UQAR
Université du Québec
à Rimouski

VERS UNE DESCRIPTION ÉVOLUTIVE ET UNE EXPLORATION EFFICACE DES CONCEPTS ET DES ARTEFACTS D'ARCHITECTURE MICROSERVICES

Mémoire présenté

dans le cadre du programme de maîtrise en informatique

en vue de l'obtention du grade de maître ès sciences

PAR

© Gabriel Morais

Lévis, Québec, Canada

Juillet 2021

ACKNOWLEDGEMENTS

I would like to thank my employer, Mouvement Desjardins, including its leadership and my fellow employees for being supportive all the years I have been studying and in particular, the time I have been working on this thesis. My thanks also go to Prof. Dr. Mehdi Adda, my supervisor for excellent advice and support.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SOURCES

RÉSUMÉ

L'adoption de l'architecture Microservices (MSA) pour la conception de systèmes logiciels est une tendance en industrie et en recherche. De nature compositionnelle et distribuée, les systèmes basés sur l'architecture Microservices sont composés de services ayant une responsabilité restreinte et bien définie, visant un isolement complet dans une perspective de non-partage de ressources. Les systèmes basés sur des microservices sont souvent classés comme de systèmes « Cloud-Native ».

L'adoption de l'architecture Microservices représente un changement de paradigme technologique et managérial comportant des défis, notamment : la taille, la portée et le nombre de services, et leurs interopérabilité et réutilisation. Outre ces défis, la compréhension, l'adoption et l'implémentation des principes fondamentaux de ce style architectural sont des challenges qui impactent la conception d'architectures microservices efficaces et cohérentes.

En effet, l'absence d'un large consensus sur certains principes et termes clés de cette architecture mènent à sa mauvaise compréhension et par conséquent à des implémentations incorrectes. Cette absence de consensus est une manifestation concrète de l'immaturité de cette architecture qui mène à des défis lors de la formalisation des connaissances.

Également, il manque une méthode uniforme capable de supporter les concepteurs lors de la

modélisation des microservices, notamment dans l'agencement des différentes composantes. À cela s'ajoute l'absence de modèles conceptuels pouvant guider les ingénieurs dans les premières phases de conception de ces systèmes. Plusieurs approches ont été utilisées pour la modélisation d'architectures microservices, tels que : formelle et informelle, manuelle et automatique et toutes les combinaisons de ces quatre, mais ces approches ne répondent pas à tous les défis rencontrés par les concepteurs.

Pour faciliter la modélisation des microservices et rendre le processus plus efficace, il est nécessaire de développer des approches de conception et de représentation alternatives. Dans cette perspective, nous proposons une approche ontologique capable de répondre autant aux défis de conception que de représentation des architectures microservices. Dans ce mémoire, nous vous présentons nos résultats de recherche dont la principale contribution est une ontologie du domaine des architectures Microservices définie en suivant les principes de logique de description et formalisée en utilisant le langage « Web Ontology Language » (OWL), une technologie clé du Web sémantique. À cette ontologie nous avons donné le nom d'« Ontology of Microservices Architecture Concepts » (OMSAC).

OMSAC contient suffisamment de vocabulaire pour décrire les concepts qui définissent l'architecture Microservices et pour représenter les différents artefacts composant ces architectures. Sa structure permet une évolution rapide et est capable de prendre en charge les enjeux liés à l'immaturité actuelle de ces architectures.

En tant que technologie d'intelligence artificielle (IA), les ontologies possèdent des capacités de raisonnement avancées auxquelles il est possible d'ajouter d'autres technologies pour les étendre et ainsi répondre à différents besoins. Avec cet objectif, nous avons utilisé OMSAC conjointement avec des techniques d'apprentissage machine pour modéliser et analyser des architectures microservices afin de calculer le degré de similitude entre différents microservices appartenant à différents systèmes.

Ce cas d'utilisation d'OMSAC constitue une contribution supplémentaire de notre recherche et renforce les perspectives de recherche dans l'assistance, l'outillage et l'automatisation de la modélisation des architectures microservices. Cette contribution montre également la pertinence de la recherche de mécanismes permettant de faire de l'analytique avancée sur les modèles d'architectures.

Dans des travaux de recherche futurs, nous nous intéresserons au développent de ces mécanismes, et planifions la conception d'un assistant intelligent capable de projeter des architectures microservices basées sur les meilleures pratiques et favorisant la reutilisation de microservices existants. Également, nous souhaitons développer un langage dédié afin d'abstraire les syntaxes d'OWL et du langage de requête SPARQL pour faciliter l'utilisation d'OMSAC par les concepteurs, ingénieurs et programmeurs qui ne sont pas familiers avec ces technologies du Web sémantique.

ABSTRACT

The use of Microservices Architecture (MSA) for designing software systems has become a trend in industry and research. Adopting MSA represents a technological and managerial shift with challenges including the size, scope, number, interoperability and reuse of microservices, modelling using multi-viewpoints, as well as the adequate understanding, adoption, and implementation of fundamental principles of the Microservices Architecture. Adequately undertaking these challenges is mandatory for designing effective MSA-based systems.

In this thesis, we explored an ontological representation of the knowledge concerning the Microservices Architecture domain. This representation is capable of addressing MSA understanding and modelling challenges. As a result of this research, we propose the Ontology of Microservices Architecture Concepts (OMSAC), which is a domain ontology containing enough vocabulary to describe MSA concepts and artifacts and in a form to allow fast evolution and advanced analytical capabilities.

PREFACE

Some parts of Chapter 1, 2, 3, 4, 5 and 7 have been published in G. Morais and M. Adda, "OMSAC—Ontology of Microservices Architecture Concepts," 2020 11th IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON), Vancouver, BC, Canada, 2020, pp. 0293-0301. Being the first author, I was responsible for all major areas of concept formation, data collection and analysis, development, and manuscript composition. Professor Dr. Mehdi Adda was the supervisory author and contributed to manuscript edits.

CHAPTER 1

INTRODUCTION

Microservices Architecture (MSA) is a recent architectural style, considered as a Cloud-native architecture [32] that has met significant success in the industry and has had a growing interest in academia [33]. MSA handles complexity by decomposing large systems and by bringing modularity "to the next level," [38] facilitating the reuse of components [91]. Also, MSA represents a shift in the way software systems are designed, developed, deployed, and operated.

The lack of knowledge regarding the core concepts and principles of this emerging architectural style can lead to unsuitable implementations [15] and furthermore is a challenge for early and unexperimented adopters [72]. Thus, formalizing and sharing knowledge about MSA core principles is mandatory to ensure its proper implementation. Moreover, learning MSA principles and practices should be facilitated for the designer and can be accomplished by giving them access to formalized MSA knowledge [90]. Having access to this knowledge could speed up the creation of lacking supporting tools (e.g., design patterns and methods).

Many studies have been published on Microservices Architecture principles and concepts [50, 96], mainly taxonomies and classifications [90], patterns and anti-patterns catalogues [119, 12]. However, there are still critical issues in this domain including the absence of a broad consensus on the formal definition of Microservices Architecture [142], and a clear perspective in developing widely accepted recurrent solutions and architectural strategies as much in the industry as in academia [94].

Furthermore, architecting MSA-based systems means the adoption of a compositional design approach. Thus, the resulting architecture is a set of microservices composed to meet business requirements. This compositional approach encourages reuse, facilitating systems development, and is one of the Microservices Architecture (MSA) masterpieces [91]. Therefore, designers composing a microservices-based system should be able to identify existing reusable microservices that could be used as-is or extended to compose new systems, leading to faster systems development [9, 18] and increased quality. Thus, identifying similar microservices is a prerequisite to enable microservices reuse.

In addition, the granularity of microservice-based systems can lead to an exponential difficulty in modelling architectures in a fine-grained way [18]. These aspects have an impact on microservices reuse, interoperability and interchangeability. Consequently, it seems necessary to implement intelligent mechanisms to deal with the complexity of modelling microservices architectures. This then makes the challenge twofold because it becomes necessary to describe microservices architectures that allow multiple viewpoints and then build mechanisms that allow features and microservices identification and discovery [72, 21]. Consequently, modelling microservices calls for alternative modelling approaches to manage these challenges.

We believe that solving these two challenges will lead to facilitating the identification of microservices holistically and allow for the analysis based on multiple aspects of a microservices architecture. This is necessary to identify reuse opportunities and to establish compatibility mandatory to interoperability and interchangeability.

With this in mind, we have proposed the Ontology of Microservices Architecture Concepts (OMSAC) [85], which is an ontology for describing the Microservices Architecture domain.

This ontology contains a terminological component (TBox) composed of enough vocabulary to describe MSA concepts and modelling microservices architectures.

Our approach aims to support MSA-based systems modelling and to improve automated analysis of large MSAs by providing a tailored domain ontology. The fundamental characteristics of this approach are flexibility and extension, because our approach allows MSA systems to be represented holistically or decomposed into individual viewpoints and thus can meet different stakeholders' specific information needs. All views can be derived from the single underlying OMSAC ontology, represented with the Web Ontology Language (OWL2 DL) [54]. This enables automated processing, one example would be, to efficiently identify similar microservices in large knowledge bases. Instead of identifying manually reusable microservices by exploring and comparing different models and representations, OMSAC allows designers to describe microservices-based architectures using the same syntax and thus to produce models which can be analyzed using automatic mechanisms.

1.1 PROBLEM

Besides innate limitations that are characteristic of a recent architectural style, we need to question and examine the following challenges for MSA adoption: Lack of mechanisms facilitating microservices reuse [9], adaptation to the expertise of companies and teams [89], and insufficient supporting tools [72].

It is also apparent that the modular nature of MSA-based systems calls for a high level of microservices reuse. However, microservices-based system's architectures lacked effective representation and description support, as well as discovery and exploration mechanisms, resulting in challenges for identifying reusable microservices [9]. These challenges can lead to work duplication [89], and jeopardize the positive impact of MSA implementation.

Moreover, when implementing MSA, we must consider that not all organizations and individuals involved have the same level of knowledge and have adopted MSA's principles in the same fashion [33]. Therefore, it is necessary to apply specific strategies for each application, organization and team [134].

The popularity of MSA has created a need for tool support such that practitioners can navigate their way through this novel design style. However, we believe that practitioners need not have to wait until MSA reaches a high level of maturity to access its available knowledge. Thus, the necessary tools need to be able to deal with the immaturity of MSA and its constant evolution. Such tooling could enhance MSA understandability and prevent unsuitable implementations [15].

We believe modelling has a great potential to address several of the challenges that we have outlined. Nevertheless, there is a "lack of conceptual models able to support engineers since the early phases of MSA development" [79], and a lack of "a uniform way to model autonomous and heterogeneous microservices at a level of abstraction that allows easy interconnection through dynamic relation" [79].

Various approaches have previously been proposed to represent microservices architectures including: Informal drawings [5], UML based diagrams [77], Domain-specific Languages (DSL) [25], directed graphs [5], programming languages [61], and modelling languages [101]. However, all of these approaches failed to address common challenges in software modelling, such as analysis and exploration of multi-viewpoints and modelling in different granularity levels [95]. Indeed, each viewpoint remains a separate model in these approaches, leading to analyzing and exploring them separately.

The need for a holistic view is exacerbated when making design decisions in MSA because modelling microservices-based systems requires making decisions based on functional, technological, operational, and organizational aspects in a complex environment. In this context, we were interested in a flexible way to organize, store, and share knowledge in this field. Our key research questions are therefore:

RQ-1 How to provide a vocabulary allowing to describe both MSA knowledge and artifacts?

RQ-2 How to describe microservice-based architectures allowing to explore them from various viewpoints?

RQ-3 How to improve microservice's identification and reuse using its description?

RQ-4 How can heterogeneous stakeholders and their information needs be responded to by a holistic modelling of microservices-based systems?

1.2 MOTIVATION

Microservices Architecture has been generating considerable interest in systems modernization. Many companies consider MSA as the main lever for reducing costs [32] and time to market [72]. Thus, proper design and implementation of MSA architectures have become paramount. In this context, software architects, developers and programmers' knowledge and expertise on MSA core principles are crucial.

However, there have been divergences of opinion and practice on some aspects of the MSA. It is an intrinsic factor in any maturing architectural style or technology. It can cause misunderstanding and confusion when designing and building MSA systems, leading to slowing down MSA adoption and furthermore jeopardizing its benefits. Likewise, the lack of established modelling strategies and support tools is a barrier to easy MSA adoption.

The core concept of our research for this thesis has been motivated by these challenges. We have focused on facilitating knowledge sharing among MSA practitioners and researchers, mainly by providing them with a description of Microservices Architecture core principles in the form of terminology that can be applied for knowledge representation and modelling, and for analyzing microservices-based systems.

1.3 RESEARCH OBJECTIVES AND CONTRIBUTIONS

In this thesis, we have been interested in representing and sharing knowledge related to Microservices Architecture. For this purpose, we are proposing a description of Microservices Architecture core concepts to be available in a human and machine-readable representation. This description must contain a formal specification of the core concepts and vocabulary. It must be specified in a way allowing querying based on concepts and revealing the relationship among them, direct relations, and hidden ones. Existing works classified MSA concepts yet did not intend to construct support tooling.

We have considered ontologies as a support to build such a description. Indeed, ontologies can respond to this need because they are able to represent knowledge in an evolutive form, understandable by both humans and machines [125].

Despite the interest in using ontologies to represent concepts of a domain [92], no one, to our knowledge, has applied it to MSA concepts and principles. Rather, ontologies have been applied within microservices for building applications in various domains, as in the Internet of Things (IoT) [69] and the Web [129].

1.3.1 RESEARCH OBJECTIVES

The principal aim of this thesis is to develop an ontological representation of MSA concepts, which will allow the development of knowledge-based systems that could act as supporting tools to architects, programmers, or people in the technology field who wish to incorporate MSA for building software.

For this purpose, the objectives of this thesis are:

- Reviewing literature to collect terminology, definitions and concepts related to MSA.
- Building a terminological ontology of MSA concepts based on the terminology collected from the literature.
- Applying the developed ontology in use cases identified in the literature.

1.3.2 CONTRIBUTION

The major contribution of our work is the Ontology of Microservices Architecture Concepts (OMSAC), an evolutive and intelligent representation of Microservices Architecture concepts and principles, using Web Ontology Language (OWL) and based on existing taxonomies. OMSAC aims to represent Microservices concepts meaningfully in an evolutive and extensible manner. We presented OMSAC's preliminary results at the 11th IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON) in Vancouver, British Columbia, Canada [85].

Within the thesis structure, we will present one use case that demonstrates the potential of using OMSAC to address practical concerns in the Microservices Architecture field. The use case shows an ontology-driven conceptual modelling approach to design Microservices-based systems in multiple viewpoints using OMSAC. The created models are then used for analyzing the modelled microservices architectures to identify similarities among the microservices composing them. The results of this experiment serve as an additional contribution to enrich the research in the MSA field by proposing new research perspectives in modelling and analysis of microservices-based systems.

1.4 THESIS STRUCTURE

This thesis is comprised of eight chapters. This chapter (Chapter 1) gives a brief introduction into our subject and introduces our motivation and contributions. Chapter 2 provides a theoretical introduction to relevant concepts and background, followed by Chapter 3 which is focused on an in-depth review of related work. Chapter 4 presents the methodology used in this thesis. Chapters 5 and 6 describe the OMSAC ontology and an application use case, respectively. In Chapter 7, we evaluate OMSAC, and in Chapters 8, we are concerned with the limitations and perspectives of our subject, and we close this chapter with our conclusion, which gives a brief summary and discusses our future work.

CHAPTER 2

BACKGROUND

This chapter describes the background knowledge required for understanding core concepts developed in this thesis. It begins by introducing Cloud Computing concepts, followed by examining Microservices and Description Logics concepts, and concludes with our findings on ontological concepts.

2.1 CLOUD COMPUTING, CLOUD SOFTWARE ARCHITECTURES AND CLOUD-NATIVE SOFTWARE

Cloud Computing has been a trend in research and industry for a number of years. It is considered a significant shift in the way systems are built and deployed [113, 32], and a revolution in how computing capabilities are used [113].

Many companies have been able to successfully utilize on-demand higher computing capabilities of Cloud Computing at affordable costs [114]. What has now become a reality is a computing model based on a “pay-for-what-you-use” [113]. Indeed, because Cloud Computing is able to provide shared computing capabilities, costs have been dramatically reduced and as a result expensive investment in isolated IT infrastructures is now avoided. This is because previous approaches required significant investments in computing capabilities often only needed for a limited time [78].

Apart from this technological shift, Cloud Computing arises from the need to use novel approaches to address recent challenges in meeting end-user expectations in availability, multi-platform compatibility, and fast feature delivery [32]. Inarguably, "modern applications call for enabling rapid iteration and frequent releases, zero downtime, and a massive increase in the volume and variety of the devices connected to it," [32] pushing companies to change their practices in designing and deploying software.

2.1.1 DEFINING CLOUD COMPUTING

Despite the popularity of Cloud Computing, some confusion persists about the term Cloud [32]. Here we will look at what characterizes Cloud-native software and cloud architecture.

From a user's perspective, we can define *Cloud Computing* as an on-demand and adaptive IT infrastructure shared by various end-users [78], where computation and storage spaces are provided when needed. Another factor is that the management of physical components is transferred from the business' IT team to a Cloud provider [113]. In total, there are three main models for providing Cloud Computing that exist: Software as Service (SaaS), Infrastructure as a Service (IaaS), and Platform as a Service (PaaS) [113]. Each of these models address different customer needs. Figure 2.1.1 shows the different levels of management's transfer compared to on-premise IT infrastructures.

From a technical perspective, Cloud Computing is "a distributed Internet-based software system providing shared resources, often virtualized, as tiered services," [97] and which is "highly distributed and constantly changing" [32]. Those systems are composite, service-oriented, resilient, scalable, and stateless [113]. They rely on virtualization technologies and self-adaptation approaches, dealing with constant uncertainty, and are dynamically managed [97].

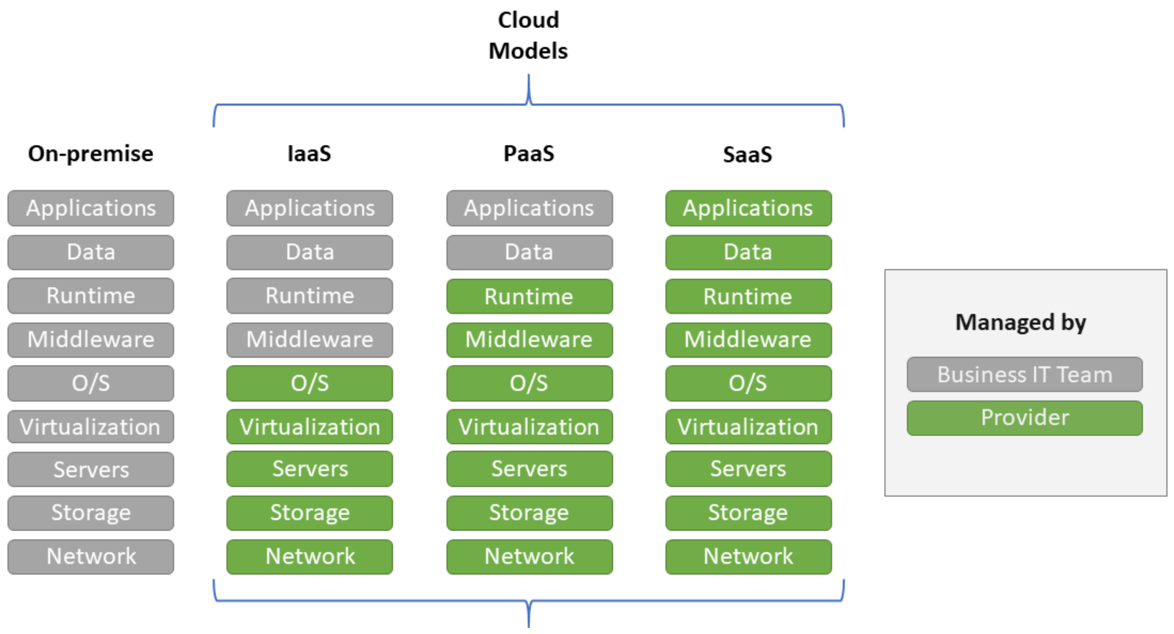


Figure 2.1: Comparison of the Levels of Management in Cloud Computing Models
On-premises computation is managed by the business IT team. Management is progressively transferred to the provider in IaaS, PaaS and SaaS models. In the SaaS model, infrastructure is complete managed by the provider.

2.1.2 COMPOSABILITY, MODULARITY AND UNCERTAINTY

To fully understand Cloud Computing and related concepts; we need to determine what are composability, modularity and uncertainty, as these three concepts become the foundation of the benefits, concerns, and challenges present in Cloud Computing.

Sosinsky [113] defined composability as the capacity of arranging modular components in order to build a composable system; and modularity as the faculty of a component to be “A self-contained and independent unit that is cooperative, reusable and replaceable.”

In this context, uncertainty represents the changing context, the heterogeneity of components, and the distributed nature of these systems [113].

2.1.3 CLOUD COMPUTING BENEFITS AND CONCERNS

Cloud Computing brings benefits such as software resilience, agility, flexibility and independence in components' evolution and recovery [97, 32]. It also provides "limitless server capacity," [114] reduced time to market, and designing, building, and deploying systems at minimal costs [113].

However, these systems face particular challenges and concerns. In fact, the components of these systems are arranged to form a whole, which leads to specific configuration and management concerns [32]. To ensure that the system works correctly, we must take into account the management of deployed instances, the configuration of the software execution cycle, and the handling of eventual outages [32]. Included is the distributed nature of those systems which leads to synchronization challenges, and concerns in the management of network latency and security [113].

2.1.4 CLOUD-NATIVE AND CLOUD SOFTWARE ARCHITECTURE

Some confusion can emerge when talking about Cloud concepts. We could consider any service available through the Internet or deployed in an outsourced IT infrastructure as a Cloud software [113]. However, being deployed on the Cloud does not indicate that the software is cloud-native; nor that it follows a cloud software architecture. This then proves that Cloud Computing and Cloud-native are two distinct concepts. To further purport this distinction, Davis [32] argued: "Cloud-native is about how and cloud computing is about where," clearly underlining the independence of the two concepts. This is demonstrated by the fact cloud-native software can be deployed in non-cloud computing infrastructures as long as those infrastructures provide the required computation power.

Davis [32] defined *Cloud-native* as a new architectural style that meets requirements such as software high availability and responsivity in a context of “large and fluctuating request volumes.” Furthermore, *Cloud-native* architectures connect devices to form a distributed “data fabric of unprecedented size,” [32] performing effective storage and processing [113].

Cloud-native software is composed of three parts [32]:

- Cloud-native application, containing the software business logic.
- Cloud-native data, storing the states of the cloud-native software.
- Cloud-native interactions; configurations that describe the software composition.

In the Cloud-native approach, both application and data are based on a modular and distributed approach which requires new methods that can address concerns of data synchronization and ensure that interactions are done in a loosely coupled way [32].

Cloud software architecture is defined as the “abstract model” employed to describe cloud software systems using adequate elements to represent application components and their relationships, including the platform they are deployed in, and the associated management elements. It “expresses a full technology stack from hardware to middleware platforms to applications” [97].

2.2 MICROSERVICES AND MICROSERVICES ARCHITECTURES

The following section features core concepts of the Microservices Architecture style.

2.2.1 MICROSERVICES ARCHITECTURE

Microservices have emerged from the industry [21]; they are small, autonomous services that work together and are “focused on doing one thing well” [91]. They are supported by a compositional paradigm based on “independent, self-manageable containerized components and cloud-native services” [97]. Microservices based architectures bring modularity “to the next level,” [38] making it possible to tackle complexity by decomposing it [38].

According to Jamshidi et al. [68], the concept behind *microservices* has been explored by the industry, particularly by Amazon and Netflix. In contrast, the term “Microservice” was not employed at that time [68]. Netflix as well as Amazon [111] have designed lightweight services, and architected systems based on them since 2008 [72]. Both companies explored a novel way to design applications to maximize Cloud Computing gains and this is why Microservices Architecture is considered to be the first Cloud-Native architecture [48] and suggested as a Cloud Computing architectural pattern [97]. In addition, Microservice adoption is often associated with organizational transformation, and in most cases in a system modernization effort, as legacy migration [33].

