Master's thesis

# Automated Feedback on Interaction Oriented Software Architecture


Kyle Dingenouts, BSc
k.w.c.dingenouts@uu.nl
5929164


April 3, 2023


**Supervisors**
Dr. ir. J.M.E.M. (Jan Martijn) van der Werf
Dr. ir. X. (Xixi) Lu

Utrecht University
Faculty of Science
Master Business Informatics

**Abstract**
Software architecture can describe software systems: it is a composition of viewpoints to describe the system. This paper focuses on the interaction between components in a system. These can be modeled as a choreography, a BPMN-like model capturing all possible interaction scenarios between two components. In this paper, we show that it is feasible to analyze a composed set of these choreographies: a tree of choreographies in which each member may refer to another. The two major components of the analysis are 1) the correctness by structure: a choreography follows strict grammar and assumptions and is therefore guaranteed sound and 2) the choreography is transformed to a Petri net which is checked by an external state explosion tool for proper completion. This paper shows the theoretical techniques to verify a composed choreography, and implements the solutions into a single educational modeler tool: INORA2.

# Contents


**1 Introduction** **5**
1.1 General . . . 5
1.2 Problem Statement . . . 6
1.3 Research Questions . . . 6
1.4 Research Method . . . 7
1.5 Running Example . . . 7
1.6 Thesis roadmap . . . 8

**2 Basic Notions** **10**
2.1 Software Architecture & Views . . . 10
2.2 Choreographies & User Scenarios . . . 11
2.3 Interaction Model . . . 13
2.4 Composed Choreographies . . . 15
2.5 Petri Nets . . . 17

**3 Related Work** **22**
3.1 Introduction . . . 22
3.2 Translating a Choreography to Petri Nets . . . 22
3.3 Tool-based Verification and Analysis of Petri nets . . . 24
3.4 Feedback on Models . . . 25
3.5 Summary . . . 28

**4 A Robust Translation** **29**
4.1 Constraints to INORA protocols . . . 30
4.2 Petri Net Transformation . . . 30
4.3 Block structured Choreographies . . . 34
4.4 Summary . . . 39

**5 Implementation** **40**
5.1 What is INORA2? . . . 40
5.2 Algorithms . . . 48
5.3 Documentation, licensing and installation . . . 52
5.4 Version Comparison . . . 53
5.5 Summary . . . 53

**6 Petri net model checking** **54**
6.1 Model . . . 54
6.2 DAME LoLA . . . 56
6.3 Summary . . . 56

**7 Application walk-through** **57**

**8 Conclusions** **67**
8.1 Answers to Research Questions . . . 67
8.2 Discussion and limitations . . . 68
8.3 Future work . . . 69
8.4 Acknowledgements . . . 70

**A INORA meta model**

## B Algorithm for generating Choreography Tree

## C Algorithm for generating Petri net

## D Petri net translation segments

## E DAME LoLA

# Chapter 1

# Introduction

This chapter briefly introduces the research field, basic concepts and defines the problems this thesis aims to solve.

## 1.1 General

One of the most important aspects of the software industry is describing a system as a whole. A description covering all relevant perspectives is called the *Software architecture*. “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.” [1]. As it is impossible to capture the essence and detail of a system architecture in a single model, the system is considered in terms of multiple viewpoints. Within the research field of “interaction-oriented software architecture”, the most important viewpoints are the functional and concurrency viewpoints. [2]. These will be elaborated in chapter 2.

In [2] the author proposes to connect these two viewpoints and to describe them using a specific set of models. This approach is called INORA, which depicts a functional view of a system with a model composed of containers containing functions called the interaction model. These functions can communicate with each other using protocols, which is an aggregated model depicting a set of all possible interaction scenarios between the participants. We describe the contents of such a protocol using a choreography. This is modeled in the concurrency viewpoint of a software architecture using a modified version of the BPMN Choreography notation.

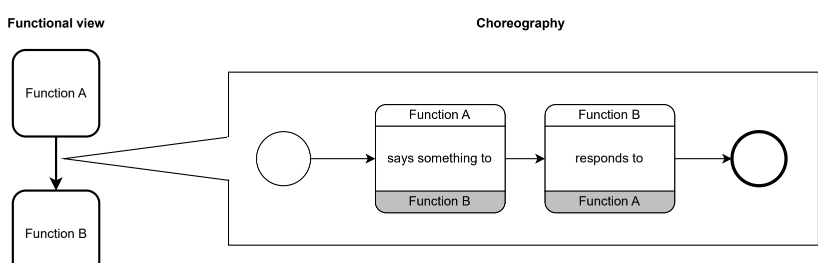



Figure 1.1: Relation between the interaction model and choreography.

The framework for working with protocols and their relation to the interaction model is a result of a thesis [2]. In this research the framework is taken as a basis, and extended upon. At the moment INORA has a very basic implementation written as a plugin for the Eclipse modeler, using the theory as explained in [2]. The theory describes in-depth all the aspects of the interaction mode and the choreographies that are proposed and suggests that it is possible to perform model-checking in INORA using Petri nets. This would result in not only a notation for architects to depict the functional system and how each component communicates but also provide an extensive and formal analysis of the behavior of the interactions, resulting in useful feedback on the to-be system.

[2] only scratches the surface of the translation, possibilities for analysis, and a way to implement this in a tool. This research aims to fill these shortcomings by formalizing and implementing solutions to the

analysis problems. The resulting tool is an addition to the software architecture field, as it provides the ability to model and check INORA models.

## 1.2 Problem Statement

For any system modeled with INORA this thesis aims to give feedback on the quality of the models, and therefore the system. Although notation and an initial semantics of INORA are provided in [2], the current proof-of-concept tool only supports simple modeling capabilities. In this thesis, we want to extend the tool to define formal semantics to be able to provide feedback on the quality of the model.

At this moment, literature shows that there is no method to automatically check the quality of protocols in INORA. The protocols in INORA are modeled with a modified and more strict version of the BPMN Choreography. Literature shows it is possible to translate such models to more formal models, such as Petri nets. The papers [2] and [3] already define a mapping for translating a single BPMN Choreography model to a Petri net. Petri nets have properties that can be checked for to infer the correctness of a model. However, current INORA choreographies may have references to other protocols, as a system is often a composition of multiple components that work together. [4] states that if all components are considered correct, the composition will also be correct. This is, however, an assumption that does not always hold. INORA currently lacks the proper analysis tools to verify whether a composition of components, communicating via protocols, is correct. The major problem is, therefore, the lack of methods and tools to guarantee correctness of INORA models.

**Problem statement**
This research aims to provide formal semantics and tool support in order to give architects feedback to create sound INORA models, such that the quality of software architecture is improved.

When a formal translation is proposed it needs to be implemented in a tool itself to provide automation. The current INORA tool has no support for translation and verification. The translation needs to support mapping to pseudo-code so that it can be implemented. Through the tool constructed Petri nets need to be checked for certain properties and the quality of the model needs to be determined and feedback must be given back to the user. The result of this thesis is a formal translation method and verification method, implemented in an updated tool: INORA2.

## 1.3 Research Questions

The problem statement raises one main goal for this thesis. The goal is to give software architects automated formal feedback on software design. For this goal to be achieved, a (set of) tool(s) needs to be created which can be used to model both the interaction model and choreographies for any software system, run analysis, and display feedback.

To solve the problem at hand, a research question needs to be stated. It is divided into five sub-questions which will be answered in the same order throughout the thesis.

*RQ: How can software architects best be provided with automated feedback on the quality of an Interaction Oriented Software Architecture?*

To answer this main research question a couple of sub-questions need to be answered.

$SQ_1$ *How is a formal translation of a composed choreography to Petri nets defined, so that allows for verification?*

A method needs to be defined that can make the translation from a (composed) choreography to a formal Petri net. This translation needs to be non-redundant, well defined, should not introduce livelocks and deadlocks , and should be easily implemented with any programming language.

$\boxed{SQ_2}$ *What techniques can be used to check for the soundness of a Petri net?*

Petri nets can be tested on common properties such as deadlocks, livelocks, and soundness. How does a combination of these imply "correctness", or more formally "soundness" and which methods can be used to check this?

$\boxed{SQ_3}$ *How can the formal translation method and the verification techniques best be implemented in a tool so that it returns feedback?*

A tool is required to make use of the translation proposed by $SQ_1$. Furthermore, it will use verification techniques to generate feedback on the model. How will this tool work and what functions will it need to support to make optimal use of the findings of $SQ_1$ and $SQ_2$?

## 1.4 Research Method

The three sub-research-questions stated in section 1.3 are answered following a formal research method. The *Design Cycle* as described by Wieringa [5] is used as a handlebar, as all sub-questions can be categorized by each phase in the cycle. Note that the design cycle is used instead of the engineering cycle as there will be no *Treatment implementation*. The term "implementation" is used extensively throughout the thesis, yet our definition differs from the one given by Wieringa. Wieringa defines a *Treatment implementation* as using the validated and evaluated *artifact* in its original social problem context [5]. The term in the context of this thesis, however, is defined as an algorithm and (a set of) tools that translate the theoretical framework to a digital prototype. This discrepancy in terminology is also extensively mentioned in Wieringa's book.

The design cycle consists of three phases: (i) the *problem investigation* (ii) the *treatment design* and (iii) the *treatment validation*. As the name implies, these can be performed in an cyclic manner. In the time span of this thesis there is only time to perform on iteration of the cycle. The sub-questions can be divided into the three phases with the following distribution:

| | SQ1 | SQ2 | SQ3 |
|---|---|---|---|
| (Phase 1) Problem Investigation | | ✓ | |
| (Phase 2) Treatment design | ✓ | | ✓ |
| (Phase 3) Treatment validation | | | ✓ |

In the first phase *Problem investigation* the gap in the literature and the reason for this thesis are discussed. It also uses the existing literature to answer $SQ_2$ as Petri net verification and feedback are covered in a wide range of research. The second phase *Treatment design* solves the problems found in the literature by defining a formal translation and developing a (set of) tool(s) to implement the translation and automated feedback. This answers $SQ_1$ and $SQ_3$. Finally, the third phase *Treatment validation* will bring see whether the design is feasible by creating a tool implementation.

After answering each sub-question the main research question will be answered: *how* software architects can best be provided with *feedback*, and what this feedback consists of.

## 1.5 Running Example

Although the concepts in the thesis can be quite abstract, the goal is to eventually apply them in the software architecture field. Therefore examples and theorems that are found will be brought back into a generic context. This will be done with a running example, which is described below. The running example is a fictional and simplified system that could, for example, be found in a check-in travel environment.

To service a public transportation company (PTC), a check-in-like system is used. It consists of four different types of devices:

- **PTC System** - This is the main server for authorizing travels, storing trips, and keeping track of balance.
- **Trip Terminal** - This is a terminal from which a traveler can check in and check out with a badge.
- **Service Terminal** - This provides service functionalities such as viewing past trips and current balance. For simplification reasons, the function to deposit money onto the badge has not been added to the FV.
- **Employee Terminal** - This terminal provides employees from PTC to check whether a badge is checked in and show the trip information.

A usual interaction would be a passenger (referred to as "user") that wants to travel by train. The user checks in with his/her card at the **Trip Terminal**, which sends an authorization request to the **PTC system**, checks for sufficient balance, and sets a non-finished trip. Once the user checks out at his/her final location, the **Trip Terminal** finds the latest trip, marks it finished by setting the final destination and deducts money from the credit. Another common interaction would be the railway personnel checking whether a passenger has checked in. Such an interaction would start by scanning the card from the **Employee Terminal**, which asks the **PTC system** to check whether the given card has an active trip. This will result in an answer displayed on the screen of the **Employee Terminal**.

## 1.6 Thesis roadmap

The concepts in this thesis are related to each other in the domain of Interaction-Oriented Sofware Architecture. As a guide, this section provides a textual roadmap and a visual representation of the relations between the concepts.

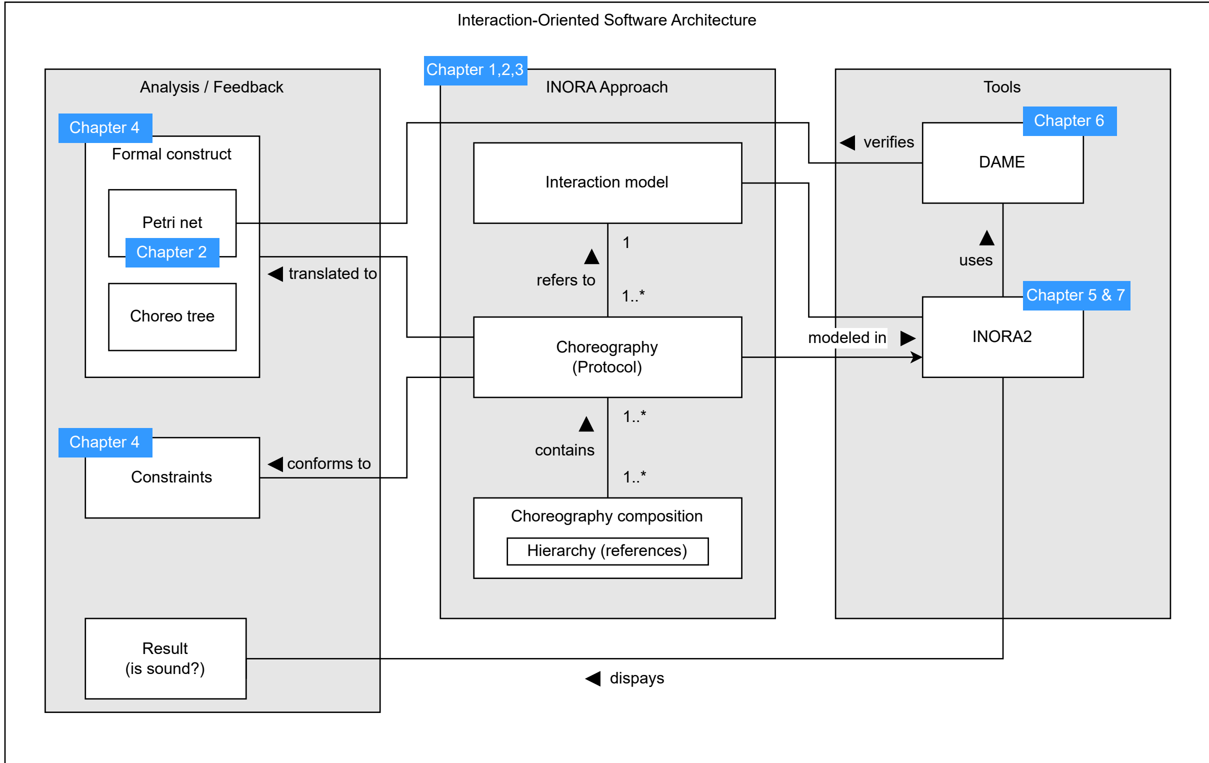



Figure 1.2: Conceptual overview of concepts in this thesis.

In **Chapter 1** we cover the basic introduction to the topic and basic concepts. Furthermore, the problem is identified and research questions are formed. We also describe the research method and provide a

brief description of the running example. In **Chapter 2** the concepts used in the research are covered extensively. It provides formal definitions for software architecture and its viewpoints, explains the crucial concept of choreographies and their compositions, discusses possible modeling languages described in literature, and introduces the interaction model. Another major topic discussed in this chapter is Petri nets, which is a crucial concept in this thesis. If gives formal definitions, conditions, assumptions, and properties of Petri net constructs. **Chapter 3** puts a focus on related work concerning BPMN to Petri net translation, (tool-based) verification, and feedback visualization.

The following chapters are concrete solutions to the problems stated in the introduction. **Chapter 4** covers all the aspects of the needed formalization. It provides strict constraints to choreographies and describes the procedures to generate a Petri net and (conditionally) a computationally cheaper choreography tree. **Chapter 5** then follows this theory up by describing the implementation called INORA2. It describes the application, and shows how the procedures described in chapter 4 are implemented and their results are combined to derive an analysis outcome. **Chapter 6** introduces another tool called DAME, which describes the support software for Petri net validation called LoLA 2.0 and INORA2 interactions with it. **Chapter 7** shows a walk-through for INORA2. A common interaction between INORA2 and end-users is described and depicted in this chapter.

Lastly, **Chapter 8** draws the conclusions, answers the research questions, and discusses the research. Additionally, future work is provided.

The appendices cover: **(A)** the meta-model for INORA models **(B)** choreography tree generation algorithm **(C)** Petri net generation algorithm **(D)** Petri net translation segments **(E)** extensive description of the DAME tool.

# Chapter 2

# Basic Notions

This chapter introduces the basic concepts surrounding software architecture and introduces crucial definitions and the context.

## 2.1 Software Architecture & Views

The software architecture of a system describes software elements, (externally visible) properties of those elements and their relationships [6, 1]. It can be used to model out a system and is useful in design, development, and during extension and maintenance as its purpose is to help understand the system [7]. As it is impossible to capture the essence detail of a system architecture in a single model, the system is considered in terms of multiple viewpoints. A viewpoint is collection of patterns, templates and conventions to create a view [2]. Each viewpoint has its own purpose and can be used by certain stakeholders in the context [7, 8]. Together all viewpoints give a holistic view of the system. Although [7] introduces seven different viewpoints, the focus within this thesis is on the functional and concurrency viewpoints.

The functional viewpoint (FV) describes the functional elements and their interfaces and primary interactions within the system. The shapes of other viewpoints highly depend on the FV and it is, therefore, one of the most important viewpoints [7]. The functional viewpoint contains components that may contain other components and/or functions. Functions are coupled by dependencies (within the same parent component) or protocols (between functions in different components). In line with the title of this thesis, the aim of Interaction Oriented Architecture is to model the interactions between these components.

The concurrency viewpoint maps functional elements to concurrency units and shows its coordination. 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 [7]. These interactions are captured in the concurrency viewpoint models, which may exist out of user scenarios and choreographies. These will be extensively covered in section 2.2.

According to [7], several pitfalls exist when modeling in the concurrency viewpoint. These should be avoided to achieve a high-quality model. Examples of these pitfalls are (i) excessive complexity (ii) process deadlocks (iii) process livelocks (iv) race conditions. The concurrency viewpoint should not cover very detailed interactions (e.g. a full TCP handshake should not be modeled), but rather give a high-level overview of the interactions between components in the system.

The aim of this thesis, therefore, is to automatically test whether these concurrency models are free of semantic errors.

## 2.2 Choreographies & User Scenarios

The concept of a *choreography* is already mentioned in section 2.1. According to [7], choreographies are modeled in the concurrency viewpoint (Definition 2.1) of a software architecture. This section will elaborate further on the concept.

> **Definition 2.1**: Concurrency viewpoint [7]
>
> 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.

Thus, the concurrency viewpoint maps the components defined in the functional viewpoint to a view that shows the interactions between those components. In contrast to historic system designs, almost any system nowadays makes use of concurrency. Systems are event-driven and need to react to "things" happening on the system. The concurrency viewpoint typically consists of three parts [7] (i) a process model describing the inter-process (components) communication structure (ii) a state model describing the possible state for each component during run-time and the possible transitions and (iii) a number of analysis techniques to confirm that the model is sound. In this context, a system is sound when it has the possibility to terminate [9].

As software components commonly work concurrently and often use message-based communication, asynchronous communication is implied [10]. Asynchrony, however, introduces a certain complexity [11] which is inherent to the concurrency viewpoint. This asynchronous property of these models often makes it difficult for humans to quickly spot and correct errors. Deadlocks, for example, usually arise as the final state of a complex sequence of operations on jobs flowing concurrently through the system and are thus generally difficult to predict [12].

The models in the concurrency viewpoint can be modeled in a wide variety of languages. For process modeling, architects could use an extended version of UML, formal concurrency modeling languages such as LOTOS [13], CSP [14] and CCS [15], or an informal model created by the architect itself. For state models, an architect could look into graphical notations such as UML, Petri nets [16] and SDL or non-graphical notations such as the Finite State Process language [17].

Several modeling techniques exist and they can be indexed on the concurrency concerns listed by [7]. Such an indexation is made by [2] and is found in table 2.1. This indexation gives an overview to help decide which techniques can be used for modeling interaction between components. In [2] is stated, however, that "All interaction-modeling methods (..) have in common that they cannot express the communication of a set of models. Therefore it is not possible to show consistency between a set of choreographies.".

| | Async. communication | Visual notation | Formality | Compositional design | Tool support | State- vs. Action-based | Model 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 | × | ✓ |

Table 2.1: A comparison of possible techniques to model the concurrency viewpoint. The ✓and × show whether they address the concern. In the "Formality" column, the *F* and *S* translate to *Formal* and *Semi-form* respectively. In the column "State- vs Action-based" the *A* and *S* translate to *Action* and *State* respectively [2].

To address most aspects and still keep a clear and simple model, INORA is proposed in [2]. INORA models the components and their functions with a visual notation in the functional viewpoint (named the Interaction Model) and uses a combination of two techniques for modeling the concurrency viewpoint (named the Protocol definition). The interaction model is a theoretical framework that maps containers, functions, and protocols between them. A meta-model of INORA can be found in Appendix A. It is extensively covered in Klijs' thesis and although it is a crucial part of INORA and forms the basis for choreographies, the full explanation is left out of the scope of this thesis.

A major component of INORA are the choreography i.e. protocol definitions. Table 2.1 shows that both Business Process Model and Notation (BPMN) and Petri nets [16] comply with most concerns. The major differences between the two are Formality and their State- vs. Action-base. As these two can complement each other, both can be used in modeling concurrency. Previous studies [2, 18, 3] propose that non-formal visual notations of processes such as BPMN can be translated to a formal notation such as a Petri net.

The Business Process Model and Notation (BPMN) is a standard notation for capturing business processes, especially at the level of domain analysis and high-level systems design. The notation inherits and combines elements from a number of previously proposed notations for business process modeling, including the XML Process Definition Language (XPDL) and the Activity Diagrams component of the Unified Modelling Notation (UML) [18, 19, 20]. Such models are often created in the early phases of system development. Making sure such a model is correct avoids the pitfalls listed in [7], will result in a better system, and avoids hard and costly corrections [18].

A creator of a choreography models all user scenarios (possible actions of a user and their consequential indirect actions within the system) in BPMN Choreography notation which has a visual representation, is semi-formal, and is action based. Such a semi-formal notation can then be translated to a formal one: a Petri net [3]. This yields many advantages, which will be discussed in section 2.5. An example with an explanation of a BPMN notation of the choreography is given in Figure 2.1.

Another formal language that a Petri net can be translated to, is Pi-Calculus. This mostly addresses the same concerns as Petri nets and is a mature language. A study [21] names three advantages to use Petri nets over Pi-Calculus:

1. Despite their graphical nature, Petri nets have strict and formal semantics. This makes it a solid basis, without ambiguities that may exist in a process algebra such as Pi-Calculus.

2. Petri nets are state-based rather than action-based (event-based). This means that states are noted explicitly and are not inferred and are often suppressed as can be the case with process algebras. This may seem a subtle difference, yet this is of the utmost importance for workflow modeling.

3. There is an abundance of analysis techniques for Petri nets. Petri nets can therefore be checked on correctness using one or more of the many methods available.

Another advantage that can be spotted in table 2.1 is the availability of tools to work with Petri net. Although both techniques have support tools, Petri nets are covered by wider assortment of tools, which is a clear advantage as this thesis aims to design a new tool that uses Petri nets. [21] also lists some disadvantages for using Petri nets. It might be more complicated to implement three types of patterns: (i) *multiple instance patterns*, the burden of joining and splitting is carried by the modeler (ii) *advanced synchronization patterns*, because the firing rule only supports two types of joins: the AND-join (transition) or the XOR-join (place) and (iii) *cancellation pattern*, as the firing of a transition is always local there can be no global token adjustments resulting from an error for example. These disadvantages, however, are subtle and not especially relevant for the problem in this thesis, and therefore do not outweigh the advantages.

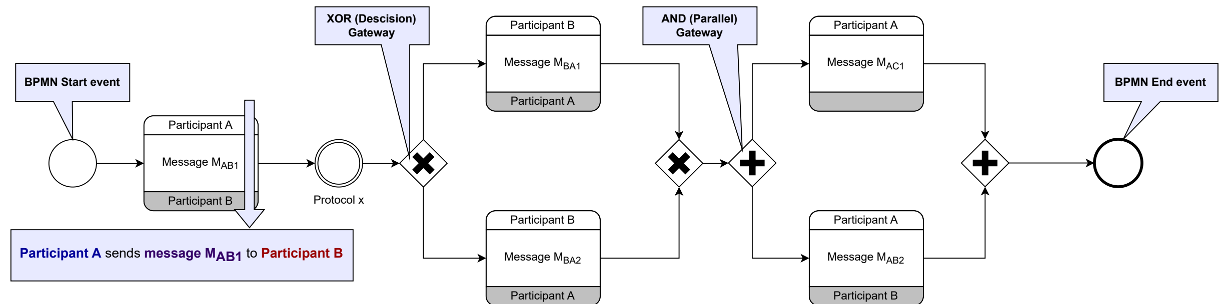



Figure 2.1: Example BPMN Choreography, adapted from [2].

Choreographies can thus be translated to Petri nets but are initially modelled in BPMN Choreographies. To adapt the default BPMN 2.0 Choreography notation to display interaction between software components (protocol definition), [2] proposes three changes

1. A message $M_x$ has a sender, represented at the top of an activity, and a receiver which is represented at the bottom. The default BPMN uses an envelope icon to depict messages, whereas [2] has chosen to name the "activity" itself with the message.

2. The INORA choreography only supports AND-gateways and XOR-gateways as opposed to supporting all possible BPMN gateways.

3. Intermediate events are used to reference other protocols. These can be read as "substitute full protocol $x$ here", but for the readability and re-usability, they are displayed by a single node.

With these changes, a strictly limited semi-formal method of describing protocols (interactions) within the concurrency viewpoint is given. This can then be used to formalize and verify the protocol.

## 2.3 Interaction Model

Choreographies thus represent all possible scenarios of interaction between components. These interactions, however, should somehow be placed in an overview of the system to see how they relate and when they are executed. To map these choreographies to a context view, we use the functional viewpoint. Arcs between functions are directly linked to choreographies. We call this view the *Interaction Model*.

This system in the running example can be modeled in this way. Figure 2.2 shows all (simplified) functionalities of PTC, and their communication channels. These channels are labeled, and for each of these arcs we can create an INORA protocol. The legend defines the four distinct types of elements in INORA. Contrary to [2], in this thesis there is no semantic difference between internal and external protocols. Originally, it was assumed that interactions between components could be *asynchronous*, and interactions within components are *synchronous*. We drop this assumption, as it might not always apply

in real-life systems, and this assumption has no effect once we start checking on composed choreographies.

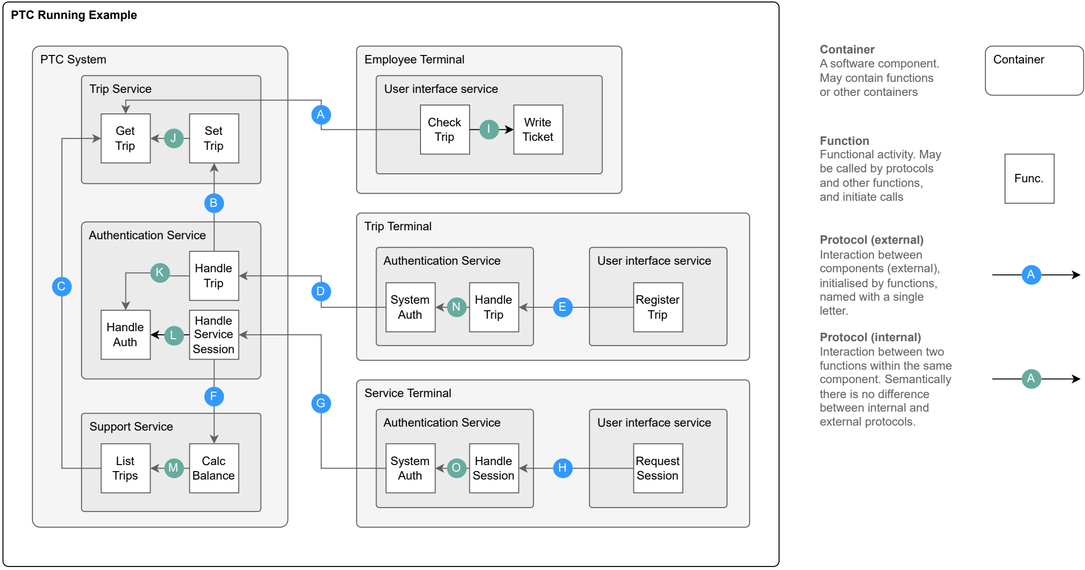



Figure 2.2: Running example for this thesis: PTC.

Figure 2.3 displays the INORA representation of the choreography in protocol D. This is an intermediate protocol as it can never be initialized without protocol E. The start event is triggered by the "System Auth" function in the Trip Terminal's Authentication Service. It also references protocol B and uses the "Handle Auth" function in the PTC System's Authentication Service. Note that this is a simplified version for demonstration purposes. It is up to the modeler to decide the level of granularity of the model. The modeler could decide to model all possible error messages, or simply return an "generic error" as the interaction of handling, receiving, and displaying an error might be the same for a set of different errors.

