Bruno Borlini Duarte

An Ontology-based Reference Model for the Software Systems Domain with a focus on Requirements Traceability

Vitória, ES

2022

Bruno Borlini Duarte

An Ontology-based Reference Model for the Software Systems Domain with a focus on Requirements Traceability

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Doutor em Ciência da Computação.

Universidade Federal do Espírito Santo – UFES

Centro Tecnológico

Programa de Pós-Graduação em Informática

Supervisor: Prof. Dr. Vítor E. Silva Souza

Co-supervisor: Prof. Dr. Giancarlo Guizzardi

Vitória, ES

2022

Bruno Borlini Duarte

An Ontology-based Reference Model for the Software Systems Domain with a focus on Requirements Traceability/ Bruno Borlini Duarte. – Vitória, ES, 2022-

149 p. : il.; 30 cm.

Supervisor: Prof. Dr. Vítor E. Silva Souza

Tese de Doutorado – Universidade Federal do Espírito Santo – UFES
Centro Tecnológico
Programa de Pós-Graduação em Informática, 2022.

1. Software Systems 2. Ontologies. I. Souza, Vítor Estêvão Silva. II. Universidade Federal do Espírito Santo. IV. An Ontology-based Reference Model for the Software Systems Domain with a focus on Requirements Traceability

CDU 02:141:005.7

Bruno Borlini Duarte

An Ontology-based Reference Model for the Software Systems Domain with a focus on Requirements Traceability

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Doutor em Ciência da Computação.

Trabalho aprovado. Vitória, ES, 29 de Abril de 2022:

Vitória, ES
2022

Para minha Mãe, que sempre viveu por mim, me dando todo suporte possível. Devo essa e qualquer outra conquista que eu venha a alcançar nessa vida a ela.

Acknowledgements

Agradeço a Deus por todas as bençãos e por ter tido a oportunidade de fazer mestrado e doutorado.

Agradeço primeiramente a minha mãe Celina e a minha tia Lourdes por todo amor incondicional e por todo suporte que me deram durante toda minha vida. Sem as duas eu simplesmente não estaria aqui hoje.

Agradeço a minha irmã Karina e a minha namorada Aline por sempre estarem ao meu lado me dando apoio e me ajudando.

Agradeço a meu orientador Vítor e a meu coorientador Giancarlo por todo conhecimento que me proporcionaram durante essa fase da minha vida.

Agradeço aos professores da graduação, do mestrado e agora do doutorado por tudo que me ensinaram nesses longos anos, em especial ao professor Ricardo de Almeida Falbo, que nos deixou cedo demais.

Agradeço a minha amiga e colega de mestrado Beatriz por sempre ter me ajudado e tirado minhas dúvidas quando precisei. Eu não teria conseguido sem ela.

Abstract

Software plays an essential role in modern society, as it has become indispensable in many aspects of our lives, such as social, business and even personal. Because of this importance, many researchers are dedicated to study the nature of software, how it is related to us and how it is able to change aspects in our society. It is accepted by the scientific community that software is a complex social artifact. This notion comes from the fact that a modern software system can be understood as the combination of interacting elements that exist inside a computer, such as programs and data, and in our world, such as sensors, other systems or even people, all of which are specifically organized to provide a set of functionalities or services and so, fulfill its purposes.

A major concern in the development of modern complex software-based systems, is ensuring that the design of the system is capable of satisfying the current set of requirements. In this context, it is widely accepted in the scientific literature and in international standards that the requirements have an important role in the software process. Because of that, requirements need to be developed, refined, managed and traced to their origins, in a controlled engineering process, to control their changing nature and mitigate risks. In order to support these activities, we argue, based on the conceptual modeling scientific literature, that we can use ontologies to provide a better understanding of the software systems domain, reducing the inherent complexity and improving the requirements engineering process.

In this work, we propose an ontology-based requirements traceability theory centered in different types of software systems requirements. Based on that, we developed the Reference Ontology of Software Systems (ROSS) and the Ontology of Software Defects Errors and Failures (OSDEF). ROSS and OSDEF are domain ontologies about the software systems that are intended to be used together and combined with other existing ontologies, as reference models for requirements traceability. Besides, we developed machine-readable operational ontologies, based on the reference versions of ROSS and OSDEF. The operational ontologies are created to support an ontology-based requirements traceability process that is based on the relationships that exist between the concepts in the ontologies.

Keywords: Software Systems, Software Requirements, Requirements Traceability, Ontologies, ROSS, OSDEF, UFO.

Resumo

Sistemas de Software desempenham um papel essencial na sociedade moderna, pois eles se tornaram indispensáveis em vários aspectos de nossas vidas: sociais, empresariais e até pessoais. Por conta dessa relevância do software para a sociedade, muitos pesquisadores se dedicam a estudar a natureza do software, como ele se relaciona conosco e como é capaz de mudar aspectos em nosso mundo. É aceito pela comunidade científica que o software é um artefato social complexo. Essa noção vem do fato de que um sistema de software moderno pode ser entendido como a combinação de elementos que interagem entre si, sendo que parte deles existem dentro de um computador, como programas e os dados, enquanto a outra parte existe fisicamente em nosso mundo, como sensores, componentes mecânicos ou mesmo pessoas, todos os quais são especificamente organizados para fornecer um conjunto de funcionalidades ou serviços e, assim, cumprir seus propósitos.

Uma grande preocupação no desenvolvimento de sistemas modernos e complexos baseados em software, é garantir que o projeto do sistema seja capaz de satisfazer o conjunto atual de requisitos. Nesse contexto, é amplamente aceito na literatura científica e em padrões internacionais que os requisitos de um sistema de software têm um papel crucial durante seu ciclo de vida e por isso precisam ser desenvolvidos, refinados, gerenciados e rastreados até suas origens, em um processo de engenharia controlado, a Engenharia de Requisitos, para controlar sua natureza mutável e mitigar riscos ao desenvolvimento do sistema de software. Para dar suporte a essas atividades, baseados na literatura científica de modelagem conceitual, nós propomos a utilização de ontologias de domínio, como modelos para um melhor entendimento do domínio de sistemas de software, reduzindo a complexidade inerente e melhorando o processo de Engenharia de Requisitos.

Neste trabalho, nós propomos um método para utilização de ontologias de domínio como ferramentas para rastreabilidade de requisitos de software centrado na definição de diferentes tipos de requisitos de sistemas de software. Nós desenvolvemos a Ontologia de Sistemas de Software (ROSS) e a Ontologia de Defeitos, Erros e Falhas (OSDEF). ROSS e OSDEF são ontologias de domínio sobre os sistemas de software que se destinam a serem usadas em conjunto e combinadas com outras ontologias existentes, como modelos de referência para rastreabilidade de requisitos. Além disso, desenvolvemos ontologias operacionais legíveis por máquina, baseadas nas versões de referência do ROSS e OSDEF. As ontologias operacionais são criadas para dar suporte a um processo de rastreabilidade de requisitos baseado em ontologias que é baseado nas relações que existem entre os conceitos nas ontologias.

Palavras-chave: Sistemas de Software, Requisitos de Software, Rastreabilidade de Requisitos, Ontologias, ROSS, OSDEF, UFO.

List of Figures

List of Tables

List of abbreviations and acronyms

UFO	Unified Foundational Ontology
ROSS	Reference Ontology of Software Systems
OSDEF	Ontology of Software Defect Errors and Failures
SABiO	Systematic Approach for building Ontologies
CQ	Competency Question
BREQ	Business Requirement
STREQ	Stakeholder Requirement
SYSREQ	System Requirement
PROGREQ	Program Requirement
WWW	World Wide Web
HTML	HyperText Markup Language
XML	eXtensible Markup Language
SPARQL	SPARQL Protocol and RDF Query Language
RDF	Resource Description Framework
OWL	Ontology Web Language

Contents

1 Introduction

In this chapter we discuss the context in which Software Systems are inserted in our lives, as we, as a society, become more dependent on them. We briefly examine their properties and how researchers and practitioners have been working to better understand, develop and manage such systems in the best possible way to support us. This context and the possibilities of research around it are discussed as the motivation for the development of this thesis. The chapter also presents the main objectives of the thesis, an overview of the research method that was adopted, the related publications and the organization of the manuscript.