Apart from Netflix and Amazon, other widely known earlier adopters and promoters of this new architectural style were companies as Spotify, Twitter, and The Guardian [72, 111]. Their role in promoting Microservices as an architectural style is undeniable and their research contributions demonstrate their impact in the evolution of Microservices [111]. Although, academic research on this topic is still at an early stage, creating a disparity between the state-of-the-practice and the state-of-the-art [111].

As a service-based architectural paradigm, the distinction between Microservices and Service-oriented architecture (SOA), which is a type of architecture that results from applying service orientation when designing a system [7], is controversial [142]. Indeed, it could be considered “one way of doing SOA,” as the two main definitions of microservices, one from Lewis and Fowler [76] and the other from Newman [91], have SOA pendants [142]. Many existing SOA patterns and best practices can be found in the Microservices literature.

However, differences exist, for example, in the decentralized characteristic of those systems, which is more emphasized in microservices papers than in earlier SOA literature [142]. Thus, it can also be considered an independent style [38]. Regardless of the connections between MSA and SOA, both share a common base which is the service composition approach.

2.2.2 DEFINITION

Despite the success of Microservices, no widely accepted definition has been attributed [38]. Rather, it is often defined through a set of properties representing its core characteristics [142]. The most recurrent Microservice properties in the literature are independency (share nothing), modularity, organization around a specific functionality, and single responsibility [48]. However, not every microservices architecture will apply these properties, as observed by Bogner et al. [21]. In practice, commitment to them vary.

Zimmermann [142] thus has proposed a set of seven tenets to describe Microservices Architecture which are the following: Fine-grained interfaces, business-driven development, Cloud-native design principles (isolated state, distribution, elasticity, automated management and loose coupling), polyglot programming and persistence, lightweight container deployment, configuration automatization and management of performance and fault.

Also, concerns about business processes and the organizational management have been used to define Microservices Architecture, since those concerns rely on how the industry has implemented these architectures [21]. Indeed, studies about the state of the practice have been carried out and suggested that the adoption of Microservices is often associated with organizational transformation, and in most cases, in a system modernization effort, such as legacy migration [33, 72]. For instance, using DevOps to facilitate the implementation of “You build it, you run it” is suggested by Microservices Architecture researchers and practitioners [38], inducing some to consider that DevOps and Microservices Architectures are inseparable [79]. These organizational concerns seem to be at the heart of industrial reflection on Microservices Architecture; thus, we cannot exclude them when analyzing Microservices impacts.

Dragoni et al. [38] proposed a concise definition of Microservices Architecture: “A microservice architecture is a distributed application where all its modules are microservices.” In this thesis, we have used the definition from Dragoni et al.

2.2.3 GAINS AND CHALLENGES ENCOUNTERED

Implementing Microservices Architecture is associated with gains and pains (challenges and issues) [111]. The most relevant gains are e.g., high system availability and performance [72], breaking complexity into manageable components [38], efficient handling of maintenance and evolution [21], efficient handling of outages [32], system portability [111], higher reuse rate [91], and reduced time-to-market [72].

On the other hand, what has created challenges directly related to MSA principles include finding the appropriate size and number of microservices, correctly bounding business contexts, polyglotness management, and having adequate skill and expertise [33]. A number of studies also suggested that the inherent complexity of distributed architecture have also played an essential role in the challenges faced by industry when implementing MSA [21, 72]. What is evident is that shared persistence, monitoring, exchange mechanisms, security and modularization challenges seem to be exacerbated when building microservices-based systems [111].

In fact, microservice size seems to be a fundamental misconception among practitioners implementing MSAs [79]. It can lead to granularity, and single responsibilities issues, as there is no established size to qualify as a Microservice [79]. Indeed, the size of a microservice depends on the capabilities it implements and on the defined bounded context [33].

What appears to be the most important barrier for its adoption is the insufficient expertise and skills on MSA because it represents a “major paradigm change” for practitioners [72]. In addition, the lack of tooling, appropriate patterns, and formalized knowledge of the domain can lead to unsuitable implementations [90]. Thus, practitioners and academia call for tooling and adequate discussion on design practices [38] and patterns [90].

Despite its success in the industry, Microservices Architectures remain a novelty and are considered to be an immature architectural style [33], as there are many open issues and challenges to be addressed as well as optimizations to be performed [38].

2.2.4 MODELLING MICROSERVICES

When modelling microservices-based systems, designers need to consider modelling the application domain, the business requirements, the technical boundaries, the operational aspects, and the different organization’s flows (interactions and responsibilities) [91]. Adopting MSA is often associated with a change in the development’s paradigm (e.g., adoption of Domain-driven design [142]) and organizational approaches (e.g., DevOps adoption [79]), which must be taken into account when modelling such systems.

Modelling approaches in Microservices Architecture have not been decided upon. Indeed, Mazzara et al. [79] have reported the “lack of conceptual models able to support engineers since the early phases of MSA development” and the lack of “a uniform way to model autonomous and heterogeneous microservices at a level of abstraction that allows easy interconnection through dynamic relation.”

Model-driven approaches have been presented as an effective way to design microservice-based architectures, as these approaches are particularly adapted to address modelling challenges of complex distributed systems [102]. However, we have observed that approaches for modelling microservices remain heterogeneous in practice. Indeed, various approaches to represent microservice models have been applied: Informal drawings [5], UML based diagrams [77], Domain-specific Languages (DSL) [25], directed graphs [5], programming languages [61], and modelling languages [101].

Modelling microservice-based systems using these methods does not address common challenges in software modelling including the analysis and exploration of multi-viewpoints and modelling in different granularity levels [95], because in the previous approaches that were explored earlier, each viewpoint remained a separate model. This means that the resulting modelled viewpoints are analyzed and explored separately.

2.2.5 SIMILARITY METRICS IN MICROSERVICES

Determining similarity among microservices is paramount to addressing certain challenges namely: Reuse, interoperability, and interchangeability. Despite the importance of similarities, they seem to be mainly considered in Microservices migrations to support the decomposition of a monolithic application into microservices [108]. Indeed, approaches based on similarity detection have been applied to identify similar monolith components, which could be put together in the same microservice, or to assess the accuracy of the generated microservices architecture [40, 29].

Nevertheless, recent works from the Software Product Lines (SPL) community have demonstrated interest in the research of mechanisms to identify and establish variability among microservices [81, 9]. Their research intends to encourage joint research on SPL and MSA, as both fields consider reuse as key for faster system development and maintenance. Thus, the perspective of integrating SPL in the development of microservices-based systems has unveiled research questions on microservices identification and variability [9].

It is important to note that establishing similarity metrics in microservices-based systems is a challenge on its own because of the diversity of criteria that could be used to establish it. Benni et al. [18] have proposed a set of four factors to establish whether a microservice is interchangeable including: *sharing feature sets*, *interaction compatibility*, *communication compatibility*, and *no coevolution*. Taking this into account, properly establishing the similarity of microservices architectures remains an open challenge.

2.2.6 DISAMBIGUATION

Throughout this thesis, we use *Microservices Architecture* and *MSA* as synonyms to designate the architectural style while *microservices architecture*, without capitals, to designate a system architecture designed using the MSA style. Also, we use *Microservice* to designate the concept of implementing the MSA style while *microservice*, without capitals, to generically designate an instance of Microservice or pseudo Microservice (i.e., services labelled as Microservices but related to a Microservice anti-pattern). The terms *taxonomy*, *ontology* and *classification* have different meanings in this thesis and follow the disambiguation proposed by Van Rees [126].

2.3 DESCRIPTION LOGICS

Description Logics (DL) is a group of knowledge representation formalism that express a domain's knowledge using formal semantics based on logic [11]. It is composed of "sub-languages of first-order logic of differing complexity," [37] supporting various expressiveness levels. It organizes knowledge within a knowledge base by defining the domain's relevant concepts as a terminology (terminological component or TBox), containing intentional knowledge with little indication that it will change. It then relies on these concepts to represent properties of particular individuals or objects existing in the domain (assertion component or ABox), containing circumstantial and extensional knowledge, and thus consequently is likely to change [1].

Building a description logics knowledge base follows the human inference model of concepts and individuals classification. It defines a hierarchy between a terminology's concepts, allowing "to structure the terminology in the form of a subsumption hierarchy" [11]. In these terminological components, concepts are defined once and in an acyclic manner. They "are neither defined in terms of themselves nor in terms of other concepts that indirectly refer to them" [11]. These definitions are statements explaining relations between concepts.

These knowledge bases are often represented as directed acyclic graphs where the nodes represent the concepts and the edges their relationships [11]. This manner of structuring the concepts provides relevant information on the relationship and properties of the terminology's concepts and individuals defined using them. It enables reasoning, which is a key component of DL.

2.3.1 REASONING

In description logics, reasoning is the capacity of finding implicit consequences of "explicitly represented knowledge" [11]. This is a key capability of DL languages that allows inferring additional knowledge to strengthen DLs' modelling power [73]. It depends on DL language expressivity and in "reasoning algorithms" [73], called reasoners [124]. Most expressive languages can face high inference complexity and be undecidable [11]. In contrast, the lightest languages may not be expressive enough to represent specific concepts from a domain [37]. When designing description logics languages, one must look for the equilibrium between expressiveness and reasoning complexity according to the language application intent and available reasoning capabilities.

Baader et al. [11] have categorized four basic logical inferences in description logics: Satisfiability, subsumption, equivalence, disjointness. Satisfiability relates to the "non-contradictory" nature of the knowledge base, establishing whether it respects the domain terminology. In other words, it verifies if relations among concepts do not lead to a contradiction. Subsumption relates to the general nature of concepts compared hierarchically. It verifies if a concept is more general than the one it subsumes. Equivalence relates to the capacity of identifying concepts that are the same, and disjointness is the capacity of identifying concepts that are different [11].

Likewise, reasoners can verify knowledge base consistency by proving assertions are consistent, that is, whether the assertions in the ABox satisfy concepts' description from the TBox. Satisfiability and consistency checks are valuable to determine a knowledge base's meaningfulness [11].

Besides unveiling implicit knowledge, reasoners provide explainability by exposing the reasoning process. It supports inference understanding, which is particularly valuable to help users understand inconsistencies [124].

2.3.2 OPEN AND CLOSED WORLD ASSUMPTIONS

An essential concept in description logics reasoning is the open and closed world assumption [124]. It determines how the reasoner will infer when establishing proof [11]. A proof can be true, false or unknown in an open-world assumption, while only true or false are admitted in a closed-world one [124]. The difference in the two assumptions' mode is that the open-world assumption is aware that the represented knowledge is incomplete, explaining its impossibility to infer some assertion [11]. Using open-world assumption impacts reasoning in the knowledge base's actual and future state since new knowledge that is added does not invalidate previous proofs [53]. In contrast, the closed-world assumption on the absence of knowledge will consider the proof false.

Grimm et al. [53] have proposed a meaningful and concise illustration of these assumptions with the following: "Assum[ing] we only know that Peter is a person, from this information we can neither conclude that Peter is a vegetarian, nor that he is not one," applying the open-world assumption will conclude: *We do not know*. Indeed, "we admit the fact that our knowledge of the world is incomplete." Adding information about Peter, as "Peter is indeed a vegetarian, does not change any positive or negative conclusions." Whereas applying the closed-world assumption to the initial representation would conclude that Peter is not vegetarian. The updated information will change the previous conclusion as it will now conclude the proof is true.

Using this illustration, the choice between applying one or another of these assumptions will depend on the domain represented and the inferences one would make. Thus, the closed-world assumption would be suitable for handling situations where complete knowledge about the domain is necessary to obtain meaningful conclusions. For instance, if a train's departure is not scheduled in the train timetable, the usual "conjecture is that there is no such train," [53] other conjectures, as *we do not know*, would be meaningless for passengers. In comparison, the open-world assumption would be adequate for situations where incomplete knowledge has no impact on meaning. For instance, in a social network, considering only *A* is a friend of *B* and *B* is a friend of *C*, is *A* a friend of *C*? In this case, the conjecture *we do not know* is acceptable.

2.3.3 MODELLING WITH DESCRIPTION LOGICS

Description Logics have been employed as modelling languages for knowledge representation since the mid-1980s. They have been applied mainly for ontological modelling, being the foundational piece of the Web Ontology Language [73]. We can consider them as an "object-centered modelling language" [11] as they explicitly define objects, their properties, and relationships among them. In these models, each element relates to a concept from the domain of discourse. Thus, modelling using Description Logics requires modellers to define these concepts and describe their relations.

Baader et al. [11] have suggested that the capacity of exploiting the model's description "to draw conclusions about the problem at hand is a particular advantage of modelling using Description Logics," compared with other modelling approaches. However, modelling capabilities can be limited by the expressiveness of description logics languages [73].

2.4 ONTOLOGIES

This section presents foundational concepts related to ontologies and their applications in software engineering.

2.4.1 DEFINITION

The commonly accepted definition of ontology in systems engineering is: "an explicit specification of conceptualization" [125]. The term "ontology" is borrowed from philosophy, where it designates a branch whose subject of interest is the nature of being. In software engineering, ontologies are models which represent and codify a subject of matter [35]. They include a vocabulary of relevant terms as well as their significance, allowing the structure to ease knowledge sharing [98].

Those models are descriptive and normative, and they aim to formalize domains into knowledge structures by generically describing relevant concepts, their properties, and the relationships among them. Those descriptions are "silent regarding the specific values of the attributes that are assumed by particular instances of the concept" [88].

For instance, in the automotive domain, a vocabulary of terms defines the concept of *vehicle*, *car* and *motorcycle*, and the property "is a." Assertions as "a car is a vehicle" or "a motorcycle is a vehicle" are relationships between those terms[1]. This vocabulary allows describing the automotive domains generically. Based on these atomic terms and properties, it is possible to represent any car or motorcycle from different manufacturers. Because each representation is

[1]Complete automotive ontology is available at https://schema.org/docs/automotive.html

an instance of a concept, we can infer that any given instance can belong to a concept by the existing hidden relations, for example if A is a car; then, A would be a vehicle. Instead of other knowledge structures, ontology allows for high expressiveness and reasoning capabilities.

A knowledge base can be considered as "an instantiation (or an extension) of an ontology" [88]. In this context, ontology only designates the representational vocabulary [57] while the knowledge base contains the vocabulary and the assertions on the ontology.

2.4.2 KNOWLEDGE REPRESENTATION

Ontologies serve as a basic structure for domain knowledge representation [27]. Various domains and disciplines have used ontologies and applied them according to the needs and viewpoints of domain description, which could lead to improper communication, and an ambiguous domain description. Indeed, the diversity of domains and stakeholders' perspectives may cause conceptual and terminological incompatibilities [49], resulting in misunderstanding of concepts [124]. Thus, ontologies deal with heterogeneity [125].

To address these issues, ontologies precise semantics to ensure domain description's correctness and consistency. Consequently, they act as an unambiguous communication and mediation mechanism [35, 22] allowing shareable and understandable domain knowledge [98].

We use formal ontology languages to build ontologies, and those languages allow us to express "membership of an object to a category, generalizable relationship between categories, and designate the type of that objects linked in a relationship" [28]. Concretely, the knowledge representation is based on named graphs which are data models for objects and their relationships. In these graphs, we represent the objects as nodes and their relationships as edges, organizing knowledge through subject-predicate-object statements [28, 6].

Using ontologies was propelled by the Semantic Web [124] where it is currently a major technology for knowledge representation and sharing. In this manner, technology formalization and normalization have been at the Semantic Web community's core, resulting in the proposal of foundational standards and recommendations for knowledge representation and exploration, and have been adopted widely in the field.

These technologies compose a technological stack [3, 37]. Berners-Lee [19] has proposed a conceptual representation of the different technologies composing the Semantic Web. This representation is known as the "Web layer cake," [45] where technologies are classified into five layers: Representation, reasoning, query, trust and interaction [44]. Figure 2.2, shows an excerpt of this "layer cake" containing the technologies used in this thesis.

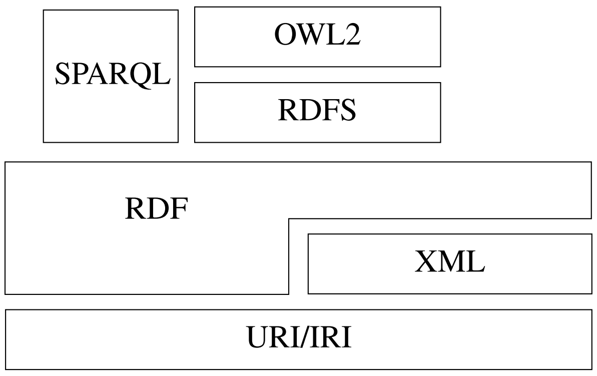


Figure 2.2: Excerpt of the Semantic Web Layer Cake [45]

The Resource Description Framework (RDF), a W3C recommendation [55], is considered the framework for building higher level languages [4]. It is flexible and allows easy creation of data models by describing a resource with simple statements formed by what is known as a triple that is composed of a subject, a predicate, and an object. Each part of a statement is a resource identified using a Uniform Resource Identifier (URI) or an Internationalized Resource Identifier (IRI), ensuring its uniqueness [124]. The statement can be formalized using a representation language (e.g., XML).

However, RDF lacks logical inference capabilities and for this reason the RDF Schema (RDFS) was proposed. RDFS is a vocabulary defining basic terminology for the Semantic Web, including the concepts of class, properties and hierarchy [3, 37]. Hierarchy and inheritance make it possible to gather what is in common and to isolate resource's particularities [3]. RDFS allows elementary inferences, such as property inheritance over a hierarchy of types and type inference from a domain or range of restrictions, while at the same time remaining simple [6].

One point to note is that the logical capabilities of RDF and RDFS combined remain limited. For instance, using them does not allow describing logical relations as disjointness, or expressing cardinalities. For enhancing logical capabilities, the W3C recommended the Web Ontology Language (OWL). OWL allows expressing complex logical statements into the knowledge representation, as disjointness and cardinalities.

Throughout this thesis, we have used the OWL2, an evolution of the first OWL recommendation [3]. OWL2 relies on the Description Logics [124], to build knowledge representations and thus introduces concepts such as disjunction, equality, cardinality, symmetry, value restrictions and logical specifications between objects [73]. Using OWL2 allows us to make more complex inferences like "equality and inequality, number restrictions, the existence of objects and others," [124] thus resulting in richer ontologies. Despite its more extensive logic capabilities than RDF and RDFS, OWL remains a lightweight language [6].

2.4.3 CHALLENGES

According to Oberle [92], the challenges of using ontologies in software development are twofold: Human and technological. First, he identified human skills as a challenge, particularly because of the lack of expertise in ontological modelling and the difficulty of hiring experts in this field which is a result of it not being a widespread discipline. Then, he identified that integrating ontologies into existing systems can be difficult because of the lack of knowledge in ontological modelling.

The complexity of the domain results in the difficulty of developing ontologies whose representation of knowledge is coherent and unambiguous [83]. Added to this, widespread consensus on the domain knowledge's representation is essential for ensuring consistency in ontological representation and allowing ontology reuse [92].

Reusing ontologies is challenging and practically unsustainable [27], as ontologies are artifacts built to represent specific knowledge linked to a precise task in a particular domain. Undeniably, the interoperability and the reuse of ontologies require consensus. *Foundational ontologies* are general vocabularies that have reached a large consensus for cross-domain applications [92]. They play a significant role in enhancing ontology interoperability and reuse [27]. *Domain-specific ontologies* formalize vocabulary in a specific universe of discourse [65], and they also allow interoperability and reuse but in a limited universe.

In fact, Gruber [57] suggested that ontological commitments allow sharing vocabulary coherently and consistently. However, he noted that these commitments do not guarantee completeness, as ontologies describe a limited part of the domain required to address specifics needs [123]. These agreements can be classified according to their desired scope which are the following:

- Individual—no agreement needed as it is used in a specific context (e.g., a project in a company [20]).
- Community—the agreement is reached for usage in a community sharing identical or complementary needs (e.g., domain ontologies).
- World—the agreement is universal (e.g., foundational ontologies) [92].

Hence, the desired scope of the knowledge representation defines whether the use of a foundational ontology is necessary, meaning that the greater the scope of an ontology, the greater the need for a foundational ontology. For instance, community or world ontologies need a broad and formal consensus because they aim to share a common ground and enhance reuse at different levels whereas an individual ontology does not require these features [92].

2.4.4 ONTOLOGY APPLICATIONS IN SOFTWARE ENGINEERING

In software engineering, ontologies have mainly been applied to build knowledge-based systems and develop Semantic Web applications [137]. However, in recent years there has been a noticeable growing interest surrounding ontologies as a mechanism to improve interoperability in various software engineering contexts [34] as well as in automation tasks, as dynamic software configuration [116].

Indeed, ontologies have been applied in various software development processes, such as requirements specification, conceptual modelling, programming, database design, or automatic code generation [106].

Bhatia et al. [20] have explored ontologies applications in software engineering and identified twenty established or potential applications in tasks of the development process, which they gathered under three intentions including as use in describing processes (e.g., the maintenance process and the quality assurance process), as artifacts in the development process (e.g., requirements, architectural models, design patterns), and as a medium to store referential information that supports the development process (e.g., documentation, description of technologies).

Furthermore, ontologies play different roles in the software life cycle which include: Completeness and consistency checker (e.g., model validation), facilitator of understanding (e.g., common vocabulary [22]), integrator (e.g., defining commonalities) and reuse enabler (e.g., resources interoperability, linked data) [100]. Ontologies can also be applied in different stages of the software life cycle in conception [43], development and at runtime [65].

These applications are not exclusive. Nonetheless, they help us understand the large application scope of ontologies in software engineering.

2.4.5 ONTOLOGIES IN SOFTWARE MODELLING

To date, applying ontologies in software modelling has been mainly done within a model-driven development approach. They have been used for requirements specification, conceptual and database modelling, and model validation [118, 106].

A model described using ontologies is considered explicit, unambiguous, and machine-processable [4, 22]. They offer explicit and consensual domain representation to support activities such as model check, consistency validation, automatic support of software modelling, and tool development [56]. They also allow combining information from various sources to build a whole that enlarges knowledge and enhances understanding and reasoning [4], which justifies the interest in using this technology in modelling activities.

One instance of this, applying ontologies in conceptual modelling produces several benefits, such as an adequate specification of the semantics, reasoning on the content of a conceptual model [128], enhancing the domain's structural and behavioural description [43], and improving domain-specific knowledge reuse [128]. Conceptual modelling "focuses on communication, learning and problem solving among human users." Using ontologies extends this definition to machine understanding, as ontologies are understandable by humans and machines [125].

Guizzardi et al. [62] have proposed the term Ontology-driven Conceptual Modelling (ODCM), which they have defined as a discipline that applies ontological theories to develop engineering artifacts "for improving the theory and practice of conceptual modelling." ODCM encompasses ontology applications in conceptual modelling beyond the use of ontological theories [127]. Indeed, ontologies (e.g., foundational and domain ontologies) have also been used to improve semantic integration to facilitate the interoperability of models, methods, languages and paradigms [127]. The ODCM is also suitable for handling complexity in modelling large systems and describing complicated domain aspects. Likewise, it improves reusability, reliability, and domain understanding.

Ontologies have also been applied in the Software Product Lines (SPL) to support and automate modelling activities such as validation and integration in feature models [141], models annotation [71], architecture derivation [39], and variability modelling [8].

2.5 THE USE OF ONTOLOGIES IN CLOUD COMPUTING AND MICROSERVICES ARCHITECTURES

In Cloud Computing there are several works put forth by different authors about the use of ontologies to represent its concepts. In 2008, Youseff et al. [139] proposed a unified Cloud Computing ontology to enhance Cloud Computing understanding. In recent years ontologies have been used to deal with the complexity of Cloud Computing, that are mainly the following: The features, diversity and heterogeneity of providers and inter-cloud portability [34, 16]. One example of this is presented by Al-Sayed et al. [2] who experimented with ontological representation to describe functional and non-functional Cloud Computing features using the OWL language.

