THESIS / THÈSE

MASTER IN COMPUTER SCIENCE PROFESSIONAL FOCUS IN SOFTWARE ENGINEERING

Modelling microservice-based applications

ROMAIN, François

Award date:
2022

Awarding institution:
University of Namur

University of Namur
Faculty of Computer Science
Academic year 2021–2022

Modelling microservice-based applications

François Romain

Supervisor : Antonio Brogi, Jacopo Soldani

Promoter : ________________________ (Signature for approval of the deposit - EAR art. 40)
Jean-Marie Jacquet

Thesis submitted in partial fulfilment of the requirements for the degree of
Master in Computer Science.

Acknowledgment

I would like to thank, first of all, Jacopo Soldani who was able to follow me on a daily basis and advise me to help me in the realisation of my thesis. I do not forget either the feedback from Antonio Brogi who was always very constructive.

Thanks also to my promoter Jean-Marie Jacquet for his proofreading and advice.

Abstract

More and more, software architectures are evolving. Microservices appeared in the last ten years or so. They are gaining more and more ground and their popularity is ever increasing. They have their own advantages and disadvantages and, like any software, they need to be tested. Analysis tools are already present in the literature, and modelling languages exist for certain aspects. Here, we propose a solution based on Petri nets and a algebra process to analyse the communications between the different services and verify whether or not they can interact correctly together. More generally, we will try to answer to the following questions : what will the problems be when connecting them together, where will the error occur, will it be propagated, etc.? These are all questions that we will try to answer by proposing this new analysis and modelling method.

Keywords : microservices, process algebra, Petri nets, modelling, analysis

Résumé

De plus en plus, les architectures de logiciels évoluent. Depuis une dizaine d'années, les microservices sont apparus. Ils gagnent de plus en plus de terrain et leur popularité ne fait qu'augmenter. Ils ont leurs propres avantages et inconvénients et comme tout logiciel, il est nécessaire de les tester. Des outils d'analyse sont déjà présents dans la littérature et des langages de modélisation existent pour certains aspects. Ici, nous proposons une solution basée sur les réseaux de Petri et une algèbre de processus pour analyser les communications entre les différents services et voir si oui ou non, ils peuvent interagir correctement ensemble. Quels seront les problèmes lorsqu'on les connecte ensemble, où l'erreur se produira-t-elle, sera-t-elle propagée, etc. Ce sont autant de questions auxquelles nous allons essayer de répondre en proposant ce nouveau modèle d'analyse et de modélisation.

Mots-clés : microservices, process algebra, réseaux de Petri, modélisation, analyse

Contents

1 Introduction **1**
1.1 Overall objectives of the thesis . . . 1
1.2 Context . . . 1
1.3 Working method . . . 2
1.4 Work process . . . 2

2 An introduction to microservices **3**
2.1 A brief history of microservices . . . 3
2.2 The monolithic style . . . 3
2.3 What is microservice ? . . . 4
2.3.1 Componentization via Services . . . 4
2.3.2 Organised around Business Capabilities . . . 5
2.3.3 Products not Projects . . . 5
2.3.4 Smart endpoints and dumb pipes . . . 5
2.3.5 Decentralised Governance . . . 5
2.3.6 Decentralised Data Management . . . 5
2.3.7 Infrastructure Automation . . . 6
2.3.8 Design for failure . . . 6
2.3.9 Evolutionary Design . . . 6
2.4 Pros and cons . . . 6
2.4.1 Pros . . . 6
2.4.2 Cons . . . 7
2.5 Examples of applications using microservices . . . 7
2.6 Conclusion . . . 7

3 Modelling and verification tools for microservices **9**
3.1 Tools . . . 9
3.1.1 µTOSCA . . . 9
3.1.2 GSMART . . . 10
3.1.3 ucheck . . . 11
3.1.4 MICO . . . 12
3.1.5 LEMMA . . . 12
3.1.6 Jolie . . . 13
3.1.7 Conclusion . . . 14
3.2 Modelling languages . . . 14
3.2.1 Event-B . . . 14
3.2.2 Process algebra . . . 17
3.2.3 Rules for the transition system . . . 19
3.2.4 Petri net . . . 20
3.2.5 Petri Net Markup Language . . . 23
3.3 Conclusion . . . 25

4 A process algebra based approach to model microservices **27**
4.1 The sock shop model . . . 27
4.1.1 The queue . . . 27
4.1.2 The shipping . . . 28
4.2 Different scenarios . . . 28

4.2.1 Failure-free scenario 28
4.2.2 Failure-Aware scenario 28
4.3 π-calculus and CCS based language 29
4.3.1 Definition of the language 29
4.3.2 Rules for the transition system 30
4.3.3 How to (compositionally) specify microservices in this model 30
4.3.4 Failure-free scenario 32
4.3.5 Failure-aware scenario 37
4.3.6 Replicas 39
4.4 Conclusion 40

5 Reasoning on microservices with Petri Nets 41
5.1 Translate process calculus to PNML 41
5.1.1 An algorithm to generate Petri nets from π-calculus based language . . . 41
5.1.2 Implementation of this process 42
5.1.3 How can the behaviour of the model be preserved ? 43
5.1.4 Example 43
5.2 Petri nets based model 44
5.2.1 Failure-free scenario 44
5.2.2 Failure-aware scenario 48
5.3 Analysis of Petri nets 52
5.3.1 Semantical analysis 52
5.3.2 Coverability graph 53
5.3.3 Token game 53
5.3.4 Other properties 54
5.4 Conclusion 55

6 Conclusion 57

7 Appendices 63
7.1 PNML example 63
7.2 PNML of the queue component 65
7.3 PNML of the shipping component 67

Chapter 1

Introduction

This document starts with a small introduction to the subject by explaining the problem and in particular where it comes from.

To do this, the problem and the purpose of the work will be presented and then, the context of the work will be taken into account. After that, I will explain how the work was constructed and how I was able to reason to produce this result

1.1 Overall objectives of the thesis

The main objective of this document is to develop a **modelling** and **analysing** technique to determine whether the **microservices** forming and application successfully **integrate**.

This new **model** should enable specifying several things :

- The **microservice** forming an application by defining some of its properties :
  - **Input** / **Output** behaviour : It is necessary to define how microservices can communicate to model them correctly.
  - **Failure** behaviour : This helps to understand where services can be problematic and how they react to the errors of other services.
- The **interactions** occurring among such microservices
  - **Static** / **dynamic** microservices' connections (**output-to-input**) : How two services can communicates together ? Which one communicate with the other one ?
  - Interactions among microservices' **replicas**

With all these descriptions of the different services, the second step is to perform some analysis. The analysis should enable checking :

- A message sent by a service **can be handled** by the recipient service (request, reply and "error reply").
- Microservices can tolerate the **unexpected failure** of the microservices they interact with (No reply).
- Each microservice's **replica is eventually "invoked"**.

The analysis is mainly based on the interaction between the microservices to see if they can work together. The objective is to see how the services react to each other's behaviour and what happens if there is a failure somewhere. How is error propagation handled ?

1.2 Context

There are already modelling and analysis tools for applications, and more specifically for microservices. Unfortunately, they don't offer all the modelling and analysis possibilities and freedoms that I wanted to meet with these different objectives. These tools obviously have their advantages and disadvantages which will be reviewed later in this document.

1.3 Working method

The analysis is essentially based on the interactions between microservices and the integration between them in order to meet the various objectives stated above.

For that, I have browsed the literature to know what has already been done for microservices (in particular because it is a different architecture from the more "classical" applications, said in monolithic style). I was able to discover several tools and models. In the end, the proposed solution is based on a process algebra and, more specifically, a subset of the π-calculus.

1.4 Work process

Throughout the work, I worked through iterations. After going through the state of the art, I regularly came back with a model to analyse the pros and cons. Once this was analysed, corrections were made to the model to get a better model. Once this model was obtained, I was able to move on to the different analyses. First of all with π-calculus and then to think about Petri nets to see what they could bring more.

The rest of the document is structured as follows : first, an introduction to microservices to understand what they are and how they differ from what we are used to seeing. Then, to finish the state of the art, a review of the different tools and modelling languages will be done to see what exists and understand why it is necessary to develop a new tool. After that, the modelling will come to be able to analyse the microservices using a process algebra approach. To continue the analysis and go even further, a solution will be proposed to adapt this model to Petri nets to have even more possibilities of analysis. Finally, a last chapter will close the whole and will propose reflexions for possible future work.

Chapter 2

An introduction to microservices

This part is important because it tells us what microservices are, their history and what has already been done in literature. This allows us to have a better understanding of what microservices are and how to model them.

First of all, the most important is to understand the background and what microservice means. To that end, it is useful to compare it to the monolithic style. Once this is done, one might ask why some people want to move away from monolithic structures. Obviously, microservices have some gains and some pains, but what are they ? Knowing this, can we find real use cases, or is this still a theoretical model ? This whole first chapter aims to answer these questions.

2.1 A brief history of microservices

The term "micro web services" appeared for the first time during a conference on cloud computing in 2005 and the term "microservices" at an event for software architects in 2011 [1]. It is therefore something relatively new, although already known. But it is only recently that the popularity of these has begun to grow, particularly because of the various advantages they offer.

The desire to create microservices is first of all to get away from the "more classic" monolithic style (this style is described just below). This new paradigm for programming by the composition of small services has been built on the concepts of SOA [2]. The different services must be able to communicate using lightweight protocols and offer more freedom in development. It is easier to divide a big problem into several sub-problems. Microservices break down complex tasks into smaller processes.

It is possible to classify services into several categories [1] :

- **Central services** : Handle business data persistence and apply business rules and other logic.
- **Composite services** : Organize either a number of central services in order to fulfil a common task, or aggregate information from several central services.
- **API services** : Open up a functionality externally to e.g. allow third parties to develop creative applications that utilize the underlying functionality in the system landscape.

Of course, these services are independent but will have to work together to form an application.

2.2 The monolithic style

One of the best known software architectures is the following : a client-side user interface, a database and a server-side application. The server handles the request, processes the logic, interacts with the database if it's needed and finally, responds to a message that the client can display to the user. It's a well-known architecture, but once you make a change, you must redeploy all your application (where there have been changes). Also, it is sometimes difficult to

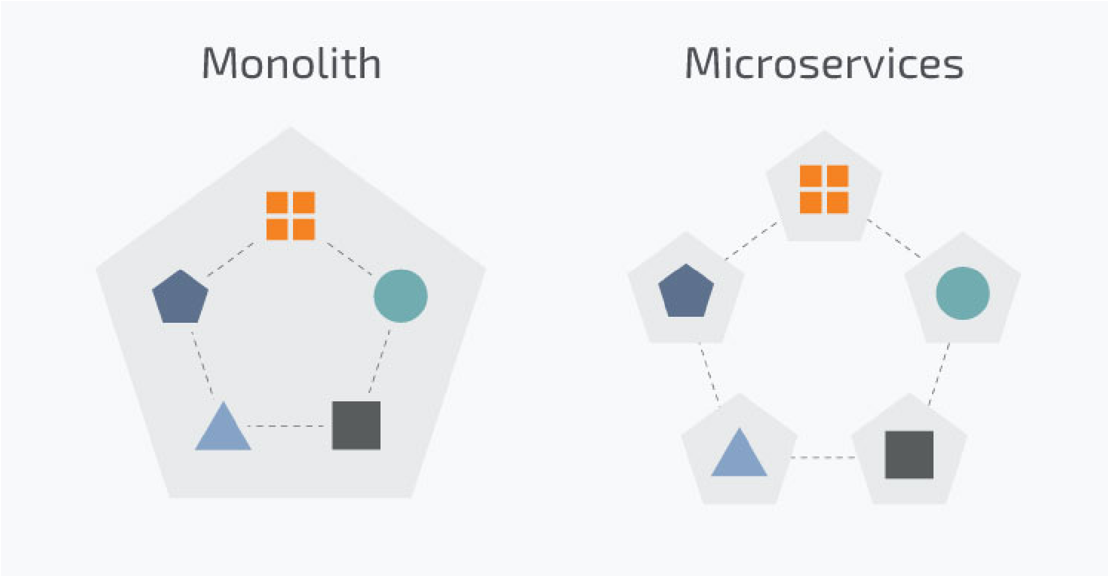


Figure 2.1: Differences between monolithic architecture and MSA [4]

make changes because they can have side effects on other parts of the application. It is therefore necessary to understand the architecture of the whole application. These frustrations have led to the emergence of the microservice architectural style : building applications as suites of services.

2.3 What is microservice ?

M. Fowler and J. Lewis define the microservice architectural style as "*an approach for developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API*" [3]. For N. Dragoni et al. [2], they have another definition of a microservice. "*A microservice is a cohesive, independent process interacting via messages.*" These definitions are slightly different, but they have the same idea of independent process / service.

When we talk about microservice, we usually mean Microservice Architecture (MSA). This architecture differs from the one called monolithic (described in the section before). Figure 2.1 illustrates the differences between these architectures.

The main difference we can observe is the following : The monolithic architecture is composed of a "single large block" that contains all the components, whereas the microservice architecture is broken down into many "smaller blocks". These "blocks" represent the application. Therefore, each microservice manages all the components it needs (interface, business layers and databases). In [3], Fowler and Lewis define nine characteristics. The next subsections will explain what these features are. After that, in section 2.4, we will discuss advantages and disadvantages of microservices.

2.3.1 Componentization via Services

An application is composed of several parts. One main reason for using services is that if you change a part, you just have to redeploy the affected service and not all the application (if the interface of the latter is not modified). Another reason is that you can reuse service within multiple applications. e.g., a service that handles message encryption may well be used by application A and application B.

But this has repercussions on trade. They are no longer done internally and so it is necessary to use more or less heavy communication protocols depending on the resources available or the need (security for example). One solution for a service is to expose an API so that it can be integrated.

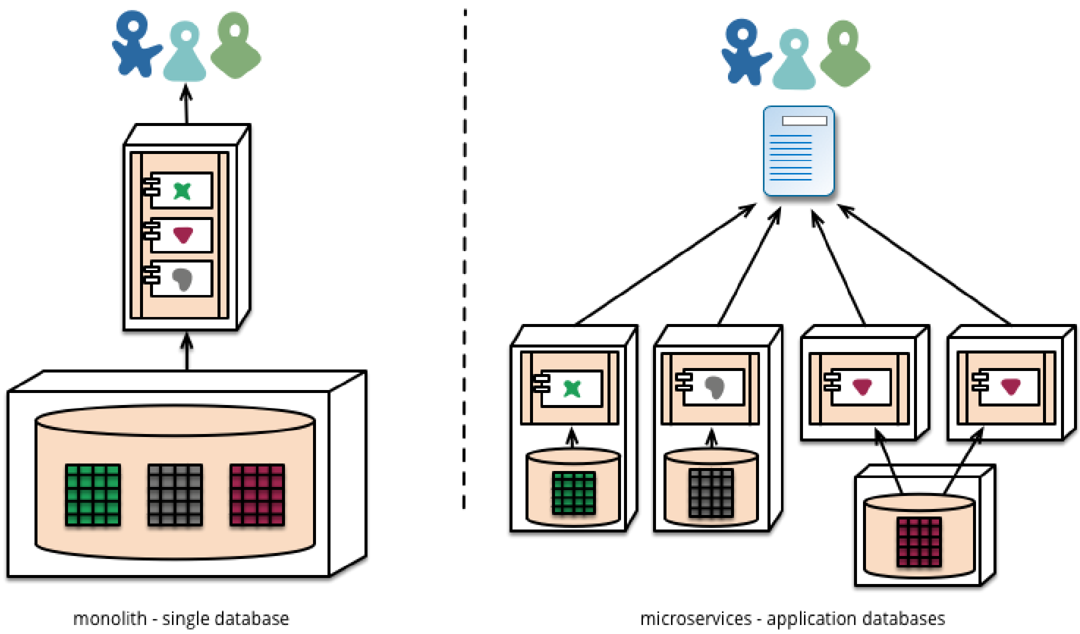


Figure 2.2: Differences between monolithic architecture and MSA database viewpoint [3]

2.3.2 Organised around Business Capabilities

Team building is done in a different way. There is no longer a front team and a back team, but the profiles are mixed between the teams. Each team is cross-functional, including the full range of skills required for the development: user experience, databases, and project management.

2.3.3 Products not Projects

It is no longer a model where the team carries out a part of a project, but the service becomes a project in its own right. It must be able to run properly, and the team that develops it becomes responsible for it.

2.3.4 Smart endpoints and dumb pipes

Applications built from microservices aim to be as decoupled and cohesive as possible, receiving a request, applying the appropriate logic and producing a response. To this end, communication can be based on REST protocols. Another common approach is messaging over a lightweight message bus. The most important thing is to have well-defined endpoints.

2.3.5 Decentralised Governance

Dividing the different components into smaller services offers possibilities to build them according to your wishes and needs. If one team prefers to use Node.js to create a simple report page or another prefers to use C++ for a particularly complex real-time component, that's possible too. It is also possible to replace one database with another as needed. Of course, just because it's possible to do so doesn't mean it should be done, but it does offer possibilities.

2.3.6 Decentralised Data Management

The use of databases for microservices is another point of variation from monolithic applications. It is no longer a question of sharing a single database with a multitude of tables and columns for the whole app, but the approach here is to let each service manage its own database. Figure 2.2 shows this difference. One of the advantages of this is that using poly stores is easy. There is no problem using a SQL database for one service and a Redis database for another.

However, this has other consequences. If two services have related data, then verification systems will need to be put in place to ensure that the contact remains in a consistent state. One approach that can be used is to use transactions, but that requires time coupling, which is something that should be avoided in microservices. It is better to aim for coordination between the different services.

2.3.7 Infrastructure Automation

Automation of the infrastructure is the logical extension of continuous deployment. Since there may be a lot of deployments, there is a need for an automated system to make this easier.

With the emergence of the cloud and services like Amazon Web Service (AWS), this has become a common task.

2.3.8 Design for failure

When using microservices, each component must expect the worst. It is possible that another service will not respond, that a call will fail, and these use cases must be managed. This is obviously a disadvantage because it adds complexity.

This problem is not specific to microservices but this is all the more important because of the division of services. In [5], the authors propose several patterns to react to these failures, such as *Retry pattern*, *Circuit Breaker pattern* or *Fail Fast pattern*. Once again, there are several ways to react, like *Probabilistic Model Checking*.

Since services can fail at any time, it's important to be able to detect the failures quickly and, if possible, automatically restore service.

2.3.9 Evolutionary Design

Thinking of the application as being modular allows it to evolve more easily. It is possible to add or remove a part of the application at any time. e.g. if I want to change my billing service, it just needs to have the same interface as the previous one. Or if I want to add a temporary feature to my website, I can simply develop a service that will do it.

2.4 Pros and cons

Of course this architecture has its pros and cons as already briefly explained previously. This part aims at making a small summary of these elements. This paper [6] also gives some ways of thinking about it.