1.1 Context and Motivation

Software plays an essential role in modern society, as it has become indispensable in many aspects of our lives, such as social, business and even personal. Because of this importance, many researchers are dedicated to study the nature of software and how it is related to us, changing aspects in our world. It is accepted by the scientific community that software is a complex social artifact (IRMAK, 2013; WANG et al., 2014a). This notion comes from the fact that a modern software system can be understood as the combination of interacting elements that exist inside a computer, such as programs and data, and in our world, such as sensors, other systems or even people. These system elements are specifically organized to provide a set of functionalities or services and so, fulfill its purposes (ISO, IEC, 2017b; BOURQUE; FAIRLEY et al., 2014) as components of the software system.

Software Systems are computer-based artifacts, they are capable of existing through time, being replicated many times and having dozens of different versions, while still maintaining their identity (WANG et al., 2014b). A classic example of theses properties can be observed in Microsoft Windows, an operating system that has been created over 30 years ago, received many updates and was released under many different versions, but still maintains its identity as Microsoft's operating system.

Despite their special properties, software systems are still artifacts, they can inherit defects and are susceptible to failures that can range from having a small impact to being so critical that may cause huge material and social losses. In other words, one can say that software systems have a significant value in our society as we are heavily dependent on them. However, as their value grows, the risks involved in their creation and maintenance also grow (SALES et al., 2018). Because of this high-value/high-risk characteristic, many researchers have dedicated their works to better understand and to better represent the software domain and the artifacts that are part of it.

Because of that, planning, building and maintaining complex/critical software systems is not a trivial task, as software is not, by any means, a static artifact. Software Systems change and evolve during their operation cycle (BOURQUE; FAIRLEY et al., 2014); complexity and criticality of Software Systems are constantly growing (CHENG; ATLEE, 2007); new functionalities and features are inserted at the same pace that obsolete ones are removed; during a software development process (SDP) or even during system operation, new requirements will eventually emerge, in different levels of abstraction, from different sources and the existing ones will need to be revised, which also adds variability to the domain. In other words, regardless of the reasons, requirements inevitably change (ISO, 2018) and as they are the foundation of a software system, the latter also change in consequence.

Further, one of the most important properties of software systems is it inherent complexity, because of the heterogeneity of their components, and the unpredictability and openness of their environments. In fact, the success of a software system heavily depends on how well it is capable to keep fulling its requirements while existing in a changing environment (CHENG; ATLEE, 2007). Besides, requirements are not simple entities, in the sense that they exist in different levels of granularity and refinement. In this context, it is widely accepted in the scientific literature and in international standards that requirements need to be developed, refined and managed in a controlled engineering process, to control their changing nature, mitigating risks and to allow for the ability to trace back the relations between them, their origins and to other products of the software system domain (ISO, 2018; BOURQUE; FAIRLEY et al., 2014; GOTEL; FINKELSTEIN, 1994; RAMESH, 1998; KANNENBERG; SAIEDIAN, 2009). Requirements Traceability emerges a supportive process to manage the relationships that exist around software system requirements.[1]

In one of the oldest definitions presented in the literature, requirements traceability (RT) is defined as *"the ability to follow the life of a requirement, in both a forwards and backwards direction"* (GOTEL; FINKELSTEIN, 1994). For example, a very common case of requirements traceability within the software systems context happens between test cases and the requirements which they are intended to test.

Traceability between requirements and different types of software artifacts plays a major role in system life-cycle, supporting activities such as system validation, change impact analysis, project visibility and regulation compliance (NAIR; VARA; SEN, 2013) and it is considered a necessary activity for software quality, even for organizations with early stages of maturity (CMMI Institute, 2018).

Despite the benefits mentioned above, unfortunately, many organizations fail to implement proper requirements traceability in their software projects and/or products. In

[1] Requirements Traceability is discussed in Section 2.2.

many cases a requirements traceability matrix (which is one of the main components in traceability analysis) is created, but as the software evolves, the requirements traceability activity is left aside in a phenomenon called *traceability decay* (MÄDER; GOTEL, 2012).

Traceability decay and other challenges, like poor tool support, (TUFAIL et al., 2017) that are directly related of the *Requirements traceability problem* (GOTEL; FINKELSTEIN, 1994) have been studied and discussed in the literature by practitioners and researches over the decades. Many proposals and techniques to support and improve the performance of the requirements traceability process have emerged in the literature. One of the most promising approaches suggests the utilization of reference models for supporting requirements traceability. The purpose of a such models is to reduce the task of creating application-specific models over a problem domain (RAMESH; JARKE, 2001). In other words, the model is intended to represent/cover the intended domain in the best way possible. The user of the model selects relevant parts of the reference model and adapts them to the problem at hand, in order to configure solution that solves the problem. However, to our knowledge, most of the models that exist in the current literature,[2] are usually incomplete, in the sense that they do not cover the whole software domain, not considering important artifacts and entities that exist inside the software domain. Besides, most models consider software requirements in a simplistic way, as a concept that is only important during the development of the software system, and becomes less important as the system enters in production and as mentioned earlier in the chapter, this notion is considered wrong and outdated by modern standards and capability models, that consider the existence of many types of requirements.

In this context, an approach for requirements traceability based on the utilization of reference models needs to fulfill three requirements: (i) cover the software domain in the best possible way; (ii) consider the existence of software requirements with different levels of granularity and that enforce their importance over the whole software process, not only during development; and (iii) be based on well-founded and tested conceptual modeling theories for the development of the proposed reference model.

In order to pursue (i) and (ii) we need to understand the nature of software systems, the importance of requirements as the foundation of the software process, their representation as specifications and their relations with other software artifacts In their well-know research, Pamela Zave and Michael Jackson (ZAVE, 1995; ZAVE; JACKSON, 1997; GUNTER et al., 2000) extensively discussed the subject and proposed an information framework and a reference model about Requirements Engineering (RE) process and its main artifacts: Requirements (R), Assumptions (A), Specifications (S), Programs (P) and the Machine (M) where the system exists. The reference model of RE was created to provide a solid theory about RE and is widely known and accepted by the RE community.

[2] These traceability models mentioned are discussed in Section 7.5

Based on Zave and Jackson's work, Wang et al. (WANG et al., 2014b; WANG et al., 2016) presented the Ontology of Software Artifacts (OSA). OSA defines distinctions between a set of types of existing artifacts that make up our notion of *software*, in its different perspectives (computational and social) and aims to explain the ontological nature of these artifacts in the light of Zave and Jackson's model.

However, although Wang's work presents an ontological analysis of the (Software) Artifacts that are usually associated to the general conceptualization of *software* (e.g Code, Program and Software System) and their properties, it does not focus on the artifacts that are part of the software process.[3] The Software System process is composed by many other concepts that were not discussed by Wang and his colleagues, specially when we look to Software Systems as first-class citizens for Organizations strategic plans. Besides, Wang's analysis is heavily focused on the further development of the concept of Assumption and does not improve over the concept of Requirement as it represents the capabilities of a Software System and the benchmark of its success or failure.

Due to that, we believe that Zave and Jackson's reference model can be further improved by considering other (Software) Artifacts that are part of the software process, but that were not discussed by them or by Wang, in their respective works. In a simplistic way, we intend to do that by adopting and reusing, into our work, concepts of ontologies that conceptualize about the software systems domain.[4] Furthermore, the concept of Requirement needs to be further developed, not only as a high-level goal, as presented in OSA, but as an essential Information Item that exists through the software process and that can exist in many levels of abstraction.

For (iii), we consider the following information that was raised so far: (a) the previous works of Zave & Jackson and Wang; (b) the Software Systems domain is a complex, multi-agent, multi-artifact and multi-phase domain; (c) Software Systems are heterogeneous artifacts, composed by elements with distinct natures that are strongly related with each other and (d) the vital role of requirements in it (ISO, 2018; ISO, 2017; BOURQUE; FAIRLEY et al., 2014). We argue that, based on the considerations listed above, we can use reference ontologies (GUARINO, 1998; GUIZZARDI, 2007) to provide a better understanding of the software systems domain, reducing the complexity and improving the reference models originally proposed by Zave & Jackson and Wang. Reference Ontologies are widely accepted in the literature as knowledge-supporting tools to provide semantics for complex domains, acting as reference models and solving communication problems and ambiguities between entities of a domain. Besides, machine readable operational ontologies (GUIZZARDI, 2007), based on well-founded reference models, are capable to

[3] Concepts that are part of ROSS and OSDEF will appear highlighted in Sans Serif throughout the text of this thesis.