To the best of our knowledge, in the Microservices Architecture field, ontologies have not been used to represent core Microservices concepts, as seen in Cloud Computing. Instead, ontologies have been applied within microservices for building applications on the Internet of Things (IoT) [69] and on the Web [130].

CHAPTER 3

RELATED WORK

This chapter is comprised of two themes: Reviews of related work on the definitions, taxonomies and modelling of Microservices, and our evaluation of the use of ontologies for software modelling.

3.1 CONCEPTS, TAXONOMIES AND CLASSIFICATIONS OF MICROSERVICES

3.1.1 CORE CONCEPTS, TAXONOMIES AND CLASSIFICATIONS

Zimmermann [142] is one of the first researchers proposing a review of the concepts related to Microservices Architecture. Throughout his work, “Microservices Tenets,” he recorded Microservices tenets extracted from research literature and from two widely known gray literature works from Lewis and Fowler [76] and Newman [91].

He suggested defining MSA that follows one of the classical approaches which would be based on design intent, principles and patterns, as well as based on constraints. We can therefore observe that one of the results of his study is a classification of MSA principles according to five viewpoints: Logical, development, process, physical and cross-cutting concerns. Although his work gives an in-depth view of key MSA concepts, it does lack schematized relationships between these principles.

In his seminal paper of 2018, Garriga [48] defined a taxonomy of Microservices concepts from life cycle and organizational aspects. This taxonomy covers concepts from discovery, understanding and analysis to implementation, as programming languages, techniques, platforms and tools. It also schematizes the relationships among them and the proposed taxonomy aims to be a framework to support the analysis of the Microservices domain. Garriga further applied it in a corpus of 46 works within literature that was published as early as 2016, and it was used to identify solutions to Microservices architectural issues and challenges.

In addition, Soldani et al. [111] have explored the benefits and drawbacks of the MSA implementation and suggested a taxonomy of Microservices Architecture's "pains and gains," schematized by phase and concern related to the software life cycle. In their work, they recorded advantages, challenges and issues from a set of 51 industrial studies. This approach is interesting as it fosters better knowledge about practitioners' viewpoints on MSA advantages and pitfalls, yet this taxonomy does not schematize relationships among the terms defined.

3.1.2 SPECIALIZED TAXONOMIES AND CLASSIFICATIONS

We noted that several studies have been conducted on specific aspects of the Microservices Architecture paradigm, such as patterns and anti-patterns. One such study was from Osses et al. [94] who proposed a taxonomy, which categorizes patterns and conceptual tactics in the Microservices domain. This taxonomy could be used to identify architectural patterns addressing common conceptual problems in developing MSA-based systems.

Within their study, they reviewed a large corpus of academic and industrial studies on this topic (totalling 124 papers) and which they classified observed patterns into 11 categories. This included the following: IoT patterns, DevOps patterns, Front-End patters, Back-End patterns, Orchestration patterns, Migration patterns, Communication patterns, Behaviour patterns, Design patterns, and Mitigation patterns.

In addition, they observed that the industry formalizes architectural patterns based on their own experience when finding solutions to concrete problems and which have an intention of recording them only for future use. Their work allows for the possibility of obtaining a high-level view of the concerns and aims of practitioners, as well as the concepts used by them, when these practitioners formalized knowledge about MSA practices.

In comparison, Balalaie et al. [12] proposed a taxonomy and catalogue of fifteen Microservices' migrations patterns focused on migration planning. The identified patterns have been collected from industrial migrations projects and analyzed in reference to quality factors that were organized in the two following groups: Architectural and operational. They formalized these patterns by using a template containing the following attributes: Reuse intention, reuse situation, context, problem, solution, challenges, technology stack, and similar patterns. This work enhanced the understanding of the challenges and issues of migrating systems to microservices-based architectures, and provided concepts and terms which could describe them.

The work of Neri et al.'s [90], which follows the same direction of Garriga's work [48], suggested a "taxonomy of design principles, architectural smells and corresponding refactorings." This taxonomy was applied to analyze the literature for extracting bad-smells and their associated refactoring actions. In contrast to Garriga's work, this taxonomy had a more extensive coverage of terms related to bad-smells and their relation to Microservices Architecture design principles.

Similarly, Taibi et al. [115] proposed a catalogue and taxonomy of the most common Microservices Architecture's anti-patterns. They recorded anti-patterns from the experience of practitioners who implemented Microservices Architecture. Their catalogue is composed of twenty anti-patterns and aims to help practitioners avoid implementing anti-patterns in their projects. Their work differs from Neri et al.'s [90] in that it covers a more significant number of bad-smells.

In 2020, Tighilt et al. [119] put forth a catalogue of sixteen anti-patterns recorded from the literature and 67 open-source systems developed following the MSA paradigm. In this work, they formalized the anti-patterns by following a pattern template containing the following attributes: Name, context, general form, symptoms, consequences, refactored solution, advantages of refactoring, and trade-offs. This work differs from Taibi et al.'s [115] work in that it described anti-patterns in a more detailed way by adding information about anti-patterns symptoms, effective use and implementation.

Table 3.1 summarizes the works reviewed in this section.

Work	**Type**			**Scope**		
	Taxonomy	*Classification*	*Catalogue*	*Definition Principles*	*Practice*	*Patterns Anti-patterns*
Balalaie et al. [12]	✓		✓			✓
Garriga [48]	✓			✓		
Neri et al. [90]	✓			✓		✓
Osses et al. [94]	✓			✓		✓
Soldani et al. [111]	✓				✓	
Taibi et al. [115]	✓		✓		✓	✓
Tighilt et al. [119]			✓			✓
Zimmermann [142]		✓		✓		

Table 3.1: Overview of the related work on Microservices concepts.

3.2 MODELLING APPROACHES

The following section covers modelling approaches in Microservices as well as the use of logical and ontological approaches in modelling microservices and distributed systems.

3.2.1 MODELLING MICROSERVICES ARCHITECTURES

Sorgalla et al. [112, 101] have proposed the *Language Ecosystem for Modelling Microservices Architecture* (LEMMA) [112], which is a collection of modelling languages shaped to address particular Microservices development aspects in a model-driven approach. Their aim was to support a "model-driven workflow of microservice development in distributed, DevOps-based teams" [101] and to facilitate model transformations.

Overall, LEMMA could represent various viewpoints [112], as each modelling language relates to different DevOps viewpoints [101], thus supporting domain, service and operational aspects. Their approach offers an import mechanism that links different viewpoints. This mechanism is similar to importing in programming languages. By linking modelling languages, they "enabled model reuse across viewpoints and subsequent composition of comprehensive architectural models of MSA-based software systems" [101].

The three modelling languages in the Sorgalla et al.'s [112] approach are:

- The Domain Data Modelling Language, which is applied to represent domain-specific and domain-driven design concepts.
- The Service Modelling Language, which is applied to model microservices' services interactions.
- The Operation Modelling Language, which is applied to define the microservices' deployment infrastructure.

Sorgalla et al.'s [112] suggestions are similar in direction to our research, as they aim to provide mechanisms allowing the modelling of microservices-based systems while considering different viewpoints. However, there are differences in their approach as they have chosen separated models to represent different viewpoints, whereas in our approach, specific viewpoint models are subsets of a multi-viewpoint unique model. Another key difference to be noted concerns the technology stack. For this approach, they relied on modelling languages while we relied on an Ontology-driven conceptual modelling approach.

Mendonça et al. [81] have proposed using feature models to describe microservices-based systems. They have suggested an automatic approach based on *Multi-Objective Evolutionary Algorithms* (MOEAs) to extract feature models from existing microservices systems. Accordingly, practitioners could then analyze those feature models to establish possible microservices reuse, as-is or after some customization. This work is related to the use of Software product line (SPL) approaches in modelling MSAs with the aim of "leverage the benefits of microservices-based systems, mainly related to interoperability" [81].

Applying an SPL approach in MSA allows for the quick creation of new microservices-based systems. However, in this approach, only functional aspects are taken into account. As seen previously, it is suitable when modelling MSAs to have a multi-viewpoint perspective, because multiple factors influence those architectures. Nevertheless, Mendonça et al. [81] provide valuable insights for enhancing reuse in microservices-based systems.

The work of Benni et al. [18] could be considered as a continuity of the work from Mendonça et al. [81], as they have addressed the interchangeability challenge to enhance microservices reuse in an MSA-SPL perspective. Their approach differs in that they have considered implementation and dependency aspects when establishing interchangeability, whereas Mendonça et al. limited their reuse strategy to functional aspects.

Furthermore, Benni et al. highlighted the limitation of coarse-grained descriptions and the difficulties of compelling fine-grained descriptions to better identify implemented features. In fact, they pointed out the interest of identifying the end-points exposing them along with the features. However, in a Microservices context where multiple microservices versions might implement different features, fine-grained identification could be complex to implement. Thus, their approach used a feature-to-service method, which links microservices to each feature they implement in a coarse-grained way. In addition, they proposed a set of four factors to establish whether a microservice is interchangeable. These factors are: *Sharing of feature sets*, *interaction compatibility*, *communication compatibility*, and *no coevolution*.

Even if MSA advocates designing microservices with low coupling in a perspective of high independence and sharing nothing, it does not prevent hidden dependencies associated with development practices and organizational aspects. Indeed, microservices can evolve closely, which Benni et al. designated as *coevolution*. Thus, it is critical for establishing interchangeability to be aware of these dependencies.

Although work from Benni et al. [18] and Mendonça et al. [81] both show some similarities in aspect to what our work has produced, their work differs in the microservices aspects represented. In our approach, we are proposing a holistic view of microservices-based systems, from requirements to deployment. We considered that such a view is a requirement to accurately model microservices systems, in order to establish similarity which would be then followed by interoperability and reuse.

3.2.2 LOGICAL AND ONTOLOGICAL APPROACHES IN DISTRIBUTED SYSTEMS MODELLING

In 1987, Yau et al. [138] proposed an approach based on directed graphs (DG) and first order logics to formalize software component interconnection to enhance validity and integrity check. They proposed addressing the problem of modelling software component interaction by applying an approach based on artificial intelligence. Their approach described abstract and implementation aspects in the same model.

They defined a vocabulary based on first-order logic, named the DG Vocabulary, and which they applied to interpret the interconnection information of software components. Thus, they formalized this interpretation as axioms. The DG Axioms are “a set of logic clauses.” Those axioms compose the directed graph representing the software component interconnection and are used with “automated reasoning techniques to facilitate validity and integrity checking” of the component interconnection model.

Their approach shows parallels to what we could accomplish with an ontological representation and some of the elements of their approach can be related to OWL-DL components. Indeed, the DG Vocabulary could be considered as a TBox, the DG Axioms as an ABox, and the automatic reasoning based on the axioms as the inference engine.

Thirty years later, Yuan [140] proposed the *Semantic Architecture Workbench* (SAW), an approach which represented software architectures for sharing and reasoning based on OWL2 ontologies. This approach was thought to provide capabilities for architecture discovery, style validation, consistency check, and visualization. Yuan argued that one could apply this workbench at both design and execution time.

Furthermore, SAW stores the ontology axioms in a triple store, uses “a set of semantic processing tools to manipulate, infer, and render ontological knowledge,” and relies for visualization on Protégé, a well known ontological engineering tool. It also has the potential to expose the formalized architecture via API end-points, allowing specialized external tools to access knowledge to build a *Software Architecture Instrumentation* (SAIN) [140] environment that can extend ontological representation by adding capabilities such as ontology completion.

Our approach and what was proposed from both the works of Yau et al. [138] and Yuan [140] intend to formalize software models using logics and multi-level information in the same model. In particular, both Yuan’s approach and ours share the same vision of using ontologies as a tool for capturing, managing, and sharing knowledge in a software engineering context, in addition to using the same ontology language. There are several differences to note, however, including that we shaped our approach to be applied in an intensive component and distributed context whereas the approach from Yuan is generic and was not experimented within such a context. As well, the approach of Yau et al. was applied in monolithic systems.

Table 3.2 summarizes the works reviewed in this section.

Works	Modelling Languages	SPL Approaches	Logical Approaches
Benni et al. [18]		✓	
Mendonça et al. [81]		✓	
Sorgalla et al.[112, 101]	✓		
Yau et al. [138]			✓
Yuan [140]			✓

Table 3.2: Overview of the related work on Microservices modelling and logical modelling approaches.

3.3 CONCLUSION

Despite having different problems, the main ideas of these works can be explored for the context of microservices concepts and functionalities representation and effective microservices reuse. Thus, our approach considered all the technological and theoretical insights proposed by the approaches discussed in this chapter. The holistic view of microservices-based systems architectures proposed in this thesis could provide modellers with the necessary information to handle modelling and analysis challenges in Microservices Architecture.

CHAPTER 4

METHODOLOGY

In order to fully answer the research questions presented in Section 1.1, we first began by searching to establish whether reusing an existing ontology of the Microservices domain was a possibility. This first step would then determine the methodology applied in our work.

An existing ontology which could potentially meet our goals and be reused is one that should contain terms that allow the description of MSA's theoretical, business, technological, and organizational aspects. In order to identify possible ontology candidates, we relied on related work, previously presented in this thesis, and on known ontologies' repositories as suggested in the work of McDaniel et al. [80]. We thus queried the following repositories: Linked Open Vocabularies[1], OntoHub[2], ROMULUS[3], and COLORE[4].

What we determined was that the existing ontologies did not meet our needs, and although we found a potential ontology called the *Unified Ontology of Cloud Computing*, which was proposed in the work of Youseff et al. [139], it is no longer available. Apart from that ontology, none of the researched ontologies from the above repositories were adequate for our requirements and after a thorough investigation, no ontology describing the Microservices' domain

[1]https://lov.linkeddata.es/
[2]https://ontohub.org/
[3]http://www.thezfiles.co.za/ROMULUS/
[4]https://github.com/gruninger/colore/tree/master/ontologies

was found in these repositories. In turn, the existing ontologies related to the software engineering field were foundational, as *DOLCE* [46], or core, as the *Core Software Ontology* [93] with limited scope.

In the absence of such an existing ontology, using different domain ontologies that adequately met these criteria would clearly result in a patchwork ontology, and thus lead to an unnecessary time-consuming reengineering process, that we wished to avoid. Using different domain ontologies, would have required choosing functional pieces, customizing certain terms to prevent ambiguity [63, 27], and then creating the missing pieces. Having taking all of this into consideration, we determined that this approach was simply not adequate and developing a new ontology was clearly justified [47].

This chapter presents the methodology used to build and evaluate the proposed ontology.

4.1 ONTOLOGY DEVELOPMENT METHODOLOGY

The general methodology applied is based upon the methodology for building ontologies in the Semantic Web put forth by Charlet et al. [27]. This methodology is composed of four steps: Data gathering and analysis, semantic normalization, ontological commitment, and operationalization.

As this methodology was created with a general use purpose in mind, it did not provide a detailed framework that could be applied for each step. Consequently, we applied other methodologies, methods, and techniques to complete the different steps and address specific challenges and needs. In the following section, we have detailed each step of our methodology.

4.1.1 DATA GATHERING AND ANALYSIS

This step consists of gathering consistent data to describe the domain of interest. The outcome is a corpus of accurate data containing a consensual definition of the terms related to the domain. We completed this step in conjunction with a review of the related work.

Thus, we queried articles indexed in Google Scholar, Science Direct and Wiley Online, using the following search strings: *Microservice[s] taxonomy[ies]", "microservice[s] architecture[s] taxonomy[ies].* These queries showed a significant number of results, and in most cases, two terms were not directly related. As a result, we chose to rely on a recent literature review on this topic [64] to identify the appropriate core taxonomy.

At the end of this analysis, we selected the taxonomy of the Microservices Architecture proposed by Garriga [48], which appeared to achieve a certain consensus [64], as our core taxonomy. In addition, we selected the taxonomy of MSA gains and pains proposed in the work of Soldani et al. [111] and the classification of Microservice tenets proposed by Zimmermann [142] as a data set for a comparison to Garriga's taxonomy.

To complete our analysis, we selected two other recent works, both published after Garriga's taxonomy paper which were from Taibi et al. [115] and from Neri et al. [90], and with a Microservices architectural anti-patterns scope. Due to the fact that Microservices Architecture has yet to be considered mature enough and is in continuous evolution, it seemed essential to include those works in this activity in order to identify likely evolutions leading to discordant concepts. One example would be a consensual pattern which might evolve into a consensual anti-pattern.

We then selected a study that used these principles, patterns and anti-patterns for creating a catalogue of MSA migration patterns [12], concretely applying these concepts because this work may have revealed gaps in the existing taxonomies and real-world applications.

4.1.2 SEMANTIC NORMALIZATION

In this step, we had to establish a consensual terminology that was based on the referenced corpus. We accomplished this by identifying standard and discordant definitions and then made decisions about which one to consider or to discard. We applied two complementary methods to accomplish this task: OntoClean [60] and WHAT-knowledge [74]. In the following paragraphs, we have described each application.

OntoClean

OntoClean provides a formal basis to explain ontological choices. It also allows validation during modelling by exposing misuses, inconsistent and inappropriate choices, and helps to define the modeller's purposes and terms scope. OntoClean focus on the following four concerns:

- Essence—the stability of the concept in time.
- Identity—the differentiation of concepts.
- Unity—a concept is part of other concepts.
- Dependence—a concept needs other concepts to exist.

WHAT-knowledge

WHAT-knowledge is a knowledge classification approach put forth by Kudryavtsev et al. [74] and is based on simple questions which allowed us to understand and identify terms and their relations. We used the following competency questions proposed by these authors to guide us in our data exploration.

- What is it?

- What is the structure of the entity?
- What does the term mean?
- What is the entity part of?
- What are the parts of the entity?
- What are the subclasses or instances of the entity?
- What is the type of the entity?
- What is the relationship between entities?

Those competency questions supported us in identifying essence, identity, unity and dependency concepts, and broader, narrower and related terms.

4.1.3 ONTOLOGICAL COMMITMENT

In this step, we investigated the existing relationship between the different terms, other than subsumption. We highlighted the fact that in the methodology proposed by Charlet et al. [27], commitment did not mean the agreement around an ontology but rather the coherent construction of logical relationships based on the unveiled terms from the reference corpus.

We thus precisely named the relations previously classified as “related to” and determined, when it was needed, the cardinality of these relations. Furthermore, we referred to standard dictionaries of the English language to define these relations [135, 14].

4.1.4 OPERATIONALIZATION

This step aimed to formalize the identified terms and their relations in an ontological language. The first task in this step was language selection and accordingly, we selected a mature, standardized language, which is expressive enough to design the domain while at the same time having tool support for conception and maintenance.

We chose to formalize our ontology using the OWL2-DL, which uses the Description Logic *SROIQ*. It allowed us to express concepts and relations as subsumption, equivalence, disjunction, negation, existence, nominal and value restrictions. Furthermore, it can qualify cardinalities and represent reflexive, symmetric, transitive, irreflexive relations, and constructs [67]. OWL2 DL applies the open-world assumption and is decidable [73].

Secondly, we chose to rely on an *Ontology Design Pattern* (ODP) [47] to support the translation of the resulting taxonomy from previous steps into an ontology. Using an ODP for building ontologies is a suitable method for designing ontology. It can be compared with the design patterns used in Software Engineering because of their shared objectives and benefits. An ODP aims to provide solutions to known ontological design issues and enhance ontology mapping, alignment and reuse. Knowing this, we relied on the ODP-based method proposed in the work of Villazon et al. [132], and which these authors have named the *Pattern-Based Method for Re-engineering Non-Ontological Resources*.

This ontological development method comprises three activities:

1. Non-Ontological Resources Re-Engineering—data gathering and analysis.

2. Non-Ontological Resources Transformation—data transformation in an ontological component (e.g., TBox, ABox).

3. Ontology Forward Engineering—fine-tuning relations between terms, existential constraints and properties.

As the previous steps were completed in the first activity, we applied only the last two activities to support the tasks in this step.

Non-Ontological Resources Transformation

We aimed to transform our Non-Ontological Resources (NOR) into an ontology by applying an ODP in this activity. Following this method, we first identified the TBox transformation as the desired transformation approach. Later, we identified our data model as a NOR composite formed by a conceptual map and a simple text list.

We then explored the pattern catalogue [117] to find patterns related to a TBox transformation based on conceptual schema under the form of conceptual maps and lists. Although we did not find a pattern matching our context, we were able to identify a family of patterns that were related to it. Based on this, we decided to use the *Pattern for Re-engineering a Classification Scheme following the Adjacency List data model into an ontology Schema (PR-NOR-CLTX-02)* [117, 131] because such an adjacency list is a classic way of representing a graph [31]. It allowed us to work easily with our NOR.

At this point, we limited relationships between entities to those suggested by the pattern: *SubClassOf* and *relatedClassOf*.

Ontology Forward Engineering

As we used OWL2-DL in the previous activity, some levels of the Ontology Forward Engineering had already been completed. We modelled properties and rules within this activity and refined relationships relying on OntoClean methodology to define them accurately. Notably, we labelled properties as rigid (always essential), non-rigid (sometimes essential) or antirigid (never essential) [60].

Classifying properties using OntoClean helped us to identify *qualified cardinality*, *disjointness* and the *straightness* of the relations among concepts, which were then formalized in OWL2-DL.

4.2 EVALUATION METHODOLOGY

Evaluating an ontology means to verify that it was built correctly and then validate that it adequately describes the domain of interest [59]. Verification is often called *Ontology Quality* and is focused on the structural aspects and logical consistency of the ontology. Validation is often called *Ontology Correctness* and is focused on the "distance between the real world" and the modelled ontology. As highlighted by Hlomani et al. [66], these approaches are complementary and not exclusive. Various approaches, supported by various methods, can be used to evaluate an ontology including, *gold standard-based*, *application or task-based*, *user-based*, or *data-based* [66].

The *gold standard-based* approach relies on a proven ontology of the domain, which is used to compare it to the one under validation. To the best of our knowledge, such an ontology of the Microservices' domain is not available, and using this approach to evaluate our ontology would therefore not be possible.

Concerning the user-based approach, it relies on the user experience in which a number of users use the ontology. Although, various users would apply the proposed ontology, a higher level of bias could potentially arise since validation is based on user experience [66]. This means that evaluating our ontology using the user-based approach would not be effective.

Lastly, the data-based approach needs a non-ontological equivalent model to be used as a benchmark. The ontology we propose in this thesis was built based on the reference corpus discussed in Chapter 3, which is used as the benchmark in our evaluation process. Not to mention, the application or task-based approach validates an ontology's effectiveness in a real application, as a use-case scenario [42, 66], which is a practical approach to validate correctness concerning the real world.

The methodology applied to evaluate our ontology is consequently based on evaluation aspects from those proposed in the work of Hlomani et al. [66], and takes into consideration the application-based and the data-based approaches. We devised this evaluation methodology in a two-step process which is the following: Quality and correctness, with each process using adequate methods, that we have expanded on below.

4.2.1 QUALITY

In the quality evaluation process, we verified the ontology structure through consistency and computational efficiency aspects. By verifying consistency, we ensured that the ontology did not include contradictions and is logically correct. Simultaneously, computational efficiency relates to the capacity of a reasoner to infer the ontology. We took an automatic validation method supported by recognized technologies in the domain of ontology verification for assessing them.

Consistency was verified using OOPS! [99][5] a web-based application that automatically detects common structural errors and inconsistencies in an ontology, and the HermiT OWL Reasoner [51] v.1.4.3.456. Computational efficiency was verified using the Pellet [110] reasoner and SPARQL [30] queries when evaluating the correctness of the ontology.

4.2.2 CORRECTNESS

In the correctness evaluation process, we verified the ontology's correctness through completeness and coverage. By verifying completeness, we ensured that "questions the ontology should be able to answer, can be answered" [66]. By verifying coverage, we ensured that the ontology can produce a correct representation of an aspect of the modelled domain. In other words, we validate that the ontology can be applied to build assertions on the domain.

For both processes, we referred to the competency evaluation method proposed by Fox et

[5]http://oops.linkeddata.es

al. [42] for the requirements' assessment. The key point of this method is the proven ontology competency. According to this approach, an ontology is considered valid if it satisfies requirements specified by competency questions. Using relevant competency questions is suggested to evaluate an ontology, as the questions specify a set of queries under evaluation, the ontology should be able to answer.