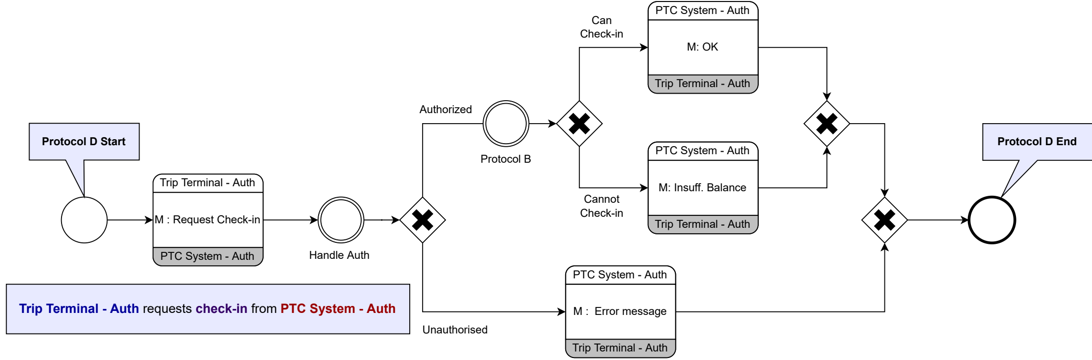



Figure 2.3: INORA Choreography of protocol D from running example.

## 2.4 Composed Choreographies

In the previous sections, we discussed choreographies and their relation to the interaction model. In a choreography, it is possible to reference one or more protocols, which themselves can then reference protocols, etc. This recursive tree of referencing is called a composed choreography. Figure 2.4 shows an example of such a reference and its substitution: Protocol D is referring to protocol B.

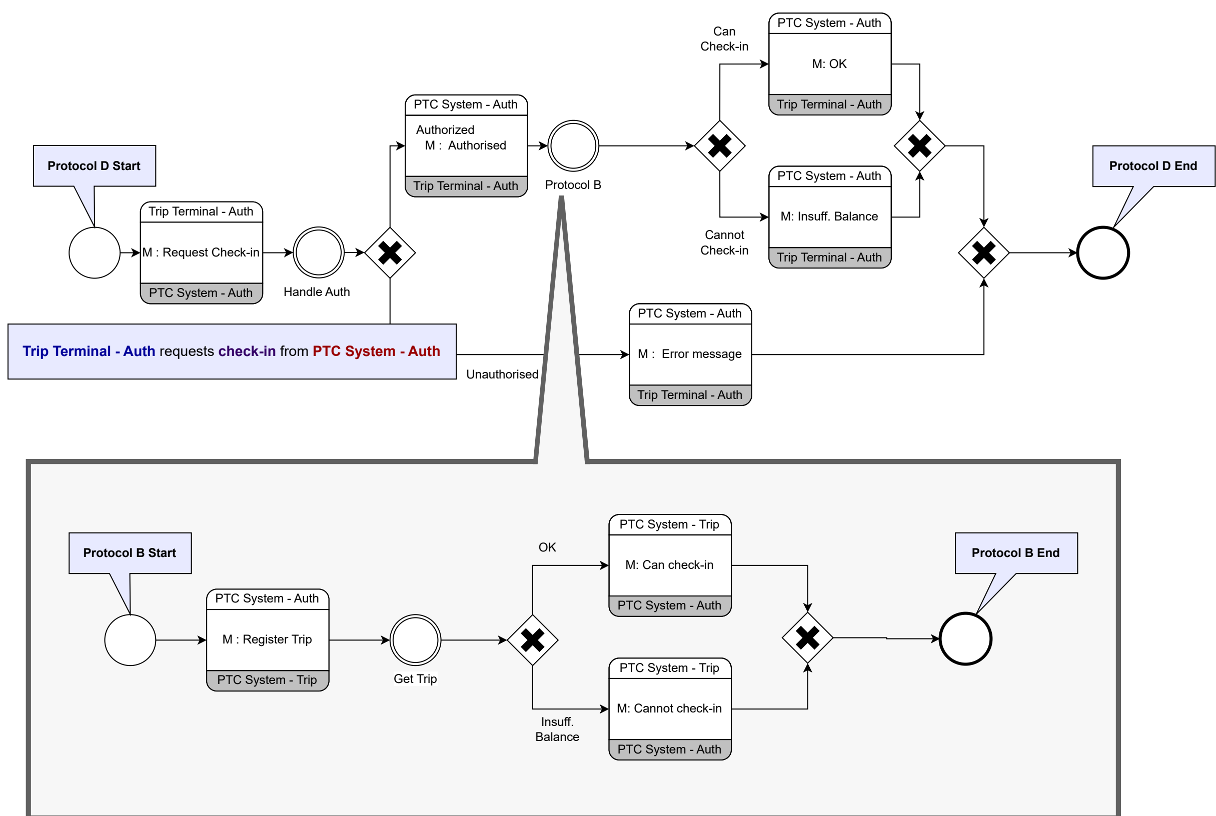



Figure 2.4: Substituting reference to Protocol B with its literal choreography to construct a "composed structure".

The behavior and notation of composed choreographies i.e. *compositions* are defined as followed.

**Definition 2.2**: Protocol composition

Let the set of abstract choreographies $C$ be derived from an interaction model $M_i$.

The set $C$ contains five protocols: $C = \{A, B, C, D, E\}$.

Now assume that $(A \text{ refs } B), (B \text{ refs } C), (B \text{ refs } D)$. Protocol $E$ is not referred to or referred by other protocols.

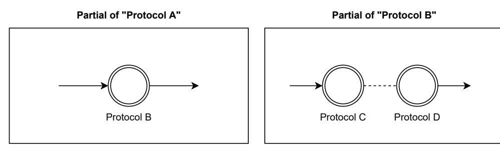
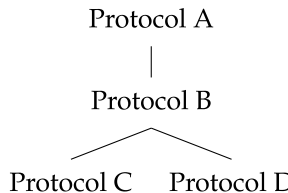



Figure 2.5: Example protocols: protocol A refers protocol B, and protocol B in its turn refers to protocol C and protocol D.

The reference tree looks like:




Starting from the root of the tree, we call the *composition of* $A$ $\{A, B, C, D\} \in A^c$. The hierarchy based on edges is denoted as: $r(A^c) = \{(A \rightarrow B), (B \rightarrow C), (B \rightarrow D)\}$. Note that $E$ is not included in the composition of $A$ as it is independent in regards to $A$.

Additional conditions are given to refine the composition. Assume a simplified abstract interaction model as seen in Figure 2.6

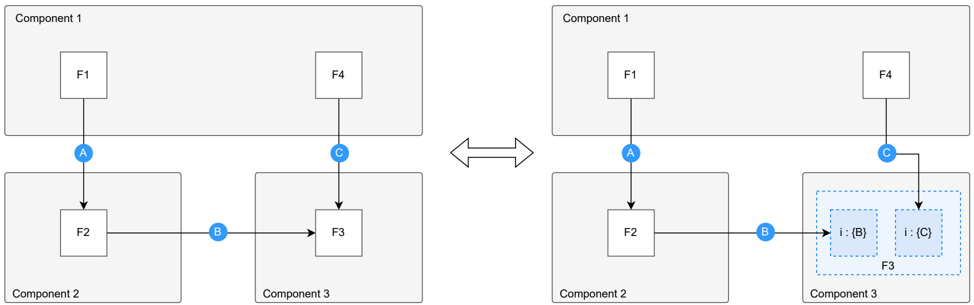



Figure 2.6: Abstract example of functions communicating by protocols: $A$, $B$ and $C$.

1. **Compositions of protocols are trees.** In this thesis the condition is that there is *no* internal state machine which needs to be considered for each function. In Figure 2.6 function $F3$ is used by both protocol $B$ and $C$, but due to this general assumption we assume $B$ is using an instance of $F3$ called $i : F3_B$ and $C$ is using an instance of $F3$ called $i : F3_C$. This means that it would not be possible to end up in a dead- or livelock between the use of $B$ and $C$ in $F3$. The scope of this research covers the interaction between components and not their inner workings.

2. **A composition of a sound protocol itself is sound *iff* all referenced protocols are sound.** The composition of a protocol $P$ is called $P^c$ and is composed by replacing all references by their protocol nets recursively.

   For example, in Figure 2.6 a sound protocol $A$ references ("uses") another sound protocol $B$, and the composition of A ($A^c$) is sound. If protocol $A$ were to reference a non-sound protocol $!B$, the composition of A ($A^c$) would not be sound.

## 2.5 Petri Nets

As stated in the previous section, formalizing the choreography can be done using Petri nets. The formal definition of a Petri net by [16] is given in definition 2.3.

**Definition 2.3**: Petri net, adapted from [22]

A Petri net is a tuple, $N = (P, T, F, W)$ where:
- $P = \{p_1, p_2, ..., p_m\}$ is a finite set of places
- $T = \{t_1, t_2, ..., t_n\}$ is a finite set of transitions, such that $P \cap T = \oslash$
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs
- $W : F \rightarrow \mathbb{N}^+$ is a weight function

Note: when $W$ is not explicitly defined, the weights of all weight of for all arcs will be set to 1. $(\forall f \in F : W(f) = 1)$

A marking of $N$ is a function $M : P \rightarrow \mathbb{N}$, where $m(p)$ denotes the number of tokens in place $p \in P$. If $m(p) > 0$, place $p$ is marked. Given a Petri net $N$ with marking $m$, the pair $(N, m)$ is called a marked Petri net. The initial marking is described by $(N, M_0)$.

A Petri net is an abstract and formal model of information flow. The properties, concepts, and techniques of Petri nets are being developed in a search for natural, simple, and powerful methods for describing and analyzing the flow of information and control in systems, particularly systems that may exhibit asynchronous and concurrent activities [23]. It addresses all concerns mentioned in [7] and has a couple of mathematical properties that can be exploited to quickly verify a net. In this thesis, Petri nets are depicted as directed graphs, which is the standard for describing Petri nets. Places are represented as circles, transitions as squares, arcs as arrows, and the marking as black dots within places. These black dots are also called "tokens". Any place holds at any time $0..n$ tokens [16]. An example Petri net is given in Figure 2.7.

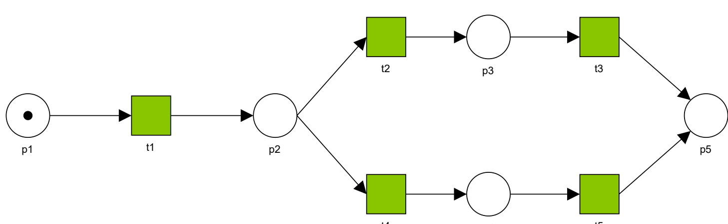



Figure 2.7: The graphical representation of a basic Petri net. Adapted example from [2].

The mathematical notation corresponding to this net is given in figure 2.1. Note that this shows a useful resemblance with storing objects in arrays, a property that is used to actually store Petri nets in computer code.

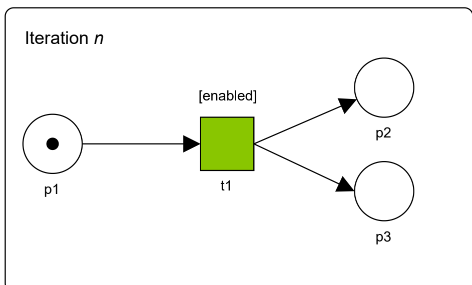
$P = \{p1, p2, p3, p4, p5\}$
$T = \{t1, t2, t3, t4, t5\}$
$F = [(p1, t1), (t1, p2), (p2, t2), (p2, t4), (t2, p3), (p3, t3), (t3, p5), (t4, p4),$
$(p4, t5), (t5, p5)]$
$W = \oslash$
$M_0 = [p1]$

Any transition $t \in T$ has a preset and a postset. The definition is given in definition 2.4.

**Definition 2.4**: Presets and postsets, adapted from [2]

Each transition in a net $t \in T$ has a preset $\bullet t = \{p \in P \mid (p, t) \in F\}$ and postset $t\bullet = \{p \in P \mid (t, p) \in F\}$.

A basic concept of a Petri net is to check whether a certain transition is *enabled* [23]. A certain transition can only move tokens if it is enabled. A transition $t \in T$ is enabled in $(N, m)$, denoted by $(N, m)[t\rangle$, iff $W((p, t)) \leq m(p)$ for all its input places $\forall p \in \bullet t$. An enabled transition can *fire*, which will result in a marking $m'$ iff $m'(p) + W((p, t)) = m(p) + W((t, p))$, for all $p \in P$ and such a transition is denoted by $(N, m)[t\rangle(N, m')$. In this research we assume the default weight of 1 for each arc. A firing thus will consume the tokens and produce one in each of the output places. Therefore each place in $\bullet t_0$ needs to have *at least* one token to be enabled. An example is given in figure 2.8.



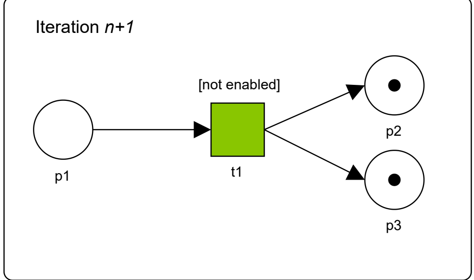




Figure 2.8: Two states of the same Petri net with iteration $n$ having $t1$ enabled, and iteration $n + 1$ being the result after the firing of $t1$.

This also implies that all transitions in a Petri net follow *interleaving semantics* rather than *true concurrency* [24]. This means that when two transitions $A$ and $B$ are fired "simultaneously" they either follow sequence $A \rightarrow B$ or $B \rightarrow A$ to get to the transitioned state, whereas true concurrency may fire them both concurrently. An example can be seen in figure 2.9. As the verification tools that are used in this thesis use interleaving semantics, all examples and logic will too.

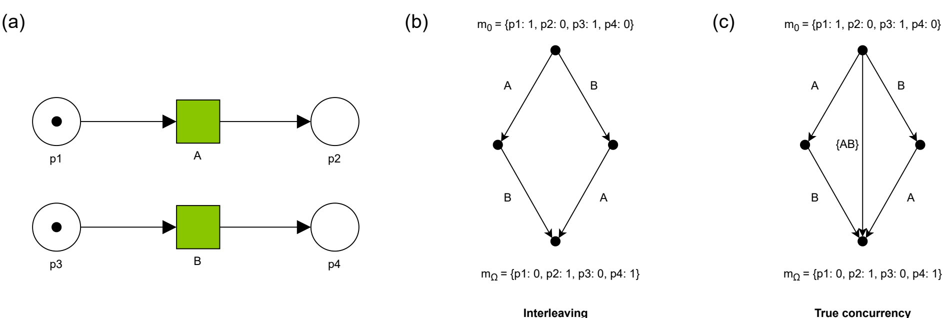



Figure 2.9: Differences between interleaving semantics (b) and true concurrency (c) depicted as possible transitions for a Petri net (a).

There are two stricter types of Petri nets that are relevant in the context of this thesis: state machines and marked graphs. The official definition is given in definition 2.5

**Definition 2.5**: State machine (S-Net), marked graph (T-net), adapted from [25]

A Petri net is a *state machine* (S-net) if $\bullet t \leq 1$ and $t\bullet \leq 1$ for $\forall t \in T$. A

Petri net is a *marked graph* (T-net) if $\bullet p \leq 1$ and $p\bullet \leq 1$ for $\forall p \in P$.

In this thesis, we make use of a certain type of Petri net to model interactions: an *Open net*. An open Petri net is defined in 2.6.

**Definition 2.6**: Open Petri net

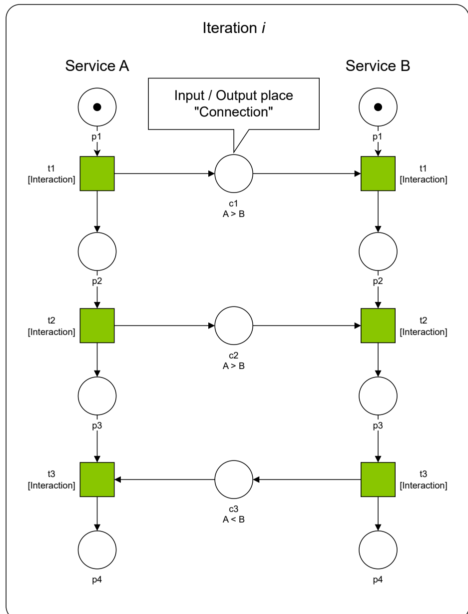
An open Petri net is a Petri net with certain places designated as inputs and outputs via a cospan of sets. We can compose open Petri nets by gluing the outputs of one to the inputs of another [26].

Formally, it is defined as, $N = (P, I, O, T, F, W)$ where:
- $T, F, W$ are defined as in Definition 2.3
- $P \cup I \cup O$ is a union of all intermediate, input and output places, with $(P, T, F, W)$ called the internal net and $I \cup O$ called its interface.

The following conditions and implications hold:

$$\begin{aligned}
\forall i \in I &: \bullet i = \varnothing \\
\forall o \in O &: o\bullet = \varnothing \\
\forall t \in T &: \bullet t \cap I \neq \varnothing \Rightarrow t\bullet \cap O = \varnothing \wedge t\bullet \cap O \neq \varnothing \Rightarrow \bullet t \cap I = \varnothing
\end{aligned}$$

The properties of an Open Petri net can be used to synchronize with another participant net. The papers [27] [28] describe how these open nets and their connection places are used to describe interactions of two asynchronous services. For example, two (identical) nets can interact by consuming and producing tokens in the interface, as can be seen in Figure 2.10.



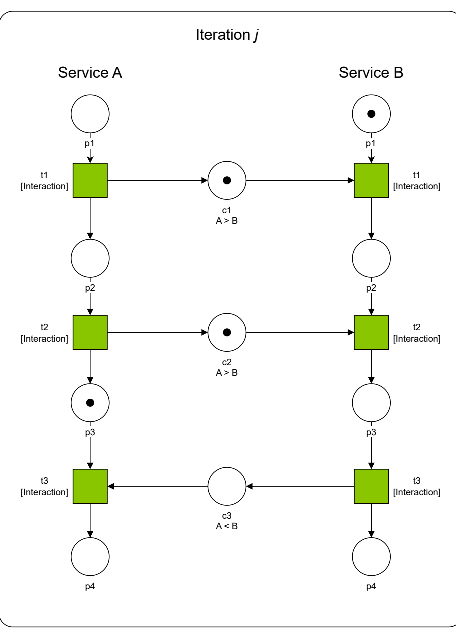




Figure 2.10: Two partners can asynchronously act when communicating with each other. In this case Service A produces down its tokens before Service B does anything at all, which is allowed behavior. They are "synced" once A needs an answer from B.

More formally, the composition of two Petri nets using the interface is defined as

**Definition 2.7**: Composition of Open nets, adapted from [29]

Let $N = (P, I, O, T, F, W)$ and $N' = (P', I', O', T', F', W')$ be two open nets with $P \cap P' = \varnothing$, $T \cap T' = \varnothing$, $I \cap I' = \varnothing$ and $O \cap O' = \varnothing$, the *composition* $N \bigoplus N'$ is the open net $(P \cup P', T \cup T', F^{\bigoplus}, (I \cup I') \setminus (O \cup O'), (O \cup O') \setminus (I \cup I'))$, where $F^{\bigoplus}(x, y) = F(x, y)$ if $(x, y) \in (P \times T) \cup (T \times P)$, $F'(x, y)$ if $(x, y) \in (P' \times T') \cup (T' \times P')$, and 0 otherwise.

This interaction-oriented approach of syncing a multiplicity of open nets using an interface is the basis for constructing participant nets. Each participant in a choreography can be seen as a service, and they communicate via interface places. Both nets are identical and "glued" together to model the interaction between components. The following chapters will deep-dive into this topic.

## Petri Net Properties

A Petri net is a formal modeling technique that holds several useful properties. It is possible to perform certain checks on a net to see whether the properties hold and conclude if the net is of high quality. Properties of Petri nets are extensively researched in literature since their introduction in 1977 [23].

To analyze Petri nets a reachability graph needs to be constructed. A reachability graph is a directed graph $G = (V, E)$ with $V$ being a class of distinct reachable markings from $m_0$ and $E$ being a set of directed arcs between two markings in $V$. Any enabled transition can change the marking of the Petri net, and therefore the reachability graph ensures that all possible markings are known and it is also known how which firing sequences can lead to a certain marking. The reachability graph is also known as the *State Space*. It starts with the given initial marking $m_0$ and flows to all possible states. There may be infinite states, as the tokens in a certain place can be any natural number $n \in \mathbb{N}^+$. This is solved by writing the increase and decrease of tokens as $wn$ where $w$ stands for the weight of the arc. An example (with weight $w = 1$) is given in figure 2.11.

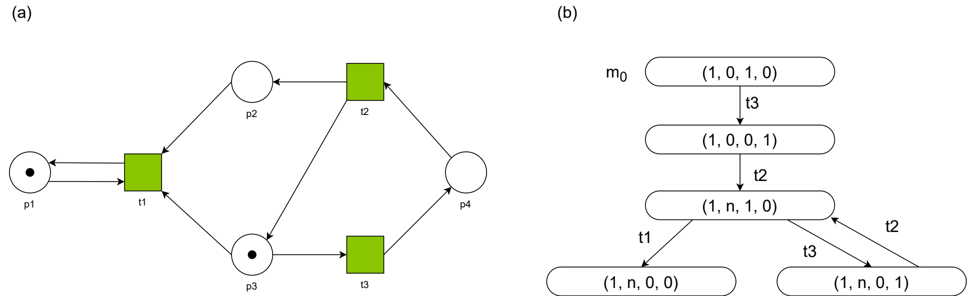



Figure 2.11: An example Petri net (a) and its reachability graph (b), adapted from [30].

Using the state space, it is possible to analyze a Petri net on the following properties [30]:

- **Safeness and Boundedness:** A place in a Petri net is called *k-bound* if it does not contain more than $k$ tokens in all reachable markings from $M_0$ (called $V$ in the reachability graph): $\forall m \in V : m(p) \leq k$. This guarantees an upper bound of the in-process parts [31]. The absence of upper bounds at a place may indicate a weakness in the design and should be avoided. A Petri net is *bounded (k-safe)* if all places in the net are *k-safe*. As in this thesis the assumption is made that $k = 1$, a Petri net can simply be called *safe*. Each marking is stored in the state space and therefore safeness can easily be checked.
- **Liveness:** A deadlock exists if there is a marking reachable from $m_0$ where no transitions can be fired. It is possible to check for deadlocks (and thus whether a Petri net is "live") using the state space. This is done by looking for terminal nodes in the state space: a node without a successor. This is called a *terminal node*. Although liveness and deadlocks are not the same, [32] states that

as long as the controlled-siphon property [33, 34] holds, they are equivalent and therefore that liveness implies an absence of *deadlocks*.

- **Free of livelocks:** An computational heavier problem is finding the livelocks in a Petri net. Livelocks are defined as "unintended cyclic terminal strong components of the state space" [35]. According to [36] a Petri net is free of livelocks if the state space and its strongly connected component graph (SCC graph) [37] are isomorphic and contain no self-loops or "if the state space contains self-loops or if there exists at least one strongly connected component that consists of more than one node (i.e., the number of nodes in the SCC graph is less than the ones in the state space), then we need to check if all terminal components are trivial (i.e., consist of a single node and no arcs)" [36].

- **Soundness:** In literature [38, 39] the soundness of a process is defined as: "A process is sound if for each state that can be reached from the initial state, a firing sequence exists that leads the system to the final state.". Substituted INORA choreographies (or composed systems) can be viewed as a *service tree* [2]. A service tree is a composition of components that is non-cyclical; the children provide services to the parents [9]. A composed system is sound of each of the components is sound.

  A sound system has three properties [39]:

  - **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 $\Omega$.
  - **Proper completion** When the final marking is reached, there should be no tokens left in the rest of the model. This is followed from **weak termination** meaning that from every reachable state of a system, some final state can be reached [40].
  - **No dead transitions** No transitions in the net should be dead. A transition is dead if the transition is never enabled in any reachable marking from $m_0$.

The analysis techniques mentioned in this research all use the reachability graph/state space of a Petri net. A major aspect that should not go unnoticed is the state space explosion problem. It entails that the number of states of even a relatively small net is often far greater than can be handled in a realistic computer [41]. Therefore reduction techniques need to be applied to tackle this problem. Tools that are mentioned in the next section make use of these reduction techniques. Their inner workings will not be explained here, as it is outside of the scope of this thesis.

# Chapter 3

# Related Work

There exists a wide range of related literature in the scope of the general topic of the thesis. It is, however, necessary to narrow down the scope in a way that it connects to the design cycle and puts focus on the research questions. Answering the research questions will both solve the problem introduced and guide a logical flow through the thesis.

## 3.1 Introduction

The chapter will cover related work belonging to the scope of each subquestion, and the problem in general. A major part of the thesis will be refining and formalizing the translation of an INORA Choreography to a Petri net. Translations of such semi-formal notations to Petri nets have already been researched, this section will review this literature and highlight their shortcomings. Papers covering the verification and general properties of Petri nets, as well as how to display the outcome as feedback are discussed. Finally, best practices from the literature for building a set of tools and consequentially evaluating it in an empirical study are covered.

As there are five concrete sub-questions to be answered in this thesis there is a rather narrow scope for each of the sub-topics. There is no formal literature review method used in this thesis, as there is no large set is papers that need to be indexed. Each search for literature starts off by using keywords and on basis of expert knowledge. Consequentially, papers that are relevant are used in forward- and backward snowballing to look for foundational or follow-up papers on the topic. Each paper used needs to meet either of two quality standards. It needs to be either (i) an academic paper that is published with at least more than one citation, or (ii) a renowned university-approved master's thesis. Other sources such as documentation for tools may also be used but will be documented with footnotes indicating that it is not an academic yet trustworthy source.

## 3.2 Translating a Choreography to Petri Nets

In the thesis describing INORA [2], choreographies are depicted with an adapted version of BPMN 2.0 [42, 43]. In general, a choreography is a global representation of the interactions between multiple organizations or organizational units involved in a common business process [44]. An example of such a notation can be found in Figure 3.1. Choreographies are not solely used to describe business processes, but are also used to describe a systems concurrency on a high level when designing software as is proposed with INORA. It is a helpful method to capture all possible user scenarios in one model as it captures all interactions between system components, without exposing their internal structure [45]. It is defined as a type of concurrency model, as such a choreography can define [18]

1. subprocesses that can be executed at the same time (concurrently)
2. interruptions as a result of exceptions
3. interactions between process participants

The representation of the choreography in BPMN 2.0 yields several advantages. It is (i) the de-facto standard for representing any process in a very expressive graphical way [46] and is an increasingly popular method [47] (ii) it is already a semi-formal language and addresses most concerns raised in

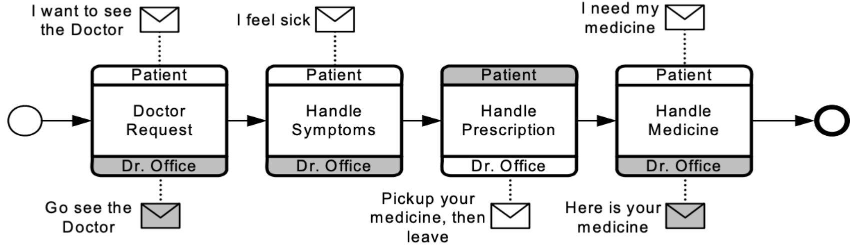



Figure 3.1: Example of an choreography in BPMN 2.0 [20]

[7] and (iii) is intuitive and flexible [48]. The adapted version of BPMN Choreographies used in INORA only uses the AND-gateway and the XOR-gateway and displays the messages between the sender and receiver as the name of the activity, which further simplifies the capabilities of the choreography [2].

The choreography is thus an intuitive way to model an interaction between software components, yet it is a semi-formal language and thus lacks the formal semantics and properties to verify this proposed system [3]. A translation is needed to formally verify these choreographies so that it can be used to automatically validate it [47].

The INORA Choreographies are a stricter version of BPMN Choreographies, and a translation for any BPMN Choreography to a Petri net can be used to translate the INORA version. BPMN Choreographies actually are BPMN models with each activity having a sender and a receiver. Therefore a BPMN to Petri nets validation should also work for INORA choreographies as tasks are labeled with the corresponding messages as stated by [2]. The net formed after the translation is then copied over for each participant (distinct set of all sender and receivers in the choreography) and connected by message places [2].

A mapping for translating BPMN Diagrams to Petri nets can therefore be used to translate INORA choreographies. Such translations have already been proposed in several papers. [49] and [38] laid the theoretical foundation to translate BPMN Tasks, Events and Gateways to Petri net elements in 2007. It is further refined in [18]. The basic building blocks for translating a BPMN model to a Petri net can be found in Figure 3.2. Further studies [3] define the reason for translation, investigate how the translation is done, and list problematic features of BPMN. A major unsolved problem is the relation between "correctness" and whether a given choreography is realizable. If a Petri net that is translated from a choreography is operable (definition 3.1), realizability is assured. Yet operability is undecidable [29].

**Definition 3.1**: Operability, adapted from [29]

An open net $N$ (with interfaces to connect components) with an initial marking $m_0$ and a given set of final markings $\Omega$ is called operable if there exists an open net $N'$ with an initial marking $m_0'$ and a set of final markings $\Omega'$ such that for any marking $m \in \mathbf{R}_{N \bigoplus N'}(m_0 \bigoplus m_0')$ there exists a marking $m_f \in \Omega \bigoplus \Omega'$ such that $m_f \in \mathbf{R}_{N \bigoplus N'}(m)$. $N'$ as above is called a partner of $N$.

A Petri net belonging to a choreography itself can be "correct" (checked and free from problems listed in section 2.5) yet it may not be realizable. In [25], the author proves that if the skeletons of two state machine nets (S-net) are isomorphic with respect to some function $\rho$ and their composition agrees on $\rho$, it is possible to assure realizability. This property is used in [2] to copy over a Petri net for each participant and connect them via the message exchange places. This is called the participant net. There is no additional literature that explicitly uses this for verification, and the use defined in [2] lacks a formalization and has some unresolved problems such as not defining strict conditions to assure AND and XOR gateways do not introduce decision-making problems. This is a clear gap in literature this thesis aims to fill.