[4] This subject is further discussed in chapters 4, 5 and 7.

support requirements traceability and ontology-based reasoning on the domain of study.[5]

These requirements for our research were raised based on the conclusions and research directions presented by the works of Zave, Jackson, Gunter, van Lamsweerde and Wang, in addition with the knowledge about reference models for requirements traceability presented in the related literature. More precisely, in his PhD thesis, Wang (2016) suggested that the reference models produced by Zave & Jackson and by himself could be further improved to provide a model requirements traceability. Due to that, we started our research in the scientific literature and in international standards to understand the requirements for such reference model. Furthermore, the intention of adopting domain ontologies about the software process (the ontologies of SEON) and proceeding with our previous research, but focusing in requirements traceability, was in our plans since the very beginning of this research.

1.2 Research Hypothesis

Considering, as previously mentioned, that:

- Software Systems are complex social artifacts, that are deeply inserted and needed in modern society;
- Software Requirements are the foundation of any software system;
- Requirements Traceability is considered a mandatory policy by many international standards on software systems; Although it is hard to implement and to maintain, as it tends to decay as the software system evolves through its operation.
- Ontologies have been extensively and successfully used as support tools for knowledge representation, knowledge sharing, systems interoperability and semantic-based reference models inside the software domain;

The research hypothesis of this thesis is:

Well-grounded domain ontologies can support semantic requirements traceability. Reference ontologies about the software systems domain can define the important domain concepts (entities) and the relations that exist around them. Operational ontologies can be used as machine-readable assets that provide the support for querying over the software artifacts data produced during the software process.

Based on the research hypothesis presented above, the focus of this work is on the development of an ontology-based requirements traceability theory. This thesis presents

[5] Reference and Operational ontologies are discussed in Section 3.

two well-grounded ontologies about the software systems domain, the Reference Ontology of Software Systems (ROSS) and the Ontology of Software Defects Errors and Failures (OSDEF). These ontologies were created to be used as reference models for the software systems domain, representing the main artifacts that are part of that domain.

ROSS is focused on different types of requirements with distinct levels of granularity and on other artifacts that are related to them. OSDEF is focused on representing the concepts of *failures* and *defects* that exist in the software process and that are directly related to software change/evolution. Moreover, it is important to explain that, although ROSS and OSDEF are complete ontologies on their own (they are not sub-ontologies of another ontology), they are not capable to cover the entire software domain on their own, because of its size and complexity. That is true especially for OSDEF, that was created to conceptualize about a very specific part of the software systems domain.

Due to that, ROSS and OSDEF were designed to be integrated with other software-related ontologies that are part of the Software Engineering Ontology Network (SEON) (RUY et al., 2016), in order to represent other types of software artifacts.[6] In fact, our reference model and our approach for requirements traceability, which are discussed in Chapter 7, are based on reusing concepts from ontologies that conceptualize about parts of the software domain that are not discussed by ROSS or OSDEF. For example, if the users desire to retrieve traces between a set of requirements, the programs that implement them and test cases associated to those programs. they must reuse the Reference Ontology on Software Testing (ROoST) (SOUZA; FALBO; VIJAYKUMAR, 2013) together with ROSS, since ROoST is the ontology of SEON that conceptualizes about the software testing domain.

1.3 Research Objectives

The main objective of this thesis is to provide a reference model for the software systems domain that focuses particularly on requirements traceability, that allows us to trace from low-level concepts such as programs at runtime all the way to high-level concepts such as business requirements.

This general objective can be decomposed in the following specific objectives:

- Develop a set of reference ontologies that are able to properly represent and cover the software system domain, by proposing new reference models and reusing existing ones;

[6] Ontologies are intended to be (re)used together and associated, based on their concepts, in ontology networks. This concept is discussed in Chapter 3.

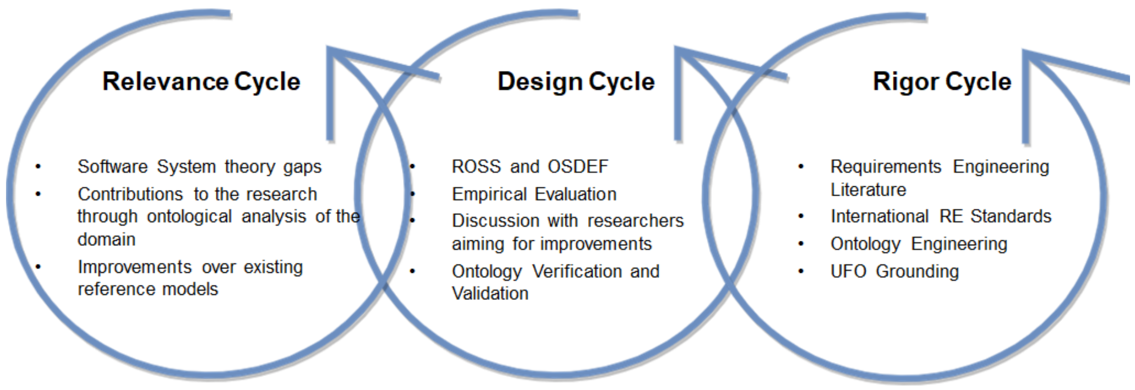

Relevance Cycle
Software System theory gaps
Contributions to the research through ontological analysis of the domain
Improvements over existing reference models
Design Cycle
ROSS and OSDEF
Empirical Evaluation
Discussion with researchers aiming for improvements
Ontology Verification and Validation
Rigor Cycle
Requirements Engineering Literature
International RE Standards
Ontology Engineering
UFO Grounding

Figure 1 – Overview of the Design Science Paradigm cycles (HEVNER, 2007) adapted for the development of this research.

- Apply the knowledge from the reference models into operational ontologies, in order to allow the implementation of ontology-based traceability;
- Demonstrate, through a proof of concept, the feasibility of a requirements traceability based on operational ontologies.

1.4 Research Method

In this work, the research method follows the Design Science Methodology (WIERINGA, 2014). A Design Science research intends to improve the state-of-the-art through the introduction of new and innovative artifacts and the process for creating these artifacts (HEVNER, 2007). The method considers three closely related activity cycles: Relevance, Design and Rigor.

The *Relevance Cycle* starts the research and it defines the problems to be addressed/opportunities, the research requirements and the criteria for evaluating the results. The *Rigor Cycle* refers to the use and generation of knowledge by connecting the design science activities to the scientific knowledge base, experience and expertise of the participants in a research project. Accuracy is achieved through the proper application of existing fundamentals and methodologies (HEVNER, 2007). Finally, for the central part of the paradigm, the *Design Cycle* refers to the development and evaluation of artifacts or theories to solve the identified problems. It can be considered the heart of design science as it is where most of the work in this paradigm is done. Moreover, it draws the requirements for the research as inputs from the *Relevance Cycle* and the theories and methods for design and evaluation of the artifacts produced from the *Rigor Cycle*.

Figure 1 summarizes the discussion about the design, rigor, and relevance cycles, and highlights the main elements of each cycle in the context of this thesis.

Regarding the Relevance Cycle, the research opportunity was to work on the development of an ontology-based requirements traceability theory that is based on well founded domain ontologies about the software domain. The problem, perceived during a related research about requirements management is that requirements traceability, as a support process, has a considerable positive impact on the development of software systems. However, it is hard to implement and even more to maintain it, as the software system changes in its existence. Additionally, many proposals that exist in the literature about "semantic requirements traceability" are based on models that adopt very little formalism.