Based on this method, we formalized these competency questions using the SPARQL query language, and assessed them in the data-based and application-based approaches.

Use Case Choice

For evaluating this ontology taking into account the application-based approach, we relied on a use-case composed of different microservices-based systems. We chose these systems based on the following criteria:

1. Being a native microservices architecture, from the same business field.
2. Being composed of at least five microservices.
3. Sharing at least one functionality.
4. Being developed in different programming languages.
5. Using at least one platform-provided service.
6. Having a detailed description of the target infrastructure.
7. Having a detailed description and full access to the documentation and source code.
8. Being used as a baseline in other pair reviewed researches.

4.3 TOOLING

For the ontology development, we relied on Protégé [87], version 5.5, which is a mature ontological open-source platform and is also widely used. It provides an interactive graphical interface for ontology design, visualization, and manipulation, and had the ability to allow loading, designing and editing of OWL2 ontologies.

We relied on Stardog [122], version 7.5.1, as a triple store to store the ontology for ontology evaluation and exploration. Our choice of using Stardog as a triple store is based on the fact that it is a well-known commercially triple store solution, which performs as well as its open-source counterparts [105, 10]. As the performance of triple stores was out of our scope, we relied on a solution which proposed advanced features, such as advanced visualization and machine learning capabilities, yet was easy to use. In addition, Stardog allowed us to deploy an instance of a Stardog triple store in a Docker container which enhances its portability and allowed us to obtain the content of the knowledge base and the results of the SPARQL queries in a graphical form. Using Stardog Studio [121], version 1.31.0, allowed us to build and execute different SPARQL queries. For the purpose of this thesis, we used Stardog under academic licensing.

CHAPTER 5

THE ONTOLOGY OF MICROSERVICES ARCHITECTURE CONCEPTS—OMSAC

This chapter presents the Ontology of Microservices Architecture Concepts (OMSAC). We begin by presenting the intent of OMSAC, followed by a detailed description of the structure of OMSAC, its classes and properties. Lastly, we discuss modelling challenges faced when modelling OMSAC.

5.1 INTENT

OMSAC aims to support developers in exploring, understanding and using Microservices Architecture concepts to build microservices-based systems architectures. Hence, the domain of the ontology is the Microservices Architecture domain. In this thesis, we are limiting the ontology to cover only concepts and principles, recorded as MSA principles, patterns and anti-patterns by the works discussed in Chapter 3.

The OMSAC's terminological component (TBox) provided the terms needed to describe both MSA-related concepts and MSA artifacts (microservices and microservices-based architectures). This component provided descriptions of various aspects of the MSA life cycle for such things as conception, design, operations, technology, organization, migration and maintenance. The assertion component (ABox) varies depending on the application context and can be used for analyzing MSA-based architectures.

5.2 DESCRIPTION

This section presents the OMSAC structure and the concepts which compose it.

5.2.1 STRUCTURE

OMSAC concepts are, in fact, OWL classes and properties. These concepts are organized in six high-level classes which include: *MicroservicesArchitectureConcepts*, *Microservice-sArchitecture*, *Microservice*, *CrosscuttingConcepts*, *CrosscuttingConcerns* and *AntiPatterns*. Apart of MicroservicesArchitecture and Microservices, all classes have various subclasses. Figure 5.2.1 shows a graphical representation of the OMSAC classes with the corresponding meaning described in Table 5.1.

Class	**Description**
APIGateway	An API gateway is a routing service responsible for managing requests, load balancing, monitoring, authentication, and call compositions through microservices, according to clients requests and microservices end-points [32].
Aggregator	An aggregator interacts with all the other microservices to derive the application functionalities. It can be passive, only invoking the appropriate microservices and displaying the results, or active, applying business logic to the process in addition to the passive tasks [70].
AntiPatterns	The counterpart of patterns. The difference between patterns and anti-patterns is that anti-patterns provide an inadequate solution to a problem [24].
ApplicationLayer	Protocols used in inter application communications.
ArchitecturalConcerns	Design concerns related to the architectural aspects of the microservices-based system.
ArchitecturalPatterns	Patterns used in architectural tasks.
Availability	It implies handling service-level and low-level failures that require persistence and recovery techniques [48].

Table 5.1 continued from previous page

Class	Description
BoundedContext	Related functionalities are combined into a single business capability implemented by a microservice.
Cache	Not persistent and volatile memory mechanism used to hold instructions and data temporarily. It intends to reduce the variance when accessing a high-speed storage processor and main memory [26].
Choreography	Inform each part of the system of its job, and let it work out the details, like dancers all finding their way and reacting to others around them in a ballet [91].
CircuitBreaker	Mechanism acting the same way of an electric circuit breaker and used to handle microservices calling failures during a service's outage [91].
Client	Microservice's consumer.
Cloud	On-demand and adaptive IT infrastructure shared by various end-users [78]
CloudNative	Applications and systems designed to be deployed and exploited in a Cloud Computing infrastructure [32].
Communication	Anti-patterns related to the communication between microservices.
Composition	Concerns how to combine microservices following business process [91].
ConcernsIssue	Wrong separation of concerns [115].
ConfigServer	Service providing configuration to microservices in a microservices architecture [32].
Configuration	The particular hardware elements and their interconnection in a computer system for a particular period of operation [26].
Container	A container is a computing context that uses functionality from a host that it's running on [32].
ContinuousDelivery	Continuous delivery is a deployment practice supported by automation that allows building software to be released to production at any time [76].
ContinuousIntegration	Continuous integration is the practice of integrating work frequently. It is supported by automation to reduce integration errors and quickly develop cohesive software [76]

Table 5.1 continued from previous page

Class	**Description**
ControlLoops	Concerns the different degrees of self-adaptation of a microservice, it includes controls as monitoring, analysis, planning and execution, used to automatically adjust the microservice behaviour [48].
CouplingIssue	A cyclic chain of calls between microservices and direct communication between microservices [115].
CrossCuttingConcepts	General software engineering concepts related to various aspects of microservices systems.
CrossCuttingConcerns	Mostly regard Quality of Service (QoS) aspects that have to be tracked within the microservices' lifecycle, supported by the infrastructure through specific artifacts and independent of individual microservices [48].
DataExchange	Concerns related to the communication between microservices and microservices and supporting services [48].
DataStorage	Concerns related to how design and manage data storage in a distributed system, respecting isolation and share nothing principles [48, 91, 32].
Database	The database is a software system providing capabilities for the organization and management of a body of information [26].
Decentralized	No centralized component is responsible for configuration, management, or policy control for the system as a whole [103], as well decision-making and control are transferred to the teams that own the services themselves [91].
Decomposition	Rearchitecting an application to a set of services [12].
Deployment	Facilitating an application's deployment process and removing deployment anomalies [12]. Deployment encompasses how and where services are actually hosted and deployed [48].
DesignConcerns	It implies thinking about the boundaries of microservices that will maximize their upsides and avoid some of the potential downsides, focused on loose coupling and high cohesion [48].
DesignPattern	Patterns used during design.
Discovery	Concern of exposing and retrieving a microservice [91].
DiscoveryIssues	Anti-patterns related to microservices exposition and retrieve.

Table 5.1 continued from previous page

Class	Description
Distributed	A system in which several independent interconnected logical and physical components can cooperate to achieve a common goal [103].
DynamicLocation	Dynamic allocation is made dynamically while the system is running, rather than statically at first initiating the system [26].
Dynamicity	Enabling an application to change in runtime without affecting its end users [12].
Elasticity	Refers to the capability to rapidly adjust the overall capacity of the platform where the system is deployed [48].
EventBus	A service which broadcasts and filters event messages.
Evolutionary	Microservices can evolve easily with minimal changes to the existing configuration of the system [70].
Failure	Abrupt cessation of normal functioning [135].
FaultTolerance	The system remains at least partially available and functional even if some of its nodes, applications, or communication links fail or misbehave [103].
Feature	Visible aspects of the software system [23]
FineGrained	Composed of small microservices responding to a well delimited functionality [91].
Function	The purpose of a microservice or service.
Functional	Requirement related to functional aspects of a system.
Functionality	The set of capabilities associated with computer software [135].
Granularity	Relates to the complexity of the functionality implemented by a microservice, often represented by the number of requirements implemented by functionality [91].
Implementation	It defines concerns related to the complexity of the system due to the huge number of microservices running asynchronously in a distributed computer network [48], as well as to concerns related to the technology used for developing and operating microservices [142, 48].
Independency	Represents loose coupling and high cohesion by asserting that each microservice is operationally independent of others, and communication is done through their published interfaces [79]

Table 5.1 continued from previous page

Class	Description
IndependentDeployment	Deployment processes which avoid dependency between microservices [91].
IndependentEvolution	The evolution of a microservice is completely decoupled from other microservices [91].
IndependentManagement	Microservices are managed in isolation, and often by different teams [32].
Infrastructure	Physical and virtual resources supporting the Microservices Architecture [48].
Instance	A running copy of a microservice [26].
Interaction	It is when an component interacts with another component through a set of input devices to achieve a task [26].
InteractionModel	Concerns related to the communication flow between microservices and supporting services [48].
Internal	Anti-patterns impacting an individual microservice [115].
Isolation	Each component is autonomous and communicates with others only via structured message protocols [103]. Also, the state of a microservice component does not affect other microservices [70].
Language	Concerns the languages used from a polyglot perspective. It relates to the choice of the correct language for the right task [48].
LooseCoupling	Microservice components depend less on each other to avoid impacts when doing modifications [70].
Management	Encompasses the responsive reaction to failures and changing environmental conditions, minimizing human intervention [32].
MessageQueue	Queues used to forward messages asynchronously; they store the message until complete transfer [26].
Messaging	This term gathers the message switching systems used to transfer data under messages of any length [26].
Microservice	A small, autonomous service that works with other microservices, is “focused on doing one thing well” [91] and communicates through lightweight protocols [76].
MicroservicesArchitecture	A microservices architecture is a distributed application where all its modules are microservices [38].

Table 5.1 continued from previous page

Class	**Description**
MicroservicesArchitectureConcepts	Class gathering core concepts of the Microservices Architecture.
MigrationPatterns	Patterns used when migrating to Microservices Architecture [12].
MitigationPatterns	Patterns used to address mitigation challenges.
Modifiability	Increasing the ability to change an application with the least side effects and without affecting its end users [12].
Modularity	The system is decomposed into different modules which enhance development, testing, and understandability [70].
Monitoring	Enabling an application to be monitored in runtime effectively [91, 32]
MonitoringIssue	Lack of usage of monitoring systems, including systems to monitor if a service is alive or if it responds correctly [115].
NoSQL	Databases which do not follow the relation model [26].
NonFunctional	Requirement related to aspects other than functional.
Observable	Capacity to monitoring microservices behaviour in a distributed infrastructure, by aggregating logs and stats [91].
OnPremise	The infrastructure that belongs to the system's owner, in opposition to outsourced infrastructure [78]
OperationalConcerns	Concerns related to how operate the microservices-based system. [12].
OperationalPatterns	Patterns used to handle operational challenges.
Organizational	Anti-patterns related to organizational aspects.
OrganizationalAspects	Aspects related to development and operationalization process. [12].
Others	Technical anti-patterns that are not related to communication and do not impact an individual microservice only [115].
Owner	One to whom microservice's property belongs.
Patterns	The description of a recurrent problem and its appropriate solution [104, 75].
PersistanceIssue	Anti-pattern related to data persistence [115].

Table 5.1 continued from previous page

Class	**Description**
Persistence	The property of data that continues to exist after a process accessing it has finished [26].
Platform	The computer architecture and equipment using a particular operating system [135].
Polyglotness	Enables each microservice developer to choose the best language, platform, or database technologies, independently of the other microservices in the application [70]. This can lead to applying different programming languages and data stores in the same architecture [12].
Principles	A comprehensive and fundamental assumption in the Microservice Architecture domain.
ProgrammingLanguage	High-level language used for developing computer programs [135].
Protocol	An agreement that two or more components utilize to structure their conversations. Protocols may be implemented in both hardware and software, and are often classified in layers according to their communication scope [103].
Provider	One who supplies microservices architectures with technological components, as infrastructure and services.
Relational	A database management system that supports the relational model [26].
Reliability	It refers to a system capable of performing well without halting, according to its requirements, and is fault-tolerant [91, 32].
Requirement	Essential elements a system should have in order to meet its aims [135].
ResourceEfficiency	Decreasing the amount of resources needed for an application's deployment [12].
Role	The part played by a stakeholder in a microservices architecture in an organizational perspective.
Runtime	Concerns related to the time when a microservice is running.
Scalability	Capability to rapidly adjust the platform's overall capacity by adding or removing resources and minimizing human intervention [48].
Security	Concerns related to security challenges.
Server	A device storing a system on a network that provides a service to other systems connected to the network [26].

Table 5.1 continued from previous page

Class	**Description**
ServerFarm	A collection of networked processors providing large-scale computing services [26].
Service	Software supporting microservices architecture [48].
ServiceDiscovery	Allows clients to make requests to a dynamically changing and extensive set of transient service instances [48].
ServiceRegistry	Service that stores microservice instances' addresses and allows invoking microservices. Each microservice registers itself during its initiation [12].
SharedPersistance	Two microservices access and manage the same database [115].
SharingNothing	Microservices does not share resources which could avoid their independence [32, 91].
SingleResponsibility	Microservices have only a responsibility [142, 91].
SizeIssue	Several business processes implemented in the same service, leading to a monolithic system [115].
Stateless	States in a microservices architecture are stored outside the applications in dedicated data services [32].
Storage	Relates to devices that can retain data for subsequent retrieval [26].
TeamOrientedIssue	Anti-patterns related to the team's dynamics [115].
Technical	Anti-patterns related to technical aspects in the microservices' development and operationalization [115].
Technology	Gather concepts related to the technology stack used by microservices-based architectures.
TechnologyOrientedIssue	Anti-pattern related to technology and tools used [115].
ToolSupport	Concerns the supporting tools need to develop and operate a microservice-based system. It includes the adequate choice of the kind and number of tools [48].
TransportLayer	Protocols used by the network [26].
Understanding	Perceiving the current situation of an application [12].
Validation	Concerns the quality assessment of microservices at runtime [48].

Table 5.1 continued from previous page

Class	Description
Virtualization	Encompasses the different degrees of platform abstraction, isolation and sharing [48].
Visioning	Deciding on the final situation of an application after the migration [12].
YouBuildItYouRunIt	The team who develops and builds a microservice operates it [38].

Table 5.1: Definition of the classes of the OMSAC's terminological component.

OMSAC contains a set of Object Properties that allows descriptions of different relationships between the various classes. Moreover, those properties implement a set of rules that governs the relations among classes, including the representation of existential conditions, class intersections and unions, which allowed us to represent MSA concepts and artifacts accurately. Table 5.2 shows these properties and their meaning.

Object Property	Definition
addresses	The fact of dealing with a concern.
basedOn	Formed from a principle.
calls	Temporarily transfer control of computer processing to another functionality.
codedIn	The service or microservice is written in a given programming language.
communicatesBy	The protocol-based way a microservices communicates with other microservices and services.
communicatesThrough	Describes when a microservice or service communicates using an intermediate.
composedOf	Formed of components.
has	Hold or maintain as a possession.
composes	Be the components of.
concerns	Relates to.

Table 5.2 continued from previous page

Object Property	**Definition**
configuredBy	Be interrelated so as to fit it for a designated task.
configures	Interrelates system's components and infrastructure so as to fit it for a designated task.
dependsOn	Need the help of other microservice or service to accomplish one task.
deployable	That can be deployed.
deployableAs	How the component can be deployed.
deployableOn	That can be deployed on a target infrastructure.
exploitedBy	Utilized by a microservice.
exploits	Make use of a resource.
exposedBy	Microservice or service rendered visible for the public by an exposition service.
exposes	Renders a microservice or service visible for the public.
implementedBy	Is fulfilled by.
implements	Applies or makes concrete some functionality.
interactsWith	Acts upon one another microservice or service.
metBy	Conformed with exactitude and precision by a functionality.
respondsTo	Fulfills or meets a requirement.
providedBy	Supplied by a provider.
provides	Supplies.
relatesTo	Connect a service or technology with a functionality.
reliesOn	Is supported by a pattern or anti-pattern.
unsuitedFormOf	Not a proper form of.

Table 5.2: Definition of the object properties formalized in the OMSAC's terminological component.

5.3 MODELLING CHALLENGES

In this section, we have reviewed significant challenges we had encountered with modelling OMSAC.

5.3.1 MICROSERVICE AND MICROSERVICES ARCHITECTURE

One challenging concept when modelling OMSAC was the relationship between a Microservice and a Microservices Architecture. In literature, many authors interchange the description of these two words—*Microservice* and *Microservices Architectures*, and this makes defining a distinction between them is neither clear, nor accurate. Often, the same concepts are used to define both. The difference between them is often expressed by the fact that a microservices architecture is a composition of microservices [38].

Consequently, we understood that these concepts likely represent the same concept, thus the same entity. We relied on the OntoClean notion of identity and unity to determine whether they are the same and to avoid ontological inconsistency. By doing so, we determined they are different concepts. We inferred that the term *Microservices Architecture* represents microservices-based architectures which are composed of microservices, and this follows the definition suggested in Dragoni et al. [38].

The “composed of” property, as its inverse “composes,” could suggest that *Microservice* is a subclass of *Microservices Architecture*. Again, we relied on OntoClean and we were able to verify which of its rigid, semi-rigid, and anti-rigid concepts best characterized the relationship between the terms *Microservice* and *Microservices Architecture*.

We determined “composes” as an anti-rigid property, as not all instances of the *Microservice* class have to belong to an instance of the *Microservices Architecture* class, and be “composed of” as a rigid property, as each instance of the *Microservices Architecture* class must at least relate to one instance of the *Microservice* class. According to Guarino et al. [59] "anti-rigid properties cannot subsume rigid properties." Consequently, Microservice cannot be a subclass of Microservices Architecture.

As a result, the accurate relationship between them is then a membership, as an instance of the *Microservice* class can be related to different instances of the *Microservices Architecture* class. This behaviour is similar to the social entities example presented by Guarino et al. [60], in which a person can be a member of various social entities, so can a microservice be related to different microservices architectures, and thus, the membership does not affect their essence.

Therefore, a user can query an OMSAC-based knowledge base to learn if a given microservice is a member of a microservices architecture, or whether a given microservice makes up a microservices architecture.

5.3.2 MICROSERVICE RELATED TO DISCORDANT CONCEPTS

An instance of the *Microservices Architecture* class is composed of one or more microservices, and the whole relationships of each microservice form a coherent set of axioms based on MSA principles. Hence, a microservice can be related to MSA principles throughout different relations. It may lead to linking a microservice to discordant concepts without compromise logical consistency. For instance, an individual labelled as a *Microservice* is related to the *Isolate Deployment* concept and also to the *Shared Persistence* concept; the former is a pattern and the latter an anti-pattern. This behaviour exists in some industrial implementations of MSA [15, 21] and is a challenge when defining a logical structure for describing microservices. As OMSAC intends to describe the Microservice domain as close as possible to the state of the practice, we have chosen to consider this situation as logically consistent.

5.3.3 CONCEPTS ORGANIZATION

The intent of OMSAC goes beyond describing Microservices and Microservices Architecture. The aim of OMSAC is to also allow the discovery of MSA principles. Which means that in the OMSAC context, we needed to instantiate an entity allowing exploration of the principles related to MSA.

The solution was to create the concept of *Microservices Architecture Concepts* that could gather MSA native principles, but without having any direct relation with *Microservice* and *Microservices Architecture* entities. This entity becomes in OMSAC the *MicroservicesArchitectureConcepts* class, which is independent of the *Microservice* and *MicroserviceArchitecture* classes. Therefore, with OMSAC, it is possible to both describe a given microservice composing a microservices architecture and describe concepts of the Microservices Architecture domain.

5.3.4 PATTERNS AND ANTI-PATTERNS

OMSAC was required to permit fast evolution in an immature domain. Thus, the question was, 'What should occur if an MSA pattern became an anti-pattern?' This behaviour exposes the need to OMSAC to allow an instance of the pattern concept to change over time. The rationale above was used to identify the relations between the terms *Microservice*, *Microservices Architecture* and the following concepts and their subclass: Anti-patterns, crosscutting-concerns and crosscutting-principles. Using OMSAC, it was possible to describe a pattern and an anti-pattern with the same set of concepts from cross-cutting principles and concerns, and meet the above requirement.

Appendix A provides a graphical representation of OMSAC containing all classes and relationships.

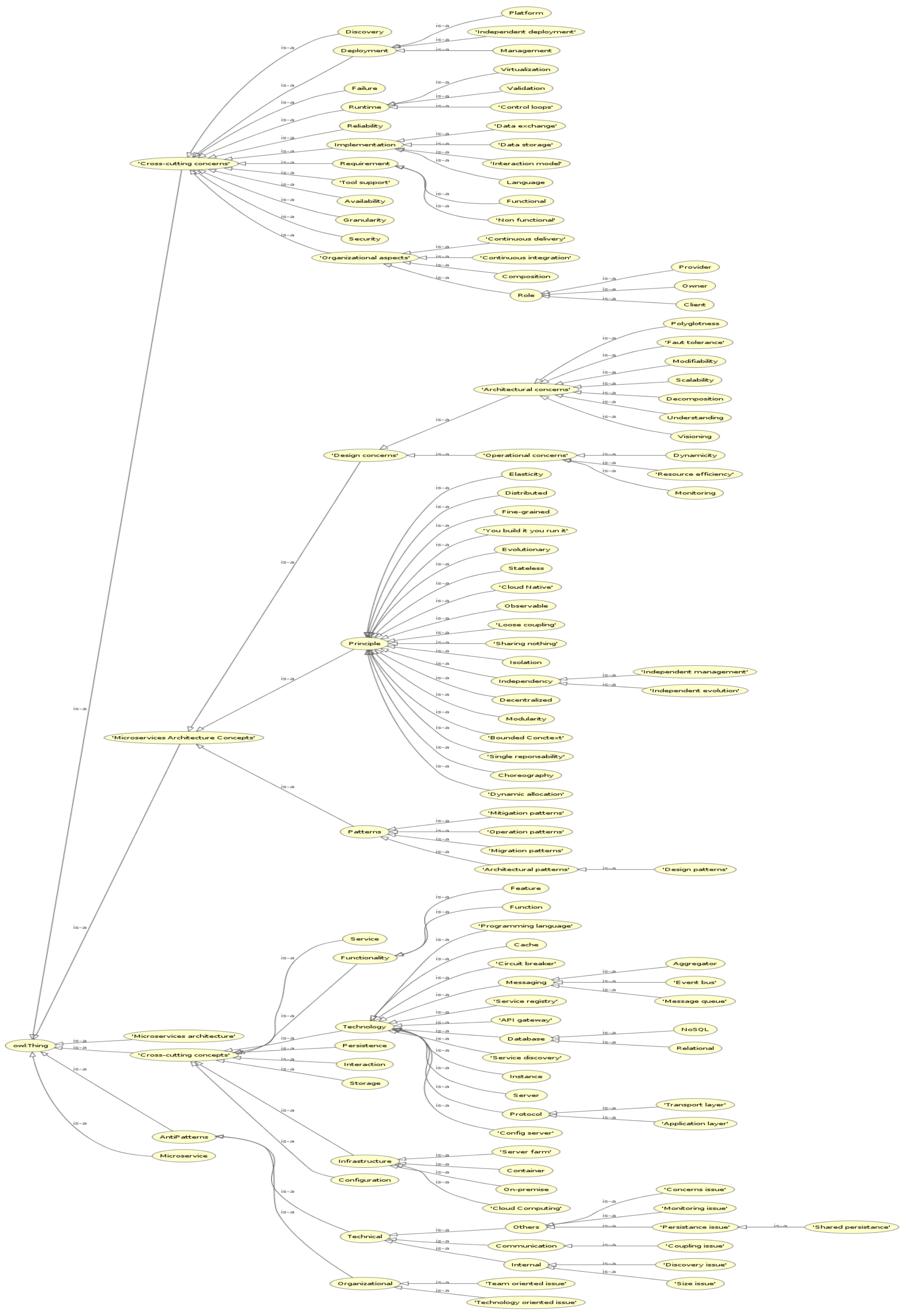


Figure 5.1: OMSAC Classes

CHAPTER 6

USE CASE—MODELLING AND ANALYSIS OF MSA BASED SOFTWARE ARCHITECTURES AND SIMILARITY METRICS

This chapter presents an application of OMSAC in modelling and analyzing microservices-based systems. Our intention was to apply OMSAC to create ontological-based models that describe microservices-based architectures to allow their representation in a multi-viewpoint form. By modelling microservices architectures using an ontology, we expected to be able to address such challenges as identification of microservices functionalities, microservices discovery, interconnections and dependencies, and in addition, other challenges such as microservice reuse, interoperability and interchangeability.

