integration) can be used as a registry to locate Web Services (and their descriptions) on a network.

Micro Services are a relative new approach to SOA, first introduced in 2012 (Lewis & Fowler, 2014). It is an approach to developing a single application by composing a number of small services. Each of these services run in a separate process and have their own state. Communication between the services occurs over a lightweight mechanism (often REST over HTTP). They can be defined as "an independently deployable component of bounded scope that supports interoperability through message-based communication. Microservice architecture is a style of engineering highly automated, evolvable software systems made up of capability-aligned microservices." (Nadareishvili et al., 2016). The main difference between a general Micro Service architecture and a Web Service architecture is that Micro Services can use any (predefined) protocol for communication, whereas Web Services use XML.

3.3 Modeling components

In the first section of this chapter, we have looked at the definition of component-based software design. In the previous section, we have discussed Service Oriented Architecture as an implementation of component-based software design. When using SOA (or any other implementation of component-based software design) it is necessary to be able to model the components of a system, as we have identified in the previous chapter (Chapter 2).

The components used by a software system can be modeled as a *static structure* (Hofmeister et al., 1999). The static structure is part of the functional viewpoint that was mentioned in Chapter 2. There are a large number of available types of models to model components:

- **Architecture Description Language**
  One group of languages to model components is ADL. There are a large number of languages that fall in this group. A list[6] composed by Malavolta et al. (2012) (last updated in November 2015) contains 128 languages. These languages either originated in industry or in academia. Some of the languages have open-source or commercial tool support. All languages are closely related but have slightly different architectural elements, different syntax or semantics (Malavolta et al., 2012). Most languages focus on one specific operational domain.

- **Unified Modeling Language**
  Another group of languages is based on UML. One of these languages is the UML Component Diagram (Bell, 2004; Hofmeister et al., 1999). One of the advantages of using a language based on

[5] https://www.w3.org/TR/wsa-reqs/

[6] The original list referred to in the paper from Malavolta et al. (2012) is not online anymore. A cached version can be retrieved from Archive.org: https://web.archive.org/web/20151121185404/http://www.di.univaq.it/malavolta/al/

UML is that UML models are quite simple and standardized. Many professionals are acquainted with other types of UML diagrams (Lüer & Rosenblum, 2001). For non-professionals, however, UML diagrams are harder to understand.

– **Palladio**
There are also languages mainly targeted at analyzing the static structure of an architecture. One of these approaches is Palladio[7]. Palladio is focused on predicting the Quality of Service properties of component-based software architecture.

Malavolta et al. (2012) show in their survey that neither UML, ADL, and Palladio have not one type that is widely used in industry.

As opposed to the technical approaches to modeling the static structure of an architecture, there is also a simpler *lines-and-boxes approach*. This approach might especially be useful when drawing models that are also used when communicating with non-technical stakeholders. One lines-and-boxes approach is called the Functional Architecture Model (FAM). They are also designed to express software components and modules (Brinkkemper & Pachidi, 2010).

3.4 Conclusion

In this chapter, we have defined component-based software design using the following definition: "A component implements a coherent set of functionalities. These functionalities should only be available to the environment via explicitly defined interfaces. The component should be as independent as possible. It may use other components, but all dependencies should be known and the dependencies may only be used via the components' interfaces." and have shown the different types of interfaces a component has.

Next, we introduced Software Oriented Architecture as a way of designing component-based systems. Components in a SOA should be autonomous and self-sufficient. They should be dynamically locatable and invokable. Services should be interoperable and accessible over a network. We have introduced the registry-pattern as an example of SOA. We have discussed various communication protocols and implementations that are often used in SOA-approaches.

Components in a SOA can be modeled using various modeling techniques. They all model the static structure of the architecture. We have looked at three technical models and at a lines-and-boxes approach.

We have now looked at Software Architecture (in Chapter 2) and component-based design. In this chapter, we have identified existing methods to model the static structure of a system. To model interaction between components (the dynamic structure of a system), we use different techniques. They are introduced in the next chapter, Chapter 4.

[7] https://www.palladio-simulator.com/science/

4 Existing notations to model interactions

As mentioned in the two previous chapters, software is often built in components. Communication between these components is vital to create a working system. Communication between components occurs in a particular sequence. This sequence of communication can span a multitude of components. Therefore, communication sequences and all communication in a system can become quite complicated. It is therefore important to design and document the interactions between components in order to reason about them. Furthermore, by modeling the interactions between components, it can be ensured that the designed communication can actually be implemented.

In this chapter, we first look at how to capture interactions using scenarios. Next, we take a look at different attributes that we can use to classify interaction modeling approaches. Finally, we mention current approaches and classify them by the attributes that we have introduced. With this, we can answer SQ1.

4.1 Capturing interactions using scenarios

The Oxford dictionary[8] defines a scenario as “a description of how things might happen in the future”. This is basically how scenarios are used in software engineering as well. In software engineering, scenarios are partial descriptions of a system’s and its environment’s behavior arising in situations (Benner et al., 1993). The description of a whole system thus consists of multiple scenarios, each describing a different situation that a system should handle. These scenarios can be used when designing a software architecture. The software architecture must be able to execute every scenario.

There are many ways to model scenarios. One example is a sequence diagram (Li et al., 2004). Figure 6 presents a sequence diagram of our running example. This displays the sequence of communication of the event that a customer walks up to the ATM and starts their session. The user interface shows a message to insert the bank card. When the user does so, the user interface will ask for the pin. Once the pin is entered, the bank card information, along with the pin is sent to the authentication service. The authentication service then communicates with the correct bank system to request a session token. The session token is then passed back to the user interface module, which stores the token and presents the options to the user.

There are also other (more technical) ways to model scenarios. One of these methods is choreographies. With choreographies, you can simulate the behavior and mathematically prove that a certain sequence of communication ends in the required result. *Realizability* of the software project can be ensured - i.e. the software project as designed can be constructed and is logically proven to work. A realizable software system should communicate exactly as it is specified. The realizability of a choreography and thus the system is defined by Basu et al. (2012):

> **Definition 4.1** (Realizibility as defined by Basu et al. (2012)). A choreography is *realizable* if there is a way to implement a set of components that conform to the choreography.

There is, however, not one harmonized approach to modeling choreographies. In the next sections, we look at different modeling techniques for component interaction.

4.2 Classifying component interaction modeling techniques

In this section, we introduce a set of attributes that can be used to compare approaches. We use: allows for asynchronous communication, has a visual notation, formal versus semi-formal versus non-formal, allows for compositional design and tool support. Furthermore, we classify the languages by these classification attributes: state-based versus action-based, models interactions, and implementation-specific versus implementation-agnostic. In the rest of this subsection, we describe all these attributes and mention on what theoretical basis we have selected the attribute.

[8] https://www.oxfordlearnersdictionaries.com/definition/english/scenario

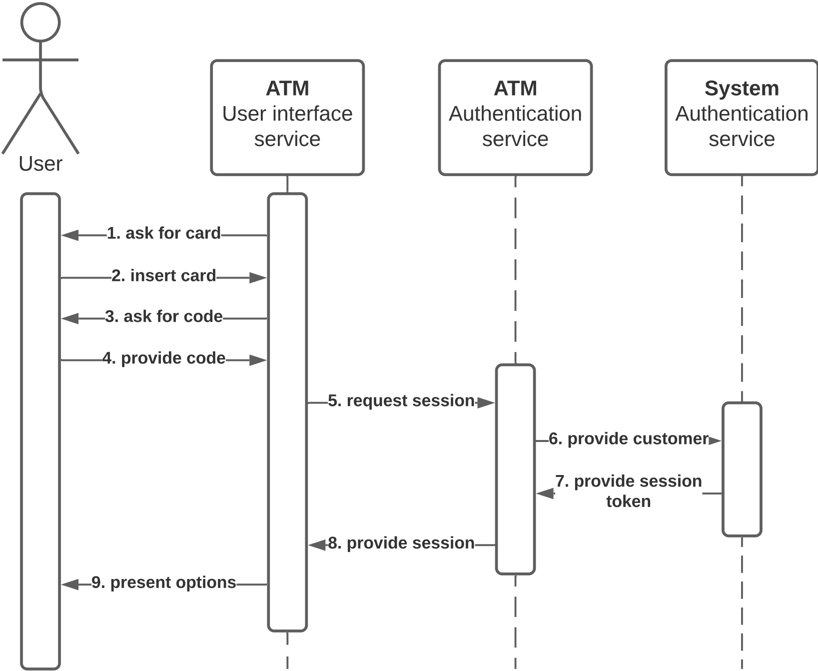


Figure 6: Sequence diagram of the authentication process of the ATM running example

Allows for asynchronous communication

There are two types of (software) communication. On the one hand, there is synchronous communication. When using synchronous communication, multiple parties establish a communication session. Once this session is established, two-way communication is possible. There is no restriction on which party may send a message. (There can be protocols in place that do define restrictions.) Every party listens for (and reacts to) messages.

With asynchronous communications, parties do not actively listen for messages. One party may send a message to another party, but that other party does not immediately have to reply to the message. There is no session for the communication.

In software architecture, both types of communication are used. Synchronous communication is necessary where a real-time answer is required. If we look at our running example, this can for example be the case with requesting the balance. The user expects the balance to show within a couple of seconds. Asynchronous communication can be used when an answer is not needed immediately. In our running example, this can be the system that requests an ATM to send a log back to the system. This reply of the ATM doesn't have to include the log; the log can be sent later asynchronously.

This attribute is based on C4 in the definition of software components provided in Chapter 3, Definition 3.1 by Chaudron and de Jong (2000): The timeliness of output of a component should be independent of its location in a composition. This corollary implies that the component should be able to allow for asynchronous communication; it should not have to rely on the input.

Has a visual notation

A model is the abstraction of the object (interaction in our case) it models. There are two main types of models: a visual model and a mathematical model. Some models are a combination of both. A visual model consists of a set of elements that have a semantical meaning in order to represent the object. A mathematical model consists of logical definitions and rules.

A mathematical model has the large advantage that it can be (automatically) validated. A purely visual model doesn't have this property but is far easier to understand. Visual models are therefore a good fit when designing interactions.

A survey by Malavolta et al. (2012) showed that a graphical representation of a model are often perceived as useful by architects.

Formal versus Semi-Formal versus Non-Formal

Another attribute of a model is its formality. Non-formal models are models with no explicit, agreed-upon, notation. This can, for example, be any boxes-and-lines model that is used when sketching an architecture.

Semi-formal models are models whose notation and semantics are formalized in a specification.

Formal models are formalized in a specification and allow for automatic formal validation. This entails that these models have a mathematical or logical basis.

The survey by Malavolta et al. (2012) showed that most organizations prefer semi-formal languages. This is because formal languages do not allow for effective communication among stakeholders in a simple way.

Allows for compositional design

As said before in this thesis, interaction models can become very large. A technique that allows for compositional design allows a model to be split up into multiple smaller models.

This attribute is based on the definition of software components provided in Definition 3.1 by Chaudron and de Jong (2000). Because we want to support these software components, we also want to be able to have a compositional design of the system.

Tool support

Every model can be represented on paper. Most models also have a notation that can be drawn in any type of general-purpose drawing application. Some modeling techniques, however, have extended tool support that provides (semantic) validation or analysis, for example.

We qualify this attribute using the following notation:

- $\times$ when no tool support is available.
- **0** when general tool support exists that allows drawing models.
- **+** when tool support exists and the tool provides some type of validation *or* analysis (that is useful for interaction analysis).
- **++** when the tool provides both validation *and* analysis.

Lago et al. (2014) highlight the importance of tool support for a modeling language.

State-based versus action-based

Every system has a state. The state of a system contains all information necessary to respond to present and future requests without reference to the history of the state. The systems' state is changed by interactions to create a new state. The state can be explicitly or implicitly represented in a model. If the state is explicitly represented in the model, we call this model *state-based*. If only the actions that modify the state are captured in the model and the underlying state is implicit, we call this model *action-based*.

Models interactions

An interaction is a form of communication or coordination between two or more components. If the model allows for capturing this type of concept, we say that the model captures interactions.

Implementation-specific versus Implementation-agnostic

Implementation-specific refers to tight coupling to some sort of implementation. Without that implementation, a model is less useful. Models that are implementation-specific, for example, can automatically generate code in a specific language. When one does not use that language, the modeling technique is far less useful.

Implementation-agnostic models are not coupled to some sort of implementation and are a general model that can be implemented in any number of techniques or languages.

4.3 Existing methods to model component interaction

In this section, we compare different existing methods to model component interaction. Table 2 provides a summary of the comparison of the different modeling techniques and their attributes. This text in this section elucidates the attributes.

	Asynch. communication	Visual notation	Formality [a]	Compositional design	Tool support	State- vs Action-based [b]	Models interactions	Implementation-agnostic
Pi-calculus	✓	×	F	✓	+	A	✓	✓
Petri nets	✓	✓	F	✓	++	S	✓	✓
WS-CDL	✓	×	S	×	0	A	✓	×
BPEL4Chor	✓	×	S	×	0	A	✓	×
BPMN	✓	✓	S	×	++	A	✓	✓
Let's Dance	✓	✓	S	✓	++	A	✓	✓
Interface automata	✓	✓	F	✓	×	S	×	✓

[a]F = Formal, S = Semi-Formal, N = Non-Formal
[b]S = State-based, A = Action-based

Table 2: Comparison of interaction modeling technologies.

Pi-calculus

Process Algebra is a family of formal languages that allows for the specification and verification of concurrent systems (Bernardo et al., 2010). Pi-calculus is one of those languages. It is a mathematical formalism that is designed to describe and analyze properties of concurrent computation (Sangiorgi & Sangiorgi, 2011). It is a very minimal formalism that does not include any primitives (like numbers, booleans, control flow statements, etc). Pi-calculus is designed to model mobile systems: systems made up of components that communicate and change their structure as a result of the interaction. Component-based systems are concurrent systems in which components communicate. Therefore, Pi-calculus can be used when analyzing a component-based system.

It is beyond the scope of this thesis to explain how a component-based system can be translated to a Pi-calculus model that can be used for analysis. It is, however, important to note that once components and their interactions (choreographies) are formalized using Pi-calculus, this method can automatically verify the realizability of the choreographies (Abouzaid, 2006; Deng et al., 2006).

Pi-calculus supports asynchronous communication (Beauxis et al., 2008). It does not have a visual notation; it is formalized using logical expressions. Pi-calculus does allow for a compositional design (every component can be modeled as a separate expression). The model is action-based as it only models the changes in state. Pi-calculus allows for modeling interactions between components. It is an implementation-agnostic approach as it has no direct implementation. There are a number of tools that support (a subset of) Pi-calculus (not specific to choreography analysis). Some examples include PICASSO[9] and The mobility workbench (Victor & Moller, 1994).

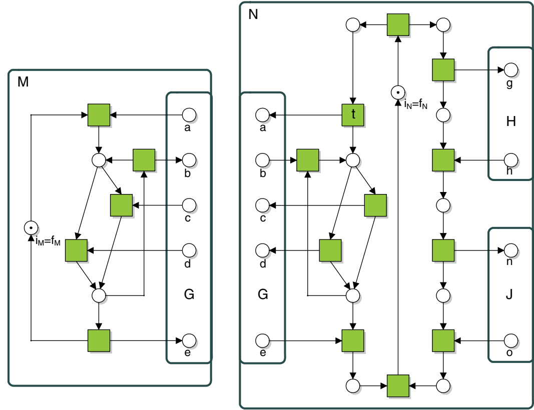
Petri nets

Petri nets pose a middle ground between mathematical definition and visual notation (Petri, 1962): they have a large mathematical basis, but can also be represented visually. Petri nets model the state of a system alongside functions (transitions) that alter the systems' state (Reisig, 1985).

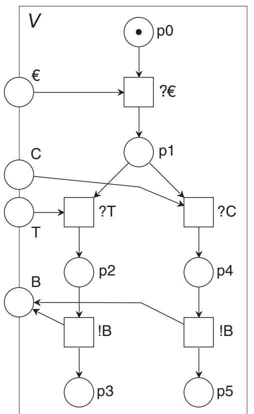
[9]https://dzufferey.github.io/picasso/manual.html

To model a choreography with a Petri net, the functioning of every component is implemented in a Petri net using a Workflow net. Workflow nets are a special class of Petri nets that have special properties. They always have one start and one end place. Furthermore, when the end place is reached, the rest of the Petri net should be empty (only one token may exist in the end-place). In order to model a whole choreography, the workflow nets of the different components need to be connected to each other to allow interaction / communication (one place is the output of one of the Petri nets and that same place is the input for another process model). By doing this, the whole system is composed.

Two very similar approaches to this idea exist. One of the approaches, by K. M. van Hee et al. (2010), calls the *output places* (the places that are connected to other components) *ports* (example of syntax in Figure 7a). The approach by Massuthe et al. (2005) calls the approach Open Workflow nets (example of syntax in Figure 7b). This approach is used by Van Der Aalst et al. (2010) to model a contract between two or more parties. The open workflow net extends the standard workflow nets by providing communication places. These communication places can be connected to other components.



(a) Approach as proposed by K. M. van Hee et al. (2010)



(b) Approach as proposed by Massuthe et al. (2005)

Figure 7: Syntax examples of Petri net Choreography approaches

Petri nets allow for asynchronous communication and allow for a compositional design. Petri nets are not implementation-specific. A special attribute of Petri nets is that they model both actions (transitions) and the systems' state (as tokens in places), Petri nets are thus state-based. It is also possible to combine multiple Petri nets to allow for modeling interactions. There are a large number of tools that allow modeling and analysing Petri nets (Thong & Ameedeen, 2015). In this thesis you will find modeled Petri nets. Those are modeled with YASPER (K. van Hee et al., 2006).

WS-CDL

Besides the logical and mathematical languages we have presented thus far, there are also technical modeling languages for choreographies. One of these technical modeling languages is called WS-CDL[10]. It is an XML-based language to describe choreographies and can be used to directly implement the choreography in a system. WS-CDL does not have an official visual notation, so it is hard to understand by non-technical actors and not suited for communication about the architecture. WS-CDL can be categorized as a semi-formal language, as it does provide a specified syntax (specified by W3C) but does not have a mathematical basis that allows for automatic validation. Some tools support modeling and generating WS-CDL. One of these tools is CDLChecker (Madiesh & Wirtz, 2008). It uses a BPMN model to generate WS-CDL.

Because of the lacking mathematical basis of WS-CDL, it is not possible to directly analyze a WS-CDL model. There are, however, some approaches to translate a WS-CDL specification into languages that do support analysis. The approach by Abouzaid (2006) supports translating a WS-CDL specification into Pi-calculus. Le and Truong (2012) propose a method to translate WS-CDL specifications into Event-B.

[10] https://www.w3.org/TR/ws-cdl-10/

As demonstrated, WS-CDL does have a visual notation and allows for analysis. But in order to have these attributes, it does require conversion into other languages. WS-CDL does support asynchronous communication. WS-CDL does not implicitly model the state of the system, only the actions. It does allow for modeling interactions between different components. It is not implementation-agnostic, as it can be directly used to implement applications.

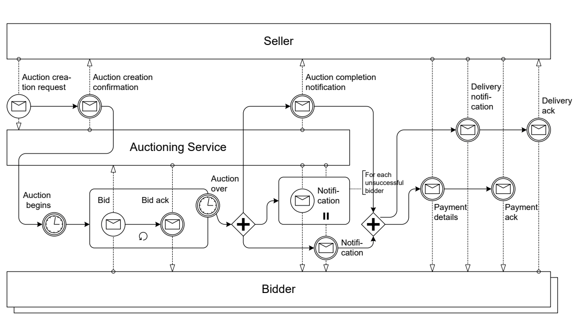
BPEL4Chor

Another technical modeling technique is called BPEL4Chor (Decker et al., 2007). It is an extension to BPEL (WS-BPEL). BPEL is a standard that is used to describe long-running business processes. Compared to WS-CDL, BPEL4Chor is less implementation-specific. It decouples the communication from the technical configuration.

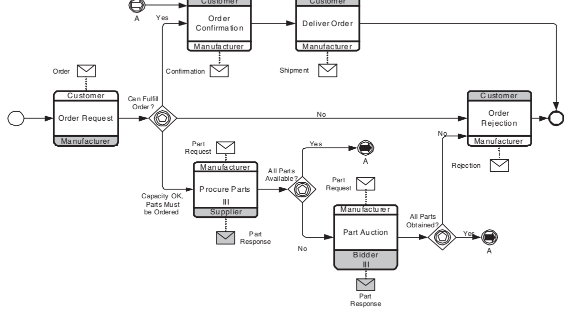
It does have the same limitation as WS-CDL in that it does not have an official visual notation. Additionally, BPEL4Chor cannot be analyzed directly. There are some proposed techniques, like the one from Lohmann et al. (2007), that allow conversion from BPEL4Chor models to Petri nets for analysis.

BPMN

In contrast to the mathematical and technical modeling techniques, there are also purely visual modeling techniques. One branch of modeling techniques is based on BPMN[11] (Cortes-Cornax et al., 2011). BPMN is the de-facto standard for modeling (business) processes. Some choreography-specific versions of BPMN have been developed, such as iBPMN (Decker & Weske, 2011) (syntax example in Figure 8a). iBPMN extends BPMN in order to focus more on the interactions between services and not on the functioning of the services themselves. BPMN 2.0 has it's own notation to model choreographies[12] (syntax example in Figure 8b). It includes choreography tasks that represent two components interacting. One of the components is the initiating party and the other component receives the message. These messages are also explicitly modeled. It is possible to model one-way interactions (only the initiating party sends a message) and two-way interactions (the receiving party also replies to the message).



(a) iBPMN as proposed by Decker and Weske (2011)



(b) Choreography model as specified in BPMN 2.0 specification

Figure 8: Syntax examples of BPMN notations

iBPMN and BPMN 2.0 can be classified as semi-formal languages. The syntax is specified, but not based on a mathematical definition for automatic validation. It is possible, however, to validate the syntax of the model. Both iBPMN and BPMN 2.0 are completely implementation-agnostic and have plenty of tool support as it is possible to use general BPMN tools. Additionally, Poizat and Salaün (2012) have created a tool that allows for modeling BPMN 2.0 choreographies in Eclipse and that provides verification and realizability checking.

Let's Dance

There are also visual modeling approaches that are not based on BPMN. Let's Dance (Zaha et al., 2006) is specifically designed for choreography modeling. It focuses solely on the interactions in a choreography

[11] https://www.bpmn.org/
[12] https://www.omg.org/spec/BPMN/2.0/About-BPMN/

and not on the implementation details of components. The language is specifically targeted at software architecture and analysis. A special thing to note about Let's Dance is that its syntax explicitly shows if a message is acknowledged or not. An example of its syntax can be found in Figure 9.

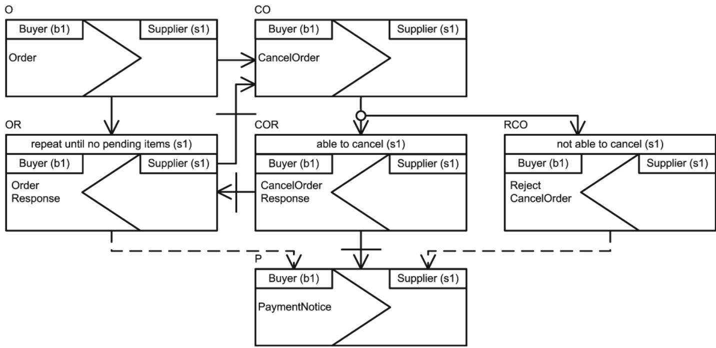


Figure 9: Example of the syntax of Let's Dance, image from Decker et al. (2008)

Let's Dance supports asynchronous communication. It can be classified as a semi-formal language. The syntax is defined in (Zaha et al., 2006), but it does not have a mathematical basis. Once again, Pi-calculus can be used to perform some analysis on the choreography (Decker et al., 2008). Let's Dance allows for a compositional design. It only models the actions and not the state of a system. It does allow for modeling interactions between components. Let's Dance is completely implementation-agnostic. Let's Dance has a tool specifically developed to model and analyze Let's Dance models: Maestro for Let's Dance (Decker et al., 2006).

Interface automata

Interface automata (De Alfaro & Henzinger, 2001) are a formal way of visually and mathematically modeling components. It can be used in the design of software components, but can also be used for validation of the system. Figure 10 presents a syntax example of interface automata.

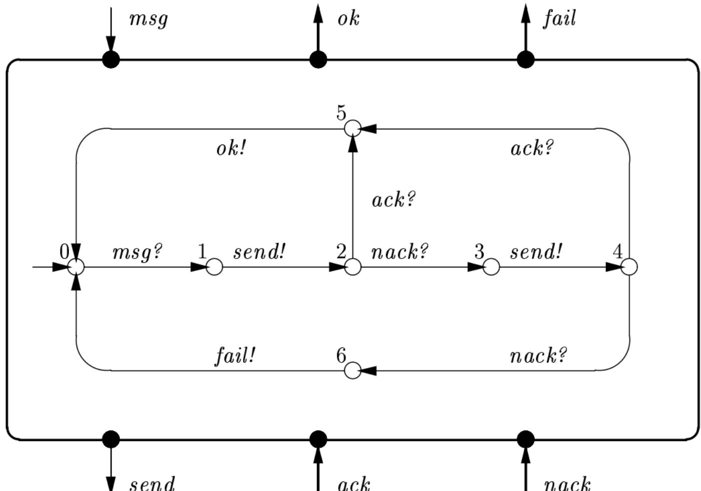


Figure 10: Syntax example of interface automata (De Alfaro & Henzinger, 2001)

Interface automata support asynchronous communication and have a visual notation. The modeling language can be classified as formal. The syntax and logic are defined in (De Alfaro & Henzinger, 2001). It allows for a compositional design. Interface automata explicitly model state. They do not directly allow modeling interaction of multiple components. The language is implementation-agnostic. There is no specific tool support for interface automata.

In this section, we have seen different approaches to modeling component interaction. With the comparison, we can answer SQ1.

SQ1: *What are current methods to model interaction?*

We have discussed seven methods to model component interaction: Pi-calculus, Petri nets, WS-CDL, BPEL4Chor, BPMN, Let's Dance, and Interface automata. We have presented attributes to classify these methods.

4.4 Modeling consistency between choreographies

As discussed in Chapter 2, the documentation of an architecture cannot be expressed in one model. Models that capture the interaction of a system can often also not be expressed in just one model. Therefore, a multitude of models is needed. Additionally, most of the time a component does not communicate with only one other component, but with multiple components. In Chapter 1 we have classified these interactions as complex interactions.

Each and every communication that is displayed in a model must be compatible with all other modeled interactions. When creating a new model it is initially hard to take every earlier created model into account. Therefore, new models are often correct on their own but break down when combined with the other models. Furthermore, when earlier models are changed, all other models have to be checked for consistency.

The current methods that we have identified in SQ1, are sufficient methods to model single interactions. It is, however, not possible to prove consistency between the single interactions with these methods. We have not seen a language that allows for explicitly compose different interactions without creating one very large model.

4.5 Conclusion

In this chapter, we discussed software scenario's as a way to model communication in a software architecture. There are a number of different approaches to modeling interaction (which we have identified in SQ1). All interaction-modeling methods, however, have in common that they cannot express the communication of a set of models. Therefore it is not possible to show consistency between different choreographies.

In the next chapter, Chapter 5, we introduce Interaction Oriented Architecture. INORA is our response to the limitation of current approaches to model complex interactions.

5 INORA: Interaction Oriented Architecture

In the previous chapters, we have established that there are various methods to model the static and dynamic structure of a system. We have seen, for example, the Functional Architecture Model in Chapter 3 that clearly visualizes the components of a software system. Furthermore, we have seen various models that document the behavior of a system in Chapter 4; for example the BPMN Choreography Diagrams and Petri nets. BPMN Choreography Diagrams have a large advantage in that they are visual, whereas Petri nets provide a formal language and can be used to analyze the system.