2.4.1 Pros

Here is a small (non-exhaustive) list of advantages of microservices :

- **Technology diversity** : As mentioned in the point 2.3.5, there is the possibility to build services with different languages.
- **Scalability** : Each part of the system is designed separately. So, if one part of the system (a service) has to be updated, changed, duplicated, etc. it's easy to do without having to restart the whole system.
- **Independent Deployment** : As each service is independent, it's a smaller module, so it can be deployed independently in any application. Independent deployment of individual services also makes continuous deployment possible, as mentioned in the point 2.3.7.
- **Fault tolerance** : As mentioned in the point 2.3.8, the system is prepared and knows how to respond if a problem occurs.
- **Fault isolation** : Linked to the point right above, as the system knows how to react to a failure, errors are not propagated between departments.
- **Easyness of verification** : It is simpler to verify and test a service instead of verifying a whole application.

2.4.2 Cons

There is a small (non-exhaustive) list of disadvantages of microservices :

- **Multiplication of calls and traffic** : Everything is no longer at the same place, but is split into different containers. They requires to interact with each other. This has an impact on the number of ports which required to be open.
- **Requires More Resources** : As mentioned just above, as services runs in different containers, you have to run them all at the same time. This requires more resources than running a single application.
- **Relatively Complex** : Once again, you don't have an only service. Coordination is required to deploy all services in all containers, make them communicate, etc.
- **Microservices architecture** : This architecture can become complicated in terms of cutting. At what point do you have to divide it into several services ?

2.5 Examples of applications using microservices

There are many applications that use microservices. On the one hand, there are some example applications to illustrate the architecture but not use in a real case. We can easily find a sock shop [7], an online boutique [8] or a train ticket for example [9].

On the other hand, there are many famous companies using Microservice Architecture. This is particularly the case for Uber [10], Amazon or Netflix [11]. Sometimes, they can combine up to 1000 services [10] !

But they are not the only ones to migrate from monolithic architecture to MSA. It is also the case of a bank [12] or an industrial system [13]. Finally, another example is this one : the migration of an event-driven integration platform [14].

2.6 Conclusion

This chapter has provided a brief overview of what microservices are, what needs they can meet and their characteristics. Also, we saw few examples of real applications.

This architecture offers some interesting things, but it can also be useful to perform checks on these different services. Just as unit tests allow to check the behaviour of a process and an integration tests the behaviour of two processes together, it is necessary for microservices to see if they can interact together. To do this, the rest of this document will focus on the tools that already exist to see what can be learned from them, how they are useful and whether there are any shortcomings.

Chapter 3

Modelling and verification tools for microservices

In the literature, some models, some tools and some languages already exist to describe and reason about microservices. They are all different, they do not model the same things, and they all have their advantages and disadvantages. This chapter presents a brief review of what has already been done and explains what these models, tools and languages are intended to do.

Analysing these different tools gives a more general overview of what they offer and whether they meet my objectives.

3.1 Tools

These tools are drawn from various research and related publications. They do not all have the same usefulness and therefore, for each one, a summary of how it works is made with a small conclusion for each tool that says whether or not the tool can meet the objectives.

3.1.1 µTOSCA

µTOSCA is a toolchain to mine, analyse and refactor microservice-based architectures [15]. The µTOSCA toolchain is composed of three main elements :

- The TOSCA model (an OASIS standard)
- µMINER
- µFRESHENER

The tool is used to specify microservices-based architectures and produces a result, in the form of a typed topology graph [16]. It analyses the application to identify its different parts, the different components, based on the Kubernetes configuration file. This analysis allows, among other things, to detect if there are architectural smells.

But how does it work ? First of all, the system tries to recognise some patterns to identify parts of the model and create corresponding nodes. There are three main types of nodes :

- **Service** : Component running some business logic
- **Communication pattern** : Component implementing a messaging pattern. We can distinguish between two types of communication :
  - **MessageRouter**
  - **MessageBroker** : Can be either *synchronous* or *asynchronous*
- **Database**

Nodes can be connected together with arrows which define the meaning of communications. The model can also be enriched by some boolean tags (circuit_breaker, timeout and dynamic_discovery). Finally, the public gateway (public endpoint of the application) is defined as the edge.

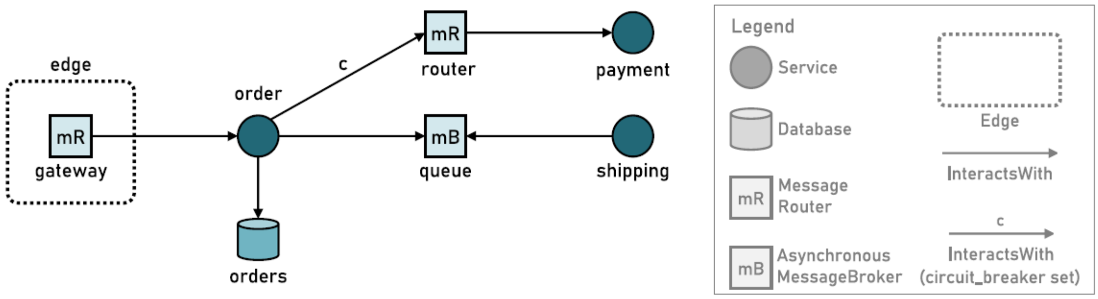


Figure 3.1: An example of architecture modelled with µTOSCA [15]

Figure 3.1 provides an example of a simple application. There is a router with access to the global network (the gateway) who interact with the order component. This component can interact with a database and with two other components using different routers. The little particularity is that there are two types of routers (one synchronous and the other asynchronous) and for the router for the payment, there is a circuit_breaker. This means that there is a management in case of failure.

Why does this tool not meet the objectives ?

This tool is focusing on interactions, but the main objective is to detect some architectural smells which potentially affect the application. These communications are modalised by arrows without considering the actual communication protocols, and the results of communications are not evaluated. The tool aims to propose an automated process to bring some possible solutions for these smells.

My objectives are to see if two components can interact together and handle all possibility to have a system who lives in any cases. This tool offers some possibilities, but it is not enough.

3.1.2 GSMART

GSMART is the acronym of Graph-based and Scenario-driven Microservice Analysis, Retrieval and Testing. GSMART enables the automatic generation of a "Service Dependency Graph (SDG)" by which to visualise and analyse dependency relationships between microservices as well as between services and scenarios. It also enables the automatic retrieval of test cases required for system changes to reduce the time and costs associated with regression testing [17].

GSMART works in three main phases :

- Extract information about the structure to construct a Service Dependency Relationship (SDR);
- Check status of service tests and generate test code;
- Combine information to create a SDG.

It's useful because it makes the difference between service and endpoint. The model symbolises interactions and allows us to detect cyclic dependency. It makes the difference between weak and strong cyclic dependencies.

- **Weak dependency** : A cycle occurs among multiple services, but not among multiple endpoints. It is a possible design error that could lead to resource dependency and competition.

- **Strong dependency** : A cycle occurs among multiple endpoints, which could cause unlimited service calls leaning to system malfunction or crash.

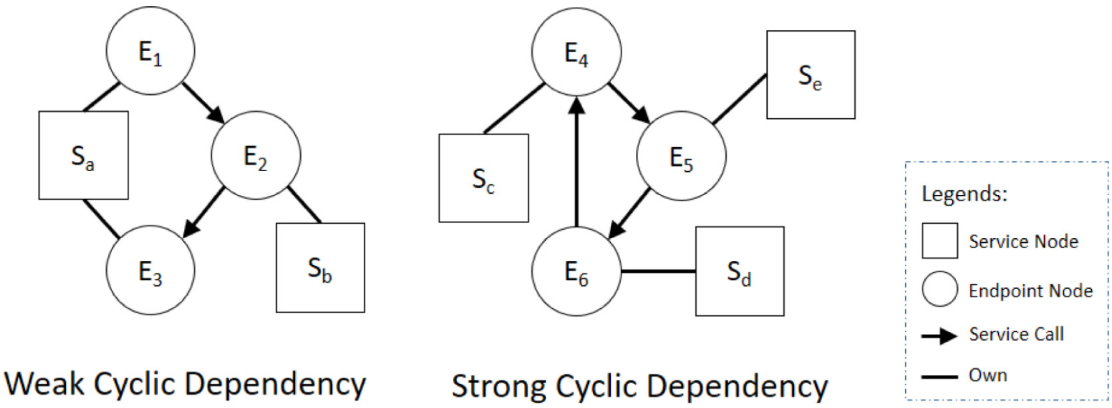


Figure 3.2: Concept illustration of cyclic dependency [17]

Figure 3.2 shows the difference between weak and strong dependency. For the weak dependency, on the left, we can see that endpoint E_1 calls E_2 which in turn calls E_3. There are some dependencies between these endpoints. The scenario is a bit different for the strong cyclic dependency on the right, where E_4 calls E_5 which calls E_6 which calls E_4. There is a cycle and this might cause some unexpected problems such as malfunction or crash.

GSMART mainly supports Java-based Spring Boot Framework and uses Gherkin [18], [19] (a behaviour-driven development language [20]) documents to specify test scenarios and service requirements.

Why does this tool not meet the objectives ?

This tool is limited to a technology based on Java. Also, the tool proposes an analysis oriented towards dependencies and not communications between the different components. It is therefore not possible to check that the executions are correctly terminated and that the expected behaviour is respected.

3.1.3 ucheck

The paper [21] proposes a system, `ucheck`, that checks and enforces invariants in microservice based applications. It works as follows : ucheck assumes that users provide it with a model for each microservice, which specifies the set of messages a microservice can send and how its local state changes in response to it. Given these models, the approach is to use formal verification to check whether a given invariant would hold based on the models and, then, use programmable virtual switches (e.g. [22], [23]) to detect cases where a microservice's behaviour deviates from his model.

Also, with the given specifications, it computes a set of possible interactions and when a service sent a message, it checks if an equivalent can be found in the set of plausible messages. If not, it raises an exception to indicate that.

Why does this tool not meet the objectives ?

The model cannot access the state and cannot detect all invariant violation. The model requires some specifications that are given through a model. Moreover, the model is based more on verification than on analysis of communications. An objective of the thesis is to understand and view the interactions between components. Here, it is just a formal verification that the service is working well. This tool remains interesting, but is limited to certain aspects.

3.1.4 MICO

The paper [24] proposes a model-driven and pattern-based approach for composing microservices, which facilitate the transition from architectural models to running deployments. Using a unified modelling for MSAs (Micro Services Architectures), including both their integration based on Enterprise Integration Patterns (EIPs) and deployment aspects, the approach enables automatically generating the artefacts for deploying microservice compositions.

MICO (acronym for Microservices Composition approach) works as follows : MICO unifies and synergically combines the architectural, integration, and deployment aspects of MSA-based applications, enabling a "one-click" transition from modelled integration of microservices to running deployments.

This approach is based on four main steps. Figure 3.3 illustrates this process.

1. **Import application** : Create the microservices' composition.
2. **Define integration model** : Define how microservices can communicate.
3. **Generate deployment artefact** : Transformation from the MICO model to obtain a deployable artefact to run it easily. This can pass by a container or an orchestration platform.
4. **Deploy applications instances** : The deployment artefact generated just before is launched.

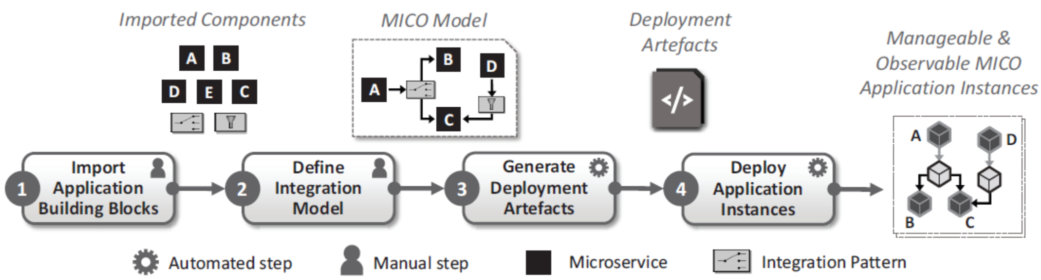


Figure 3.3: Overview of the pattern-based microservice composition approach [24]

MICO uses Kubernetes as container orchestration and OpenFaaS [25] for managing integration logic.

Why does this tool not meet the objectives ?

This tool is focussed on facilities to make a transition from architectural models to running deployments, and not analysis communications between components. The tool does not analyse the different communications for information, it only provides a support tool, not a verification tool.

3.1.5 LEMMA

LEMMA presents a modelling method that systematises SAR (Software Architecture Reconstruction) of microservice architectures with the goal of facilitating its execution [26].

SAR is an iterative reverse engineering process that derives a representation of software architecture from artefacts like documentation or source code [27]. It aims to document architecture implementations, which lacks thorough documentation and enables subsequent architecture analysis. SAR consists of four phases [27]:

- **Raw view extraction** : Extracts some architecture information from architecture-related artefacts.
- **Database Construction** : Transforms this information and stores it in databases.

- **View fusion and manipulation** : Combines views (and information) to improve the accuracy.
- **Architecture analysis** : Analyses the architecture to try to answer some hypothesis.

All of this depends on different viewpoints which show different information and target different profiles :

- **Domain viewpoint** : This viewpoint focuses on the domain and is useful to construct domain models.
- **Technology viewpoint** : This viewpoint focuses on service developers and operators.
- **Service viewpoint** : Based on Service Modelling Language [28], this view targets the Dev perspective in DevOps-based MSA teams [29].
- **Operation viewpoint** : This viewpoint targets operation perspective and helps to construct operations models that describe service deployment and infrastructure.

An automated process will analyse documents, trying to recover a maximum amount of information to create a coherent architecture description.

Figure 3.4 illustrates the SAR modelling method by enlightening the 6 steps. The process is repeated for each file or file set.

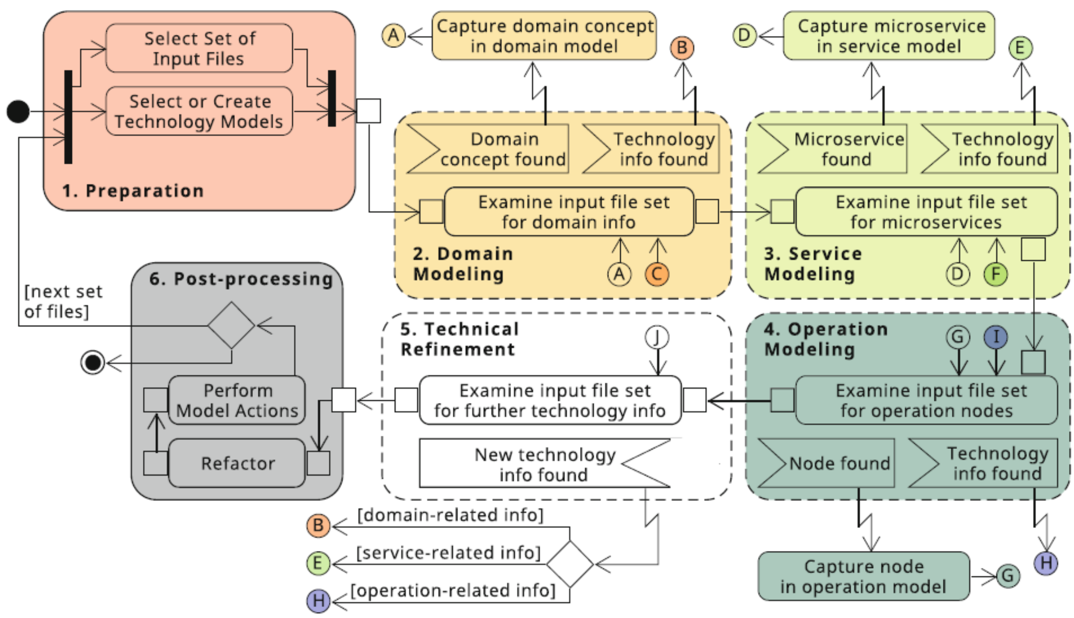


Figure 3.4: Definition of the SAR modelling method in a UML activity diagram. [26]

Why does this tool not meet the objectives ?

The objective of this tool is to present a modelling method to facilitate the SAR construction and allows an easier analysis. It does not provide a way of visualising and understanding communications to see if the services can work well together.

3.1.6 Jolie

Jolie is a language-based approach to the engineering of microservice architectures [30]. It works with different components :

- **Interfaces** : An interface describes the functionalities of the component. Interface is composed of a set of operations that can be remotely invoked. This interaction can be *synchronous* or **asynchronous**. Since interfaces are sets of operations, it has been natural to consider the usual set-theoretical operators, such as union and intersection.
- **Ports** : Microservices can run on different environments and different technologies. A communication port specifies three key elements : *interface*, *communication technology* and *data protocol*. Each service has one or many ports, with two kinds possible : *input* and *output*.
- **Workflows** : Service interactions may require to perform multiple communications. e.g. component A makes a request to the component B who needs to contact component C to respond. But unfortunately, the programming of such workflows is not natively supported by mainstream languages.
- **Processes** : Even though the workflow defines the behaviour of the service, at runtime, the service may interact with multiple clients and other external services. A process is a running instance of a workflow, and a service may include many processes executing concurrently.

The Jolie language is an imperative language that targets microservices directly. It helps to construct the different concepts. These concepts are separate, so it's simple to change the communication (or other) without changing anything else.

The language based approach allows having a better understanding of the service, by the specifications of the four artefacts explained before.

Why does this tool not meet the objectives ?

This specification is based on the four aspects explained just before, and does not focus on interactions between components.

3.1.7 Conclusion

All the models and tools presented above offer interesting features, but none of them offer something that meets all objectives. And so, in order to meet the different objectives explained at the beginning, it is necessary to look further to find something that would be easier to adapt. This is why the next part explores the different modelling languages.

3.2 Modelling languages

This second part aims at looking at several languages proposed in the literature to see what possibilities they offer. They all have their strengths and weaknesses.

The aim here, in the first instance, is not to compare them but to see what they offer to choose the one that best suits the objectives of the thesis.

3.2.1 Event-B

Event-B is a formal method for system-level modelling and analysis. Key features of Event-B are the use of set theory as a modelling notation, the use of refinement to represent systems at different abstraction levels and the use of mathematical proof to verify consistency between refinement levels [31].

An Event-B machine consists of a collection of variables, invariants on those variables, and a collection of guarded events that may update the machine variables. An Event-B development consists of a collection of machines linked by refinement [32].

Event-B is an evolution of B (also known as classical B). It is a simpler notation, which is easier to learn and use.

The basic features of B are [33]:

- To parse formal texts according to a certain surface syntax that corresponds to the traditional structure of either mathematical formulae or program fragments.

- To recognise in formal texts the presence of quantifiers and of variables.
- To analyse the non-freeness of variable occurrences with respect to quantifiers.
- To perform (in a formula) the multiple replacement (substitution) of free occurrences of certain variables by some other formulae (provided this is safe from the point of view of variable capture).
- To recognise in formal texts the presence of certain meta-variable occurrences (each meta-variable stands for any formula).
- To pattern match a formula against another one containing such meta-variables. A successful pattern matching results in the construction of a filter assigning a formula to each meta-variable.
- To instantiate a formula by means of a filter (that is, to replace the meta-variables of the formula in question by the formulae assigned by the filter).