As well, in this chapter, we have detailed the modelling process and the analysis approaches we implemented. As this use case is part of our evaluation process, the findings are discussed in Chapter 7.

6.1 DESCRIPTION

This following section introduces the use case, which will be used throughout this chapter as a running example. It is based on three microservices-based systems: *Hipster Shop*[1],

[1]https://github.com/jacobkrueger/SPLC2020-Microservices-Challenge/blob/GoogleCloudPlatform/microservices-demo/README.md

eShopOnContainers[2] and *Vert.x*[3]. We have modelled them using a set of OMSAC concepts which are presented in Figure 6.1.

The systems we chose are native web-based microservices architectures from the shopping domain. They have characteristics that meet the necessary requirements for this evaluation process. In fact, these systems are composed of microservices that share features as well as meet similar user requirements. Their architecture is composed of between seven and ten microservices, developed in different programming languages (C#, Go, Java, Javascript, Python), and use platform-provided services.

These systems come from a set of six microservice-based systems proposed by Assunção et al. [9] as a baseline to assess variability in microservices architectures. Their source code and documentation are accessible and easy to examine, and they have been explored and analyzed in previous works. These characteristics meet the requirements described in our methodology to qualify as a relevant use case for OMSAC evaluation.

We accomplished this use case in two phases. In the first phase, we modelled the three systems presented above. In the second phase, we analyzed them by applying different techniques and measurements to explore different stakeholders' viewpoints, to identify similarities between the microservices, and to discover potentially interchangeable microservices based on functional and technical aspects.

[2]https://github.com/jacobkrueger/SPLC2020-Microservices-Challenge/blob/dotnet-architecture/eShopOnContainers/README.md

[3]https://github.com/jacobkrueger/SPLC2020-Microservices-Challenge/blob/sczyh30/vertx-blueprint-microservice/README.md

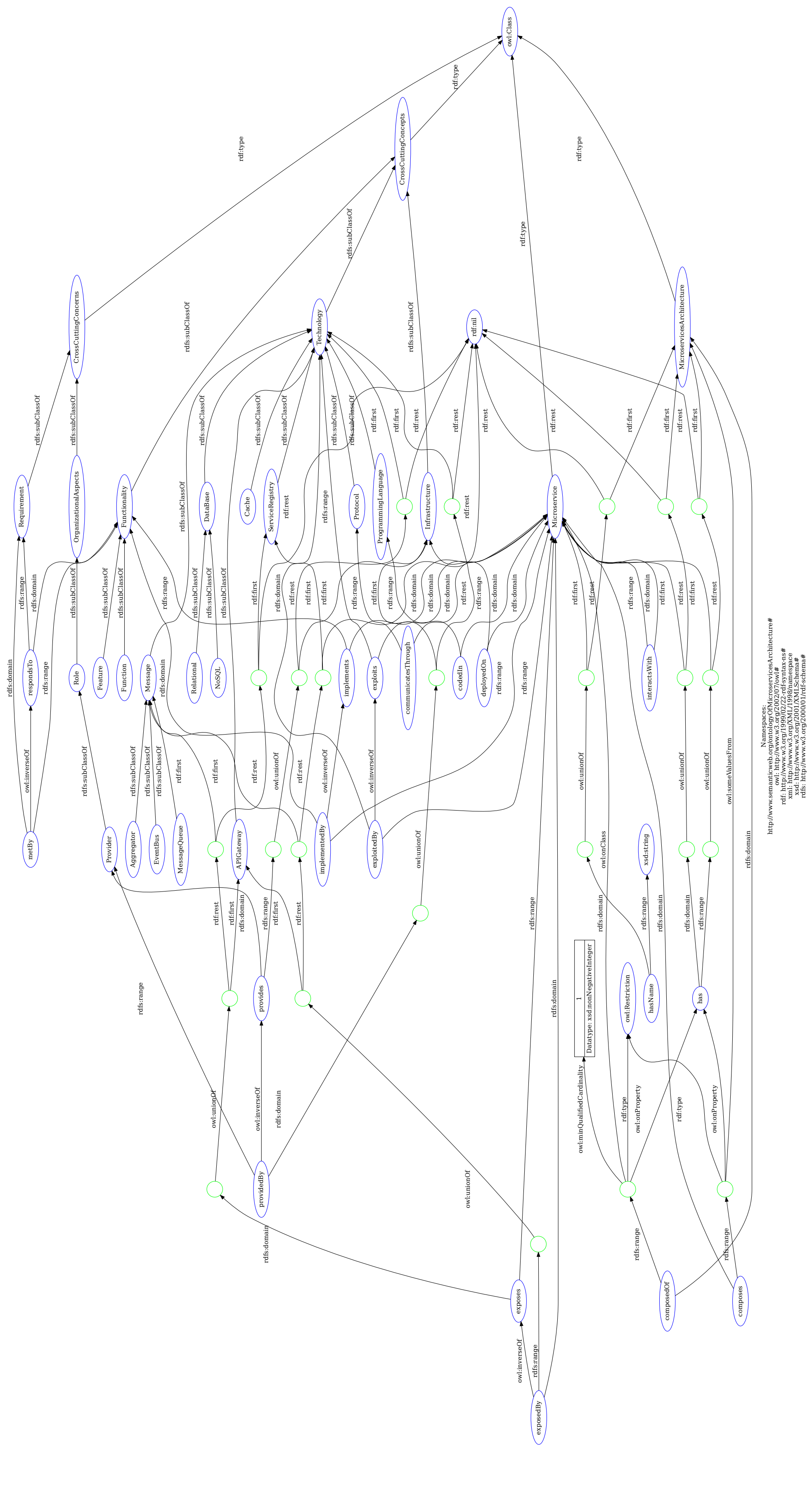


Figure 6.1: OMSAC subset used in this use case.

The outcome of this use case was a knowledge base containing models representing functional, technological and operational aspects of these three systems. This knowledge base can be used for microservices exploration and to measure similarity among microservices. In this use case, we have focused on these two functionalities and applied three approaches for measuring similarity which were machine learning, EdgeSim metric, and a manual-informal analysis made by an expert.

6.2 MODELLING

We achieved this phase following two steps, of identification and translation. The identification step allowed identifying the different aspects related to each microservice that was needed to model these systems. In the translation step, we modelled these microservices-based systems using the retrieved information and the OMSAC TBox.

6.2.1 IDENTIFICATION

We first identified the necessary aspects for modelling these systems in order to achieve our goals and in doing so, we established the following information that was needed to extract from the data. This is listed below:

1. The name of the microservices composing each system.
2. The business requirements each microservice responds to.
3. The features implemented by each microservice.
4. The services used by each microservice.
5. The communication protocols used by each microservice.
6. The operational dependencies between the microservices.

7. The programming language each microservice is coded in.

8. The coevolution among microservices.

We have primarily relied on the results from Benni et al. [18] and Mendonça et al. [81]. Those works have completed an in-depth analysis of the three systems in order to extract functional and technical aspects of each microservice composing them. Despite the detailed information provided by these works, we have identified a lack of operational aspects while modelling these systems with OMSAC.

Consequently, we explored the files related to the deployment processes of these systems from their source code repositories in order to extract the deployment dependencies and the platform-provided services that they use. We mainly analyzed the *Dockerfiles*. In these files, it was possible to find information about service creation, platform configuration (e.g., port exposition, instances number, management of environment and systems variables), and we were able to unveil deployment dependencies (e.g., deployment sequence and services and applications requirements).

Exploring these *Dockerfiles* allowed us to expose and understand the existing dependencies among microservices. We found explicit dependencies that can be described as one microservice that depends on the successful deployment of another one, and implicit dependencies which can be described as microservices sharing the same instance of a service.

These two exhibited dependencies, as well as the unveiled deployment aspects that previous works had not analyzed need to be considered when modelling microservices because these aspects are paramount when analyzing microservices for interoperability or reuse. Table 6.1 provides a sample of the data extracted for the microservice *Basket Microservice* from the *"eShopOnContainers"* system. All the data extracted from the three systems can be found in the Appendix B.

Features	Interactions	Coevolution	Services	Communication
Create Cart, Get Cart, Add item, Update cart, Checkout Cart	Catalog Microservice, Ordering Microservice, Identity Microservice	Catalog Microservice, Ordering Microservice	Cache Redis	gRPC

Table 6.1: A Sample of the Basket Microservice's Analyzed Data.

6.2.2 TRANSLATION

Once we collected and analyzed the data, we mapped the identified concepts to the respective classes and relations in OMSAC's TBox which is presented in Figure 6.1. Then, we modelled the systems by creating individuals and linking them following the identified concepts and unveiled relations from the identification step using Protégé.

The outcome of this step is an ABox composed of the three systems that was modelled using the OMSAC's vocabulary and the OWL2-DL language, stocked in a Turtle [17] file. Listing 6.1 shows an excerpt of this ABox containing some aspects of the Basket Microservice.

```
###   http :// www.semanticweb.org/WebBasedMicroservicesModels#BasketMicroservice
msa:BasketMicroservice  rdf : type  owl:NamedIndividual ,  <http :// www.semanticweb.org/ontologyOfMicroservicesArchitecture#Microservice> ;
<http :// www.semanticweb.org/ontologyOfMicroservicesArchitecture#codedIn> msa:CSharp ;
<http :// www.semanticweb.org/ontologyOfMicroservicesArchitecture#communicatesThrough> msa:gRPC ;
<http :// www.semanticweb.org/ontologyOfMicroservicesArchitecture#dependsOn> msa:CatalogMicroservice ,  msa:OrderingMicroservices   ;
<http :// www.semanticweb.org/ontologyOfMicroservicesArchitecture#deployableAs> msa: IsolatedContainer    ;
<http :// www.semanticweb.org/ontologyOfMicroservicesArchitecture#deployableOn> msa:AKS ;
<http :// www.semanticweb.org/ontologyOfMicroservicesArchitecture# exploits > msa:RedisCacheEShop ;
<http :// www.semanticweb.org/ontologyOfMicroservicesArchitecture#implements> msa:CheckoutCart ,  msa:CreateCart   ,  msa:GetCart  ,  msa:UpdateCart ;
<http :// www.semanticweb.org/ontologyOfMicroservicesArchitecture# interactsWith > msa:CatalogMicroservice   ,  msa: IdentityMicroservice    ,  msa:OrderingMicroservices   ;
rdfs : label   "Basket␣Microservice"@en .
```

Listing 6.1: Excerpt of the OMSAC's ABox serialized in Turtle

The result was that we created a Stardog triple store instance in a Docker container to store the knowledge base. Then, we uploaded the OMSAC TBox, and the ABox which was created in this step.

6.3 ANALYSIS

This section presents the approaches used to analyze the OMSAC-based models of the three systems.

We used the following competency questions (CQ) to explore the content of the knowledge base. These competency questions examined the different aspects of these systems and were driven by the objectives in our use case.

CQ-1 Which microservices compose a system?

CQ-2 Which features are implemented by a given microservice?

CQ-3 Which requirements are met by a microservice?

CQ-4 Which features are related to a given feature?

CQ-5 What is the shortest path between two features?

CQ-6 Where is the microservice deployed?

CQ-7 Which interconnections exist between the microservices?

CQ-8 Which services are used by a microservice?

CQ-9 Which microservices are similar?

We have translated these competency questions into SPARQL queries, which execution outputs are sub-graphs extracted from the knowledge base. These sub-graphs represent different views of these systems. Also, we relied on these sub-graphs as input to measure similarities among microservices. Similarity metrics were measured using three different approaches, which we detailed below.

6.3.1 MEASURING SIMILARITY

In this subsection, we present the three approaches used to measure similarity among the microservices contained in the knowledge base. These approaches have as input a microservice which will be compared to other microservices in the knowledge base.

1. Machine Learning Model

Stardog provides a native similarity model based on machine learning techniques, which can measure similarity among individuals in a knowledge base. This similarity model is an ensemble model based on three different approaches: Syntactic, semantic, and structural. The syntactic similarity is measured on the characters composing the labels by applying the following such examples "edit distance, fuzzy string matching or trigram cosine similarity." The semantic similarity is measured on the meaning of the labels by exploiting "a manually curated lexical database (e.g. WordNet) or a separately trained word embedding model." The structural similarity is measured through the schema's structure, which detects "relationships having the same source and target types" [109].

Concretely, we created these models using SPARQL "INSERT" queries, where we declared the features and the prediction variables which the model would use. Also, we fed the model with a graph extracted from the knowledge base. This graph was used as the training data by the model. Listing 6.2 shows the SPARQL query used to create this model.

```
prefix omsac: <http :// www.semanticweb.org/ontologyOfMicroservicesArchitecture#>
prefix msa: <http :// www.semanticweb.org/WebBasedMicroservicesModels#>
prefix spa: <tag: stardog : api : analytics :>

INSERT {
    graph spa:model {
        :simModelBenni a spa:SimilarityModel ;
                spa:arguments (? features ?protocols ? interactions ?coevolutions ) ;
                spa: predict ?microservice .
    }
}
WHERE {
    SELECT
    (spa:set(? feature ) as ? features )
    (spa:set(? protocol ) as ? protocols )
    (spa:set(? interaction ) as ? interactions )
    (spa:set(? coevolution ) as ? coevolutions )
    ?microservice
    {
        ?microservice omsac:implements ?feature .
        OPTIONAL{?microservice omsac:interactsWith ?interaction.}
        OPTIONAL{?microservice omsac:communicates ?protocol.}
        OPTIONAL{?microservice omsac:dependsOn ?coevolution.}
    }
    GROUP BY ?microservice
}
```

Listing 6.2: Creation of a Stardog Similarity Model Based on the Criteria in the Work of Benni et al. [18]

2. EgdeSim

The EdgeSim metric was proposed by Mitchell et al. [82] to measure similarity among graphs, and took into account the internal and external edges between entities, which could be weighted. This metric had been used to calculate the similarity between two clusters having the same number of edges in software decomposition approaches.

The general formula of the EdgeSim metric is:

$$EdgeSim(A,B) = \frac{Y}{E} \tag{6.1}$$

Where A, B are the graphs representing each cluster, Y is the sum of the edges' weights that are of the same type in both graphs (inter and intra-edges), and E is the sum of the weight of all edges in the graph. When all edges have the same weight, we set the value to 1.

When comparing *Hipster Shop*, *eShopOnContainers* and *Vert.x* architectures, graphs A and B do not represent clusters but models. Hence, the edges in these graphs represent the relations between the components of the model, and the number of edges may therefore vary. As the modelled application in the knowledge base is not required to have the same number of relations, we restricted the number of edges to the number of relations of the input microservice to measure the similarity correctly. Also, we considered the edges to have the same weight and consequently, we set the number of relations of the input microservice to *E*.

The next step was that we implemented the EdgeSim algorithm using SPARQL queries. Listing 6.3 shows the request used to calculate the sum of *Y*. To see the complete version of this implementation, refer to the source code companion [86] of this use case.

```
SELECT ?microservice
       ? similarMicroservice  ((? yFeature+? yInteracts +?yProtocol+?yDependsOn) as ?Y)
  {
    SELECT ?similarMicroservice
    (count( distinct  ? feature )  as  ?yFeature)
    (count( distinct  ? interaction )  as  ? yInteracts )
    (count( distinct  ?protocol )  as  ?yProtocol)
    (count( distinct  ?dependencies)  as  ?yDependsOn)
    WHERE
    {   VALUES ?microservice {msa:BasketMicroservice}
        ? feature  ^omsac:implements ?microservice,  ? similarMicroservice  .
        OPTIONAL{?microservice omsac:dependsOn ?anotherMicroservice.
             ?anotherMicroservice  omsac:implements ?dependencies.
             ? similarMicroservice  omsac:dependsOn ?anotherMicroservice2.
             ?anotherMicroservice2  omsac:implements ?dependencies.}
        OPTIONAL{?microservice omsac:interactsWith ?otherMicroservice.
              ? otherMicroservice  omsac:implements ? interaction .
              ? similarMicroservice  omsac: interactsWith  ? otherMicroservice2 .
              ? otherMicroservice2  omsac:implements ? interaction .}
        OPTIONAL{?protocol ^omsac:communicates ?microservice, ?similarMicroservice.}
    }
    GROUP BY ?similarMicroservice
  }
```

Listing 6.3: EdgeSim metric based on the criteria introduced in the work of Benni et al. [18]

3. Manual

The manual-informal analysis of these systems was done by a practitioner using the data from the first step of our approach, and which we cleaned to enhance understanding. We limited the analysis to the microservices implementing the cart management functionality and limited the information provided to the features, interactions, coevolution, platform-provided services, and communication technologies related to them.

6.4 FINDINGS

In this section, we discuss the use case findings. We begin by first presenting findings related to model exploration. Then, we present the results and findings from the measured similarity.

6.4.1 EXPLORING THE MODELS

As mentioned previously, when we query the knowledge base, the results will be sub-graphs, as the knowledge base itself is a graph. Depending on the criteria used when querying it, the resulting sub-graphs will be able to address different stakeholder information needs, and this is what we observed with the competency questions presented above. Different criteria call for different classes and relations in the knowledge base, and in Table 6.2 the OMSAC's classes and relations used for each query are provided. The complete queries are available on the source code companion [86] of this use case.

Indeed, results for CQ1 to CQ5 are sub-graphs that focus on the system's functional aspects; thus, representing functional models, which can be valuable in addressing the information needs of both business and functional analysts. These sub-graphs are represented as RDF triples, and they can be visualized graphically. Listing 6.4 shows the sub-graph extracted for the CQ3 in the RDF representation, while Figure 6.2 provides the graphical visualization of this RDF graph. Also, Figure 6.3 shows the query and graphical representation of the shortest path between the features *Get product* and *Create order* which is one possible answer to CQ-5.

Competency Questions	OMSAC Classes	OMSAC Relations
CQ1 - Which microservices compose a system?	MicroserviceArchitecture Microservice	isComposedOf
CQ2 - Which features are implemented by a microservice?	Microservice Functionality	implements
CQ3 - Which requirements are met by a microservice?	Functionality Requirement Microservice	implements respondsTo
CQ4 - Which features are related to a given feature?	Functionality	interactsWith dependsOn
CQ5 - What is the shortest path between two features?	Functionality Microservice	all
CQ6 - Which are the technical dependencies of a microservice?	Microservice Technology	exploits, exposedBy communicatesThrough
CQ7 - Where is a microservice deployed?	Microservice Infrastructure	deployedOn
CQ8 - Which interconnections exist between microservices?	Microservice	interactsWith dependsOn
CQ9 - Which microservices are similar?	Microservice Functionality Technology	implements interactsWith communicatesThrough dependsOn, codedIn

Table 6.2: Use of OMSAC elements to respond to the Competency Questions

```
@prefix omsac: <http :// www.semanticweb.org/ontologyOfMicroservicesArchitecture#> .
@prefix msa: <http :// www.semanticweb.org/WebBasedMicroservicesModels#> .

msa:BasketMicroservice omsac:implements msa:CheckoutCart .
msa:CheckoutCart omsac:respondsTo msa:UserShoppingCartManagement .
msa:BasketMicroservice omsac:implements msa:CreateCart .
msa:CreateCart omsac:respondsTo msa:UserShoppingCartManagement .
msa:BasketMicroservice omsac:implements msa:GetCart .
msa:GetCart omsac:respondsTo msa:UserShoppingCartManagement .
msa:BasketMicroservice omsac:implements msa:UpdateCart .
msa:UpdateCart omsac:respondsTo msa:UserShoppingCartManagement .
```

Listing 6.4: RDF graph of the answer to the Competency Question *Which requirements are met by a microservice?*

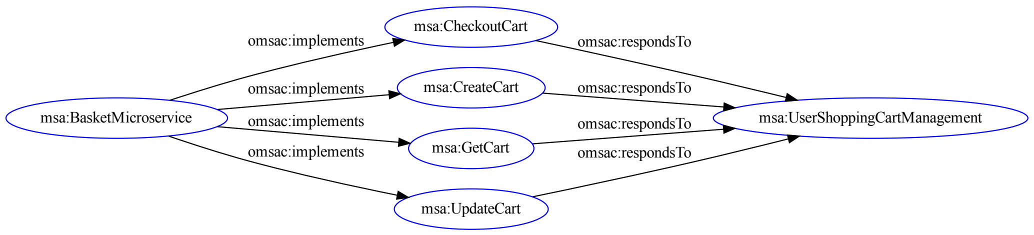
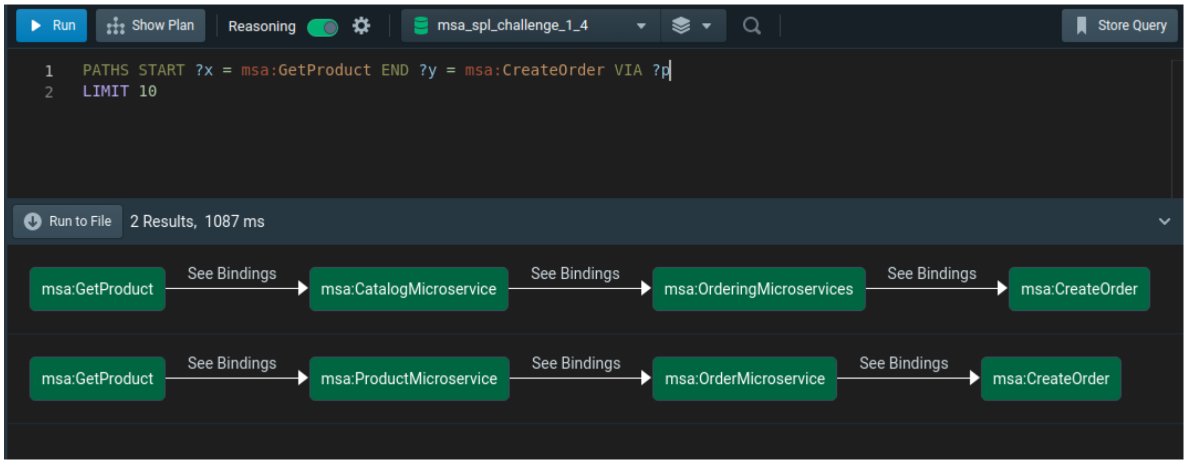


Figure 6.2: Graph of the answer to the Competency Question *Which requirements are met by a microservice?*



```
PATHS START ?x = msa:GetProduct END ?y = msa:CreateOrder VIA ?p
LIMIT 10
```

2 Results, 1087 ms

```
msa:GetProduct → msa:CatalogMicroservice → msa:OrderingMicroservices → msa:CreateOrder
msa:GetProduct → msa:ProductMicroservice → msa:OrderMicroservice → msa:CreateOrder
```

Figure 6.3: Query and graph of the answer to the Competency Question *What is the shortest path between two features?*

On the other hand, results for queries responding to the competency questions CQ6 to CQ8 focus on technical aspects; thus, the sub-graphs extracted represent technological models that could address the needs of operators and integrators. Listing 6.5 shows the sub-graph extracted for the CQ7 in the RDF representation, while Figure 6.4 provides the graphical visualization of the RDF graph.

```
@prefix omsac: <http :// www.semanticweb.org/ontologyOfMicroservicesArchitecture#> .
@prefix msa: <http :// www.semanticweb.org/WebBasedMicroservicesModels#> .

msa:BasketMicroservice  omsac: exploits  msa:RedisCacheEShop ;
    omsac:exposedBy msa:APIGatewayShoppingMobile ;
    omsac:communicatesThrough msa:gRPC ;
    omsac:exposedBy msa:APIGatewayShoppingWeb , msa:ShoppingAggregatorMobile , msa:ShoppingAggregatorWeb .
```

Listing 6.5: RDF Graph of the answer to the Competency Question *Which interconnections exist between the microservices?*

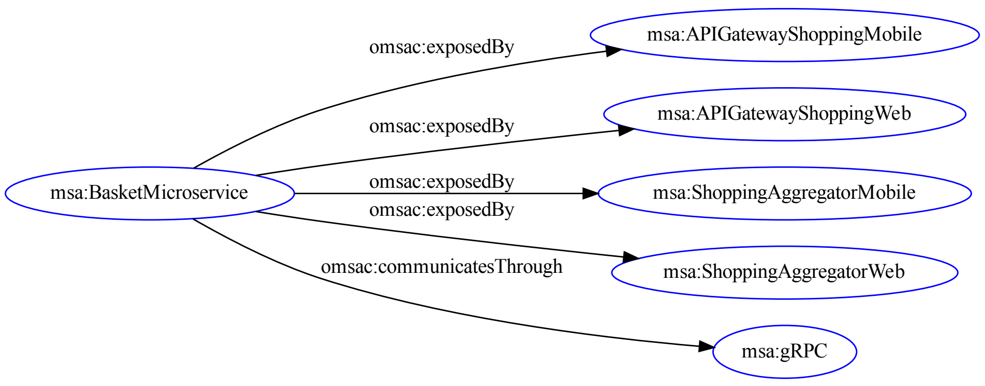

msa:BasketMicroservice
omsac:exposedBy
msa:APIGatewayShoppingMobile
omsac:exposedBy
msa:APIGatewayShoppingWeb
omsac:exposedBy
msa:ShoppingAggregatorMobile
omsac:exposedBy
msa:ShoppingAggregatorWeb
omsac:communicatesThrough
msa:gRPC

Figure 6.4: Graph of the answer to the Competency Question *Which interconnections exist between the microservices?*

Thus, the first finding was that modelling these three systems with OMSAC allowed us to explore various viewpoints using the same model.