In order to achieve the objectives presented in the previous section and to develop de domain ontologies[7] necessary for this work following the rigor required for the Design Science paradigm, the domain ontologies proposed in this thesis are created and evaluated based on the ontology engineering method SABiO, Systematic Approach to Build Ontologies (FALBO, 2014). SABiO was chosen as the ontology engineering method for this research because: (i) it was created specifically for the development of domain ontologies; (ii) it has been successfully applied in the development of ontologies in the Software Engineering, and Requirements Engineering area, which the main research field of this thesis; and (iii) because it acknowledges the importance of utilization of foundation ontologies as base for the creation of domain ontologies. SABiO is properly presented and explained in Section 3.4.

Furthermore, the knowledge base adopted is based on the scientific literature and in international standards and capability models that are widely accepted and used by practitioners around the world.

More precisely, the part of the knowledge related to requirements traceability used for the development of this research is based on the results of two systematic literature reviews (SLR) (KITCHENHAM; CHARTERS, 2007) about requirements traceability. The first one was conducted by Nair, Vara and Sen (2013) and the second one by Tufail et al. (2017). Mapping studies were used to obtain an overview perspective over the requirements traceability literature and to identify the sub-areas of the research domain, including the most relevant papers.[8]

The part that is related to ontologies in software engineering is directly related to previous works. This research is an extension of the work developed in (DUARTE et al., 2018), where we proposed the Software Ontology (SwO) and Reference Software Requirements Ontology (RSRO), reference ontologies about the software domain. Because of that, the research conducted for this thesis is based on the same international standards and scientific works, such as, ISO 12207 (ISO, 2017), ISO 29148 (ISO, 2018), Zave

[7] Ontology definition, types and uses are discussed in Chapter 3.
[8] Both SLRs are further discussed in Section 2.2.

and Jackson's work and the Unified Foundational Ontology (UFO) (GUIZZARDI, 2005; GUIZZARDI, 2007; GUIZZARDI; FALBO; GUIZZARDI, 2008; GUIZZARDI et al., 2013).

We believe that this knowledge baseline is solid to support the research that is being conducted, as the knowledge and results generated by this research are relevant and contribute to the growth of that base.

For the Design Cycle, the main artifacts produced in this thesis are the two domain ontologies, ROSS and OSDEF, created to improve the reference models that already existed in the literature, proposed by Zave, Jackson and Wang. Additionally, ROSS and OSDEF intend to fill a conceptual gap that exists in SEON[9].

Regarding evaluation, both ontologies were evaluated through ontology verification and validation techniques proposed by SABiO. Furthermore, as a proof of concept, we also performed an empirical evaluation of the capability of ROSS and OSDEF to be used as tools for requirements traceability using data of an ATM System prototype.

1.5 Published Work

This Section presents the published works related to this thesis.

- Duarte, Bruno Borlini; Guizzardi, Giancarlo; Guizzardi, Renata; Falbo, Ricardo de Almeida; Souza, Vítor E. Silva. **Ontological foundations for software requirements with a focus on requirements at runtime**. Applied Ontology, p. 73-105, 2018.

  In this first paper, we presented the Software Ontology (SwO) and the Reference Software Requirements Ontology (RSRO) and started the discussion about the relations between Requirements and Programs, in the context of the Software Systems Domain. SwO and RSRO are discussed in Section 3.5;

- Duarte, Bruno Borlini; Guizzardi, Giancarlo; Guizzardi, Renata; Falbo, Ricardo de Almeida; Souza, Vítor E. Silva. **Towards an ontology of software defects, errors and failures**. In: International Conference on Conceptual Modeling (ER). Springer, Cham, p. 349-362, 2018.

  For the second paper we presented OSDEF and discussed Failures, Defects and Errors in the context of the software systems domain. OSDEF was created to explain the ontological nature of the concepts that are usually overloaded in the literature as "software anomalies". OSDEF draws knowledge from a well-know ontological pattern of Events presented in UFO-B to conceptualize about this very specific part of the Software Systems domain;

[9] SEON is presented in Section 3.5

- Duarte, Bruno Borlini; Guizzardi, Giancarlo; Guizzardi, Renata; Falbo, Ricardo de Almeida; Souza, Vítor E. Silva. **An ontological analysis of software system anomalies and their associated risks**. Data & Knowledge Engineering (DKE), Elsevier, v. 134, p. 101892, 2021.

  Finally, we presented ROSS, an ontology that reuses SwO and RSRO to extend Zave and Jackson's work over the importance of requirements for the software systems domain. ROSS was created to be the backbone ontology for our ontology-based requirements traceability approach.

1.6 Organization

The remainder of this thesis is structured as follows:

Chapter 2 summarizes the baseline knowledge related to this work. First, we discuss the Requirements Engineering field. After that we focus on the Requirements Traceability research.

Chapter 3 discusses the ontological foundations adopted in this work. First we focus on UFO (GUIZZARDI, 2005; GUIZZARDI, 2007; GUIZZARDI et al., 2013), the foundational ontology used to ground ROSS and OSDEF. Second, we present SABiO (FALBO, 2014), the ontology engineering method adopted for the development of both ontologies. Lastly, we present SEON (RUY et al., 2016) the Software Engineering Ontology Network that connect several domain ontologies that are directly related to our proposal.

Chapter 4 presents ROSS, the Reference Ontology of Software Systems, the first contribution of this thesis.

Chapter 5 presents OSDEF, the Ontology of Software Defects, Errors and Failures, the second contribution of this thesis.

Chapter 6 presents the Evaluation processes for both ROSS and OSDEF, based on the normative defined by SABiO.

Chapter 7 presents our approach for requirements traceability based on domain ontologies. The approach uses operational versions of both ROSS and OSDEF, that are based on the reference models presented in the previous chapters, as tools for the execution of SPARQL queries (W3C, 2013) over the data of an ATM Simulation System.

Chapter 8 presents the conclusions of this thesis, revisits its main contributions and outlines directions for future research.

2 Baseline

This chapter presents concepts and proposals in both Requirements Engineering and Requirements Traceability domains that were relevant for the development of this thesis.

The chapter is structured as follows: Section 2.1 presents an overview of the Requirements Engineering domain focused on the development of Software Systems. Section 2.2 discusses Requirements Traceability. Section 2.3 summarizes the chapter.

2.1 Requirements Engineering for Software Systems

Requirements are the foundation and the building blocks of any software system project, since they are the basic knowledge for the other phases of the software process, such as design, coding and testing (LAMSWEERDE, 2000). However, poorly specified/missing requirements are recurrently recognized as a major cause of problems in software systems projects, such as exceeded costs, failure to meet client's expectations and project delivery delays (BOURQUE; FAIRLEY et al., 2014). Besides that, software systems are constantly growing in size and in complexity, which makes the *requirements problem* (BELL; THAYER, 1976) a bigger concern and a constant research challenge (LAMSWEERDE, 2009).

Requirements Engineering (RE) is a systematic and interdisciplinary effort that mediates between the domains of the client, the stakeholder that acquires or procures a product or service, and the supplier, an organization or individual that enters into an agreement with the client for the development of a product or service (ISO, 2018). RE is extensively discussed in international standards (ISO, 2018; ISO, 2017) and in the scientific literature (ZAVE, 1995; LAMSWEERDE, 2000), that are focused on promoting a better understanding and scientific advances of the activities of discovering, eliciting, developing, analyzing, verifying (including verification methods and strategy), validating, communicating, documenting and managing requirements.

However, if based only on this definition, one can understand requirements as simple and static artifacts, that are created early in a software system development process and only used when needed, a conclusion that is far from the truth. The fact is that the term *requirement* itself is not consistent in the software industry. In some cases, a requirement is simply an abstract statement about a desire of service that a system should provide. At the other extreme, it is a detailed, formal definition of a system function (SOMMERVILLE, 2016). In other words, the term requirement is loosely used to refer to different levels of formality.

Moreover, as they are critical, requirements are also complex entities: (i) requirements can be represented in many forms and in different levels of abstraction (ISO, 2018); (ii) requirements are not independent entities since they are directly related to many other artifacts produced during the software process, including other requirements. (SOMMERVILLE, 2016); and (iii) requirements change and (need to) evolve through the existence of a Software System, as the environment around them also changes (BOURQUE; FAIRLEY et al., 2014; LAMSWEERDE, 2000). Obviously, from a project management perspective, it would be desirable to freeze the original set of requirements during the entire development process of a software system and even later during system operation, in order to avoid delays and more costs. However, in practice, this is rarely possible. This happens because requirements are not static entities as their understanding keeps evolving during the software development process (ISO, 2018). In fact, changes in the requirements are necessary for a Software System to evolve and to keep producing the desired results, since it is expected that the environment where the software system exists will suffer changes.