3.2.1.1 The syntax

The primary concept in doing formal developments in Event-B is that of a model. A model contains the complete mathematical development of a Discrete Transition System. It is made of several components of two kinds : machines and contexts [34]. The Figure 3.5 illustrate this.

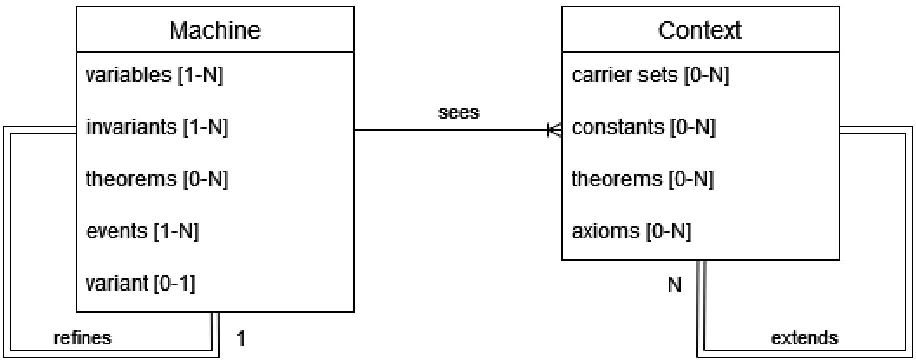


Figure 3.5: Model of Event-B

As we can see, the machine is composed of a lot of things and, among others, a set of events. An event represents a transition. It is essentially made of guards and actions. The structure of an event can be defined as follows :

$$< event\_identifier > \hat{=}$$
$$\textbf{status}\ \{normal,\ convergent,\ anticipated\}$$
$$\textbf{refines} *\ < event\_identifier >\ ...$$
$$\textbf{any} *\ < parameter\_identifier >\ ...$$
$$\textbf{where} *\ < label >:< predicate >\ ...$$
$$\textbf{with} *\ < label >:< witness >\ ...$$
$$\textbf{then} *\ < label >:< action >\ ...$$
$$\textbf{end}$$

The guards (expressed with the keywords **where** just above) are optional and work like preconditions. The action defines how the state is affected. The keyword **where** defines parameters, and **with** is used when an event refined another one.

An action can be deterministic. In which case, there is a list of distinct variable identifiers structured like this :

$$< variable\_identifier\_list > := < expression\_list >$$

The second list represents a set of expressions which have the same size as the first list. Only variables in the list are modified.

Alternatively, an action can be non-deterministic :

$$< variable\_identifier\_list > : | \ < before\_after\_predicate >$$

The *before_ after_ predicate* can refer to the (variable) state of the machine before the action is performed. It can also define predicates that must be checked after the action has been performed.

You can specialise this non-deterministic notation :

$$< variable\_identifier > :\in < set\_expression >$$

This form is just a special case of the previous one. It can always be translated to a non-deterministic case.

When a specialisation (*refine*) is performed, all specialised elements must receive this value. Each element is then defined by a predicate. When the concrete event is also parameterised, then an abstract parameter which is the same as a concrete need not be given an explicit witness for one reason : it is always the corresponding concrete parameter.

3.2.1.2 Example with Event-B

This is a small example of a context represented with Event-B. The example is taken from [35]. It is divided in two parts :

- The description of the context
- The description of the machine (which requires the context to work)

The description of a context

This defines a context name `Celebrity_c0` with three constants `k, c and P`. The various axioms are mathematical formulae that define invariants for the context. They must always be respected.

```
CONTEXT
        Celebrity_c0
CONSTANTS
        k
        c
        P
AXIOMS
        axm1:   P ⊆ ℕ
        axm2:   c ∈ P not
        axm3:   k ∈ (P ∖ {c}) ↔ P
        axm4:   k∼ [{c}] = P ∖ {c}
        axm5:   k∩(P ◁ id)=∅ n
END
```

The description of a machine

This machine evolves in the context define just above and has a name `Celebrity_0`. One more variable is defined with an invariant to specify the kind of variable. Here, we know that $r \in P$ and $P \subseteq \mathbb{N}$. The different events specify the comportment of the machine.

```
MACHINE
    Celebrity_0
SEES
    Celebrity_c0
VARIABLES
        r
INVARIANTS
        inv1:   r ∈ P not theorem
EVENTS
        INITIALISATION:
        THEN
                act1:   r :∈ P
        END

        celebrity:
        THEN
                act1:   r := c
        END
END
```

Why does this language not meet the objectives ?

This language allows modelling some process and their context, focussed on mathematical notation. But the syntax is really long, and the descriptions are tedious to do with many things to describe. Here I am looking for a simple and fast specification to be able to model the whole system easily.

3.2.2 Process algebra

According to [36], the term process algebra is used in different meanings. It refers to the behaviour of a system, in particular the execution of a software system, the actions of a machine or even the actions of a human. It offers the possibility to describe this behaviour with an algebraic / axiomatic approach.

π-calculus and CCS are examples of process algebras. There are others such as CSP and ACP. Here, I will have a focus on π-calculus and CCS.

3.2.2.1 Calculus of Communications Systems

Introduced by Robin Milner (like π-calculus) around 1980 [37], CCS is very close to the π-calculus. It's a process calculus, and its actions model indivisible communications between exactly two participants.

CCS is useful for evaluating the qualitative correctness of properties of a system, such as deadlock or livelock [38].

The syntax

Given a set of action names, the set of CCS processes is defined by the following BNF grammar [39] :

$$
\begin{aligned}
P ::= {} & 0 \\
& \mid \alpha.P_1 \\
& \mid A \\
& \mid P_1 + P_2 \\
& \mid P_1 \mid P_2 \\
& \mid P_1[b/a] \\
& \mid P_1 \backslash \alpha
\end{aligned}
$$

The parts of the syntax are :

- **The inactive process** : 0
- **An action** : $\alpha.P_1$ defines that α is executed and then, the process P_1 continue. An action can be :
  - **input** : a
  - **output** : $\overline{a}$
  - **silent action** : τ
- **Process identifier** : A is the name of a process. It is defined by an equation of the form $A \stackrel{\text{def}}{=} P_1$
- **Alternative** : Symbolised by the +, the process P_1 or P_2 can be proceeded according to the one which can do a step
- **Parallel composition** : P_1 and P_2 execute in parallel, either in an interleaving fashion or simultaneously
- **Renaming** : $P_1[b/a]$ is the process P_1 with all actions a renamed b
- **Restriction** : $P_1 \backslash \alpha$ is the process P_1 which cannot produce the action α

Why this language not fit the objectives ?

Here, there is no possibility to exchange data between components during the communications of actions, which makes it difficult to model communications.

However, some transition rules of the language and the properties it allows checking are interesting. It is a good base to extend.

3.2.2.2 π-calculus

The π-calculus is a process calculus. One strength of the π-calculus is that it allows channel names to be communicated along the channels themselves. It is not a programming language for producing executable programs, but it is used to understand its concepts.

The main difference with CCS is that π-calculus offers the possibility to have multiple participants.

This language is minimal. It's a more mathematical formalism which allows describing and analysing properties of concurrent computation.

The syntax

The syntax of the π-calculus is quite simple. It is composed by these following elements :

- **A term** : defines a thread (P, Q ...)
- **Communication** :
  - **input** : $c(x)$: receives on channel c and binds the value to x
  - **output** : $\overline{c}\langle y \rangle$: sends the value y over channel c
- **Concurrency** : $P \mid Q$: runs P and Q simultaneously
- **Replication** : $!P$
- **Creation** of a new name : $(\nu x)P$
- **The end** of a process (nil) : 0

Example with π-calculus

Just below, there is a small example. There are three unnamed threads which communicate on the channel x (only known by the first two components) [40] :

$$\begin{aligned}(\nu x)\ (\ & \overline{x}\langle z\rangle.\ 0 \\ & |\ x(y).\ \overline{y}\langle x\rangle.\ x(y).\ 0\ ) \\ & |\ z(v).\ \overline{v}\langle v\rangle.\ 0\end{aligned}$$

After one step :

$$\begin{aligned}(\nu x)\ (\ & 0 \\ & |\ \overline{z}\langle x\rangle.\ x(y).\ 0\ ) \\ & |\ z(v).\ \overline{v}\langle v\rangle\rangle.\ 0\end{aligned}$$

The first two components communicate on the channel x. The name y becomes bound to z. The second and third parallel components can now communicate on the channel name z. The next step is :

$$\begin{aligned}(\nu x)\ (\ & 0 \\ & |\ x(y).\ 0 \\ & |\ \overline{x}\langle x\rangle.\ 0\ )\end{aligned}$$

The third process received x and bound the name. The channel x can be used for sending the name x. After that, all concurrently executing processes have stopped and the process is finished :

$$\begin{aligned}(\nu x)\ (\ & 0 \\ & |\ 0 \\ & |\ 0\ )\end{aligned}$$

Why does this language not meet the objectives ?

Just like CCS, this language offers a very good base for a reflexion, but there are no differences on the different types of communication. We don't know how to distinguish errors, failures, specify that the process can crash, etc. This language is a little limited and needs to be extended to perfectly fit the objectives.

Here, we can only say that there are exchanges, but we do not know what kind of exchanges are. This language offers a model, but it had to be completed with a methodology. It is the reason why, later, I will add some features to this language so that it can meet all the objectives.

3.2.3 Rules for the transition system

The transition rules of the system are very important because they help us to define how the language behave. They are based on π-calculus transitions rules [41].

The table 3.1 defines the operational semantics and different rules.

Process call If the process P is called, the process is evaluated.

Prefixing If an element α prefixes another, then a transition can be triggered and can, potentially, affect the state of the process.

Parallelism, Interleaving The parallel composition of two processes can do a step if one of them can do so.

Type of rule	Rule
Process call	$P = A$ $\frac{A \xrightarrow{\alpha} B}{P \xrightarrow{\alpha} B}$
Prefixing	$\alpha\ .\ P \xrightarrow{\alpha} P'$ $\tau\ .\ P \xrightarrow{\tau} P$
Parallelism, Interleaving	$\frac{P \xrightarrow{\alpha} P'}{Q \mid P \xrightarrow{\alpha} Q \mid P'}$ $\frac{Q \xrightarrow{\alpha} Q'}{Q \mid P \xrightarrow{\alpha} Q' \mid P}$
Parallelism true concurrency	$\frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\overline{\alpha}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$
Alternative	$\frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'}$ $\frac{P \xrightarrow{\alpha} P'}{Q+P \xrightarrow{\alpha} P'}$
Replicas	$\frac{P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} !P \mid P'}$

Table 3.1: Language's operational semantics

Parallelism true concurrency If there is a transition α on the process `P` and its complementary on the process `Q`, both transitions are performed, by the parllel composition of the two processes, yielding a silent step.

Alternative If there is an alternative and there is a transition by one of the processes, then this transition is performed, regardless of the place of the process in the alternative.

Replicas If a rule applies to a process, this rule also applies to all his replicas.

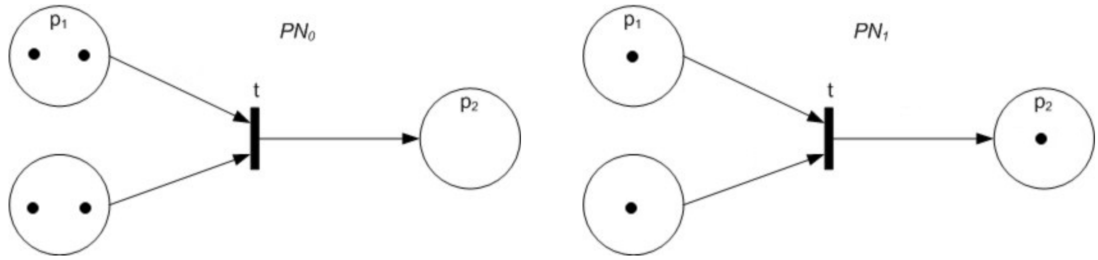
3.2.4 Petri net

A Petri net is a graphical tool for the description and analysis of concurrent processes which arise in systems with many components (distributed systems) [42]. It allows you to see and understand the system differently. It is also possible to define a Petri net as a mathematical modelling language for the description of distributed systems.

Petri nets elements are composed of only 4 components :

- **Places** : Represent the different possible states of the modelled application. Places may contain a discrete number of marks called *tokens*.
- **Transitions** : Define how we can move from a state to another one. All places coming in a transition must have a token to trigger the transition. If it's not the case, the transition cannot be triggered.
- **Arcs** (directed) : Arcs run from a place to a transition or vice versa, never between places or between transitions.
- **Tokens** : They indicate whether transitions can be triggered or not. There may be several in a place.

Figure 3.6 illustrates a very simple net with 3 places and a transition. The transition t can be triggered because there are tokens in both places arriving at the transition. The transition is triggered and a token is pulled from places on the left and pushed on the place P_2.



(a) Petri net before the transition (b) Petri net after the transition

Figure 3.6: Example of a Petri net

3.2.4.1 Notation for Petri nets

A basic Petri net is a triple (P, T, F) where P is a finite set of places, T is a finite set of transitions $(P \cap T * \emptyset)$, and $F \subseteq (P \text{ x } T) \cup (T \text{ x } P)$ is a set of arcs (flow relation) [43].

This definition can be extended to an extended Petri net as a tuple (P, T, F, W, A, L, R, H), where :

- (P, T, F) is a basic Petri net
- $W \in F \rightarrow \mathbb{N}\backslash\{0\}$ is an (arc) weight function
- $A \subseteq A$ is a set of (activity) labels,
- $L \in T \rightarrow A \cup \{T\}$ is a labelling function,
- $R \in T \rightarrow 2^P$ is a function defining reset arcs, and
- $H \in T \rightarrow 2^P$ is a function defining inhibitor arcs.
- M represents a state and M_0 is the initial state of the net.

This extension of Petri nets allows having more data on the net and allow checking more properties than before the extension.

3.2.4.2 Mathematical properties of Petri nets

One thing that makes Petri nets interesting is that, like process algebras, they have some properties that allow you to check certain things. But the main interest of Petri Nets is the ease of understanding they provide. Just below, there are 3 important properties which are possible to prove with Petri nets.

Reachability

As explained in [44], the central algorithmic problem for Petri nets is reachability : whether from the given initial configuration there exists a sequence of valid execution steps that reaches the given final configuration.

Proving reachability is not always easy. One way is to use the semi-decision technique to find if indeed a state can be reached, by finding a set of necessary conditions for the state to be reached then proving that those conditions cannot be satisfied [45]. Another algorithm can be found on [46].

Liveness

According to [47], the concept of liveness is closely related to the complete absence of deadlocks in operating systems. A Petri net (N, M_0) is said to be live if, no matter what marking has been reached from MO, it is possible to ultimately fire any transition of the net by progressing through some further firing sequence.

This means that a live Petri net guarantees deadlock-free operation, no matter what firing sequence is chosen. If we can prove this, we prove that the system cannot be blocked. It can be very important to prove this property for critical systems.

Boundedness

Once again according to [47], a Petri net (N, M_0) is said to be k-bounded or simply bounded if the number of tokens in each place does not exceed a finite number k for any marking reachable from M_0, i.e., $M(p) <= k$ for every place p and every marking $M \in R(M_o)$. A Petri net (N, M_O) is said to be safe if it is 1-bounded.

This means that for a 2-bounded net, there will never be more than 2 tokens in one place. By verifying that the net is bounded or safe, it is guaranteed that there will be no overflows in buffers or registers, no matter what firing sequence is taken.

Some properties presented above can be checked with invariants [48].

But there are also several subtypes of Petri Net. The following parts explain some of them and give some of their advantages.

3.2.4.3 Open Petri net

Open Petri net is a specialisation of Petri Net. According to [49], an "open" Petri net is one 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 with the inputs of another.

Open networks remain very similar to "classical" networks but with one major difference: Inputs and outputs can be added to the networks, they are no longer closed. This makes it easier to connect networks together.

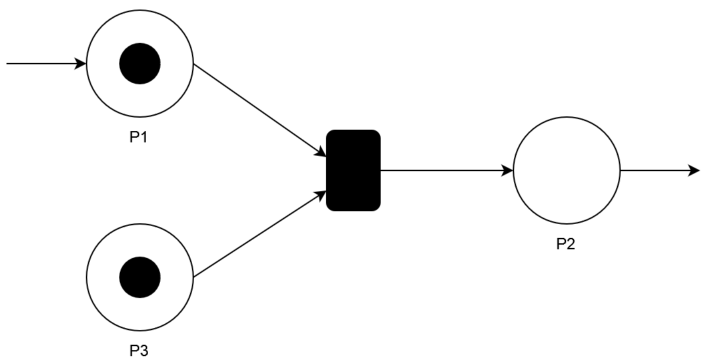


Figure 3.7: Example of an Open Petri net

Figure 3.7 illustrates quite the same net as before, but with input and output places clearly defined. The place P_1 is an input and the place P_2 is an output.

3.2.4.4 Workflow net

Workflow nets (WF-nets) are a subclass of Petri nets, intending to model the workflow of process activities [50]. The behaviour is similar to "classical" Petri nets, but here, transitions represent the tasks or activities, and places are states between these actions.

They have become one of the standard ways to model and analyse workflows. Typically, they are used as an abstraction of the workflow that is used to check the so-called soundness property (defined below). This property guarantees the absence of livelocks, deadlocks, and other anomalies that can be detected without domain knowledge [51].

Some tools exist to model these workflow nets as WoPeD [52], for example.

A workflow net is an extended Petri net if and only if [43]:

- There is a single source place i , i.e. $\{p \in P \mid \bullet\, p = \emptyset\} = \{i\}$
- There is a single sink place o, i.e. $\{p \in P \mid p\bullet = \emptyset\} = \{o\}$
- Every node is on a path from i to o, i.e. for any $n \in P \cup T : (i, n) \in F^*$ and $(n, o) \in F^*$ where F^* is the reflexive transitive closure of relation F
- There is no reset arc connected to the sink place, i.e. $\forall_{t \in T} o \ni R(t)$

Soundness

According to [43], an extended Petri net is sound if and only if the following three requirements are satisfied :

- Option to complete : $\forall_{M \in R(N,[i])} [o] \in R(N, M)$. It is always possible to reach the final state $[o]$ from the initial state $[i]$.
- Proper completion : $\forall_{M \in R(N,[i])} (M \geq [o]) \Rightarrow (M = [o])$. If a token is put in place o, all the other places should be empty, as it is the end of the net.
- No dead transitions : $\forall_{t \in T}, \exists_{M \in R(N,[i])} (N, M)\langle t \rangle$. There can be no dead transitions in the initial state $[i]$. A dead transition is a transition that can never fire.

Another notation found on [53] :

- Option to complete : $\forall M, M_0 \rightarrow^* M : \exists M', M \rightarrow^* M' : M' \geq [o]$
- Proper completion : $M_0 \rightarrow^* \wedge M \geq [o] \Rightarrow M \equiv [o]$
- No dead transitions : $\forall t \in T$, $\exists M : M_0 \rightarrow^* M$ and t is enabled in M

3.2.4.5 Utility of Petri Nets

Petri Nets can be very useful for many things. Firstly, they provide a graphical representation of a situation. It's sometimes easier to understand a process with a graphical view. Secondly, some tools allow for checking certain properties (explained above) by using these nets (e.g. WoPeD).