6.4.2 FINDING SIMILARITY AMONG MICROSERVICES

We evaluated the CQ9 by applying the three different approaches presented in Sub-section 6.3.1, using a scenario. Each approach was expected to identify a similar service suitable to replace the *Basket Microservice* of the *eShopOnContainers* system in this scenario. A similar service was thought to be identified among all the services from the three systems using the criteria proposed in the work of Benni et al. [18].

We used this scenario to ensure the feasibility of the manual approach, as it would be hard for a human being to analyze all aspects of the 25 microservices that compose the three systems. Also, the manual approach was used as a baseline to estimate the accuracy of the other approaches by considering the gap between the microservices chosen by the expert and those suggested by the Stardog similarity model and the EdgeSim metric.

Furthermore, regarding the manual approach, we were further interested in which aspects the expert had considered to be the most and the less relevant for establishing similarity. To represent this perceived relevance, we relied on weights of each aspect. Relevant aspects were set to 1, less relevant to 0,5, and irrelevant or out of scope to 0.

The expert analysis outcome was that the *Shopping Cart Microservice* from the *Vert.x* system ended up being the most similar one in terms of implemented features. However, the technological differences between this microservice and the *Basket Microservice* were perceived as barriers to interchange them. Therefore, based on the expert analysis, the *Hipster Shop* system's *Cart Service* seemed to be more suitable to replace the *Basket Microservice*. Table 6.3 summarizes the results of the analysis conducted by the expert.

Microservice	**Basket Microservice**	**Cart Service**	**Shopping Cart Microservice**
Functional	9/9	2/9	5/9
Technical	2/2	2/2	0/2
Expert Score	100,00%	61,11%	27,78%

Table 6.3: Manual-informal expert's similarity analysis

Then, we compared the results obtained from the other measure approaches to those from the manual approach. Table 6.4 shows the metrics obtained with each technique for the

Microservice	Expert	Stardog Similarity Model	EdgeSim
Basket Microservice	100,00%	100,00%	100,00%
Cart Service	61,11%	56,57%	22,22%
Shopping Cart Microservice	27,78%	33,54%	27,78%

Table 6.4: Comparison of the similarity metric using the different approaches.

Basket Microservice. The results showed that the similarity measured using Stardog's machine learning approach performed better than the EdgeSim metric and was close to the expert's one in the identification of interchangeable microservices.

The second finding from this use case was that it was possible to identify similar microservices using the OMSAC-based models of the three systems coupled with measurement techniques. Depending on the criteria that were used, functional, operational, and technical aspects could also be considered when measuring such similarity. We employed functional and technology aspects stored in the same model in the scenario used.

Briefly, this use case showed that using OMSAC-based models to represent microservices-based system architectures was valuable for exploring the information of these systems from various viewpoints and for extracting metrics based on these descriptions. Compared to other approaches requiring model manipulation, meaning, for example, model transformation and merge for extracting a holistic view, modelling these systems using OMSAC allowed describing functional, technical, and operational aspects using a unique model without the need for model transformation.

CHAPTER 7

EVALUATION

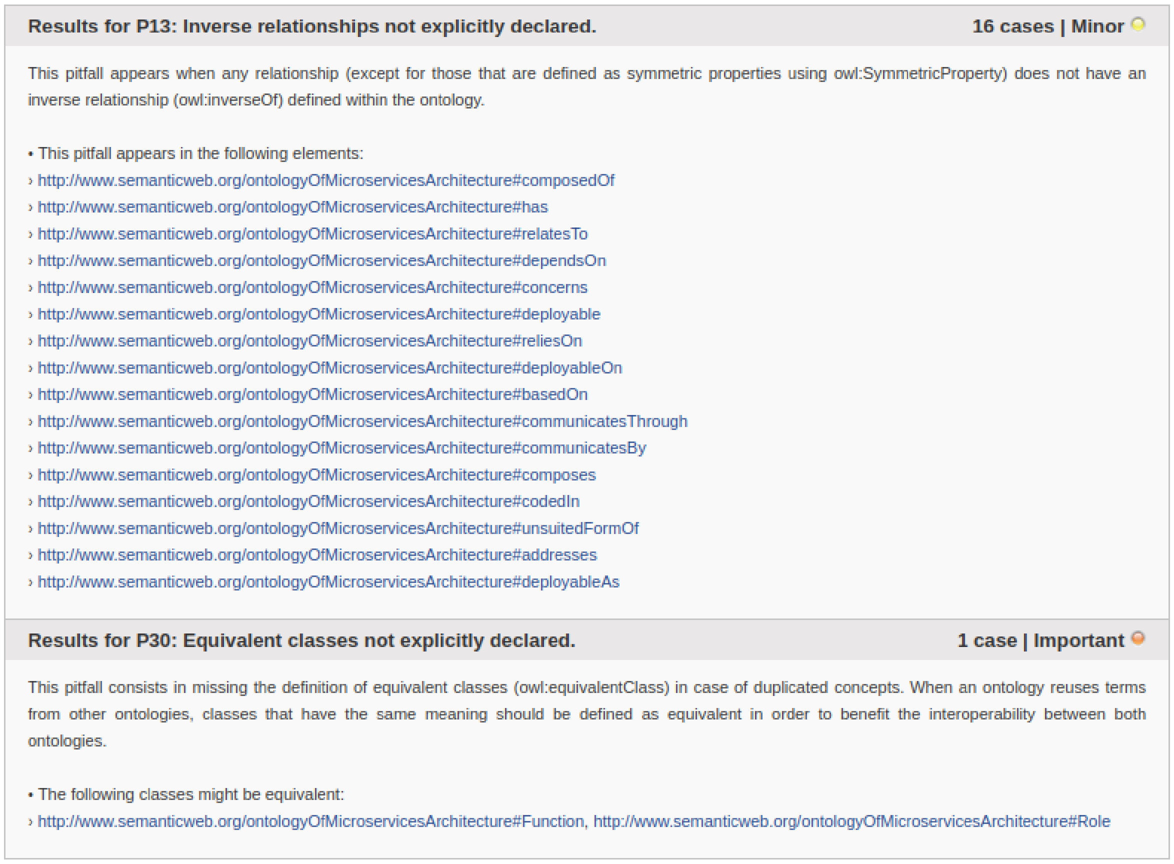
In this chapter, we present the evaluation of OMSAC following the validation and verification process defined in Chapter 4. We begin the chapter by first presenting and discussing the quality evaluation based on the automatic approach. Secondly, we present and discuss the evaluation based on the data-driven and application-driven approaches, with the latter being accomplished by using a use case. The use case was presented in details in Chapter 6. Lastly, we evaluate OMSAC concerning the research objectives.

7.1 QUALITY

This section presents the results from the quality evaluation of OMSAC. We evaluated OMSAC's quality using two automatic approaches which were OOPS! and HermiT inference engine.

7.1.1 OOPS! STRUCTURAL AND CONSITENCY VALIDATION

OOPS! allows verifying different pitfalls in an ontology. These pitfalls are recorded in a pitfall catalogue, which provides a solution to address them. They are organized in the following importance levels: Critical, important and minor. We assessed the OMSAC ontology using advanced validation based on the structural dimension and the consistency criteria.



Results for P13: Inverse relationships not explicitly declared. **16 cases | Minor**

This pitfall appears when any relationship (except for those that are defined as symmetric properties using owl:SymmetricProperty) does not have an inverse relationship (owl:inverseOf) defined within the ontology.

• This pitfall appears in the following elements:
› http://www.semanticweb.org/ontologyOfMicroservicesArchitecture#composedOf
› http://www.semanticweb.org/ontologyOfMicroservicesArchitecture#has
› http://www.semanticweb.org/ontologyOfMicroservicesArchitecture#relatesTo
› http://www.semanticweb.org/ontologyOfMicroservicesArchitecture#dependsOn
› http://www.semanticweb.org/ontologyOfMicroservicesArchitecture#concerns
› http://www.semanticweb.org/ontologyOfMicroservicesArchitecture#deployable
› http://www.semanticweb.org/ontologyOfMicroservicesArchitecture#reliesOn
› http://www.semanticweb.org/ontologyOfMicroservicesArchitecture#deployableOn
› http://www.semanticweb.org/ontologyOfMicroservicesArchitecture#basedOn
› http://www.semanticweb.org/ontologyOfMicroservicesArchitecture#communicatesThrough
› http://www.semanticweb.org/ontologyOfMicroservicesArchitecture#communicatesBy
› http://www.semanticweb.org/ontologyOfMicroservicesArchitecture#composes
› http://www.semanticweb.org/ontologyOfMicroservicesArchitecture#codedIn
› http://www.semanticweb.org/ontologyOfMicroservicesArchitecture#unsuitedFormOf
› http://www.semanticweb.org/ontologyOfMicroservicesArchitecture#addresses
› http://www.semanticweb.org/ontologyOfMicroservicesArchitecture#deployableAs

Results for P30: Equivalent classes not explicitly declared. **1 case | Important**

This pitfall consists in missing the definition of equivalent classes (owl:equivalentClass) in case of duplicated concepts. When an ontology reuses terms from other ontologies, classes that have the same meaning should be defined as equivalent in order to benefit the interoperability between both ontologies.

• The following classes might be equivalent:
› http://www.semanticweb.org/ontologyOfMicroservicesArchitecture#Function, http://www.semanticweb.org/ontologyOfMicroservicesArchitecture#Role

Figure 7.1: OMSAC's Structural Dimension Analysis Executed Using *OOPS!*.

Figure 7.1.1 shows the results from *OOPS!* in reference to the structural dimension. In this figure, we can observe that two pitfalls were identified: P13—Inverse relationships that are not explicitly declared and P30—Equivalent classes that are not explicitly declared. The P13 pitfall is a minor level one. It considers that for each object property, an inverse property may exist. As it is a “nice to have” [99] that does not impact OMSAC performances, we decided not to add the suggested inverse relationships.

The P30 pitfall is an important level one. However, the equivalent classes identified by *OOPS!* were not equivalent in meaning in the context of OMSAC. Indeed, the OMSAC’s “Function” concept is related to software’s characteristics as its “Role” concept is related to the organizational concepts of ownership and responsibility. Thus, we ignored this pitfall.

Concerning the consistency assessment of OMSAC using *OOPS!*, no pitfalls have been found.

7.1.2 HERMIT LOGICAL VALIDATION

As HermiT is the native inference engine of Protégé, we conducted the quality evaluation using its interface. When a logical inconsistency is detected, the inference engine shows an error message and points to the classes and relations where the logical issue was detected. Hermit did not detect any logical inconsistency in OMSAC.

7.2 CORRECTNESS

This section presents the evaluation of OMSAC through completeness and coverage of the ontology in regard to the Microservices Architecture domain. This evaluation allowed us to assess OMSAC’s correctness. We conducted this evaluation in two phases. We began by evaluating the OMSAC’s TBox using a data-based approach based on the competency questions and the data taken from literature review. Secondly, we evaluated OMSAC using an application-based approach through a use case representing a real-world application of

OMSAC to model an aspect of the Microservices domain.

7.2.1 DATA-BASED EVALUATION

As discussed previously, competency questions were a significant piece of the methodology used to build OMSAC. In addition, the competency questions approach has been widely adopted for evaluating ontologies [41, 58, 128]. Therefore, it was logical for us to use competency questions in order to validate that OMSAC meets its intended purpose. The following competency questions were used to evaluate OMSAC's terminological component, and it was expected that OMSAC should be able to answer them.

CQ-1 What is the definition of Microservices Architecture?

CQ-2 What is the definition of Microservice?

CQ-3 Which concepts in the ontology could be used to describe a microservice?

CQ-4 Which relations could exist between a microservice and a microservices architecture?

CQ-5 Which concepts in the ontology could be used to describe a design pattern?

CQ-6 Which concepts in the ontology could be used to describe an anti-pattern?

CQ-7 Which concepts in the ontology could be used to describe both patterns and anti-patterns?

CQ-8 How could a microservice be related to a provider?

CQ-9 How could a microservice be related to a technology?

CQ-10 How could a microservice be related to a pattern?

CQ-11 How could a microservice be related to an anti-pattern?

CQ-12 Which patterns could be used by a microservices architecture?

CQ-13 Which anti-patterns could be present in a microservices architecture?

CQ-14 How a pattern could be related to an anti-pattern?

We began by translating these competency questions into SPARQL queries which we executed against the OMSAC's TBox stored in a Stardog triple store instance. Following, we present some of these queries (Listings 7.1, 7.2 and 7.3), all queries are available in this thesis source code companion [84]. The answers for these competency questions were compared to the reference corpus and the relations identified during the ontology building.

```
SELECT ?definition
WHERE {
    ?MSA rdf:type omsac:MicroservicesArchitecture .
    ?MSA rdfs:comment ?definition.
}
```

Listing 7.1: SPARQL Query for the Competency Question ***What is the definition of Microservices Architecture?***

```
SELECT ?definition
WHERE {
    ?MSA rdf:type omsac:Microservice .
    ?MSA rdfs:comment ?definition.
}
```

Listing 7.2: SPARQL Query for the Competency Question ***What is the definition of Microservice?***

```
SELECT ?relation
WHERE {
    ?microservice rdf:type omsac:Microservice .
        ?msa rdf:type omsac:MicroservicesArchitecture.
    ?microservice ?relation ?msa.
}
```

Listing 7.3: SPARQL query for the competency question *Which relations could exist between a microservice and a microservices architecture?*

The data-based evaluation allowed us to verify that no information was lost during the ontology building process and that the identified relationships between the concepts were correctly formalized in the TBox of OMSAC.

7.2.2 APLICATION-BASED APPROACH

The evaluation of OMSAC applying the application-based approach was conducted using the use case presented in Chapter 6. This use case aimed to represent microservices-based systems using the OMSAC TBox to create an ABox containing assertions about the domain. This ABox was then stored among the TBox in a triple store, forming a knowledge base that we evaluated through competency questions.

Applying OMSAC to describe microservices-based systems allowed us to evaluate the completeness and coverage of the OMSAC TBox. Even if the use case contained a limited number of microservices, it was possible to verify that the vocabulary proposed by OMSAC was rich enough to describe various aspects of these systems. In effect, we created a unique model containing all the relevant information using the OMSAC vocabulary, which meant having the functional, technological, operational and organizational information together into the same knowledge base.

We then explored this knowledge base through competency questions, which demonstrates that we could query the model using various criteria and then obtained information that would meet different information needs. Likewise, we used automatic and manual approaches to find similar microservices in the knowledge base, showing that the OMSAC's vocabulary is compliant with the human and machine understandability characterizing ontologies.

7.2.3 COMPUTATIONAL EFFICIENCY

Computation efficiency was evaluated when querying both OMSAC's TBox and the use case's knowledge base through SPARQL queries using the Stardog triple store. Stardog uses Pellet as reasoner, which provided us with a different reasoner to assess OMSAC computational capabilities. We observed that Stardog Studio was able to execute the queries we submitted, thus no reasoning issue was detected.

In addition, we did not observe any execution delay issue. Indeed, the average query time were close to 0.5 seconds, and the most time-consuming queries, which were those for calculating the EdgeSim metric, had peaked 1 second.

7.3 EVALUATION AGAINST THE RESEARCH QUESTIONS

In this section OMSAC is evaluated with regard to the research questions presented in Section1.1.

RQ-1 How to provide a set of terms allowing to describe both MSA knowledge and artifacts?

RQ-1 asks for a vocabulary able to describe both the knowledge and the artifacts related to the Microservices Architecture. As presented in the two chapters of this thesis, 'Background' and 'Related Work', MSA concepts and artifacts share a common base vocabulary. Thus, organizing a common vocabulary to represent both core principles of MSA and the artifacts composing this kind of system architecture was a feasible aim.

The evaluation of OMSAC demonstrated that organizing MSA-related vocabulary in an ontological form would provide a classification system that could represent different aspects of a microservice-based system. Furthermore, ontologies allow term evolution as each concept is defined atomically, and complex concepts and ideas are expressed by creating relations among these atomic concepts. Thus, a concept can have different relations and then be applied to respond to different representation needs. Consequently, we considered that an ontological approach was adequate to respond to RQ-1.

RQ-2 How to describe microservice-based architectures allowing to explore them from various viewpoints?

Answering RQ-1 provided the foundation required to answer the RQ-2, as the former is a prerequisite to the latter. To allow a holistic representation of the MSA concepts and artifacts, it was mandatory to have a common vocabulary and an efficient mechanism able to gather this information accurately and efficiently. OMSAC provided such vocabulary (RQ-1), and the resulting OMSAC model of the MSA concepts allowed for the exploration of microservices-based architectures from different viewpoints.

For example, it is possible to query a microservices-based architecture modelled using OMSAC to accomplish the following: Obtain a term definition, expose a design pattern applied, and retrieve information about functional aspects of a specific microservice.

The ontological description is, in fact, a directed graph composed of the variety of concepts and relations allowed by the ontology's TBox. This graph can then be queried using various criteria to provide information according to different viewpoint needs. Thus, we addressed challenges in the analysis and exploration of multi-viewpoints and modelling in different granularity levels because, in OMSAC-based models, all viewpoints were stored in the same model, which allowed for the immediate analysis and exploration of these viewpoints.

RQ-3 How to improve microservice's identification and reuse using its description?

Microservices-based systems are distributed systems composed of microservices and services provided by their deployment platform. Identifying those components to allow their reuse is paramount in such distributed architectures and is notably not an easy task. Various aspects must be considered when identifying a microservice that is likely to be reusable in a given context. Thus, RQ-3 is a significant question to this challenge in microservices architecting, where reuse depends on microservice identification.

The complexity of these architectures and the diversity of their representations can jeopardize reuse. Indeed, before establishing reuse, it is mandatory to retrieve adequate information, which can be hard to accomplish when it is stored in different information sources such as models, documents, graphs.

As established in the evaluation mentioned above, OMSAC-based models support the description of various aspects of microservices-based systems. The ontological nature of these models makes them intelligent because we can make inferences on the knowledge they represent. In addition, they are extensible because we can add advanced artificial intelligence techniques like machine learning to them.

Indeed, in the use case presented in Chapter 6 and referred to in this evaluation, we used the OMSAC-based representation as the input to measure similarity metrics. According to the desired criteria, this measure can be computed based on sub-graphs extracted from the knowledge base and using different techniques, such as machine learning. Providing a synthetic measure of similarity that allowed us to deal with the complexity of these architectures and thus facilitated reuse decisions. Consequently, modelling a microservices-based system using an ontology, such as OMSAC, seemed to answer RQ-3 adequately.

RQ-4 How can heterogeneous stakeholders and their information needs be responded to by a holistic modelling of microservices-based systems?

As discussed in the RQ-2, OMSAC is compliant with different viewpoint exploration. Consequently, it can provide different information able to meet various stakeholder information needs.

Compared to other MSA representation approaches in which each viewpoint composes different models, OMSAC-based representation avoids model transformations and merge to provide a holistic view. In fact, operations on models can be hard to execute and time-consuming as they depend on their represented form such as drawings, graphs, documents and different models, and thus using an OMSAC-based representation allows all the information to be represented in the same model—using the same syntax and formalism, avoiding both model manipulation and drastically reducing execution time.

Thus, responding to specific information needs of stakeholders means extracting from the knowledge base an information sub-set using the criteria provided by the stakeholder, without having any model manipulation such as transformation or merging. Consequently, modelling a microservices-based system using the OMSAC approach correctly responds to this research question.

CHAPTER 8

CONCLUSION

In this chapter, we discuss the limitations and perspectives of this thesis, as well as review its contributions and future work.

8.1 LIMITATIONS

This research also comes with limitations. The most prominent one concerns the external validity of the evaluation results. Indeed, as highlighted by Hlomani et al. [66], ontology evaluation contains a part of subjectivity that influences the perceived ontology coverage and completeness. Using a real-world use case to assess the quality of an ontology has limited the possible bias when evaluating coverage and completeness. However, the microservices-based systems used in this use case have a limited scope and are composed of only 25 microservices, which threatens this evaluation. Consequently, future research must apply OMSAC to more complex and larger microservices-based architectures.

In addition, considering a limited number of microservices and criteria when measuring similarity could threaten validity, however, this was an intentional decision as we wanted to involve a manual approach conducted by a human expert who would not be able to comprehend and analyze larger models manually. Despite this limitation, the evaluation shows it is a promising application of OMSAC. Thus, we expect that our approach is even more valuable when the complexity of the MSA exceeds the cognitive abilities of humans.

Lastly, it is important to note that because the MSA domain is constantly evolving, there may be, new terms, concepts, principles, definitions or technology that might have risen and were not considered by our work.

8.2 PERSPECTIVES

Using formalized knowledge for building machine learning models is a rising research domain. It has been proposed in the work of Von Rueden et al. [133] under the name of "informed machine learning." Informed machine learning aims to address existing limitations in the development of machine learning models based on data because available amount of data is often not enough to generate a generalizable model.

In this direction, Wilcke et al. [136] have suggested using existing knowledge expressed as knowledge graphs as default models for machine learning. According to them, knowledge graphs enable an "end-to-end learning approach," prevents transformation activities of raw data, simplifies the integration and harmonization of heterogeneous knowledge, and provides a natural way to integrate different prior knowledge. This kind of knowledge can make it easier to identify relationships within the data and acts as prior knowledge for creating machine learning models with limited or heterogeneous datasets.

Considering that an ontology formalized in OWL2 is a knowledge graph and an artificial intelligence technique based on description logics, we believe that it qualifies to be used as prior knowledge to build informed machine learning models. Therefore, an ontology could be used to restrict the universe of assumptions, construct the learning algorithm, and be used as the training dataset.

In addition, other artificial intelligence techniques could extend ontological representation capabilities [13]. For instance, we could consider probabilistic approaches as Bayesian networks [120], which allow logical descriptions of relations to be considered and have an acyclic directed graph structure [107]. Furthermore, theoretically, it is possible to use ontologies as primary dataset in order to create Bayesian networks [36]. Appendix C provides the results of a proof of concept we conducted to evaluate the use of OMSAC's TBox to build a Bayesian Network.