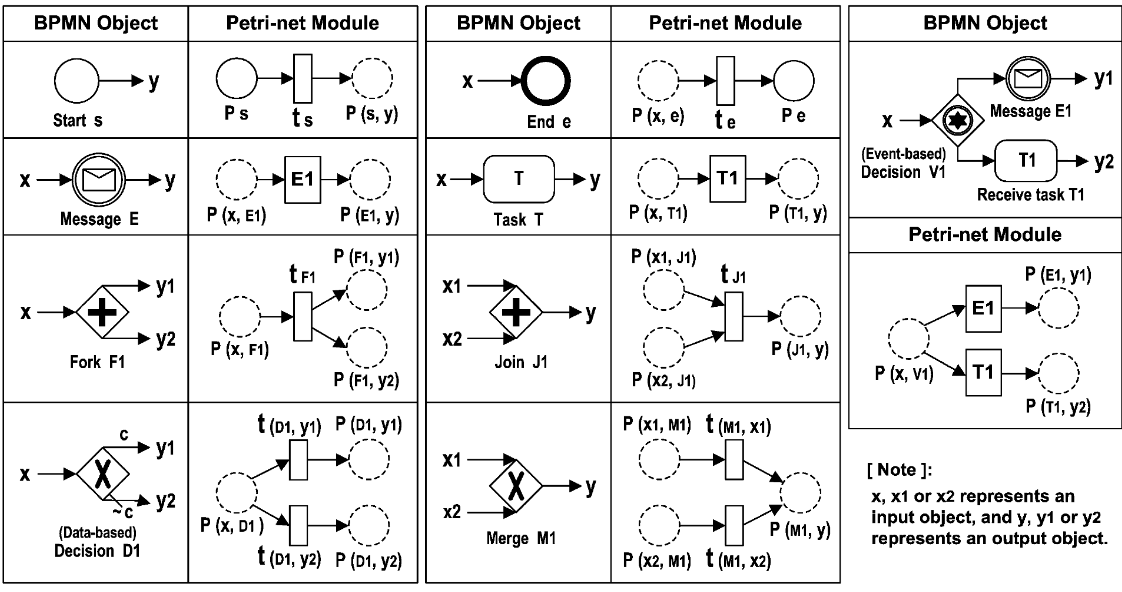



Figure 3.2: Building blocks for BPMN Diagram to Petri net translation by [18]

With regard to translating BPMN to Petri nets in general, more recent studies show that there are approaches to automatically execute a translation to Colored Petri Nets [47]. This is later refined by including interaction (choreography diagrams) within the model [50, 51]. A 2018 study [51] gives automatic translation steps that include pre-processing of the BPMN model so that it is strict and complies with formal BPMN semantics. Furthermore, it includes Petri net refinement and applies reduction rules, and covers step-wise Petri net element (place and transitions) coordinate assignment for a clear depiction of the final net.

The INORA choreography may reference other protocols, which can be substituted to construct a "composed structure". Such a composition is not considered a choreography anymore, as it may violate restrictions such as having a single *begin* and *end* event. This is partially covered in [2], but conclusive research is missing in this field.

## 3.3 Tool-based Verification and Analysis of Petri nets

A crucial element in answering the research question is to do an automated verification and analysis of a constructed Petri net. This section will cover literature concerning the (automated) checking of Petri nets to give a background on $SQ_3$.

### LoLA

Any combination of properties mentioned in section 2.5 can defer a certain quality of a given Petri net. Checking for these properties can be done automatically. There are several tools developed to automatically and optimally check for these properties and run analysis on the nets. Examples of tools are 1) Wolflan [52], which checks if the Petri net is valid and sound, 2) INA [53], which checks the net on a number of properties and shows the reachability of coverability graphs and 3 )LoLA [54], which tackles the state space explosion by using reduction techniques and can run CTL formulas to check for a number of Petri net properties.

In this thesis, LoLA 2.0 (Low Level Analyzer 2) is used to verify Petri nets resulting from choreography translation. The two main reasons for this are 1) it is highly optimized and achieves a high speed for tasks 3) it is the Petri net checking project that received the most recent updates. LoLA 2, in comparison to its predecessor, is based on a strict modularisation and has the integration of various standard tools [55]. "Through its code quality and its frequent comparison to other tools in the yearly model checking contests, LoLA 2 has become one of the most reliable verification tools for distributed systems" [55]. The tool uses explicit model checking as opposed to symbolic model checking. This means that LoLA generates and evaluates states one by one. Checking for a certain property in a Petri net is called "querying".

LoLA 2 traverses a subset of the reachable states that is as small as possible yet, by construction, sufficiently large for evaluating a given query [56]. This means that it needs to construct a different subset for every distinct query. Where implicit models either give all the answers (to queries) or result in a time-out during the generation of their diagram, explicit tools often return an incomplete list of answers which generally is enough. Furthermore, LoLA 2 applies on-the-fly verification [56] which speeds up the process significantly. [56] found that LoLA 2 was the fastest explicit model checking tool during the 2015 Model Checking Contest. Although LoLA 2 does not get recommendations as the most versatile Petri net verification tool in the comparison done in 2015 [57], it supports all the functionalities which are needed in this thesis to verify choreographies. Combined with the performance this tool is chosen to execute the (close to) real-time verification.

LoLA and LoLA2 have already regularly been used in practice to verify a wide range of problems. It is used for checking compliance (a set of rules) on the Petri net models that represent data fragment case management [58]. This is done by entering temporal logic [59] into LoLA 2, which supports the languages LTL/CTL to do this.
In [60], the Business Process Execution Language for Web Services (BPEL) is transformed to a Petri net so that it can be used as input in LoLA and the paper proves that it is well suited for computer-aided verification. It uses a similar basis as this thesis: translating a semi-formal process to Petri nets. This same approach is also used in PlanICS which is defined as "a system that solves the Web service composition problem by dividing it into several stages. The first phase, called the abstract planning, deals with an ontology that contains a hierarchy of classes describing sets of real-world services and processed object types." [61]. The paper uses LoLA as an abstract planning engine.
Furthermore, LoLA 2.0 is already included in Petri net tools. An example is RENEW [62], which is a Petri net IDE that combines the advantages of Petri nets and object-oriented programming for the development of concurrent and distributed software systems. RENEW offers model verification through LoLA.

This thesis uses LoLA 2.0 as a standalone service, which is connected to any service that wants to verify a Petri net. This is done by running LoLA 2.0 on a docker container and providing a REST protocol endpoint for services to connect to. This method (i) assures safety as LoLA is not run within the tool itself (ii) guarantees that *any* service that needs a LoLA verification can use it as long as it is authenticated and communicates via the REST protocol (iii) LoLA only needs to be compiled once on a given machine (iv) LoLA can be hosted on a powerful machine, such as a server, to provide performance on complex queries. This docker image with LoLA and Python is explained in detail in chapter 6.

## 3.4 Feedback on Models

A part of the problem is researching the feasibility of showing *feedback* on the models. This section shows the types of feedback and possibilities of handling the derived results from an analysis.

### Feedback Composition

As stated in section 3.3 there are several properties on which a Petri net can be verified. If a given Petri net is *safe*, *live*, *free of livelocks* and *sound* it is of high quality. If it fails on one or more properties, it is probably of lower quality. However, it does not have to mean the net is of extremely low quality. Furthermore, it is quite hard to test for *soundness* as it consists of multiple sub-properties that can require a lot of computational power.

It is, therefore, infeasible to calculate a single quality "score" by combining all outcomes for each test. Such an aggregation suggests an arbitrary definition of quality. A formal combination framework, however, for qualification is outside of the scope of this research. Feedback will therefore be composed of these properties as individual weaknesses/strengths, which an architect can view in a summarised manner.

### Translation of Feedback to Context

Once a Petri net is verified and queried for certain properties there is a "result" for each property. Results may be simple boolean answers such as $\{deadlock : no, sound : yes\}$ but are often more complex. It may exist of a mathematical notation for a counter-problem as a proof, or return (a set of) weak *places*, *transitions* and *arcs*. These formal results need to be translated back to human-friendly feedback which

makes sense in the original context. There are multiple ways to do this.

One of the aims for this thesis is to minimize the interaction of the modeler with the formal model (Petri nets). Simply displaying the LoLA feedback such as "$m_x \in M$ has no successor marking which implies a deadlock" is not useful for a user as the abstract representations of the net have no translation to their choreography. This problem is described by [63] as "formal verification is (...) usually achieved using model transformations. However, the verification results are available in the formal domain which significantly impairs their use by the system designer who is usually not an expert in the formal technologies". This problem is solved by either 1) embedding the context information in the formal model [64] or 2) creating an additional intermediate model that consists of a "mapping" between the context-model (source) and the formal model (target) [65]. Both guarantee traceability, which is the term used for linking elements in one model to the other. To avoid pollution of the formal model (Petri net) [63] an intermediate model is best to use. This is called a *Source2Target model*. Its use definition is depicted in figure 3.3.

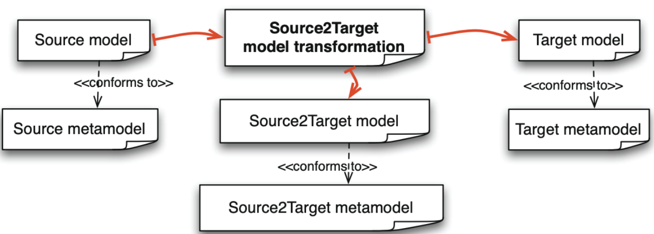



Figure 3.3: A translation model that assures traceability between elements in the context (source) and formal (target) modeling language [63].

This thesis uses an adapted version of the *Source2Target* to assure traceability. Furthermore, it is possible to simplify the terminology used in the feedback. Livelocks, for example, may be translated to "possible infinite loop". This ensures that the architect does not need prior knowledge of these terms.

### Visualization of Feedback

Feedback on translated Petri nets needs to be displayed in the context of the choreography itself. As there is a lack of research on the semi-formal choreographies translation to formal Petri nets, logically there is also no research on how to display temporal problems (such as (dead)locks as a result of synchronicity problems) on non-temporal models such as choreographies in the context of software interactions. This is also a clear gap in the research.

A recent paper [66] has made recommendations for visual feedback on BPMN models, however, that are coherent with the needs of the modelers. It found that three different categories of need exist as to *what* feedback should be provided: (i) alternatives to correct the problem (ii) problem description (iii) type of problem. Furthermore, 7 categories are identified as to *how* this feedback should be visually displayed. These can be seen in figure 3.4.

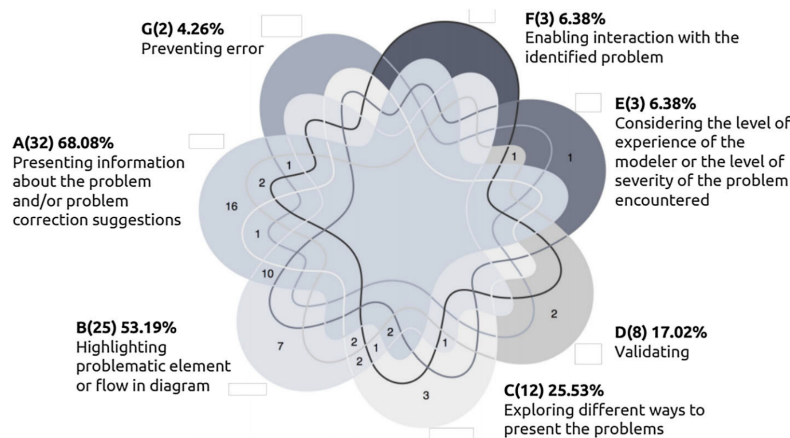



Figure 3.4: Seven categories of concern as to how to display feedback on BPMN models, image from [66].

From this research, some important conclusions can be drawn. For our implementation, the following recommendations are the most important:

- A basic convention that is used is the color scheme, using *red* for errors and *yellow* or *orange* for warnings [67].
- The decision to use either preemptive or non-preemptive feedback highly depends on the user's experience with modeling. Giving the choice to the user results in the best user satisfaction. However, much of in INORA the choice is made to trigger it *non-preemptive manually*. The two main reasons for this are that (i) the generic rules and the Petri net of non-complete models will often not comply with the property analysis and therefore bloat the feedback pane with unnecessary errors and (ii) running a constant stream of Petri net verifications may be very expensive and a waste of computer resources.
- Problems can be marked with visuals on the model itself and are directly linked to a detailed explanation in the list of feedback. Clicking on either of the two will highlight the other.

The goal is to include all errors and warnings onto the BPMN model itself. A general image by [66] as to how the feedback can be shown is seen in figure 3.5.

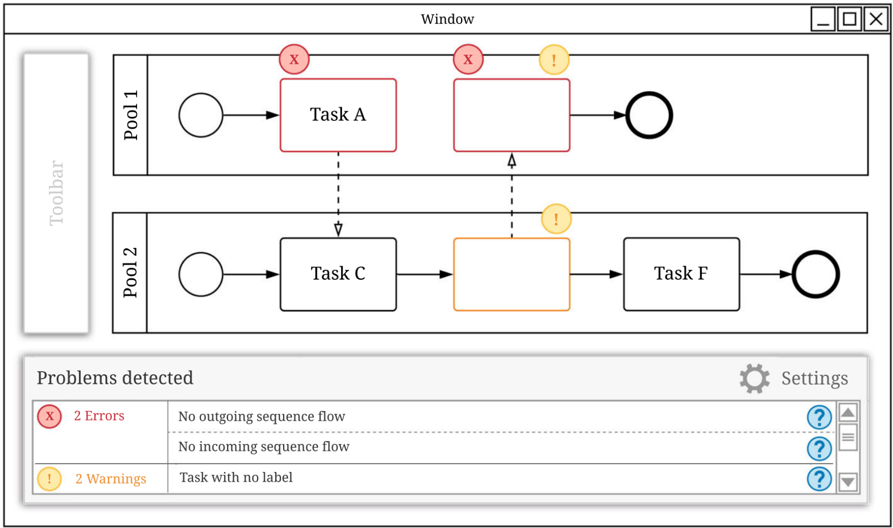



Figure 3.5: A wireframe model of some tool implementing feedback as stated in the section above, image from [66].

## 3.5 Summary

In this chapter related work is used to provide background information and identify gaps in the literature on the automatic verification of choreographies. Current literature shows the use of choreographies and why and how they can be translated to Petri nets. We then highlight the problems and shortcomings in the existing literature. The major shortcomings that are found are 1) limited research on participant nets to assure realizability 2) lack of description of the behavior of composed choreography structures 3) displaying feedback on interaction problems found by semi-formal-to-formal translation of choreographies to Petri nets 4) deciding which combination of techniques can provide soundness. Furthermore, this chapter shows how and on what properties Petri nets can be (automatically) checked and what the impact is on the quality of a net. In the last section, we described how to translate these results into potential user feedback.

# Chapter 4

# A Robust Translation

One of the key aspects of the problem this research is trying to solve, is to provide formal semantics so that we can check models for soundness. In [2] the author presented an adapted version of [38] to translate semi-strict BPMN choreographies to Petri nets, using duplication and connection of sub-nets. In this chapter, this proposed translation will be used as a foundation. It will then be formalized, extended and partially rewritten so it can be used to run an efficient analysis. To achieve this, additional bounds to the choreography are introduced, which will be discussed and introduced in this chapter.

In this chapter, an simple choreography is used to connect the abstract definitions and translation segments to a context. In this protocol $a$ requests something from $b$ (signage $m?$), $b$ then either responds with “valid” or “not valid” (both signage $m!$), and the protocol finishes.

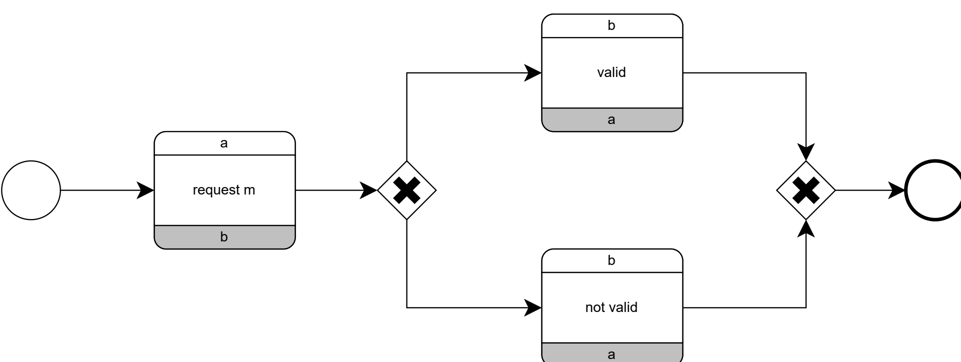



Figure 4.1: An abstract basic protocol

## 4.1 Constraints to INORA protocols

Before any translation can be made, the INORA protocol must comply with a series of constraints. The constraints on the contents (CoC) of such a protocol $P$ in a non-graphical format are:

| $P$ | **rule** | **cardinality** | **type** | **with** | **conditions** |
|---|---|---|---|---|---|
| | (c01) must have | 1 | StartEvent | | 1 outgoing arc |
| | (c02) must have | 1 | EndEvent | | 1 incoming arc |
| | (c03) must have | $n > 1$ | Message | | 1 incoming and 1 outgoing arc<br>+ 1 Message label<br>+ 1 Initiating participant $i$<br>+ 1 Receiving participant $r \neq i$ |
| | (c04) may have | $n$ | Reference | | 1 incoming and 1 outgoing arc |
| | (c05) may have | $n$ | Fork Gateway (XOR) | | 1 incoming and $m > 1$ outgoing arcs |
| | (c06) may have | $n$ | Fork Gateway (AND) | | 1 incoming and $m > 1$ outgoing arcs |
| | (c07) may have | $n$ | Join Gateway (XOR) | | $m > 1$ incoming and 1 outgoing arcs |
| | (c08) may have | $n$ | Join Gateway (AND) | | $m > 1$ incoming and 1 outgoing arcs |
| | (c09) must have | $n > 2$ | Directed Arc | | each connecting $a_i$ with $a_j$ |
| | (c10) may have | $n$ | Other Label | | - |
| | (c11) cannot have | $n$ | $else$ | | - |

Furthermore, the following generic constraints hold.

- **(Gc01) No ambiguous gateways.** Each gateway should be explicitly labeled with the icon corresponding to either “XOR” or “AND”. All other gateways present in the BPMN notation are not allowed as they unneccesarily complicate interactions in translation.
- **(Gc02) Forks and joins are only modeled with gateways.** This is implied in the CoC: it is not allowed to split or join arcs directly from a non-gateway element. Although generic BPMN allows for unlabeled splits/merges implying an AND/XOR gateway, protocols will only support explicit gateways.
- **(Gc03) Use valid participants.** Each participant in a Message should a valid *function* in the Interaction Model.
- **(Gc04) Use BPMN stardards.** For the visual notation the BPMN 2.0 choreography graphical standard is used [43, 45], for constraints not explicitly described in this chapter.
- **(Gc05) Follow up decisions with an activity.** A decision (XOR-fork) should always directly be followed by an activity in all outgoing paths. This ensures decidability: decisions are made by the protocol itself.

## 4.2 Petri Net Transformation

For each of the distinct events and structures in a BPMN choreography a basic sound Petri net representation can be created. The definitions by [2] are used as a basis, and are refined and formalized.

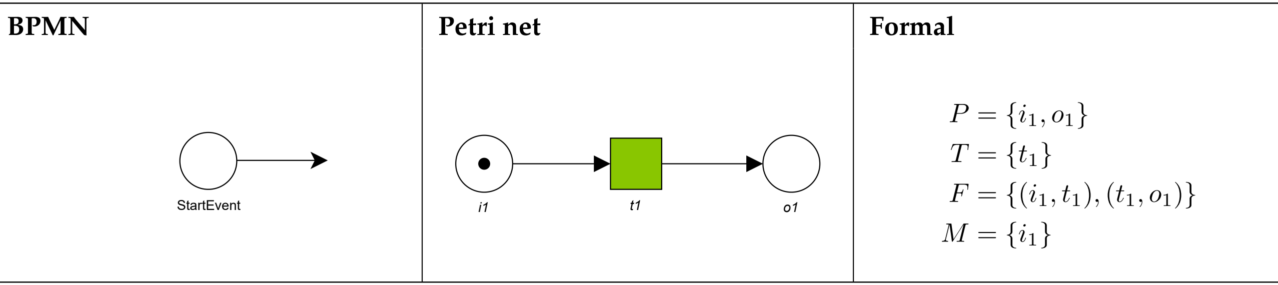
### Building Blocks

Each possible element in any choreography has its own representation in the Petri net. In this section, we define for each possible element (and its sub-behavior e.g. fork or join gateways) how a translation can be made. The official Petri net notation (as denoted in Definition 2.3) is used. Gateways are the only elements that are allowed to have an $1 : n$ interaction with other elements. Therefore, when $n$ is mentioned, it represents a set of natural numbers from 1 to $n$.

| **Translation** |
|---|

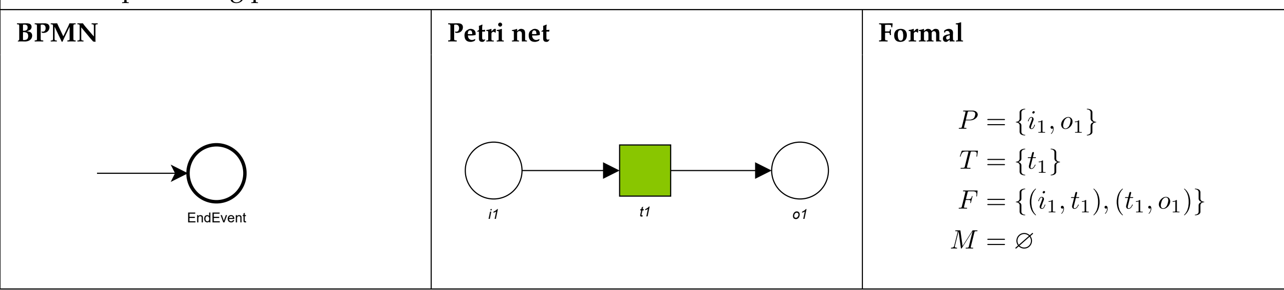
**Start Event**

The start event can be translated to Petri nets by creating a place $i_1$, a transition $t_1$ consuming tokens in $i_1$, and a producing place $o_1$. This is the only translation in which a token is placed initially: the input place for the start event is the start of a Petri net.



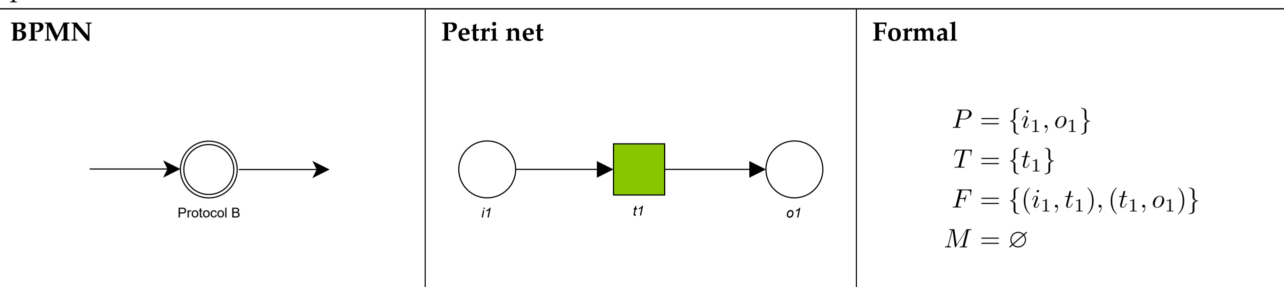

**End Event**

The end event can be translated to Petri nets by creating a place $i_1$, a transition $t_1$ consuming tokens in $i_1$, and a producing place $o_1$.



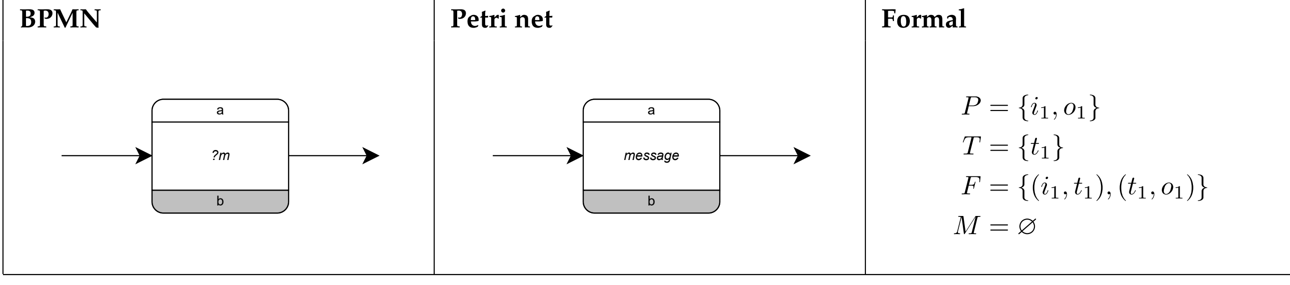

**Reference Event**

The reference event can be translated to Petri nets by creating a place $i_1$, a transition $t_1$ consuming tokens in $i_1$, and a producing place $o_1$. The workings of transition $t1$ are to be refined with the corresponding protocol.



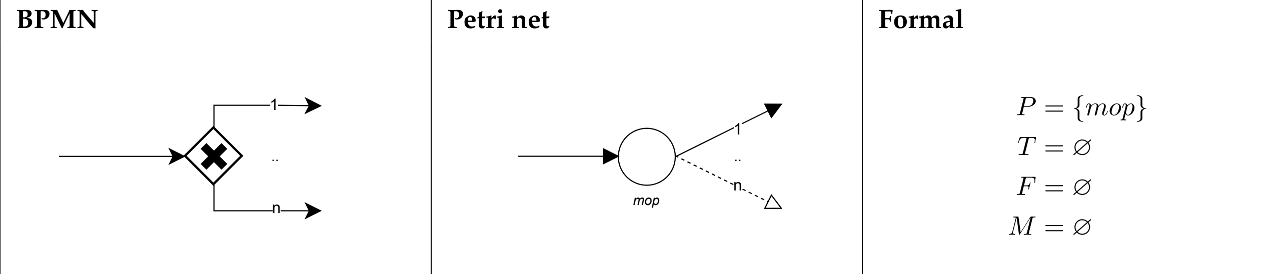

**Interaction** (aliases *message* and *activity*)

The interaction can be translated to Petri nets by creating a place $i_1$, a transition $t_1$ consuming tokens in $i_1$, and a producing place $o_1$. This translation is correct under the condition that an interaction can only have one incoming and one outgoing arc, as defined in the generic constraints.



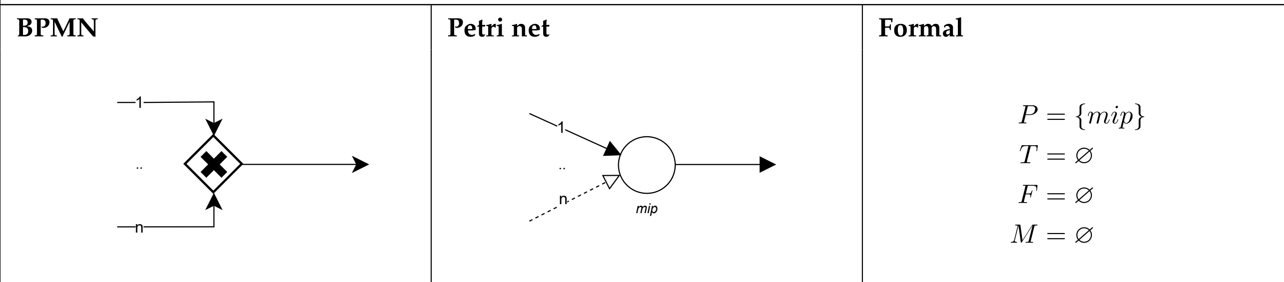

**XOR Gateway (fork)**

An XOR gateway that splits intro multiple branches can be translated to a Petri net using a single place. This place has the properties that it can have 1 incoming arc, and $m = n = arcs_{out}$ out arcs. In the Petri net representation, these arcs are drawn for clarity, although they are not part of the actual translation as they indicate arcs *between* translation segments. An important characteristic of this translation is that as there are no transitions in this translation segment, the XOR step itself cannot "make a decision". The decision-making is done by the transitions connected to the outgoing arcs.



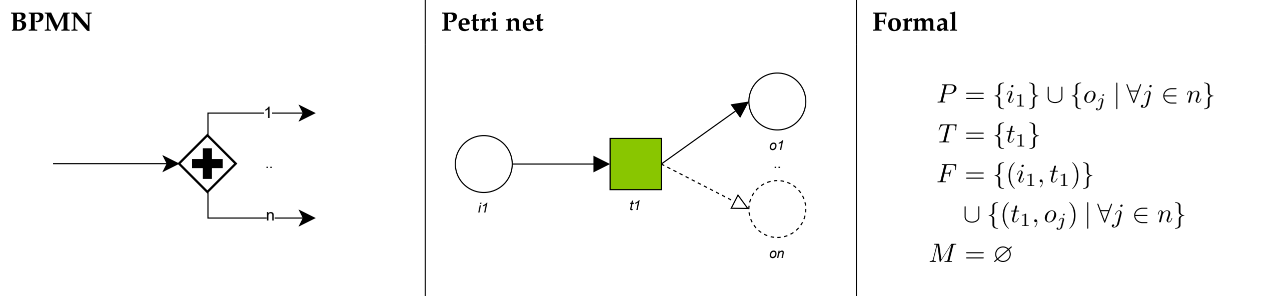

**XOR Gateway (join)**

An XOR gateway that joins multiple branches can be translated to a Petri net using a single place. This place has the properties that it can have 1 outgoing arc, and $m = n = arcs_{in}$ in arcs. In the Petri net representation, these arcs are drawn for clarity, although they are not part of the actual translation as they indicate arcs *between* translation segments. An important characteristic of this translation is that as there are no transitions in this translation segment, the XOR step itself cannot "make a decision". The decision-making is done by the transitions connected to the outgoing arcs.