In this thesis, we aim to create a systematic approach to design and analyze complex interactions between components. Therefore, we have to both model the static and the dynamic aspects of a system in one model. As the previous chapters have shown, such a method does not currently exist. We introduce **In**teraction **Or**iented **A**rchitecture (INORA for short) to create a method that does allow for modeling both the static and dynamic aspects in one model as a response to SQ2. INORA is a set of models that model both the static and the dynamic structure of an architecture. It consists of the following components:

1. **The Interaction Model -** The Interaction Model defines both the static and dynamic structure of a system: the organization of components and the allowed interactions between them.
2. **A set of Protocol definitions (represented as BPMN Choreography Diagrams) -** The BPMN Choreography Diagrams specify the interactions as Protocols in the Interaction Model: how do the components interact.

In Figure 11, the domain model of INORA is represented. It consists of three parts: the Interaction Model, the BPMN Choreography Diagram, and the Representation. In this chapter, we will zoom into the different parts. We first start with introducing the Interaction Model by providing the conceptual model. Next, we discuss the notation and provide some examples. Thereafter, we introduce Protocols (which we model as BPMN Choreography Diagrams). We conclude this chapter by providing an example of INORA by creating the Interaction Model and various Protocols for our running example.

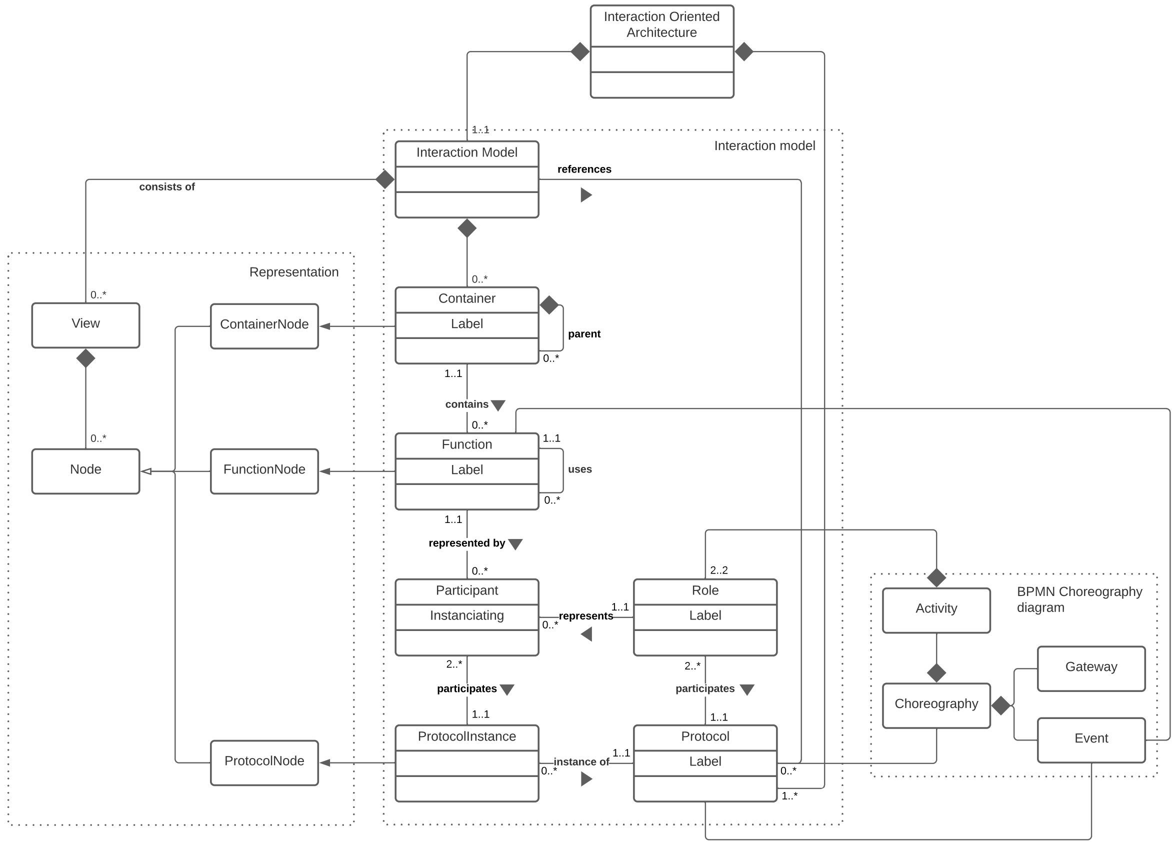


Figure 11: Domain Model of INORA

5.1 The Interaction Model

In this subsection, we introduce the Interaction Model as part of INORA. Firstly, we present the meta-model of the Interaction Model. Then, we discuss the semantics of the meta-model. Next, we discuss the syntax and notation. Lastly, we discuss the pragmatics of the Interaction Model. In Chapter 6 we discuss the formal semantics of the Interaction Model.

5.1.1 Elements

We introduce the Interaction Model to define both the static and dynamic structure of a system. This allows for defining the organization of components and displaying the allowed interactions between those components. It provides an overview of the organization and the interactions in the system to create a clear picture of the system as a whole. The *Interaction Model* consists of three main elements: *container*, *function* and *protocol*. There are some additional elements because *protocols* can be reused. The meta-model of the Interaction Model is depicted in Figure 12.

Every *Interaction Model* consists of zero or many *containers*. Every *container* contains either one or many other *containers* **or** one or many *functions* (*C2*). We define the parent, where parent(a , b) entails that a is the parent of b. The container cannot contain itself or any of its parents (*C1*). Every *function* can use zero or many other *functions*, where uses(a , b) entails that a uses b. It can, however, not use itself or any other function that it is dependent on (*C3*). If a function is used by another function we call this function *dependent*, otherwise we refer to it as being *independent*. A function can only use a function within the same container (*C4*).

The *Interaction Model* references zero or many *protocols*. Every *protocol* has at least two and at most many *roles*.

Every *function* can participate in a *protocol instance* through the *participant*. All *functions* that participate in the *protocol instance* have to be in a different *container* (*C5*). One *participant* in the relation between *protocol instance* and *participant* can instantiate the *protocol instance* (*C6*). The *protocol instance* is an instance of the *protocol*. The *participants* fulfills a *role*. The terms for *participant* and *role* are taken from Decker and Weske (2011). A function (participant) can only instantiate one protocol (*C7*).

A function cannot interact with any function that any of its using-ancestors is dependent on or any function that participates in a protocol of any of its ancestors (*C8*).

The *container*, *function*, *protocol* and *role* all have a *label* attribute. This label is displayed in the graphical notation of the Interaction Model.

The participate or *uses* relation on the function may not be cyclical. That would create a dependency loop. To be able to formally express this, we define an additional relation on functions:

$$interacts_with = \{(a, b) \mid (a, b) \in IW \wedge (b, a) \in IW\}$$

where:

- $(a, b) \in IW$ defines whether a function a uses function b or interacts with function b via a protocol, i.e.,
  $IW = \{(a, b) \mid (a, b) \in uses \vee \exists\, p \in \text{ProtocolInstance} : \text{participates}(a, p) \wedge \text{participates}(b, p)\}$
- $(a, p) \in participates$ lifts the participates relation from Participants to Functions, i.e.,
  $participates = \{(a, p) \mid \exists\, x \in \text{Participant} : (x, p) \in participates \wedge (a, x) \in represented_by\}$

Using the relations expressed in Figure 12 and the additional relation, we define the following constraints on the Interaction Model:

C1 **The transitive closure of *parent* is irreflective.**

C2 **A Container can either contain other Containers or one or multiple Functions.**

$\forall c \in \text{Container} : (\exists f \in \text{Function} : \text{contains}(f, c)) \implies \neg(\exists d \in \text{Container} : \text{contains}(c, d))$

C3 **The transitive closure of *uses* is irreflective.**

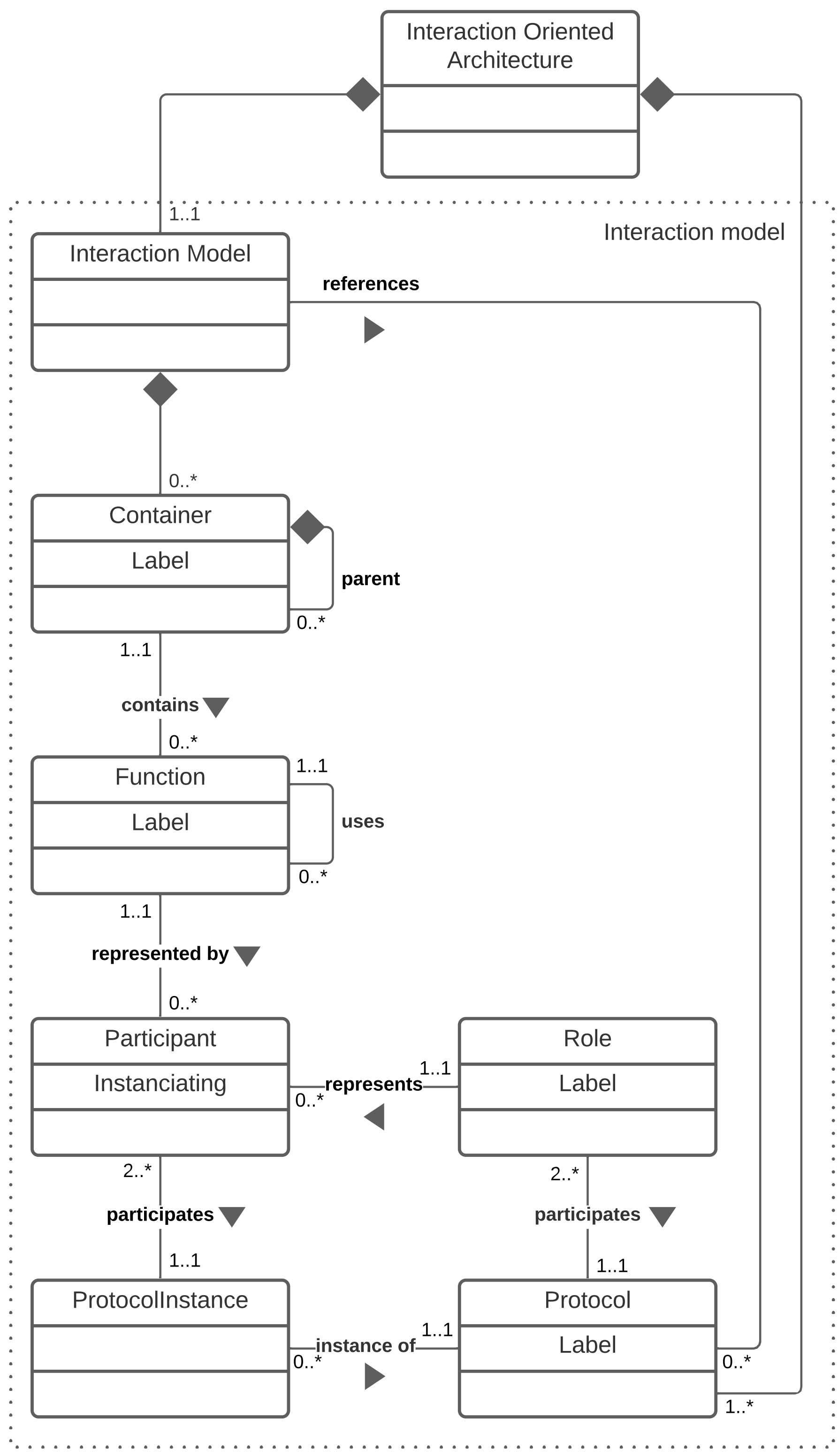


Figure 12: Conceptual model of the Interaction Model (zoomed in from Figure 11)

C4 **A uses-relation can only be between two Functions that are contained in the same Container.**

$\forall f, g \in \text{Function},\ c \in \text{Container} : (\text{contains}(\ c\ ,\ f\ ) \land \text{uses}(\ f\ ,\ g\ )) \rightarrow \text{contains}(\ c\ ,\ g\ )$

C5 **Every Function that is represented by a Participant that participates in the same ProtocolInstance has to be in a separate Container.**

$$\begin{aligned}&\forall p \in \text{ProtocolInstance},\ partA, partB \in \text{Participant},\ f, g \in \text{Function},\ c \in \text{Container} :\\ &\quad (\ \ \text{participates}(\ p\ ,\ partA\ ) \land \text{participates}(\ p\ ,\ partB\ ))\\ &\quad \land\ \text{representedBy}(\ partA\ ,\ f\ )) \land \text{representedBy}(\ partB\ ,\ g\ )) \land \text{contains}(\ c\ ,\ f\ ) \land f \neq g\\ &\quad)\ \implies \neg\,\text{contains}(\ c\ ,\ g\ )\end{aligned}$$

C6 **At most one Participant can be instantiating per ProtocolInstance.**

$$\begin{aligned}&\forall p \in \text{ProtocolInstance},\ partA, partB \in \text{Participant} :\\ &\quad (\ \ \text{participates}(\ p\ ,\ partA\ ) \land \text{participates}(\ p\ ,\ partB\ )\\ &\quad \land\ partA.instanciating \land partB.instanciating \land partA = partB\\ &\quad)\end{aligned}$$

C7 **A Function that is represented by a Participant can only instantiate one protocol.**

$$\begin{aligned}&\forall f \in \text{Function},\ part, partX \in \text{Participant} :\\ &\quad \text{representedBy}(\ f\ ,\ part\ ) \land part.instanciating\\ &\quad \implies (\text{representedBy}(\ f\ ,\ partX\ ) \implies \neg partX.instanciating)\end{aligned}$$

C8 **The transitive closure of** *interacts_with* **is irreflective.**

5.1.2 Notation

We have presented the meta-model of the Interaction Model. In this section, we present the syntax of the Interaction Model. As discussed in Chapter 2, when modeling systems, it is often useful to have multiple views of the same system in order to highlight different aspects. The Interaction Model supports having multiple views or representations of one Interaction Model definition. This is depicted in Figure 13.

The meta-model illustrates that an *Interaction Model* can be represented by zero or more *Views*. Every *view* contains zero or more *Nodes*. The *ContainerNode* represents a *Container*, the *FunctionNode* represents a *Function* and the *ProtocolNode* represents the *ProtocolInstance*. This allows us to hide certain nodes from a view. It is, for example, possible to hide entire containers in a view to only show a certain part of the system.

In Table 3, every element in the Interaction Model is displayed. A basic example of the Interaction Model can be found in Figure 14. Boxes K, L, M and O depict *ContainerNodes*. The *ContainerNodes* are drawn as large rounded boxes with a black border that have the label in the top left of the box. Please note that in principle, it is possible to draw an outermost container around boxes K, L, and M. It is, however, not required to draw this outermost container.

The smaller (non-rounded) boxes F, $F1$, $F2$, $F3$ represent *FunctionNodes*. The labels of functions are horizontally and vertically centered in the box. Functions can be placed anywhere in a container. *ProtocolIntances* A and B are represented in the figure by the small circles with the label in the absolute center. The dashed arrow between $F1$ and $F3$ depicts a using-relation that can be read as follows: $F1$ *uses* $F3$. The protocol between $F0$ and $F1$ can be read as follows: *Protocol A is instantiated by* $F0$ *and involves* $F0$ *and* $F1$. The protocol between $F2$ and $F1$ is read as follows: *Protocol B is instantiated by* $F2$ *and involves* $F2$ *and* $F1$.

In this example the labels are arbitrary, but in practice, we recommend using names as labels. If a protocol label exists more than once in the diagram, it refers to the same *Protocol* because multiple *ProtocolInstances* of the same *Protocol* can exist.

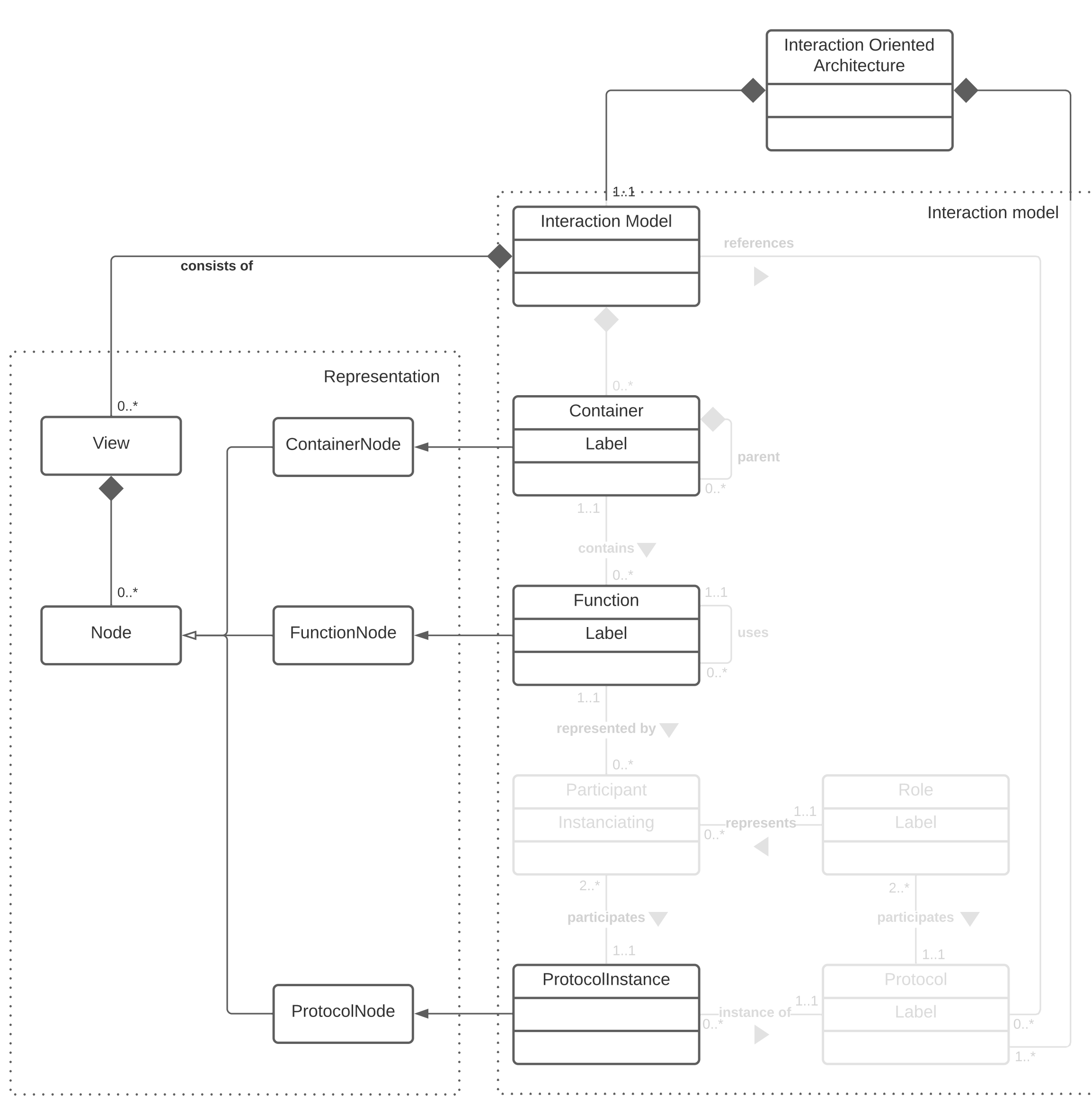


Figure 13: The Interaction Model with views (zoomed in from Figure 11)

Element	Description
K	An empty Container K
F	A function F
A	A Protocol A
F → A	A Function F that instantiates Protocol A
A — F	A Function F that participates in Protocol A
F1 ⇣ F2	A Function $F1$ that uses Function $F2$

Table 3: The elements of the Interaction Model

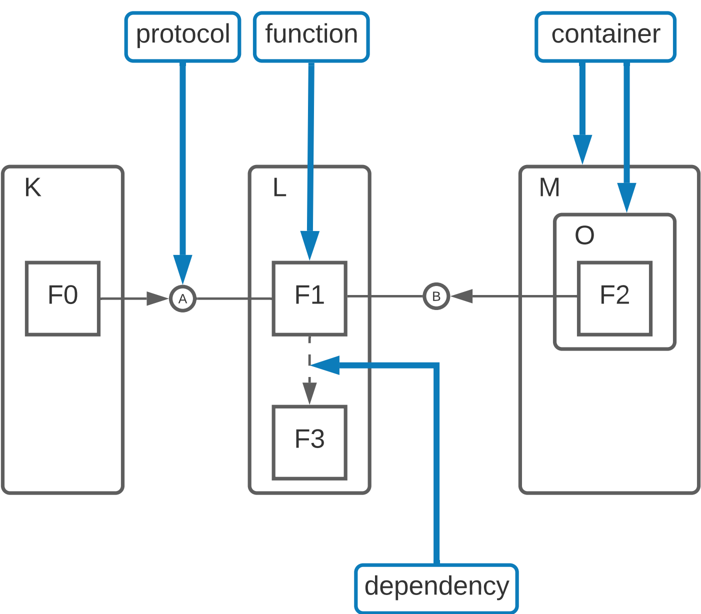


Figure 14: The elements of the Interaction Model

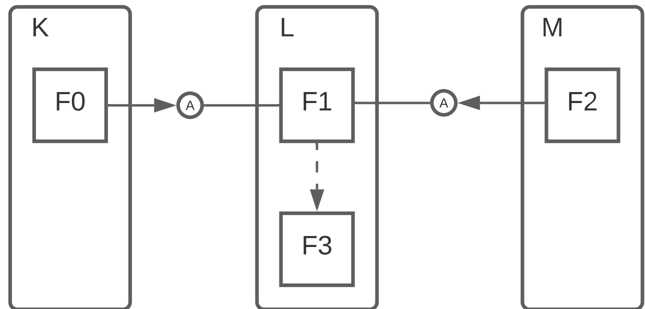


Figure 15: Example Interaction Model with two protocol instances of the same protocol

We read the Interaction Model in Figure 15 as follows: We have three containers: K, L and M. Function $F0$ is contained in container K, functions $F1$ and $F3$ are contained in container L and function $F2$ is contained in container M. In this Interaction Model, only one protocol exists: protocol A. Note that, compared to Figure 14, we have two instances of the same protocol here. The interaction between $F0$ and $F1$ should be read as follows: Function $F0$ instantiates protocol A, both function $F0$ and $F1$ participate. Function $F2$ can also instantiate protocol A. If function $F2$ instantiates the protocol, we read it as follows: Function $F2$ instantiates protocol A, both function $F2$ and $F1$ participate. The dependency is read as: function $F1$ uses function $F3$.

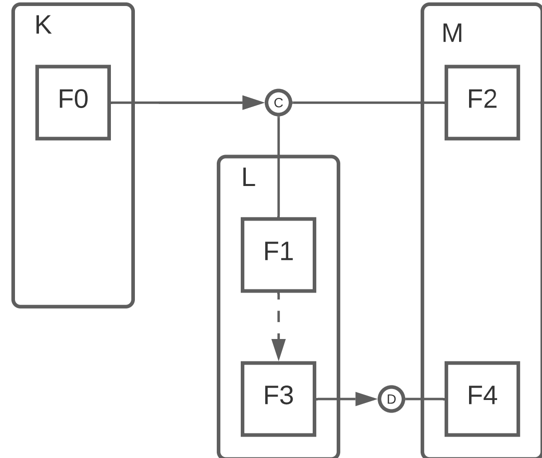


Figure 16: Example Interaction Model with a protocol with three participants and a dependency that instantiates a protocol

We read the Interaction Model represented in Figure 16 as follows: We have three containers: K, L and M. Function $F0$ is contained in container K, functions $F1$ and $F3$ are contained in container L and functions $F2$ and $F4$ are contained in container M. In this Interaction Model, two protocols exists: protocol C and protocol D. The first protocol should be read as follows: protocol C is instantiated by $F0$, $F1$ and $F2$ participate in the protocol. The dependency is read as: function $F1$ uses function $F3$. Next, the protocol that is instantiated by $F3$ is read as follows: function $F3$ instantiates protocol D, $F3$ and $F4$ participate.

The formal semantics of how to interpret the Interaction Model in Figure 15 and Figure 16 is explained in Chapter 6.

5.2 Protocols

As illustrated in Figure 11, INORA consists of the Interaction Model (that was presented in the previous section) and a set of Protocol definitions. Protocols can be expressed in any modeling language that allows for the following concepts:

1. A message with a sender and a receiver.
2. A notion of choices in the execution path of the protocol.
3. A modeling element that can represent a using-relation or another protocol.

If a using-relation between two functions exists (A uses B), the used function (B) has to be represented as an intermediate event in all Choreographies in which the dependent (A) occurs. In Figure 17, F1 has to be represented in Protocol A (because F0 uses F1).

Furthermore, when a Function that participates in a Protocol (A) also initiates another Protocol (B), that Protocol (B) has to be represented in Protocol A. In Figure 17, B has to be represented in Protocol A (because $F2$ initiates B). In Figure 16, D has to be represented in Protocol C (because $F3$ is used by $F1$ and $F3$ initiates D).

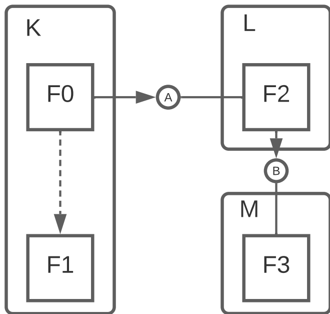


Figure 17: Example of intermediate events in an Interaction Model

In Chapter 4 we presented the BPMN (2.0) Choreography Modeling language. BPMN is in the basis a language to model processes. BPMN models start in one or multiple start events and end in one or multiple end events. Between those start and end events, a process is captured using a number of elements. The most basic element is the *Task*, which represents a task in the process to perform. Additionally, to provide concurrency and choice in the process, gateways can be placed in the process. AND-Gateways signal the parallel execution of a part of a process. XOR-Gateways signal a choice in the process: only one of the outgoing branches of the XOR-Gateway may be executed.

BPMN 2.0 introduced a specification for modeling choreographies. In this thesis, we will be using a subset of this language with small adaptions to express our protocols. The BPMN specification[13] mentions the following about Choreography Diagrams:

> A **Choreography** is a type of process but differs in purpose and behavior from a standard **BPMN Process**. [...] **Choreography** formalizes the way business *Participants* coordinate their interactions. The focus is not on orchestrations of the work performed *within* these *Participants*, but rather on the exchange of information (**Messages**) between these *Participants*.

The required elements of a protocol modeling language presented above are implemented as follows in our BPMN Choreography Diagram:

1. For the *Message*, the sender of the message is represented in the top band of the activity and the receiver in the bottom band. We adapt the contents of the *Message* a bit from the standard BPMN specification. In the specification, the contents of the message are represented by drawing

[13] https://www.omg.org/spec/BPMN/2.0/PDF, page 315

an envelope above the activity (with a possible reply message). We have chosen to present the message in the content of the activity itself. This makes it very clear which participant sends a message to which other participants.

2. Our notation of the BPMN Choreography Diagram does only allow the use of XOR-gateways and AND-gateways. Other types of BPMN Gateways are not supported.

3. We use BPMN intermediate-events to represent other Protocols and using-relations.

Figure 18 presents the symbols of the BPMN Choreography Diagram that we use in this thesis. We use the standard BPMN start and end events to signal the start and the end of the choreography. It is allowed to have multiple start and end events.