3.2.5 Petri Net Markup Language

The *Petri Net Markup Language* (PNML) is an XML-based interchange format for Petri nets. PNML supports any version of Petri net since new Petri net types can be defined by so-called *Petri Net Type Definitions* (PNTD) [54]. PNML is a syntax for high-level Petri nets, which is designed as a standard interchange format for Petri net tools [55].

The design of PNML was governed by 3 principles : **readability**, **universality** and **mutuality**.

The version 2009 is the current active version of PNML grammar [56]. It defines multiple concepts like the definition of PNML Core Model, labels used by PT Nets nodes and arcs, PT Nets type declaration, etc.

3.2.5.1 The syntax of PNML

Here is an example of PNML of a very simple Petri net, represented on Figure 3.8. The code below was managed by WoPeD [52]. The following parts do not repeat the whole code, but explain the parts step by step. The whole generated model can be found in the annexe 7.1. Sections below only represent places, transitions and arcs. The formal syntax as the net or specific instructions for the tool does not explain here.

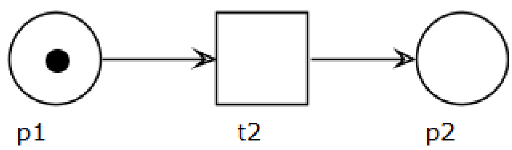


Figure 3.8: Very simple Petri net

A place

The representation of the place is :

```xml
<place id="p1">
  <name>
    <text>p1</text>
    <graphics>
      <offset x="40" y="80"/>
    </graphics>
  </name>
  <graphics>
    <position x="40" y="40"/>
    <dimension x="40" y="40"/>
  </graphics>
  <initialMarking>
    <text>1</text>
  </initialMarking>
</place>
```

The elements of this representation are :

- **Place - id** : Define the id of the place.
- **Name** : This represents the name of the place, and the **graphics** sub element is the positioning of the text (*p1*).
- **Graphics** : This represents the positioning of the place.
- **InitialMarking** : This is an optional element which defines the number of tokens at the start of the net.

A transition

The representation of the transition is :

```xml
<transition id="t2">
  <name>
    <text>t2</text>
    <graphics>
      <offset x="130" y="80"/>
    </graphics>
  </name>
  <graphics>
    <position x="126" y="40"/>
    <dimension x="40" y="40"/>
  </graphics>
  <toolspecific tool="WoPeD" version="1.0">
    <time>0</time>
    <timeUnit>1</timeUnit>
    <orientation>1</orientation>
  </toolspecific>
</transition>
```

The elements of this representation are :

- **Transition - id** : Define the id of the transition.
- **Name** : This represents the name of the place, and the **graphics** sub element is the positioning of the text (*t2*).
- **Graphics** : This represents the positioning of the place.
- **Toolspecific** : Utility for WoPeD.

An arc

The representation of the arc is :

```
<arc id="a2" source="p1" target="t2">
  <inscription>
    <text>1</text>
    <graphics>
      <offset x="500.0" y="-12.0"/>
    </graphics>
  </inscription>
  <graphics/>
  <toolspecific tool="WoPeD" version="1.0">
    <probability>1.0</probability>
    <displayProbabilityOn>false</displayProbabilityOn>
    <displayProbabilityPosition x="500.0" y="12.0"/>
  </toolspecific>
</arc>
```

The elements of this representation are :

- **Arc** : Define the arc with :
  - **id** : The id of the arc.
  - **source** and **target** : Represent the elements to which the arc is linked and define the direction of the arc.
- **Graphics** : This represents the positioning of the place.
- **Toolspecific** : Utility for WoPeD.

3.3 Conclusion

This chapter has allowed us to see what already exists, both in terms of modelling and analysis. Some tools are more oriented towards microservices, while others such as modelling languages are more general. Each tool has its advantages and disadvantages, but unfortunately, none of them completely corresponds to the objectives. It is therefore necessary to adapt these tools to be able to model and analyse microservices from another point of view.

Chapter 4

A process algebra based approach to model microservices

All models seen before are different, and they don't necessarily correspond to the objectives or only partially. A new model is therefore required. This chapter will present and explain how a π-calculus based language works and tests it to prove some properties. After that, a modelling based on different Petri nets will be used to prove other properties.

But before all, it is useful to introduce the subject on which the analyses are based. They are based on a sock shop [7].

4.1 The sock shop model

The sock shop is a standard in microservices examples. It is a very well-known project with a lot of documentation and tutorials. "Sock Shop simulates the user-facing part of an e-commerce website that sells socks. It is intended to aid the demonstration and testing of microservice and cloud native technologies." [7] Moreover, everything is open source, which makes it easy to break down and understand how the different services work. As this is an example project, not everything may represents reality, but it is still a functional project.

The following analysis does not include all the proposed services, but only some of them. The main objective is not to model the whole application but to understand how it is possible to model the services and interactions in order to check if the proposed model fit the objectives or not.

Why do we focus only on some services ? Here the objective of the thesis is to show that the model works. If it is possible to define that the model works with one component and continues to work with a second, it is possible to assume that no matter which component or components, the model will work. Moreover, it is easier to describe the whole research process by keeping the same components to have a certain coherent sequence between the different parts.

4.1.1 The queue

The queue is a very simple service. The service receives messages from a producer and can forward them to a consumer. The service therefore consists of an input and an output. The output can be the expected behaviour or an error message.

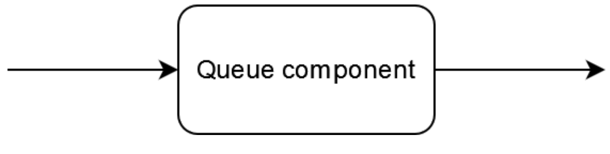


Figure 4.1: Overview of the queue

Figure 4.1 illustrates a simple overview of the component and its various possible connections. Here, the internal behaviour of the queue is not specified. It can be a first-in/first-out, as well as it can be a last-in/first-out. The thing we are interested in here is the communication with other components, not how messages are handled within the component.

4.1.2 The shipping

The shipping component remains really simple, but offers a bit more possibilities than the queue. It can receive a message and have multiple options :

- Directly responds without any particular processing
- Processes the message, contacts the queue and, in any case, responds (regardless of the response of the queue)

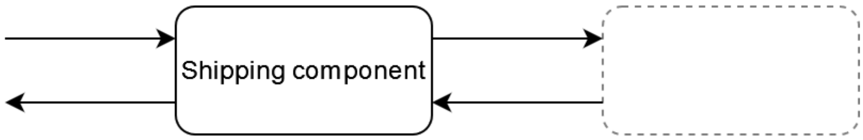


Figure 4.2: Overview of the shipping component

Figure 4.2 illustrates a simple overview of the shipping component and its various possible connections. The dotted shape represents another service (the queue) and the figure 4.3 illustrates how they can connect together.

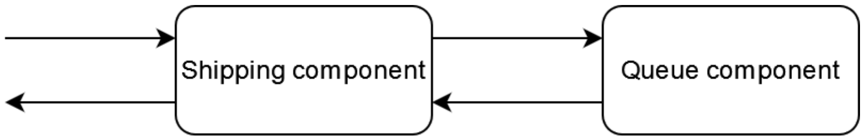


Figure 4.3: Overview of the shipping and the queue together

4.2 Different scenarios

For ease of understanding, it is easier to follow the same pattern of analysis throughout the document. This section will therefore present the two imagined scenarios.

First, a scenario without failure and, secondly, the same scenario with some additions of failures.

4.2.1 Failure-free scenario

This "failure-free scenario" corresponds to the expected behaviour of the application. The processes runs perfectly and there are no errors detected. This is the scenario for an utopian world where everything always goes as planned and the application never crashes.

The model takes the various components as they are designated to represent them as accurately as possible.

4.2.2 Failure-Aware scenario

As explained before, this scenario is **the same** as the one explained just before. There is only one major difference : the system can crash at **any** moment. What does it mean ? Here are some examples of a crash or unexpected behaviour :

- A service is running in a container, and this container gets killed by the Docker engine because of lack of resources. The service cannot respond anymore. If this happens during a process, it can be annoying.
- A service uses a database to store some information and there is a problem during the reading and there is no failure management. This is not the expected behaviour of the database.

4.3 π-calculus and CCS based language

As it was explained before, there are different models already existing with all their own benefits and disadvantages. But there is no language that perfectly fits to all objectives. Is it therefore useful to design a new language / extend an existing language. In order not to start from scratch, it is also useful to draw inspiration from well-known languages that have already proven themselves. The following sections therefore explain this language based on π-calculus (subsection 3.2.2.2) and CCS (subsection 3.2.2.1). Afterwards, a method is proposed to specify microservices in this new formalism to be able to test the different scenarios and see which properties can be checked.

4.3.1 Definition of the language

As π-calculus and CCS are both an inspiration, the syntax is very close to that of these languages, with a few minor differences :

- **Communications** are a little different
- **Errors** are clearly specified

The box below shows the syntax of the model.

$$\begin{aligned}
x? &\rightarrow Receive\ something\ on\ channel\ x \\
x?(y) &\rightarrow Receive\ name\ y\ of\ a\ channel\ on\ channel\ x \\
x! &\rightarrow Send\ something\ on\ channel\ x \\
x!(y) &\rightarrow Send\ name\ y\ of\ a\ channel\ on\ channel\ x \\
x?? &\rightarrow Receive\ ERROR \\
x!! &\rightarrow Send\ ERROR \\
\tau &\rightarrow Internal\ behaviour
\end{aligned}$$

The syntax is defined by these following elements :

- **A term** : Defines a process
- **Communication** : One of the major difference is here. A problem in previous language was that the types of communication were not differentiated. Here, it is the case and they are separates in two categories :
  - **Input** : It can be a normal message $x?$ or an error message $x??$
  - **Output** : It can be a normal message $x!$ or an error message $x!!$

  In both cases, input and output, the language makes the differences between some types of communications. But these communications are part of the expected behaviour. This error message is different than a failure. It is just another type of communications and will require a specific handle. A $x!$ will always match with a $x?$ and a $x!!$ will require a $x??$ to have a perfect match.
- **Concurrency** : $P \mid Q$: Runs P and Q in parallel
- **Replication** : $!P$
- **Alternative** : $+$: The process will choose the right path, according to the possible inputs
- **The end** of a process (nil) : 0

For now and the next part of the document, for all models, the usage of short names for components / services (e.g. `q = queue`, `s = shipping`, etc.) is a standard.

4.3.2 Rules for the transition system

To have a language that perfectly matches my expectations, it is necessary to add a transition rule : the process can stop at every moment (only in the failure-aware scenario). This is not present in CCS, but exists in CSP. Here, this is used to specify that the process can fail at any time in the failure-aware scenario.

This failure is different than an error. An error can be an exception raised, it's a specific type of message but this stop is a failure during the execution of the component. The state become δ, this means that the process cannot continue.

4.3.3 How to (compositionally) specify microservices in this model

This part provides guidelines for composing the model. Following these steps will ensure that nothing has been forgotten during the specification. It is possible to imagine an algorithm or a tool that would do all this automatically. These steps are described below. After that, an example of a transformation is carried out on the shipping component of the sock shop [7].

4.3.3.1 An algorithm to specify microservices

To define microservices, we need to analyse their behaviour step by step. To do this, there are several possible scenarios. Each element will have its correspondence.

- **Receive a message** : `s?` or `s??` if it is an error.
- **Send a message** : `s!` or `s!!` if it is an error.
- **Internal process** : Always modelled by τ.
- **Alternative** : `+`
- **Parallelism** : `|`

These transformations must be done for all possible execution paths of the process to obtain the complete model of the process.

4.3.3.2 Example

The code of the shipping component is open-source and can be found here [57]. The component is written in Java. The following will explain the different steps of the transformation.

The first part to be modelled is this one :

```
@RequestMapping(value = "/shipping", method = RequestMethod.GET)
public String getShipping() {
    return "GET ALL Shipping Resource.";
}
```

After that, another part will be modelled and will enhance the model.

Receive a message This is the first step for the creation : generate the input of the model. To do this, you need to identify the component inputs. Here, with Java Spring, they are marked with the `@RequestMapping` annotation.

This leads to a first model with only the input as content :

$$s?$$

Internal process The internal process also requires to be modelled. It is symbolised by the symbol τ.

This leads to this model :

$$s?\,.\,\tau$$

Send a message The service can send a message. This can be an invocation of another component or can be a response message. In both cases, the modelling is the same : $s!$.

This leads to this model :

$$s? \,.\, \tau \,.\, s!$$

Once these steps are done, we have a first model for the component. But the component is not just about that, you have to repeat these steps for the other possible access paths to construct the complete behaviour of the component. Here, the further modelling will be based on the following code :

```
1  @ResponseStatus(HttpStatus.CREATED)
2  @RequestMapping(value = "/shipping", method = RequestMethod.POST)
3  public
4  @ResponseBody
5  Shipment postShipping(@RequestBody Shipment shipment) {
6      System.out.println("Adding shipment to queue...");
7      try {
8          response = queue.convertAndSend("shipping-task", shipment);
9      } catch (Exception e) {
10         System.out.println("Unable to add to queue (the queue is probably
               down). Accepting anyway. Don't do this " +
11                 "for real!");
12     }
13     return shipment;
14 }
```

Receive message - line 2 Just like the previous example, the `@RequestMapping` annotation specifies the input.

$$s?$$

Internal process - line 6 After that, the next step is the internal process of the component. It doesn't affect the state of the component.

$$s? \,.\, \tau$$

Send message - line 8 This corresponds to the communication with the queue component.

$$s? \,.\, \tau \,.\, q!$$

Receive message - line 8 Here, the line 8 also corresponds to a response. This is the expected behaviour. It is therefore modelled by the expected behaviour.

$$s? \,.\, \tau \,.\, q! \,.\, q?$$

Send message - line 13 In case of success, after the `try`, the process will continue and respond to the caller.

$$s? \,.\, \tau \,.\, q! \,.\, q? \,.\, s!$$

Receive error (`e.g. QueueException`) - line 10 In case if the queue doesn't respond with an expected message, the process can catch an `Exception`. This will lead to construct an alternative with this error management. This is the first case of an error and will be noted `q??`.

$$\begin{aligned} s? \,.\, \tau \,.\, q! \,.\, (\, & q? \,.\, s! \\ & + q?? \,) \end{aligned}$$

Send message - line 13 Finally, after the `catch`, the system will respond to the caller.

$$\begin{aligned} s? \,.\, \tau \,.\, q! \,.\, (\, & q? \,.\, s! \\ & + q??\ s! \,) \end{aligned}$$

No response (e.g. Timeout) - line 10 The timeout is handled by the `catch(Exception)` in Java. This allows to modelise the unexpected error.

$$\begin{aligned} s? \,.\, \tau \,.\, q! \,.\, (\, & q? \,.\, s! \\ & + q??\ s! \\ & + \tau \,) \end{aligned}$$

Final step - Send message - line 13 Just like other, for this component, in all case the end is to respond to the caller.

$$\begin{aligned} s? \,.\, \tau \,.\, q! \,.\, (\, & q? \,.\, s! \\ & + q?? \,.\, s! \\ & + \tau \,.\, s! \,) \end{aligned}$$

Combine and simplify Once all the models have been created, they need to be put together and made into a composition. Sometimes they also need to be simplified. It is possible to do this with the + alternatives.

Here, we need to combine these two models :

$$s? \,.\, \tau \,.\, s!$$

$$\begin{aligned} s? \,.\, \tau \,.\, q! \,.\, (\, & q? \,.\, s! \\ & + q?? \,.\, s! \\ & + \tau \,.\, s! \,) \end{aligned}$$

This will lead to this final model :

$$\begin{aligned} s? \,.\, (\, & \tau \,.\, s! \\ & + \ \tau \,.\, q! \,.\, (\, q? \,.\, s! \\ & \qquad\qquad + \ q?? \,.\, s! \\ & \qquad\qquad + \ \tau \,.\, s! \,) \\ &) \end{aligned}$$

4.3.4 Failure-free scenario

As explained before in the point 4.2.1, this "failure-free scenario" assumes that the component always responds and follows the expected behaviour with no adverse effects. The component cannot "crash" but can respond with an error message. This scenario allows ensures the correct expected behaviour of the component.

4.3.4.1 Services modelling

First of all, before connecting components together, it is necessary to define them individually to see their behaviour. It is sufficient to apply the method explained in the previous point to obtain the representation of the component.

The queue component

Just below, there is the queue component representation with the error management directly in the model. We can obtain this model by applying the process explain just before.

$$
\begin{aligned}
q? \,.\, (\ \tau \,.\, q! \qquad & (4.1)\\
+\, \tau \,.\, q!!\ ) \qquad & (4.2)
\end{aligned}
$$

We can see the behaviour of the component :

- 4.1 : The queue responds with OK
- 4.2 : The queue responds with an error (e.g. an exception)

The shipping component

The shipping component is a component which interacts with the queue but for now, there are no interactions with the queue. Only the internal behaviour is shown below.

$$
\begin{aligned}
& s? \,.\, (\ \tau \,.\, s! && (4.3)\\
& \quad +\ \tau \,.\, q! \,.\, (\ q? \,.\, s! \qquad // A && (4.4)\\
& \qquad\qquad +\ q?? \,.\, s!\ )\ // B &&\\
& \qquad\qquad +\ \tau \,.\, s!\ ) \quad // C &&\\
& \quad) && (4.5)
\end{aligned}
$$

The behaviour is the following one :

- First, in 4.3, the queue can respond without processing the message
- Or, in 4.4, the queue processes the message, contacts the queue and after that, there are multiples solutions :
  - **A** : The queue responds with OK
  - **B** : The queue responds with KO
  - **C** : There is no response from the queue or other Exception

Here we can already see that whatever happens, the component will never return a message. Of course, this behaviour may not be completely realistic, and should be taken into account in future reflections.

4.3.4.2 Services composition

The composition of 2 services equals $S_1 \mid S_2$. For this first example, the composition is between services `queue` and `shipping`.

$$
\begin{aligned}
\mathit{queue} \mid \mathit{shipping} \;\equiv\;\; & s? \,.\, (\ \tau \,.\, s! +\ \tau \,.\, q! \,.\, (\ q? \,.\, s! +\ q?? \,.\, s! + \tau \,.\, s!\ )\ )\\
& \mid q? \,.\, (\ \tau \,.\, q! + \tau \,.\, q!!\ )
\end{aligned}
$$

Note : For an easier reduction and visibility of the expressions, they have been put on a single line (but they are still the same as presented in the previous point).

4.3.4.3 "Termination" of a service

Before we verify if 2 services can compose together, it is useful to check if a service can "terminate". The termination is represented by the fact that the process reaches the end of its behaviour without any problems or error and thus becomes an inactive process represented by `0`. The process has followed the expected execution path.