**AND Gateway (fork)**

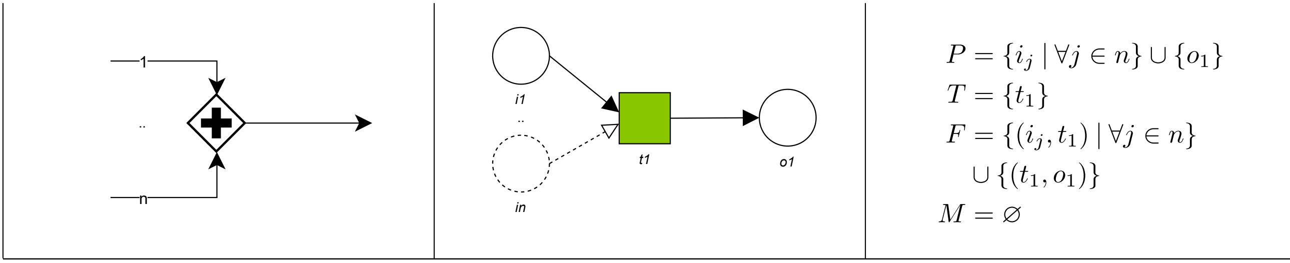
An AND gateway that splits into multiple branches can be translated with one input place $i1$, one transition $t1$ that consumes from the in place and produces in all of its out places, and $n = arcs_{out}$ out places.




**AND Gateway (join)**

An AND gateway that joins multiple branches can be translated with $n = arcs_{out}$ out places $o_n$, one transition $t_1$ that consumes from all of the in places and produces in its out place.

| BPMN | Petri net | Formal |
|---|---|---|




## Procedure

In the previous section, all individual translation segments are defined. This section defines a step-by-step procedure to apply and combine all translation blocks. We use the protocol given in 4.1 to bring each step into context and provide an example.

The translation procedure is defined:

1. **Check whether the tree violates the generic constraints (CoC) given in section 4.1.**
   In our protocol from Figure 4.1 we check for all constraints $c \in \{c01..c11\}$. It does not violate any of them. Furthermore, the generic constraints are not violated either:

   Gc01 (OK) All gateways are explicitly labeled with XOR labels.

   Gc02 (OK) There are not merges or splits without involvement of a gateway.

   Gc03 (OK) The interaction model is not given, therefore this condition is always OK.

   Gc04 (OK) BPMN standards are followed for any non-explicit constraint.

   Gc05 (OK) Each outgoing path of the XOR fork is directly followed up with an activity.

2. **Translate each element**
   The first step is to generate a raw net that looks like a Petri net, but is not a valid Petri net yet. It maps the Petri net translation segments to each other. This means simply walking over all elements in the choreography from the start event and replacing them with their Petri net notation respectively.

   For the protocol depicted in Figure 4.1, this would result in the following net. This is not a Petri net by definition, as there are mappings between places indicated by the red lines. We will call this a "raw" net.

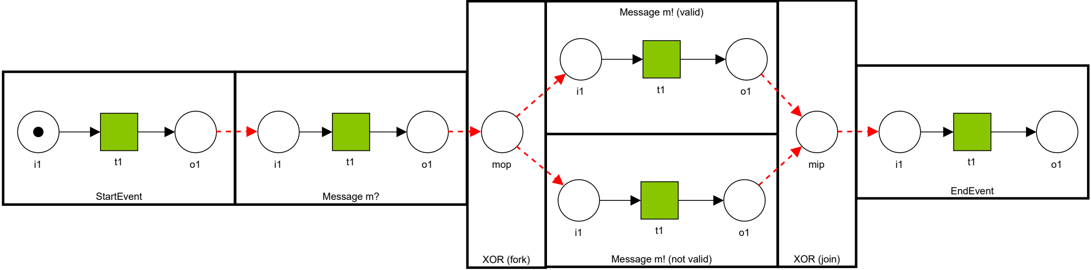



Figure 4.2: Step 1 result: "raw" net, with each translation segment indicated seperately.

3. **Merge duplicate places**
   In the raw net there are flows between two places, and places cannot move tokens. Therefore this violates the Petri net rules. Semantically it means they behave as a single place, and therefore multiple directly connected places can be merged and outgoing and incoming flows connected to these groups are assigned to the merged place. The Petri, after this refinement, net would look like this:

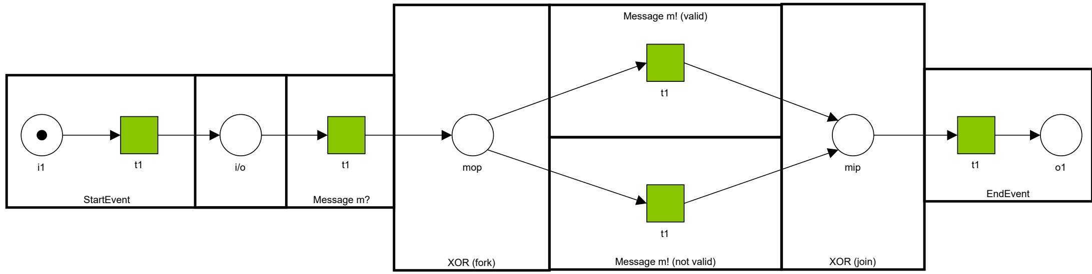



Figure 4.3: Step 2 result: "refined" Petri net

4. **Create a participant net**
   The last step is to create the net involving the communication between the components, called the participant net. Components participating in an interaction are called "participants" in this context. Interaction can only happen with a *message* element, and a message element always mentions two participants: the initiator in the upper band, and the receiver in the lower band.

   (a) Find all participants in the protocol (check each of the message bands). In this protocol, the participants are $U = \{a, b\}$.

   (b) Copy over the net generated from the previous step for each unique participant. The result is two identical nets $U_c = \{C_a, C_b\}$.

   (c) For each of the nets in $U_c$, look for every message in which its corresponding participant is the initiator and connect it to messages of other nets in $U_c$ in which their corresponding participant is the receiver. Such a connection is drawn from one transition to another transition, so connection places called the *interfaces* are needed, as defined in definition 2.7.

   After these steps, the participant net is created which can be seen in Figure 4.4.

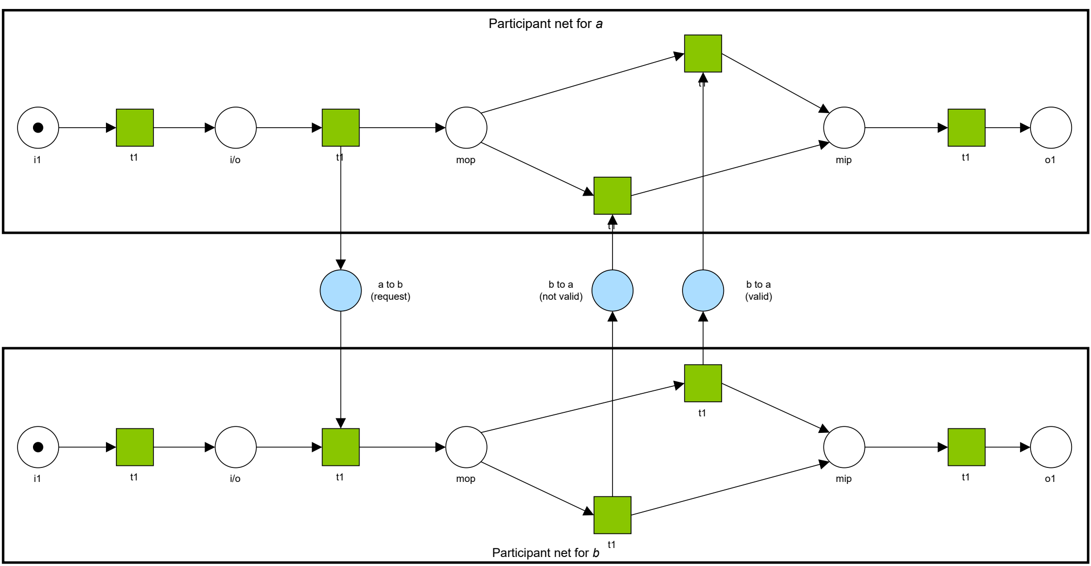



Figure 4.4: Step 2 result: "participant" Petri net

Executing this procedure results in the final Petri net, on which analysis can be performed.

## 4.3 Block structured Choreographies

Up until now, the literature has covered mostly the translation from a protocol directly to a Petri net. If a protocol complies with the constraints given in the previous section it is always possible to generate a corresponding Petri net and verify the soundness of the protocol by doing Petri net analysis. However,

checking Petris net for certain properties (e.g. liveness, deadlocks, reachability) can be very expensive due to two reasons: 1) the complexity of the verification algorithms is inherently quite high [23, 68, 69] even if efficient tools such as LoLA 2.0 are used [54, 70] 2) Petri net checking needs to be done by some optimized external tool, and therefore I/O operations (potentially over a network) need to be done, which can be expensive in an analysis. Therefore it is essential to skip Petri net checking when it is not strictly necessary and only do it when soundness is truly uncertain.

Soundness can be guaranteed if it is a choreography tree that follows a certain grammar resembling a process tree, and is block-structured. We call this correctness by constructing.

**Definition 4.1**: Process tree [71] [72]

A process tree is a tree representation of a process. Each leaf node and each internal node respectively represent an action and an operator in the process. A choreography tree in this thesis follows a specific grammar given in Definition 4.2.

The grammar called the Choreography Tree Definition, is given in Definition 4.2 and is used to set definitions as to when a protocol is a choreography tree.

**Definition 4.2**: Choreography Tree Definition

**Grammar**

$$A ::= message$$

$$\begin{aligned} S ::= &\ start \\ &\mid end \\ &\mid A \\ &\mid \times (S, S) \end{aligned}$$

$$\begin{aligned} T ::= &\ S \\ &\mid \Omega\ (T, A) \\ &\mid \rightarrow (T..) \end{aligned}$$

**(1) Property: Associativity of operators**
For $\phi = \{+, \times, \rightarrow\}$ the associative property holds, which means that

$$\phi(\phi(A, B), C) \stackrel{\ell}{\equiv} \phi(A, \phi(B, C)) \tag{4.1}$$

This Tree Protocol Definition grammar $T$ has three possible nodes: $S$, a loop $\Omega$ with another node in $T$ and an message $M$, and a sequence of actions consisting of any amount $\{t_1...n\} \in T$. This implies a couple of constraints to a choreography.

1. A sequence may contain any number of nodes of type T. This is the node with the largest distinct set of nodes and therefore a sequence in a choreography may exist of any other node defined in the grammar.
2. A loop (expressed with a XOR gateway directing back to a earlier XOR gateway in the choreography) should always exist of a choice between a back-arc to an activity $A$ and a $T$ continuing the choreography forward with any node. This implies that an "empty" back arc (known as a $\tau$-step) is not allowed.

3. A $T$ maybe also be a substitute node $S$, which is a subset and has its own defined sub-grammar.

Nodes of type $S$ imply the following constraints.

1. An $S$ can be either a *start*-event, an *end*-event or an $A$. An $S$ may also be operator XOR ($\times$).
2. Operator XOR can only contain exactly two $S$. Due to property 1, associativity for operators $\{\times, \rightarrow\}$ , it is still possible to give a correct representation of three or more items in one of these operators. The tree $\times(A, B, C, \triangle)$ is rewritten to $\times(A, \times(B, \times(C, \triangle)))$. The equivalency guarantees this. Operator $\{\times\}$ containing only one element is not useful: they pollute the tree and are therefore not allowed.

An $A$ is a message block with a signage. A signage indicates the message flow between participants, given in Definition 4.3.

**Definition 4.3**: Signage of messages

A message in a choreography is signed. Signage implies the message flow. A message from a component $\triangle$ to component $Y$ should at least be answered once by $Y$ in every possible path, or: $\triangle \rightarrow Y$ should always be followed by at least once by $Y \rightarrow \triangle$ to comply with correct asynchronous behavior in a choreography.

The translation blocks in a Petri net can also be marked with such signage. For example, a transition is marked with:

$$Y \rightarrow \triangle = m?$$
$$\triangle \rightarrow Y = m!$$

This is derived from the properties of an Open Petri net: communication from one service should be answered by the other at least once.

## Additional Constraints

To ensure the soundness of a tree, two additional constraints are stated. Properties concerning pure BPMN diagramming relate to the 7PMG guidelines [73], covering basic BPMN validity.

- **Block structure**
  One of the most important bounds is that the choreography is block-structured. This means that once a gateway produces branches (e.g. XOR gateway gives two options), such branches should eventually be merged by another XOR gateway before the end-event. Formally, each gateway that has $n(> 1)$ outgoing arcs has a merging partner that merges all $n$ arcs somewhere in the choreography. This means that $|G_{AND}|$ and $|G_{XOR}|$ should be divisible by 2.

  This is related to 7PMG guideline G4.

- **Signage in blocks**
  In a protocol choreography a interaction $m$ is signed with either $m!$ or $m?$. In a choreography with two participants $P = \{A, B\}$ an interaction request from $A$ to $B$ should always be answered by $B$ at least once in each possible path: when $A$ sends a message $m?$ to $B$ it should be followed by $m?$. $B$ acknowledges the request of $A$ and sends a response. This is needed as the protocol choreography describes how components interact, which implies that they should all have an active role in the protocol.

  For the translation this means that in a sequence ($\rightarrow$) the messages should alternate for at least one cycle (e.g. $\rightarrow (?m, !m, ?m, !m)$ with $m = sign(a) \mid a \in A$). For an XOR ($\times$) or AND (+) this means that the signage *before* the operator should be inverted for the first $A$ node in each of the branches after the operator:

$$\rightarrow (?m, \times(!m, \times(!m, !m)))$$

Note that in this bound it is assumed that there are two participants ($|P| = 2$). This is a loose bound, and it is possible to introduce more than two participants in the protocol. It is, however, not possible to directly tell if the signage fulfills a correct alternating pattern. We will then use Petri net exploitation in the implementation to check if the protocol is sound.

## Checking for Trees

Given the protocol, we need to determine whether it is a choreography tree. This is done by executing each of the following steps listed below. The implementation is written down in pseudo-code in 5.2. During these steps the protocol described in Figure 4.1 is used to bring each step into context.

1. **Check whether the tree violates the generic constraints (CoC) given in section 4.1.**

   *This step is identical to the first step in the Petri net procedure.*

2. **Check for loops**

   In the protocol there are no loops. This step *would* find the entries and exits of all loops in the diagram if (a) loop(s) *would be* found.

3. **Verify the block structure of the tree**

   The protocol is block-structured, as all opened gateways are closed by its counter partner. The XOR fork ($\times$) splitting into 2 branches is eventually followed by a join XOR ($\times$) merging the 2 branches before continuing.

4. **Verify the signage**

   In our protocol the two possible paths are:
   ```
   p1 : (StartEvent) -> (Message m?) -> (=valid) -> (Message m!) -> (EndEvent)
   p2 : (StartEvent) -> (Message m?) -> (=not valid) -> (Message m!) -> (EndEvent)
   ```

   The signage of all paths alternate at least once: `sign(p1) = m?m!` and `sign(p2) = m?m!`. Therefore this protocol does not violate this step.

5. **Construct the tree, and check if the EndEvent is reached.**

   Using depth-first traversal, construct the tree. The implementation. A detailed pseudo-code can be found in section 5.2. The resulting tree belonging to this protocol is depicted in Figure 4.5.

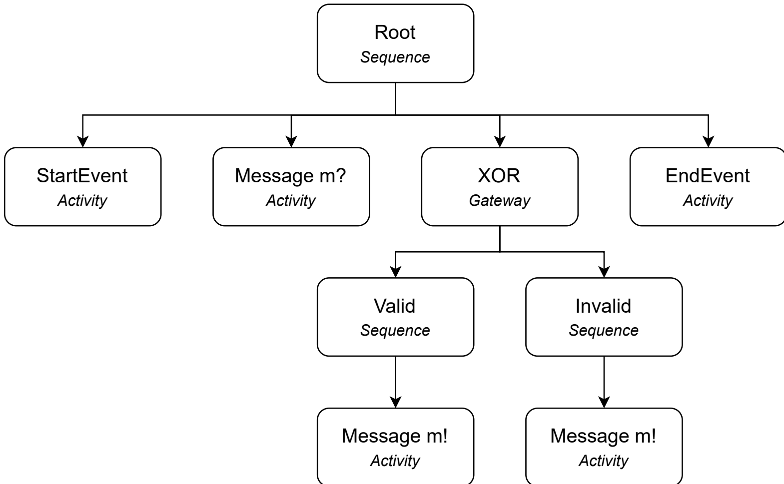



Figure 4.5: Choreography tree

Which can also be written in parenthesis notation, similar to the defined grammar:

$$T = \rightarrow (S_s, A_{m?}, \times(\rightarrow (A_{m!}), \rightarrow (A_{m!})), S_e) \tag{4.2}$$

The pseudo-code of this algorithm can be found in section 5.2.

If all steps are successful, the protocol is a choreography tree and we can guarantee its soundness. If any of the steps fail, it is not possible to determine whether the protocol is sound, but it still might be. If the choreography tree cannot be constructed, a Petri net translation and verification needs to be done.

## Complex Structures

Up until now, in this section, we have left out the AND operator. This is done intentionally, as it introduces a variety of complexity and problems. Special cases are known to result in inconsistency when translating it with previously specified building blocks. We classify these cases as complex constructions. One such a construction is a XOR gateway directly followed by one or more parallel (AND) gateways. To achieve this, an additional rule with equivalency needs to be introduced.

**+ Rule: XOR followed by AND**
If within the tree one or more children of an XOR are directly followed by an AND, rewrite it to the language equivalent ($\stackrel{\ell}{\equiv}$)

$$\times(+(X, Y), \triangle) \stackrel{\ell}{\equiv} +(\times(X, \triangle), \times(Y, \triangle))$$

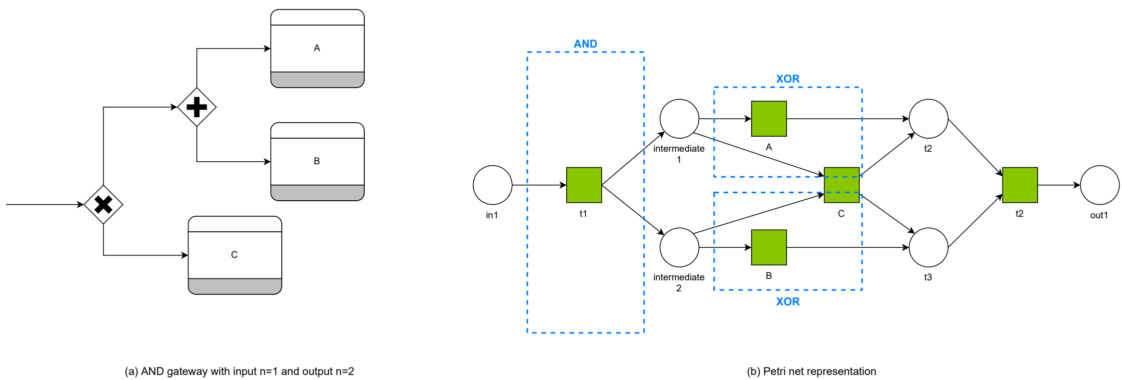
The Tree Protocol Definition has a special rewrite rule for this condition. This is used to form a correct translation. The *choreography tree* representation of the choreography snippet below is

$$\times(+(A, B), C)$$

which is rewritten to the language equivalent

$$+(\times(A, C), \times(B, C))$$

which has a sound translation as seen in (b). Note that component $C$ refers to the *same* element, although it is denoted twice.




(a) AND gateway with input n=1 and output n=2

(b) Petri net representation

If the equivalence condition is encountered, the part of the tree will be rewritten according to Rule 1. Such a rewritten version of the tree is called a *Choice Normal Form*. We can guarantee that the behavior is equivalent. This rewrite condition solves the problem of synchronicity in a tree. Because the resulting Petri nets use interleaving semantics we need to enforce that after an XOR operation, both branches can be executed.

This is just one example of a complex structure that can be achieved by using certain combinations of gateway flows. The complexity it introduces and the vast and ongoing research in the field of decidability in Petri nets is the reason that we leave it out of the current scope of this research.

### Petri net vs. Choreography Tree

As the Petri net can always be generated without the strict constraints needed for a choreography tree, the analysis run on a Petri net always dominates the choreography tree. However, once a choreography tree can be generated correctly, it also implies that the Petri net analysis will also result in soundness but the reverse does not hold.

Therefore we can skip the Petri net analysis once the choreography tree is constructed, as the analysis of a Petri net can be computationally heavy.

## 4.4 Summary

This chapter covered the procedure and necessary building blocks for translating a given choreography/protocol to 1) a Petri net and 2) a choreography tree. As a basis, a set of generic constraints are defined with which a protocol should always comply. To avoid a potentially costly translation and analysis of a Petri net, we show that it is possible to guarantee soundness if a choreography tree can be generated. As a Petri net can always be generated, and the choreography tree only in certain semi-strict conditions, the Petri net analysis will always be the dominant analysis.

# Chapter 5

# Implementation

Until now, a lot of theory has been discussed regarding translating choreographies to Petri nets, choreographies to choreography trees, and analyzing Petri nets. The research question(s) implies that some automation is needed to provide automatic feedback. This chapter introduces a custom tool that handles automation and support: INORA 2.0.

## 5.1 What is INORA2?

INORA2 is a completely rewritten version of the original INORA tool, created from scratch by the author. Whereas the its predecessor was built as a plugin for Sirius, INORA2 is written in Typescript and can be used as a standalone application. It offers a variety of analysis and insight modules, and refines the possibility to easily create and manage INORA models.

The application is project-driven: to make use of the power of INORA you should create or load a project. Once a project is loaded, the application has four major components.

1. **Main menu**: a menu bar providing all navigational and app- and project controls.
2. **File browser**: for quick navigation through all models, with the capability of search
3. **Bottom bar**: a context-specific bottom bar that allows for settings and environment-specific buttons.
4. **Main panel**: all the main content is projected to this panel.

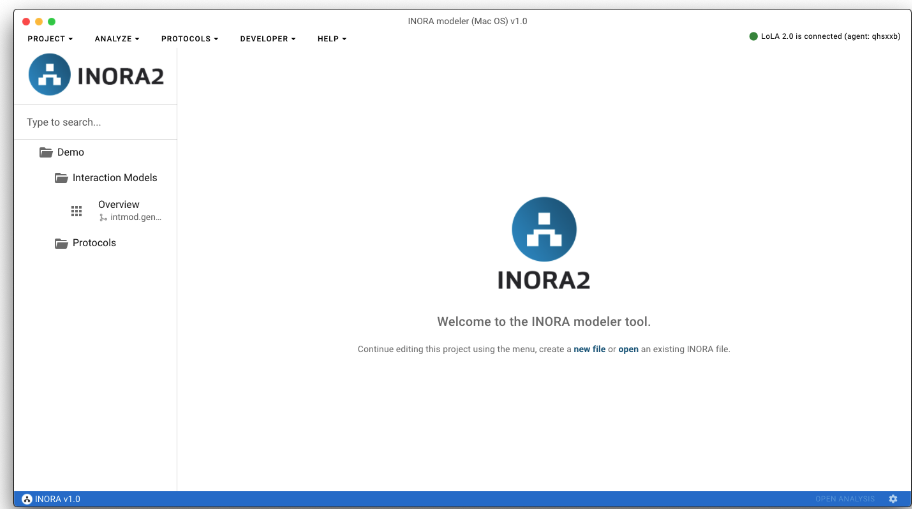



Figure 5.1: Basic INORA application frame

## Interaction modeler

The core of any INORA project is the interaction model. Only one interaction model can exist per project. Guidelines to draw interaction models is laid out in [2], which are adapted to create a slightly different version of the interaction model called Interaction Model Version 2 (IMV2). IMV2 is used in INORA2. In addition to some visual changes, the main difference is that there is no more distinction in communication within and across components. All interaction is now simply a called protocol, and all protocols should be modeled with a choreography. The reason for this original differentiation synchronicity. The assumption that interaction within components is always synchronous is dropped, as practically modeling with this assumption raises limitations to the system.

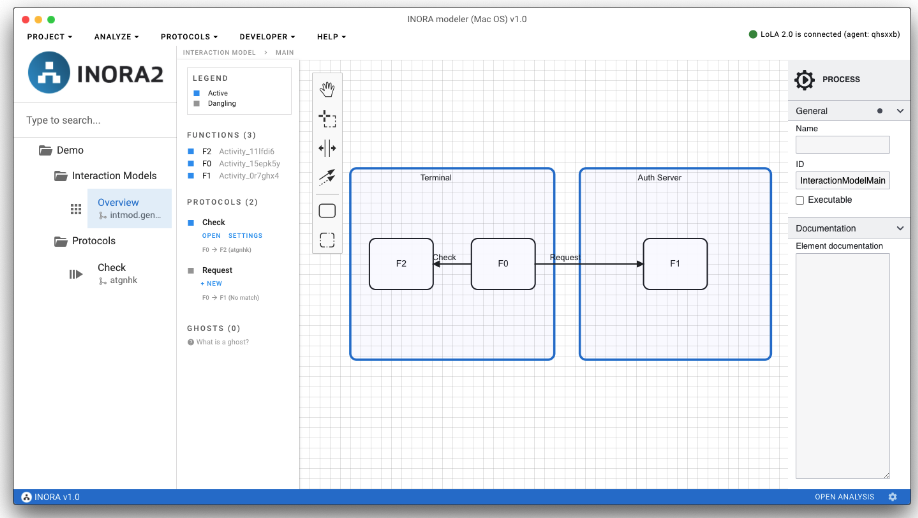



Figure 5.2: The INORA interaction modeler with the controls and two side panels.

INORA has a powerful tool to model interaction models. The editor provides the necessary means to create the three main components of an interaction model: 1) a function 2) a component in which functions *may* be placed and 3) a connection between any two functions. A run-time screenshot of the modeler can be seen in Figure 5.2. Items can be created, renamed, resized, connected and copied. Selecting an item brings up the possible modeling options as a pop-up and a way to control attributes in the right-side panel. The left-side panel will provide an overview of the contents, it displays:

- **All functions drawn** A list of all functions in the interaction model, and their *activity status*. The status is indicated by a square which is blue if the function participates in at least one protocol, and grey if it is *dangling*. By convention, dangling functions either will be connected in the future or should be reconsidered; "why is the modeler drawing a function that is not participating in any commutation in an interaction model?"
- **All protocols drawn** A list of all protocols i.e. arcs drawn between functions. As with the function their status is indicated by a blue/grey square, which means either a protocol is drawn **and** is modeled in the projects protocols (active, blue) or a protocol is drawn but is not (yet) modeled (dangling, grey). An active protocol provides the users with actions to open the modeler or its settings directly, whereas a dangling protocol offers to option to directly create a new protocol and start modeling.
- **Ghost protocols** A list of all protocols found in the project but that are not connected to any arcs drawn in the interaction model. The user is provided with the actions to (re-)attach such a protocol, or delete it.

## Protocol modeler

The protocol modeler is the true heart of INORA2. It allows users to quickly and efficiently model INORA protocols conforming to the requirements defined in previous chapters.

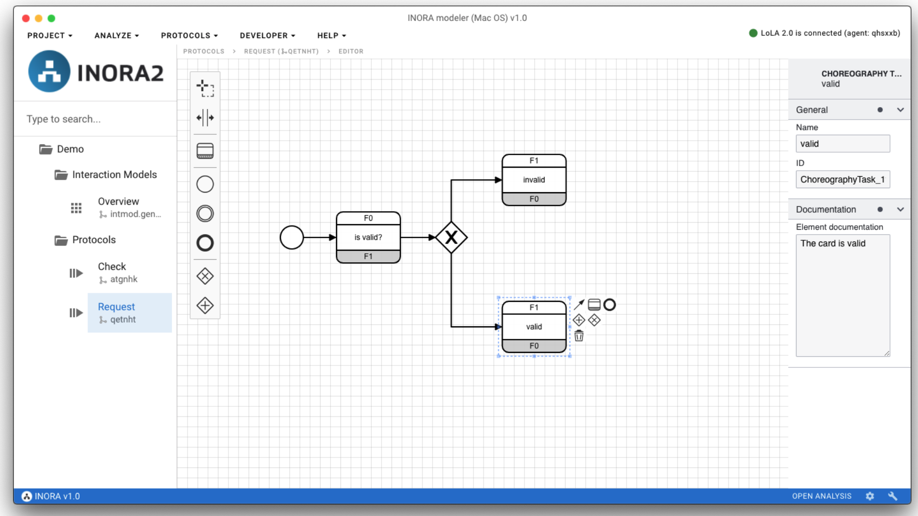


Figure 5.3: The INORA protocol modeler creating a simple XOR choreography.

Similarly to the interaction model it provides a modeling canvas, a side panel and a creation menu. It provides the possibility to create any protocol element from scratch, or (by selecting an element) selecting a logical follow-up in the pop-up menu. It also allows the user to control attributes, resize and move elements, drag and drop any element, rename or label any element and use canvas actions such as mass select and zoom. This modeler is an adapted version of the `chor-js` framework [74].

Elements in this tool *may be* linked to elements in the interaction model. While conventionally the expectation is that users model only with elements in the interaction model, this is not a restriction. As INORA2 is an educational tool, it is still possible to model generic interactions between an initiator and receiver for demonstration purposes. If used to actually draw active protocols from the interaction model, this user can make use of two ways to link the protocol to the interaction model:

- **Direct linking** Each protocol has a settings panel, in which the protocol can be explicitly linked to an interaction model element. This works even if the functions mentioned in the protocol are not used in the arc in the interaction model. This freedom is left to the user.