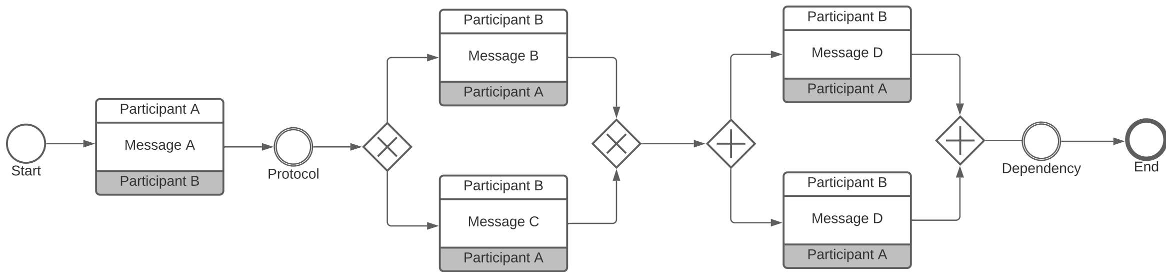


Figure 18: Elements used for the expression of Protocols in the Interaction Model

Figure 19 shows how Protocols are represented in BPMN Choreography Diagrams. Every *Protocol* has a *Choreography*. In Figure 18 we can identify *Message A*. This message has a sender and a receiver: the composition relation between *Activity* and *Role*. The two *intermediate event*s in Figure 18 refer to a *Function* and a *Protocol*.

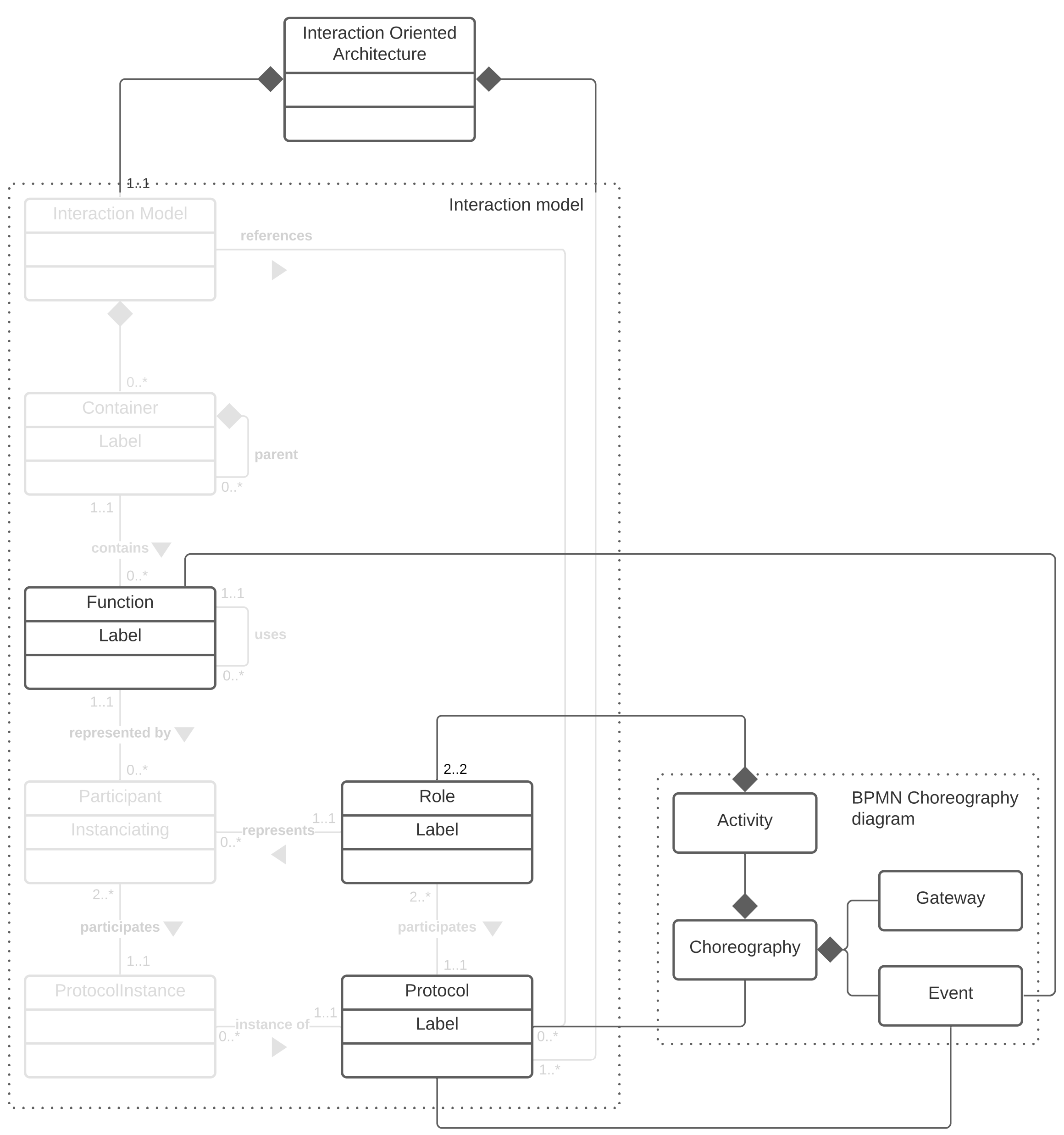


Figure 19: Relation between the Interaction Model and BPMN Choreography diagrams

5.3 Running example

In this section, we present an example of the Interaction Model, as it can be used in practice. We go back to our running example of the ATM (Chapter 1). Consider the following part of the system: when someone interacts with an *ATM*, a session with the *Bank Authority* needs to be established through the *System*. These three participants in the example are all separate entities. We, therefore, translate them to separate containers in our Interaction Model (Figure 20). The interactions between the functions in the Interaction Model are displayed in the BPMN Choreography Diagrams in Figure 21.

The *ATM* has the *User Interface Service* that allows the user to interact with the *ATM*. This interface presents the different options to the user. Once the user inserts the bank card, the *User Interface Service* needs to initiate a *session request* through Protocol A in the ATM. In Protocol A, the session is requested to the *Authentication service* in the *ATM*. In order to do this, the *System Authenticate* function is invoked. After that, the request is either accepted (and a session token is sent) or denied (and a failure message is sent).

The *System Authenticate* function in the *Authentication Service* in the *ATM* needs to send the bank card information to the system and validate the pin code to establish a session with the bank authority. It does so in Protocol B; it requests the session at the *System* in the *Handle ATMauth request*. As represented in the Protocol, first Protocol C is instantiated in order to *Validate [the] ATM* in the *ATM Validation Service*. Protocol C is a simple protocol between *Handle ATMauth request* and *Validate ATM*, that either result in a success message or a failure message. After the execution of Protocol C in Protocol B, the *Bank Authenticate* function is invoked in the *Authentication Service* in the *System*. This function invokes the *Handle auth request* function in the *Authentication service* of the *Bank authority* through Protocol D. Protocol D either returns a session token or a failure message. Once Protocol D is executed, Protocol B can return either a session token or a failure message to *System Authenticate* in the *Authentication service* in the *ATM*.

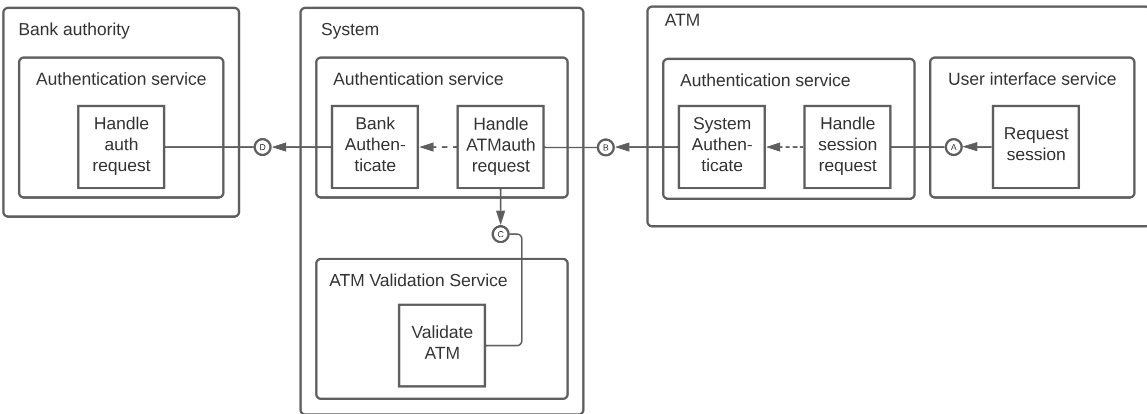
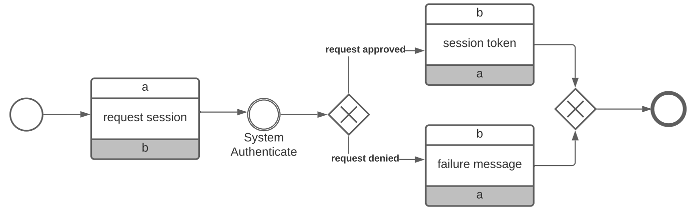
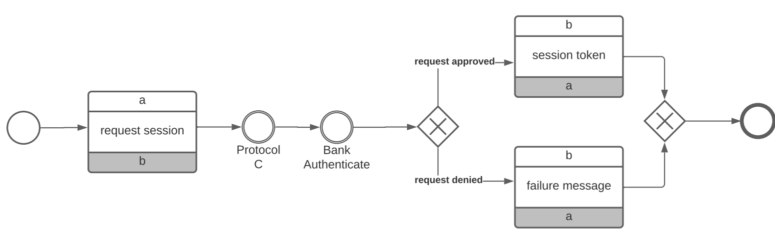


Figure 20: Interaction Model of the ATM running example

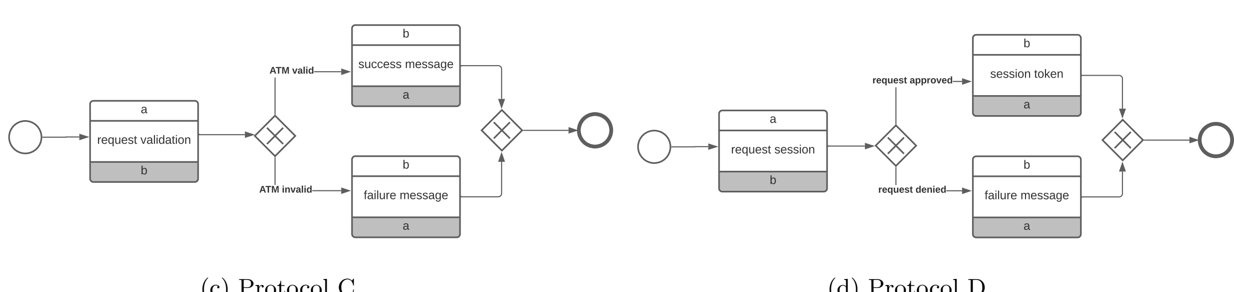
Important to note is that in the Interaction Model, we cannot express multi-instance. To put it in the words of our running example: we cannot express that we have multiple Bank Authorities and ATMs. We will further discuss this problem in Chapter 8.



(a) Protocol A



(b) Protocol B



(c) Protocol C (d) Protocol D

Figure 21: Protocols belonging to the ATM running example

5.4 Conclusion

In this chapter, we have presented INORA. INORA provides us the answer to SQ2.

> SQ2: *What are the concepts and relations required to design complex interactions between components?*

The Interaction Oriented Architecture (INORA) consists of the Interaction Model and Protocols. For the Interaction Model, we have discussed the notation and showed examples of the syntax. Additionally, we have introduced the BPMN Choreography notation that we use in this thesis.

In the next chapter, Chapter 6, we will look at the semantics of INORA that allows us to compose a system and perform analyses on this system.

6 The semantics of the Interaction Oriented Architecture

In the previous chapter, we have presented INORA and its components: the Interaction Model and the Protocols. We can use these models to create one model that represents the whole system. We can use this model to analyze the system.

The process of creating a composed model consists of the following steps:

1. Convert static structure of the Interaction Model into a *Container net.*
2. Translate the BPMN Choreography Diagrams of the protocols in the Interaction Model into *Protocol nets.*
3. Refine the Container and Protocol nets into the *system.*

This chapter is structured as follows. First, we provide the mathematical definition for Petri nets and Workflow nets. Next, we provide a model of the relation between the Interaction Model and Protocols. Then we introduce the strategy to combine the models. The next three sections each describe a step in the process of creating a composed model. The next section performs the described steps to our running example (of which the Interaction Model and Protocols are defined in Chapter 5. At the end of that section we have refined the Container and Protocol nets of our running example into the system. We conclude this chapter with some mentions of possible analysis that can be performed on composed models.

6.1 Petri nets

We use Petri nets to define the semantics of INORA. We have already shortly discussed Petri nets in Chapter 4 as a method to model single interactions. In INORA, we use Petri nets as building blocks to connect all Protocols in one composed model.

This section provides a mathematical foundation for Petri nets. Petri nets are named after Carl Adam Petri (Reisig, 2012), who laid the foundations of a model capturing local concurrency in his PhD thesis (Petri, 1962). They are a way to model and simulate the state of a system. It has both a graphical and a formal notation, allowing mathematical analysis.

Definition 6.1 (Petri nets)**.** A Petri net is a 3-tuple (P, T, F), where P is the set of *places* and T the set of *transitions*. Sets P and T are disjoint ($F \cap T = \emptyset$). F is the multiset of *flows*, mapping each place-transition and transition-place pair to a certain *weight*. F is therefore defined as $F : (P \times T) \cup (T \times P) \rightarrow \mathbb{N}$. We write $f \in F$, if $F(f) > 0$. $F((t,p))$ retrieves the weight of the flow from transition $t \in T$ to place $p \in P$. $F((p,t))$ retrieves the weight of the flow from place p to transition t.

Definition 6.2 (Marking, Marked Petri net, System)**.** Given a Petri net $N = (P, T, F)$, a *marking* $m : P \rightarrow \mathbb{N}$ assigns a number of tokens to each place in the Petri net. The pair (N, m) is called a *marked Petri net.* A *system* (N, m, Ω) is a Petri net N with an initial marking m and a set of final markings $\Omega \subseteq (P \rightarrow \mathbb{N})$.

Places are graphically represented by a bordered circle. Transitions are represented by a rectangle (often with a background color). Tokens in places are represented by filled circles, or by a number written in the place. Flows are represented by arrows between places and transitions or vice-versa.

An example of a formal representation of a Petri net $((P, T, F), m_0)$ is provided in Figure 22. The graphical representation is presented in Figure 23. The Petri net contains five places, five transitions, and ten flows.

Definition 6.3 (Pre- and postsets in Petri nets)**.** All transitions and places in a Petri net have a preset ($\bullet\, e$) and a postset ($e\, \bullet$). The preset of a transition $t \in \mathrm{T}$ is defined as $\bullet\, t = \{p \in \mathrm{P} \mid (p,t) \in F\}$. Similarly, $t\, \bullet = \{p \in \mathrm{P} \mid (t,p) \in F\}$. In our example (Figure 23) $\bullet\, A\ = \{p\}$ and $A\, \bullet = \{q\}$.

$P = \{p, q, r, s, t\}$
$T = \{A, B, C, D, E\}$
$F = [(p, A), (A, q), (q, B), (q, C), (B, r), (C, s), (r, D), (s, E), (D, t), (E, t))]$
$m_0 = [p]$

Figure 22: Mathematical representation of Petri net 1

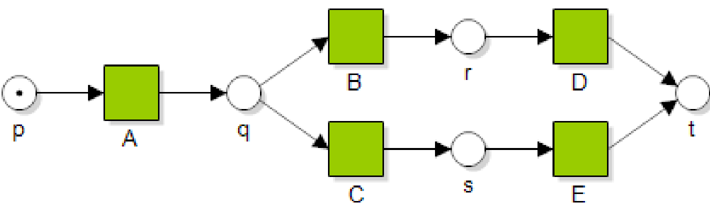


Figure 23: Graphical representation of Petri net 1

To be able to *fire* a transition, it needs to be enabled. This means that for all places in the preset of that transition ($\bullet\, t$), there needs to be at least as many tokens in that place as the weight of the flow between the place and the transition requires. Transition $t \in T$ is therefore enabled if $\forall p \in \bullet\, t : \mathrm{F}(\,(p, t)\,)| \leq \mathrm{m}(\,p\,)$. In our example, transition A is enabled, because place p has a token. There is no order in which transitions need to fire. Any enabled transition can be fired arbitrarily.

When a transition is fired, a new marking is produced (m'). The required tokens in the preset are consumed and the tokens are produced according to the weight of the flows between the place and the transitions in the postset i.e. $\forall p \in \mathrm{P} : \mathrm{m}'(\,p\,) = \mathrm{m}(\,p\,) - \mathrm{F}(\,p\,,\,t\,) + \mathrm{F}(\,t\,,\,p\,)$. The example Petri net has initial marking $[p]$. Only transition A is enabled, which produces the following marking after firing $m' : [q]$.

6.2 Workflow net

Workflow nets are normal Petri nets with some additional constraints. Figure 23 is an example of a Workflow net ($i = p$ and $f = t$, $\bullet\, p = t \bullet = \emptyset$)

Definition 6.4 (Workflow nets)**.** A workflow net W is a tuple (P, T, F, i, f), where (P, T, F) is a Petri net, $i \in P$ is the initial place, and $f \in P$ is the final place, such that $\bullet\, i = f \bullet = \emptyset$ and all nodes $(P \cup T)$ are on a path from i to f.

6.3 Analyze INORA

As described before, INORA aims to create an approach that allows for modeling complex interactions. It should, therefore allows for modeling both the static and the dynamic structure of a system. We also want to be able to automatically analyze the created models for validity. As said earlier in this chapter, we use Petri nets to define the semantics of INORA. These Petri nets can be used to analyze the system as a whole as well.

Figure 24 shows how we can convert the containers in the Interaction Model into Petri nets, which we call *Container nets.* Every container is translated into a Petri net that contains a transition for every independent function. More about translating the Interaction Model into the Container net will be explained in Section 6.4.

In Chapter 5 we have seen the Interaction Model with its references to Protocols and the BPMN Choreography Diagram that represent these Protocols. Figure 19 showed the relation between the Interaction Model and the BPMN Choreography Diagrams. These BPMN Choreography Diagrams can be translated into Petri nets as well. We will elaborate on how to do this in Section 6.5.

The Container net and Protocol nets can then be used to compose the system. The relation between these nets is displayed in Figure 25

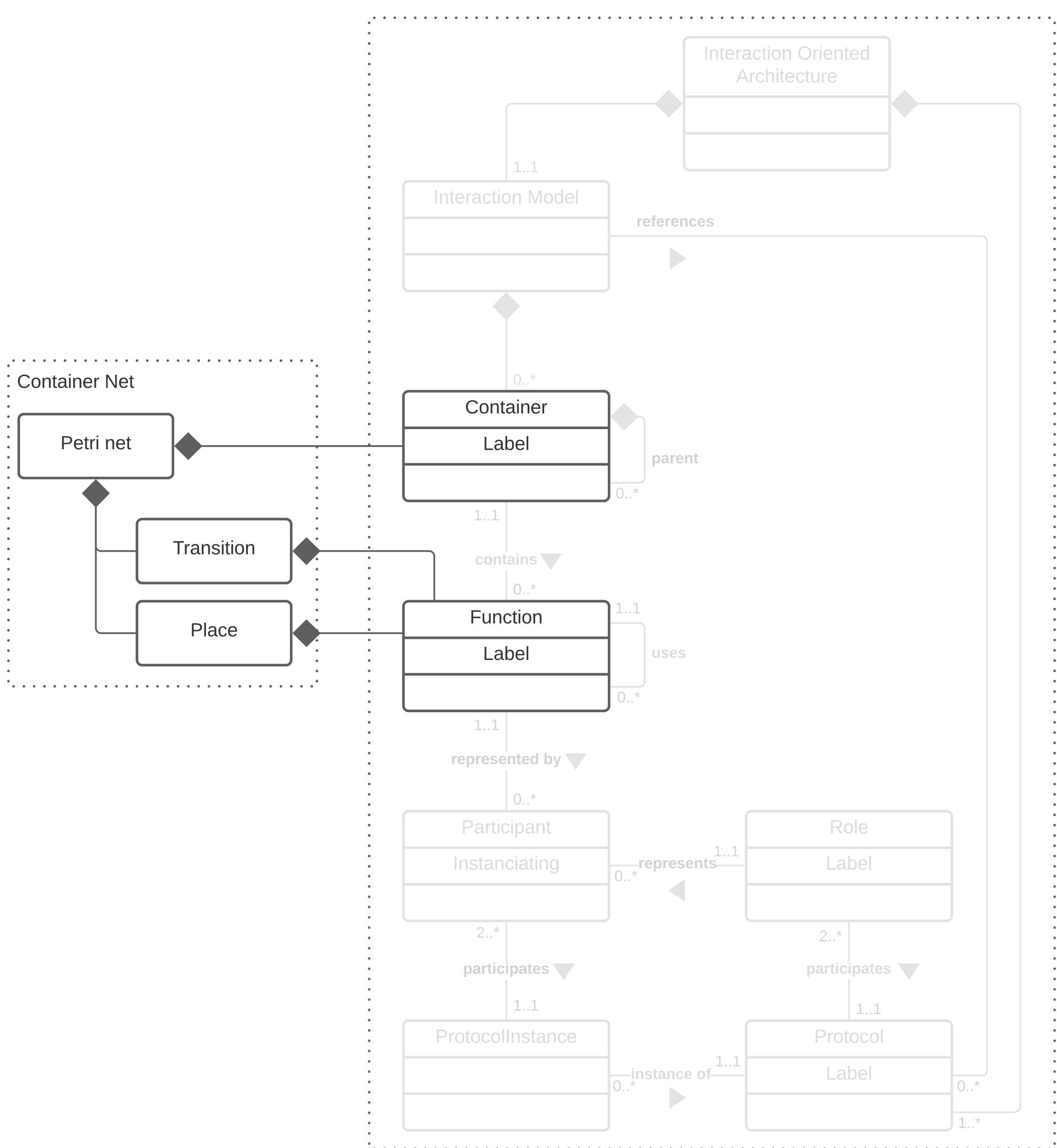


Figure 24: Relation between the Interaction Model and the Container net

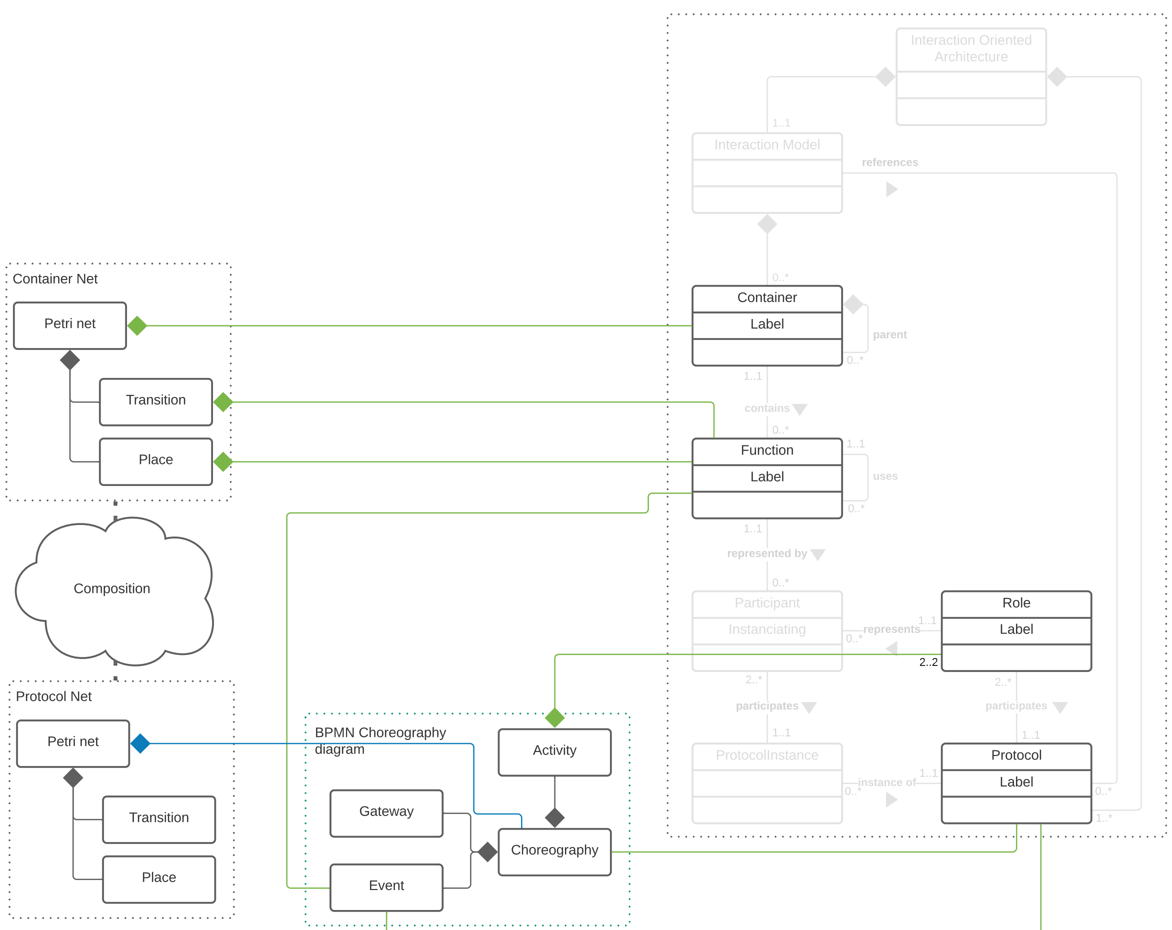


Figure 25: Relation between the Container net and Protocol nets

6.4 Converting the Interaction Model to the Container net

The containers and functions in the Interaction Model can be translated into a Petri net. We call these types of Petri nets, *Container nets*. Containers are translated to a single place p with a token ($m = \{p\}$). Every independent function in the Interaction Model is translated into a transition t with a self-loop to a single place per container (as can be seen in Figure 26) $\bullet t = t \bullet$. This translation results in a so-called *flower model*: there is one place per container to which all transitions are connected. If an independent function participates in multiple protocol instances, we duplicate the transition to allow refining those multiple participations (more later in this chapter). This duplication is called *emergency duplication* (Berthelot, 1978). In emergency duplication, we copy a transition with the same label.

The blue boxes in the Petri nets in this Chapter have no semantic meaning in Petri net analysis, we use them solely to illustrate the components in the Container net.

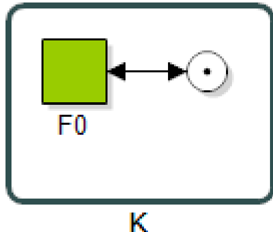


Figure 26: Translation of a container to a Container net resulting in a Flow Model

Example

In this box, we present two examples of translations from Interaction Models to Container nets. These Interaction Models are the same as the models presented in Figure 14 and Figure 16 in Chapter 5, but are repeated here for convenience.

First, we look at the example model in Figure 27 and the corresponding translation in Figure 28. As can be seen, every Container is translated into a place with a token and all the independent functions are translated into a transition. Function $F3$ is thus missing from this translation because it is not independent. The transition representing function $F1$ is duplicated because it participates in two protocol instances.

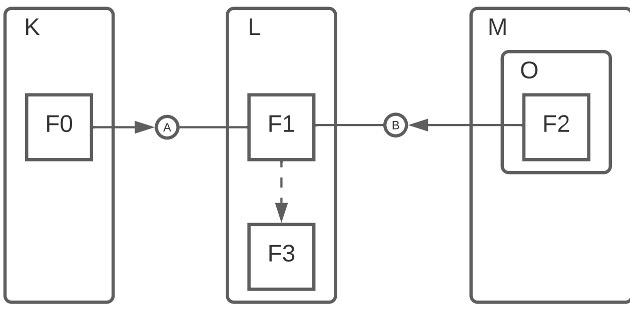


Figure 27: Example Interaction Model 1

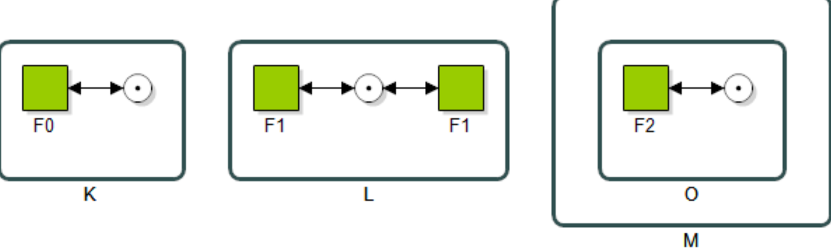


Figure 28: Example of the Container net belonging to the Interaction Model presented in Figure 27

Next, we look at the example model in Figure 29 and the corresponding translation in Figure 30. As can be seen, once again, every Container is translated into a place with a token and all the independent functions are translated into a transition. Function $F3$ is again missing from this translation because it is not independent.