The queue component

The initial state of the queue is :

$$\begin{aligned} queue \;\equiv\; & q? \,.\, (\, \tau \,.\, q! \\ & \qquad + \tau \,.\, q!! \,) \end{aligned}$$

After one step :

$$\begin{aligned} queue \;\equiv\; & (\, \tau \,.\, q! \\ & \; + \tau \,.\, q!! \,) \end{aligned}$$

There is an alternative. Two paths are possible.

Scenario 1 : The first option is chosen and the next step state is :

$$queue \;\equiv\; \tau \,.\, q!$$

The internal behaviour (τ) is processed and the only thing remaining is the `q!`.

$$queue \;\equiv\; q!$$

Scenario 2 : The other alternative is chosen and the next state is :

$$queue \;\equiv\; \tau \,.\, q!!$$

Just like Scenario 1, the internal behaviour (τ) is processed and the only thing remaining is the `q!!`.

$$queue \;\equiv\; q!!$$

In both cases, the last step is completed, and the process comes to an end.

$$queue \;\equiv\; 0$$

As a conclusion, with these simple derivations, it was possible to prove that the process ended well (terminate) in both cases.

The shipping component

The initial state of the shipping is :

$$\begin{aligned} & s? \,.\, (\, \tau \,.\, s! \\ & \qquad + \; \tau \,.\, q! \,.\, (\, q? \,.\, s! \\ & \qquad\qquad\qquad + \; q?? \,.\, s! \\ & \qquad\qquad\qquad + \; \tau \,.\, s! \,) \\ & \qquad) \end{aligned}$$

After 1 transition, the state is as follows :

$$\begin{aligned} & \tau \,.\, s! \\ + \; & \tau \,.\, q! \,.\, (\, q? \,.\, s! \\ & \qquad + \; q?? \,.\, s! \\ & \qquad + \; \tau \,.\, s! \,) \end{aligned}$$

The `s?` has been done, and the service has a choice. The current alternative is in red.

Scenario 1 : The first alternative is chosen.

$$shipping \equiv \; \tau \,.\, s!$$

The internal process τ is performed as well as the `s!` for the response. The state then changes to `0` to indicate that it is finished. This case works well, the system reaches an end.

$$shipping \equiv 0$$

Scenario 2 : The second alternative is chosen.

$$\begin{aligned} shipping \equiv \; & \tau \,.\, q! \,.\, (\, q? \,.\, s! \\ & \qquad + q?? \,.\, s! \\ & \qquad + \tau \,.\, s! \,) \end{aligned}$$

The internal process τ is realised as well as the `q!`. There is now a new state and the process waits another trigger to continue.

$$\begin{aligned} shipping \equiv \; & q? \,.\, s! \\ & + q?? \,.\, s! \\ & + \tau \,.\, s! \,) \end{aligned}$$

Depending on the trigger, the right alternative will be chosen, and the process will reach an end.

$$shipping \equiv 0$$

In both cases, the process reaches an end and terminates.

Queue and shipping composition

The initial state is the following one, taken from the point just before :

$$\begin{aligned} queue \mid shipping \;\; \equiv \;\; & s? \,.\, (\, \tau \,.\, s! + \; \tau \,.\, q! \,.\, (\, q? \,.\, s! + \; q?? \,.\, s! + \tau \,.\, s! \,)\,) \\ & \mid q? \,.\, (\, \tau \,.\, q! + \tau \,.\, q!! \,) \end{aligned}$$

The process needs an external trigger to start. In our case, we need to be able to activate the shipping, which in turn communicates with the queue. For a clearer comprehension, it is useful to define the different possible scenarios :

1. The shipping component chooses the first alternative and responds directly without communicating with the queue. This is the blue part of the model.
2. The shipping communicates with the queue and receives an OK message (`q!`). This is the red part of the model.

3. The shipping communicates with the queue and receives an error message (`q!!`). This is the green part of the model.
4. The shipping communicates with the queue but never receives a response. This is the yellow part of the model.

Scenario 1 : As this path only involves one component and has been proven before, it will not be discussed further.

Scenario 2 : Just after the `s?`, the process enters the second alternative. The state is the following one :

$$\begin{aligned} queue \mid shipping \ \equiv\ & q! \,.\, (\, q? \,.\, s! + \ q?? \,.\, s! \ + \tau \,.\, s! \,) \\ & \mid q? \,.\, (\, \tau \,.\, q! + \tau \,.\, q!! \,) \end{aligned}$$

The shipping component sends a message to the queue and this is the resulting state :

$$\begin{aligned} queue \mid shipping \ \equiv\ & q? \,.\, s! + q?? \,.\, s! + \tau \,.\, s! \\ & \mid \tau \,.\, q! + \tau \,.\, q!! \end{aligned}$$

The queue fits into the first alternative :

$$\begin{aligned} queue \mid shipping \ \equiv\ & q? \,.\, s! + q?? \,.\, s! + \tau \,.\, s! \\ & \mid \tau \,.\, q! \end{aligned}$$

The queue sends the response to the shipping component and terminates.

$$\begin{aligned} queue \mid shipping \ \equiv\ & s! \\ & \mid 0 \end{aligned}$$

There is only one action left for shipping to finish as well. A final state is reached with both components who have terminated.

$$\begin{aligned} queue \mid shipping \ \equiv\ & 0 \\ & \mid 0 \end{aligned}$$

Scenario 3 : As the beginning is the same as the scenario 2, the modelling is done from the point where it changes, when the state of the queue is different. The queue fits into the second alternative (reply with an error) :

$$\begin{aligned} queue \mid shipping \ \equiv\ & q? \,.\, s! + q?? \,.\, s! + \tau \,.\, s! \\ & \mid \tau \,.\, q!! \end{aligned}$$

The queue sends the response to the shipping component and terminates.

$$\begin{aligned} queue \mid shipping \ \equiv\ & s! \\ & \mid 0 \end{aligned}$$

Just like the scenario 2, there is only one step remaining to reach an end. A final state is reached with both components which have terminated.

$$\begin{aligned} queue \mid shipping \ \equiv\ & 0 \\ & \mid 0 \end{aligned}$$

Scenario 4 : In this "failure-free scenario", it is assumed that the process will never crash. This case should never happen.

These scenarios prove that, in this "failure-free scenario", the `queue` and the `shipping` component fit and can interact together.

4.3.5 Failure-aware scenario

This "failure-aware scenario" is the same as the previous scenario, with one major difference : the system can "crash" at any time. As explained in subsection 4.2.2, a crash can be a service which stops working, a network problem and therefore the communication does not reach its destination, etc.

For all services, the δ represents a `deadlock`, that means the component is in a blocked state and does not respond to the expected behaviour. e.g. the process is stuck.

4.3.5.1 Services modelling

This new rule slightly affects the components. It is therefore necessary to redefine them. To have a better understanding, it is easier to specify places where the component can crash by adding these δ states in the model.

The queue component

For this model, as it is the first, it is useful to specify where the process can fail. This is the model of the component :

$$
\begin{aligned}
& q? \,.\, (\, \tau \,.\, q! && (4.6)\\
& \quad + \tau \,.\, q!! && (4.7)\\
& \quad + \tau \,.\, \delta \,) && (4.8)\\
& + \delta && (4.9)
\end{aligned}
$$

It is possible to extract 2 information :

- 4.6 and 4.7 are the same as the failure-free scenario (4.3.4).
- 4.8 and 4.8 show explicitly where the process can fail.

The shipping component

As the queue before, the shipping needs to be corrected by adding places where the system can fail. This leads to a new model :

$$
\begin{aligned}
& s? \,.\, (\, \tau \,.\, (\, s! \\
& \qquad + \; \tau \,.\, q! \,.\, (\, q? \,.\, (\, s! + \tau \,.\, \delta \,) \\
& \qquad\qquad + \; q?? \,.\, (\, s! + \tau \,.\, \delta \,) \\
& \qquad\qquad + \; \tau \,.\, (\, s! + \tau \,.\, \delta \,) \\
& \qquad\qquad + \; \tau \,.\, \delta \,) \\
& \qquad + \tau \,.\, \delta \,) \\
& \quad + \; \tau \,.\, \delta \,)
\end{aligned}
$$

Previously, the third option, the one which handles the no reply of the queue component, was never used. This part was not relevant to the failure-free scenario, as it was assumed that the queue would never be in trouble.

4.3.5.2 Services composition

The services' composition is the same as the failure-free scenario (4.3.4.2) : `queue | shipping`. The failure behaviour is only described for the queue component, but it is implicitly present for the shipping too.

$$\begin{aligned} queue \mid shipping \;\equiv\;\; & s? \,.\, (\, \tau \,.\, s! + \tau \,.\, q! \,.\, (\, q? \,.\, s! + q?? \,.\, s! \,+\, \tau \,.\, s! \,)\,) \\ & \mid q? \,.\, (\, \tau \,.\, q! + \tau \,.\, q!! + \tau \,.\, \delta \,) + \delta \end{aligned}$$

4.3.5.3 Termination of services

The behaviour of the components is the same as before, so there is no need to check that they finish well (except for the shipping component). Here, we want to analyse the behaviour of the other components when a component fails.

The shipping component

As there is a new alternative, it is necessary to check if the process can finish in this case. As the beginning is the same as the scenario failure-free scenario, the modelling is done from the point where it changes.

$$\begin{aligned} shipping \equiv\; & q? \,.\, s! \\ & + \, q?? \,.\, s! \\ & + \, \tau \,.\, s! \end{aligned}$$

The third alternative is chosen.

$$shipping \equiv \tau \,.\, s!$$

And finally, the process reaches a final state.

$$shipping \equiv 0$$

Queue and shipping composition

The initial state is the following one, taken from the point 4.3.5.2 :

$$\begin{aligned} queue \mid shipping \;\equiv\;\; & s? \,.\, (\, \tau \,.\, s! + \tau \,.\, q! \,.\, (\, q? \,.\, s! + q?? \,.\, s! \,+\, \tau \,.\, s! \,)\,) \\ & \mid q? \,.\, (\, \tau \,.\, q! + \tau \,.\, q!! \,) \end{aligned}$$

The point of failure of the queue is not important here because in all cases the shipping processes the information in the same way.

First, after the trigger of the shipping `s?` and the call of the queue `q!`, the state is the following one :

$$\begin{aligned} queue \mid shipping \;\equiv\; & q? \,.\, s! + q?? \,.\, s! \,+\, \tau \,.\, s! \\ & \mid q? \,.\, (\, \tau \,.\, q! + \tau \,.\, q!! \,) \end{aligned}$$

This is the state after the queue has failed :

$$\begin{aligned} queue \mid shipping \;\equiv\; & q? \,.\, s! + q?? \,.\, s! \,+\, \tau \,.\, s! \\ & \mid \delta \end{aligned}$$

The shipping will never have a response and so enters the third alternative which is the management of the error.

$$\begin{aligned} queue \mid shipping \; &\equiv \tau \;.\; s! \\ &\mid \delta \end{aligned}$$

The last step is to process the last actions and terminate.

$$\begin{aligned} queue \mid shipping \; &\equiv 0 \\ &\mid \delta \end{aligned}$$

The failure did not propagate further because the shipping component handled the error.

4.3.6 Replicas

Services can also be replicated. There are the same, but in multiple instances. This changes the interactions between the components. Since the application architecture is modified, it is necessary to check again if the properties given earlier are still correct or not.

The replication of the queue is equivalent to have $Q_1 \mid Q_2$, with Q_1 and Q_2 both queue components. A composition with the shipping could be :

$$shipping \mid queue \mid queue$$

With the model :

$$\begin{aligned} shipping \mid queue \mid queue \; \equiv \; & s? \;.\; (\; \tau \;.\; s! + \; \tau \;.\; q! \;.\; (\; q? \;.\; s! + \; q?? \;.\; s! \; + \; \tau \;.\; s! \;)\;) \\ & \mid q? \;.\; (\; \tau \;.\; q! + \tau \;.\; q!! \;) \\ & \mid q? \;.\; (\; \tau \;.\; q! + \tau \;.\; q!! \;) \end{aligned}$$

Multiple cases are possible with these replicas :

1. The shipping communicates with a component and the exchange is good
2. The queue fails and the system stops like before
3. The replicated queue fails and there is no impact

Scenario 1 : It is the same behaviour as before, the process will reach an end. The final state will be :

$$\begin{aligned} queue \mid shipping \; &\equiv 0 \\ &\mid 0 \\ &\mid q? \;.\; (\; \tau \;.\; q! + \tau \;.\; q!! \;) \end{aligned}$$

Scenario 2 : The shipping started to communicate with the first queue :

$$\begin{aligned} queue \mid shipping \; &\equiv q? \;.\; s! + \; q?? \;.\; s! \; + \; \tau \;.\; s! \\ &\mid \tau \;.\; q! + \tau \;.\; q!! \\ &\mid q? \;.\; (\; \tau \;.\; q! + \tau \;.\; q!! \;) \end{aligned}$$

The state when the queue fails :

$$\begin{aligned} queue \mid shipping \; &\equiv q? \;.\; s! + \; q?? \;.\; s! \; + \; \tau \;.\; s! \\ &\mid \delta \\ &\mid q? \;.\; (\; \tau \;.\; q! + \tau \;.\; q!! \;) \end{aligned}$$

The τ transition will be triggered, as the point before.

$$\begin{aligned} queue \mid shipping \ &\equiv \tau \ . \ s! \\ &\mid \delta \\ &\mid q? \ . \ (\ \tau \ . \ q! + \tau \ . \ q!! \ ) \end{aligned}$$

And the process will reach an end.

$$\begin{aligned} queue \mid shipping \ &\equiv 0 \\ &\mid \delta \\ &\mid q? \ . \ (\ \tau \ . \ q! + \tau \ . \ q!! \ ) \end{aligned}$$

Scenario 3 : The behaviour is similar to before, but the replicated process fails :

$$\begin{aligned} shipping \mid queue \mid queue \ &\equiv q? \ . \ s! + \ q?? \ . \ s! \ + \ \tau \ . \ s! \\ &\mid q? \ . \ (\ \tau \ . \ q! + \tau \ . \ q!! \ ) \\ &\mid \delta \end{aligned}$$

This as no impact on communicating components. The final state will be :

$$\begin{aligned} shipping \mid queue \mid queue \ &\equiv 0 \\ &\mid 0 \\ &\mid \delta \end{aligned}$$

4.4 Conclusion

In this chapter, we were able to get an overview of what the sock shop is by analysing mainly two of its components : order and shipping. Then, the analysis of these components continued with the modelling of these components using the π-calculus based language. We ensure that it is possible to describe microservices by using π-calculus like process algebra through a certain method. Manual analysis could be performed. This process was done through two closely related scenarios. The first one aimed at modelling the components by writing the behaviour they were supposed to have (taking into account communications, possible error handling, ...) and the second one introduced the deadlocks. The component can crash at any time. This was to determine one thing : does the process work and terminate correctly ? Once checked, how does the integration of two processes together work ? It is possible to check all this manually but, obviously, this will not scale to a big system and other tools are needed to verify bigger systems and other properties. That is precisely what the next chapter will do by introducing Petri nets.

Chapter 5

Reasoning on microservices with Petri Nets

The specification, given with this π-calculus and CCS based language, allows checking if the process terminates. But it is only one property, and it is a little limited. As explained before, Petri nets offer a lot of possibilities and allow checking more properties. A solution is therefore to transform and adapt this model to use Petri nets.

5.1 Translate process calculus to PNML

As we need to transform the model based on π-calculus into Petri nets, the simplest way is to use PNML. In the article [58], the authors propose a method to translate CSP specifications to equivalent Petri nets where they generate Petri nets from executions.

In [59], the authors proposed another method to model and analyse by using a Petri nets based approach. The authors show that the behaviour is preserved by commenting on the fact that the modelling was preserved "by construction".

This part therefore adapts these methods by proposing an inspired algorithm. Then a proof will come to ensure the equivalence of the two models, and the process will be carried out on a component to illustrate how it works.

5.1.1 An algorithm to generate Petri nets from π-calculus based language

This process can be laborious. It is therefore useful to create an algorithm to follow. These are the different steps :

1. Create the structure of the PNLM document. As explained before, some fields are mandatory, e.g. <pnml> or <net>.

2. Generate the initial place. The net has to be initialised with an initial place with a token. It's the entry of the net.

3. For each element[1] of the language, a different action is required (depending on the type of element) :

   (a) **Place** : A place is symbolised by a point (`.`) in the language. It's a state between two transitions.

   (b) **Transition** : A transition is represented by any action. E.g, the (`x?`) or the (τ) will be translated into a transition.

   (c) **Arc** : Once a place and a transition have been generated, they must be connected by an arc. This arc requires the source and the target destinations.

[1]An element is defined by the syntax. E.g. the `x?` is an element who corresponds to a communication, τ represents the internal behaviour, etc.

4. If there is an alternative, the last place before the alternative must be retained in order to make the link between the place and the transition. The process is the same as explained above for the different parts of the alternative, only the initial place changes. This process can be repeated as long as there are alternatives. This process can also be embedded.

5. If there is a parallel composition, it is necessary to generate a new net which will follows the same rules as explained previously. These two nets will evolve in parallel.

6. Finally, the network is closed by adding a final state after the last transition.

In PNML, each place, transition and arc need an id. These are automatically generated during the process.

This process only generates the PNML syntax without all the graphic part. It will be necessary to rework the model in order to display it in an editor such as WoPeD.

Nevertheless, this process can be fastidious and very long. Automating it is therefore a good idea. So that is what I did, an implementation can be found here [60].

5.1.2 Implementation of this process

This implementation works in multiple steps :

1. Transposes the π-calculus based model to simple PNML

2. If necessary, reviews this PNML to compose nets together

These steps are described below.

Step 1 : This corresponds to the algorithm described just above in the subsection 5.1.1. This is a python script that will read step by step the file that contains a description made with this π-calculus based language. In an iterative way, the PNLM file will be built. At each element, a correspondence is made between the elements, as explained earlier.

This process will work and be sufficient in the following cases :

- A simple description without concurrency or alternative. e.g. `q? . ` τ ` . q!`

- A description with an alternative. e.g. `q? . (` τ ` . q! +` τ ` . q!!` This alternative can be embedded.

In case of a parallel composition, the script will produce two separates nets. If we want to combine them, we will have to use a second script, which is explained below.

Step 2 : This corresponds to another part of the analysis. Once the nets have been studied and analysed separately, it is necessary to be able to combine them to see how they interact together. This works as follows :

1. Walk through the Petri nets to see the inputs and outputs

2. Links the corresponding inputs and outputs. e.g. `q!` and `q?`

3. Changes the links between places and transitions so that the final result is a single Petri net

It is important to note that to easily obtain inputs and outputs, it is useful to use Open Petri nets.

5.1.3 How can the behaviour of the model be preserved ?

Just like in [59] where the authors proved that the semantic is preserved "by construction", I also use this method to create the PNML file.

The method presented just above transforms each element of the CCS based language to en element of the Petri net. Each state is mapped into a place and each possible action / transition is mapped into a transition. Also, all possible paths are represented by the arcs.

The initial state of the net is the same as the initial state of the model. The net is waiting for a transition, just as the model expects a trigger as input.

As all elements have a correspondence "by construction", we can say that there is an equivalence between the two models.