  Example: The user decides to model a simple interaction model between functions *F0* and *F1*. The user links this protocol to *Protocol A* in the interaction model. INORA2 now understands that the choreography and the protocol arc are linked. Note that if a user creates a choreography directly from the interaction model, the linking will automatically take place.

- **Participant reference** In the choreography model itself any interaction (denoted by a message activity) uses two participants: the initiator and the receiver. These two participants can be selected by clicking on the band of an interaction. Possible participants are:

  - Generic Initiator
  - Generic Receiver
  - A function in the interaction model.

  An example of such a participant selection can be seen in Figure 5.6

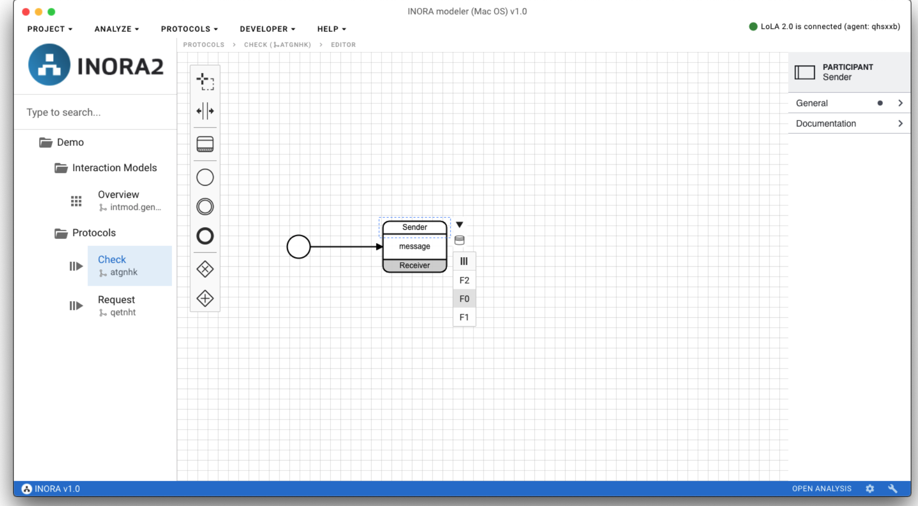



Figure 5.4: Selecting a participant for an interaction in the INORA choreography modeler.

## Protocol analysis

One of the aims of the thesis was to show feasibility of automated analysis on (composed) choreographies. A major part of the INORA tool is therefore dedicated to analysing the choreography. Both from the menu and directly from a choreography modeler view, the corresponding analysis can be brought up.

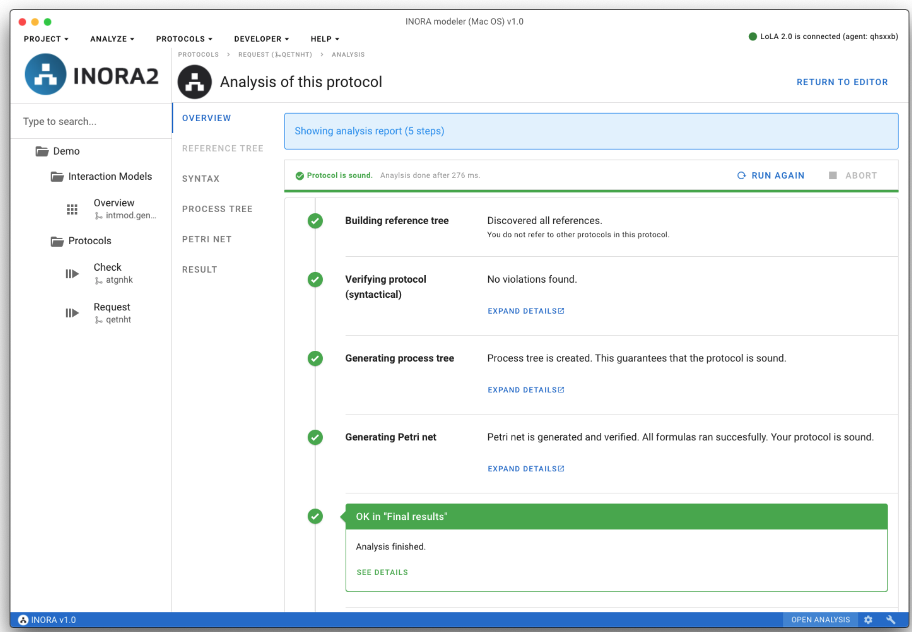



Figure 5.5: The analysis overview of a protocol in the given choreography is considered sound.

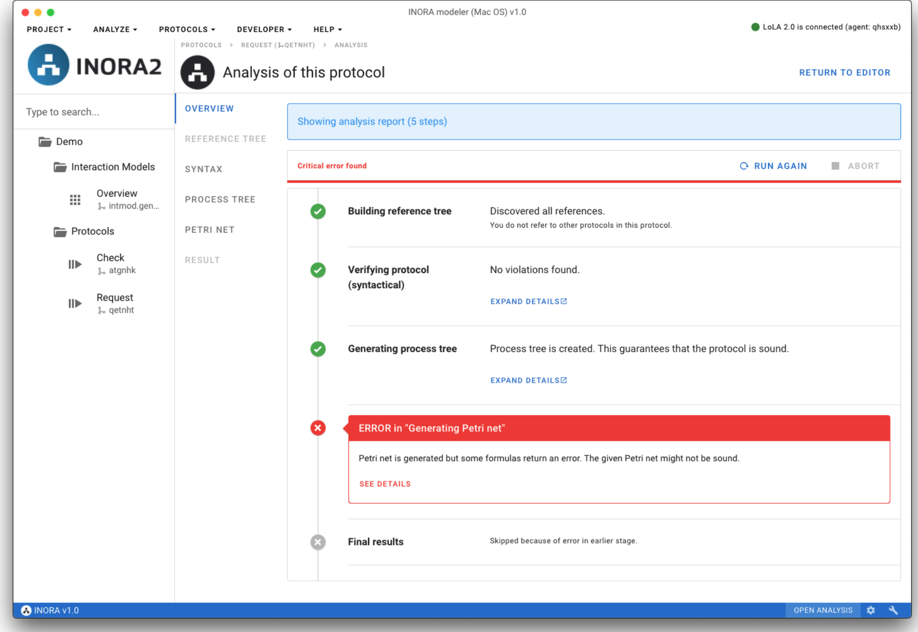



Figure 5.6: The analysis overview of a protocol in the given choreography does not fulfill the criteria of soundness.

**Analysis steps**

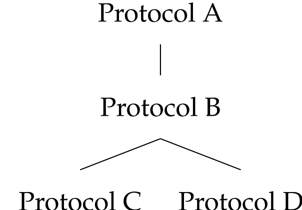
The analysis consists of multiple steps which are executed in a serial manner: they await the previous step before starting its own. Each step is part of the analysis and checks for a single thing. The main view of the analysis is the stepper overview. The user can quickly see the status and outcome of each analysis step, before opening up the details window per step.

The steps included in this version of INORA2 are:

1. **Build reference tree**
   Protocols may refer to other protocols. The first step of the analysis is therefore to discover all (indirectly) referred protocols.

   Due to the conditions stated in section 4.1 we know a protocol can only be sound *iff* all of its referenced protocols are sound. It is not necessary nor desired to substitute all protocols in the model, as this can lead to more complexity in the analysis. Instead, it is possible run the analysis on all protocols that are referred from this protocol recursively using a tree from the discovery. Starting with the "bottom" protocols (protocols that do not refer other protocols, and thus can be checked by themselves), moving up a branch until reaching the current protocol under analysis.

   The reference tree of Protocol A in Figure 5.7 looks like:

   This step would run an analysis on Protocol C and D, and if both are considered sound it will continue to B. If the referenced protocols $\{B, C, D\}$ are all sound, this step succeeds.

2. **Syntax verification**
   The analysis checks whether the syntax of the protocol is in line with the generic constraints listed in section 4.1.

3. **Generating choreography tree**
   INORA2 checks whether it can generate a valid choreography tree using the rules specified in section 4.3, and using the algorithms in section 5.2.

   This can either fail, after which a warning is thrown and the next step is executed. It might also succeed, which allows the user to open up a detailed window to explore and manually analyze the choreography tree. In this case, the protocol should be correct by construction.

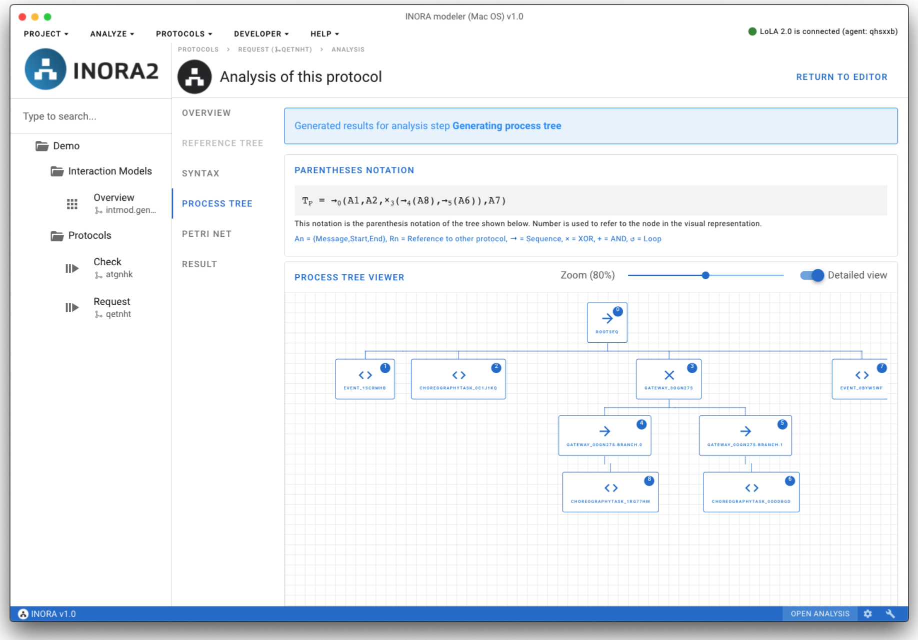



Figure 5.7: Process tree details window in INORA2.

4. **Generating Petri net**
   This step can be divided into two sub-steps:

   - **Generation**
     Contrary to generating a choreography tree, this transformation should always succeed if the generic constraints have been respected. It uses the techniques discussed in section 4.2 and implements it using an algorithm described in 5.2. The user can see and investigate the Petri net in the details window as seen in Figure 5.8

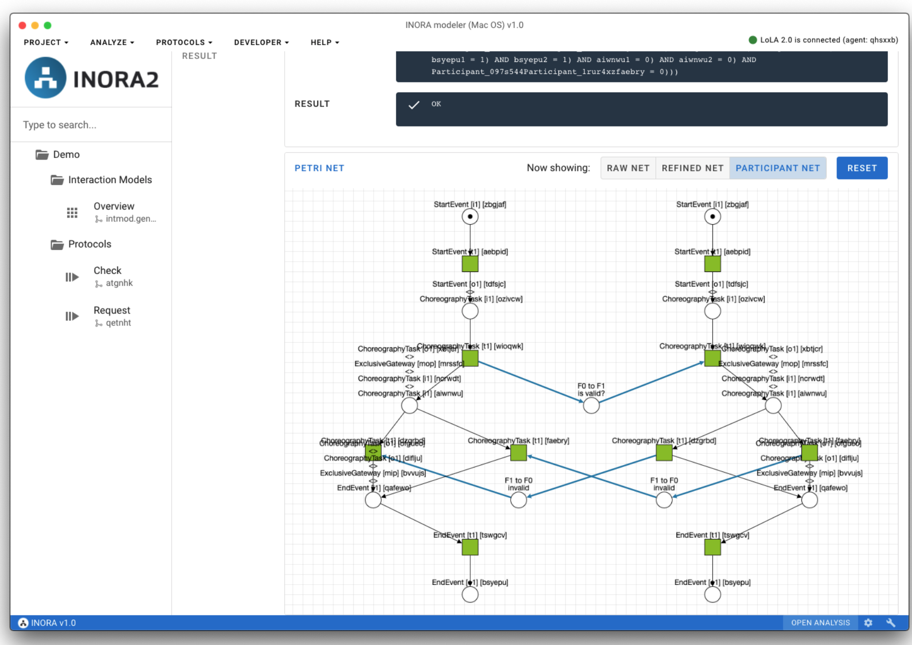



Figure 5.8: Petri net details window in INORA2.

Another powerful INORA2 feature is to investigate a Petri net error within the Petri net UI. If an error is found, and a counter-sequence can be generated, INORA offers the user a sequential scenario in which such an error can occur. It than shows it on the Petri net itself, and provides a UI to click through the sequence, as can be seen in Figure 5.9.

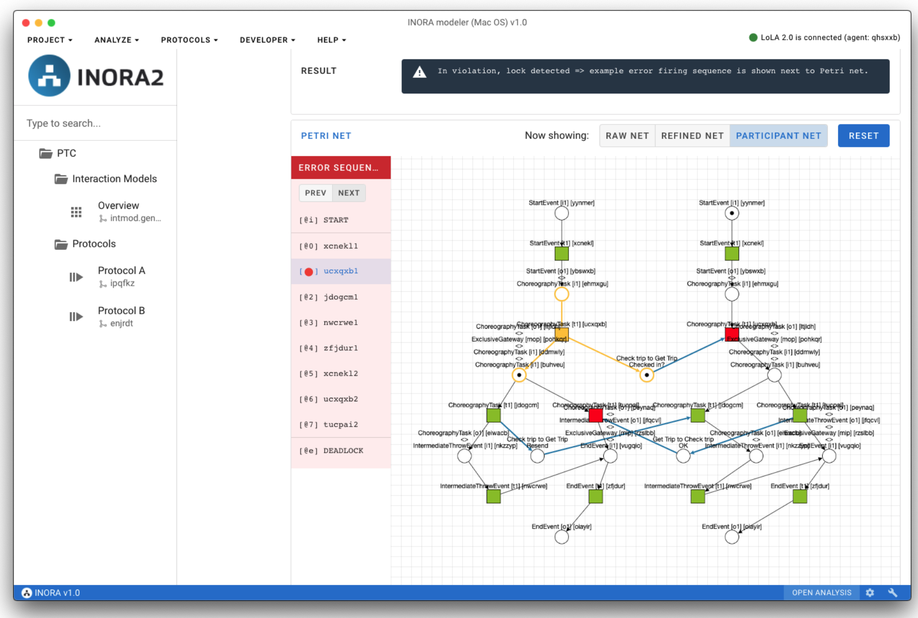



Figure 5.9: Clicking through an error sequence to show how the protocol can potentially end up in a deadlock. The yellow marking indicates the nodes and arcs involved with that transition firing, and the red marking shows that a transition is partially active i.e. has at least one but not all tokens necessary to fire in its consumer places.

- **Validation**
  After the generation is done, INORA2 looks in its internal state whether the user has connected a valid DAME LoLA client. DAME LoLA is an external tool that can verify a Petri net, and it is described in section 6. If the user has connected a valid client, the Petri net along with a checking formula is sent to DAME LoLA. It awaits its response and shows either an error or success in the details window. If the Petri net is not sound, an example trace is given to the user.

  INORA verifies the net by running a LoLA formula (CTL [75]) that checks for: **always eventually all** tokens should be in the end places (output place of end event): given $P$ being all the places in the net and $P_\Omega \subset P$ all places that are end-event outputs, the formula checked for is:

  $$AG(EF(\forall p \in P_\Omega \ : \ m(p) = 1 \ \wedge \ \forall p \in P \setminus P_\Omega \ : \ m(p) = 0))$$

  For example, if the sets $P = \{p_1, p_2, p_3, p_4, p_5, p_6\}$ and $P_\Omega = \{p_5, p_6\}$ are used, the resulting formula in its explicit state is:

  ```
  AG(EF(p1=0 AND p2=0 AND p3=0 AND p4=0 AND p5=1 AND p6=1))
  ```

  This guarantees the soundness property of **weak termination**.

  If no DAME LoLA client is connected at all, this analysis step cannot check for the validity of the protocol. The Petri net translation is still made, and the user can view it, but the correctness can solely be derived from the (not always applicable) choreography tree generation.

- **Final result**
  To summarize all steps, a single outcome is given to the user: the choreography is sound / not sound. The decision table as depicted below is used to derive this final outcome. Note that the decision was made to display an elaborate "analysis report" as the final result of the analysis. In section 3.4 we discussed the mapping of feedback to an easily understandable model context, as modelers might not be an expert on formal feedback. We found, however,

that giving concise and temporal feedback is a complex research field. Therefore it is decided to not include it in the scope of this thesis. Two of three guidelines laid out in section 3.4 have been fulfilled. An example is using red for critical errors, yellow for non-critical warnings, and green for a successful result.

| **Correct references** | No | Yes | | | | |
|---|---|---|---|---|---|---|
| **Valid syntax** | 3** | Incorrect | Correct | | | |
| **Process Tree** | | 2** | Can be generated | Cannot be generated | | |
| **Petri net** | | | * | No DAME LoLA client | Is correct | Is incorrect |
| **Outcome** | **Composition Error** | **Syntax Error** | **Sound choreo** | **Undecisive** | **Sound choreo** | **Not sound** |

Table 5.1: Decision table to conclude correctness of a choreography.

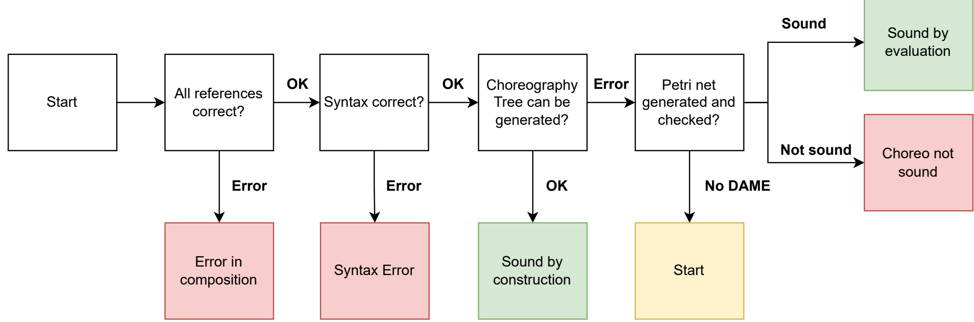



Figure 5.10: Visual flow derived from the Decision table

Due to the constraint definitions and theory defined in chapters 3 and 4 the outcome can be *sound* when either the Petri net evaluation of choreography tree generation succeeds.

## 5.2 Algorithms

The tool uses multiple algorithms to perform analytics and to offer supportive tools to work with interaction models and choreographies. In this section, two major algorithms are discussed: transforming the choreography to a choreography tree and translating the choreography to a Petri net.

### Data Format

Before we can tackle how to transform a choreography into a given output, it is important to understand how the modeler stores a choreography. In INORA we make use of an adaptation of `chor-js` [74] to model choreographies. For visualization, these are stored in XML notation. However, as INORA is written in a front-end web framework, it is useful to deal with these visual objects as actual (JavaScript) objects. The connecting bridge between XML and run-time elements is a tool called `bpmn-moddle`[1] (we refer this as Moddle from now on).

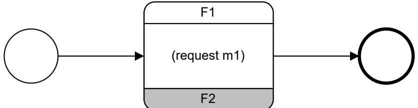
Moddle takes as input either XML or a list of run-time elements, then verifies and translates these to their counterpart. The run-time elements are the nodes and edges (called "flows"). In INORA we are only interested in the nodes themselves, as the flows can be implicitly derived from their attributes.

**Definition 5.1**: Two-way translation of visual (XML) to run-time elements.

---

Consider the following simplified protocol.

[1]https://github.com/bpmn-io/bpmn-moddle




This is the visual representation of an XML diagram. The actual XML code is omitted as the end-user never sees it, and it is not used by INORA (with the exception of storage purposes).

The diagram contains 3 nodes:

*StartEvent*, *ChoreographyTask* and *EndEvent*

The result is an array with three Moddle elements.

```
result = [ModdleElement, ModdleElement, ModdleElement]
```

**Code snippet 5.1:** Result of calling Moddle on XML diagram.

Each `ModdleElement` has the following attributes:

```
ModdleElement = {
  $type: string, // What BPMN event? e.g. StartEvent
  id: string,
  name: string, // Name of the element in diagram
  incoming: [ModdleElement], // List of references (in flow)
  outgoing: [ModdleElement], // List of references (out flow)
  ...
}
```

**Code snippet 5.2:** Attributes of a Moddle element

In each of the sections below, the list of Moddle elements as seen in 5.1 is used. The start element can be found by selecting on the `$type` attribute and the diagram can be traversed with the `incoming` and `outgoing` references.

## Generating a Choreography Tree

The first major step in choreography analysis is generating a choreography tree to check for correctness by construction. For a process tree to form, there is a strict set of conditions that need to hold. The algorithm is denoted in pseudo-code in snippet B.1.

Before this translation is fired the analysis stepper has already checked for basic constraints. However, nothing can be said about the structure of the choreography before the actual algorithm is launched. This algorithm, therefore, performs both checking and construction tasks simultaneously and may fail in doing so.
The algorithm is a set of four main steps, each with its own sub-steps:

1. **Preprocess the input**

    (a) Filter the ModdleElements to only include nodes and no flows.

(b) Map the list of ModdleElements to a simplified integer graph for quick analysis. This can then be translated in later steps once information on nodes is actually required.

2. **Find the start and stop nodes of all loops in the graph**

(a) Find the strongly connected components (SCC) in the graph [76].

(b) For each SCC, find the single entry node (start of loop) and the single exit node (end a loop). These are stored. If more are found for a single SCC, throw an error.

(c) To check whether the strongly connected component itself has more SCCs in it, take the SCC and perform the SCC check on it recursively.

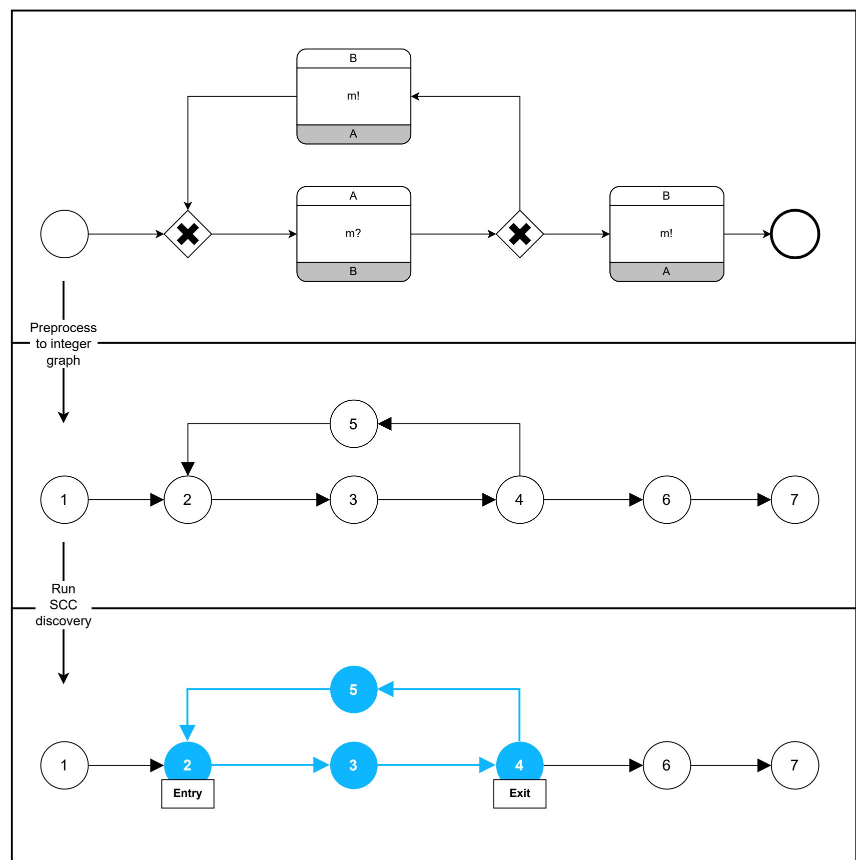



Figure 5.11: Single entry, single exit detection of loops as described in step B.

(d) Return a list of all loop-starting nodes, and loop-ending nodes.

An example can be seen in Figure 5.11.

3. **Check for block structure**

(a) Each forking gateway should be matched with exactly one joining gateway. Assume a walker with two stacks of tokens in it: a red stack for XOR gateways, and a blue stack for AND gateways. The walker would up the corresponding stack with one token for each fork gateway encountered, and lower the corresponding stack with one if a join gateway is encountered. Starting with initial state $S_i = \{R : 0, B : 0\}$ at the start event, after each possible path the state at the end event should equal $S_i$, $\forall p \in P : S_i = S_{ep}$.

(b) If block-structured, continue; else throw an error, it will not be possible to construct a tree.

4. **Attempt to construct choreography tree**

(a) Traverse the choreography from the StartEvent using a depth-first method.

(b) If the node is already visited, continue Mark node as visited

(c) For each node

- Create a choreography tree node if it is an 1) activity 2) reference 3) fork node (AND or XOR) 4) start of a loop. If possible children may be appended to this node
- Pop the parent stack if join node (future nodes in the path trace will use the previous parent).
- Append newly created node to the children of its current parent.
- If this is a loop exit, also directly append the back-activity to the loop and mark it as visited.
- If this is a loop, sequence or fork node, add itself to the parent stack. Future nodes in the path trace will be appended as a child to this node.
- Add its children to the DFS stack.

(d) Return constructed tree.

The pseudo-code algorithm can be found in Appendix B.

### Transformation to Petri net

As stated the previous section, the algorithm for building a tree may fail, which is considered acceptable behavior. The analysis then falls back on the computational heavier transformation to a Petri net. This is done using the algorithm described in this section.

1. **Preprocess the input**
   The algorithm uses the same ModdleElement input as used in the choreography tree algorithm. No further preprocessing is needed as element attributes are used consistently in the algorithm.

2. **Use Depth First Traversal**
   As each choreography node may be translated into multiple Petri net elements (e.g. a start event consists of two places and a transition, connected by two flows.), it is needed to track the IDS of these newly constructed elements and to which choreography element they belong. Use a depth-first manner to walk over the choreography, and use a *naive* translation. This means that we do not use any logical checks, and simply translate each part into its corresponding Petri net structure. Start with the StartEvent and:

   (a) Check if we have already visited this element. If so, it might still have open input places, which need to be connected. Example: a join XOR node might need to merge two paths into a single node. We may have already discovered and created this node for *path1* and gave it two input places. Now we again encounter it in *path2*. Therefore we create a flow to its one remaining input possibility. Then continue to the next node.

   (b) Using the type of the ModdleElement, find the corresponding generic Petri net mapping. A list of mappings used by INORA2 can be found in Appendix D, these are the JSON notation of the formal notation given in section 4.2. For example, the Parallel fork gateway is translated using the segment:

```
{
    translationSegment: 'ParallelGateway_fork',
    content: {
        places: [
            'i1',
            'on'
        ],
        transitions: [
            't1'
        ],
        edges: [
            'i1>t1',
            't1>on'
        ]
    }
}
```

**Code snippet 5.3:** Translation segment used for Petri net translation in INORA2.

The algorithm uses the segment to calculate how many places and transitions need to be created. Assume a parallel fork gateway with two outgoing branches. It would create

- 3 places: $i1, o1, o2$
- 1 transition: $t1$
- 3 flows: $(i1 \rightarrow t1), (t1 \rightarrow o1), (t1 \rightarrow o2)$

It has $|\{i1\}| = 1$ input place which is used to connect the item this current iteration came from (trace), and $|\{o1, o2\}| = 2$ possible output places. This information is stored in a global tracking store, that keeps track for each ModdleElement the list of input and output places that are still open, so they can be connected in later iterations (see step (a)).

(c) All children are pushed to the depth-first stack, and the next iteration is started.

3. **The resulting net is refined**
After the depth-first traversal has reached completion, the result is a place-transition net in which multiple places are directly connected to each other. This is in violation with Petri net rules, and therefore directly sequential places need to be merged (as shown in 4.3), so that a valid Petri net is created.

4. **Duplicate and connect**

(a) Count all unique participants mentioned in the choreography.

(b) Copy the Petri net over for each participant, and track to which participant this belongs.

Example: a choreography modeling a simple interaction between components $A$ and $B$ needs to be copied two times.

(c) For each copied net $c_d$ (with $d$ being the dominant participant e.g. copy for both $A$ and $B$ result in $N = \{c_A, c_B\}$), find all the interaction transitions in which the *dominant component* is the initiator.

(d) For each, create a connection place and create flows between the initiating net and the same transition in the receiving net.

5. **The participant net is returned**

The pseudo-code algorithm can be found in Appendix C.

## 5.3 Documentation, licensing and installation

INORA2 is created as an educational tool. It is open source and distributed under the MIT license [2]. To make maintenance, extension, and forking easy, the tool is accompanied by documentation within the Git repository under `/documentation/*`.

The tool can be downloaded from the official Utrecht University Gitlab:

`https://git.science.uu.nl/interaction-oriented-architecture/inora-open-source-modeler`

To use the pre-compiled version, host the contents of the `/inora/dist/` directory on any webserver (e.g. Apache[3] or Nginx[4]). Alternatively, use the `/inora/dist_electron/` to directly run the Windows and Linux executable files. We have not provided a Mac OS executable, as an Apple Developer account was needed to sign the application.

The developer versions can be launched by running `npm` or `docker`. The documentation provides further instructions to launch and maintain INORA2.

[2] https://choosealicense.com/licenses/mit/
[3] https://httpd.apache.org/
[4] https://www.nginx.com/

## 5.4 Version Comparison

Whereas the original INORA project was created as a plugin for Eclipse and Obeo Studio using the Sirius project, INORA2 is a standalone tool that is completely rewritten.