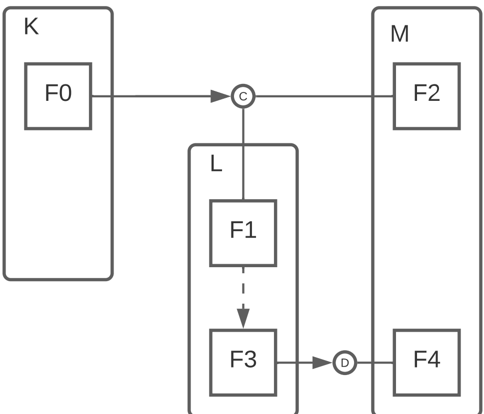


Figure 29: Example Interaction Model 2

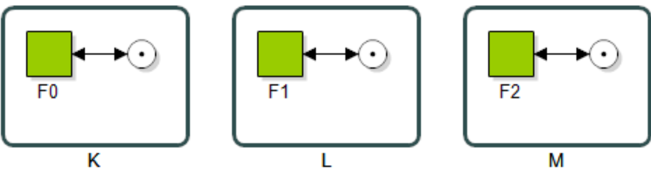


Figure 30: Example of the Container net belonging to the Interaction Model presented in Figure 29

6.5 Converting the Protocols to Protocol nets

The BPMN Choreographies that are used to define the Protocols in the Interaction Model have to be converted to a Petri net as well. We call these types of models Protocol nets. To correctly translate a Choreography to a Protocol net, the following steps have to be performed:

1. Translate the BPMN Choreography Diagram into a Petri net, resulting in a Participant net.
2. Copy the resulting Petri net for every participant, resulting in a Copied Participant net.
3. Create a place for every message en connect it to the participant's Petri nets, resulting in a Protocol net
4. Refine the transitions that represent a dependent function or a protocol, resulting in a Refined Protocol net for that transition.
5. Reduce all silent transitions.

Example

During the explanation, we will use the simple Interaction Model as presented in Figure 31, with Protocol A (Figure 32) and Protocol B (Figure 33).

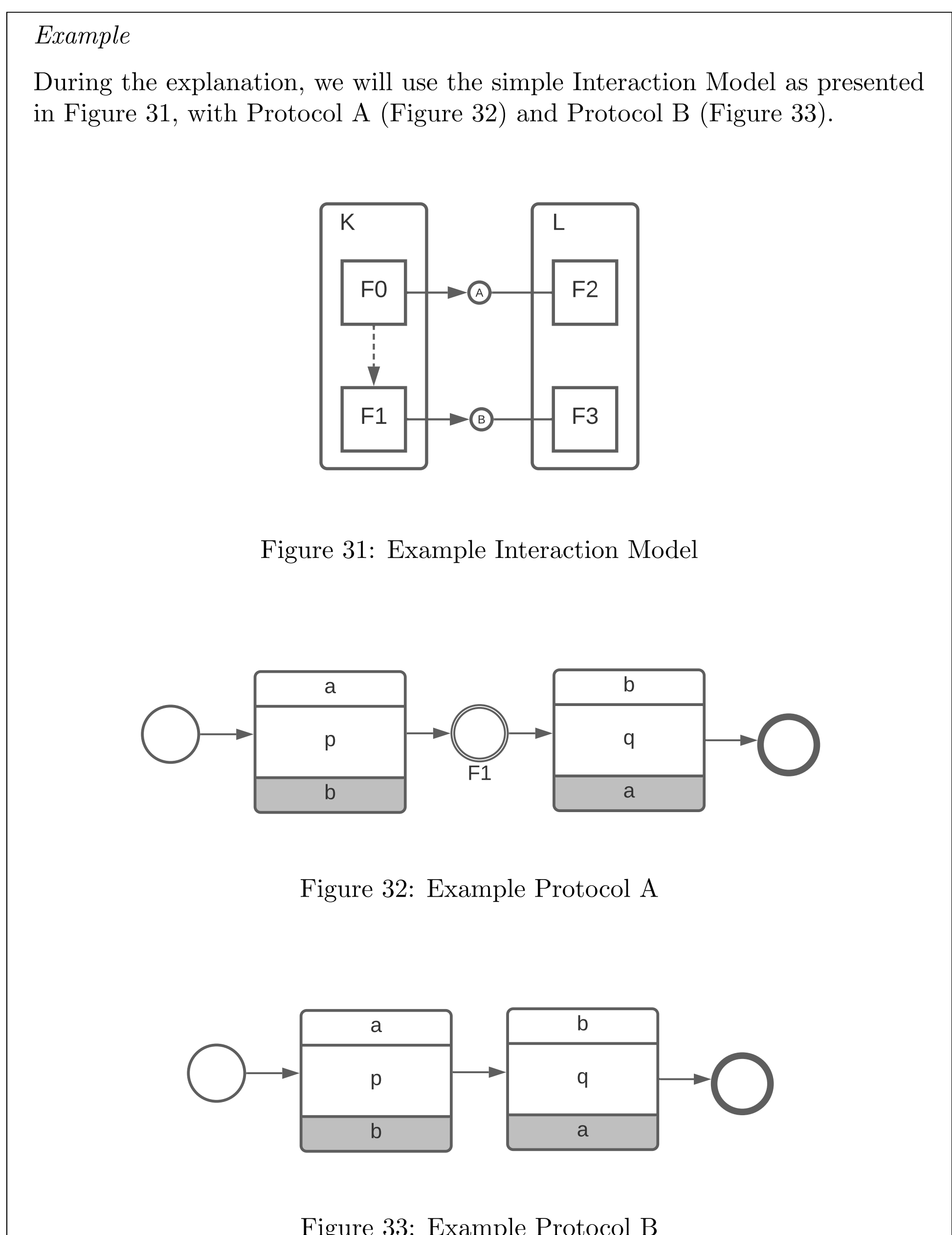


Figure 31: Example Interaction Model

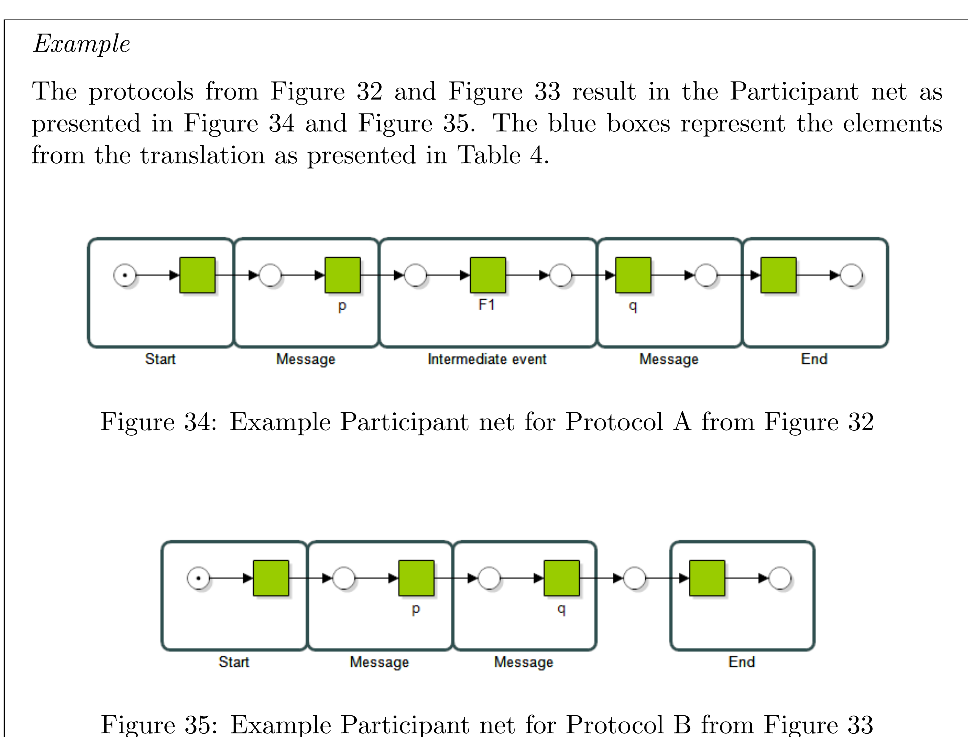
Figure 32: Example Protocol A

Figure 33: Example Protocol B

Step 1: Translate the BPMN Choreography Diagram into a Petri net.

In literature, some translation methods for translating BPMNs to Petri nets are proposed. One example is the mapping proposed by (Raedts et al., 2007). We have adapted the method to support BPMN Choreography Diagrams by changing the standard BPMN Tasks to BPMN Choreography Messages. Additionally, for our needs, the mapping also needs to support intermediate events. We have added a mapping for those: a simple transition that consumes a token from one place and emits one token in another place. Furthermore, the mapping of the XOR gateway as proposed by (Raedts et al., 2007) does not suffice our needs, as our analysis methods do not support the (non-standard) Petri net XOR gateway. We have therefore modified the translation for XOR gateways. Every transition representing the incoming tasks in the XOR Gateway is connected to every transition representing the outgoing tasks. The adapted and extended mapping is presented in Table 4. Only the elements within the blue boxes belong to the translations - the other elements are there for providing context to the translation.

The mapping of the BPMN should always result in a Workflow net: a Petri net with only one token in the initial marking and the final marking. Furthermore, the place-transition combination representing the events should be safe. If the mapping of the BPMN Choreography does not result in such a Workflow net, places should be added to result in an initial and final marking that does satisfy this need.

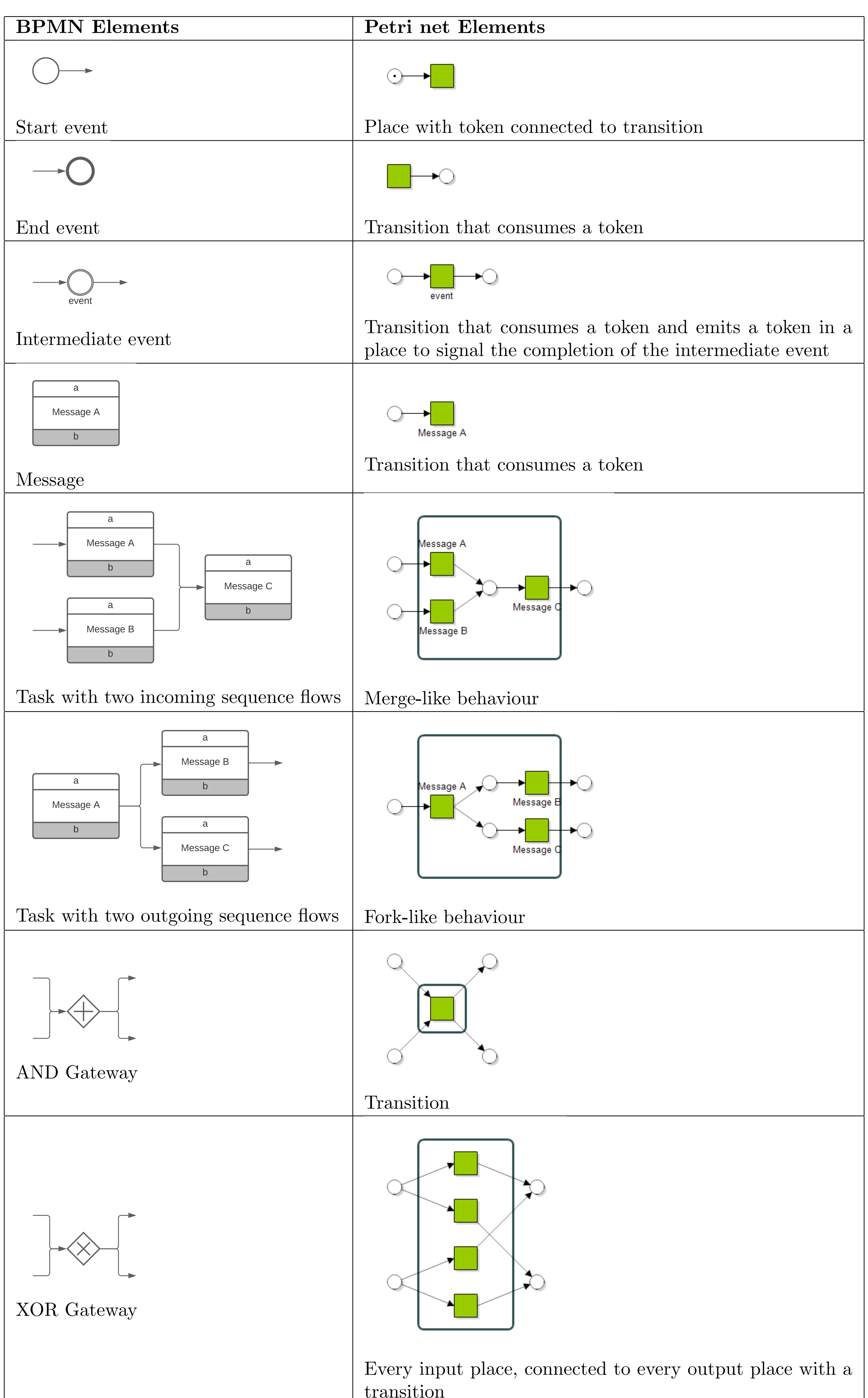


Example

The protocols from Figure 32 and Figure 33 result in the Participant net as presented in Figure 34 and Figure 35. The blue boxes represent the elements from the translation as presented in Table 4.

Figure 34: Example Participant net for Protocol A from Figure 32

Figure 35: Example Participant net for Protocol B from Figure 33



BPMN Elements	Petri net Elements
Start event	Place with token connected to transition
End event	Transition that consumes a token
Intermediate event	Transition that consumes a token and emits a token in a place to signal the completion of the intermediate event
Message	Transition that consumes a token
Task with two incoming sequence flows	Merge-like behaviour
Task with two outgoing sequence flows	Fork-like behaviour
AND Gateway	Transition
XOR Gateway	Every input place, connected to every output place with a transition

Table 4: Mapping of BPMN elements to Petri nets, modified from Raedts et al. (2007)

Step 2: Copy the resulting Petri net for every participant.

Every Participant net now has to be copied for every participant. In the case that a protocol has three participants, for example, the Participant net has to be copied three times.

Example

The Participant nets from Figure 34 and Figure 35 are copied as presented in Figure 36 and Figure 37. The blue boxes in the figures represent the participants in the protocols.

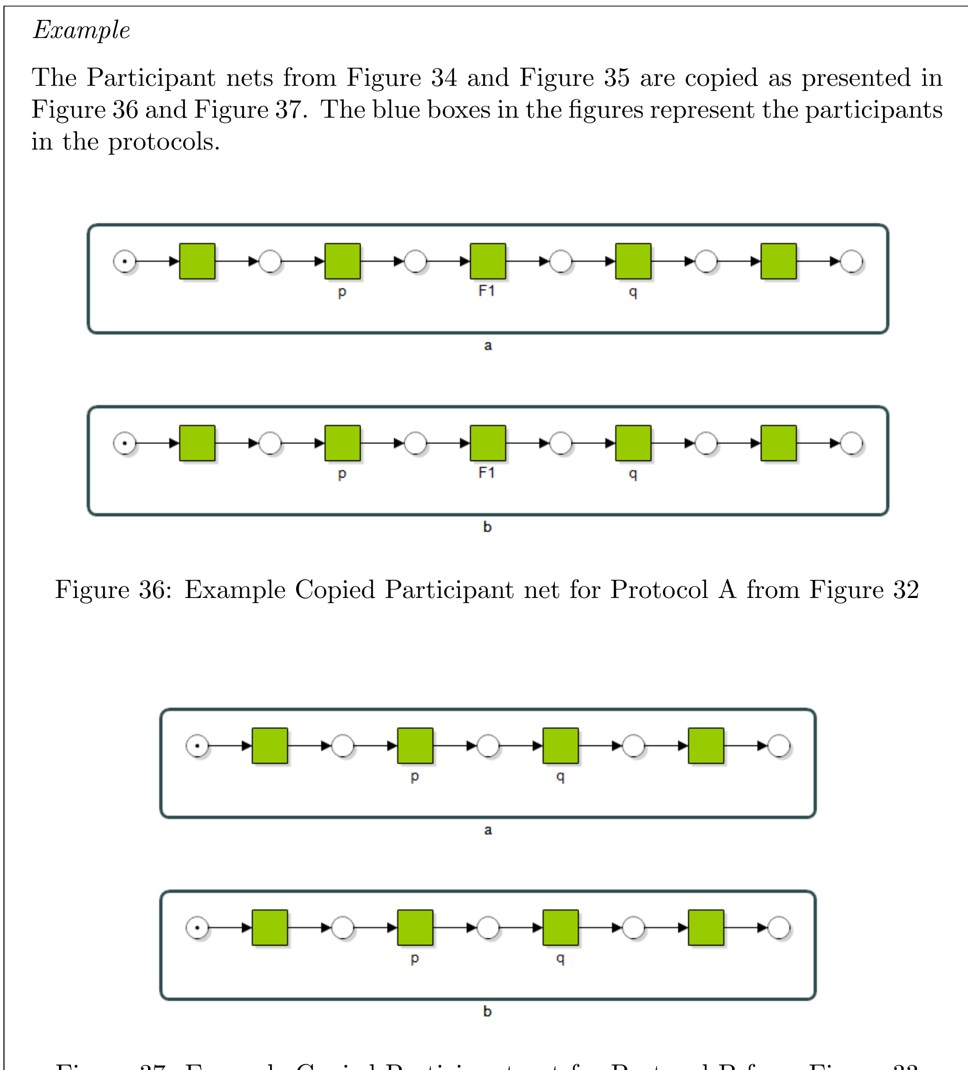


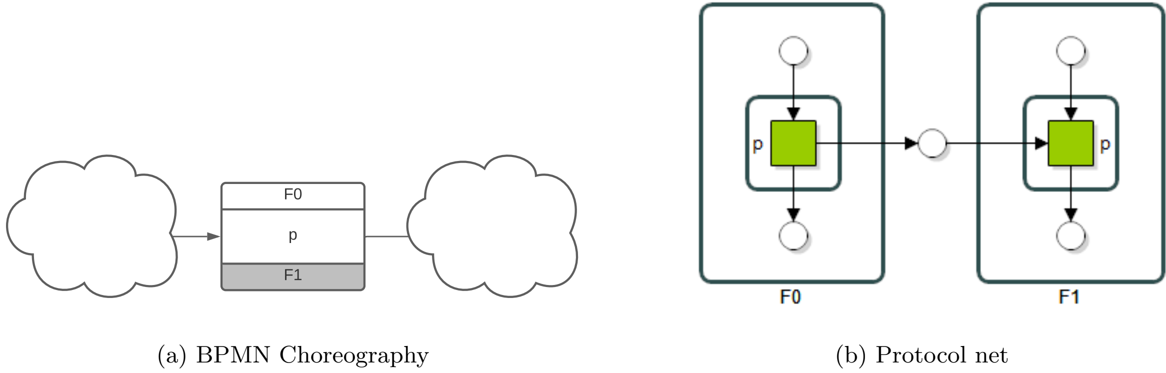
Figure 36: Example Copied Participant net for Protocol A from Figure 32

Figure 37: Example Copied Participant net for Protocol B from Figure 33

Step 3: Create a place for every message and connect it to the participant's Petri nets

A place needs to be created for every message in the BPMN Choreography diagram. This place then has to be connected to the corresponding transition in the Copied Participant net: the transition belonging to the sender of the message is the incoming flow of the place and the transition belonging to the receiver of the message is the outgoing flow of the place. An example of this translation can be seen in Figure 38a and Figure 38b.

Now that the Copied Participant nets have been connected by the message places, we have a Multi-Workflow net: the Protocol net.



(a) BPMN Choreography

(b) Protocol net

Figure 38: Example of the connection of the message-place to the corresponding transitions

Example

The Copied-Participant nets from Figure 36 and Figure 37 are enriched with the messages as presented in Figure 39 and Figure 40.

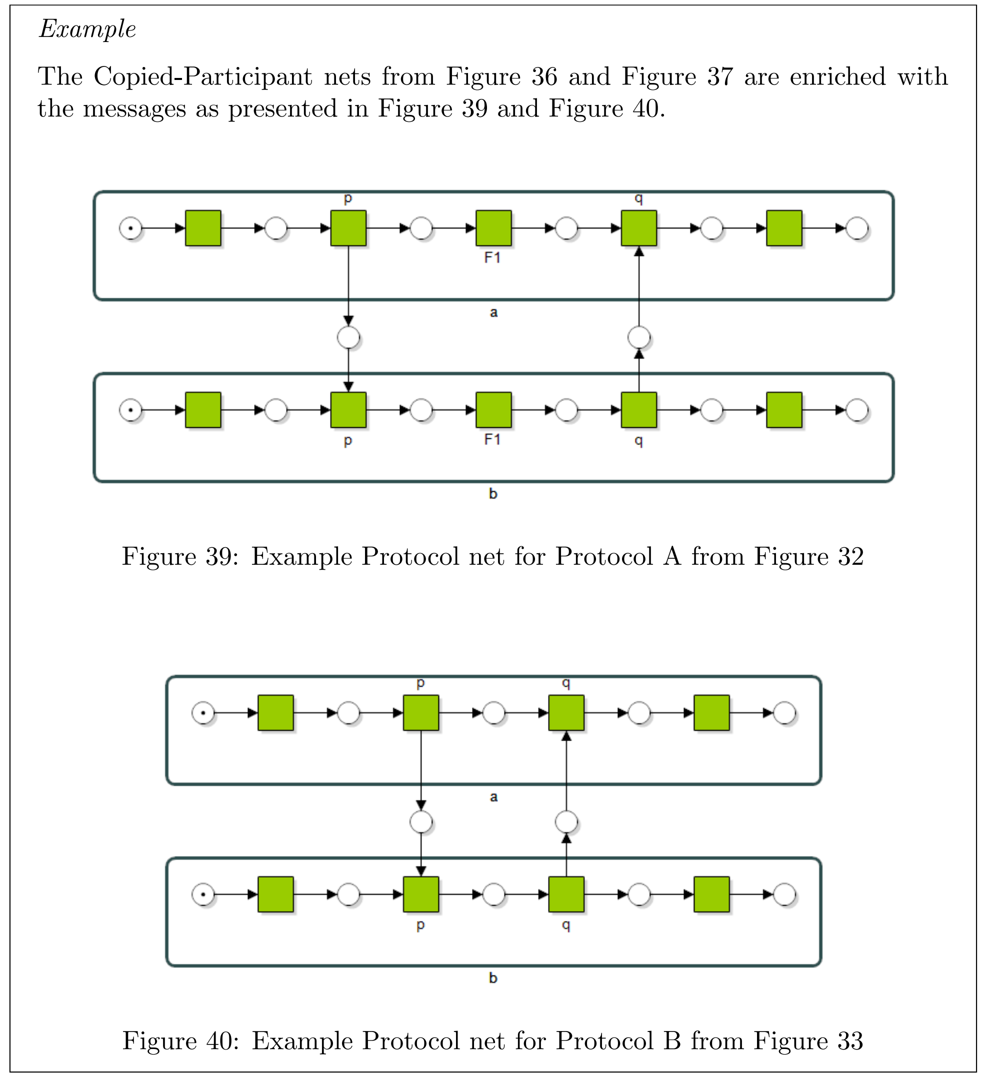


Figure 39: Example Protocol net for Protocol A from Figure 32

Figure 40: Example Protocol net for Protocol B from Figure 33

Step 4: Refine the transitions that represent a dependent function or a protocol.

Every dependent function and protocol is represented in the Choreography by an intermediate event. In the translation performed in step 1, these events are translated into transitions. In step 2, we have copied the Participant net for every participant. In the Participant net of the participant that holds the dependent function or protocol, the transition representing that function or protocol has to be refined with the composed Container net (so after performing all the steps presented in this chapter). In the other participants, the label of the transition representing the intermediate event can be removed (so it

will be reduced in the next step). The resulting net is a Refined net for Protocol x with x and x.

Example

Protocol B does not contain any intermediate events. So, the model from step 3 (Figure 40) does not have to be refined any further.

Protocol A, on the other hand, does contain an intermediate event. In our Interaction Model, presented in Figure 31, we can see that Function F0 uses Function F1. In Protocol A, F0 is Participant a (because F0 initiates the choreography) and F2 Participant b. Because F1 is an intermediate event in Protocol A, we need to refine it at the side of Participant a (F0) (and remove the label of the intermediate event on the side of Participant b (F2)).

For the refinement of F1, we need the fully composed model from F1. We can obtain this model by following all steps in this chapter. Figure 41 represents the composed model for Protocol B.

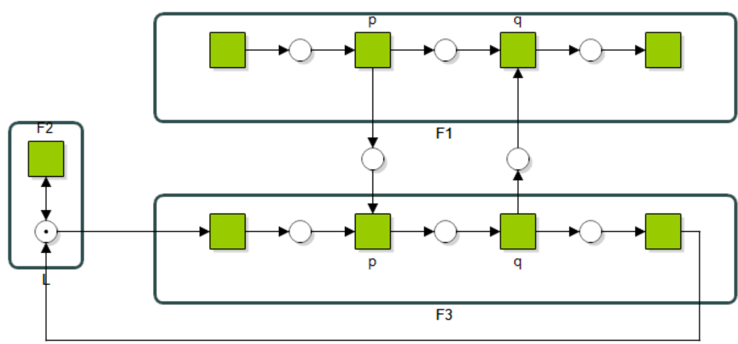


Figure 41: Composed net for Protocol B with F1 and F3

We can use this model to refine F1 in Protocol A. First of all, we remove the label of the F1 transition on the side of b (surrounded by a blue box for clarification). The F1-transition on the side of a is refined by connected the completely composed model of F1, as presented in Figure 42.

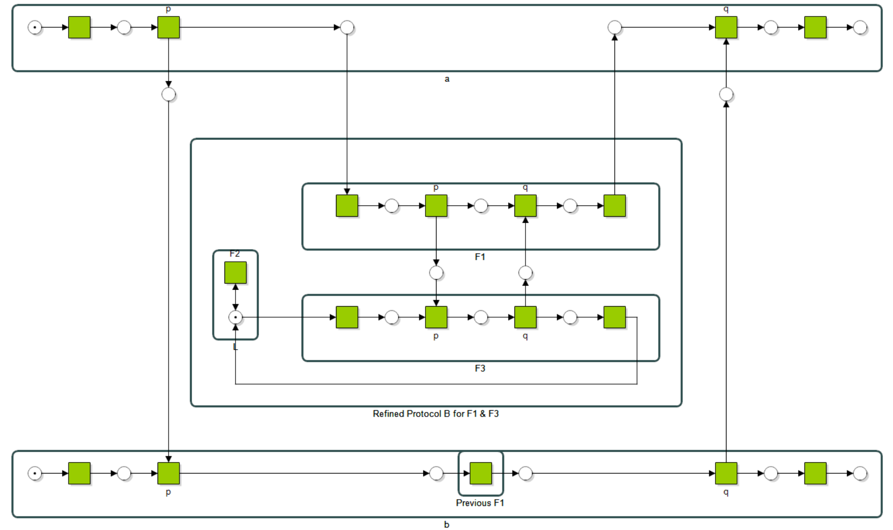
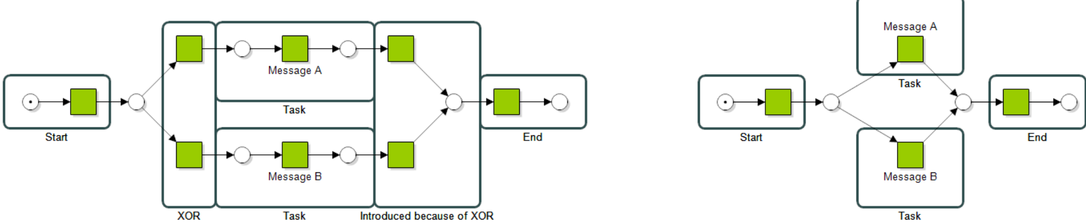


Figure 42: Refined Protocol net for Protocol A with a and b

Step 5: Reduce all silent transitions.