In other words, during the software process and operation, new requirements will emerge and old ones will need to be changed or even removed. This process is expected and it directly impacts on software system's features and functionalities, in order to keep it relevant (LEHMAN; RAMIL, 2001).

Within this context, many authors dedicated their research to discuss this *requirements problem* and to propose models, frameworks and theories to better understand requirements, their changing nature and their connection with other artifacts that are produced during the software process.

In what follows, we describe research that discussed and proposed the definition and utilization of reference models for Software Systems. The works are presented in chronological order, for better understanding, starting from Zave and Jackson's work, which is considered, by many scholars, as a landmark of the requirements engineering research field. Besides, it is important to explain that these works were adopted and used as a source of knowledge for the development of ROSS and OSDEF and thus, are a relevant background for the work that is being described here.

2.1.1 Zave and Jackson's model

In their work (JACKSON; ZAVE, 1995; ZAVE; JACKSON, 1997; GUNTER et al., 2000), Zave, Jackson and Gunter discussed four common problem areas of RE and presented a set of definitions and a formula of relevant RE elements to tackle them: $S, K \vdash R$.

The formula states that in order to fulfill a set of requirements (R), an implementable specification (S) associated with relevant domain knowledge (K) is necessary. By their definitions, a requirement (R) is a high-level prescriptive statement about a desired effect that exists only in the environment[1] where the system-of-interest exists. In other words, a requirement should contain *nothing but* information about the environment and that everything else should be understood as "implementation bias". For example, considering an ATM Software System, any type of description about how the the requirement *The ATM must be able to communicate with the bank server* should be implemented, is considered implementation bias. On the other side, the specification (S), which is derived from the requirement, is responsible for describing the behavior of the machine.[2]

Finally the domain knowledge (or domain assumptions) (K) are the elements responsible to bridge the gap between the requirements and its specifications. In this context, Zave and Jackson emphasize that although every SE project will have unique properties and different emphases, the formula and the terminology presented will work with all of them, as the relation between requirements, specifications and domain assumptions does not change. Moreover, they also emphasize that incorrect assumptions will have a serious effect on the satisfaction of the requirements, as the specification that is derived heavily depends on these assumptions. In fact, the importance of assumptions for the Requirements Engineering research and for Software Systems domain were already being discussed by Lehman in his *Principle of Software Uncertainty* (LEHMAN, 1989; LEHMAN, 1996) as a part of the theory of the *Laws of Software Evolution.*

Few years later, Gunter et al. (2000) improved the existing formula by adding two new concepts: the machine (M), as the programming platform and the program (P), as the artifact that is intended to implement the specification inside a computer, creating the WRSPM reference model. Besides that, the model further discussed the conceptualization around the environment in which the Software System exists. Figure 2 depicts the five artifacts and the portions of the world in which they exist.

This representation, and the knowledge around it, developed in previous works (JACKSON; ZAVE, 1995; ZAVE; JACKSON, 1997), was proposed to be used as a reference model and a basic framework for RE, providing a discussion about the key elements of the RE process, their main attributes and their relations. Moreover, it is important to explain that the reference model focuses on the Specification artifact, as it is in the center of the model, existing in the environment, but also having to respect the same basic properties of the program. In other words, the specification S must be created in the environment in a way that it follows a set of rules that allows it to be implemented in a program P, for a

[1] The concept of environment used by Zave and Jackson denotes a portion of the real world in which the system-of-interest will operate to produce a result.

[2] The concept of Machine originally presented refers to the computational artifact being built. Zave and Jackson decided to avoid the term "System" as they considered it too generic, in the sense that an aircraft, or even a living organism can be considered a system.

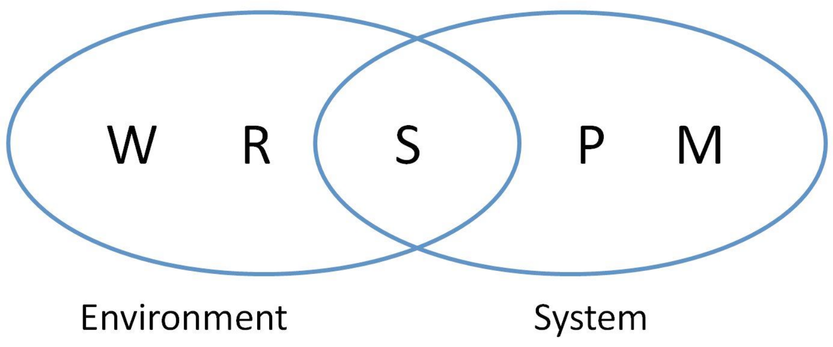


Figure 2 – Gunter, Zave and Jackson's RE reference model presenting the 5 key artifacts for the RE process (GUNTER et al., 2000). W represents the knowledge about the *World*, R represents *Requirements* of the System; S denotes the *Specification* of the System; P is *Program* that is the implementation of S for M, the *Machine*.

machine M.

2.1.2 van Lamsweerde's model

Lamsweerde (2009) extended Zave and Jackson's work by focusing on the relation between the problem world, which needs a solution, and the machine, which is created to provide it. He suggests that in order to make sure that a machine solution will correctly solve a problem, this machine should be properly anchored on the problem world,[3] which needs to be correctly delimited, structured and characterized. In order to do that, the types of statements about the world should be understood.

Figure 3 depicts van Lamsweerde's extended classification among the types of statements that exist in the problem world. An expectation is a prescriptive assumption about a specific behavior of a member/component in the problem world that the machine cannot control. For example, an operator must manually input new calculation parameters every time some change happens in the Software System's domain. In this case, a requirement can only be fulfilled if the operator inputs the new calculation parameters. Domain hypotheses are descriptive assumptions about the problem word, in the sense that they do not prescribe any expected behavior from a person/element in the problem world. Domain hypotheses are not expected to hold every time, on the other side, Domain properties are descriptive statements that are based on natural laws, that are expected to hold invariably.

Based on these new definitions, van Lamsweerde presented an extension of Zave and Jackson's original formula: $S, K \vdash R$ which was extended to: $S, K, D \vdash R$. The new formula states that the requirements will be satisfied whenever the specification is met, provided that the assumptions and domain properties hold. Compared to Zave and Jackson's work, van Lamsweerde's proposal focus on the types of statements that exist in

[3] The problem world is generally a complex organizational and technical world, grounded on rules and constraints that will directly affect the machine solution.

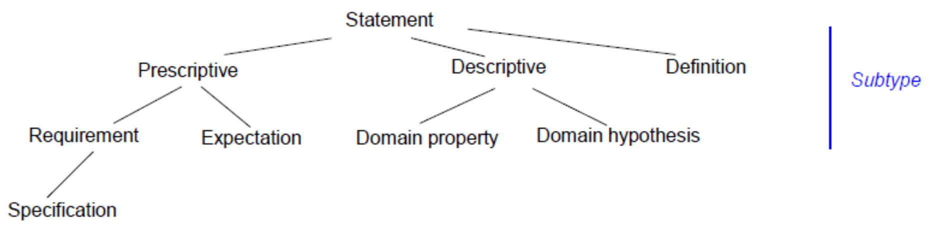


Figure 3 – van Lamsweerde distinction among the types of statements in the problem world (LAMSWEERDE, 2009).

the problem world to decompose the concept of assumption, defining that an Expectation is a prescriptive assumption while a domain hypotheses is a descriptive assumption. Moreover, van Lamsweerde proposes the concept of domain property, which is used to extend the original formula from (JACKSON; ZAVE, 1995), as an intrinsic property of the domain, which is not mutable. However, by doing that, he does not take into account the formula proposed in (GUNTER et al., 2000).

2.1.3 Wang et al.'s model

Most recently, Wang et al. (2014b) presented the Ontology of Software Artifacts (OSA), which is depicted in Figure 4, as a reference ontology (GUIZZARDI; FALBO; GUIZZARDI, 2008) founded on the concepts and definitions of Requirement, Specification, Machine and Program presented by Zave and Jackson and grounded on DOLCE (MASOLO et al., 2003). OSA reuses these definitions and the axiomatization of the foundational ontology DOLCE, to create an ontology about the artifacts that constitute our notion of software and their essential properties.