Consequently, exploring the use of OMSAC TBox and OMSAC-based knowledge bases coupled with other artificial intelligence techniques would be a promising perspective for addressing microservices architectures modelling and analysis, notably building intelligent mechanisms using a limited amount of data. Such intelligent mechanisms would be able to handle modelling challenges, including identification of microservices for reuse, interoperability and interchangeability.

8.3 CONCLUSION AND FUTURE WORK

This thesis presented an evolutive approach based on an ontological representation to formalize Microservices Architecture concepts and artifacts. Our intent has been twofold: Provide a shared vocabulary for exploring and understanding MSA concepts and using those concepts for designing microservices architectures.

Based on this intent, we have proposed the Ontology of Microservices Architecture Concepts (OMSAC), which provides terminology for describing concepts in the domain of Microservices Architecture. These concepts are beyond the core and consensual principles of MSA. OMSAC also includes concepts to describe MSA's artifacts as models. To the best of our knowledge, it is the first ontological formalization of MSA principles to describe both MSA concepts and artifacts.

We have relied on previous works in the classification and taxonomy of the Microservices Architecture domain to identify the terms and their relationships for building an ontology rich enough to express complex concepts. Also, we move towards methodologies and methods in ontology engineering to assist us in designing, building and validating OMSAC.

The evaluation process presented a use case in which we applied OMSAC as a tool for an ontology-driven conceptual modelling approach for modelling microservices-based systems. This approach allows modelling and analyzing microservices-based systems. OMSAC-based models represent MSA systems holistically or decomposed into individual viewpoints that meet different stakeholders' specific information needs. We used machine learning and manual analysis by a human expert to analyze these systems.

These models support the identification of existing microservices and provided services, which could be reused as-is or in an extended version and highlight technical and platform-driven concerns mandatory for efficient designing. Also, we have shown this approach provides designers with synthetic similarity metrics by using various criteria, which could support microservices identification for reuse. Likewise, we demonstrated that using machine learning techniques can simplify the computation of such metrics. However, we identified a lack of which MSA properties we need to consider when establishing accurate similarity metrics. Our investigation in this field is still in progress.

The limited capacity of humans experts to deal with the complexity of microservices-based systems quickly becomes evident. Therefore, applying intelligent approaches to address inherent modelling challenges faced by human analysts is mandatory. We demonstrated that using an ontological approach along with machine learning techniques to represent and analyze microservices architectures would be a promising research perspective to handle these challenges.

Future work will establish the MSA's most suitable properties to be used in a machine learning model to classify components of an MSA-based system and build variability metrics. We also plan to develop a domain-specific language (DSL) to encapsulate OWL2 and SPARQL queries that will enhance OMSAC implementation and provide a simplified vocabulary shaped for domain experts. Furthermore, we plan to develop supporting tools and an intelligent model projection mechanism based on OMSAC and machine learning techniques.

BIBLIOGRAPHY

[1] Mehdi Adda. *Intégration des connaissances ontologiques dans la fouille de motifs séquentiels avec application à la personnalisation web*. PhD thesis, Lille 1, 2008.

[2] Mustafa M Al-Sayed, Hesham A Hassan, and Fatma A Omara. Cloudfnf: An ontology structure for functional and non-functional features of cloud services. *Journal of Parallel and Distributed Computing*, 2020.

[3] Dean Allemang and Jim Hendler. *Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL*. Elsevier Science, 2011.

[4] Dean Allemang, Jim Hendler, and Fabien Gandon. *Semantic Web for the Working Ontologist: Effective Modeling for Linked Data, RDFS, and OWL*. Association for Computing Machinery, New York, NY, USA, 3 edition, 2020.

[5] Nuha Alshuqayran, Nour Ali, and Roger Evans. A systematic mapping study in microservice architecture. In *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 44–51. IEEE, 2016.

[6] Grigoris Antoniou, Paul Groth, Frank van Harmelen, and Rinke Hoekstra. *A Semantic Web Primer*. Information Systems. MIT Press, 2012.

[7] Ali Arsanjani, Grady Booch, Toufic Boubez, P Brown, David Chappell, John deVadoss, Thomas Erl, Nicolai Josuttis, Dirk Krafzig, Mark Little, et al. Soa manifesto. *SOA Manifesto*, 35:82–88, 2009.

[8] Timo Asikainen, Tomi Männistö, and Timo Soininen. Kumbang: A domain ontology for modelling variability in software product families. *Advanced Engineering Informatics*, 21(1):23–40, 2007.

[9] Wesley KG Assunção, Jacob Krüger, and Willian DF Mendonça. Variability management meets microservices: six challenges of re-engineering microservice-based webshops. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A-Volume A*, pages 1–6, 2020.

[10] Ghislain Auguste Atemezing and Florence Amardeilh. Benchmarking commercial rdf stores with publications office dataset. In *European Semantic Web Conference*, pages 379–394. Springer, 2018.

[11] Franz Baader, Diego Calvanese, Deborah McGuinness, Peter Patel-Schneider, Daniele Nardi, et al. *The description logic handbook: Theory, implementation and applications*. Cambridge university press, 2003.

[12] Armin Balalaie, Abbas Heydarnoori, Pooyan Jamshidi, Damian A. Tamburri, and Theo Lynn. Microservices migration patterns. *Software: Practice and Experience*, 48(11):2019–2042, 2018.

[13] Nina Balcan, Avrim Blum, and Yishay Mansour. Exploiting ontology structures and unlabeled data for learning. 2013.

[14] Katherine Barber. *Canadian Oxford Dictionary*. Oxford University Press, 2004.

[15] Saša Baškarada, Vivian Nguyen, and Andy Koronios. Architecting microservices: practical opportunities and challenges. *Journal of Computer Information Systems*, pages 1–9, 2018.

[16] Nick Bassiliades, Moisis Symeonidis, Panagiotis Gouvas, Efstratios Kontopoulos, Georgios Meditskos, and Ioannis Vlahavas. Paasport semantic model: An ontology for a platform-as-a-service semantically interoperable marketplace. *Data & Knowledge Engineering*, 113:81–115, 2018.

[17] David Beckett, Tim Berners-Lee, Eric Prud'hommeaux, and Gavin Carothers. Rdf 1.1 turtle. *World Wide Web Consortium*, pages 18–31, 2014.

[18] Benjamin Benni, Sébastien Mosser, Jean-Philippe Caissy, and Yann-Gaël Guéhéneuc. Can microservice-based online-retailers be used as an spl? a study of six reference architectures. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A-Volume A*, pages 1–6, 2020.

[19] Tim Berners-Lee et al. Semantic web road map, 1998.

[20] Mohinder P. S. Bhatia, Akshi Kumar, and Rohit Beniwal. Ontologies for software engineering: Past, present and future. *Indian Journal of Science and Technology*, 9, 03 2016.

[21] Justus Bogner, Jonas Fritzsch, Stefan Wagner, and Alfred Zimmermann. Microservices in industry: insights into technologies, characteristics, and software quality. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 187–195. IEEE, 2019.

[22] Dominik Bork and Hans-Georg Fill. Formal aspects of enterprise modeling methods:

a comparison framework. In *2014 47th Hawaii international conference on system sciences*, pages 3400–3409. IEEE, 2014.

[23] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-driven software engineering in practice. *Synthesis lectures on software engineering*, 3(1):1–207, 2017.

[24] William J. Brown, Raphael C. Malveau, Hays W. McCormick, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. ITPro collection. Wiley, 1998.

[25] Antonio Bucchiarone, Kemal Soysal, and Claudio Guidi. A model-driven approach towards automatic migration to microservices. In *International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, pages 15–36. Springer, 2019.

[26] Andrew Butterfield, Gerard Ekembe Ngondi, and Anne Kerr. *A Dictionary of Computer Science*. Oxford University Press, Inc., USA, 7th edition, 2016.

[27] Jean Charlet, Bruno Bachimont, and Raphaël Troncy. Ontologies pour le web sémantique. *Revue I3, numéro Hors Série Web sémantique*, pages 43–63, 2004.

[28] Jacques Chaumier. Les ontologies: Antécédents, aspects techniques et limites. *Documentaliste-Sciences de l'Information*, 44(1):81–83, 2007.

[29] Andreas Christoforou, Lambros Odysseos, and Andreas Andreou. Migration of software components to microservices: Matching and synthesis. In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 134–146, 2019.

[30] World Wide Web Consortium et al. Sparql 1.1 overview. 2013.

[31] Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest, and Clifford Stein. *Introduction à l'algorithmique*. Science sup. Dunod, Paris, 2e éd. edition, 2004.

[32] Cornelia Davis and Gene Kim. *Cloud Native Patterns: Designing Change-tolerant Software*. Manning Publications, 2019.

[33] Paolo Di Francesco, Patricia Lago, and Ivano Malavolta. Architecting with microservices: A systematic mapping study. *Journal of Systems and Software*, 150:77–97, 2019.

[34] Beniamino Di Martino, Antonio Esposito, and Giuseppina Cretella. Semantic representation of cloud patterns and services with automated reasoning to support cloud application portability. *IEEE Transactions on Cloud Computing*, 5(4):765–779, 2015.

[35] Tharam S Dillon, Elizabeth Chang, and Pornpit Wongthongtham. Ontology-based software engineering-software engineering 2.0. In *19th Australian Conference on Software Engineering (ASWEC 2008)*, pages 13–23. IEEE, 2008.

[36] Zhongli Ding and Yun Peng. A probabilistic extension to ontology language owl. In *37th Annual Hawaii International Conference on System Sciences, 2004. Proceedings of the*, pages 10–pp. IEEE, 2004.

[37] John Domingue, Dieter Fensel, and James A. Hendler. *Introduction to the Semantic Web Technologies*, pages 1–41. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[38] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. In *Present and ulterior software engineering*, pages 195–216. Springer, 2017.

[39] Hector A Duran-Limon, Carlos A Garcia-Rios, Francisco E Castillo-Barrera, and Rafael Capilla. An ontology-based product architecture derivation approach. *IEEE Transactions on Software Engineering*, 41(12):1153–1168, 2015.

[40] Sinan Eski and Feza Buzluca. An automatic extraction approach: Transition to microservices architecture from monolithic application. In *Proceedings of the 19th International Conference on Agile Software Development: Companion*, pages 1–6, 2018.

[41] Ricardo de Almeida Falbo, Fabiano Borges Ruy, Giancarlo Guizzardi, Monalessa Perini Barcellos, and João Paulo Andrade Almeida. Towards an enterprise ontology pattern language. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 323–330, 2014.

[42] Mark S Fox, Mihai Barbuceanu, and Michael Gruninger. An organisation ontology for enterprise modelling: preliminary concepts for linking structure and behaviour. In *Proceedings 4th IEEE Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE'95)*, pages 71–81. IEEE, 1995.

[43] Frederik Gailly and Geert Poels. Conceptual modeling using domain ontologies: Improving the domain-specific quality of conceptual schemas. In *Proceedings of the 10th Workshop on Domain-Specific Modeling*, pages 1–6, 2010.

[44] Fabien Gandon. An introduction to semantic web and linked data, 2014.

[45] Fabien Gandon. A survey of the first 20 years of research on semantic web and linked data. *Revue des Sciences et Technologies de l'Information-Série ISI: Ingénierie des Systèmes d'Information*, 2018.

[46] Aldo Gangemi, Nicola Guarino, Claudio Masolo, Alessandro Oltramari, and Luc

Schneider. Sweetening ontologies with dolce. In *International Conference on Knowledge Engineering and Knowledge Management*, pages 166–181. Springer, 2002.

[47] Aldo Gangemi and Valentina Presutti. Ontology design patterns. In *Handbook on ontologies*, pages 221–243. Springer, 2009.

[48] Martin Garriga. Towards a taxonomy of microservices architectures. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10729 LNCS:203–218, 2018.

[49] Javier Garzas and Mario Piattini. An ontology for microarchitectural design knowledge. *IEEE software*, 22(2):28–33, 2005.

[50] Javad Ghofrani and Daniel Lübke. Challenges of microservices architecture: A survey on the state of the practice. volume 2072, pages 1–8. CEUR-WS, 2018.

[51] Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos, and Zhe Wang. Hermit: an owl 2 reasoner. *Journal of Automated Reasoning*, 53(3):245–269, 2014.

[52] Christophe Gonzales, Lionel Torti, and Pierre-Henri Wuillemin. agrum: a graphical universal model framework. In *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, pages 171–177. Springer, 2017.

[53] Stephan Grimm and Boris Motik. Closed world reasoning in the semantic web through epistemic operators. In *OWLED*, 2005.

[54] W3C OWL Working Group. Owl 2 web ontology language document overview (second edition), 2012.

[55] W3C RDF Working Group. Resource description framework (rdf): Concepts and abstract syntax, 2004.

[56] W3C Semantic Web Best Practices & Deployment Working Group. Ontology driven architectures and potential uses of the semantic web in systems and software engineering, 2006.

[57] Thomas R Gruber. Toward principles for the design of ontologies used for knowledge sharing? *International journal of human-computer studies*, 43(5-6):907–928, 1995.

[58] Michael Grüninger and Mark S Fox. The role of competency questions in enterprise engineering. In *Benchmarking—Theory and practice*, pages 22–31. Springer, 1995.

[59] Nicola Guarino and Christopher Welty. Evaluating ontological decisions with ontoclean. *Communications of the ACM*, 45(2):61–65, 2002.

[60] Nicola Guarino and Christopher A Welty. An overview of ontoclean. In *Handbook on ontologies*, pages 151–171. Springer, 2004.

[61] Claudio Guidi, Ivan Lanese, Manuel Mazzara, and Fabrizio Montesi. Microservices: a language-based approach. In *Present and Ulterior Software Engineering*, pages 217–225. Springer, 2017.

[62] Giancarlo Guizzardi. Ontological foundations for conceptual modeling with applications. In *International Conference on Advanced Information Systems Engineering*, pages 695–696. Springer, 2012.

[63] Harry Halpin and Patrick J Hayes. When owl: sameas isn't the same: an analysis of identity links on the semantic web. In *LDOW*, 2010.

[64] Mohammad Sadegh Hamzehloui, Shamsul Sahibuddin, and Ardavan Ashabi. A study on the most prominent areas of research in microservices. *International Journal of Machine Learning and Computing*, 9(2), 2019.

[65] Hans-Jörg Happel and Stefan Seedorf. Applications of ontologies in software engineering. In *Proc. of Workshop on Sematic Web Enabled Software Engineering"(SWESE) on the ISWC*, pages 5–9. Citeseer, 2006.

[66] Hlomani Hlomani and Deborah Stacey. Approaches, methods, metrics, measures, and subjectivity in ontology evaluation: A survey. *Semantic Web and Information Systems*, 1(5):1–11, 2014.

[67] Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The even more irresistible sroiq. *Kr*, 6:57–67, 2006.

[68] Pooyan Jamshidi, Claus Pahl, Nabor C Mendonça, James Lewis, and Stefan Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 2018.

[69] Muhammad Aslam Jarwar, Sajjad Ali, Muhammad Golam Kibria, Sunil Kumar, and Ilyoung Chong. Exploiting interoperable microservices in web objects enabled internet of things. In *2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 49–54. IEEE, 2017.

[70] Christina Terese Joseph and K Chandrasekaran. Straddling the crevasse: A review of microservice software architecture foundations and recent advancements. *Software: Practice and Experience*, 49(10):1448–1484, 2019.

[71] Nima Kaviani, Bardia Mohabbati, Dragan Gasevic, and Matthias Finke. Semantic annotations of feature models for dynamic product configuration in ubiquitous environments. In *4th International Workshop on Semantic Web Enabled Software Engineering. at 7th International Semantic Web Conference*, 2008.

[72] Holger Knoche and Wilhelm Hasselbring. Drivers and barriers for microservice adoption-a survey among professionals in germany. *Enterprise Modelling and Information Systems Architectures (EMISAJ)-International Journal of Conceptual Modeling*, 14(1):1–35, 2019.

[73] Markus Krötzsch, Frantisek Simancik, and Ian Horrocks. Description logics. *IEEE Intelligent Systems*, 29(1):12–19, 2013.

[74] Dmitry Kudryavtsev and Tatiana Gavrilova. From anarchy to system: A novel classification of visual knowledge codification techniques. *Knowledge and Process Management*, 24(1):3–13, 2017.

[75] Kiran Kumar and TV Prabhakar. Pattern-oriented knowledge model for architecture design. In *Proceedings of the 17th Conference on Pattern Languages of Programs*, pages 1–21, 2010.

[76] James Lewis and Martin Fowler. Microservices: a definition of this new architectural term. *https://martinfowler.com/articles/microservices.html*, 2014 (Accessed May, 2020).

[77] Zhiyi Ma, Jinyang Liu, and Xiao He. An approach to modeling microservice solutions. In *International Conference on Information Science and Applications*, pages 533–542. Springer, 2018.

[78] Sean Marston, Zhi Li, Subhajyoti Bandyopadhyay, Juheng Zhang, and Anand Ghalsasi. Cloud computing—the business perspective. *Decision support systems*, 51(1):176–189, 2011.

[79] Manuel Mazzara, Antonio Bucchiarone, Nicola Dragoni, and Victor Rivera. Size

matters: Microservices research and applications. In *Microservices*, pages 29–42. Springer, 2020.

[80] Melinda McDaniel and Veda C Storey. Evaluating domain ontologies: clarification, classification, and challenges. *ACM Computing Surveys (CSUR)*, 52(4):1–44, 2019.

[81] Willian DF Mendonça, Wesley KG Assunção, Lucas V Estanislau, Silvia R Vergilio, and Alessandro Garcia. Towards a microservices-based product line with multi-objective evolutionary algorithms. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2020.

[82] Brian S Mitchell and Spiros Mancoridis. Comparing the decompositions produced by software clustering algorithms using similarity measurements. In *Proceedings IEEE International Conference on Software Maintenance*, pages 744–753, 2001.

[83] Karama Ali Mohamed, Marwa Salah Farhan, and Mahmoud Mohamed Ahmed Abd Elatif. Ontology-based concept maps for software engineering. In *2013 9th International Computer Engineering Conference (ICENCO)*, pages 92–97. IEEE, 2013.

[84] Gabriel Morais. Companion source code repository of the thesis evaluation chapter. `https://gitlab.com/galbuque/thesis_companion`, 2021.

[85] Gabriel Morais and Mehdi Adda. Omsac-ontology of microservices architecture concepts. In *2020 11th IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, pages 0293–0301. IEEE, 2020.

[86] Gabriel Morais, Dominik Bork, and Mehdi Adda. Companion source code repository for the use case—modelling and analysis of msa based software architectures and similarity metrics. `https://gitlab.com/galbuque/code-companion-omsac_modellingg`, 2021.

[87] Mark A. Musen. The protégé project: a look back and a look forward. *AI Matters*, 1(4):4–12, 2015.

[88] Mark A Musen et al. Ontology-oriented design and programming. *Knowledge engineering and agent technology*, 52:3–16, 2000.

[89] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Michael Amundsen. *Microservices Architecture: Aligning Principles, Practices, and Culture*. O'Reilly Media, 2016.

[90] Davide Neri, Jacopo Soldani, Olaf Zimmermann, and Antonio Brogi. Design principles, architectural smells and refactorings for microservices: a multivocal review. *SICS Software-Intensive Cyber-Physical Systems*, pages 1–13, 2019.

[91] Sam Newman. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.

[92] Daniel Oberle. How ontologies benefit enterprise applications. *Semantic Web*, 5(6):473–491, 2014.

[93] Daniel Oberle, Stephan Grimm, and Steffen Staab. An ontology for software. In *Handbook on ontologies*, pages 383–402. Springer, 2009.

[94] Felipe Osses, Gastón Márquez, and Hernán Astudillo. Exploration of academic and industrial evidence about architectural tactics and patterns in microservices. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 256–257, 2018.

[95] Mert Ozkaya and Ferhat Erata. Understanding practitioners' challenges on software modeling: A survey. *Journal of Computer Languages*, 58:100963, 2020.

[96] Claus Pahl and Pooyan Jamshidi. Microservices: A systematic mapping study. volume 1, pages 137–146. SciTePress, 2016.

[97] Claus Pahl, Pooyan Jamshidi, and Olaf Zimmermann. Architectural principles for cloud software. *ACM Transactions on Internet Technology (TOIT)*, 18(2):1–23, 2018.

[98] Salvatore F Pileggi, AA Lopez, and Ghassan Beydoun. Ontology in software engineering. In *29th Australasian Conference on Information Systems*, 2018.

[99] María Poveda-Villalón, Asunción Gómez-Pérez, and Mari Carmen Suárez-Figueroa. OOPS! (OntOlogy Pitfall Scanner!): An On-line Tool for Ontology Evaluation. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 10(2):7–34, 2014.

[100] Heinz A Preisig and Arne Tobias Elve. Graph-based modelling with distributed systems. In *Computer Aided Chemical Engineering*, volume 43, pages 241–246. Elsevier, 2018.

[101] Florian Rademacher, Jonas Sorgalla, Sabine Sachweh, and Albert Zündorf. Specific model-driven microservice development with interlinked modeling languages. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 57–5709. IEEE, 2019.

[102] Florian Rademacher, Jonas Sorgalla, Philip Nils Wizenty, Sabine Sachweh, and Albert Zündorf. Microservice architecture and model-driven development: Yet singles, soon married (?). In *Proceedings of the 19th International Conference on Agile Software Development: Companion*, pages 1–5, 2018.

[103] Anthony Ralston, Edwin D. Reilly, and David Hemmendinger. *Encyclopedia of Computer Science*. John Wiley and Sons Ltd., GBR, 2003.

[104] Dirk Riehle and Heinz Züllighoven. Understanding and using patterns in software development. *Tapos*, 2(1):3–13, 1996.

[105] Felipe L. Rosa, Roger da Silva Machado, Tiago Thompsen Primo, Adenauer Corrêa Yamin, and Ana Marilza Pernas. Análise de desempenho de ferramentas para persistência de dados ontológicos em triplas: Experimentos e resultados. In *XII Seminar on Ontology Research in Brazil*, 2019.

[106] Francisco Ruiz and José R Hilera. Using ontologies in software engineering and technology. In *Ontologies for software engineering and software technology*, pages 49–102. Springer, 2006.

[107] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson series in artificial intelligence. Pearson, 4th ed edition, 2020.

[108] Christoph Schröer, Felix Kruse, and Jorge Marx Gómez. A qualitative literature review on microservices identification approaches. In *Symposium and Summer School on Service-Oriented Computing*, pages 151–168. Springer, 2020.

[109] Evren Sirin, Michael Howard Grove, Kendall Grant Clark, and Pedro Carvalho De Oliveira. System and method for providing prediction-model-based generation of a graph data model, March 24 2020. US Patent 10,599,719.

[110] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical owl-dl reasoner. *Journal of Web Semantics*, 5(2):51–53, 2007.

[111] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146:215–232, 2018.

[112] Jonas Sorgalla, Florian Rademacher, Sabine Sachweh, and Albert Zündorf. Modeling microservice architecture: a comparative experiment towards the effectiveness of two

approaches. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 1506–1509, 2020.

[113] Barrie Sosinsky. *Cloud computing bible*, volume 762. John Wiley & Sons, 2010.

[114] Sriramkrishnan Srinivasan. *Cloud computing basics*. Springer, 2014.

[115] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Microservices anti-patterns: A taxonomy. In *Microservices*, pages 111–128. Springer, 2020.

[116] Thyago Tenório, Diego Dermeval, and Ig Ibert Bittencourt. On the use of ontology for dynamic reconfiguring software product line products. In *Proceedings of the ninth international conference on software engineering advances*, pages 545–550, 2014.

[117] Boris Marcelo Villazón Terrazas. *A method for Reusing and Re-engineering Non-ontological Resources for Building Ontologies*. PhD thesis, Universidad Politécnica de Madrid, 2011.

[118] Phil Tetlow, Jeff Z Pan, Daniel Oberle, Evan Wallace, Michael Uschold, and Elisa Kendall. Ontology driven architectures and potential uses of the semantic web in systems and software engineering. 2006, 2006.