When refining the transitions that represent a dependent function or a protocol in the previous step, silent transitions remain (Murata, 1989). Furthermore, during the translation of the BPMN Choreography to Petri nets, silent transitions can be also introduced with for example XOR translations. Silent transitions are non-labeled transitions (sometimes labeled with ϵ). Those transitions have no semantic function in the Petri net. An example of the reduction of silent transitions is shown in Figure 43.



(a) Example of Petri net translation with silent transitions

(b) Example of Petri net where silent transitions are reduced

Figure 43: Example of the reduction of silent transitions in Petri nets

Example

With the refinement presented in Figure 42, the F1 transition on the side of b has become silent and must be removed (as presented in Figure 44).

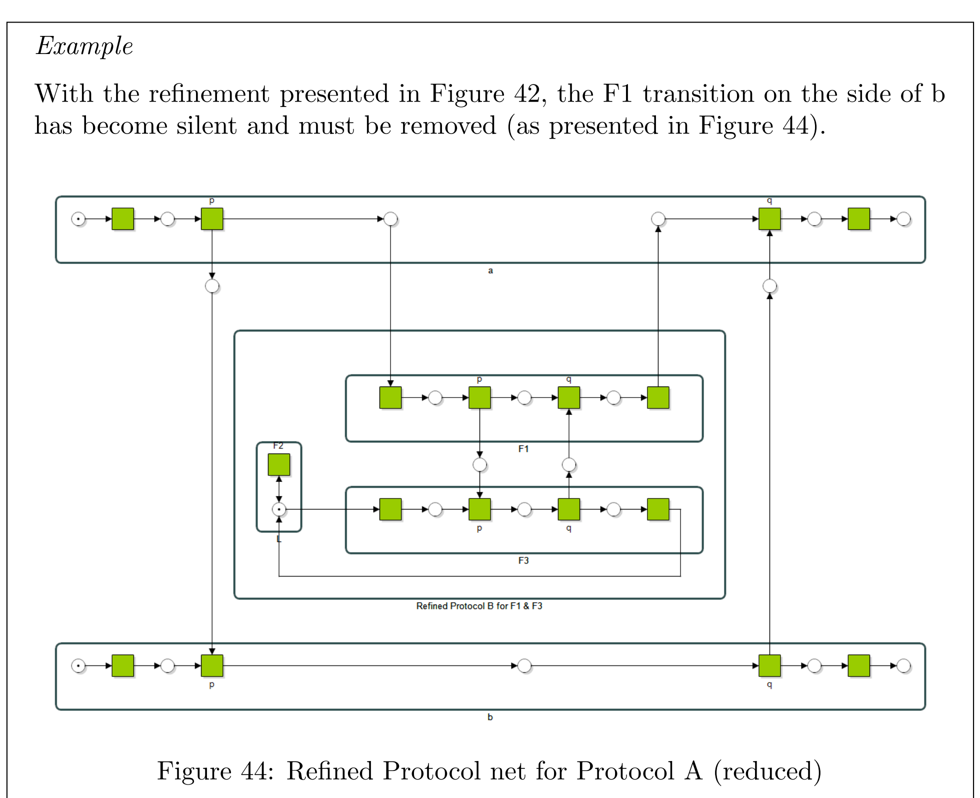


Figure 44: Refined Protocol net for Protocol A (reduced)

6.6 Compose the system

Now that we have both the Container net and the Protocol nets, we can compose the whole system. First, we will define the composition. Next, we will show this composition using the example from the previous section.

K. M. van Hee et al. (2011) propose a method for refining places in a Petri net. This does not apply to our problem, because we need to refine transitions. We use transitions because the function that the transition refers to is an action (and involves a whole protocol in our case). Conceptually, a transition is a better representation for such an action than a place. The use of the transition is not just conceptual though, when we would use the traditional approach to replace the transition with a transition - place - transition, that would allow us to use place refinement, the moment of choice is not correct anymore. You would first have to execute a protocol with a non-deterministic choice by firing the first transition in the transition - place - transition replacement. This is not correct, because the protocol should make the choice - deterministic. The only way that we can achieve this using Petri nets is by refining the transition itself with the protocol. We have used the method for place refining by K. M. van Hee et al. (2011) as an inspiration for our method for refining transitions in a Petri net.

The general idea is that we have two nets, which we want to combine into one net. If we look at Figure 45, we can identify a net N with a transition t. Net M defines the refinement for transition t. If we want to refine net N, we have to connect outgoing flows from p to all transitions that are connected to i in net M. We do the same for all incoming flows to q: we connect an incoming flow from every transition with the incoming flow to f. Lastly we remove i and f. The resulting net is presented in Figure 46. The definition for this refinement is given in Definition 6.5.

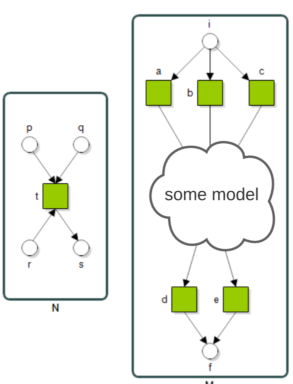


Figure 45: Net N and M

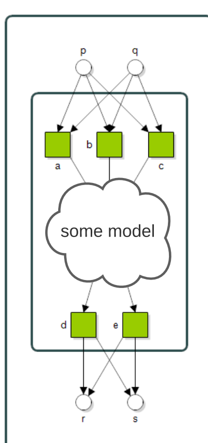


Figure 46: Refined net N

Definition 6.5 (Refinement of a single transition). Let $\mathcal{N} = (N, m_N^0, \Omega_N)$ be a system with $N = (P_N, T_N, F_N)$ and $R \subseteq P_N$ be a set of places to be refined and $M = (P_M, T_M, F_M)$ be another Petri net, such that N and M are disjoint. $N \odot M$ is a system $((P, T, F), m^0, \Omega)$ where:

- $P = P_N \cup (P_M \backslash \{i, f\})$
- $T = (T_N \backslash \{t\}) \cup T_M$
- $F = F_N \backslash ((\bullet\, t \times \{t\}) \cup (\{t\} \times t\,\bullet)$
  $\cup\, F_M \backslash ((\{i\} \times i\,\bullet) \cup (\bullet f \times \{f\}))$
  $\cup\, (\bullet\, t \times i\,\bullet) \cup (\bullet f \times t\,\bullet)$
- $m^0 = m_N^0$
- $\Omega = \Omega_N$

Our Protocol nets, however, are Multi-workflow nets. They have connected i/o-pairs (as illustrated in Figure 47). K. M. van Hee et al. (2011) define a multi-workflow net as in Definition 6.6.

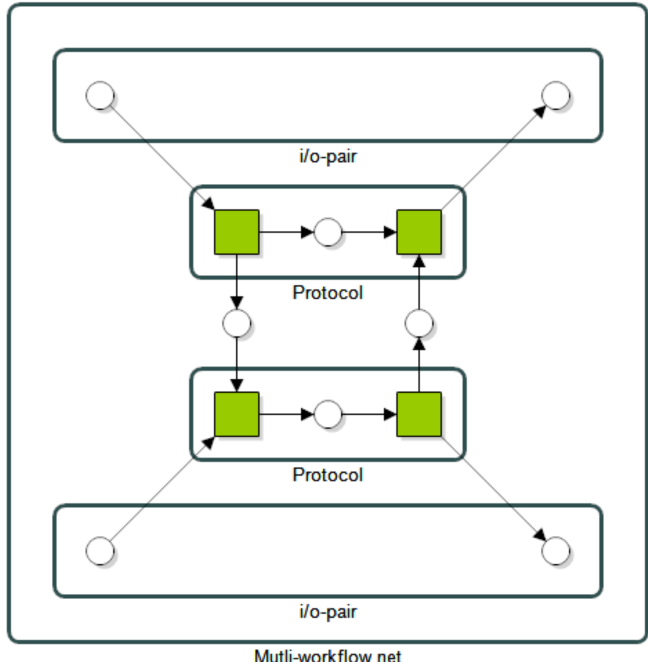


Figure 47: Illustration of Input / Output pairs

Definition 6.6 (Multi-workflow net (K. M. van Hee et al., 2011))**.** On a Cartesian product we define two projection functions $\pi_1 : \mathrm{S} \times \mathrm{T} \rightarrow \mathrm{S}$ and $\pi_2 : \mathrm{S} \times \mathrm{T} \rightarrow \mathrm{T}$ such that $\pi_1((s,t)) = s$ and $\pi_2((s,t)) = t$ for all $(s,t) \in \mathrm{S} \times \mathrm{T}$. We lift the projection function to sets in the standard way, i.e. $\pi_{\mathrm{i}}(U) = \{\pi_{\mathrm{i}}((s,t)) | (s,t) \in \mathrm{U}\}$ for $\mathrm{U} \subseteq \mathrm{A} \times \mathrm{B}$ and $i \in \{1,2\}$

A multi-workflow net (MWF net) N is a 4-tuple (P,T,F,E) where (P,T,F) is a Petri net and $E \subseteq P \times P$ is a set of i/o pairs, such that $|E| = |\pi_1(E)| = |\pi_2(E)|$ and $\bullet\pi_1(E) = \bullet\pi_2(E) = \emptyset$. The places in $\pi_1(E)$ are called the input places of N, the places in $\pi_2(E)$ are called the output places of N. Furthermore, each node $n \in P \cap T$ is on a path from an input place to an output place.

When refining the transitions in a Multi-workflow net, we want to keep the i/o-pairs connected to the correct net. The definition in Definition 6.5, is therefore not sufficient. In Definition 6.7, we define the refinement of a set of places in a Multi-workflow net. The general idea is still to combine two nets into one Petri net. In this case, one of these nets is a Multi-workflow net. This is displayed in Figure 48. The transition t and u in net N are refined by the MWF M, with the i/o-pairs $(a1, b1)$ and $(a2, b2)$. All outgoing flows of p are connected to all transitions that are connected to $i1$ in net M. All outgoing flows of q are connected to all transitions that are connected to $i2$ in net M. We do the same for the incoming flows to q and r: we connect an incoming flow from every transition that is connected to q with the incoming flow to $f1$ and an incoming flow from every transition that is connected to r with the incoming flow to $f2$. The resulting net is presented in Figure 49.

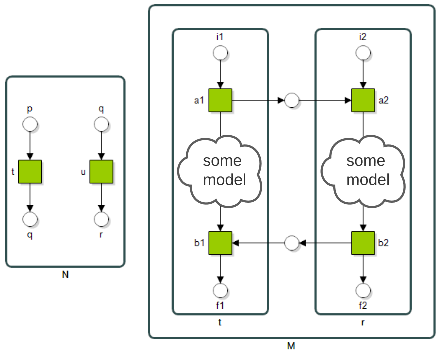


Figure 48: Net N and M

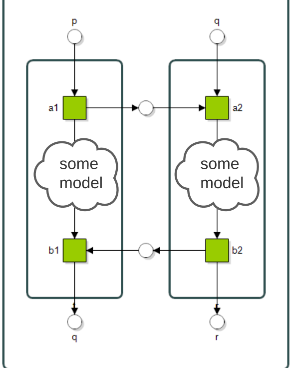


Figure 49: Refined net N

Definition 6.7 (Refinement of a set of transitions)**.** Let $\mathcal{N} = (N, m_N^0, \Omega_N)$ be a system with $N = (P_N, T_N, F_N)$ and $R \subseteq T_N$ be a set of transitions to be refined and $M = (P_M, T_M, F_M, E_M)$ be a MWF net, such that N and M are disjoint. Let $\alpha : R \to E_M$ be a total, bijective function. The refinement $N \bigodot_\alpha M$ is a system $((P, T, F), m^0, \Omega)$ where:

- $P = P_N \cup (P_M \backslash (\pi_1(E_M) \cup \pi_2(E_M)))$
- $T = (T_N \backslash R) \cup T_M$
- $F = F_N \backslash ((\bullet\, t \times R) \cup (R \times t\, \bullet)$
  $\cup\ F_M \backslash (\bigcup_{r \in R} ((\pi_1(\alpha(r)) \times \pi_1(\alpha(r))\, \bullet) \cup (\bullet\, \pi_2(\alpha(r)) \times \pi_2(\alpha(r)))))$
  $\cup\ \bigcup_{r \in R} ((\bullet\, r \times \pi_1(\alpha(r))\, \bullet) \cup (\bullet\, \pi_2(\alpha(r)) \times r\, \bullet))$
- $m^0 = m_N^0$
- $\Omega = \Omega_N$

Example

We can compose the net for Protocol B, from the previous section (Figure 33) to retrieve the resulting Composed net as also displayed in Figure 50.

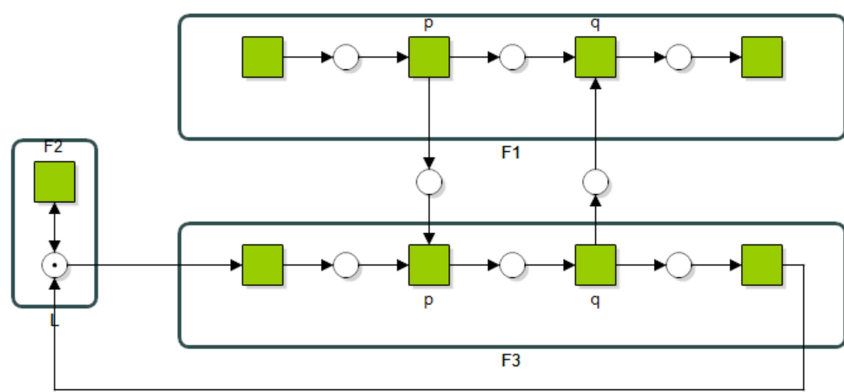


Figure 50: Composed net for for Protocol B, same as Figure 41

6.7 Running example

In this section, we present the composition of our running Example into the system. In the first subsection, we convert the static structure into a Container net. In the next subsection, we convert the Protocols to Protocol nets and finally, we compose the system.

6.7.1 Convert static structure into Container net

For our running example (Interaction Model defined in Chapter 5, Figure 20), we can create the Container net. The Container net for our example is presented in Figure 51. Every container is converted into a blue box (just for visualization) with a single place with a token. Every independent function is converted into a transition that is connected to the place. In our example, only the functions *System Authenticate* and *Bank Authenticate* are dependent and therefore not included in the Container net. We do not need to duplicate any transitions, because non of the functions participate in more than one protocol instance.

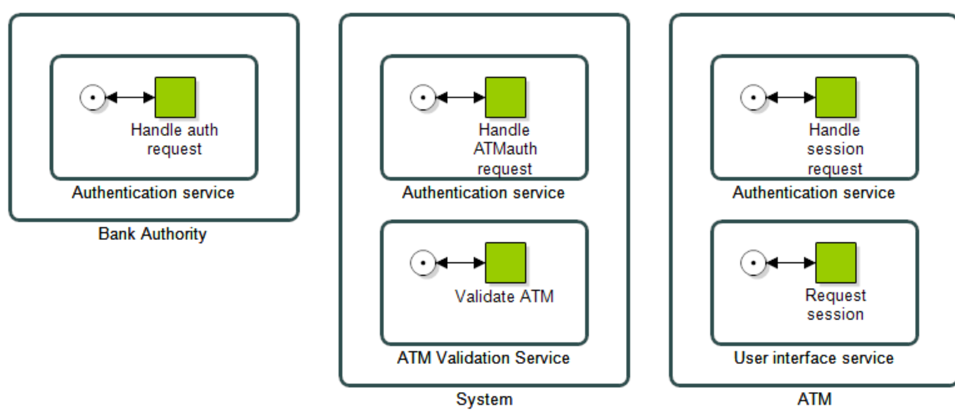


Figure 51: The Container net belonging to our running example

6.7.2 Convert Protocols to Protocol nets

We can convert the Protocols belonging to our running example to Protocol nets by performing the steps presented in this chapter.

Step 1: Translate the BPMN Choreography Diagram into a Petri net.

The translations for the protocols, presented in Figure 21, are provided in Figure 52.

Step 2: Copy the resulting Petri net for every participant.

The Copied Participant nets are presented in Figure 53.

Step 3: Create a place for every message and connect it to the participant's Petri nets

The Protocol nets are presented in Figure 54.

Step 4: Refine the transitions that represent a dependent function or a protocol.

Next, we can refine all transitions that represent a dependent function or a protocol. Only Protocol B has such a transition, so we only have to refine that protocol (Figure 57). To do so, we first have to refine the protocols and independent functions that are present in the protocol. The refined Protocol C for *Handle ATMauth request* and *Validate ATM* is presented in Figure 55. The refined net for *Bank Authenticate* and *Handle auth request* is presented in Figure 56.

Step 5: Reduce all silent transitions.

Figure 58 presents the reduced model for Figure 57.

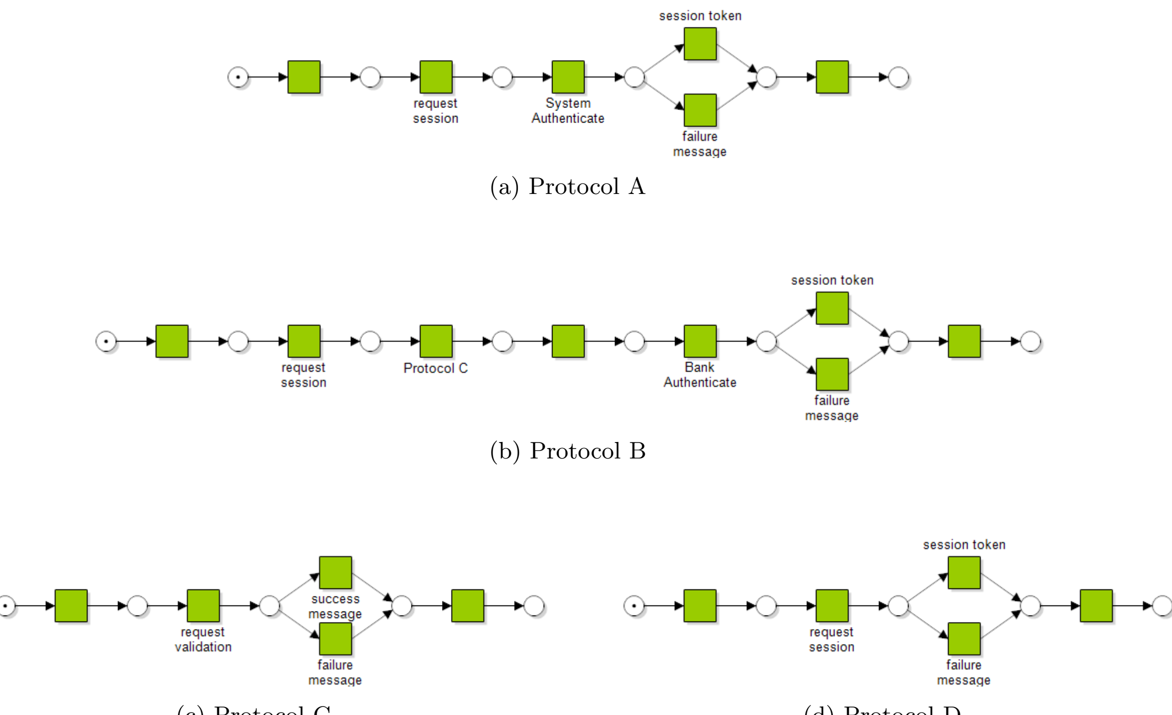


Figure 52: Participant nets from the Protocols belonging to the running example, defined in Figure 21

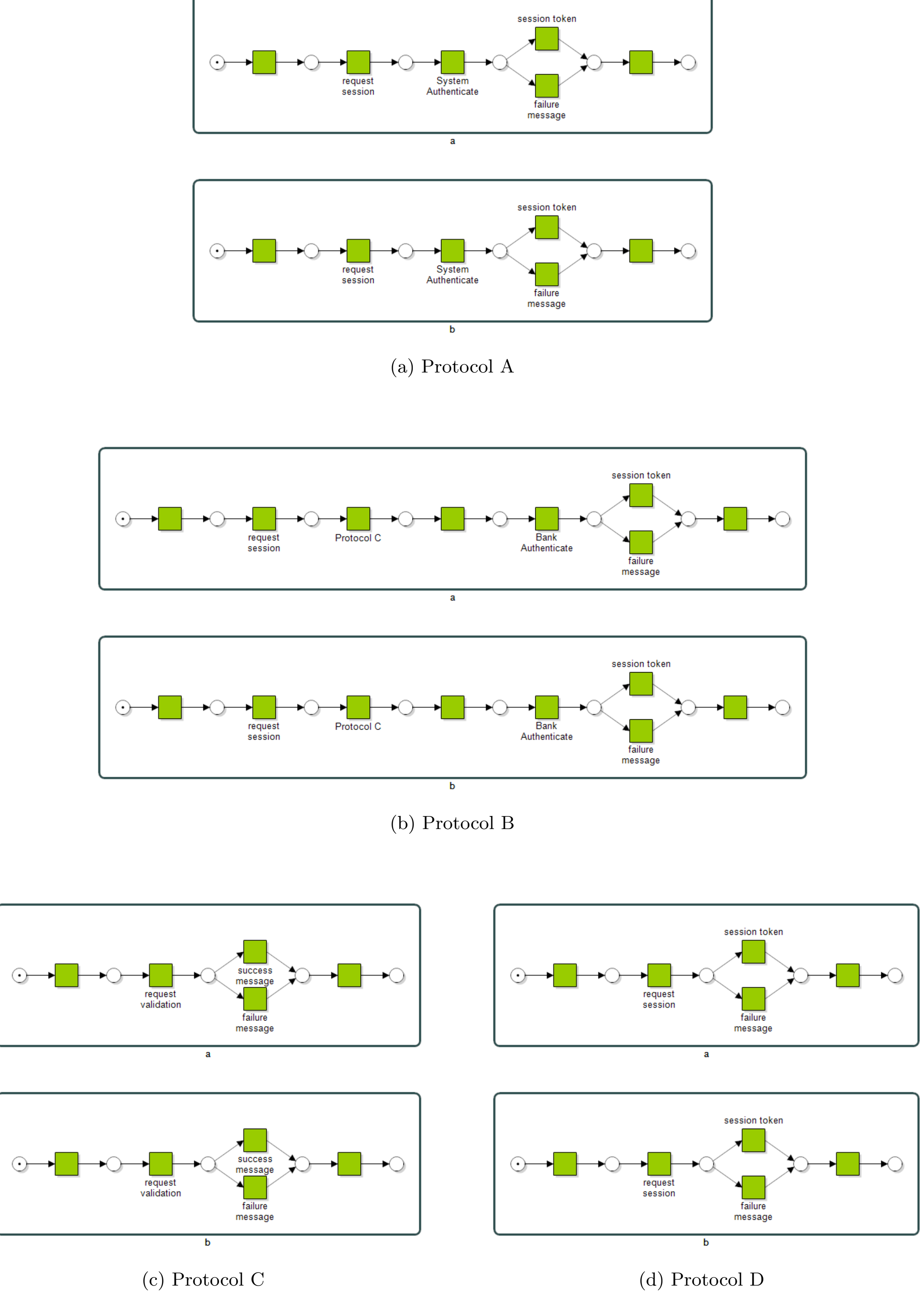
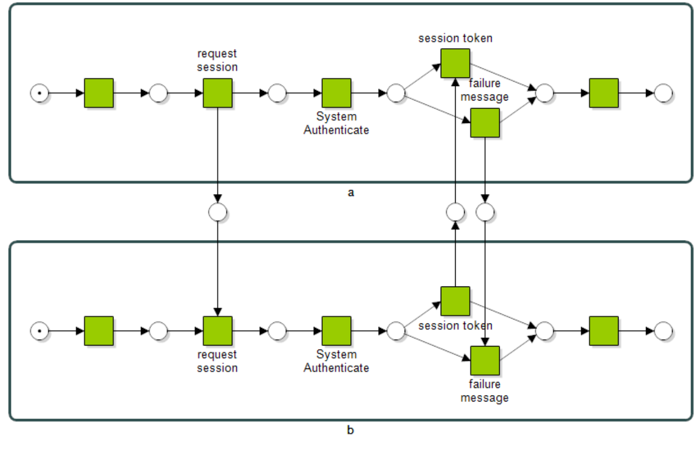
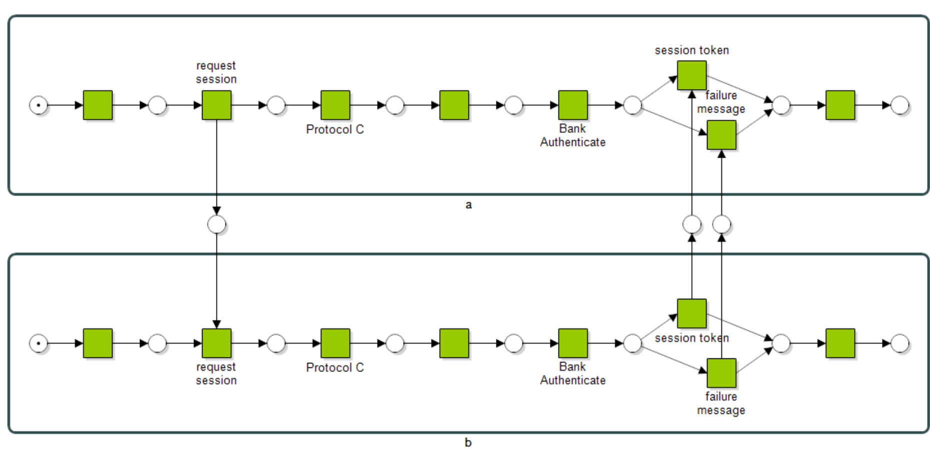


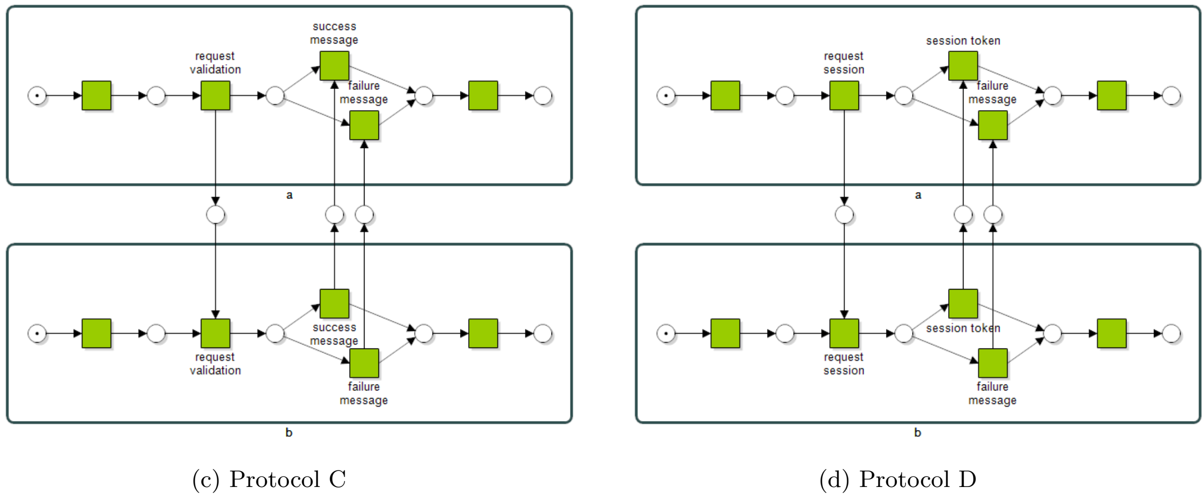
Figure 53: Copied Participant nets belonging to the ATM running example and defined in Figure 21