Another solution could be to check if all derivations can be reached ($L_{\sum}(N) = \{\varepsilon, ...\}$).

5.1.4 Example

This part shows all the steps for the transformation of the `queue` component. This corresponds to the first step of the transformation explained previously. Here, the process is shown step by step to understand how the script works.

At the beginning, we have this model to transform :

$$\begin{aligned} q? \,.\, (\, \tau \,.\, q! \\ + \tau \,.\, q!! \,) \end{aligned}$$

The first step is to create the net and generate the initial place :

```
<?xml version="1.0" encoding="UTF-8">
<pnml>
    <net id="queue-net">
        <place id="p1">
            <name>
                <text>p1</text>
            </name>
            <initialMarking>
                <text>1</text>
            </initialMarking>
        </place>
    </net>
</pnml>
```

Then, each transition is generated with elements related to, which produces this PNML specification after the first transition (the element `q?`) :

```
<?xml version="1.0" encoding="UTF-8">
<pnml>
    <net id="queue-net">
        <place id="p1">
            <name>
                <text>p1</text>
            </name>
            <initialMarking>
                <text>1</text>
            </initialMarking>
        </place>
        <place id="p2">
            <name>
                <text>p2</text>
            </name>
        </place>
        <transition id="t1">
```

```
                <name>
                    <text>q?</text>
                </name>
            </transition>
            <arc id="a1" source="p1" target="t1"> </arc>
            <arc id="a2" source="t1" target="p2"> </arc>
        </net>
</pnml>
```

This process will have to be repeated to model all possible transitions and have an equivalent model. The result of all this transformation can be found in the appendix 7.2.

As explained in the point before, this notation does not take into account the graphical positioning related to WoPeD (or other tools).

As this step is automated, it is easy to apply it to other processes, such as the `shipping` component. The result of this transformation can be found in the appendix 7.3.

5.2 Petri nets based model

Just like the analyses made earlier, these models based on Petri nets use the same components as those based on π-calculus and CCS (explained in section 4.1 above). They can be obtained by using the transformation from this language to PNML with the method explained just before.

They offer other possibilities, like a more graphical and, maybe, more easily understandable view or the possibility to check other properties, view the reachability graph, etc.

5.2.1 Failure-free scenario

As in section 4.3.4, this "failure-free scenario" assumes that the component always responds, whether they are OK or KO, but they cannot crash.

5.2.1.1 Services modelling

There are multiple ways to model these components with Petri nets or subtypes of Petri nets. Here, there are multiple diagrams with their own characteristics. A description occurs them to know these characteristics.

These representations with Petri nets (and subtypes) are sometimes more similar than the representations exposed before, as they use the same formalism about communication.

These nets can be obtained with the first tool exposed before. It will automatically generate the PNML which corresponds to the π-calculus model.

The queue component

The first representation, Figure 5.1 represents the component as simply as possible. After an internal process, the component can return an OK (`q!`) or KO (`q!!`) message. The `queue entry point` place corresponds to the state after the `q?`. This model corresponds to :

$$\boxed{q? \; . \; \tau \; . \; (\; q?? + q? \;)}$$

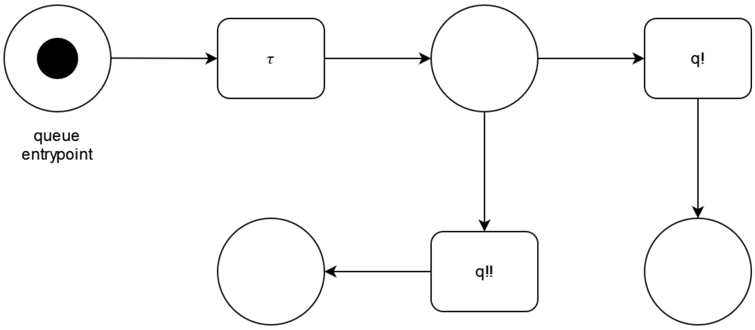


Figure 5.1: Queue component - Petri net

Here, we assume that the choice of which response will be sent is made after the internal process τ. In the Figure 5.2, the choice is made before. This representation can be obtained with this model :

$$q? \,.\, (\, \tau \,.\, q! + \tau \,.\, q!! \,)$$

Here, the representation is a little different as it uses open Petri nets.

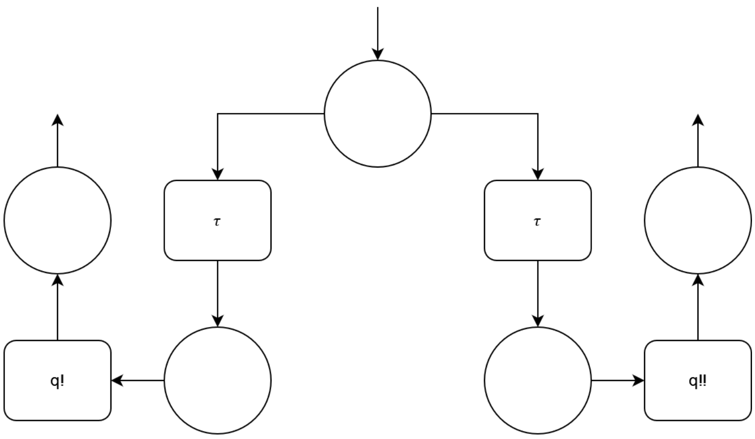


Figure 5.2: Queue component - Open Petri net

The main advantage of open nets is that it is very easy to see inputs and outputs directly. This step is not automated with the script, but it is simply adding arrows to these endpoints.

The shipping component

As the queue component above, the first representation, Figure 5.3, represents the component and its expected behaviour with multiple scenarios possibles (as explained in the section 4.3.4). Here, the central state (after the `q!`) acts like if it is the queue component. Just like the queue component, this net can be obtained by passing the following model through the translation tool.

$$s? \,.\, (\, s! + q! \,.\, (\, q? \,.\, s! + \; q?? \,.\, s! \; + \; \tau \,.\, s! \,)\,)$$

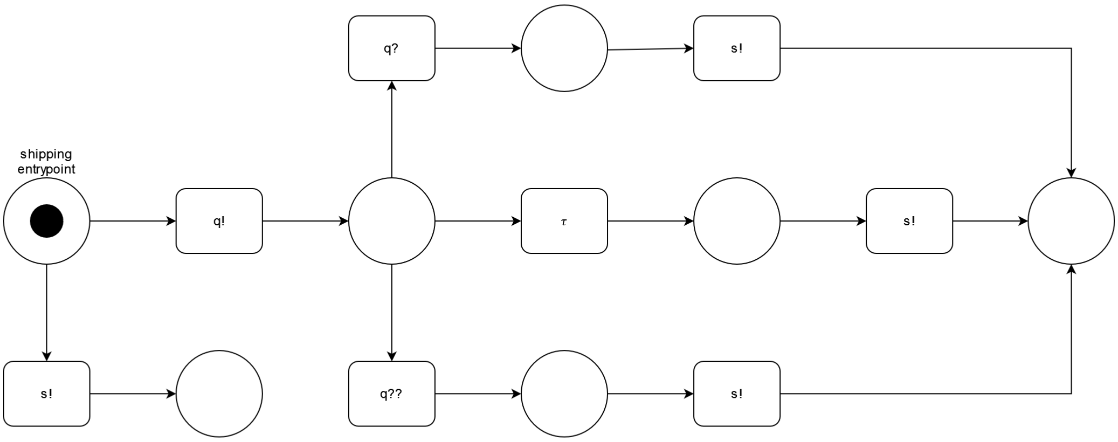


Figure 5.3: Shipping component - Petri net

Once more, just like the queue component, Figure 5.4 represents the component using Open Petri nets.

The dotted shape is not a real part of the model, but simulates the place of the queue. This representation is not "correct" as this part does not exist, but it is there to help to understand the following step : the composition. With this open Petri nets, we can clearly see where the other component will have to integrate.

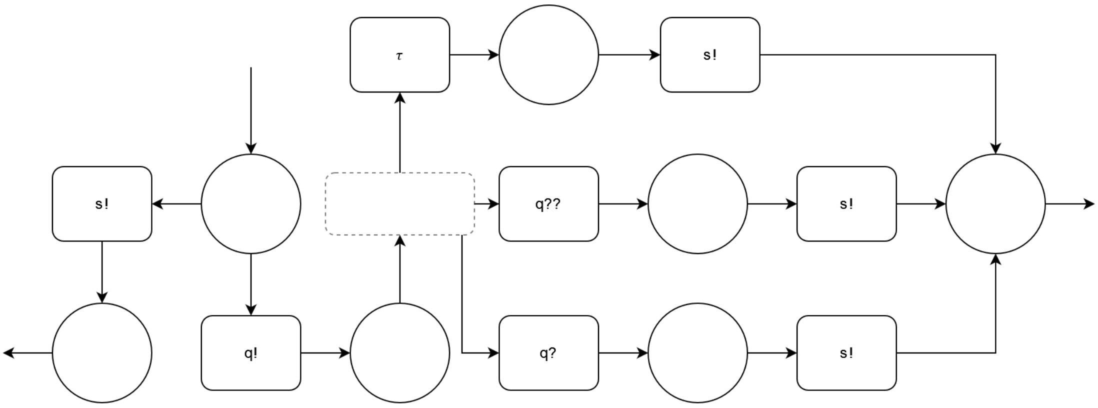


Figure 5.4: Shipping component - Open Petri net

We can clearly distinguish multiple inputs and outputs possible :

- **Inputs** :
  - On the left side, the entry of the component
  - After the dotted component, there are three inputs possibles (τ, `q?` and `q??`)
- **Outputs** :
  - On the left side, after the `s!`
  - Where there is the dotted component that refers to calling another component
  - On the right side, after the `s!`

5.2.1.2 Services composition

Once we have model each component, the next step is to compose them together. This is the goal of the second tool : be able to generate this new net from separate nets. The following Figure illustrates this composition of the queue and the shipping.

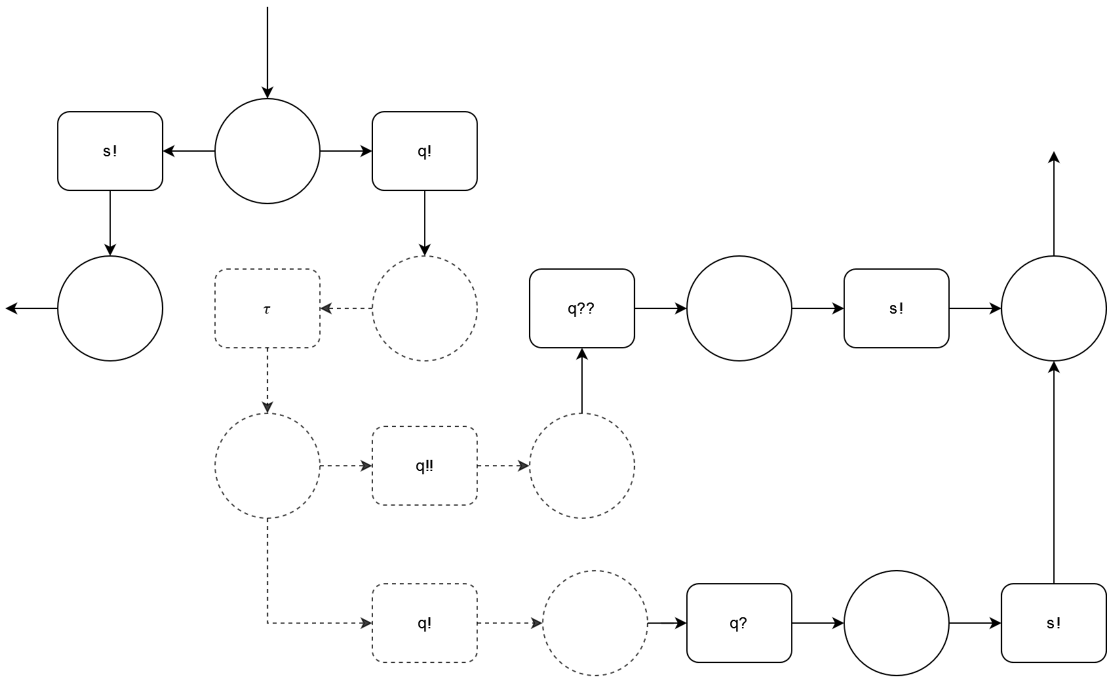


Figure 5.5: Composition of the queue and the shipping components - Open Petri net

We can distinguish two types of element : solid lines ones and doted ones. The doted component represents the queue, and the solid line component is the shipping.

For a better comprehension, there are two output places, but they are both after a transition `s!`. It is the same state (final state $[o]$).

5.2.1.3 Verify if nets are sound

An objective of Petri nets is to have the possibility to verify more properties. The representations above allow us to perform some analysis. The first objective is to check if a net is sound. To do that, one needs to check its three properties :

1. The net has the option to complete
2. The net has a proper completion
3. The net does not have dead transitions

First, before checking whether the composition is sound, the nets should be tested separately and after that, put them together.

For now and to have a good comprehension of what is done, these verifications are done manually, but they can be automated with tools. We will see it later.

The queue component

The following parts will show different properties to ensure that the net is sound. The net analysed is the one in Figure 5.1.

Option to complete There are two possible paths to terminate :

- $\tau \rightarrow$ `q!`
- $\tau \rightarrow$ `q!!`

$\Rightarrow$ The final state $[o]$ is reachable from the initial state $[i]$.

Proper completion Using the same transitions as above, no token remains in a state prior to the final state $[o]$.

No dead transitions All transitions are reachable and can be fired.

The shipping component

The following parts will show different properties to ensure the net is sound. The net analysed is the one in Figure 5.3.

Option to complete There are four possible paths to terminate :

- `s!`
- `q!` $\rightarrow$ `q?` $\rightarrow$ `s!`
- `q!` $\rightarrow \tau \rightarrow$ `s!`
- `q!` $\rightarrow$ `q??` $\rightarrow$ `s!`

$\Rightarrow$ The final state $[o]$ is reachable from the initial state $[i]$.

Proper completion Using the same transitions as above, no token remains in a state prior to the final state $[o]$.

No dead transitions All transitions are reachable and can be fired.

The queue and shipping composition

The following parts will show different properties to ensure the net is sound. The net analysed is the one in Figure 5.5.

Option to complete There are three possible paths to terminate :

- `s!`
- `q!` $\rightarrow \tau \rightarrow$ `q!` $\rightarrow$ `q?` $\rightarrow$ `s!`
- `q!` $\rightarrow \tau \rightarrow$ `q!!` $\rightarrow$ `q??` $\rightarrow$ `s!`

The transition with τ after the queue component has been removed, as it is a failure behaviour. This transition is present in the next section, failure-aware scenario 5.2.2.

$\Rightarrow$ The final state $[o]$ is reachable from the initial state $[i]$.

Proper completion Using the same transitions as above, no token remains in a state prior to the final state $[o]$.

No dead transitions All transitions are reachable and can be fired.

5.2.2 Failure-aware scenario

This "failure-aware scenario" is the same as the previous scenario, but here, the process can fail at any time. This is symbolised by this transition :

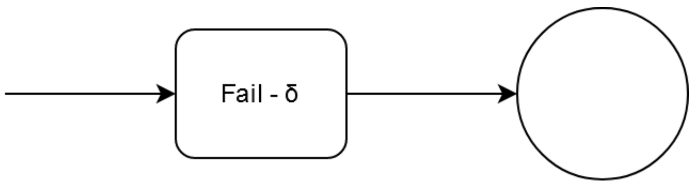


Figure 5.6: Failure transition

When this transition is fired, the state after is final and means the process is stuck. This transition can appear at any state.

5.2.2.1 Services modelling

The queue component

This net has the same component than the one presented before in Figure 5.1 but, for every state, a new possible transition which indicates that the process can stop at any time.

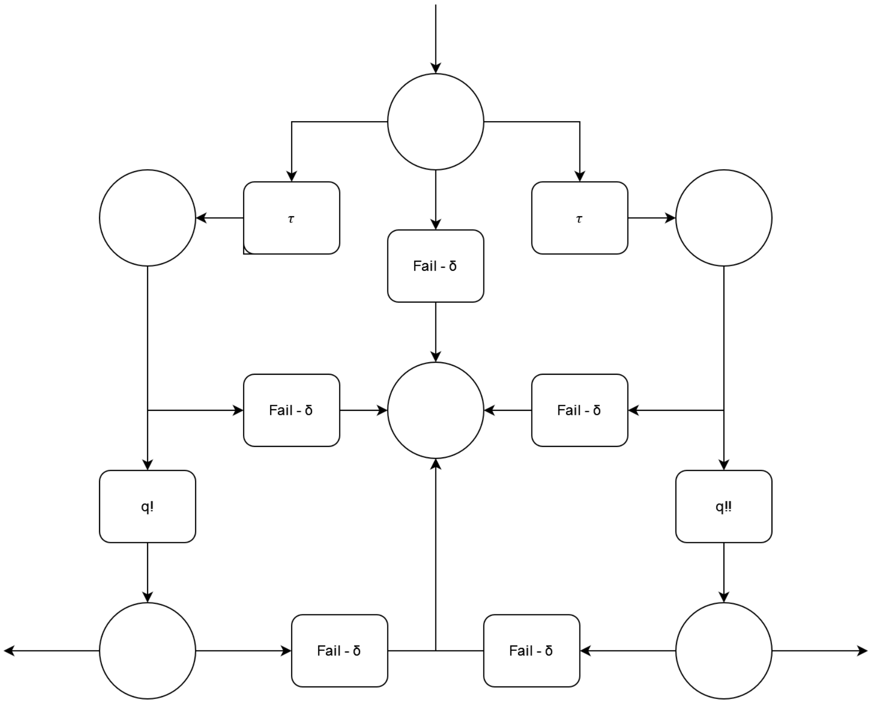


Figure 5.7: Queue component with failures - Open Petri net

The shipping component

The following Figure is the same as Figure 5.3 with the addition of failure transitions. This fail / δ state is represented by the red states.