A Program, as in Zave and Jackson's definition, is an Artifact that intends to implement a Program Specification, if a set of Machine Assumptions hold. A Software System intends to implement a Software System Specification, which presupposes a set of Domain Assumptions. A Software Product is the final artifact presented in OSA, which is composed by Software Systems and have a set of High-level Requirements as their essential property. The ontology is built so the elements depicted in purple (darker background) are essential properties and part of the identity criteria of the artifacts presented on left side of the figure, in yellow (lighter background) and these essential properties depend on Assumptions about the behavior of the machine and the world.

A couple of years later, Wang et al. (2016) presented the concept of Internal Specification, which refers to a specification that constrains the phenomena happening inside the machine. This concept was created as a distinction from the original concept of specification presented by Zave and Jackson (called External Specification by Wang), which is supposed to exist at the interface between the external world and the machine, being a *implementable* refinement of the Requirements. They also present a deeper discussion

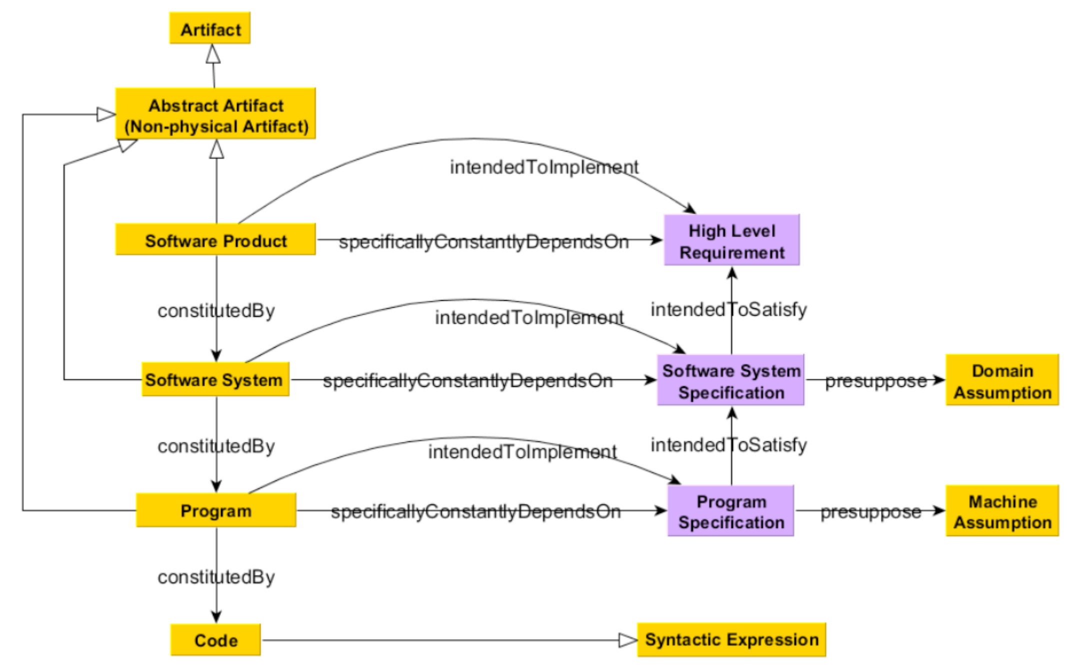


Figure 4 – Ontology of Software Artifacts (OSA) (WANG et al., 2014b).

over the concept of Assumption, suggesting the distinction between Assumptions-Used and Assumptions-Needed, as this elaboration of the original concept has practical implications on how assumptions handled during software process. For example, in the case of a failure in a Software System because of an *Assumption-Used* by the developers that does not hold when the Software System is in operation.[4]

2.2 Requirements Traceability

Traceability is defined as the degree to which a relationship can be established among two or more logical entities, especially the ones having a predecessor-successor or master-subordinate relationship to one another (ISO, IEC, 2017a).

Requirements traceability and the ability to improve software systems quality and management are being recognized and highlighted in the scientific literature for almost 25 years (GOTEL; FINKELSTEIN, 1994; RAMESH et al., 1995; RAMESH et al., 1997; PANDANABOYANA et al., 2013).

From a Configuration Management[5] perspective, Requirements Traceability improves project management, maintenance, visibility and change impact analysis capabilities for software systems. These benefits come from the fact that Requirements Traceability

[4] The preliminary ontology of assumptions is reused for the development of ROSS and is presented in Section 3.5.4.

[5] Software Configuration Management (SCM) is a support process, with the objective of identify and systematically control the configuration of a system, controlling changes and maintaining the integrity and traceability of the software system (BOURQUE; FAIRLEY et al., 2014).

makes it easier to determine the software artifacts that need to be updated to fulfill a request for change (KANNENBERG; SAIEDIAN, 2009). In this context, requirements traceability is very important support process for the software system evolution.

Because of these benefits, capability models (CMMI Institute, 2018) and international standards (ISO, IEC, 2017a; ISO, 2018) suggest that the ability of being able to trace back mutable requirements and to relate them to other artifacts in the software is indispensable for high-quality software products.

Requirements are a top priority artifact and the foundation of any software project, however, as we have showed in the previous section, many authors emphasize the importance and relevance of many others software artifacts for the software process. These artifacts are obviously related to software requirements by relations that can vary from a simple indirect association to a more relevant existential dependency relation. In fact, the relations that exist between the requirements themselves and other artifacts in the software process and their classification are a relevant part of the requirements traceability research field.[6]

One of the most accepted classifications proposes a separation between horizontal and vertical requirements traceability (LINDVALL; SANDAHL, 1996). The former refers to the activity of tracing the relations that exist between elements in different models. For example, a test case TC001 can be traced back to a requirement FR001 in a software project.[7] The latter refers to tracing the relations between elements in the same model. For example, the FR001 used in the previous example can also be directly related to other requirements.

Figure 5 presents an overview of the requirements traceability domain and depicts the concepts of horizontal and vertical traceability that were discussed above.

However, building and maintaining an efficient mechanism that provides traceability is not an easy task. Several studies (GOTEL; FINKELSTEIN, 1994; KANNENBERG; SAIEDIAN, 2009; REGAN et al., 2012; TUFAIL et al., 2017) have been presented over the years addressing major and most common problems for establishing and maintaining an effective requirements traceability policy. Problems like traceability decay, systems complexity, poor tool support, the lack of communication between stakeholders, poor data integration and even the amount of data generated are still identified as challenges to overcome in this area.

Besides, depending on the maturity level of the organization, traceability links are physically stored in spreadsheets or text-based documents, where such links tend to deteriorate during a project as time-pressured team members fail to update them. Another

[6] The relations between software requirements and other artifacts are a important part of this work and are extensively discussed in the ontology presented in Chapter 4.

[7] In this context, the test case is historically dependent on the requirement. This type of dependence is well-known and discussed in the ontology literature.