(a) Protocol A



(b) Protocol B



(c) Protocol C (d) Protocol D

Figure 54: Protocol nets belonging to the ATM running example and defined in Figure 21

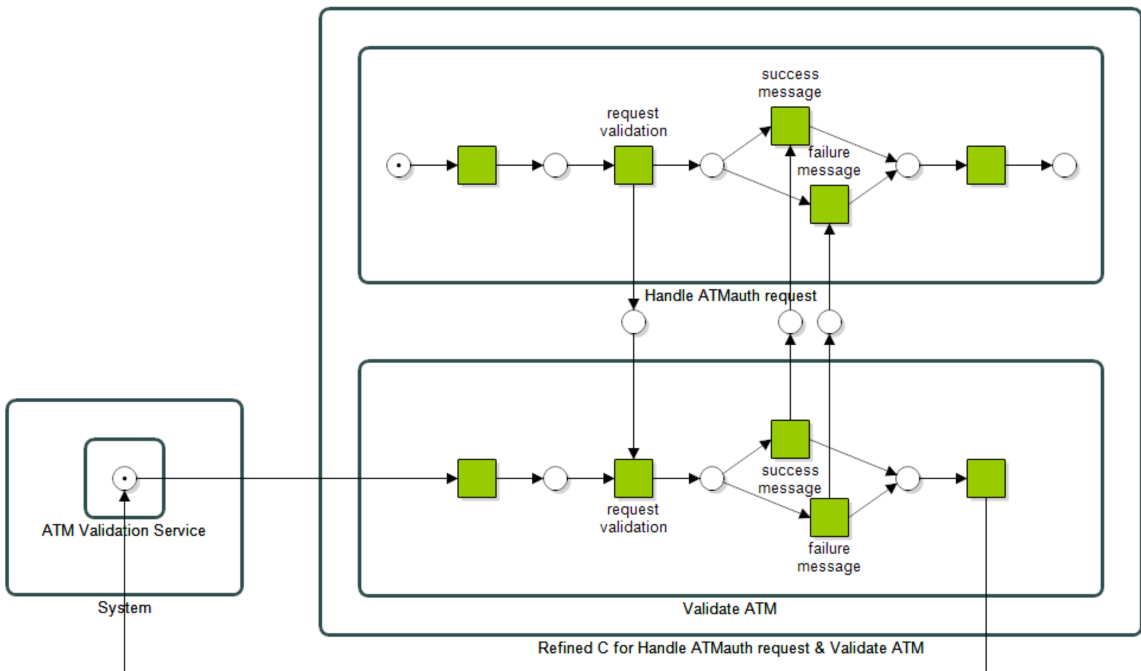


Figure 55: Protocol C refined for Handle ATMauth request & Validate ATM

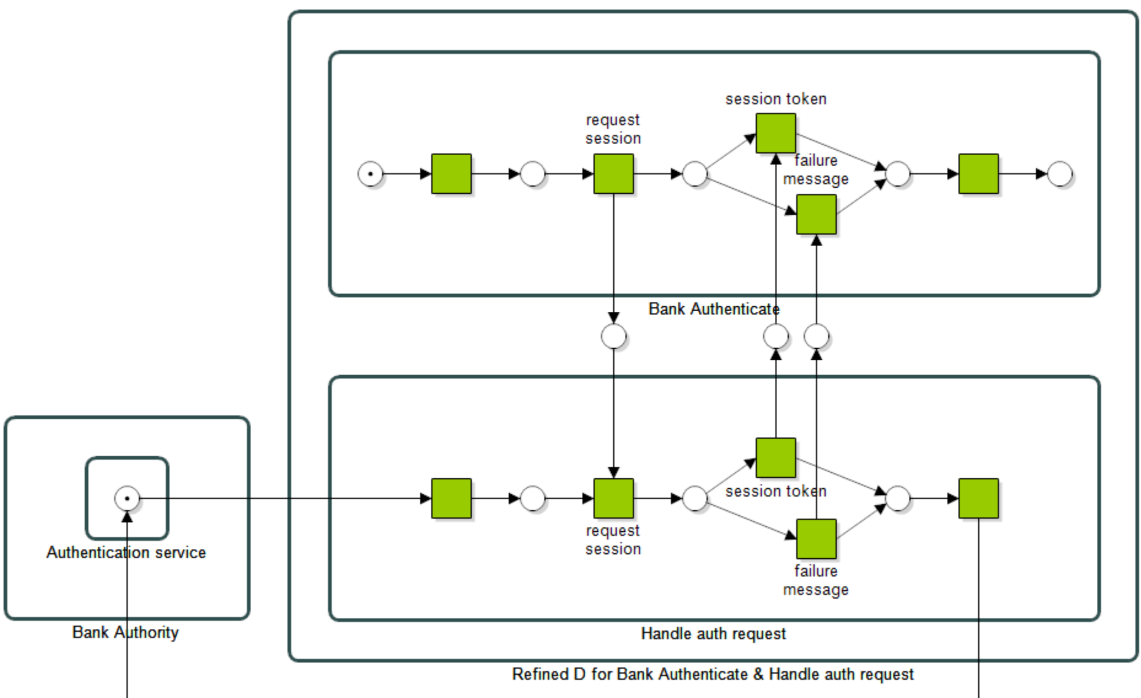


Figure 56: Protocol D refined for Bank Authenticate & Handle auth request

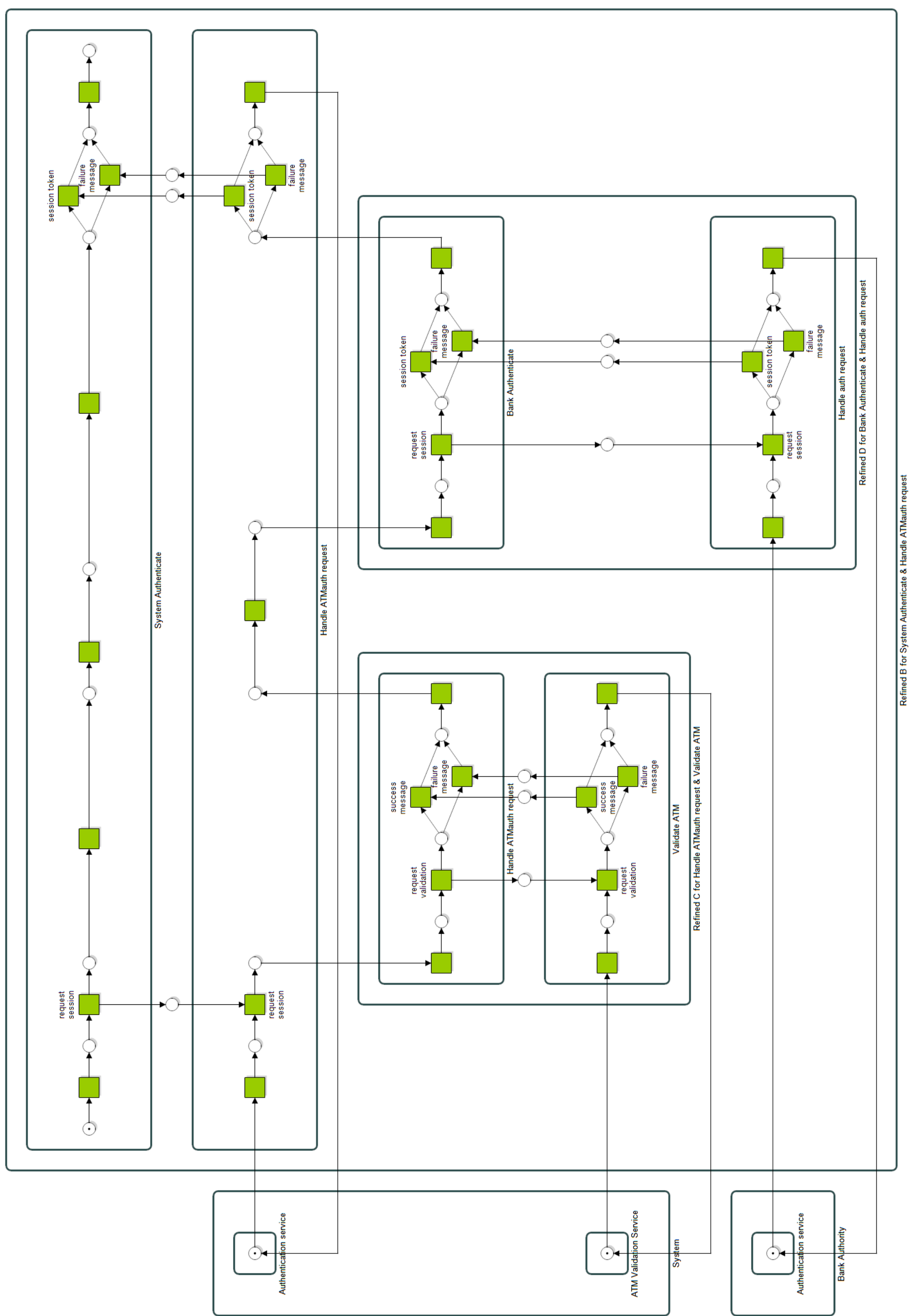


Figure 57: Protocol B refined for System Authenticate & Handle ATMauth request

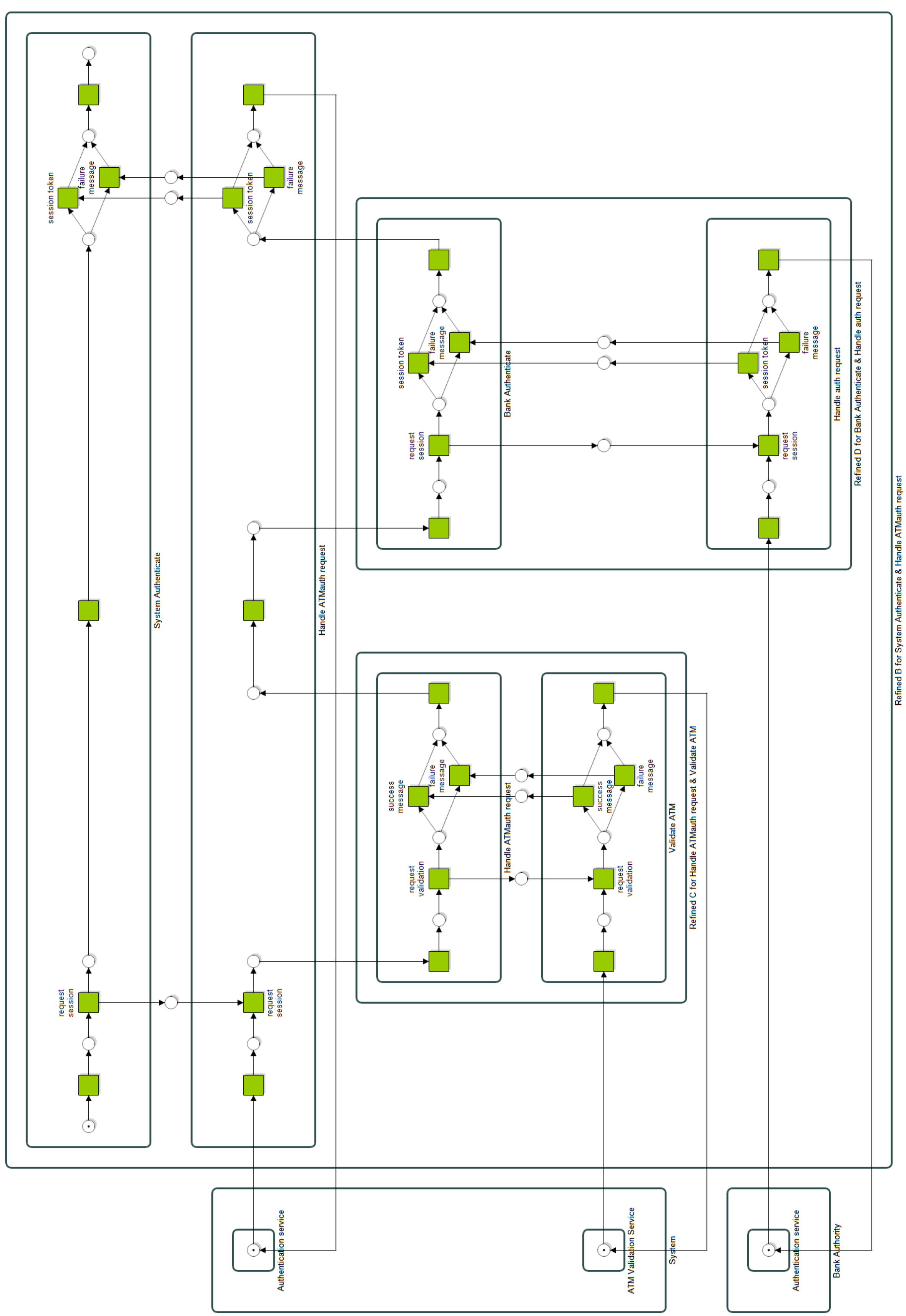


Figure 58: Protocol B refined for System Authenticate & Handle ATMauth request (reduced)

6.7.3 Compose the system

After performing the conversion into Container and Protocol nets, we can compose the system. The composed system for our Running Example is depicted in Figure 59.

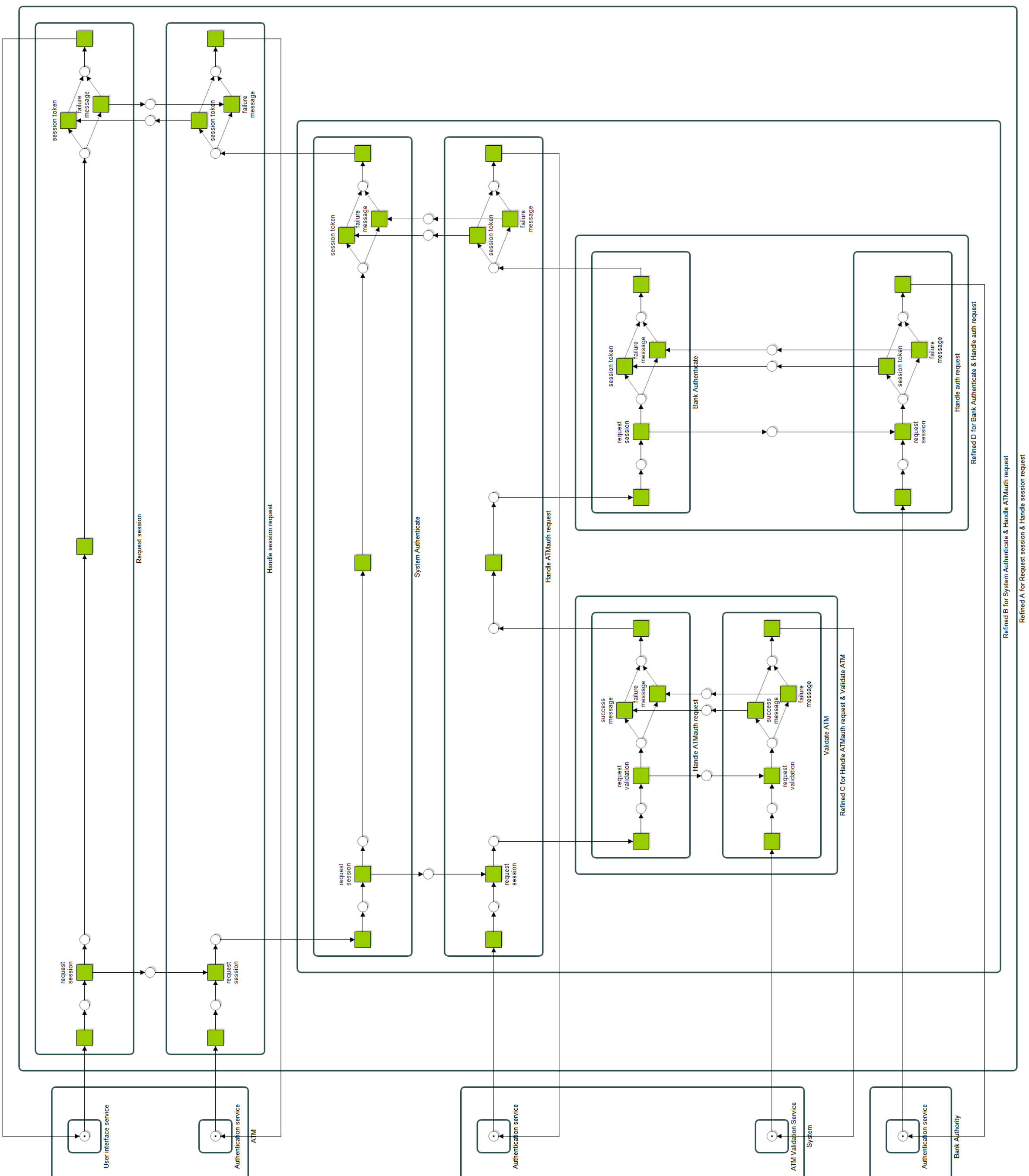


Figure 59: Composed system for Running Example

6.8 Analysis of the composed system

The composed model can be used in the analysis of the system. We can perform basic analysis on the composed Petri net:

- **Boundedness** - We can determine if the composed system is *bounded*. If a place in a Petri net is *bounded*, this place can only have a limited number of tokens at any stage of the Petri net's execution. The place p should not contain more than $k \in \mathbb{N}$ tokens. If any *unbounded* (places with no upper limit) exist in a Petri net, this is often a sign of a flaw in the design of the net - and thus the protocol.
- **Deadlocks** - We can also analyse the system for the existence of *deadlocks*. A *deadlock* is a marking from which no other marking can be reached. In other words; in a *deadlock* there are no transitions enabled to fire. A Petri net is therefore *deadlock free* if and only if at least one transition is enabled in every reachable marking.
- **Dead** - We can also determine if parts of the protocol are *dead*. A transition in a Petri net is dead if the transition is never enabled in any reachable marking.

Furthermore, we can infer *soundness* of our composed model from the protocols. Our composed system is a *service-tree* or a *component-tree* (van der Werf, 2014). A service-tree is a composition of components that is non-cyclical; the children provide services to the parents. Van der Werf (2014) showed that if the models of the individual components are sound, the model of the composition of the components is also sound.

A *sound* model has three properties: 1) it has the option to complete 2) it completes properly and 3) it has no dead transitions:

- **Option to complete** - The option to complete entails that for all markings m reachable from the initial marking m^0 it should be possible to reach the final marking Ω
- **Proper completion** - When the final marking is reached, there should be no tokens left in the rest of the model.
- **No dead transitions** - There should be no dead transitions in the model.

6.9 Conclusion

We have now shown that we can compose the structure of the Interaction Model and the underlying communication in one system. We can use this composition to simulate the system and find, for example, deadlocks.

It is, of course, possible to model the Interaction Model and Protocols in any modeling tool. The connection between the Interaction Model and the Protocols is, however, implicit then. Furthermore, it is also not possible to automatically compose a system and perform analyses on this system. This thesis, therefore, also presents a modeling tool that can be used to model the Interaction Model and Protocols. This tool is presented in Chapter 7.

The underlying composition in this chapter can be used to display feedback to the modeler in the tool.

7 Implementation

In the previous two chapters, Chapter 5 and Chapter 6, we have presented INORA. In this chapter, we present the implementation of INORA in a tool that supports creating INORA models, in order to answer SQ3. The goal of the tool is to allow a modeler to model an Interaction Oriented Architecture. Modeling both the Interaction Model and the Protocols (implemented as BPMN Choreography Diagrams) should be possible.

7.1 Implementation framework

For creating the tool, we have chosen the Sirius Project (as demonstrated by Madiot and Paganelli (2015)). On the website of Eclipse Sirius[14], they call Sirius "The easiest way to get your own modeling tool". It allows you to create a tool based on a domain model without any coding (you can, however, extend Sirius with code).

Sirius encapsulates Eclipse GMF[15] (Graphical Modeling Framework) and simplifies creating a graphical modeling tool. This is illustrated in Figure 60.

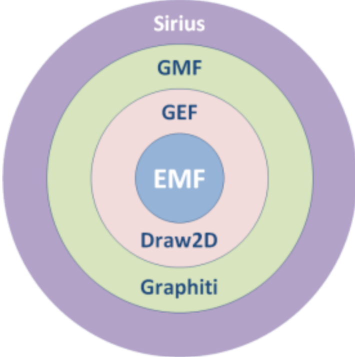


Figure 60: Hierarchy of Sirius, image from Viyović et al. (2014)

As discussed in Chapter 5 (Figure 13), the Interaction Model can be represented in multiple views. Sirius already allows for this, as it is possible to hide specific components of a model in certain representations.

7.2 Data model

The data Model of INORA as implemented in the tool is presented in Figure 61. Figure 62 highlights the part of the data model concerning the Interaction Model (as presented in Chapter 5, Figure 12). Figure 63 highlights the part of the data model that is concerned with modeling the BPMN Choreographies in the tool.

Additionally to the implementation of the data model in Figure 12 (Chapter 5) we have made some changes to the data model as implemented in the tool. First of all, the *NamedElement* was introduced. Every element in the data model that requires a label or description extends this element. Additionally, composition relations between the *InteractionModel* and the *Participant* and *ProtocolInstance* were added. This is required to allow the *Participant* and *ProtocolInstance* to be saved in the data model. The relation between *Protocol* and *Role* was converted into a composition-relation.

Additionally, the definition of the BPMN Choreography was added to the Domain Model. This was not specified in Figure 12, because INORA does not require the use of BPMN as the protocol modeling language. For the implementation of INORA, however, we have chosen to use BPMN Choreographies. The implementation of BPMN Choreographies in the tool is limited. It allows for modeling basic models and it allows modeling all required components for the use of INORA.

Every *Protocol* contains multiple *Node*s and *SequenceFlow*s. Every *SequenceFlow* originates at exactly one node and terminates at exactly one *Node*. A *Node* can be a *Message*, *Event* or *Gateway*. The