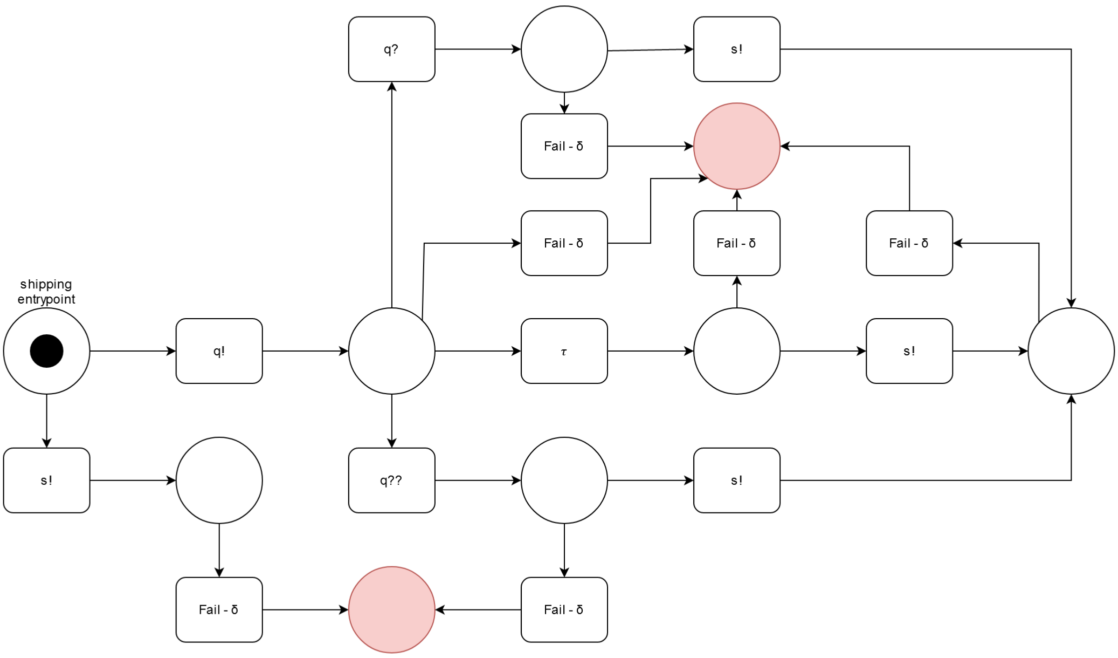


Figure 5.8: Shipping component with failures - Petri net

5.2.2.2 Services composition

This net is a composition using an Open Petri net. Red states represent the fail / δ. The doted component is the queue and the inline one is the shipping.

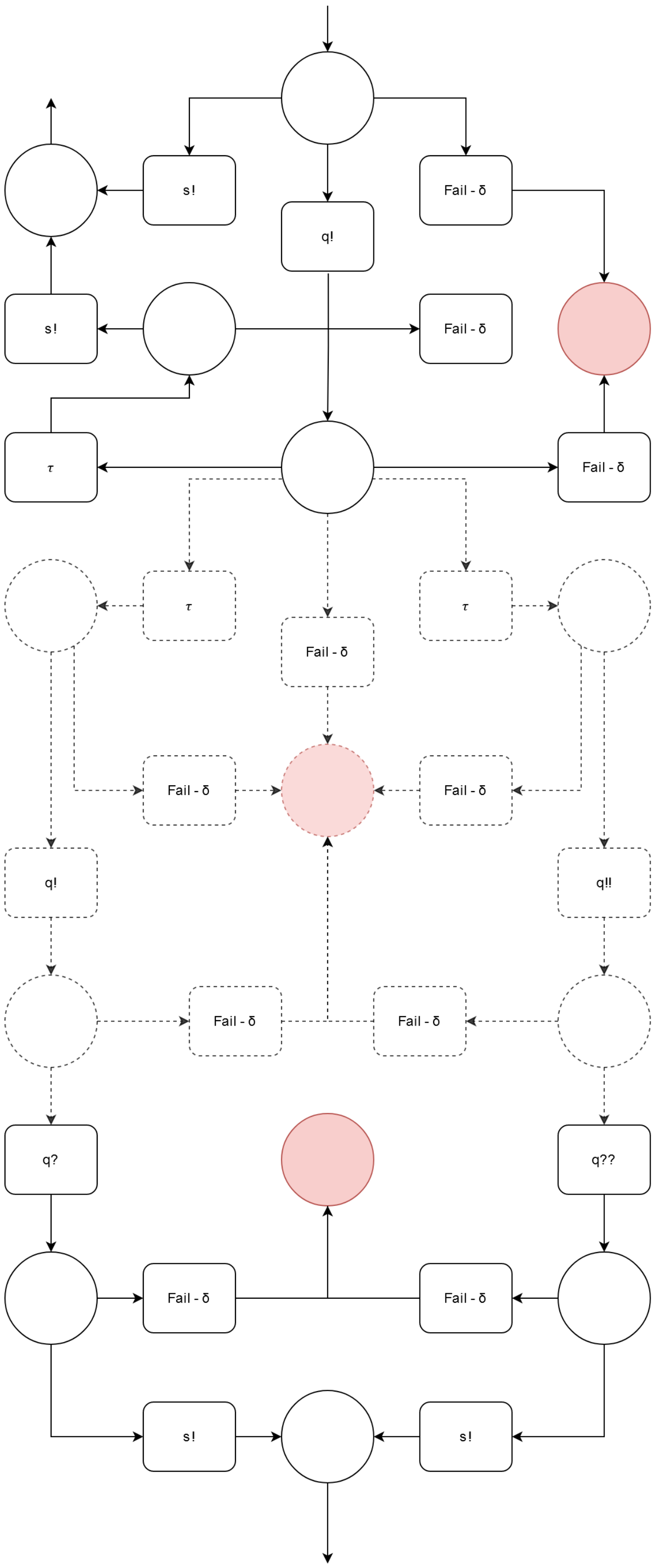


Figure 5.9: Queue and shipping composition with failures - Open Petri net

5.2.2.3 Verify if nets are sound

Once again, the three properties must be checked to ensure the soundness of the net.

The queue component

The analysis is done on Figure 5.7.

Option to complete There are only two possible paths that lead to a correct end :

- $\tau \rightarrow$ `q!`
- $\tau \rightarrow$ `q!!`

The other transitions always trigger to a fail state :

- δ
- $\tau \rightarrow \delta$
- $\tau \rightarrow$ `q!` $\rightarrow \delta$
- $\tau \rightarrow$ `q!!` $\rightarrow \delta$

$\Rightarrow$ Even if all paths don't succeed, the net has two possibilities to reach an end.

Proper completion Using the same transitions as above, no token remains in a state prior to the final state $[o]$.

No dead transitions At the beginning, all transitions are reachable and can be fired.
$\Rightarrow$ The net still sound but can fail at any time. The **liveness** is not guaranteed.

The shipping component

The analysis is done on Figure 5.8.

Option to complete There are four possible paths to terminate with an attended end, there are the same as the point before :

- `s!`
- `q!` $\rightarrow$ q? $\rightarrow$ `s!`
- `q!` $\rightarrow \tau \rightarrow$ `s!`
- `q!` $\rightarrow$ q?? $\rightarrow$ `s!`

The other transitions always trigger to a fail state :

- `s!` $\rightarrow \delta$
- `q!` $\rightarrow \delta$
- `q!` $\rightarrow$ q? $\rightarrow \delta$
- `q!` $\rightarrow$ q? $\rightarrow$ `s!` $\rightarrow \delta$
- `q!` $\rightarrow \tau \rightarrow \delta$
- `q!` $\rightarrow \tau \rightarrow$ `s!` $\rightarrow \delta$
- `q!` $\rightarrow$ q?? $\rightarrow \delta$
- `q!` $\rightarrow$ q?? $\rightarrow$ `s!` $\rightarrow \delta$

$\Rightarrow$ The final state $[o]$ is reachable from the initial state $[i]$.

Proper completion Using the same transitions as above, no token remains in a state prior to the final state $[o]$.

No dead transitions At the beginning, all transitions are reachable and can be fired.

The queue and the shipping composition

The analysis is done on Figure 5.9.

Option to complete There are four possibilities to terminate with an attended end :

- `s!`
- `q!` $\rightarrow \tau \rightarrow$ `s!`
- `q!` $\rightarrow \tau \rightarrow$ `q!` $\rightarrow$ `q?` $\rightarrow$ `s!`
- `q!` $\rightarrow \tau \rightarrow$ `q!!` $\rightarrow$ `q??` $\rightarrow$ `s!`

The other transitions always trigger to a fail state (too many to enumerate each of them, it is the same process as before).

$\Rightarrow$ The final state $[o]$ is reachable from the initial state $[i]$.

Proper completion Using the same transitions as above, no token remains in a state prior to the final state $[o]$.

No dead transitions At the beginning, all transitions are reachable and can be fired.

5.3 Analysis of Petri nets

The previous points may have seen a whole process of analysis and modelling and even some verification. Some of these actions are automated, such as the transition from π-calculus to Petri nets.

However, up to the previous point, the analyses were still done manually. All this is feasible, but on large networks it can become time consuming and tedious. This is why having tools that automate the thing can be useful.

As explained earlier, tools exist to do automated analysis on Petri nets. One of these tools is WoPeD. The aim of this section is to present some analysis possibilities without going into detail about each one because there are too many. The question to ask is what we want to check and how can these tools help us. Analyses have already been conducted earlier, so they will not be repeated here, there is no point in doing the analyses several times. The part here is just as an indication of what is possible.

5.3.1 Semantical analysis

A semantic analysis can include many different analyses. WoPeD offers the possibility to check several properties. In the previous section, I have done the soundness analysis manually to explain the process. Here it is automated although one thing differs. In my manual analysis, I focused on the properties given here: [53]. All the properties listed are explained in Figure 5.10 just below. A report is generated and it is easy to navigate and understand the errors.

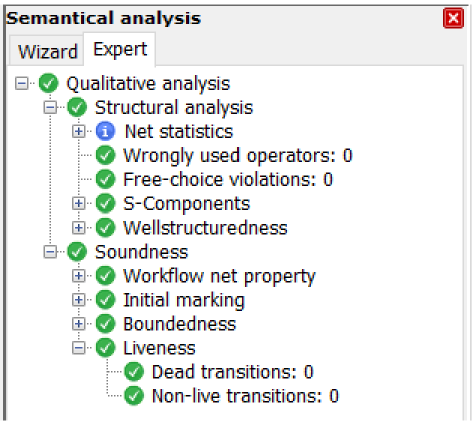


Figure 5.10: Semantical analysis of an example component

5.3.2 Coverability graph

Another possibility of this tool is to generate a coverability graph. This allows you to see all the possible paths that the process can take. This allows several things to be identified easily. Firstly, it allows you to see all the endpoints of the process. This makes it easy to see if the process has an expected behaviour. Secondly, still for the expected behaviour, it is easy to see if the transitions are the expected ones or if there is a problem in the modelling.

Figure 5.11 shows a coverability graph generated by WoPeD for the queue component. We can directly see where the process can crash (after a delta transition). Finally, we can see that the other possible solutions are to answer OK (`q!`) or KO (`q!!`). So this service meets the expected behaviour, and we can be sure that it will not cause any problem (in standalone, this situation may change with the composition of other components).

To come back to the initial objectives, we can say that this graph allows us to see the interactions of the component by specifying its inputs and outputs. Moreover, we can easily see the endpoints and its expected behaviour.

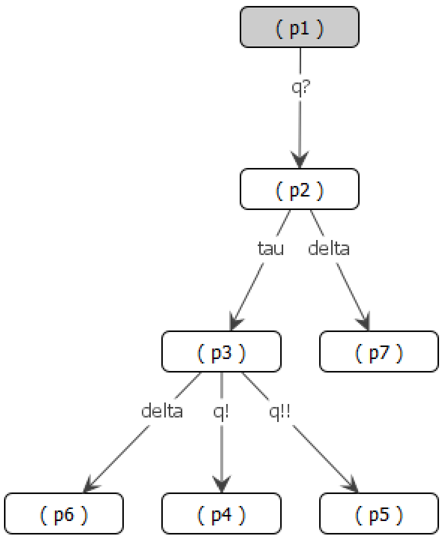


Figure 5.11: Coverability graph of the queue component

5.3.3 Token game

Another feature of WoPeD is the ability to automatically generate token games. This allows to see the different paths like the previous graph (Figure 5.11) but here in an interactive way.

Figure 5.12 below shows an example of a token game.

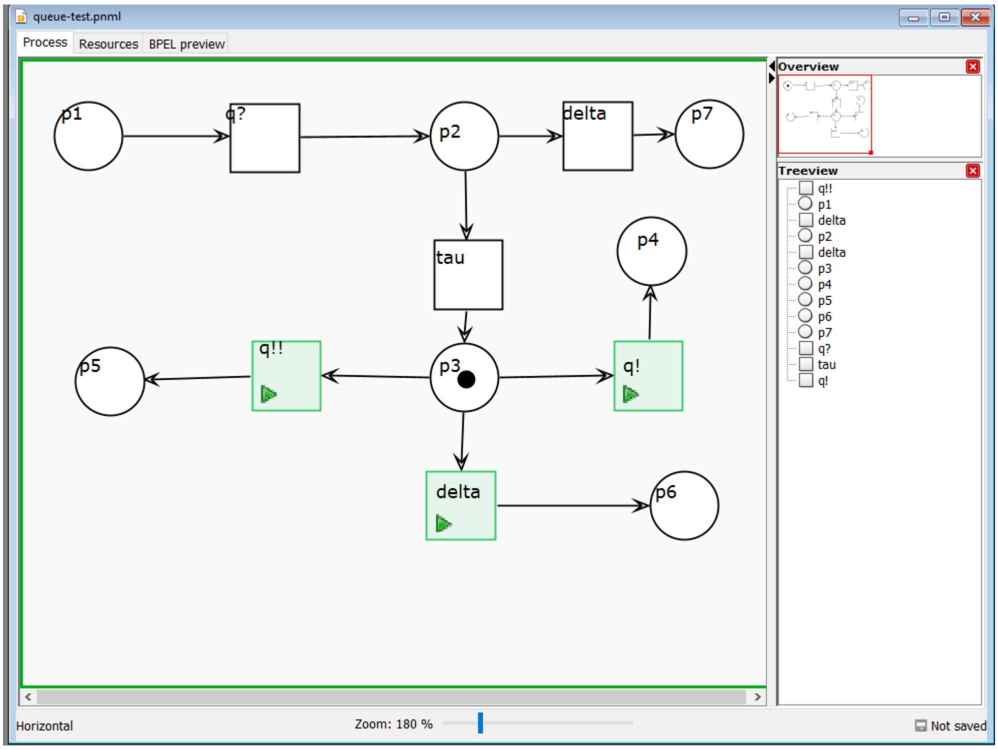


Figure 5.12: Token game in a Petri net

Figure 5.12 corresponds to the model of the queue component. The token is in the place `p3` and has three possible paths :

- `q!` : The component responds with an OK message and arrives in a final state ;
- `q!!` : The component responds with a KO message and arrives in a final state ;
- δ : The process fails, and the component is in a deadlock state.

5.3.4 Other properties

WoPed also offers other analysis possibilities. Figure 5.13 below shows the menu bar with all the possibilities. It is divided into three main parts :

- **Analysis tools** : Some of these tools have been presented above. They allow reasoning and analysis of the network at the level of its components and how it will evolve. This part allows to check the different properties.
- **Process metrics** : These metrics give information about the network such as the number of places, arcs, but also the density, etc. This information is automatically calculated on the network.
- **NLP Tools** : This section focuses on integration with NLP [61]. This is a topic that has not been discussed in this document.

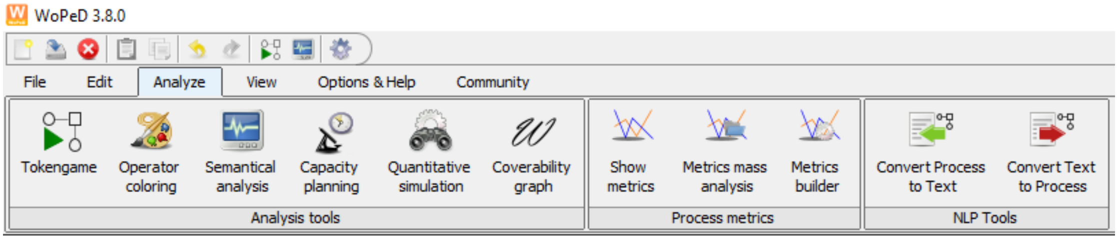


Figure 5.13: Possible analysis offered by WoPeD

5.4 Conclusion

The problem at the beginning of this chapter was that we had a model using process algebra, but the verification possibilities were limited. There was a need to extend this model to check other properties. This was possible thanks to this translation into PNML which allows automating the generation of Petri nets from the model previously made. A tool has been developed to make this automatic. These new visualisations allowed to have a new look and to understand differently the processes and their interrelations. The analysis and graphing is automated, which allows for quick and easy visualisation of the results. One of the major properties that this chapter focused on was the soundness.

Chapter 6

Conclusion

For this thesis, I was able to present a modelling and analysis tool for microservices. This one had for objectives to specify the communications and the behaviour of these, to be able to analyse the interactions and to define if they integrate well together or not.

Some modelling tools and languages already existed, but they did not fill all the boxes to meet all the objectives. Therefore, it was necessary to adapt an existing model to continue the analyses and take them even further than what had been done previously.

The overall process used to analyse these different microservices is the following. First, the processes must be specified using a process algebra based approach to define how their behaviours are. This first tool allows checking if the process terminates and terminates well with another process. Secondly, it is possible to transform this modelling into Petri nets by using PNML. This allows the analysis to be taken even further. This process is automated thanks to a tool that automatically compiles PNML files from descriptions made with process algebra.

The aim of this new approach was to highlight certain points. The most important was the communication between the components to be able to answer these questions. (1) Can one service interact correctly with another and handle its messages correctly ? (2) Can the service tolerate an unexpected error ? (3) How does scaling with replicas work ? These questions were answered this way :

1. The π-calculus model allows to check if the components can terminate together. This question of managing correctly is therefore implied because one can ensure that all paths reach a correct end. In addition, Petri nets help to highlight these exchanges by providing a more visual model and by providing a coverability graph.
2. This point has been added by the failure-free scenario. The aim was to extend the analysis already carried out to this specific case.
3. The replicas management was also be done with the π-calculus process.

This whole process, which is partly automated, therefore allows microservices to be modelled and analysed from a new perspective.

But this tool can still be improved. For the time being, the verification of π-calculus based models is done manually to see if the processes finish well. So it would be nice to develop a tool to do this automatically. This would make the tool more comfortable to use and, more importantly, easy to use on a large scale. Although it is usable, it can still be tedious to use.

Still in this automation of the process algebra, the generation of this model is handly done. There is a method with guidelines to follow, but it is difficult to develop a tool that works with all programming languages. Indeed, it is necessary either that the developer defines all the behaviour or else, to automate the process, it is necessary to do a code analysis. This tool would be a big plus, again, to allow the tool to be deployed on a large scale. But I am aware of the work that it can represent and unfortunately, it was not possible with limited time.

To stay with process automation, another possible automation, although different from the modelling and analysis already proposed, would be to propose solutions automatically to problems encountered. For example, if the analysis carried out with process algebra highlights that service A does not know how to handle some response cases, it would be nice to have recommendations of what to add. At the first place this should not be something very advanced but simply say which paths are not handled (this is only visible manually at the moment). In a second step, the tool could become more intelligent to automatically suggest adaptations to be made.

In general, the automation of modelling and analysis is always a plus as it saves time and allows for larger applications to be considered.

Finally, a last improvement that could be nice would be to have an interface from which everything can be managed. This would allow you to run the models, transform them and analyse the different models all in one place.

Bibliography

[1] L. Mauersberger, *Microservices: What they are and why use them*, May 2022. [Online]. Available: https://www.leanix.net/en/blog/a-brief-history-of-microservices.

[2] N. Dragoni, S. Giallorenzo, A. L. Lafuente, *et al.*, "Microservices: Yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*, M. Mazzara and B. Meyer, Eds. Cham: Springer International Publishing, 2017, pp. 195–216, ISBN: 978-3-319-67425-4. DOI: 10.1007/978-3-319-67425-4_12. [Online]. Available: https://doi.org/10.1007/978-3-319-67425-4_12.