The table below covers the main differences between the two versions.

| | Version 1 | Version 2 |
|---|---|---|
| **Frame-work** | - Sirius plugin<br>- Java | - Standalone<br>- TypeScript |
| **Prere-quisites** | - Installed Obeo Community Studio or Eclipse<br>- Install using Sirius<br>- Java<br>- Knowledge of Sirius modeling | - Knowledge of BPMN modeling |
| **Ease of use** | - Installation requires multiple steps<br>- Running multiple instances of Obeo Studio, with different projects running<br>- Need to understand the complex GUI of Sirius<br>- Can use the Sirius UI to make use of the full set of modeling features. | - No installation necessary, simply open up the compiled application.<br>- Simple GUI covering the needed interactions<br>- No graphical way to achieve complex connections or use plugins |
| **Features** | - Modeling interaction model<br>- Modeling choreography<br>- Basic verification<br>- Basic model management | - Modeling interaction model with clean and single-purpose UI<br>- Modeling choreography with clean and single-purpose UI<br>- Extensive evaluation: reference tree, syntax, soundness of protocol using constructing and Petri net tools<br>- Ability to do extensive analytics on Petri net using external LoLA tool<br>- Simplified and integrated model management<br>- Integrated viewers for Petri net and Choreography trees. |
| **Main-tenance** | - Steep learning curve and complex set-up to start working with the plugins source.<br>- Can use industry standards once set-up.<br>- Always need to run big software packages for maintenance.<br>- Little to no documentation.<br>- Works on all OS that can run Obeo / Eclipse. | - Need to know TypeScript and Vue + Vuetify + Vuex.<br>- Does not use a standard layout for the application source.<br>- Documentation is provided.<br>- Compiles to all OS that can run Electron applications.<br>- Modular<br>- Uses a lot of external dependencies. |

## 5.5 Summary

This chapter describes the implementation of the techniques introduced in the previous chapter. We introduce INORA2, the successor to INORA. This is a standalone application distributed as an educational tool to support software architecture modeling and analysis. The layout and features of the application are explained and the two major algorithms used and their inputs are described in-depth, followed by a set of examples. Furthermore, we compare both versions of INORA and discuss the license and documentation.

# Chapter 6

# Petri net model checking

To perform analysis on the Petri nets generated by INORA2, we can use already existing and heavily optimized tools. This chapter introduces such a tool and discusses the modifications, a newly created API, and containerization of a Petri net tool called LoLA 2.0. The modified version is called DAME LoLA. This forked project of the tool can be used by the INORA2 client to perform Petri net checks.

## 6.1 Model

The *INORA2 Modeler* and *DAME LoLA* tools are two separate tools written in distinct languages, but can be seen as a software package. The *INORA2 Modeler* can be installed on any local computer and can be connected to the *DAME LoLA* tool via an exposed endpoint. It is recommended to run *DAME LoLA* on a server as it may take a lot of computer power and resources to verify large Petri nets. We made the decision to decouple the two tools and not integrate them. The main reason for this is DAME LoLA being a versatile forked project for LoLA 2.0 that can be used in a number of contexts other than supplying INORA2 with validation. To make use of the full power of INORA it is recommended to connect to DAME LoLA. The interaction model of the components in both tools can be seen as a single system, which is shown in Figure 6.1.

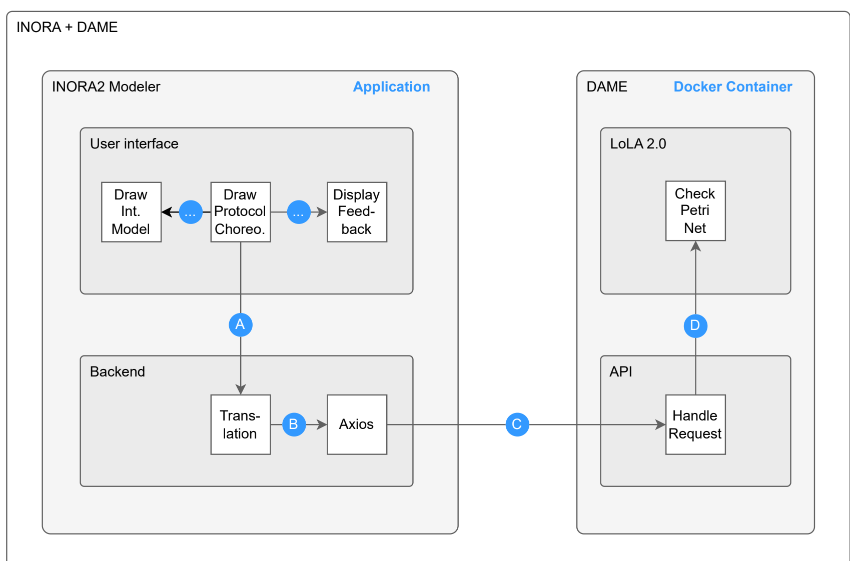



Figure 6.1: Interaction model for the INORA2 + DAME LoLA.

The interaction model displays interaction protocols $A, B, C$ and $D$, which are defined as followed:

- **Protocol A:** A user-constructed choreography in the user interface is sent to the back-end of the application to be translated to a Petri net.

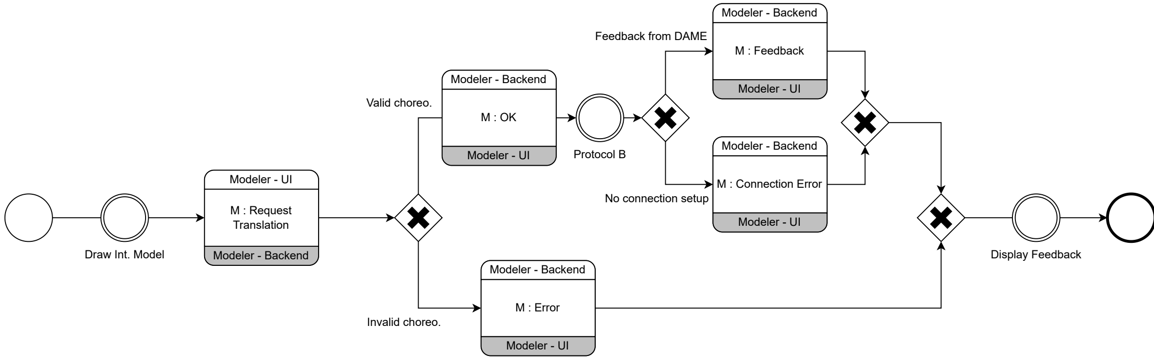



Figure 6.2: Protocol A definition.

- **Protocol B:** The translated Petri net is sent to the function that constructs and receives external calls (axios).

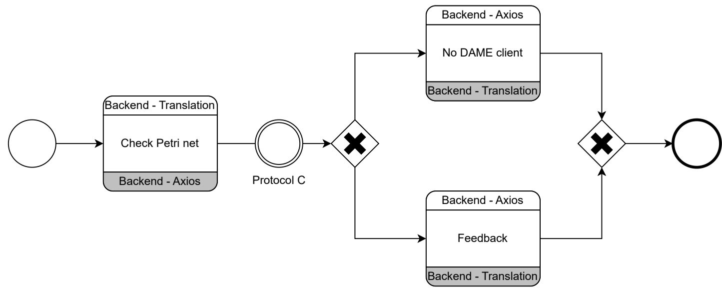



Figure 6.3: Protocol B definition.

- **Protocol C:** Axios sends a request and handles the HTTP responses and REST construction and extraction.

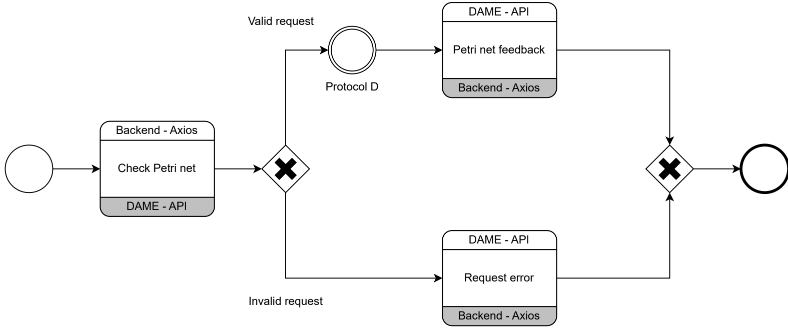



Figure 6.4: Protocol C definition.

- **Protocol D:** After receiving a request, some pre-processing is done, such as creating temporary directories and checking the syntax. The API then initiates one or more property checks on the compiled LoLA 2 tool via this protocol, and returns it via Protocol C.

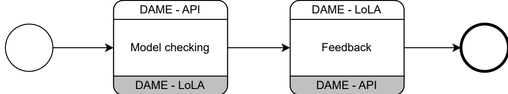



Figure 6.5: Protocol D definition.

## 6.2 DAME LoLA

Next to the INORA2 modeler, we create an additional tool called DAME LoLA. As we outsource the actual checking of Petri nets to a more optimized industry-standard tool, an interface needs to be created. Therefore the following requirements need to be achieved:

1. Compile the INORA2 source[1] code to an executable.
2. Move INORA to a container so that it can be used on any system without the hassle of installing all necessary dependencies manually.
3. Write an easily accessible interface to communicate with LoLA from a network.

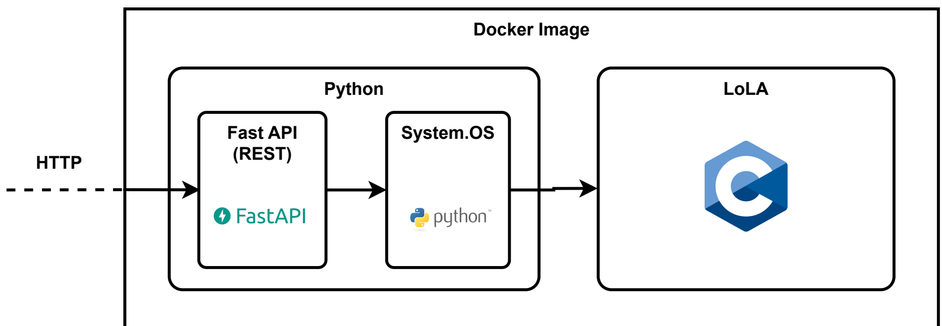



Figure 6.6: Basic DAME overview.

These requirements are implemented in an interface tool called DAME LoLA. It is open-source and has the advantage that it can be installed quickly, can be maintained with basic Python knowledge and is stable on any OS. The tool, its use, advantages and its maintainability are described extensively explained in Appendix E.

## 6.3 Summary

The checking of the Petri nets is outsourced to an optimized industry-standard tool called LoLA 2.0. A side product of this research in a containerized version of LoLA 2.0 called DAME LoLA. This chapter describes the interaction between INORA2 and DAME LoLA. Furthermore, DAME LoLA is explained in-depth in Appendix E.

[1]https://theo.informatik.uni-rostock.de/theo-forschung/tools/lola/

# Chapter 7

# Application walk-through

Using the running example, we will model a range of protocols and the interaction model using INORA2. The aim of this chapter is to demonstrate the use of INORA.

1. **Start up the application**.
   INORA can be started in three different ways:
   - Using `npm run serve` to start the development environment using npm[1]. This requires the installation of a bunch of dependencies and is slow, but allows for quick modification of the source code.
   - Using the docker image to run a container.
   - Using the native OS Electron compiled distribution to quickly open an optimized production edition.

2. **Either create a new project, or open an existing one**
   For this walk-through, we will create a new project called "PTC".

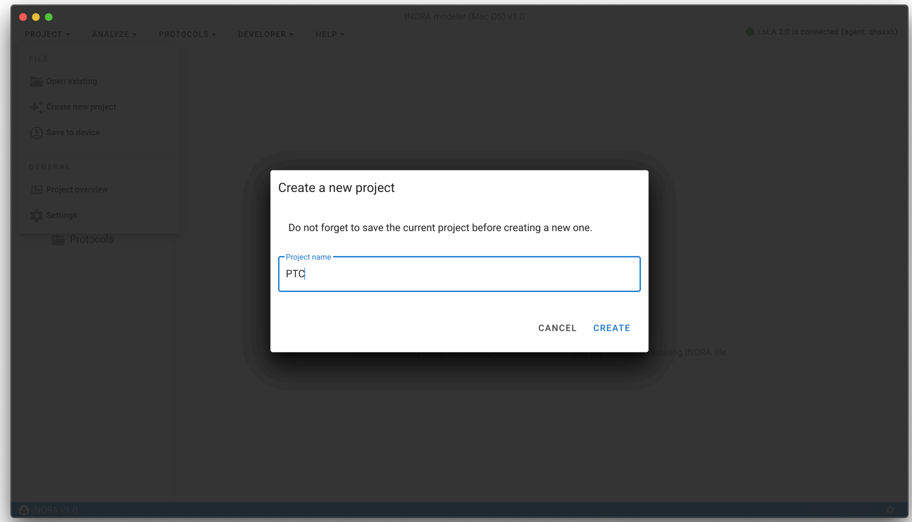



Figure 7.1: Create a new project, and name it.

3. **Start by populating the Interaction Model**.
   This is the key diagram in the project. Populate it with some initial functions, or draw out the full diagram. It is possible to later adjust this diagram, but drawing arcs between functions and naming them is the easiest way to create a choreography.

[1] https://www.npmjs.com/

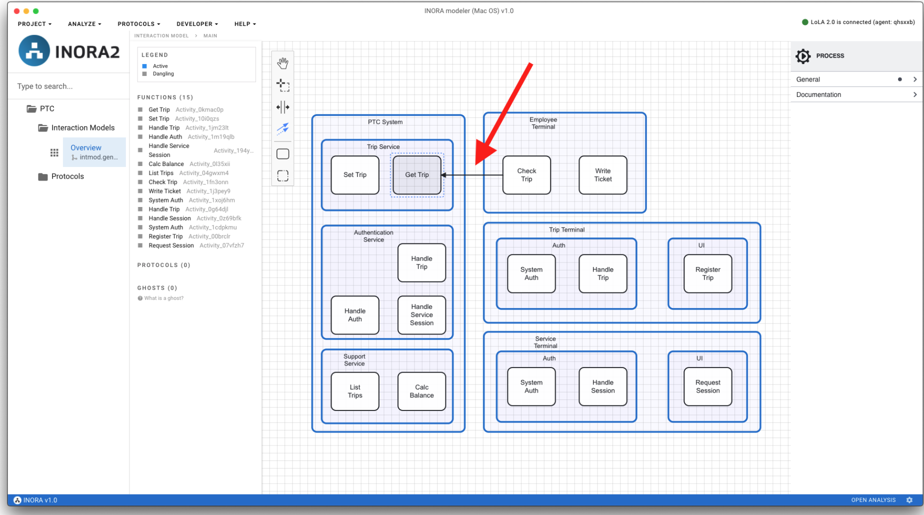



Figure 7.2: Draw the required functions, and start connecting them with protocol arcs. They can easily be connected using the connection tool in the left modeling palette.

4. **Create a new protocol.**
   The drawn arcs result in new protocol suggestions in the left overview bar. You can easily create a new protocol from this pane by clicking on "NEW".

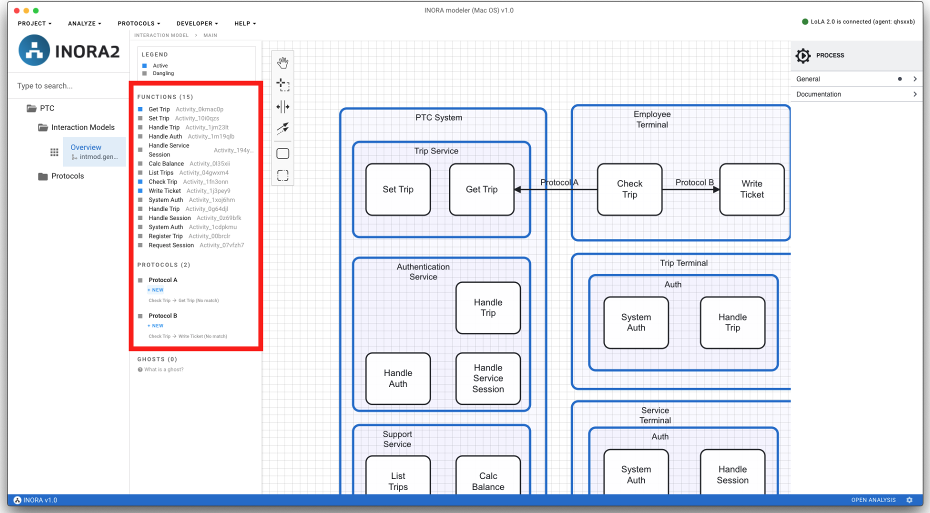



Figure 7.3: The overview panel and protocol creation button. You can also manage existing protocols and see the status of individual functions/protocols in this pane.

5. **A new empty protocol with a basic template is created.**

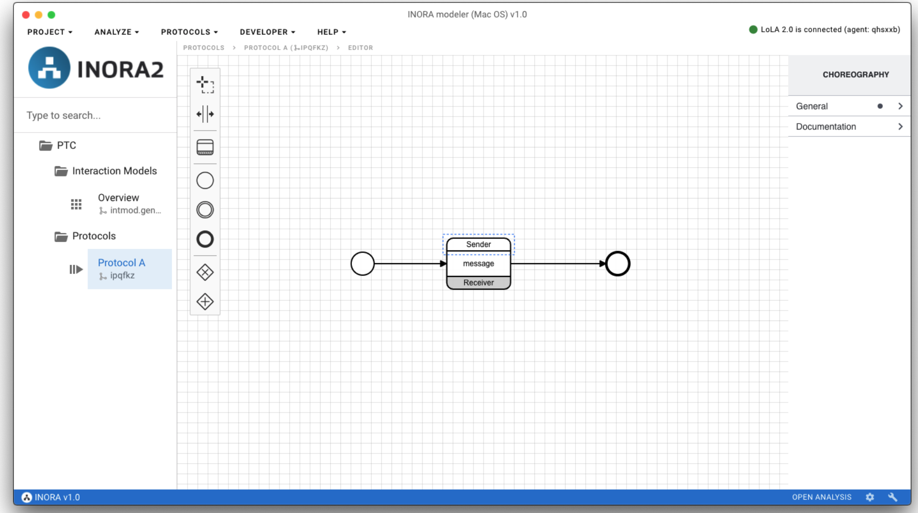



Figure 7.4: The basic template of protocol A is shown. The interaction does not yet include any functions from the interaction model.

6. **Link the participants from the interaction model.**
   By selecting the interaction activity, and clicking on the database icon in the top right pop-up menu, a list of available participants appears.

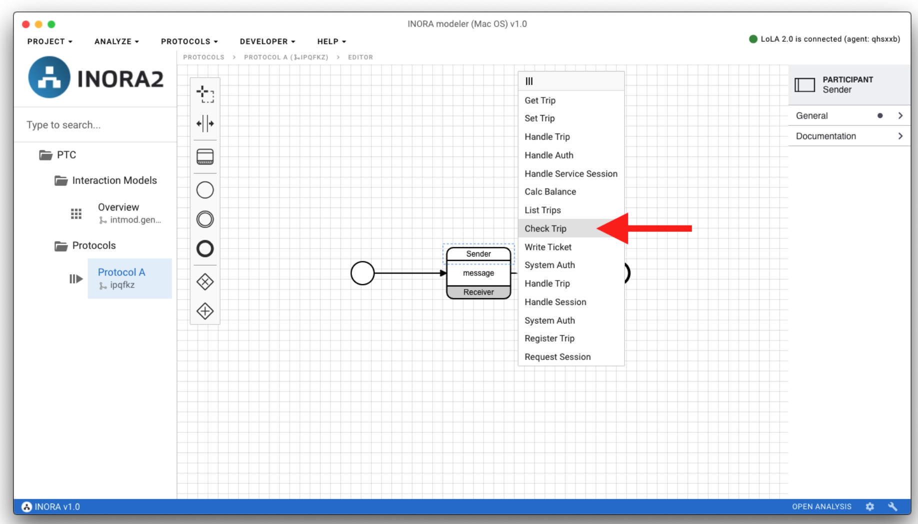



Figure 7.5: We click on the *Check Trip* participant as the initiator for the first message.

7. **Model out the full protocol A**
   Let's create a logical flow candidate for protocol A.

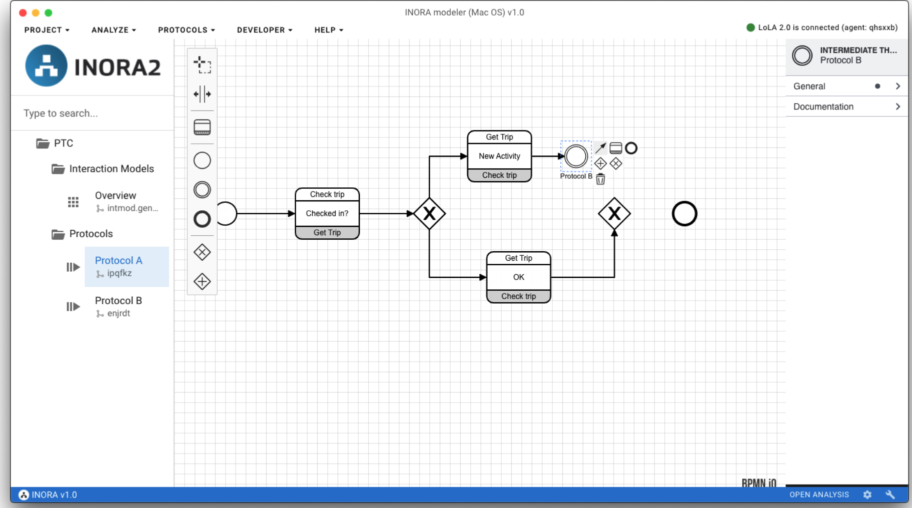



Figure 7.6: The protocol can easily be modeled using the available tools. They are designed especially for choreography modeling and have some useful features such as element suggestion, automatically flipping initiator and receiver after modeling the reverse, quick labeling and renaming, grid snapping, and auto-reconnecting after deletion to name a few examples.

8. **You can add some documentation to each element.**

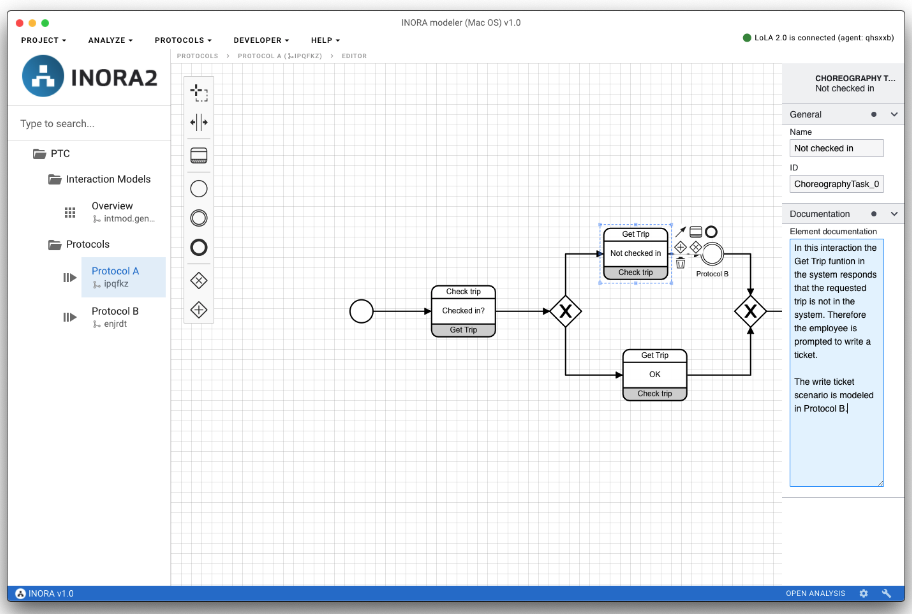



Figure 7.7: Using the right-side pane you can easily rename, alter the ID or add documentation to each element. It is recommended to describe what exactly data may look like, or if certain conditions need to hold.

9. **Our protocol is done, let's create the references.**
   We reference protocol B. To have a useful analysis of the composition $A^c = \{A, B\}$, we need to create Protocol B. Let's keep this one simple.

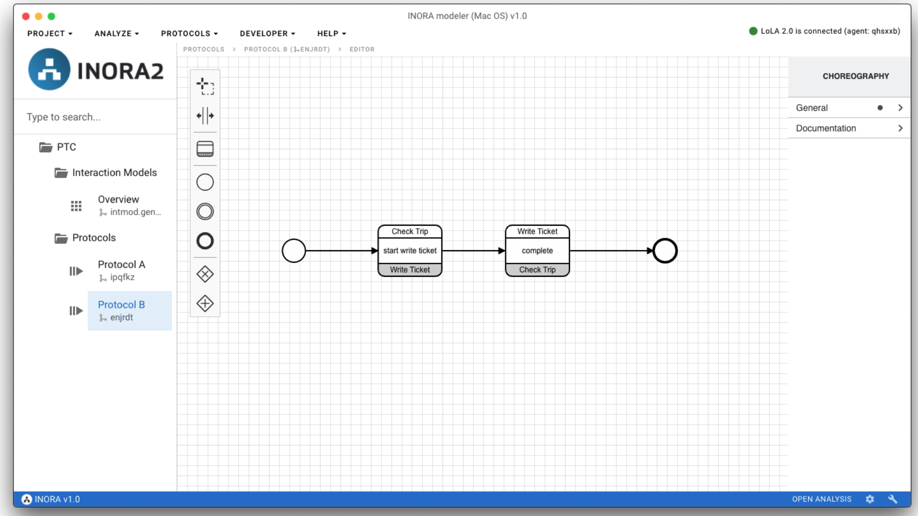



Figure 7.8: Create a simple protocol B with is called by A.

10. **Setup DAME LoLA**
To make use of the Petri net analysis, we need to set up a connection with a DAME LoLA client running LoLA (as explained in the previous chapter). If you have a DAME LoLA instance running it is extremely simple to connect it to INORA. Go to settings (or click the status in the top right bar), fill in the exposed endpoint and click connect.

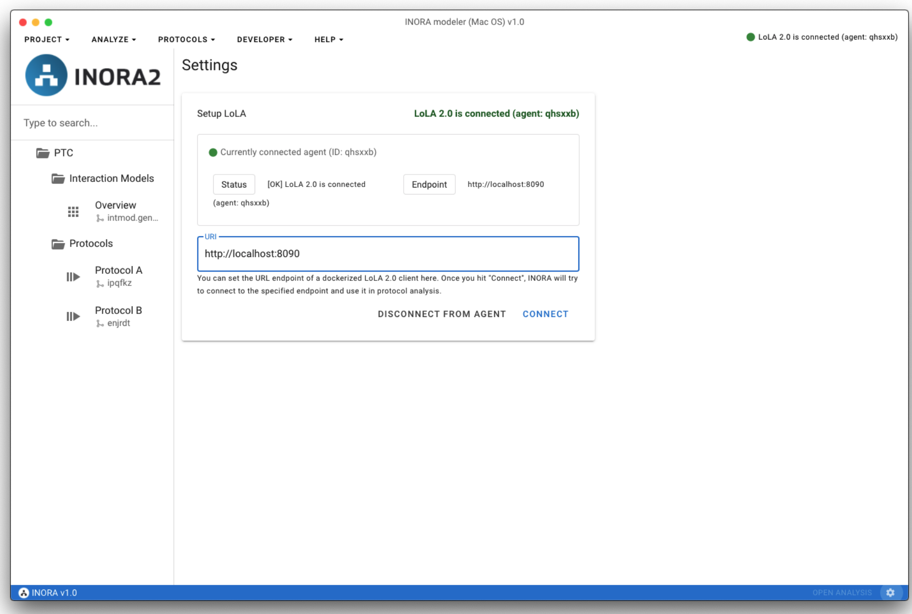



Figure 7.9: Connect to DAME LoLA .

11. **Start the analysis of protocol A.**
We can ask if this protocol is sound (and thus realizable) by running the analysis.

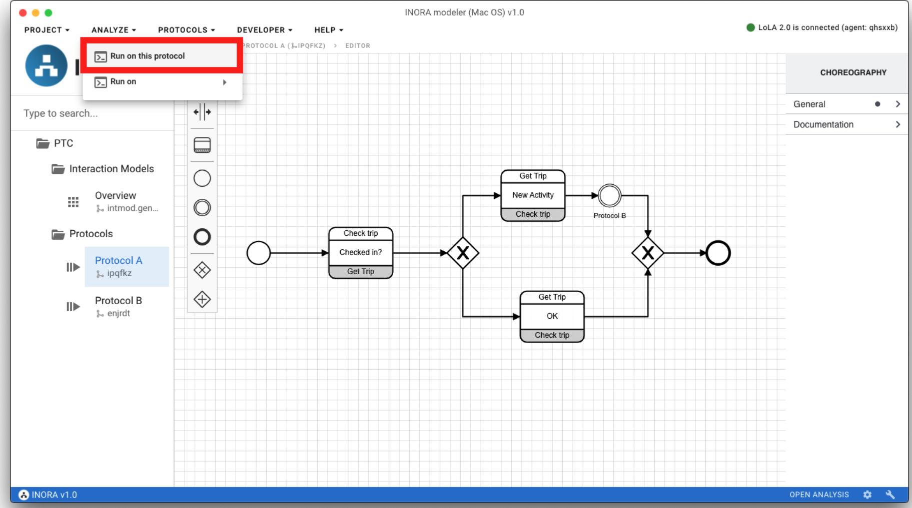



Figure 7.10: Click on the analysis menu tab, or use the button in the lower right corner to run the analysis module.

12. **Inspect the analysis overview**
    This is the main page of the analysis. A user can see the progress and results (when done) of each step in the analysis timeline.