[14]https://www.eclipse.org/sirius/
[15]https://www.eclipse.org/gmf-tooling/

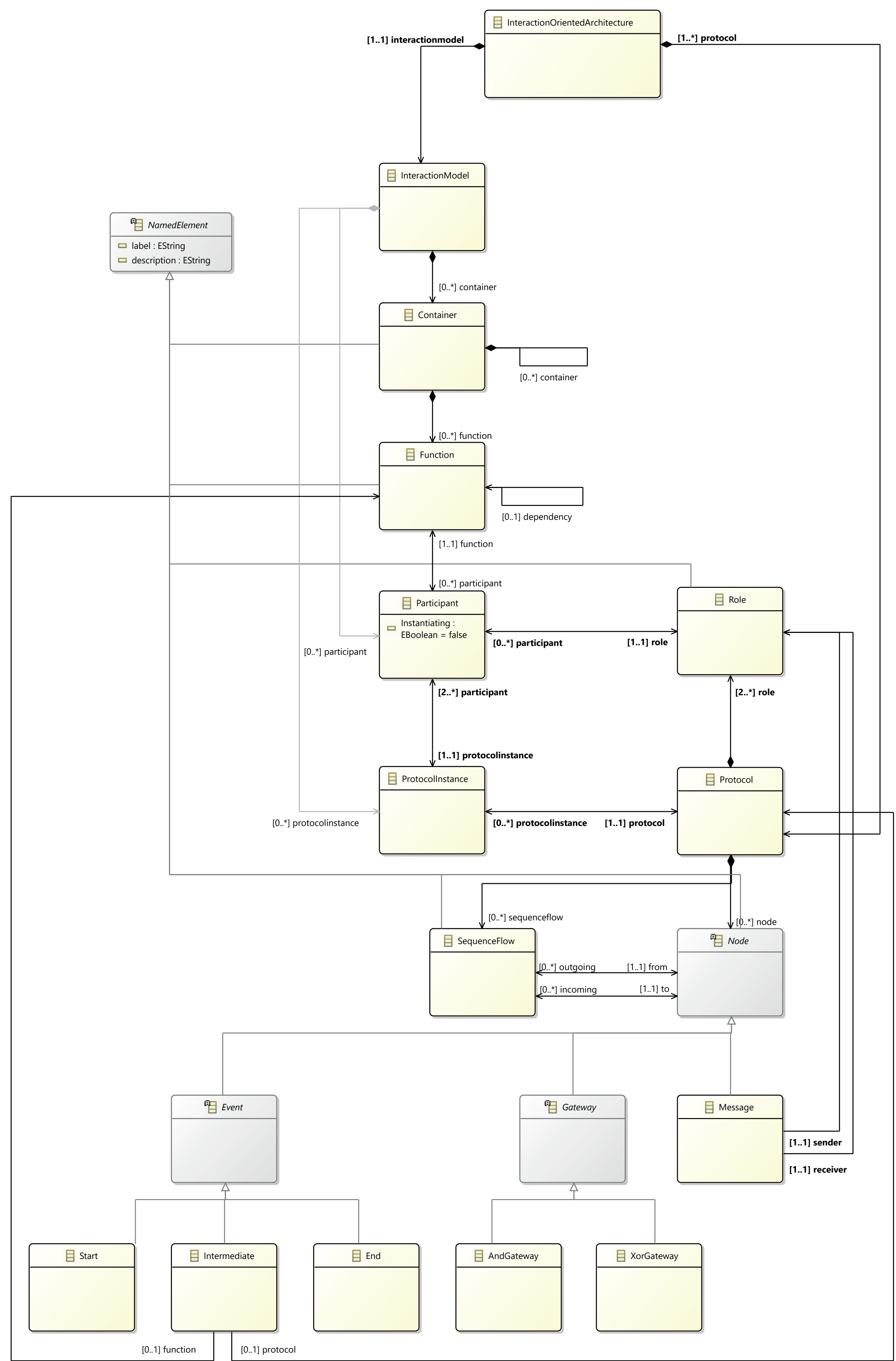


Figure 61: Data model of INORA as implemented in the tool

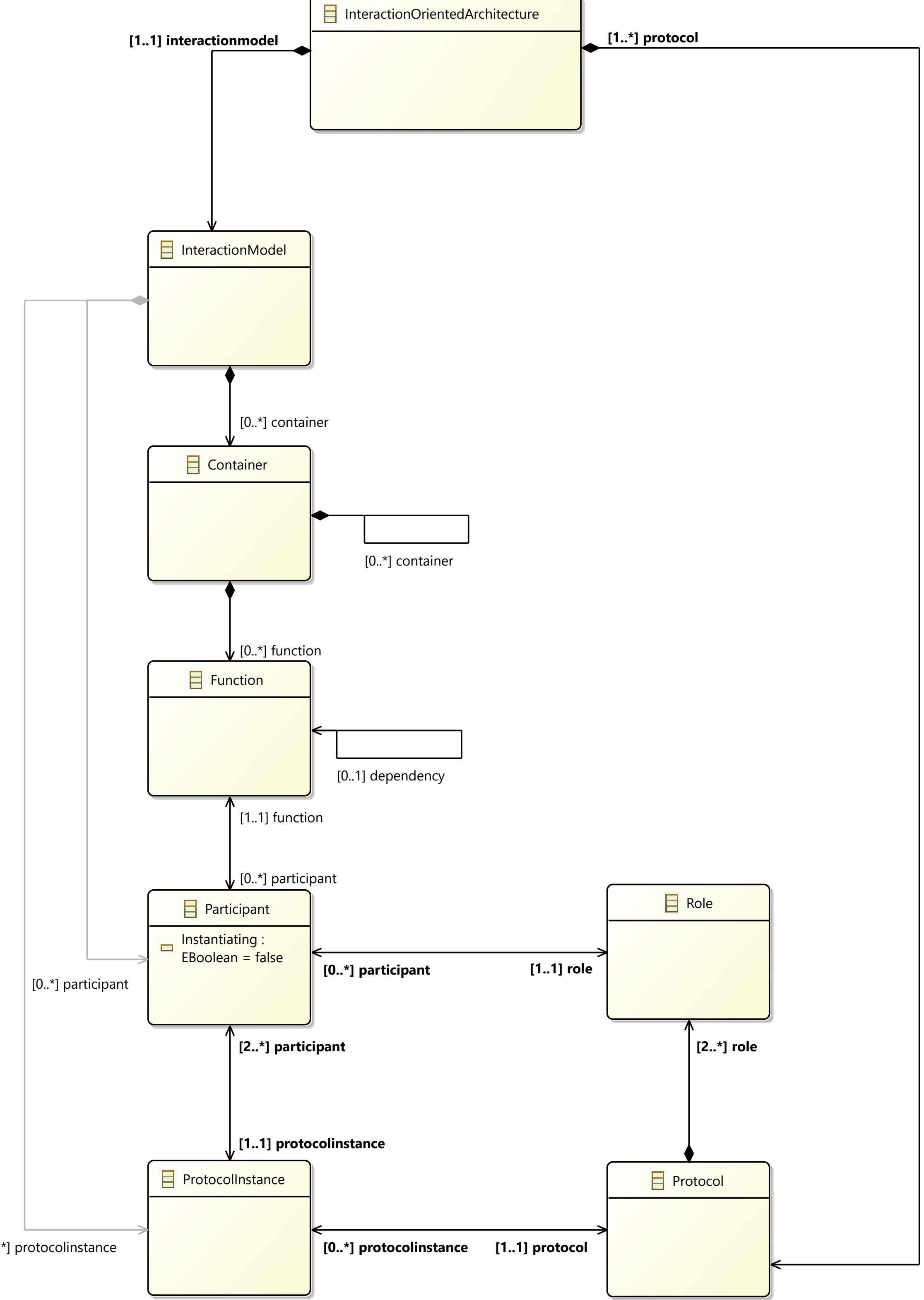


Figure 62: Data model of the Interaction Model as implemented in the tool

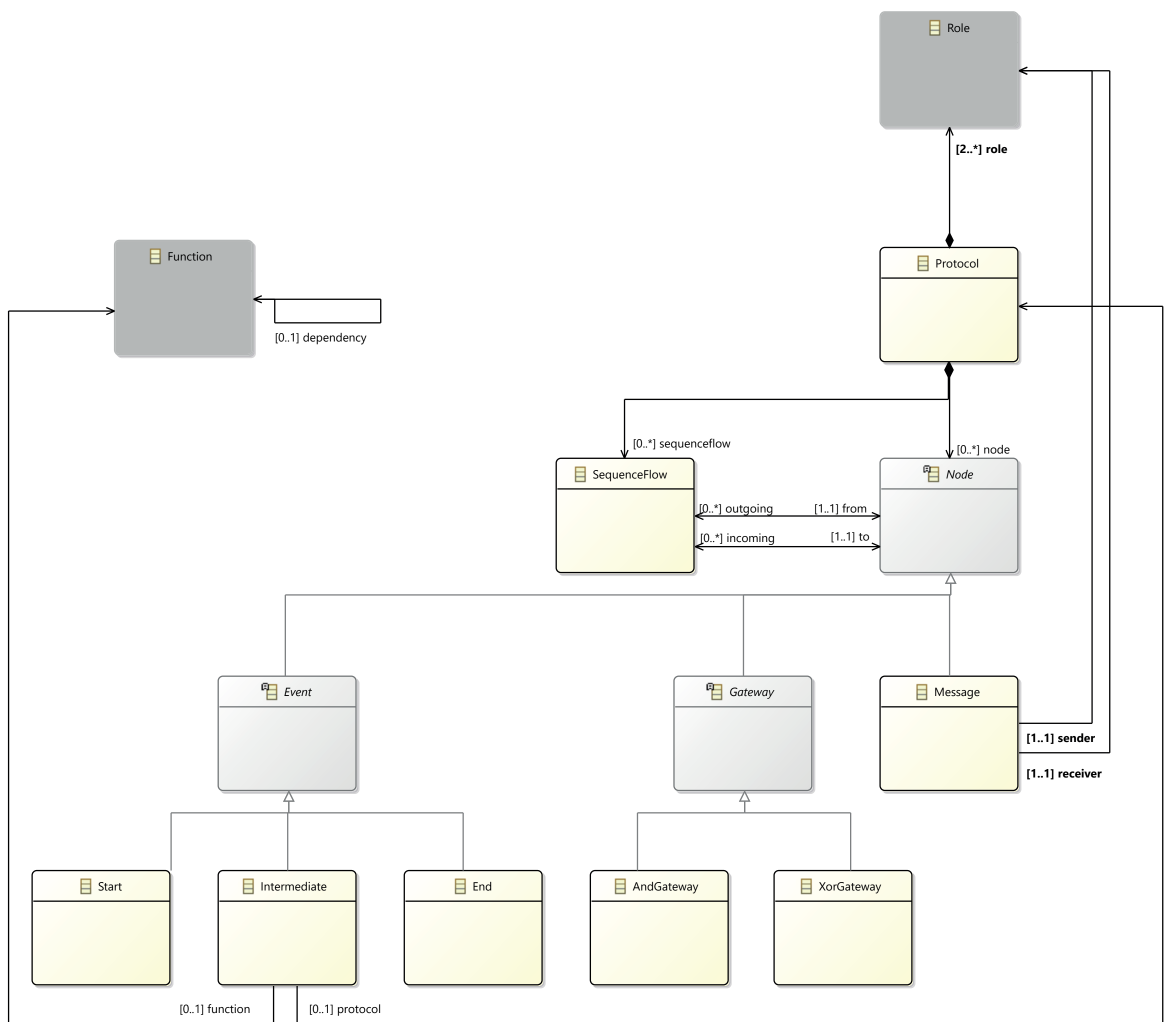


Figure 63: Data model of the BPMN Choreography Diagram as implemented in the tool

Message has exactly one sender and one receiver that each reference a *Role*. The *Gateway* can be either an *AndGateway* or an *XorGateway*. The *Event* is either a *Start*, *End* or *Intermediate* event. The *IntermedediateEvent* can reference exactly one *Function* and one *Protocol*.

7.3 Tool

In the tool, the creation of the Interaction Model and the BPMN Choreography Diagrams (Protocols) is supported. This section shows the features of both modelers. The highlights of the tool are also recorded in this screencast: https://youtu.be/aW-uxm4aLvE.

7.3.1 Interaction Model

The Interaction Model part of the tool, allows a modeler to model the interaction model of INORA. Figure 64 shows the Interaction Model of our running example as modeled in the tool.

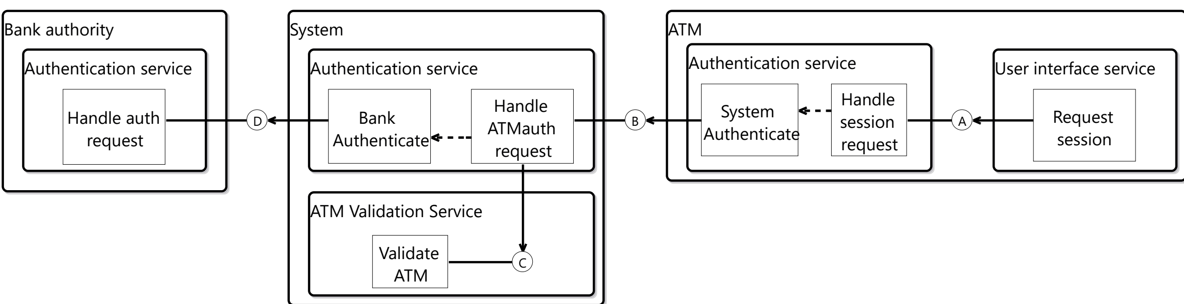


Figure 64: Interaction Model of the running example as modeled in the tool

Figure 65 shows a screenshot of the INORA tool showing the Interaction Model of the ATM running example. This is a standard Eclipse window layout. In the top left, is the *Model explorer*, where you can navigate the project. Below that the *Outline* of the currently selected model is displayed. The large area shows the currently selected model on the left and the *Palette* on the right. Below that is the *Properties* and *Problems* window. The *Properties* window shows the properties for the currently selected item.

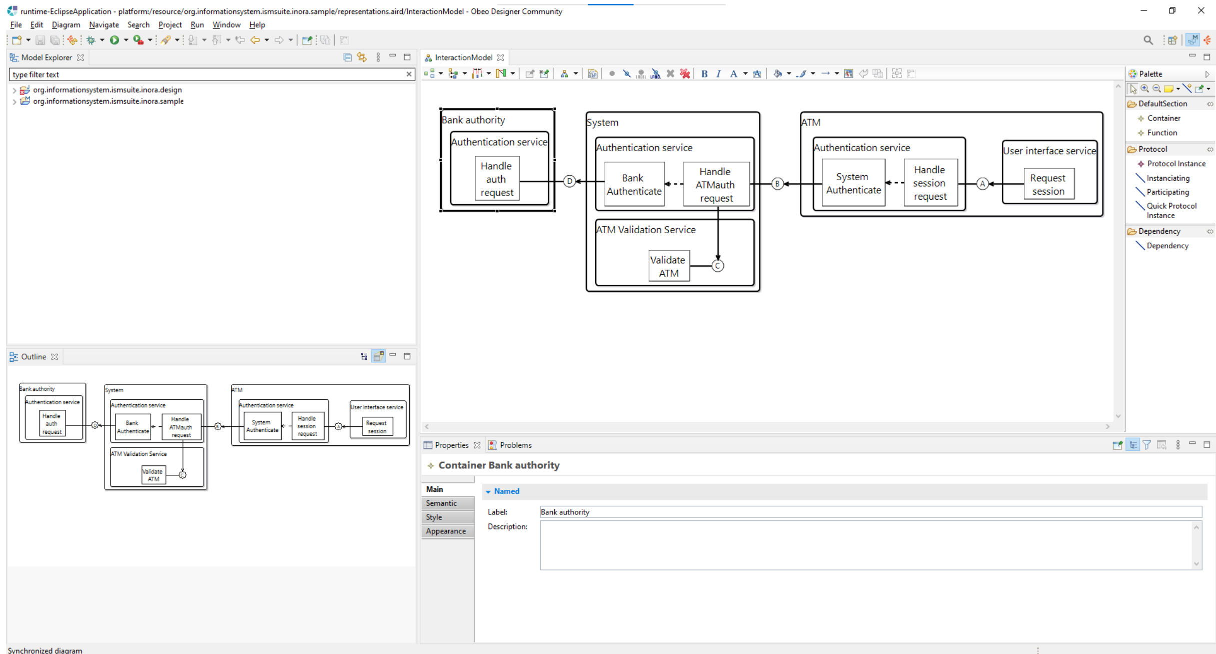


Figure 65: Screenshot of the Interaction Model in the INORA Tool

Modifying the label or description of an element

When the label or the description of any elements needs to be updated, one can select the element to open the Properties view as depicted in Figure 66. Here the label and description can be updated.

It is also possible to update the label of a named element by selecting the element and starting to type.

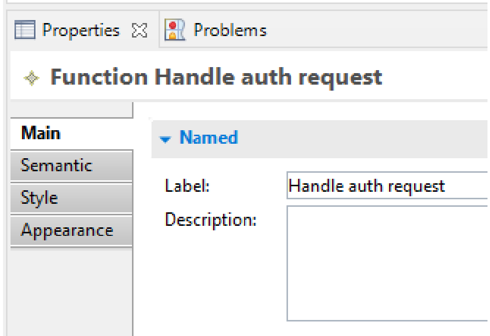


Figure 66: Screenshot of the Properties view of any Named Element in the INORA Tool

Deleting an element

When you want to delete an element, you can just select the element and press delete. When deleting a Protocol Instance, a Dialog (presented in Figure 67) will show with the option to delete the protocol as well if the selected Protocol Instance is the only instance of the protocol.

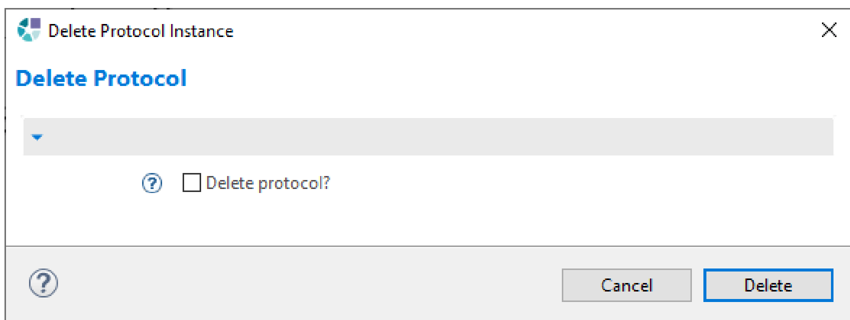


Figure 67: Screenshot of the Delete Protocol Dialog in the INORA Tool

Creating a Container or Function

When creating a container or a function, a popup will show allowing you to enter the name and description of the element (Figure 68).

Moving Containers and Functions

A container can be moved (in)to another container by dragging that container to the other container. Containers that accept other containers will highlight in gray.

The same method applies to moving functions. Hovering the function over the desired new location will highlight the container.

Creating a Protocol Instance

A Protocol Instance can be created in two ways. One can either select the *Protocol Instance* element which opens the Dialog presented in Figure 69, or select the *Quick Protocol Instance* which allows you to select two functions to quickly create a protocol between them (where the first selected function is the instantiating function). When using the later method, the Dialog as presented in Figure 70 opens.

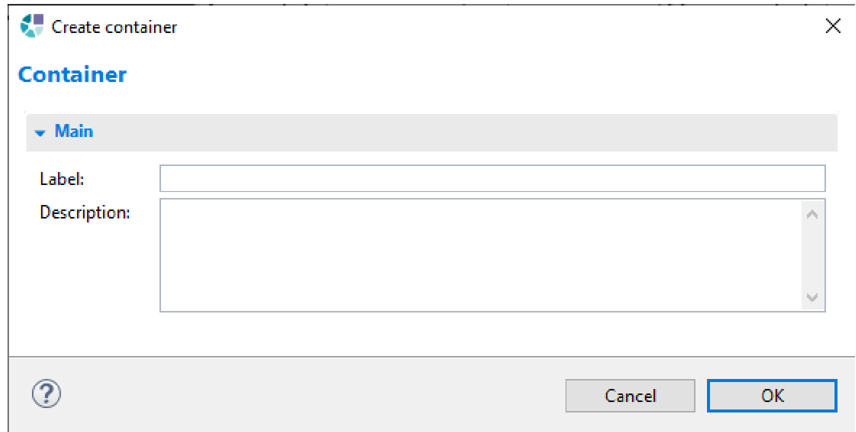


Figure 68: Screenshot of the Create Container Dialog in the INORA Tool

Under *Main*, one can select whether to create a new protocol or to use an existing protocol. When using an existing protocol, one can select the desired protocol under *Existing protocol*. When the checkmark *Create new protocol?* is checked, the fields under *New protocols* are enabled. This allows you to give the new protocol a name and assigns roles to it. When using the *Quick Protocol Instance* option is used (Figure 70), it is also possible to select the *Instantiating* and *Participating* roles.

Modifying Protocol Instance participants

When you have created a Protocol Instance (using the *Protocol Instance* element) or if you want to modify the participants of an existing protocol, you can use the *Instantiating* and *Participating* tools to connect functions to the Protocol Instance.

Once the connection is created, the Select Role Dialog opens (presented in Figure 71). This allows you to select the role.

Modifying a Protocol Instance

When selecting a Protocol Instance in the Interaction Model, the properties view for the Protocol Instance opens, as presented in Figure 72. This allows you to change the associated Protocol.

Modifying a Participating or Instantiating relation

When double clicking the relation between the *Function* and *Protocol Instance*, the

Opening a Protocol

When double-clicking the Protocol Instance, the Protocol diagram opens.

Validation

The validation can be run by right-clicking the canvas and clicking *Validate*. This will validate the following constraints:

- Every *Protocol Instance* must be instantiated by exactly one function.
- Every *Protocol Instance* should have all *Roles* of the associated *Protocol* fulfilled.
- A *Role* can not be fulfilled by more than one *Function*.

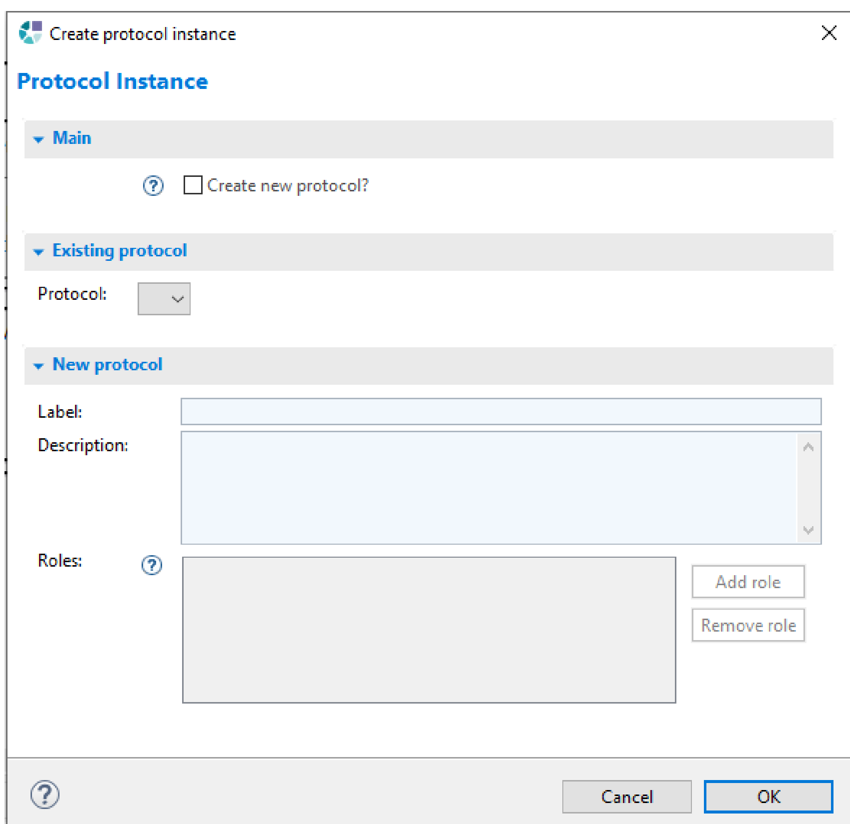


Figure 69: Screenshot of the Create Protocol Instance Dialog in the INORA Tool

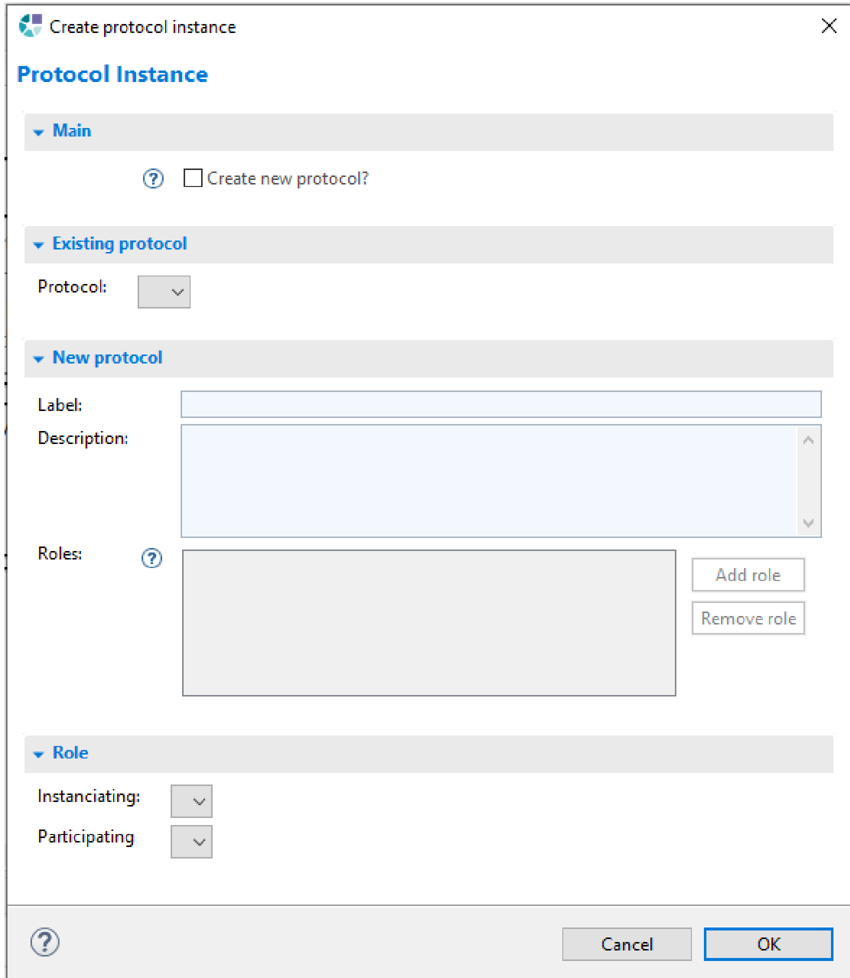


Figure 70: Screenshot of the Quick Create Protocol Instance Dialog in the INORA Tool

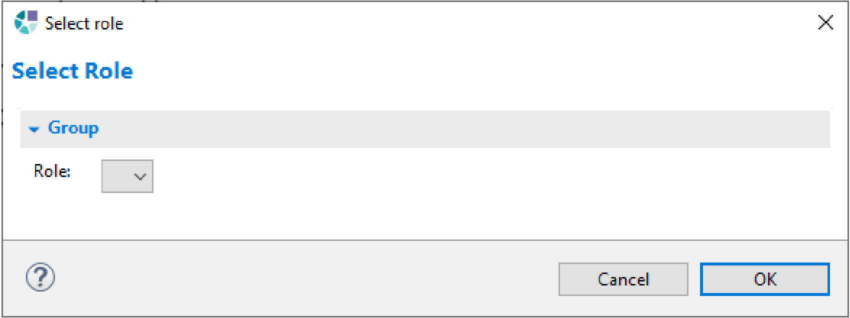


Figure 71: Screenshot of the Select Role Dialog in the INORA Tool

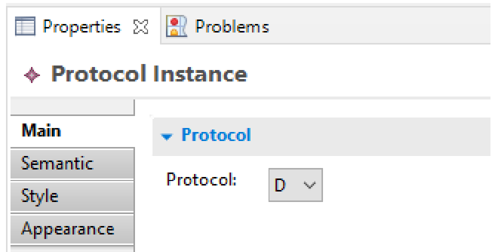


Figure 72: Screenshot of the Properties view of the Protocol Instance in the INORA Tool

7.3.2 Protocols / BPMN Choreography Diagrams

The Protocol part of the tool, allows a modeler to model the Protocols of INORA as BPMN Choreography Diagrams. Figure 79 shows the protocols of our running example as modeled in the tool.

Figure 73 shows the canvas of the Protocol design tool in the INORA tool.

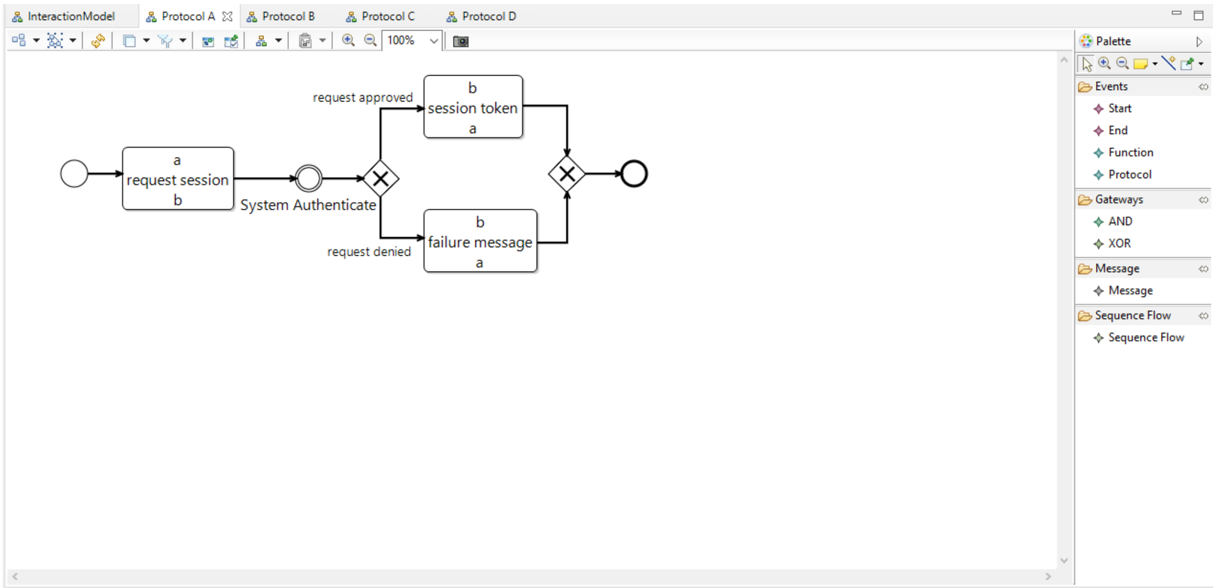


Figure 73: Screenshot of the Protocol canvas in the INORA Tool

Modifying the label or description of an element and deleting the element

In the Protocol canvas, it is also possible to modify the label and description of an element using the same methods as described in the section about the Interaction Model. Deleting an element uses the same method as in the Interaction Model as well.

Creating a start event, end event, AND gateway or XOR gateway

When creating a simple element (start event, end event, AND gateway, or XOR gateway), you can simply select the element in the Palette and click the canvas to place the element.

Creating a Function or Protocol reference (intermediate event)

When creating a reference to a Function or Protocol a dialog pops up to select the Function (Figure 74) or Protocol (Figure 75) respectively.

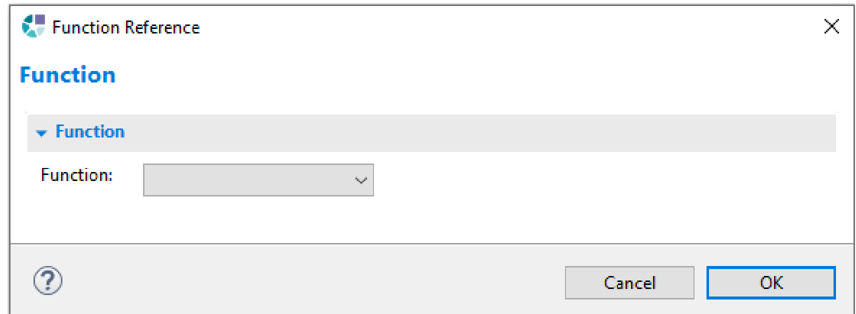


Figure 74: Screenshot of the Create Function Reference Dialog in the INORA Tool

Note: When double-clicking the reference, the corresponding model is opened.

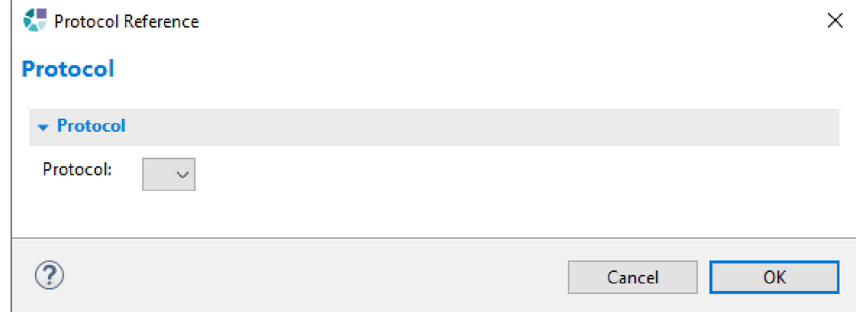


Figure 75: Screenshot of the Create Protocol Reference Dialog in the INORA Tool

Creating a Message

When creating a message, the Dialog presented in Figure 76 pops up, which allows you to type the message and select the sending and receiving roles. The *Create a new role* buttons allow you to create a new role (Figure 77).