[119] Rafik Tighilt, Manel Abdellatif, Naouel Moha, Hafedh Mili, Ghizlane El Boussaidi, Jean Privat, and Yann-Gaël Guéhéneuc. On the study of microservices antipatterns: a catalog proposal. In *Proceedings of the European Conference on Pattern Languages of Programs 2020*, pages 1–13, 2020.

[120] Geoffrey G Towell and Jude W Shavlik. Knowledge-based artificial neural networks. *Artificial intelligence*, 70(1-2):119–165, 1994.

[121] Stardog Union. Stardog studio. `https://www.stardog.com/studio/`, 2020 (Accessed April, 2020).

[122] Stardog Union. Stardog the enterprise knowledge graph platform. `https://www.stardog.com/`, 2020 (Accessed April, 2020).

[123] Michael Uschold. Ontology-driven information systems: Past, present and future. In *Proceedings of the 2008 conference on Formal Ontology in Information Systems: Proceedings of the Fifth International Conference (FOIS 2008)*, pages 3–18, 2008.

[124] Michael Uschold. Demystifying owl for the enterprise. *Synthesis Lectures on Semantic Web: Theory and Technology*, 8(1):i–237, 2018.

[125] Mike Uschold and Michael Gruninger. Ontologies: principles, methods and applications. *The Knowledge Engineering Review*, 11(2):93–136, 1996.

[126] Reinout Van Rees. Clarity in the usage of the terms ontology, taxonomy and classification. *CIB REPORT*, 284(432):1–8, 2003.

[127] Michaël Verdonck, Frederik Gailly, Sergio de Cesare, and Geert Poels. Ontology-driven conceptual modeling: A systematic literature mapping and review. *Applied Ontology*, 10(3-4):197–227, 2015.

[128] Michaël Verdonck, Frederik Gailly, Robert Pergl, Giancarlo Guizzardi, Beatriz Martins, and Oscar Pastor. Comparing traditional conceptual modeling with ontology-driven conceptual modeling: An empirical study. *Information Systems*, 81:92–103, 2019.

[129] Aad Versteden and Erika Pauwels. State-of-the-art web applications using microservices and linked data. volume 1629. CEUR-WS, 2016.

[130] Aad Versteden, Erika Pauwels, and Agis Papantoniou. An ecosystem of user-facing microservices supported by semantic models. *USEWOD-PROFILES@ ESWC*, 1362:12–21, 2015.

[131] Boris Villazón-Terrazas, Mari Carmen Suárez-Figueroa, and Asunción Gómez-Pérez. Pattern for re-engineering a classification scheme, which follows the adjacency list data model, to a taxonomy. In *Proceedings of the 2009 International Conference on Ontology Patterns-Volume 516*, pages 116–119. CEUR-WS. org, 2009.

[132] Boris Carmen Villazón-Terrazas, Mari Suárez-Figueroa, and Asunción Gómez-Pérez. A pattern-based method for re-engineering non-ontological resources into ontologies. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 6(4):27–63, 2010.

[133] Laura Von Rueden, Sebastian Mayer, Katharina Beckh, Bogdan Georgiev, Sven Giesselbach, Raoul Heese, Birgit Kirsch, Julius Pfrommer, Annika Pick, Rajkumar Ramamurthy, et al. Informed machine learning–a taxonomy and survey of integrating knowledge into learning systems. *arXiv*, pages arXiv–1903, 2019.

[134] Zhitao Wan and Ping Wang. A survey and taxonomy of cloud migration. In *2014 International Conference on Service Sciences*, pages 175–180. IEEE, 2014.

[135] Merriam Webster. Merriam-webster online. `https://www.merriam-webster.com/`, 2020 (Accessed July, 2020).

[136] Xander Wilcke, Peter Bloem, and Victor De Boer. The knowledge graph as the default data model for learning on heterogeneous knowledge. *Data Science*, 1(1-2):39–57, 2017.

[137] Lan Yang, Kathryn Cormican, and Ming Yu. Ontology-based systems engineering: A state-of-the-art review. *Computers in Industry*, 111:148–171, 2019.

[138] Stephen S. Yau and Jeffery J Tsai. Knowledge representation of software component

interconnection information for large-scale software modifications. *IEEE Transactions on Software Engineering*, (3):355–361, 1987.

[139] Lamia Youseff, Maria Butrico, and Dilma Da Silva. Toward a unified ontology of cloud computing. In *2008 Grid Computing Environments Workshop*, pages 1–10. IEEE, 2008.

[140] Eric Yuan. Towards ontology-based software architecture representations. In *2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*, pages 21–27. IEEE, 2017.

[141] Lamia Abo Zaid, Frederic Kleinermann, and Olga De Troyer. Applying semantic web technology to feature modeling. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1252–1256, 2009.

[142] Olaf Zimmermann. Microservices tenets. *Computer Science-Research and Development*, 32(3-4):301–310, 2017.

APPENDIX A

GRAPHICAL REPRESENTATION OF OMSAC

This appendix contains a graphical representation of the OMSAC TBox. Chapter 5 discusses its components and relations.

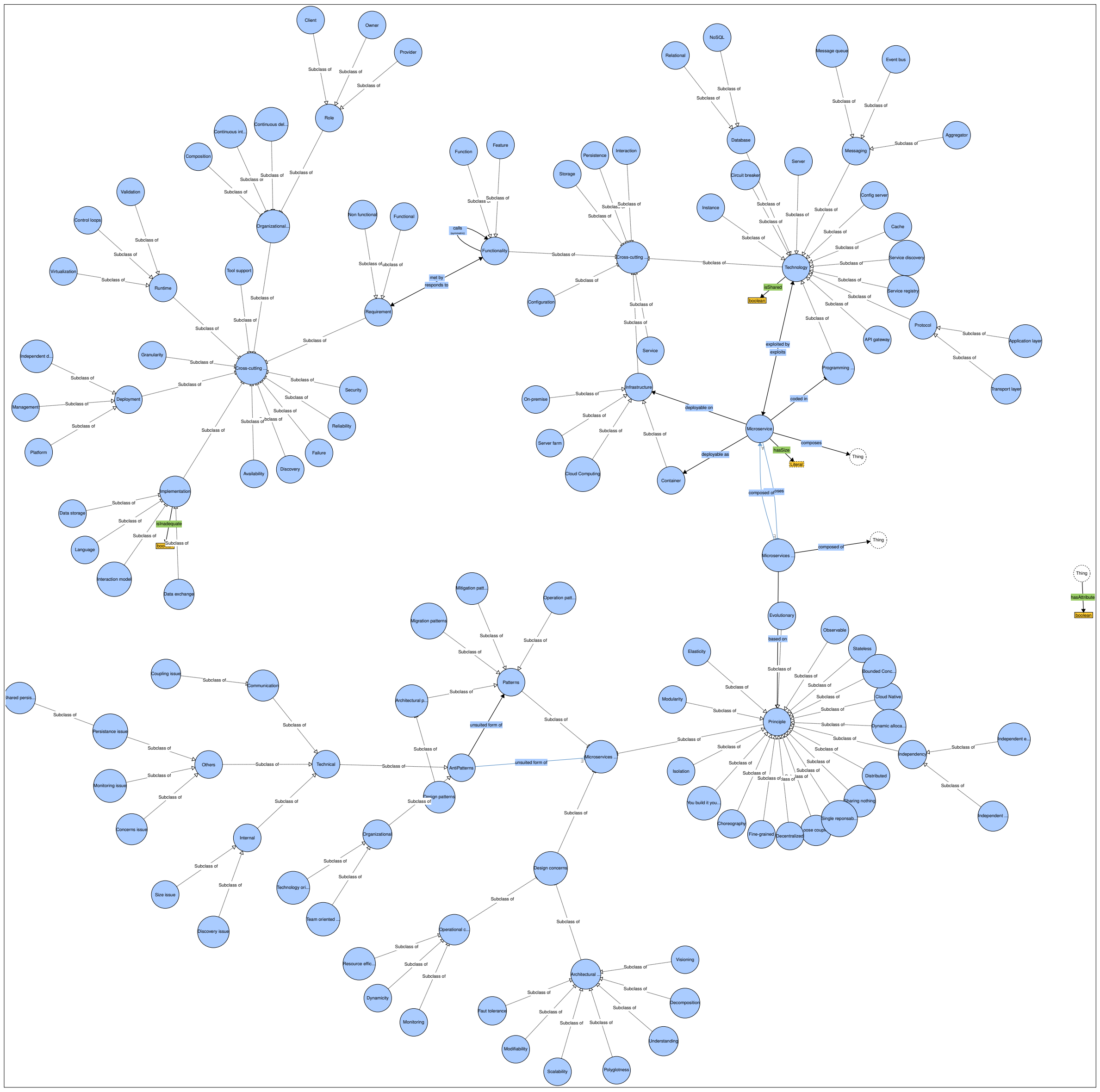


Figure A.1: Complete OMSAC's graphical representation.

APPENDIX B

MICROSERVICES-BASED SYSTEMS COLLECTED DATA

This appendix contains the data collected from the three microservices-based systems used in the use case presented in Chapter 6. The data was organized in different columns in order to unveil functional and technical aspects of these systems, as well as the dependencies and relationships between these systems.

System	Microservice	Features	Coded In	Services Used	Interactions (a)	Coevolution	Communication Technology
Hipster Shop	Ad Service	Get ads Get random ads Get ads by category	Java				gRPC
	Cart Service	Create cart Add item / update cart Empty cart Get cart	C#	Cache (Redis)			gRPC
	Checkout Service	Sum	Go		Cart Service Currency Service Email Service Payment Service Product Catalog Service Shipping Service	Shipping Service Product Catalog Service	gRPC
	Currency Service	Get supported currencies Convert currencies	JavaScript				gRPC
	Email Service	Send order confirmation	Python				gRPC

Table B.1 continued from previous page

System	Microservice	Features	Coded In	Services Used	Interactions (a)	Coevolution	Communication Technology
	Payment Service	Charge Charge Mastercard Charge Visa	JavaScript				gRPC
	Product Catalog Service	List products Get product Search products	Go			Checkout Service Shipping Service	gRPC
	Recommendation Service	List recommendations	Python		Product Catalog Service		gRPC
	Shipping Service	Get quote Ship Order	Go			Product Catalog Service Checkout Service	

Table B.1 continued from previous page

System	Microservice	Features	Coded In	Services Used	Interactions (a)	Coevolution	Communication Technology
	Front-end Service	HTTP Server Generate session ID	Go		Ads Service Cart Service Checkout Service Currency Service Product Catalog Service Recommendation Service Shipping Service		gRPC / HTTP
eShop on Containers	Basket Microservice	Create cart Get cart Add item / update cart Checkout cart	C#	Cache (Redis)	Catalog Microservice Ordering Microservice Identity Microservice	Catalog Microservice Ordering Microservice	Event Bus API Gateway (gRPC/REST)
	Catalog Microservice	List products Get products	C#	SQL Server		Ordering Microservice Basket Microservice	Event Bus API Gateway (gRPC/REST)

Table B.1 continued from previous page

System	Microservice	Features	Coded In	Services Used	Interactions (a)	Coevolution	Communication Technology
	Identity Microservice	Get a user	C#	SQL Server			Event Bus API Gateway (gRPC/REST)
	Location Microservice	Get user location	C#	Mongo Database			Event Bus API Gateway (gRPC/REST)
	Marketing Microservice	Create ads campaign Get ads Get ads by user Update ads campaign Delete ads campaign	C#	Cosmos Database Mongo Database SQL Database	Identity Microservice Location Microservice		Event Bus API Gateway (gRPC/REST)
	Ordering Microservice	Create order Get order by id / by user Cancel order	C#	SQL Server	Identity Microservice	Catalog Microservice Basket Microservice Identity Microservice	Event Bus API Gateway (gRPC/REST)

Table B.1 continued from previous page

System	Microservice	Features	Coded In	Services Used	Interactions (a)	Coevolution	Communication Technology
	Payment Microservice	Add payment Get payment record	C#				Event Bus API Gateway (gRPC/REST)
Vert.X	Inventory Microservice	Get number in stock Increase number in stock Decrease number in stock	Java	Config Server Cache (Redis)		Product Microservice	RPC REST
	Order Microservice	Create order Get order by id/ by user	Java	MySQL Database	Shopping Cart Microservice Inventory Microservice	Product Microservice User Account Microservice	RPC REST
	Payment Microservice	Add payment Get payment record	Java				RPC REST

Table B.1 continued from previous page

System	Microservice	Features	Coded In	Services Used	Interactions (a)	Coevolution	Communication Technology
	Product Microservice	List products Get product Add product Get product price Get product by page Delete product Delete all products	Java	MySQL Database		Inventory Microservice Order Microservice Store Microservice User Account Microservice	RPC REST
	Store Microservice	Add online store Delete online store Get online store	Java	Mongo Database		Product Microservice	RPC REST
	Shopping Cart Microservice	Create cart Add item / update item Remove item Get cart	Java	MySQL Database	Product Microservice Inventory Microservice		RPC REST
	Shopping UI SPA	User interface	Java				RPC REST

Table B.1 continued from previous page

System	Microservice	Features	Coded In	Services Used	Interactions (a)	Coevolution	Communication Technology
	User Account Microservice	Get a user Update user Get all users Register a user Delete a customer	Java	MySQL Database		Product Microservice Order Microservice	RPC REST

Table B.1: Extraction of the features and relevant information from microservices. (a) Other services called by the microservice.

APPENDIX C

PROOF OF CONCEPT-BUILDING A BAYESIAN NETWORKING USING OMSAC'S TBOX

This appendix presents a proof of concept conducted by us to verify the feasibility of building Bayesian Networks using the terminological component (TBox) of OMSAC. The intent behind this proof of concept was to evaluate if it would be possible to create intelligent microservices mechanisms based only on the logical description of the domain. Such a Bayesian Network should be able to learn without other datasets but the OMSAC's TBox.

CONCEPT

Create a Bayesian network-based classifier using OMSAC's TBox and without a knowledge base.

C.0.1 IMPLEMENTATION

The implementation process comprised the following steps:

- Extraction of the TBox classes.
- Extraction of the TBox relations.
- Extraction of the logical rules from the TBox.

- Building of the Bayesian Network.

Applied technologies:

- Protégé version 5.5
- Python version 3.8
- pyAgrum version 0.1.8
- Json from the Python Standard Library version 3.8
- Jupyter-notebook version 6.1.5

C.0.2 EXTRACTION

We completed the following tasks in the extraction step:

- Exporting the TBox into a Json file using Protégé.
- Development of a Python script using a Json Python Standard Library for extracting relations and rules from the TBox.
- Data transformation required to use pyAgrum [52].

Listing C.0.2 shows the code used to extract the domain and range of the TBox object properties, while Listing C.0.2 shows the code used to transform the sub-classes relations to be compliant with pyAgrum.

Listing C.1: Extraction of the domain and range of a TBox

```
for x in objectProp:
    objPropClass.append("C_"+ x)
    grClass.append("GR_" + x)
    d = getRelation(onto, x, "domain", '' )
    if d:
        domain = []
        for e in d:
            if ensembleClassesAnonymes.count(e) > 0:
                i = ensembleClassesAnonymes.index(e)
                domain.append(ensembleClassesAnonymesRenamed[i])
            else:
                domain.append(e)
        objPropDomain.append(domain)
    else:
        objPropDomain.append(["Thing"])
    r = getRelation(onto, x, "range", '' )
    if r:
        ranges = []
        for f in r:
            if ensembleClassesAnonymes.count(f) > 0:
                j = ensembleClassesAnonymes.index(f)
                ranges.append(ensembleClassesAnonymesRenamed[j])
            else:
                ranges.append(f)
        objPropRange.append(ranges)
    else:
        objPropRange.append(["Thing"])
```

Listing C.2: Transformation of subclass relations

```
classGraph = ""
for c in classesHierarchie:
    for d in c:
        classGraph += d[0] + "{False|True}" + "->" + d[1] + "{False|True};"
```

C.0.3 BUILDING THE BAYESIAN NETWORK

For building the Bayesian Network, we applied pyAgrum, version 0.1.8. We chose pyAgrum because it was easy to use and provided graphical representation of the network elements which helped us in the assessment process. We followed the steps below in order to build this network.

- Building terms hierarchy.
- Creating conditional probability tables (CPT) using the method proposed by Ding et al. [36]

In order to facilitate the manual assessment, we limited the data used to build the network to only the *subClassOf* relations from the TBox. Listing C.0.3 shows the code used to create the class hierarchy of the *OrganizationalAspects* class, and Listing C.0.3 shows the code used to create the CPT table containing the relations between the *OrganizationalAspects* class and the class *Role* and its subclasses.

Listing C.3: Creating the class hierarchy of the *OrganizationalAspects* class

```
sousclasses = "Role{False|True}<-OrganizationalAspects{False|True}->
Composition{False|True};ContinuousDelivery{False|True}<-
OrganizationalAspects{False|True}->ContinuousIntegration{False|True};
Client{False|True}<-Role->Owner{False|True};Role->Provider{False|True};"
disjointes = "Role->Composition->ContinuousDelivery
->ContinuousIntegration;Client->Owner->Provider;"
reseau2 = sousclasses + disjointes
```

```
bn2 = gum.BayesNet("Disjoint")

bn2 = gum.fastBN(reseau2)

gnb.showBN(bn2, size='100')
```

Listing C.4: Creation of CPT for the relationships between *OrganizationalAspects* and *Role* classes

```
    bn2.cpt('OrganizationAspects')[:] = [0.5, 0.5]

    bn2.cpt('Provider')[{'Owner': 0, 'Client': 0, 'Role':1}] = [0, 1]

    bn2.cpt('Provider')[{'Owner': 1, 'Client': 0, 'Role':1}] = [1, 0]

    bn2.cpt('Provider')[{'Owner': 0, 'Client': 1, 'Role':1}] = [1, 0]

    bn2.cpt('Provider')[{'Owner': 0, 'Client': 0, 'Role':0}] = [1, 0]

    bn2.cpt('Owner')[{'Provider': 0, 'Client': 0, 'Role':1}] = [0, 1]

    bn2.cpt('Owner')[{'Provider': 1, 'Client': 0, 'Role':1}] = [1, 0]

    bn2.cpt('Owner')[{'Provider': 0, 'Client': 1, 'Role':1}] = [1, 0]

    bn2.cpt('Owner')[{'Provider': 0, 'Client': 0, 'Role':0}] = [1, 0]

    bn2.cpt('Client')[{'Owner': 0, 'Provider': 0, 'Role':1}] = [0, 1]

    bn2.cpt('Client')[{'Owner': 1, 'Provider': 0, 'Role':1}] = [1, 0]

    bn2.cpt('Client')[{'Owner': 0, 'Provider': 1, 'Role':1}] = [1, 0]

    bn2.cpt('Client')[{'Owner': 0, 'Provider': 0, 'Role':0}] = [1, 0]
```

In the method suggested in the work of Ding et al. [36], probabilities of occurrence must be assigned for all root nodes. These probabilities either come from the training datasets, or are defined by experts. In our context, the training dataset is the terminological component of OMSAC. This terminological component is not completed concerning the distribution of the different classes in the real world. Neither experts have defined the theoretical distribution of these classes. We had therefore defined for the root nodes arbitrary probabilities of occurrence, i.e. 0.5. Thus, root nodes have a 50% chance of being in a true or false state.

C.0.4 FINDINGS

The Bayesian network was able to calculate correct probabilities when the logical constraint leads to a single option. Thus, the restrictions of the properties of an object whose domain and range are composed of a single class achieve the right probabilities, as we can see in Figure C.1a.

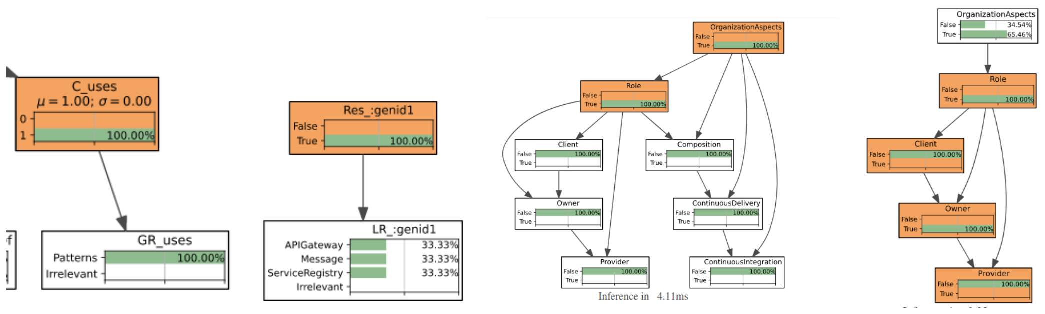
However, this network was not able to effectively identify the probabilities of an individual belonging to a given class when the possible classes are multiple, as shown in Figure C.1b, as the knowledge is limited to higher levels relationships. This was observed for the anonymous classes which represent restrictions, and the disjoint classes, as shown in Figure C.1c.

We observed the same when analyzing object properties. Indeed, the network faced the same issue with object properties having a range composed of various classes. However, when the range was composed of disjoint classes and the state (true or false) of disjointed leaf nodes were known, the calculated distribution was correct, as shown in Figure C.1d.

Thus, we noted that the definition of the domains and ranges of the restrictions present in the ontology were determining in the possibility of classifying unknown individuals. Indeed, the classification would be possible and precise if the domain and the rage were defined by a set containing a single class. When they were defined by sets containing several classes and set relations (union, intersection and disjoint), classification was only possible at the level of leaves nodes.

DISCUSSION

Bayesian Networks are efficient for describing the rules that determine the relationship between classes. This characteristic made it possible to describe the logical structure and the relations



(a) (b) (c) (d)

Figure C.1: Inferencing the OMSAC-based Bayesian Network

between the classes presented in the OMSAC TBox without using a knowledge base containing the TBox and an assertion component (A-Box).

Also, it was possible to calculate the conditional probabilities of these relations based on the description logics used to define them and on an arbitrary probability of occurrence value we set to the root nodes. However, this approach proved insufficient for the classification task. Indeed, the probabilities of occurrence being arbitrary set for the root nodes and absent for the other nodes did not make it possible to project any individual towards a given class when the identified relationships led to various possible classes.

Consequently, our experiment revealed that this Bayesian network only partially met our classification objective. On the other hand, it was shown that this network could not be generalized since the probabilities of occurrence were not based on a known universe either on practical observations.

Nevertheless, this experiment demonstrated the relevance of using formal descriptions, logical rules and conditional probabilities in the definition of classifiers since it could make it possible to create models that generalize well, provided that the probabilities of occurrence had been accurately defined.

CONCLUSION

In this experiment, it was possible to evaluate the use of machine learning techniques in both contexts of the heterogeneous dataset and the absence of a dataset. First, we started with an ontology that formally described the Microservices Architecture. Then, we tried to associate this ontology with machine learning techniques in order to create classifiers able to identify different components of a microservices architecture.

Bayesian networks and Markovian logical networks are two approaches compatible with logical descriptions. These approaches are probabilistic and therefore, the presence of a knowledge base containing enough data is mandatory for an adequate computation of the probabilities of occurrence. These probabilities are the cornerstone of the learning process of these networks.

This experiment allowed us to determine that an ontological description formalized using description logics is insufficient to build generalizable machine learning models. Indeed, to get the most of an approach for building a Bayesian network from an ontological representation, it would be necessary to know the probability of occurrence of any entity in the described domain. Unfortunately, this is not easy to establish in a heterogeneous data context without using an extensive knowledge base. Therefore, a reliable knowledge base is required to define the probability of occurrence of a random individual belonging to a given class.

To conclude, this experiment demonstrated that it is challenging to create machine learning models that generalize well in the absence of training data or exhaustive prior knowledge of the domain of interest. Consequently, machine learning techniques using ontologies without a knowledge base (TBox plus ABox) do not seem adequate. However, in the presence of a substantial knowledge base or exhaustive domain knowledge, the formalism of an ontology can be used to improve the accuracy of Bayesian networks.

Accordingly, exploring methods that would define probabilities of occurrence from limited knowledge seems an interesting perspective to enrich ontologies and make them compatible with probabilistic learning. Once the probabilities of occurrence challenge handled, it could be possible to create Bayesian networks able to make realistic projections using the restrictions presented in a T-Box.