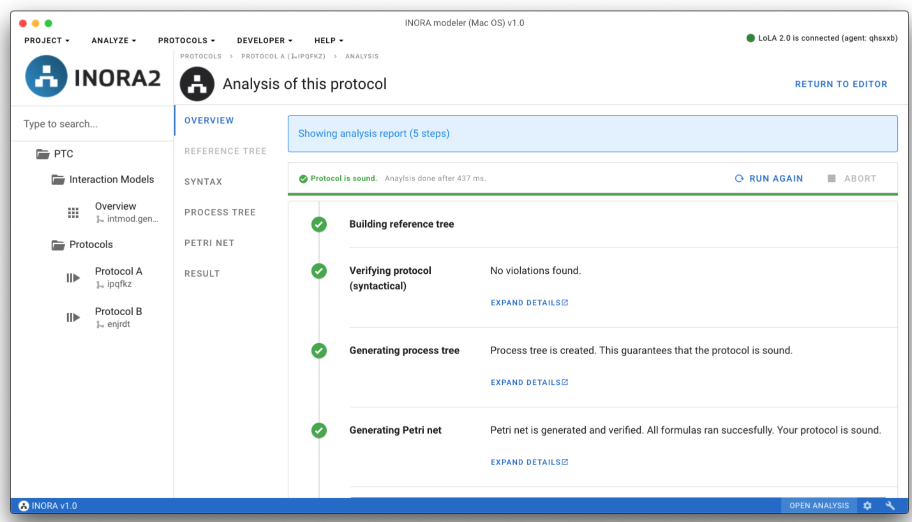



Figure 7.11: Correct protocol. All steps are green.

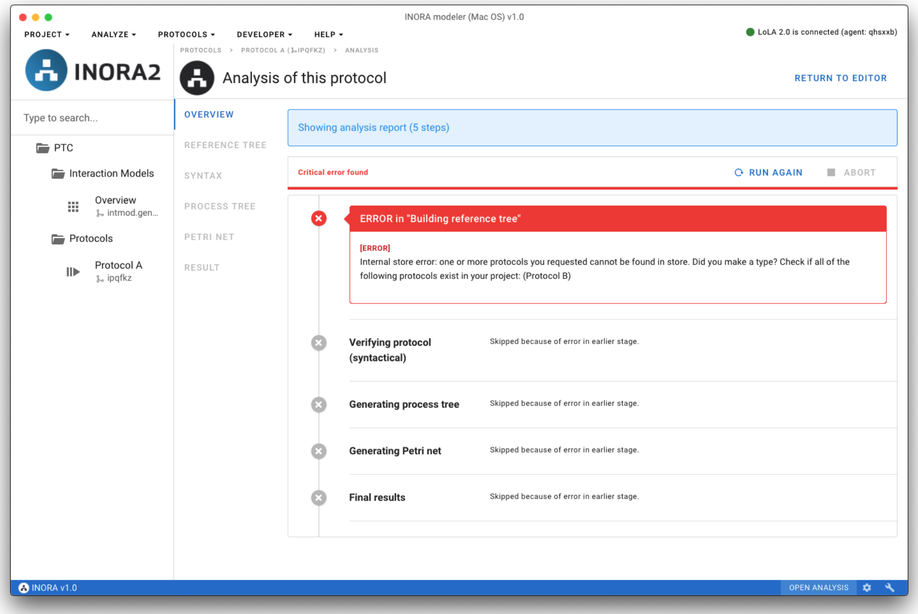



Figure 7.12: This protocol references a non-existing protocol.

13. **Open up the details of each step by clicking the button or the menu.**
    The interesting steps of course are the ones with an error. "Choreography tree" and "Petri net" also produce interesting visuals to prove that are correct.

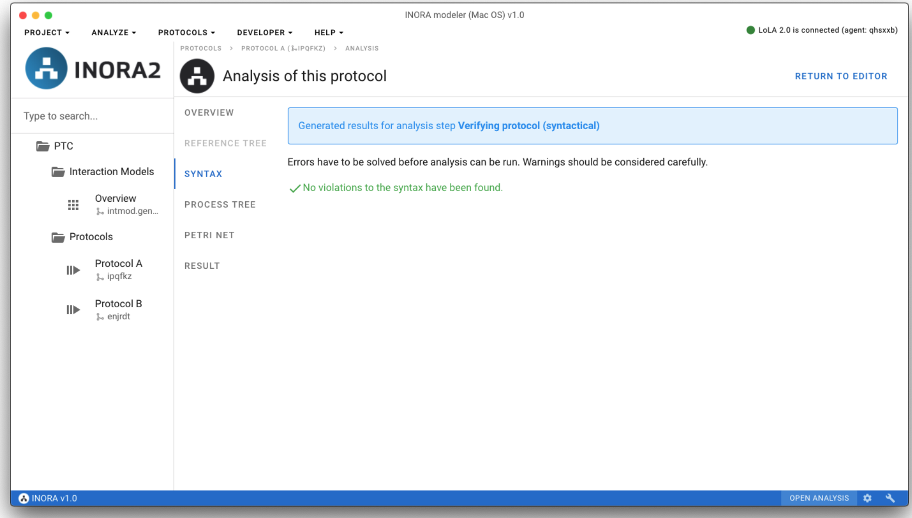



Figure 7.13: Syntax checker details. The syntax is correct.

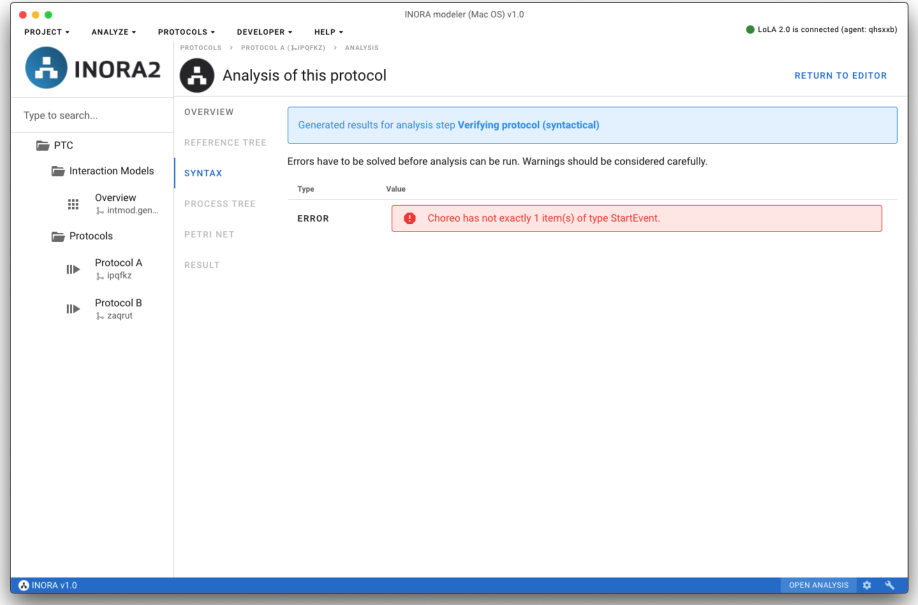



Figure 7.14: Scenario with error) Syntax checker details. The modeler has included too many StartEvents.

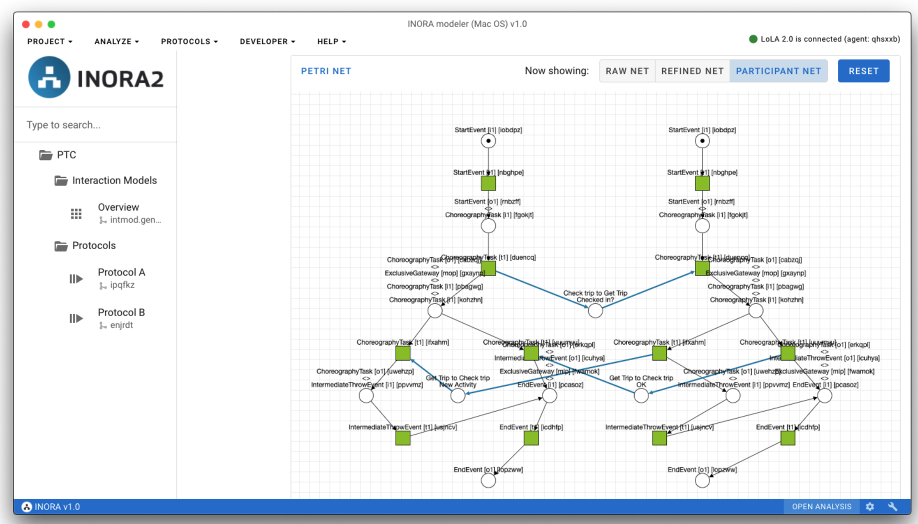



Figure 7.15: Details of the Petri net generation, the Petri net is sound.

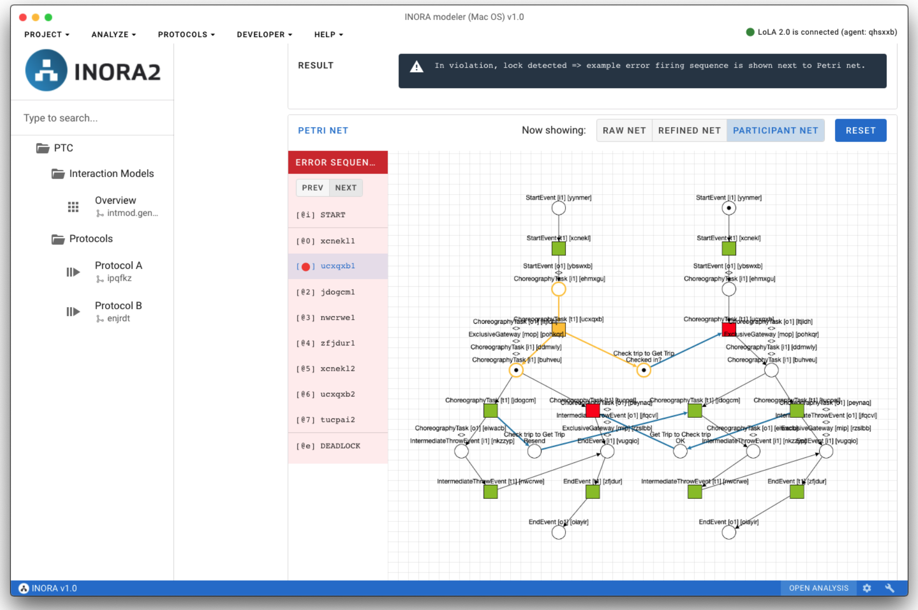



Figure 7.16: (Scenario with error) Details of the Petri net generation, the Petri net is not sound. DAME LoLA has found some errors in the Petri net. The modeler can step through an example counter firing sequence and see the formula that is verified by DAME LoLA.

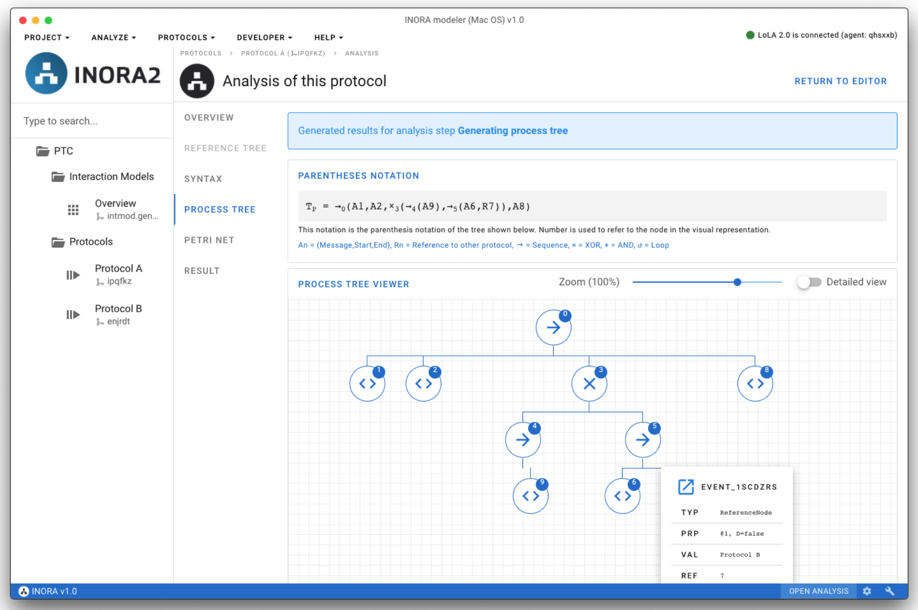



Figure 7.17: Details of the choreography tree generation, the Petri net is sound as the tree can be generated. It shows a detailed tree viewer, and its parenthesis notation for the user to quickly share the protocol in single-line format.

14. **Manage protocol settings**
    If we, for example, were to delete the protocol arc of "Protocol B" in the interaction model, the choreography in INORA2 would become a *Ghost*. This is a choreography that is modeled in INORA2 but is not linked to any arc in the interaction model (since it was deleted). If we were to re-create it, the modeler does not know that choreography B is related to protocol B. The settings of a choreography offer the option to re-connect and rename it.

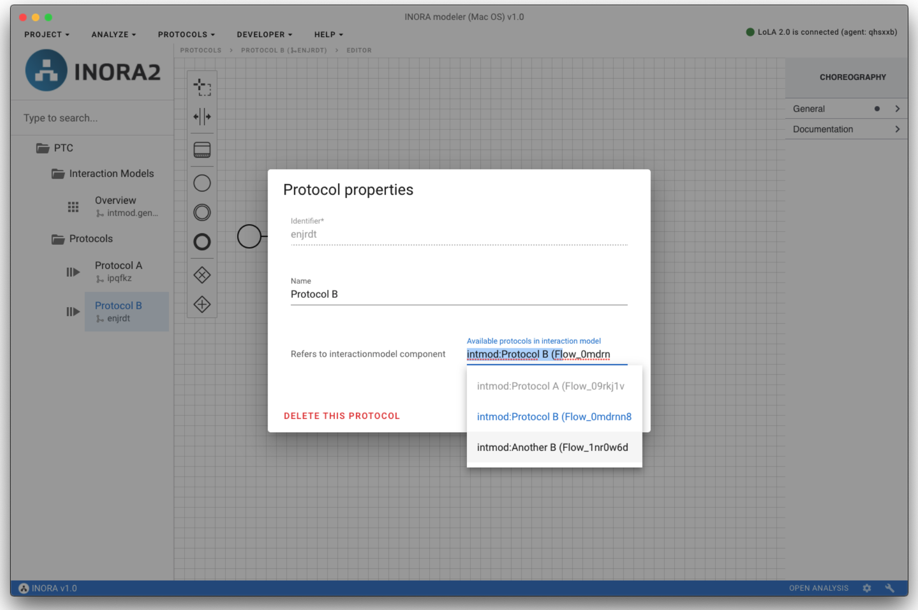



Figure 7.18: Settings of a protocol. Can be opened from the side-panel in the interaction model, or from the protocol itself.

15. **Save the project**
    All changes are always stored upon change directly to the local store. If you want to make a copy, start another project or share the models with other people it is possible to export (save) the project.

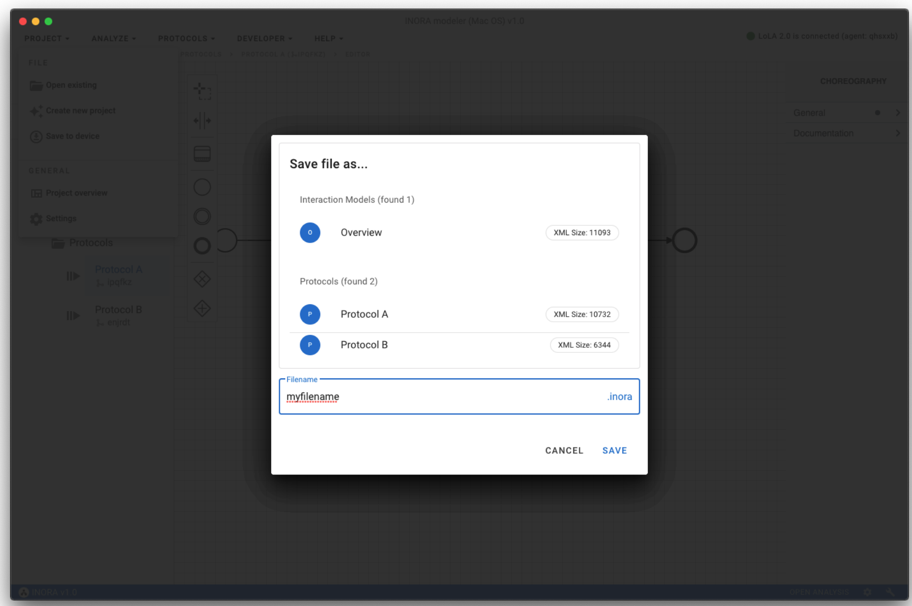



Figure 7.19: Name the project and save it as a .inora file.

# Chapter 8

# Conclusions

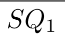
In this chapter, the conclusions regarding the research are drawn and related back to the research aim. Furthermore, the limitations and the future of the research field are discussed.

## 8.1 Answers to Research Questions

The aim of this research is to provide formal semantics and tool support in order to give architects feedback to create sound INORA models, such that the quality of software architecture is improved.

After covering basic assumptions and introducing the general context of interactions between software components and their relations to Petri nets we provide two formal techniques to transform a choreography. The main translation is the formal and refined technique to map a BPMN choreography to a Petri net. The second is showing that correctness can be a result of construction, by showing we can construct a strict-grammar choreography tree. These two techniques are then implemented into a newly created tool called INORA2, which demonstrates that it is feasible to use these techniques in modelers. Types and ways to display this feedback are also discussed and partially implemented in the tool.

The following sub-questions can now be answered.

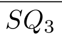


*What is a formal translation of a composed choreography to Petri nets, that allows for verification?*

In chapters 4 and 5 we described formal techniques to generate valid Petri nets from a given choreography. Each segment of a generic choreography is provided with a formal translation using visual, formal and code-like formatting. Using an example, procedure definition, and pseudo-code algorithm the steps to achieve such a translation are covered extensively.

*What techniques can be used to check for the soundness of a Petri net?*

In chapter **??** a variety of Petri net properties and their implications are discussed. In the following chapters, we apply some of these techniques to check for the soundness of generated Petri nets. We also show conditions in which a Petri net does not need to be generated and checked due to correctness by construction.



*How can the formal translation method and the verification techniques best be implemented in a tool so that it returns feedback?*

Chapters 5, 6 and 7 cover the two tools in-depth. INORA2 is a newly developed tool to model, construct and analyze interaction models and protocols. It shows feedback and supports the software architect by implementing the theory discussed in chapter 4. DAME LoLA is a wrapper that allows for easy use of LoLA 2.0. Both are combined to provide the user with extensive and automated feedback on the protocol.

By answering all these sub-questions, we can answer the main research question.

> RQ *How can software architects best be provided with automated feedback on the quality of an Interaction Oriented Software Architecture?*

The outcome of this thesis is a formal framework for translating choreographies to Petri nets and (conditionally) choreography trees. We provide procedures and formal building blocks to do this. These can be used in future research as a starting base for extending model checking and analysis on choreographies and BPMN in general. Furthermore, the theoretical techniques are implemented in a tool that supports analysis and modeling. They can be used to evaluate systems and show problems in the concurrency of interactions; both for educational purposes and in the industry.

## 8.2 Discussion and limitations

This thesis covers a broad section of literature and the inherently complex field of model-checking. Although a lot is covered, we explicitly left some in-depth topics out. There are also assumptions and conditions for our solution that might not always fit practical systems. In this section, we discuss some limitations of this research.

First and foremost, we have shown that a tool can automatically translate choreographies to a Petri net and potentially a Choreography Tree. The checks executed on the Petri net are now limited to checking for weak terminatio: checking if all tokens from the initial marking eventually always end up in all the end places, without any lingering tokens. There is a wide variety of possible model-checking formulas (some of which are already discussed in chapter **??**), and this research does not compare exactly which minimal combination can prove full correctness. It provides a way to quickly supply a very powerful analysis to a choreography but lacks the formal proofs and research to guarantee correctness on all aspects of the net.

Another limitation of the study is the visual representation of the generated feedback. We have achieved a way to automatically translate, verify and display it outcomes in the INORA tool, yet it lacks a way to display the information on the diagram itself. In section 3.4 we discussed literature related to Source2Target models and translating back feedback formal feedback to run-time GUI elements. Although this is partially achieved as a mapping between a Petri net element and a BPMN element is kept, it is not implemented in the INORA2 tool. The main reason for this is that the feedback received is inherently complex to map back. Although LoLA outcomes and our mapping can translate back a trace of sequences that can be run to end up in a deadlock, for example, it is difficult to display this on a static non-temporal view such as a general BPMN model. We did decide to implement such a firing sequence visualizer in the Petri net viewer itself, but due to scope limitations were not able to find a good way to map that back to the modeler itself. Another major obstacle to this would be the heavy computational activities that would need to be run in the background of the tool whilst modeling. This is something INORA2 currently does not support, as we have chosen a non-preemptive feedback viewer, and thus have to activate the analysis module manually.

In chapter 4.3 a language equivalence is assumed to show that a choreography tree can prove correctness by construction. This equivalency has a major "weakness". It assumes that an interaction is an atomic operation: once it happens, nothing can happen during the interaction, and the next "iteration" results in a finished interaction. This viewpoint works fine if you consider a Petri net transition because it is just a transition between two states: before and after the interaction. In the choreography tree, however, to guarantee that the behavior is exactly the same after the equivalence rewrite means that an activity $A$ is instantly completed after making a decision. This is of course not the case and thus is this equivalency considered a simplified view.

The fourth point of discussion in this research is that we have proven the feasibility of automated model-checking in interaction models and their protocols, yet have not applied any non-formal evaluation. Though the steps, translations, and evaluations seem correct, we have not conducted a case study or consulted professionals for their opinions. Therefore there is a lot of room to further validate the actual usefulness of this approach itself: both in terms of model language, types of analysis, and ease of use of INORA2 itself.

The two tools developed (INORA2 and DAME LoLA) are released as educational tools. Although they are developed with usability in mind, it is not professionally developed software. The research aims to prove feasibility and therefore offers a proof of concept tool, yet is not ready to be used in commercial applications. Mostly due to a lot of re-scoping certain supportive parts of the software are underdeveloped; the main focus of the research was to create a tool that could model and run analysis. Whereas these features received a lot of development iterations, other functions like navigation, security, customizability, and file management might lack features. The provided documentation, however, should be sufficient to further develop the tool.

## 8.3 Future work

From these discussion points and limitations, a number of opportunities and future research topics can be derived.

- **Extensive evaluation**
  One major aspect still to tackle in this research field is: is this approach "useful". A quantitative and qualitative evaluation involving parties such as professional software architects and modeling experts may be conducted in future work. It could find whether the use of the tool in practice is useful and that it works as expected. It could also test whether the theoretical outcomes of the interaction models and their protocols actually uphold in an applied system and whether they can actually cover the right interactions between software components.

- **Visual protocol feedback**
  To deliver the analysis results of the protocol directly and conveniently to the user visual feedback on the protocol modeler could be projected. As discussed in the discussion, the type of feedback we are currently checking for is complex to show on the modeler view so future research would need to look for **what** and **how** to display such feedback.

- **Petri net verification and proofs**
  As stated before, we currently perform powerful but basic checking on the Petri net. Future research could find a combination of Petri net checks that completely guarantees correctness on multiple aspects of the Petri net, and provide formal proofs as to why such a set of checks guarantees correctness. Research would focus much more on the evaluation itself and move its scope to Petri net validation of interactions.

- **Choreography tree and Petri net translation extension**
  The Choreography tree and Petri nets that we can generate from protocols are quite restricted: AND gateway, XOR gateways, References, and Loops are the only structures that a user can work with. Future research might prove which ones are actually useful, and which ones from the BPMN language might potentially be useful to add. It would then also be possible to extend the translations to include these new structures. Furthermore, it should also be possible to modify the transformations for the process tree, as this thesis simplifies them quite heavily.

- **INORA2 extension**
  The tool needs some work to be a fully operational software distribution. Future work could be to extend and stabilize INORA2. Areas on which the tool can be improved are:

  - **Better project and file management** The saving, loading and quick modification of INORA projects is a part that can really be improved, as currently there is just some basic support.
  - **Intermodel-usability** The aim of the tool is to quickly create protocols that are related to an interaction model. Future work could heavily improve the linking of interaction model flows and actual protocols, and make referencing easy with selection and potential mass query execution. This would most likely require a database.
  - **Stability and distribution**
    The INORA2 project is open-source code that can easily be ported to native OS applications with the use of Electron. It, however, also produces a lot of unstable builds, which can be solved in the future by finding all version discrepancies.

- **DAME LoLA integration**
  A more extensive integration of DAME LoLA is an area of improvement. The REST protocol could follow a more generic and easily maintainable structure, with industry-standard security protocols.

## 8.4 Acknowledgements

First and foremost I would like to thank my daily supervisor Dr. ir. J.M.E.M. van der Werf for providing me with research suggestions and overall help in solving the problems related to the topic. The discussions we have had provided me with guidance throughout the thesis. Secondly, I would like to thank my second supervisor Dr. ir. X. Lu for the feedback I have received on my thesis, which helped me in improving the quality.

Also, I would like to thank all people in our shared study space "hok" for providing a productive environment and being able to spar about algorithmic problems.

# Bibliography


[1] P. Clements, D. Garlan, R. Little, R. Nord, and J. Stafford, "Documenting software architectures: views and beyond," in *25th International Conference on Software Engineering, 2003. Proceedings.*, pp. 740–741, IEEE, 2003.

[2] M. Klijs, "A choreography-based design method for component-based software systems," Master's thesis, Utrecht University, 2021.

[3] N. Lohmann, E. Verbeek, and R. Dijkman, "Petri Net Transformations for Business Processes – A Survey," in *Transactions on Petri Nets and Other Models of Concurrency II* (K. Jensen and W. M. P. van der Aalst, eds.), vol. 5460, pp. 46–63, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. Series Title: Lecture Notes in Computer Science.

[4] M. D. McIlroy, J. Buxton, P. Naur, and B. Randell, "Mass-produced software components," in *Proceedings of the 1st international conference on software engineering, Garmisch Pattenkirchen, Germany*, pp. 88–98, 1968.

[5] R. J. Wieringa, *Design science methodology for information systems and software engineering*. Springer, 2014.

[6] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*. Addison-Wesley Professional, 2003.

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

[8] I. A. W. Group *et al.*, "Recommended practice for architectural description," *IEEE P1471 D*, vol. 5, 1999.

[9] J. M. E. van der Werf, "Compositional verification of asynchronously communicating systems," in *International Conference on Formal Aspects of Component Software*, pp. 49–67, Springer, 2014.

[10] T. Bultan, "Analyzing interactions of asynchronously communicating software components," in *Formal Techniques for Distributed Systems*, pp. 1–4, Springer, 2013.

[11] M. K. Aguilera and M. Walfish, "No time for asynchrony.," in *HotOS*, 2009.

[12] N. Viswanadham, Y. Narahari, and T. L. Johnson, "Deadlock prevention and deadlock avoidance in flexible manufacturing systems using petri net models," *IEEE Transactions on Robotics & Automation Magazine*, vol. 6, no. 6, pp. 713–723, 1990.

[13] I. Lotos, "A formal description technique based on the temporal ordering of observational behaviour. international standard 8807," *International Organization for Standardization | Information Processing Systems | Open Systems Interconnection, Gen eve*, 1988.

[14] C. A. R. Hoare *et al.*, *Communicating sequential processes*, vol. 178. Prentice-hall Englewood Cliffs, 1985.

[15] R. Milner, *Communication and concurrency*, vol. 84. Prentice hall Englewood Cliffs, 1989.

[16] T. Murata, "Petri nets: Properties, analysis and applications," *PROCEEDINGS OF THE IEEE*, vol. 77, no. 4, p. 40, 1989.

[17] N. Chomsky and G. A. Miller, "Finite state languages," *Information and control*, vol. 1, no. 2, pp. 91–112, 1958.

[18] R. M. Dijkman, M. Dumas, and C. Ouyang, "Semantics and analysis of business process models in bpmn," *Information and Software technology*, vol. 50, no. 12, pp. 1281–1294, 2008.

[19] W. P. D. Interface, "Workflow management coalition workflow standard workflow process definition interface–xml process definition language," 2002.

[20] U. R. T. Force, "Omg unified modeling language: Superstructure," *Object Management Group (OMG)*, 2010.

[21] W. M. Van der Aalst, "Pi calculus versus petri nets: Let us eat "humble pie" rather than further inflate the "pi hype"," *BPTrends*, vol. 3, no. 5, pp. 1–11, 2005.

[22] J. M. E. van der Werf, A. Rivkin, A. Polyvyanyy, and M. Montali, "Data and process resonance," in *International Conference on Applications and Theory of Petri Nets and Concurrency*, pp. 369–392, Springer, 2022.

[23] J. L. Peterson, "Petri nets," *ACM Computing Surveys (CSUR)*, vol. 9, no. 3, pp. 223–252, 1977.

[24] R. v. Glabbeek and F. Vaandrager, "Petri net models for algebraic theories of concurrency," in *International Conference on Parallel Architectures and Languages Europe*, pp. 224–242, Springer, 1987.

[25] J. M. E. van der Werf, "Compositional design and verification of component-based information systems," 2011.

[26] J. C. Baez and J. Master, "Open petri nets," *Mathematical Structures in Computer Science*, vol. 30, no. 3, pp. 314–341, 2020.

[27] K. Wolf, "Does my service have partners?," *Transactions on Petri Nets and Other Models of Concurrency II: Special Issue on Concurrency in Process-Aware Information Systems*, pp. 152–171, 2009.