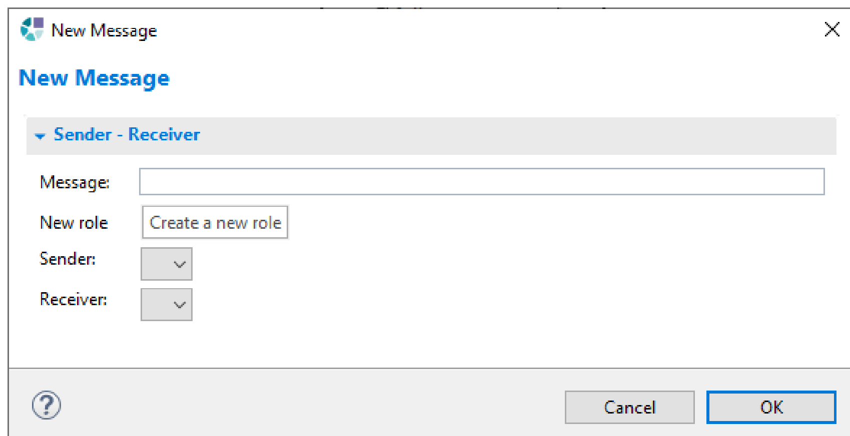


Figure 76: Screenshot of the Create Message Dialog in the INORA Tool

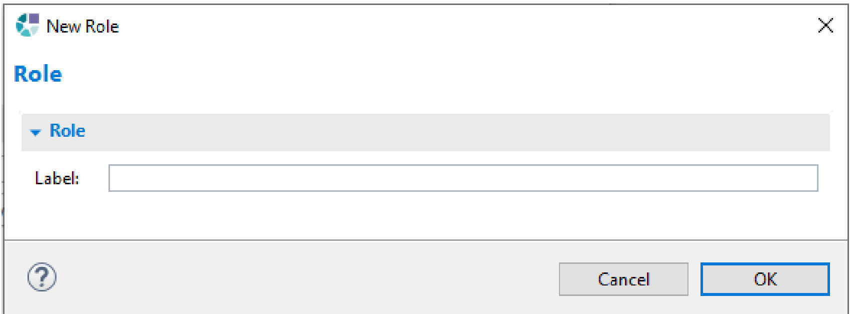


Figure 77: Screenshot of the Create Role Dialog in the INORA Tool

Modifying a Message

When selecting the *Message*, you can modify the sender and receiver (Figure 78).

Creating a Sequence Flow

To create a Sequence flow, you select the *Sequence flow* element in the Palette and select the *from* element first and the *to* element second.

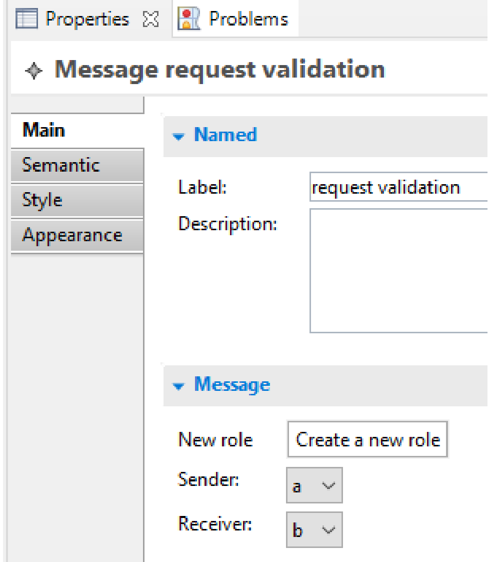
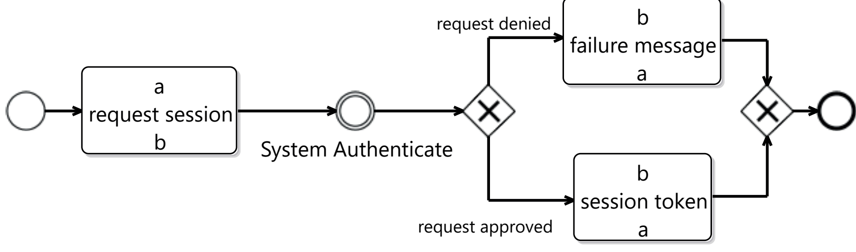


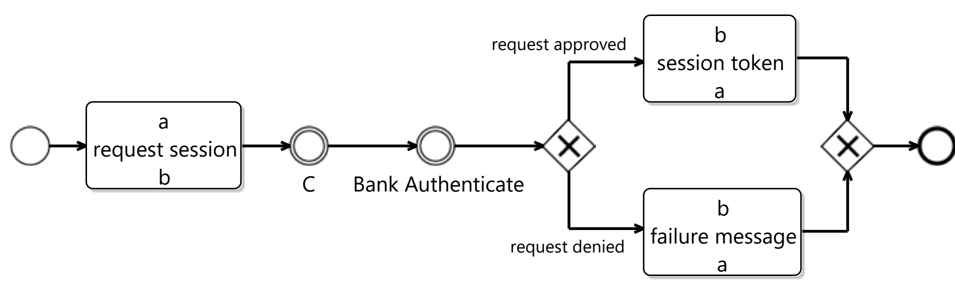
Figure 78: Screenshot of the Properties view of the Message in the INORA Tool

Validation

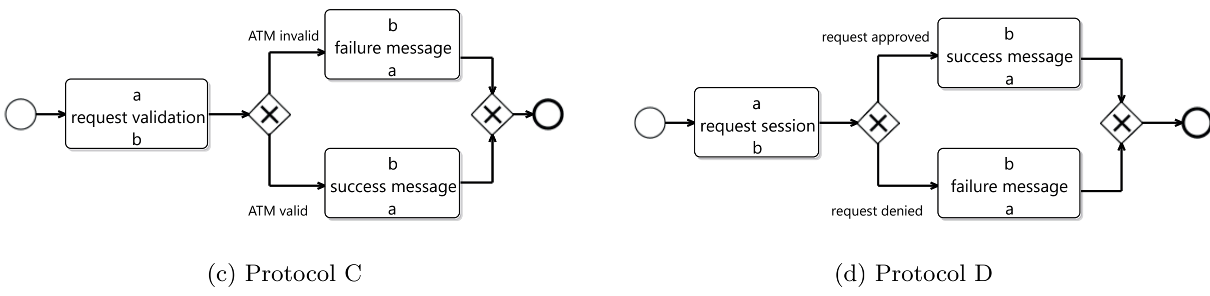
When validating the model, the tool checks that the sender and the receiver of a message are not the same.



(a) Protocol A



(b) Protocol B



(c) Protocol C (d) Protocol D

Figure 79: Protocols of the running example as modeled in the tool

7.4 Install INORA Tool

Follow these steps to install INORA. The steps are also recorded in a screencast: https://youtu.be/woHApWx20-o.

1. Download Obeo Designer Community Edition from: https://www.obeodesigner.com/.
2. Unzip Obeo Designer and start it by running *obeodesigner.exe*.
3. Go to *Help >Install New Software*.
4. Click on *Add*.
5. Provide a name. For example: *Interaction Oriented Architecture*. Provide the following URL: http://tools.architecturemining.org/inora/updatesite/.
6. Install the Interaction Oriented Architecture Feature - put the checkmarks as in Figure 80.
7. Click *Next* and click *Finish*.
8. Restart Obeo Designer

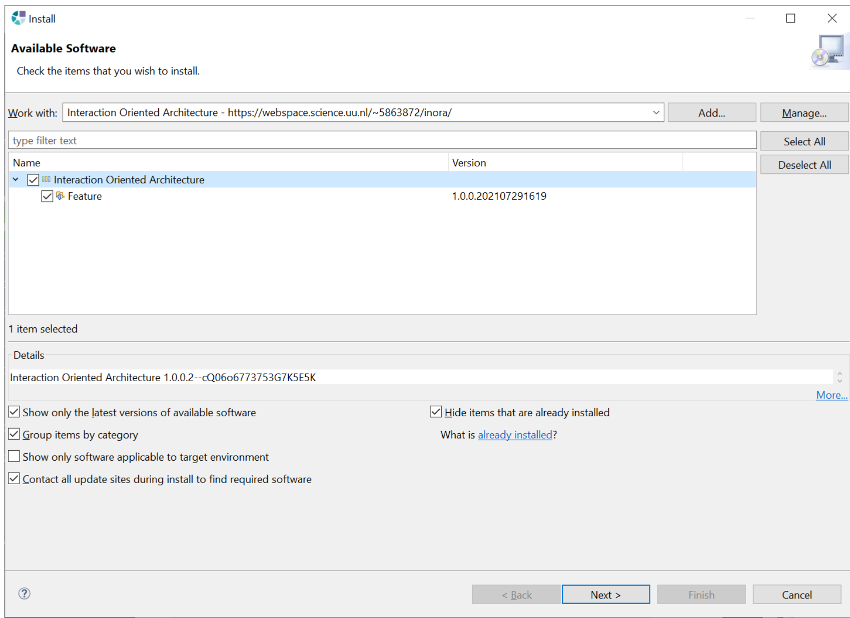


Figure 80: Screenshot of the Available Software Dialog in Eclipse Obeo Designer

7.4.1 Use the ATM example

Follow these steps to install the ATM example. The steps are also recorded in a screencast: https://youtu.be/Em6dwgVtF38.

1. Download the project from: http://www.architecturemining.org/tools/inora/.
2. In Obeo Designer, go to *File >Open Projects from File System*.
3. Next, select the downloaded file and click *Finish*.
4. You have now imported the example!

7.4.2 Create your own model

Follow these steps to create your own model. The steps are also recorded in a screencast: https://youtu.be/g2wev1dKgkc.

1. Create a new Modeling project via *File >New >Modeling project*
2. Right-click your newly created project and go to *New >Other* >InteractionOrientedArchitecture Model. Select *Interaction Oriented Architecture* as modeling object.

3. Right-click your project again and click *Viewpoint selection*. Put a checkmark before *InteractionOrientedArchitecure* and click OK
4. Open your created *.interactionorientedarchitecture* file. Right-click the *Interaction Oriented Architecture* object and select *New Child >Interaction Model*.
5. Right-click your project once again and go to *Create representation*. Next select *Interaction Model*, *Next* and *Finish*.
6. You can now start modeling!

7.4.3 Download the INORA Tool source code

The INORA Tool source-code is available on GitHub:
https://github.com/ArchitectureMining/INORA.

7.5 Conclusion

In this chapter, we have shown that we have implemented INORA in a tool. This allows us to answer SQ3.

SQ3: *What are tools and techniques to support the proposed approach?*

We have created an Eclipse Tool that supports creating an Interaction Oriented Architecture. You can model the Interaction Model and the Protocols as BPMN Choreography Diagrams.

All screencasts used in this chapter are also available through the following playlist:
https://www.youtube.com/playlist?list=PL98xzWk0NAVzlvTuXPwqjcZ9RelsyXD3E.

8 Conclusions

In this thesis, we have laid the foundations for an interaction oriented modeling method. We have shown that our approach with INORA for modeling complex interactions is feasible. In Chapter 5 and Chapter 6 we have shown that it is possible to model the complex interactions in our running example. In Chapter 7 we have shown that our tool also supports the modeling of INORA models.

8.1 Implications

In Chapter 4 we have introduced a set of attributes to qualify an interaction modeling technique. Our technique allows for asynchronous communication. In Chapter 5 we have introduced the visual notation for INORA. In Chapter 6 we have presented the semantics of INORA. Having logical based semantics, INORA is a formal language. INORA is specifically designed to allow for a compositional design of the system - it allows for modeling every protocol independently. In Chapter 7 we have introduced the foundations for the INORA modeling tool. In Chapter 6 we have shown that it is possible to use the composed model to analyze the system.

8.2 Answers to research questions

In Chapter 1 we have identified that it is often hard to validate the interactions of an entire system. We called those types of interactions: complex interactions. We formulated the following problem statement:

> *This thesis aims to improve* component-based software design *by* creating a systemic approach to design and analyze complex interactions *that* helps software architects to maintain internal consistency in the interaction design *in order to* improve the quality of the designed architecture.

We have formulated this problem statement into the following research question:

> RQ1: *What is a systematic approach to design and analyze complex interactions between components?*

We have aimed to answer the main research question by answering the following subquestions:

> SQ1: *What are current methods to model interaction?*

In Chapter 4 we have identified seven methods to model component interaction: Pi-calculus, Petri nets, WS-CDL, BPEL4Chor, BPMN, Let's Dance, and Interface automata.

The existing methods do not allow the modeling of complex interactions. To create a method that does allow for this, we have introduced INORA in Chapter 5 and Chapter 6.

> SQ2: *What are the concepts and relations required to design complex interactions between components?*

In this thesis, we have proposed the Interaction Oriented Architecture (INORA). It consists of the Interaction Model and a set of Protocols. The Interaction Model defines the components of a system and the communication between the components. The Protocols further define the contents of the communication. We have provided the formal definition for the Interaction Model. Furthermore, we have shown that the Interaction Model and the set of Protocols can be used to create a composed model of the system. This composed model can be used for analysis.

> SQ3: *What are tools and techniques to support the proposed approach?*

In Chapter 7 we have shown the implementation of INORA in an Eclipse modeling tool. The INORA tool in its current state provides the foundation for a modeling suite that allows modeling an architecture. By creating an Interaction Oriented Architecture for our running example we have shown the feasibility of our approach.

8.3 Limitations & Suggestions for Future Research

In this section, we list the limitations of this thesis and some suggestions for future work. We have divided all points into two sections: one section for the Interaction Oriented Architecture as a modeling technique and one section for all the limitations and future work for the INORA tool.

8.3.1 INORA modeling

- **Usability & Practical application**
  We have shown the feasibility of modeling complex interactions using INORA by creating an Interaction Oriented Architecture for our running example. This is, of course, just one implementation based on a fictional example.

  Future work: Future research should look at more elaborate complex interactions (for example in case studies). This will probably bring to light some exceptions which currently cannot be modeled in INORA.

  Furthermore, we have focused on the modeling language and the semantics of INORA in this thesis. We have not evaluated its use by architects.

  Future work: Future research should investigate the usability of INORA as an architecture modeling tool for interactions.

- **Protocol modeling techniques**
  We have not thoroughly investigated using other Protocol-modeling techniques. We have used BPMN Choreography Diagrams because it provides modeling elements required for modeling Protocols and it is used in practice and academia. It is, however, possible that other languages are as suitable or even more suitable for modeling Protocols than BPMN Choreography Diagrams.

  Future work: Future research could look into the possibility of other modeling techniques for modeling the protocols.

- **Multi-instance problem**
  In Chapter 5 we have mentioned that we currently do not allow modeling multi-instance containers. In this thesis, we have assumed that every container has only one instance. In practice, however, a certain container with functions can be multi-instance, i.e. running multiple times. Our approach does not handle these types of containers.

  Future work: Future research should look at the challenges that multi-instance provides and the implications on modeling those multi-instance components using INORA.

- **Dynamic instantiation**
  In the approach that we have presented in this thesis we know the whole network of components when creating the architecture. In real-world applications this is not always the case. It is therefore interesting to look at the possibility to dynamically add and/or remove components.

  Future work: Future research could look at the implications of dynamically adding or removing components from the architecture. How does this affect the design of a system?

- **Methodology**
  In this thesis, we have presented INORA as a modeling method. In Chapter 2, we have introduced the Three Peaks model. We think that the use of INORA in this process of architectural design is useful.

  Future work: Future research should point out wherein the process INORA fits in exactly and how it can help in the specification and design phases of software architecture.

- **Code generation**
  After designing the architecture using INORA, code could be generated. Herrington (2003) describes

how code can automatically be generated from models.

Future work: Future research could look at the possibilities to use INORA to generate high-level code that includes the modeled components and communications. When looking at microservices (as introduced in Chapter 3), one could for example imagine that the code generation could generate the modeled components as services and generate REST endpoints for the communication.

– **Model generation**
It would also be possible to create documentation in INORA from existing code or existing running systems. One could imagine that INORA could be generated by looking at existing code to discover components and functions or look at existing endpoints to discover communication. Another way to discover components and communication is by using process mining; the event log of the execution could show the different components and the communication between them. There are existing approaches, like the one from C. Liu et al. (2016).

Future work: Future research could look at the possibilities to automatically generate INORA models from existing systems by analyzing code or execution logs.

8.3.2 INORA tool

– **Constraints & validation in the tool**
Due to time constraints, not all validations and constraints have been added to the tool. The following constraints and validations are currently missing in the tool and should be added.

  – The transitive closure of *uses* (C3) and *interacts_with* (C8).

  – Syntax validation for the BPMN Choreography Diagrams.

  – Only present the protocol participant functions and protocols in the BPMN Choreography design tool.

  – Improvements to the visual elements used in the tool.

– **Auxiliary functions**
Lago et al. (2014) have identified auxiliary functions that should be supported in architectural modeling tools. They mention that a tool should have the ability to facilitate collaboration. At the moment the tool does not specifically support that. This property is also identified in the survey by Malavolta et al. (2012). Additionally, a tool should allow for versioning. In the basis, our tool does support this by using, for example, Git [16]. It would however be of added value to directly support versioning in the tool. This would, for example, allow to enable comparison in the tool.

– **Analysis & visualisation**
In the semantics of INORA that we have presented in Chapter 6, we have introduced translations from the Interaction Model and the protocols to Petri nets. Thereafter, we have introduced how to combine those models into one model. This allows us to perform analysis on the composed model. In the tool, this analysis should be implemented. The results of the analysis should be used to visualize issues in the model. Preferably, it also gives suggestions to (automatically) fix issues. This will help the architect to fix those issues.

– **Evaluation of the tool**
Once the INORA-tool is more complete, an evaluation of the usefulness and usability of the tool should be conducted. We expect that the tool can be used for designing interactions by architectures. Furthermore, we expect that the tool will provide novice architects with a tool to practice the creation of architectures and the understanding of concurrency in an architecture.

[16] https://git-scm.com/

References

Abouzaid, F. (2006). Toward a pi-calculus based verification tool for web services orchestrations. *Computer Supported Acitivity Coordination*, 23–34.

Abuosba, K. A., & El-Sheikh, A. A. (2008). Formalizing service-oriented architectures. *IT Professional*, *10*(4), 34–38.

Allen, R., Douence, R., & Garlan, D. (1998). Specifying and analyzing dynamic software architectures. *International Conference on Fundamental Approaches to Software Engineering*, 21–37.

Barroca, L., & Hall, J. (2000). *Software architectures: Advances and applications*. Springer Science & Business Media.

Bass, L., Clements, P., & Kazman, R. (2003). *Software architecture in practice*. Addison-Wesley Professional.

Basu, S., Bultan, T., & Ouederni, M. (2012). Deciding choreography realizability. *Acm Sigplan Notices*, *47*(1), 191–202.

Beauxis, R., Palamidessi, C., & Valencia, F. D. (2008). On the asynchronous nature of the asynchronous π-calculus. *Concurrency, graphs and models* (pp. 473–492). Springer.

Bell, D. (2004). Uml basics: The component diagram. *IBM Global Services*.

Benner, K. M., Feather, M. S., Johnson, W. L., & Zorman, L. A. (1993). Utilizing scenarios in the software development process. *Information system development process* (pp. 117–134). Elsevier.

Bernardo, M., Aldini, A., & Corradini, F. (2010). A process algebraic approach to software architecture design.

Berthelot, G. (1978). *Vérification de réseaux de petri* (Doctoral dissertation). PhD thesis, Université Pierre et Marie Curie, Paris.

Brinkkemper, S., & Pachidi, S. (2010). Functional architecture modeling for the software product industry. *European Conference on Software Architecture*, 198–213.

Chaudron, M., & de Jong, E. (2000). Components are from mars. *Proc. 15 IPDPS 2000 Workshops on Parallel and Distributed Processing, Lecture Notes In Computer Science*, 727–733.

Cleland-Huang, J., Hanmer, R. S., Supakkul, S., & Mirakhorli, M. (2013). The twin peaks of requirements and architecture. *IEEE Software*, *30*(2), 24–29.

Cortes-Cornax, M., Dupuy-Chessa, S., Rieu, D., & Dumas, M. (2011). Evaluating choreographies in bpmn 2.0 using an extended quality framework. *International Workshop on Business Process Modeling Notation*, 103–117.

Crnkovic, I. (2001). Component-based software engineering—new challenges in software development. *Software Focus*, *2*(4), 127–133.

De Alfaro, L., & Henzinger, T. A. (2001). Interface automata. *ACM SIGSOFT Software Engineering Notes*, *26*(5), 109–120.

Decker, G., Kopp, O., & Barros, A. (2008). An introduction to service choreographies (servicechoreographien–eine einführung). *it-Information Technology*, *50*(2), 122–127.

Decker, G., Kopp, O., Leymann, F., & Weske, M. (2007). Bpel4chor: Extending bpel for modeling choreographies. *ICWS 2007*, 296–303. https://doi.org/10.1109/ICWS.2007.59

Decker, G., Puhlmann, F., & Weske, M. (2006). Formalizing service interactions. *International Conference on Business Process Management*, 414–419.

Decker, G., & Weske, M. (2011). Interaction-centric modeling of process choreographies. *Information Systems*, *36*(2), 292–312.

Deng, S., Wu, Z., Zhou, M., Li, Y., & Wu, J. (2006). Modeling service compatibility with pi-calculus for choreography. *international conference on conceptual modeling*, 26–39.

Eide, E., Reid, A., Regehr, J., & Lepreau, J. (2002). Static and dynamic structure in design patterns. *Proceedings of the 24th international conference on Software engineering*, 208–218.

Flissi, A., Gransart, C., & Merle, P. (2005). A component-based software infrastructure for ubiquitous computing. *The 4th International Symposium on Parallel and Distributed Computing (ISPDC'05)*, 183–190.

Group, I. A. W. et al. (1999). Recommended practice for architectural description. *IEEE P1471 D*, *5*.

Herrington, J. (2003). *Code generation in action*. Manning Publications Co.

Hofmeister, C., Nord, R. L., & Soni, D. (1999). Describing software architecture with uml. *Working Conference on Software Architecture*, 145–159.

Jansen, A., & Bosch, J. (2005). Software architecture as a set of architectural design decisions. *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, 109–120.

Jifeng, H., Li, X., & Liu, Z. (2005). Component-based software engineering. *International Colloquium on Theoretical Aspects of Computing*, 70–95.

Lago, P., Malavolta, I., Muccini, H., Pelliccione, P., & Tang, A. (2014). The road ahead for architectural languages. *IEEE Software*, *32*(1), 98–105.

Le, H. A., & Truong, N. T. (2012). Modeling and verifying ws-cdl using event-b. *International Conference on Context-Aware Systems and Applications*, 290–299.

Lewis, J., & Fowler, M. (2014). Microservices. https://martinfowler.com/articles/microservices.html

Li, X., Liu, Z., & Jifeng, H. (2004). A formal semantics of uml sequence diagram. *2004 Australian Software Engineering Conference. Proceedings.*, 168–177.

Liu, C., van Dongen, B., Assy, N., & van der Aalst, W. M. (2016). Component behavior discovery from software execution data. *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, 1–8.

Liu, Z., & Joseph, M. (1999). Specification and verification of fault-tolerance, timing, and scheduling. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *21*(1), 46–89.

Lohmann, N., Kopp, O., Leymann, F., & Reisig, W. (2007). Analyzing bpel4chor: Verification and participant synthesis. *International Workshop on Web Services and Formal Methods*, 46–60.

Lüer, C., & Rosenblum, D. S. (2001). Uml component diagrams and software architecture-experiences from the wren project. *1st ICSE Workshop on Describing Software Architecture with UML*, 79–82.

Madiesh, M., & Wirtz, G. (2008). Ws-cdl creator: Modeling ws-cdl using bpmn. *The 2008 International Conference on Semantic Web and Web Services (SWWS08), CSREA Press.*

Madiot, F., & Paganelli, M. (2015). Eclipse sirius demonstration. *P&D@ MoDELS*, *1554*, 9–11.

Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., & Tang, A. (2012). What industry needs from architectural languages: A survey. *IEEE Transactions on Software Engineering*, *39*(6), 869–891.

Massuthe, P., Reisig, W., & Schmidt, K. (2005). *An operating guideline approach to the soa*. Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät . . .

McIlroy, M. D. (1968). Mass-produced software components. *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, 88–98.

Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, *77*(4), 541–580.

Nadareishvili, I., Mitra, R., McLarty, M., & Amundsen, M. (2016). *Microservice architecture: Aligning principles, practices, and culture.* " O'Reilly Media, Inc."

Papazoglou, M. P., Traverso, P., Dustdar, S., & Leymann, F. (2007). Service-oriented computing: State of the art and research challenges. *Computer*, *40*(11), 38–45.

Petri, C. A. (1962). Kommunikation mit automaten.

Poizat, P., & Salaün, G. (2012). Checking the realizability of bpmn 2.0 choreographies. *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, 1927–1934.

Raedts, I., Petkovic, M., Usenko, Y. S., van der Werf, J. M. E., Groote, J. F., & Somers, L. J. (2007). Transformation of bpmn models for behaviour analysis. *MSVVEIS*, *2007*, 126–137.

Reisig, W. (1985). Petri nets with individual tokens. *Theoretical Computer Science*, *41*, 185–213.

Reisig, W. (2012). *Petri nets: An introduction* (Vol. 4). Springer Science & Business Media.

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

Sangiorgi, D., & Sangiorgi, D. (2011). Pi-calculus. In D. Padua (Ed.), *Encyclopedia of parallel computing* (pp. 1554–1562). Springer US. https://doi.org/10.1007/978-0-387-09766-4_202

Schmidt, D., & Kuhns, F. (2000). An overview of the real-time corba specification. *Computer*, *33*(6), 56–63.

Sheikh, H. R. (2012). Comparing corba and web-services in view of a service oriented architecture. *International Journal of Computer Applications*, *39*(6), 47–55.

Sommerville, I. (2001). Software engineering. 6th. *Ed., Harlow, UK.: Addison-Wesley.*

Szyperski, C., Gruntz, D., & Murer, S. (2002). *Component software: Beyond object-oriented programming.* Pearson Education.

Thong, W. J., & Ameedeen, M. (2015). A survey of petri net tools. *Advanced computer and communication engineering technology* (pp. 537–551). Springer.

Van Der Aalst, W. M., Lohmann, N., Massuthe, P., Stahl, C., & Wolf, K. (2010). Multiparty contracts: Agreeing and implementing interorganizational processes. *The Computer Journal*, *53*(1), 90–106.

van der Werf, J. M. E. (2011). Compositional design and verification of component-based information systems.

van der Werf, J. M. E. (2014). Compositional verification of asynchronously communicating systems. *International Conference on Formal Aspects of Component Software*, 49–67.

van Hee, K., Oanea, O., Post, R., Somers, L., & van der Werf, J. M. (2006). Yasper: A tool for workflow modeling and analysis. *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*, 279–282.

van Hee, K. M., Sidorova, N., & van der Werf, J. M. (2010). Construction of asynchronous communicating systems: Weak termination guaranteed! *International Conference on Software Composition*, 106–121.

van Hee, K. M., Sidorova, N., & van der Werf, J. M. (2011). Refinement of synchronizable places with multi-workflow nets. *International Conference on Application and Theory of Petri Nets and Concurrency*, 149–168.

Victor, B., & Moller, F. (1994). The mobility workbench—a tool for the π-calculus. *International Conference on Computer Aided Verification*, 428–440.

Viyović, V., Maksimović, M., & Perisić, B. (2014). Sirius: A rapid development of dsm graphical editor. *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*, 233–238.

Wieringa, R. J. (2014). *Design science methodology for information systems and software engineering*. Springer.

Woods, E., & Rozanski, N. (2010). Unifying software architecture with its implementation. *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, 55–58.

Zaha, J. M., Barros, A., Dumas, M., & ter Hofstede, A. (2006). Let's dance: A language for service behavior modeling. *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, 145–162.