[3] M. Fowler and J. Lewis, *Microservices*, Mar. 2014. [Online]. Available: https://www.martinfowler.com/articles/microservices.html.

[4] R. Gnatyk, *Microservices vs monolith: Which architecture is the best choice for your business?* Oct. 2018. [Online]. Available: https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/.

[5] N. C. Mendonca, C. M. Aderaldo, J. Camara, and D. Garlan, "Model-based analysis of microservice resiliency patterns," in *2020 IEEE International Conference on Software Architecture (ICSA)*, 2020, pp. 114–124. DOI: 10.1109/ICSA47634.2020.00019.

[6] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel, "The pains and gains of microservices: A systematic grey literature review," *Journal of Systems and Software*, vol. 146, pp. 215–232, 2018, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2018.09.082. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121218302139.

[7] *Sock shop.* [Online]. Available: https://microservices-demo.github.io/.

[8] GoogleCloudPlatform, *Googlecloudplatform/microservices-demo: Sample cloud-native application with 10 microservices showcasing kubernetes, istio, grpc and opencensus.* [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo.

[9] FudanSELab, *Fudanselab/train-ticket: Train ticket - a benchmark microservice system.* [Online]. Available: https://github.com/FudanSELab/train-ticket.

[10] *Lessons learned from scaling uber to 2000 engineers, 1000 services, and 8000 git repositories.* [Online]. Available: http://highscalability.com/blog/2016/10/12/lessons-learned-from-scaling-uber-to-2000-engineers-1000-ser.html.

[11] T. Mauro, *Microservices at netflix: Lessons for architectural design*, Aug. 2021. [Online]. Available: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.

[12] A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara, "From monolithic to microservices: An experience report from the banking domain," *IEEE Software*, vol. 35, no. 3, pp. 50–55, 2018. DOI: 10.1109/MS.2018.2141026.

[13] S. Sarkar, G. Vashi, and P. Abdulla, "Towards transforming an industrial automation system from monolithic to microservices," in *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, 2018, pp. 1256–1259. DOI: 10.1109/ETFA.2018.8502567.

[14] E. Djogic, S. Ribic, and D. Donko, "Monolithic to microservices redesign of event driven integration platform," in *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2018, pp. 1411–1414. DOI: 10.23919/MIPRO.2018.8400254.

[15] J. Soldani, G. Muntoni, D. Neri, and A. Brogi, "The μtosca toolchain: Mining, analyzing, and refactoring microservice-based architectures," *Software: Practice and Experience*, 2021. DOI: 10.1002/spe.2974.

[16] J. L. Gross and T. W. Tucker, *Topological graph theory.* Courier Corporation, 2001.

[17] S.-P. Ma, C.-Y. Fan, Y. Chuang, I.-H. Liu, and C.-W. Lan, "Graph-based and scenario-driven microservice analysis, retrieval, and testing," *Future Generation Computer Systems*, vol. 100, pp. 724–735, 2019, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2019.05.048. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X19302614.

[18] *Gherkin syntax.* [Online]. Available: https://cucumber.io/docs/gherkin/.

[19] Cucumber, *Cucumber/common: A monorepo of common components - building blocks for implementing cucumber in various languages.* [Online]. Available: https://github.com/cucumber/common.

[20] M. Soeken, R. Wille, and R. Drechsler, "Assisted behavior driven development using natural language processing," in *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, Springer, 2012, pp. 269–287.

[21] A. Panda, M. Sagiv, and S. Shenker, "Verification in the age of microservices," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, ser. HotOS '17, Whistler, BC, Canada: Association for Computing Machinery, 2017, pp. 30–36, ISBN: 9781450350686. DOI: 10.1145/3102980.3102986.

[22] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "Softnic: A software nic to augment hardware," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155, May 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html.

[23] *Vpp/what is vpp?* [Online]. Available: https://wiki.fd.io/view/VPP/What_is_VPP%3F.

[24] V. Yussupov, U. Breitenbücher, C. Krieger, F. Leymann, J. Soldani, and M. Wurster, "Pattern-based modelling, integration, and deployment of microservice architectures," in *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)*, 2020, pp. 40–50. DOI: 10.1109/EDOC49727.2020.00015.

[25] O. Ltd, *Home.* [Online]. Available: https://www.openfaas.com/.

[26] F. Rademacher, S. Sachweh, and A. Zündorf, "A modeling method for systematic architecture reconstruction of microservice-based software systems," in *Enterprise, Business-Process and Information Systems Modeling*, S. Nurcan, I. Reinhartz-Berger, P. Soffer, and J. Zdravkovic, Eds., Cham: Springer International Publishing, 2020, pp. 311–326, ISBN: 978-3-030-49418-6.

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

[28] F. Rademacher, J. Sorgalla, P. Wizenty, S. Sachweh, and A. Zündorf, "Graphical and textual model-driven microservice development," in *Microservices: Science and Engineering*, A. Bucchiarone, N. Dragoni, S. Dustdar, *et al.*, Eds. Cham: Springer International Publishing, 2020, pp. 147–179, ISBN: 978-3-030-31646-4. DOI: 10.1007/978-3-030-31646-4_7. [Online]. Available: https://doi.org/10.1007/978-3-030-31646-4_7.

[29] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice architecture: aligning principles, practices, and culture.* " O'Reilly Media, Inc.", 2016.

[30] C. Guidi, I. Lanese, M. Mazzara, and F. Montesi, "Microservices: A language-based approach," in *Present and Ulterior Software Engineering*, M. Mazzara and B. Meyer, Eds. Cham: Springer International Publishing, 2017, pp. 217–225, ISBN: 978-3-319-67425-4. DOI: 10.1007/978-3-319-67425-4_13.

[31] Clearsy.com, *B.org.* [Online]. Available: http://www.event-b.org/.

[32] M. Butler, "Decomposition structures for event-b," in *Integrated Formal Methods*, M. Leuschel and H. Wehrheim, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 20–38, ISBN: 978-3-642-00255-7.

[33] J. R. Abrial, "The b tool (abstract)," in *VDM '88 VDM — The Way Ahead*, R. E. Bloomfield, L. S. Marshall, and R. B. Jones, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 86–87, ISBN: 978-3-540-45955-2.

[34] [Online]. Available: `http://deploy-eprints.ecs.soton.ac.uk/11/3/notation-1.5.pdf`.

[35] J.-R. Abrial, *Rodin tutorial*, Jan. 1970. [Online]. Available: `http://deploy-eprints.ecs.soton.ac.uk/10/`.

[36] J. C. Baeten, "A brief history of process algebra," *Theoretical Computer Science*, vol. 335, no. 2-3, pp. 131–146, 2005.

[37] R. Milner, *A calculus of communicating systems*. Springer, 1980.

[38] U. Herzog, "Formal methods for performance evaluation," in *School organized by the European Educational Forum*, Springer, 2000, pp. 1–37.

[39] D. D. McCracken and E. D. Reilly, "Backus-naur form (bnf)," in *Encyclopedia of Computer Science*, 2003, pp. 129–131.

[40] *π-calculus*, Mar. 2022. [Online]. Available: `https://en.wikipedia.org/wiki/%5C%CE%5C%A0-calculus`.

[41] P. Quaglia and B. L. S. LS, "The pi-calculus: Notes on labelled semantic," *Bulletin of the EATCS*, vol. 68, pp. 104–114, 1999.

[42] C. A. Petri and W. Reisig, "Petri net," *Scholarpedia*, vol. 3, no. 4, p. 6477, 2008.

[43] W. M. van der Aalst, K. M. van Hee, A. H. ter Hofstede, *et al.*, "Soundness of workflow nets: Classification, decidability, and analysis," *Formal Aspects of Computing*, vol. 23, no. 3, pp. 333–363, 2011. DOI: `10.1007/s00165-010-0161-4`.

[44] W. Czerwiński, S. Lasota, R. Lazić, J. Leroux, and F. Mazowiecki, "The reachability problem for petri nets is not elementary," *J. ACM*, vol. 68, no. 1, Dec. 2020, ISSN: 0004-5411. DOI: `10.1145/3422822`. [Online]. Available: `https://doi.org/10.1145/3422822`.

[45] *Petri net*, Apr. 2022. [Online]. Available: `https://en.wikipedia.org/wiki/Petri_net`.

[46] E. W. Mayr, "An algorithm for the general petri net reachability problem," *SIAM Journal on Computing*, vol. 13, no. 3, pp. 441–460, 1984. DOI: `10.1137/0213029`. eprint: `https://doi.org/10.1137/0213029`. [Online]. Available: `https://doi.org/10.1137/0213029`.

[47] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989. DOI: `10.1109/5.24143`.

[48] S. Cayir and M. Uçer, "An algorithm to compute a basis of petri net invariants," in *4th ELECO Int. Conf. on Electrical and Electronics Engineering. UCTEA, Bursa, Turkey*, 2005.

[49] J. C. Baez and J. Master, "Open petri nets," *Mathematical Structures in Computer Science*, vol. 30, no. 3, pp. 314–341, 2020. DOI: `10.1017/S0960129520000043`.

[50] W. Aalst, "Verification of workflow nets," Jan. 1997, pp. 407–426, ISBN: 3-540-63139-9.

[51] W. M. P. van der Aalst, K. M. van Hee, A. H. M. ter Hofstede, *et al.*, "Soundness of workflow nets: Classification, decidability, and analysis," *Form. Asp. Comput.*, vol. 23, no. 3, pp. 333–363, May 2011, ISSN: 0934-5043. DOI: `10.1007/s00165-010-0161-4`. [Online]. Available: `https://doi.org/10.1007/s00165-010-0161-4`.

[52] T. Freytag, "Woped–workflow petri net designer," *University of Cooperative Education*, pp. 279–282, 2005.

[53] *Ml wiki*. [Online]. Available: `http://mlwiki.org/index.php/Workflow_Soundness`.

[54] M. Weber and E. Kindler, "The petri net markup language," in *Petri Net Technology for Communication-Based Systems: Advances in Petri Nets*, H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 124–144, ISBN: 978-3-540-40022-6. DOI: `10.1007/978-3-540-40022-6_7`. [Online]. Available: `https://doi.org/10.1007/978-3-540-40022-6_7`.

[55] *Petri net markup language*, Nov. 2013. [Online]. Available: `https://en.wikipedia.org/wiki/Petri_Net_Markup_Language`.

[56] LIP6/MoVe, *Pnml*. [Online]. Available: https://www.pnml.org/.

[57] Microservices-Demo, *Microservices-demo/shipping: Shipping service for microservices-demo application*. [Online]. Available: https://github.com/microservices-demo/shipping.

[58] M. Llorens, J. Oliver, J. Silva, and S. Tamarit, "Translating csp specifications to equivalent petri nets.," in *PDPTA*, 2010, pp. 320–326.

[59] A. Brogi, A. Canciani, J. Soldani, and P. Wang, "A petri net-based approach to model and analyze the management of cloud applications," in *Transactions on Petri Nets and Other Models of Concurrency XI*, M. Koutny, J. Desel, and J. Kleijn, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 28–48, ISBN: 978-3-662-53401-4. DOI: 10.1007/978-3-662-53401-4_2. [Online]. Available: https://doi.org/10.1007/978-3-662-53401-4_2.

[60] F. Romain, *Translator from ccs to pnml*, Jun. 2022. [Online]. Available: https://github.com/UNamurCSFaculty/MEMOIRE_romainf/blob/main/translator/translator.py.

[61] K. Chowdhary, "Natural language processing," *Fundamentals of artificial intelligence*, pp. 603–649, 2020.

Chapter 7

Appendices

7.1 PNML example

```
<?xml version="1.0" encoding="UTF-8"?>
<!--PLEASE DO NOT EDIT THIS FILE
Created with Workflow PetriNet Designer Version 3.2.0 (woped.org)
    -->
<pnml>
  <net type="http://www.informatik.hu-berlin.de/top/pntd/ptNetb"
      id="noID">
    <place id="p1">
      <name>
        <text>p1</text>
        <graphics>
          <offset x="40" y="80"/>
        </graphics>
      </name>
      <graphics>
        <position x="40" y="40"/>
        <dimension x="40" y="40"/>
      </graphics>
      <initialMarking>
        <text>1</text>
      </initialMarking>
    </place>
    <place id="p2">
      <name>
        <text>p2</text>
        <graphics>
          <offset x="210" y="80"/>
        </graphics>
      </name>
      <graphics>
        <position x="209" y="40"/>
        <dimension x="40" y="40"/>
      </graphics>
    </place>
    <transition id="t2">
      <name>
        <text>t2</text>
        <graphics>
          <offset x="130" y="80"/>
        </graphics>
      </name>
```

```xml
      <graphics>
        <position x="126" y="40"/>
        <dimension x="40" y="40"/>
      </graphics>
      <toolspecific tool="WoPeD" version="1.0">
        <time>0</time>
        <timeUnit>1</timeUnit>
        <orientation>1</orientation>
      </toolspecific>
    </transition>
    <arc id="a2" source="p1" target="t2">
      <inscription>
        <text>1</text>
        <graphics>
          <offset x="500.0" y="-12.0"/>
        </graphics>
      </inscription>
      <graphics/>
      <toolspecific tool="WoPeD" version="1.0">
        <probability>1.0</probability>
        <displayProbabilityOn>false</displayProbabilityOn>
        <displayProbabilityPosition x="500.0" y="12.0"/>
      </toolspecific>
    </arc>
    <arc id="a3" source="t2" target="p2">
      <inscription>
        <text>1</text>
        <graphics>
          <offset x="500.0" y="-12.0"/>
        </graphics>
      </inscription>
      <graphics/>
      <toolspecific tool="WoPeD" version="1.0">
        <probability>1.0</probability>
        <displayProbabilityOn>false</displayProbabilityOn>
        <displayProbabilityPosition x="500.0" y="12.0"/>
      </toolspecific>
    </arc>
    <toolspecific tool="WoPeD" version="1.0">
      <bounds>
        <position x="2" y="24"/>
        <dimension x="966" y="650"/>
      </bounds>
      <scale>150</scale>
      <treeWidthRight>742</treeWidthRight>
      <overviewPanelVisible>true</overviewPanelVisible>
      <treeHeightOverview>100</treeHeightOverview>
      <treePanelVisible>true</treePanelVisible>
      <verticalLayout>false</verticalLayout>
      <resources/>
      <simulations/>
      <partnerLinks/>
      <variables/>
    </toolspecific>
  </net>
</pnml>
```

7.2 PNML of the queue component

```xml
<?xml version="1.0" encoding="UTF-8">
<pnml>
    <net id="queue-net">
        <place id="p1">
            <name>
                <text>p1</text>
            </name>
            <initialMarking>
                <text>1</text>
            </initialMarking>
        </place>
        <place id="p2">
            <name>
                <text>p2</text>
            </name>
        </place>
        <transition id="t1">
            <name>
                <text>q?</text>
            </name>
        </transition>
        <arc id="a1" source="p1" target="t1"> </arc>
        <arc id="a2" source="t1" target="p2"> </arc>
        <place id="p3">
            <name>
                <text>p3</text>
            </name>
        </place>
        <transition id="t2">
            <name>
                <text>tau</text>
            </name>
        </transition>
        <arc id="a3" source="p2" target="t2"> </arc>
        <arc id="a4" source="t2" target="p3"> </arc>
        <place id="p4">
            <name>
                <text>p4</text>
            </name>
        </place>
        <transition id="t3">
            <name>
                <text>q!</text>
            </name>
        </transition>
        <arc id="a5" source="p3" target="t3"> </arc>
        <arc id="a6" source="t3" target="p4"> </arc>
        <place id="p5">
            <name>
                <text>p5</text>
            </name>
        </place>
        <transition id="t4">
            <name>
                <text>q!!</text>
            </name>
```

```
        </transition>
        <arc id="a7" source="p3" target="t4"> </arc>
        <arc id="a8" source="t4" target="p5"> </arc>
        <place id="p6">
            <name>
                <text>p6</text>
            </name>
        </place>
        <transition id="t5">
            <name>
                <text>delta</text>
            </name>
        </transition>
        <arc id="a9" source="p3" target="t5"> </arc>
        <arc id="a10" source="t5" target="p6"> </arc>
        <place id="p7">
            <name>
                <text>p7</text>
            </name>
        </place>
        <transition id="t6">
            <name>
                <text>delta</text>
            </name>
        </transition>
        <arc id="a11" source="p2" target="t6"> </arc>
        <arc id="a12" source="t6" target="p7"> </arc>
    </net>
</pnml>
```

7.3 PNML of the shipping component

This model is "failure-free".

```
<?xml version="1.0" encoding="UTF-8"?>
<pnml>
  <net id="shipping-net">
    <place id="p1">
      <name>
        <text>p1</text>
      </name>
      <initialMarking>
        <text>1</text>
      </initialMarking>
    </place>
    <place id="p2">
      <name>
        <text>p2</text>
      </name>
    </place>
    <place id="p3">
      <name>
        <text>p3</text>
      </name>
    </place>
    <place id="p4">
      <name>
        <text>p4</text>
      </name>
    </place>
    <place id="p5">
      <name>
        <text>p5</text>
      </name>
    </place>
    <place id="p6">
      <name>
        <text>p6</text>
      </name>
    </place>
    <place id="p7">
      <name>
        <text>p7</text>
      </name>
    </place>
    <transition id="t1">
      <name>
        <text>s!</text>
      </name>
    </transition>
    <transition id="t2">
      <name>
        <text>q!</text>
      </name>
    </transition>
    <transition id="t3">
      <name>
        <text>q?</text>
      </name>
```

```
    </transition>
    <transition id="t4">
      <name>
        <text>s!</text>
      </name>
    </transition>
    <transition id="t5">
      <name>
        <text>tau</text>
      </name>
    </transition>
    <transition id="t6">
      <name>
        <text>s!</text>
      </name>
    </transition>
    <transition id="t7">
      <name>
        <text>q??</text>
      </name>
    </transition>
    <transition id="t8">
      <name>
        <text>s!</text>
      </name>
    </transition>
    <arc id="a1" source="p1" target="t1"> </arc>
    <arc id="a2" source="t1" target="p2"> </arc>
    <arc id="a3" source="p1" target="t2"> </arc>
    <arc id="a4" source="t2" target="p3"> </arc>
    <arc id="a5" source="p3" target="t3"> </arc>
    <arc id="a6" source="p3" target="t7"> </arc>
    <arc id="a7" source="t3" target="p4"> </arc>
    <arc id="a8" source="p4" target="t4"> </arc>
    <arc id="a9" source="t4" target="p7"> </arc>
    <arc id="a10" source="p3" target="t5"> </arc>
    <arc id="a11" source="t5" target="p5"> </arc>
    <arc id="a12" source="p5" target="t6"> </arc>
    <arc id="a13" source="t6" target="p7"> </arc>
    <arc id="a14" source="t7" target="p6"> </arc>
    <arc id="a15" source="p6" target="t6"> </arc>
    <arc id="a16" source="t6" target="p7"> </arc>
  </net>
</pnml>
```