[28] J. M. E. van der Werf, "Compositional verification of asynchronously communicating systems," in *Formal Aspects of Component Software: 11th International Symposium, FACS 2014, Bertinoro, Italy, September 10-12, 2014, Revised Selected Papers 11*, pp. 49–67, Springer, 2015.

[29] P. Massuthe, A. Serebrenik, N. Sidorova, and K. Wolf, "Can i find a partner? undecidability of partner existence for open nets," *Information Processing Letters*, vol. 108, no. 6, pp. 374–378, 2008.

[30] X. Ye, J. Zhou, and X. Song, "On reachability graphs of petri nets," *Computers & Electrical Engineering*, vol. 29, no. 2, pp. 263–272, 2003.

[31] V. M. Savi and X. Xie, "Liveness and boundedness analysis for petri nets with event graph modules," in *International Conference on Application and Theory of Petri Nets*, pp. 328–347, Springer, 1992.

[32] K. Barkaoui, J.-M. Couvreur, and K. Klai, "On the equivalence between liveness and deadlock-freeness in petri nets," in *International Conference on Application and Theory of Petri Nets*, pp. 90–107, Springer, 2005.

[33] G. Liu and K. Barkaoui, "A survey of siphons in petri nets," *Information Sciences*, vol. 363, pp. 198–220, 2016.

[34] K. Barkaoui and J.-F. Pradat-Peyre, "On liveness and controlled siphons in petri nets," in *International Conference on Application and Theory of Petri Nets*, pp. 57–72, Springer, 1996.

[35] A. Valmari, "Stubborn sets for reduced state space generation," in *International Conference on Application and Theory of Petri Nets*, pp. 491–515, Springer, 1989.

[36] D. Wu and W. Zheng, "Formal model-based quantitative safety analysis using timed coloured petri nets," *Reliability Engineering & System Safety*, vol. 176, pp. 62–79, 2018.

[37] A. Cheng, S. Christensen, and K. H. Mortensen, "Model checking coloured petri nets-exploiting strongly connected components," *DAIMI report series*, no. 519, 1997.

[38] I. Raedts, M. Petkovic, Y. S. Usenko, J. M. E. van der Werf, J. F. Groote, and L. J. Somers, "Transformation of bpmn models for behaviour analysis.," *MSVVEIS*, vol. 2007, pp. 126–137, 2007.

[39] W. M. Van Der Aalst, K. M. Van Hee, A. H. Ter Hofstede, N. Sidorova, H. Verbeek, M. Voorhoeve, and M. T. Wynn, "Soundness of workflow nets: classification, decidability, and analysis," *Formal aspects of computing*, vol. 23, no. 3, pp. 333–363, 2011.

[40] K. M. van Hee, N. Sidorova, and J. M. van der Werf, "Refinement of synchronizable places with multi-workflow nets: weak termination preserved!," in *Applications and Theory of Petri Nets: 32nd International Conference, PETRI NETS 2011, Newcastle, UK, June 20-24, 2011. Proceedings 32*, pp. 149–168, Springer, 2011.

[41] A. Valmari, "The state explosion problem," in *Advanced Course on Petri Nets*, pp. 429–528, Springer, 1996.

[42] T. Allweyer, *BPMN 2.0: introduction to the standard for business process modeling*. BoD–Books on Demand, 2016.

[43] C. Natschläger, "Towards a bpmn 2.0 ontology," in *International Workshop on Business Process Modeling Notation*, pp. 1–15, Springer, 2011.

[44] M. Cortes-Cornax, S. Dupuy-Chessa, D. Rieu, and M. Dumas, "Evaluating choreographies in bpmn 2.0 using an extended quality framework," in *International Workshop on Business Process Modeling Notation*, pp. 103–117, Springer, 2011.

[45] F. Corradini, A. Morichetta, A. Polini, B. Re, and F. Tiezzi, "Collaboration vs. choreography conformance in bpmn 2.0: from theory to practice," in *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*, pp. 95–104, IEEE, 2018.

[46] M. Chinosi and A. Trombetta, "Bpmn: An introduction to the standard," *Computer Standards & Interfaces*, vol. 34, no. 1, pp. 124–134, 2012.

[47] M. Ramadan, H. G. Elmongui, and R. Hassan, "Bpmn formalisation using coloured petri nets," in *Proceedings of the 2nd GSTF annual international conference on software engineering & applications (SEA 2011)*, pp. 83–90, 2011.

[48] H. Scheuerlein, F. Rauchfuss, Y. Dittmar, R. Molle, T. Lehmann, N. Pienkos, and U. Settmacher, "New methods for clinical pathways—business process modeling notation (bpmn) and tangible business process modeling (t. bpm)," *Langenbeck's archives of surgery*, vol. 397, no. 5, pp. 755–761, 2012.

[49] R. M. Dijkman, M. Dumas, and C. Ouyang, "Formal semantics and automated analysis of bpmn process models," *preprint*, vol. 7115, 2007.

[50] M. Nouioua, A. Adel, and B. Zouari, *A Novel Petri Nets-based Modeling for the correctness of interactive Business Process Models*. Dec. 2018.

[51] L. Li and F. Dai, "Transformation and Visualization of BPMN Models to Petri Nets," *IOP Conference Series: Earth and Environmental Science*, vol. 186, p. 012047, Oct. 2018.

[52] H. M. Verbeek, T. Basten, and W. M. van der Aalst, "Diagnosing workflow processes using woflan," *The computer journal*, vol. 44, no. 4, pp. 246–279, 2001.

[53] P. H. Starke and S. Roch, "Ina: integrated net analyzer," *Reference Manual*, 1992.

[54] K. Schmidt, "Lola a low level analyser," in *International Conference on Application and Theory of Petri Nets*, pp. 465–474, Springer, 2000.

[55] K. Wolf, "Petri net model checking with lola 2," in *International Conference on Applications and Theory of Petri Nets and Concurrency*, pp. 351–362, Springer, 2018.

[56] K. Wolf, "Running lola 2.0 in a model checking competition," in *Transactions on Petri nets and other models of concurrency XI*, pp. 274–285, Springer, 2016.

[57] W. J. Thong and M. Ameedeen, "A survey of petri net tools," *Advanced computer and communication engineering technology*, pp. 537–551, 2015.

[58] A. Holfter, S. Haarmann, L. Pufahl, and M. Weske, "Checking compliance in data-driven case management," in *International Conference on Business Process Management*, pp. 400–411, Springer, 2019.

[59] J. van Benthem, "Temporal logic," 1991.

[60] S. Hinz, K. Schmidt, and C. Stahl, "Transforming bpel to petri nets," in *International conference on business process management*, pp. 220–235, Springer, 2005.

[61] A. Niewiadomski and K. Wolf, "Lola as abstract planning engine of planics.," in *PNSE@ Petri Nets*, pp. 349–350, 2014.

[62] L. Cabac, M. Haustermann, and D. Mosteller, "Renew 2.5–towards a comprehensive integrated development environment for petri net-based applications," in *International conference on applications and theory of Petri nets and concurrency*, pp. 101–112, Springer, 2016.

[63] F. Zalila, X. Crégut, and M. Pantel, "A transformation-driven approach to automate feedback verification results," in *International Conference on Model and Data Engineering*, pp. 266–277, Springer, 2013.

[64] D. S. Kolovos, R. F. Paige, and F. A. Polack, "Merging models with the epsilon merging language (eml)," in *International Conference on Model Driven Engineering Languages and Systems*, pp. 215–229, Springer, 2006.

[65] F. Jouault, "Loosely coupled traceability for atl," in *Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Nuremberg, Germany*, vol. 91, p. 2, Citeseer, 2005.

[66] V. Stein Dani, C. M. D. S. Freitas, and L. H. Thom, "Recommendations for visual feedback about problems within BPMN process models," *Software and Systems Modeling*, Jan. 2022.

[67] J. Grossman, "Color conventions and application standards," in *Color in electronic displays*, pp. 209–218, Springer, 1992.

[68] N. D. Jones, L. H. Landweber, and Y. E. Lien, "Complexity of some problems in petri nets," *Theoretical Computer Science*, vol. 4, no. 3, pp. 277–299, 1977.

[69] K. Barkaoui and M. Minoux, "A polynomial-time graph algorithm to decide liveness of some basic classes of bounded petri nets," in *Application and Theory of Petri Nets 1992: 13th International Conference Sheffield, UK, June 22–26, 1992 Proceedings 13*, pp. 62–75, Springer, 1992.

[70] A. Valmari, "The state explosion problem," *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets*, pp. 429–528, 2005.

[71] W. v. d. Aalst, J. Buijs, and B. v. Dongen, "Towards improving the representational bias of process mining," in *International Symposium on Data-Driven Process Discovery and Analysis*, pp. 39–54, Springer, 2011.

[72] M. A. B. Ahmadon and S. Yamaguchi, "Convertibility and conversion algorithm of well-structured workflow net to process tree," in *2013 First International Symposium on Computing and Networking*, pp. 122–127, IEEE, 2013.

[73] J. Mendling, H. A. Reijers, and W. M. van der Aalst, "Seven process modeling guidelines (7pmg)," *Information and software technology*, vol. 52, no. 2, pp. 127–136, 2010.

[74] J. Ladleif, A. von Weltzien, and M. Weske, "chor-js: A modeling framework for bpmn 2.0 choreography diagrams.," in *ER Forum/Posters/Demos*, pp. 113–117, 2019.

[75] A. C. Patthak, I. Bhattacharya, A. Dasgupta, P. Dasgupta, and P. Chakrabarti, "Quantified computation tree logic," *Information processing letters*, vol. 82, no. 3, pp. 123–129, 2002.

[76] M. Sharir, "A strong-connectivity algorithm and its applications in data flow analysis," *Computers & Mathematics with Applications*, vol. 7, no. 1, pp. 67–72, 1981.

[77] W. Zhou, L. Li, M. Luo, and W. Chou, "Rest api design patterns for sdn northbound api," in *2014 28th international conference on advanced information networking and applications workshops*, pp. 358–365, IEEE, 2014.

[78] L. Richardson, M. Amundsen, M. Amundsen, and S. Ruby, *RESTful Web APIs: Services for a Changing World*. " O'Reilly Media, Inc.", 2013.

[79] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[80] J. M. Boyer, C. F. Wiecha, and R. P. Akolkar, "A rest protocol and composite format for interactive web documents," in *Proceedings of the 9th ACM symposium on Document engineering*, pp. 139–148, 2009.

[81] G. Barbaglia, S. Murzilli, and S. Cudini, "Definition of rest web services with json schema," *Software: Practice and Experience*, vol. 47, no. 6, pp. 907–920, 2017.

[82] N. Nurseitov, M. Paulson, R. Reynolds, and C. Izurieta, "Comparison of json and xml data interchange formats: a case study.," *Caine*, vol. 9, pp. 157–162, 2009.

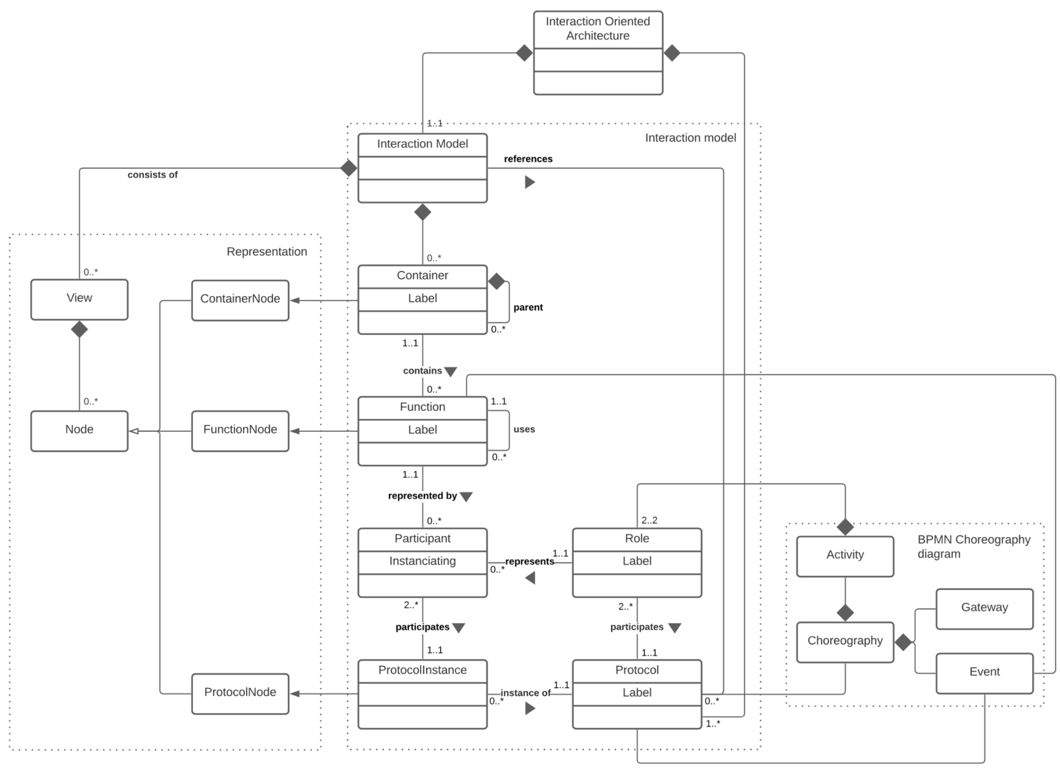
[83] F. Haupt, F. Leymann, A. Scherer, and K. Vukojevic-Haupt, "A framework for the structural analysis of rest apis," in *2017 IEEE International Conference on Software Architecture (ICSA)*, pp. 55–58, IEEE, 2017.

[84] B. B. Rad, H. J. Bhatti, and M. Ahmadi, "An introduction to docker and analysis of its performance," *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 17, no. 3, p. 228, 2017.

[85] T. Combe, A. Martin, and R. Di Pietro, "To docker or not to docker: A security perspective," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 54–62, 2016.

# Appendix A

# INORA meta model



# Appendix B

# Algorithm for generating Choreography Tree

```
// Preprocess Moddle elements
i = preprocess(m)

// Find loop entries and exits
entries, exits = detectLoops(i)

if(isBlockStructured(i)) {
  // Create a choreography tree node (output) that has and id, type and children (nodes)
  root_sequence = new Node(id, type, [children])
  // Create a stack with an array
  // Give a start node, and the parents
  stack = [[start, parents]]
  // Depth First Traversal
  while(stack is not empty) {
    // Pop the last node on the stack, and their parent array.
    node, parents = stack.pop();
    // If node is seen, do nothing
    if(seen) continue;

    // If Node is the start of a loop, create a loop structure in the tree and make it the parent
    if(node in entries) {
      self = new Node(loop + sequence);
      self.append_to(parent);
      parents.push(self);
    }

    // If Node is the end of a loop, find the back-activity, add it to the loop structure and
    go up in the parent stack.
    if(node in exits) {
      self = new Node(activity);
      self.append_to(loop);
      parents.pop();
    }

    // If Node is a join node, go up in the parent stack (this block is closed)
    if(type of node is 'join') {
      parents.pop();
    }

    // If node is fork node, create fork structure in the tree and make it the parent.
    if(type of node is 'fork') {
      self = new Node(fork + sequence);
```

```
      self.append_to(parent);
      parents.push(self);
    }

    // If node is a none structure node (e.g. simple activity, reference), add it to the current parent.
    else {
      self = new Node(activity or reference);
      self.append_to(parent);
    }

    // All children of this node, add to DFS stack
    for nodes in outgoing_arcs:
      stack.push([node, parents])
  }
}
```

**Code snippet B.1:** Pseudo code for generating a choreography tree

# Appendix C

# Algorithm for generating Petri net

```
// Filter on the start node of Moddle elements
start = getStartNode(elements)
// Create a stack to depth-first traverse over all the nodes
// In edge stores the ID of the element before current one
stack = [[startNode, inEdge]]
// Store all elements for the petri net
net = []
// Track all unique participants in this protocol
participants = []
// Track all open i/o places for each element
io = {[id]: { in: [], out:[] } }

while(stack is not empty):
    // Pop from the stack
    node, in_edge = stack.pop()

    if(seen):
        // Even if we have seen this node, might still need to connect.
        // A place can have multiple incoming arcs
        if(in_edge):
            edge = connect(in_edge, this)
            net.push(edge)
        else: continue
    seen.push(node)

    // If unseen participant, add to participant tracker
    if(node.participant not in participants) push(participant)

    // Find the corresponding translation segment from a dictionary
    segment = findTranslationSegment(node)
    // Map generic segment to context
    map(segment)

    // for each Petri net element in context
    for item in mapped_segment {
      // Create id and add to net
      id = generate_id()
      element = item.toPetri(id)
      net.push(element)

      // Mark possible input and output places for this node
      if(outplace) io.node.out.push(this)
      if(inplace) io.node.in.push(this)
    }
```

```
    // Connect incoming edge for this iteration to next
    // available input place of this node
    find_and_connect(this, in_edge)

    // populate DFS stack
    for nodes in outgoing_arcs {
      stack.push([node, out_edge])
    }
```

**Code snippet C.1:** Pseudo code for generating a Petri net

# Appendix D

# Petri net translation segments

```
/*
How do we translate a BPMN to a petri net? Define a certain "component" e.g. start event.
Then define the places used: place = "<modifier><id>", input place one = i1,
intermediate place 2 = p2, output place one = o1 etc...
Then define the transitions: transition = "t<id>"
Then define the edges: "{p/t}>{p/t}" e.g. "i1>t1"

You can use number (e.g. 1,2,3) as <id> for fixed places/transitions,
or you can use a generic "n" which maps a n = 1...n.

Example: a Parallel gateway fork will need as many out places as arcs coming out in the protocol.
Therefore we define n transitions and n "o" places.

You can also use a "merged node" ("m<modifier>p", short for Merged <modifier> Place),
which simply inserts a single place that functions as
("in" AND "o1") OR ("i1" AND "on") depending on mip/mop. It cannot have transitions or edges.
*/

export const translationModule = [
    {
        translationSegment: 'ChoreographyTask_sequence',
        content: {
            places: [
                'i1',
                'o1',
            ],
            transitions: [
                't1'
            ],
            edges: [
                'i1>t1',
                't1>o1',
            ]
        }
    },
    {
        translationSegment: 'IntermediateThrowEvent_sequence',
        content: {
            places: [
                'i1',
                'o1',
            ],
            transitions: [
                't1'
            ],
```

```
            edges: [
                'i1>t1',
                't1>o1',
            ]
        }
    },
    {
        translationSegment: 'event_start',
        content: {
            places: [
                'i1',
                'o1',
            ],
            transitions: [
                't1'
            ],
            edges: [
                'i1>t1',
                't1>o1',
            ]
        }
    },
    {
        translationSegment: 'event_end',
        content: {
            places: [
                'i1',
                'o1',
            ],
            transitions: [
                't1'
            ],
            edges: [
                'i1>t1',
                't1>o1',
            ]
        }
    },
    {
        translationSegment: 'ParallelGateway_fork',
        content: {
            places: [
                'i1',
                'on'
            ],
            transitions: [
                't1'
            ],
            edges: [
                'i1>t1',
                't1>on'
            ]
        }
    },
    {
        translationSegment: 'ParallelGateway_join',
        content: {
            places: [
                'in',
                'o1',
            ],
            transitions: [
                't1'
```

```
            ],
            edges: [
                'in>t1',
                't1>o1',
            ]
        }
    },
    {
        translationSegment: 'ExclusiveGateway_fork',
        content: {
            places: [
                'mop'
            ],
            transitions: [],
            edges: []
        }
    },
    {
        translationSegment: 'ExclusiveGateway_join',
        content: {
            places: [
                'mip'
            ],
            transitions: [],
            edges: []
        }
    }
]
```

**Code snippet D.1:** Translation segment used for Petri net translation in INORA2.

# Appendix E

# DAME LoLA

An important side product of this research is DAME: a containerized version of LoLA 2.0 with a simple connection interface. This appendix explains the project in detail.

## DAME LoLA: Petri net validation tool

As stated in section 3.3 LoLA2 is one of the most powerful tools to validate Petri nets and use them for model checking. A logical consequence is to therefore use it in INORA2. This “model checking” part of the INORA2 system is built as a standalone tool, with all interactions being executed via a generally accepted protocol. As the tool can be used to check any Petri net on properties, it has a broader purpose than solely supporting INORA2. This tool combines the power of LoLA2, an API (REST), and Docker. The working title for this combination is “DAME LolA” (**D**ocker **A**nalysis **M**icroServic**e** for LoLA).

## LoLA2

LoLA2 is a tool written by Karsten Wolf and Niels Lohmann. It can be downloaded via their website[1]. It is distributed with a GNU AFFERO GENERAL PUBLIC LICENSE which allows altering and redistributing software under the same license[2]. The license is specifically designed to ensure cooperation with the community in the case of network server software, as which LoLA2 can be categorized.

LoLA2 is written in C and distributed as source code. To do a local installation the code first needs to be compiled using the `automake` procedures. The newest versions of the C compiler, however, cannot deal with a deprecated declaration and therefore the source code of LoLA first needs to be patched. This fix is suggested by Bentley James Oakes[3]. After patching the source code, LoLA2 can simply be installed by running the following command from within the LoLA source directory [55]:

```
> ./configure
> make install
```

**Code snippet E.1:** Installing LoLA2 via command line.

After installation the tool can be used as a command-line executable. It can be used with the following syntax:

[1] https://theo.informatik.uni-rostock.de/theo-forschung/tools/lola/
[2] https://www.gnu.org/licenses/agpl-3.0.nl.html
[3] http://msdl.cs.mcgill.ca/people/hv/teaching/MSBDesign/assignments/Assign5-AToMPM_CodeGen.pdf

```
> lola [FILE] [--formula=FORMULA] [OPTIONS]...
```

**Code snippet E.2:** Running LoLA2 command line executable.

A few key arguments are

- `[FILE]` This argument supplies a file path on the system to LoLA. This file is formatted as a LOLA file (.lola) and contains the Petri net that needs to be verified.
- `[--formula=FORMULA]` This argument sets the property on which the net will be checked. This is read from a `.formula` file, due to the character limit on commands. All possible formulas and the way they are supposed to be used can be found in a PDF file which is part of the distribution.
- `OPTION[--json=output.json]` There are many different options for the lola command. These are all listed in the PDF as stated above. An important option for DAME LoLA is to write the JSON output to a file so that is can be used later.
- `OPTION[--path=output.path]` This option returns a counter-path to a query. For example, if checking for deadlocks, this option produces a firing sequence (path) which makes the net end up in a deadlock *iff* there exists one.

**REST API**

Once LoLA2 is installed on the system it can be used from within the terminal. However, it is not yet callable from external sources. Note that LoLA2 includes software to use LoLA in a network setup. This uses the UDP protocol and requires all callers to be on the same network. To make LoLA2 callable from sources that are not on the same network, another protocol needs to be used. Technically it would be possible to use the UDP protocol for this, but this limits the customizability of requests and verification.

A very commonly used method to supply applications [77] with information from external tools is Representational State Transfer (REST), which is a set of design constraints [78] that uses the HTTP protocol [79] and allows users to define any type of pattern for reading, writing and other interactions with a service via an URL [80]. In the scope of DAME LoLA, it specifically implements an HTTP POST call which can be implemented to supply the installed LoLA instance with instructions and a Petri net. This is done by calling an URL endpoint which can then parse the POST call with a given body. This body is formatted as JSON (**J**ava**S**cript **O**bject **N**otation), which is an inherent consequence of using REST [81] and outperforms XML in terms of performance [82].

As REST is not a protocol itself it can be implemented with many different programming languages and frameworks. In DAME LoLA this is done by writing an API with a Python framework called FastAPI[4]. FastAPI is a fast and low code method of exposing Python functions via REST over an URL endpoint. The main endpoint from DAME LoLA is given as an example function as can be seen in snippet E.4. Once FastAPI is imported, it can be used as a function decorator[5] for any function. In this case, a POST request can be submitted to `/analyse` and expects a LolaRequest typed variable and returns a Python dictionary. Both the class `LolaRequest` and the returned dictionary have a direct mapping to a JSON-like object, which is used as request and response bodies in the REST framework. Note that Pythons typing package is used in this code to make it clear and maintainable.

```python
1  from fastapi import FastAPI
2  app = FastAPI()
3
4  @app.post("/analyse")
5  def handle_request(request: LolaRequest) -> Dict:
6      ...
7      return Dict
```

[4]`https://fastapi.tiangolo.com/`
[5]`https://wiki.python.org/moin/PythonDecorators`

**Code snippet E.3:** Python definition exposed as REST POST endpoint using FastAPI.

### Python logic

As stated in the previous section, the Python script provides a REST API. The two main callable functions are `/status` and `/analyse`. In snippet E.4 and E.5 pseudo code of these implementations can be found.

```
# Used to check the status of the service via GET.
@app.get("/status")
async def status() -> Dict:
    ...
    return Dict

# Output (Dict/JSON)
{
  "status": True | False
}
```

**Code snippet E.4:** Input and output of /status.

```
# Used to submit LoLA files and run verification
# on Petri nets via POST
@app.post("/analyse")
def handle_request(request: LolaRequest) -> Dict:
    ...
    return Dict

# Input (REST/JSON)
{
    "formulas": ["<name>", ...],
    "lolafile": "..."
}

# Output (Dict/JSON)
{
  "analysis": [
    {
      "formula": "<name>",
      "result": {...}
    },
    ...
  ]
}
```

**Code snippet E.5:** Input and output of /analyse.

Once a LoLA call on `/analyse` has been made, the API fires a sequence of steps. During any time an error may occur. If this happens the process is interrupted, the error is logged to a `.log` file on the server and the caller is notified via an HTTP error message.

1. The call is verified to check whether it complies with the LolaRequest class.

2. Logging is now enabled. All actions as a consequence of this call are logged on the system.

3. A unique process ID is generated at random to identify all temporary files belonging to this call.

4. The request variable LolaRequest.lolacode is extracted and checked for validity. If is a valid LoLA file, the file is written to a temporary file on the system /<home>/tmp_files_in/<uid>.lola.

5. Next, all formulas are extracted from the request variable LolaRequest.formulas. This is a list of all checks that the caller wants to run on the Petri net. It may contain an arbitrary number of LoLA2 supported formulas, such as deadlock and general modelchecking. The user-friendly names are then mapped to the correct LOLA2 formula format, and written to a temporary input file. This step also ensures that there is never any input from the caller directly being run on the terminal, because if the mapping fails nothing will be run at all.

6. After all formulas are extracted, and the LoLA file written to the system, the tool will be run. This is done serially and separately for each formula, as LoLA2 does not support multiple formulas at one time. The result is written to /<home>/tmp_files_out/<uid>/<formula_name>.json. The directory /<home>/tmp_files_out/<uid>/ thus includes a separate JSON file for each LoLA formula run.

7. If the net does not comply with a formula and LoLA is able to find a path that demonstrates the non-compliance, it writes a possible firing sequence (one of many) in the /<home>/tmp_files_out/<formula_name>.path file.

8. All output files are collected from the directory and are combined into a Python dictionary. The in and out directories destroyed.

9. The combined result is returned to the caller.

## Combining LoLA and REST: Docker

Now that LoLA2 is compiled and Python handles the requests it is very useful to make this a single "package" so that is can be run from anywhere as a microservice [83]: in the cloud, on a server or on a local PC. A powerful method that supports this is containerization, and one of the most commonly used open-source tools is Docker [84] [85]. Such a container can be packaged into a Docker image: a predefined package that any user can run that includes all the code and environment settings and can be modified to the liking of the user. It ensures that the application responds the same on any system running Docker.
The DAME LoLA docker image is built around another image and ensures system security by incorporating a couple of techniques.

1. It uses the base image of gcc[6], which is a set of tools to compile C-code installed on a very lightweight Linux distribution called Alpine.

2. It copies the LoLA2 source code to the container.

3. It runs some basic configuration (setting working directory, installing packages) and runs the LoLA2 installer.

4. It installs Python3, which is necessary for the REST API.

5. It creates a limited access user called pyapi-runner that can only run Python and LoLA and access its home directory for security reasons.

6. All Python source code is copied to the container, and all directories that the API will use are created with the right access levels.

7. The tool uvicorn is used to run a web server with FastAPI. This web server is assigned to IP address 0.0.0.0 and port 8000.

[6] https://hub.docker.com/_/gcc

With the use of a docker-compose.yml runtime variables can be set, and the image can be run. Some important variables that should be set are the port and volume mapping. The ports flag should map the 8000 port from the container to a free port on the host OS, and the flag volumes is mapped to a location on the system to checkout LoLA output files on the host OS. An example of a docker-compose file running DAME LoLA can be found in snippet E.6. To run the application, make sure Docker is running on the host OS and run docker-compose up in the terminal of the docker-file directory. After is it running the endpoint can be found under the host OS IP or http://localhost:<host-port>/<endpoint>.

```
version: "3"
services:
    # Define the container to run LORD
    lola-runner:
        # Build from the dockerfile if
        # you run the source code from Git
        build: ./build
        # Or use image flag if you want to
        # pull the image from Docker Hub
        - image: lola-docker
        # Map docker ports to local ports (host:container)
        ports:
        - 8080:8000
        # Create a volume which is stored
        volumes:
        - /some/local/hostOS/path:/home/pyapi-runner/
```

**Code snippet E.6:** Example docker-compose file contents.

## Summary

In this Docker is used to install both LoLA2 and Python to a Docker image. A FastAPI tool is supplied with the image to expose the LoLA command line executable via a REST API. This package is created with security and flexibility in mind. The image will be published to Docker Hub[7] once finished under an open-source license. The source code can be found on Gitlab: https://git.science.uu.nl/interaction-oriented-architecture/lola-docker.

[7] https://hub.docker.com/