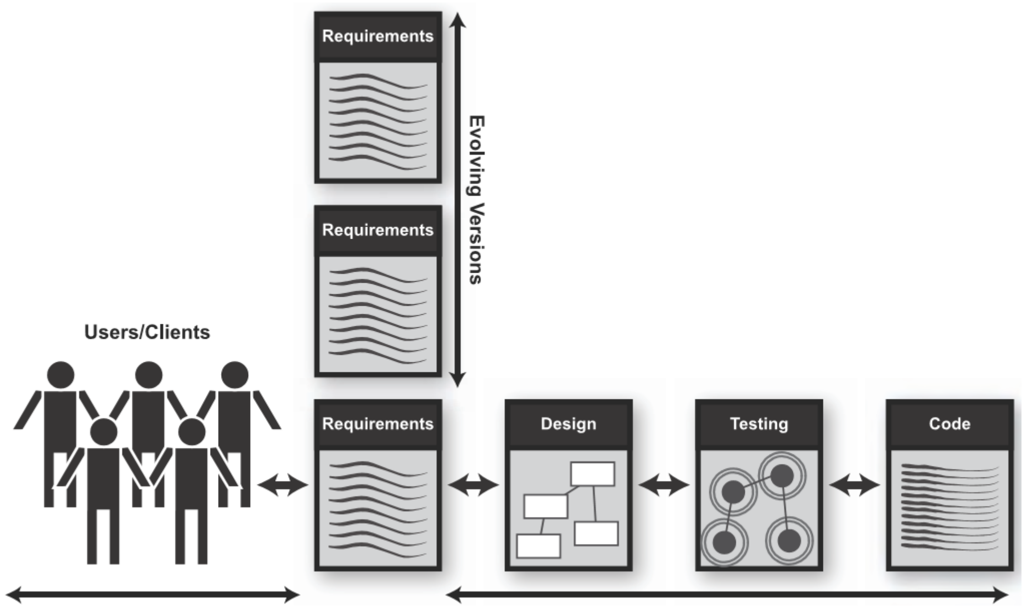


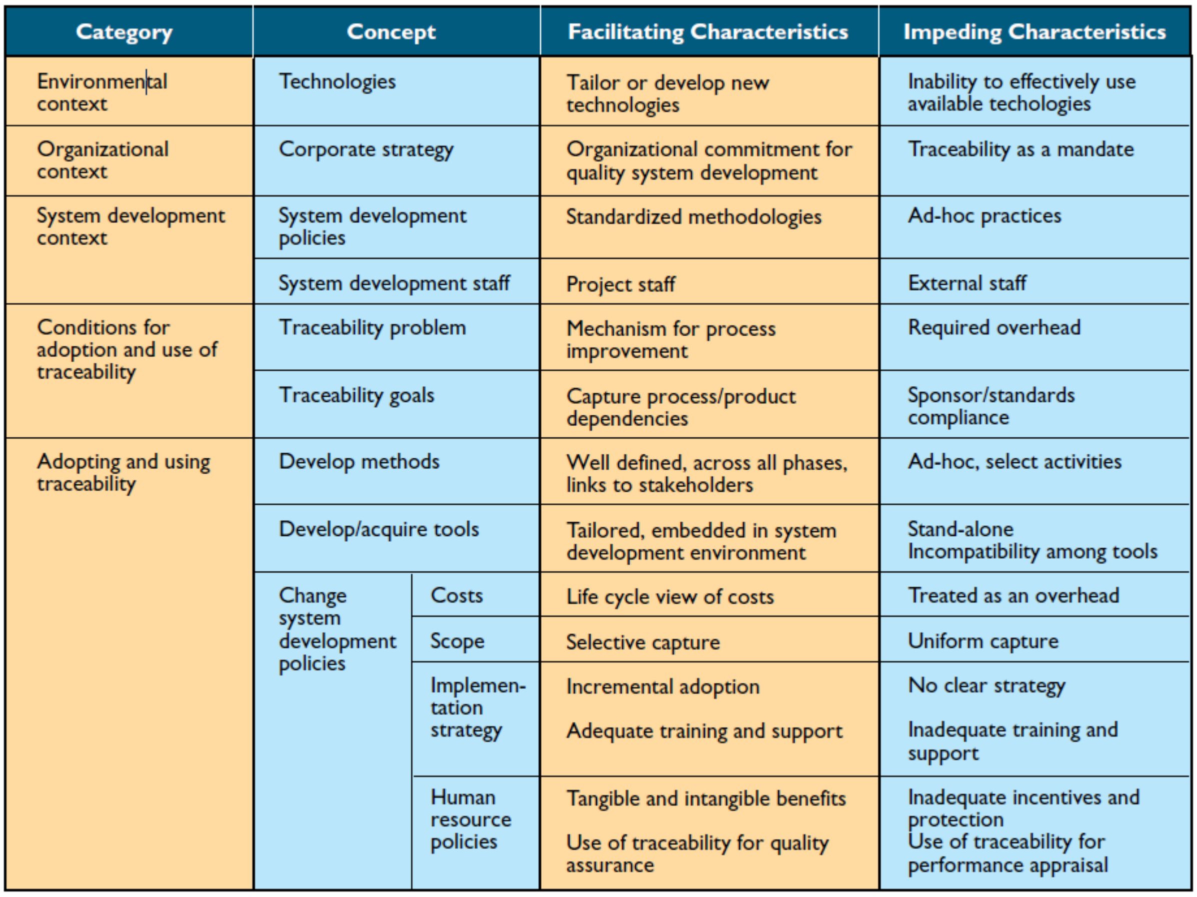
Figure 5 – Requirements Traceability Overview (KANNENBERG; SAIEDIAN, 2009).

potential solution exists in requirements management tools, such as IBM's Rational RequisitePro, however, these tools are often very expensive, making them a prohibitive to small organizations.

Because of that, unfortunately, many organizations fail to implement effective traceability practices either due to the aforementioned difficulties or because they succumb to the misconception that traceability practices return little value for the effort involved (CLELAND-HUANG et al., 2007). In his research, Ramesh (1998) states that environmental, organizational and technical factors can influence the implementation of requirements traceability positively and negatively, inside an organization.

Figure 6 depicts the results of Ramesh's research about these factors. Ramesh concludes that many organizations tend to face requirements traceability-related tasks as an overhead inside the software process(for example, as a mandatory policy to achieve CMMI level 2), because of that, they tend to abandon it before starting to notice the benefits. On the other side, organizations that are committed to the traceability process by seeing it as an opportunity for improving project management, can find its benefits and improve their processes.

Ramesh also suggests that organizations shall define their needs and goals towards the requirements traceability process. In other words, they shall define how much effort and commitment they should apply in the traceability process. Ramesh defines two types of traceability application: a low-end traceability user tends to use simple traceability schemes to model dependencies among requirements. A high-end traceability user employs detailed schemes, capturing traces between requirements and many other related artifacts. High-end users also tend to use traceability information in more elaborated ways, applying



Category	Concept		Facilitating Characteristics	Impeding Characteristics
Environmental context	Technologies		Tailor or develop new technologies	Inability to effectively use available techologies
Organizational context	Corporate strategy		Organizational commitment for quality system development	Traceability as a mandate
System development context	System development policies		Standardized methodologies	Ad-hoc practices
	System development staff		Project staff	External staff
Conditions for adoption and use of traceability	Traceability problem		Mechanism for process improvement	Required overhead
	Traceability goals		Capture process/product dependencies	Sponsor/standards compliance
Adopting and using traceability	Develop methods		Well defined, across all phases, links to stakeholders	Ad-hoc, select activities
	Develop/acquire tools		Tailored, embedded in system development environment	Stand-alone Incompatibility among tools
	Change system development policies	Costs	Life cycle view of costs	Treated as an overhead
		Scope	Selective capture	Uniform capture
		Implemen-tation strategy	Incremental adoption Adequate training and support	No clear strategy Inadequate training and support
		Human resource policies	Tangible and intangible benefits Use of traceability for quality assurance	Inadequate incentives and protection Use of traceability for performance appraisal

Figure 6 – Factors Influencing Requirements traceability in practice (RAMESH, 1998).

reasoning techniques over the traceability data.

Besides, since every organization is unique in its processes, workflows, products and resources, they have to create or adapt a method that is viable and cost-effective for the needs of the organization.

Alonso-Rorís et al. (2016) define that for a proper traceability mechanism, it is essential to establish a formal scheme for describing the information in a *semantic model* and that the application of technologies from the Semantic Web (BERNERS-LEE; HENDLER; LASSILA, 2001) enables the use of semantic traceability techniques, facilitating the management of complex relations, querying and reasoning over the data items that need to be traced. This point of view is also presented by Espinoza and Garbajosa (2011), which emphasize the use of semantics as key issue regarding traceability. In the same line, Ramesh and Jarke (2001) argue that, to be useful, traceability must be organized in a proper modeling framework.

Because of these characteristics and challenges to overcome, many researchers have focused on the development of reference models for requirements traceability as a solution that provides a picture of the reality and can be adopted based on the needs of the organization.

Reference models are prototypical models created over a specific domain with the purpose of significantly reducing the task of creating application-specific models and/or systems. The user of the reference model selects relevant parts of the model, adapts them to the problem at hand, and configures an overall solution from these adapted parts. In other words, reference models can be understood as a representative abstraction of a well-established knowledge inside a domain (RAMESH; JARKE, 2001).[8]

Besides, it is also important to understand that more standardized domains, such as the modern software systems domain, can benefit more from the desirable properties of a formalized reference model. These models should be developed based on the best practices and the condensed knowledge and later refined based on prototypes and industrial case studies.

In order to support our research on reference models for requirements traceability and to cover as much of the literature as possible, we explored the results of Systematic Literature Reviews (SLRs) (KITCHENHAM; CHARTERS, 2007; KITCHENHAM; BUDGEN; BRERETON, 2010) focused on the Requirements Traceability research field that were performed first by Nair, Vara and Sen (2013) and more recently by Tufail et al. (2017). SLRs are useful because they provide a well-defined and objective procedure for identifying the nature and extent of the literature that is available to answer a particular research question (BUDGEN et al., 2008). Besides, they compile information and the underlying knowledge about the topic that is being researched and can support the identification of gaps in the current research (KITCHENHAM; CHARTERS, 2007). For this thesis we reviewed the results of both SLRs, focusing on papers that proposed reference models for Requirements Traceability. Additionally, we performed the *snowballing* technique (i.e., search into the references of the papers considered relevant for the the work in progress) to find other related papers.

Within the results of these studies, the work of Ramesh and Jarke (2001) is considered as one of the most relevant and a precursor of model-based traceability research. They propose that the efficiency and the effectiveness of traceability heavily depends on system of objects and traceability links types that are utilized. Moreover, authors also point out to the fact that reference models can be built based on different aspects of the requirements traceability field. For example, their reference model for requirements traceability is focused on the Objects (Artifacts) that exist in the software process, while the Semantic Model proposed by Alonso-Rorís and colleagues is focused on the characterization of the traceability process domain of types of traces.

The approach proposed by Zhang et al. (2014) focuses only on the dependencies that exist between requirements. They reuse Pohl's Requirements Dependency Model and Dahlstedt's Requirements (Inter) dependency model (DAHLSTEDT; PERSSON, 2005) to

[8] The concept of ontologies being developed and used as domain reference models is discussed in Section 3.

create a new one and use it as the reference model for a requirements traceability in a small industrial-based case study.

Ahn and Chong (2006) proposed a meta-model for feature-oriented requirements traceability and an overview of a feature-oriented requirement tracing process. Their meta-model defines priorities for features and relates them with some types of software artifacts.

Serrano and Leite (2011) proposed a model for requirements traceability based in graphs, on which each edge is a trace. ITrace is focused on modeling the social networks and interactions of the software process. The model is divided in three graphs: A Graph of the social network and the information sources, a graph of the social interactions and a graph of the RE artifacts evolution.

Besides these, many other studies[9] can be found in the literature with different approaches for the development of a traceability mechanism that, based on models, intended to provide satisfactory results and overall process support for its users. These proposals and others that exist in the literature are related to our ontologies to certain degree, as they are all designed based on the utilization of reference models. However, the difference in our proposal is in fact that it extends through the software system context, by taking advantage of the process of reusing other software-related ontologies. Besides, the ontologies presented in this work, ROSS and OSDEF, are committed to a foundational ontology that defines an extensive and well-accepted conceptual modeling theory. They are also based on the knowledge presented in international standards.

Moreover, our proposal is not focused on the definition of the concept of *trace* and its sub-types. We believe that the trace between the software artifacts exists in their relations. In other words, the information that exists in the relations between artifacts, that are represented in ROSS and OSDEF, is the trace itself.

Finally, as a part of the requirements traceability research field, the utilization of reference model is often adopted as a way to represent the important aspects of the traceability process for the organization. As mentioned earlier in this chapter, requirements tracing can provide many benefits for the software process of an organization, however it tends to become more complex and more susceptible to errors as the software naturally ages and changes are made. The utilization of reference models is then an attempt to provide support for some of the choices that are made, in the sense that the organization members have to follow the directives that are proposed in the model, at the same time, it makes the process easier to understand.

[9] Other proposals are further discussed in Section 7.5

2.3 Chapter Summary

This chapter summarized part of the state-of-the-art of the Requirements Engineering and Ontologies research that was used as background of this thesis. In Section 2.1, we started the discussion by introducing the RE for Software Systems as a research field, explaining the importance of requirements, their special properties and their mutable nature.

After that, we discussed other research that are the baseline for the development of this thesis. The first, and most important one is Zave and Jackson's reference model for RE, which is based on five concepts that are crucial to the Software Systems process, namely: (i) the knowledge over the environment where a Software System exists; (ii) the requirements of the Software System; (iii) the specification that is created as a refinement of the requirements, to describe them; (iv) the programs that implements a specification, based on a programming platform; and (v) the programming platform itself, that provides a basic set of rules and functions for the software to be executed and to operate as a tool.

Based on Zave and Jackson's work, van Lamsweerde and Wang dedicated their works to improve the original formula, presented by Zave and Jackson in the middle of the nineties ($S, K \vdash R$) and the reference model. van Lamsweerde focused on the relations between the (problematic) *world*, the *machine* (a software-based system) that is created to solve it. He also discussed the *domain properties* and *assumptions* that exist between them. On the other side, Wang et al. performed an ontological analysis of the domain and proposed another extension of the reference model, by differentiating between internal and external specifications and between domain and machine assumptions. Wang et al. proposed OSA, the Ontology of Software Artifacts, that is directly related to ROSS and OSDEF, the ontologies proposed in this thesis, as they are all based on Zave and Jackson's work.

We also discussed the requirements traceability literature, focusing on the utilization of reference models as tools for the development of a requirements traceability approach. For this, we adopted, as a starting point, two SLRs that were recently performed. Traceability research has greatly focused on requirements traceability, aiming at studying how to describe and follow the life of a requirement, in both forward and backward directions. Requirements traceability has been demonstrated to provide many benefits to organizations that make proper use of traceability techniques. This is why traceability is an important part of many standards for software development, like ISO 26559 (ISO, IEC, 2017a) and CMMI. In spite of the benefits that traceability offers to the software engineering industry, its practice faces many challenges. These challenges can be identified under the areas of cost in terms of time and effort, the difficulty of maintaining trace ability through change, different view points on traceability held by various project stakeholders, organizational problems and politics, and poor tool support.

3 Ontological Foundations

This Chapter presents the ontological foundations that were used as the ground theory for this thesis. Section 3.1 discusses the concept of ontology. Section 3.2 presents UFO, the foundational ontology adopted in this work. Section 3.3 presents an overview of the Semantic Web and discusses the Linked Data technologies that were used in this thesis. Section 3.4 presents SABiO, the ontology engineering method adopted to create ROSS and OSDEF, the ontologies developed in this work. Section 3.5 presents SEON, the Software Engineering Ontology reused for the development of both ROSS and OSDEF. Finally, Section 3.6 summarizes the chapter.

3.1 Ontologies

Presently, one of the most accepted definitions of ontology, and the one used as reference in this thesis, is the one by Borst and Borst (1997) as *a formal, explicit specification of a shared conceptualization*. However, the concept of ontology and the scientific interest about it as research field are not new. The first *ontology*, in the sense of a theory about the kinds of existence, was created by Aristotle, in his work Metaphysics and Categories, as *the science of being qua being*. By this definition, *Ontology* (with a capital "o") is a branch of Metaphysics (a discipline of Philosophy) that can be understood as the study of real objects and its most general features. However, the term "ontology" was created only in the 17th century, by the philosophers Rudolf Gockel and Jacob Lorhad. More recently, at the beginning of the 20th century, Edmund Husserl defined *Formal Ontology* as an analog of Formal Logic, a discipline that aims to develop a system of general, domain-independent set of categories that can be used in the development of scientific domain-specific theories (GUIZZARDI, 2007).

Although the concept of ontology has a clear philosophical nature, it has been recognized as a promising research field in areas of Computer Science, such as Artificial Intelligence, Software Engineering, Linked Data, Database Design, Knowledge Engineering and information integration (GUARINO, 1998). In this context, ontologies can also be used as *conceptual models*; engineering artifacts, that are designed for a specific purpose, such as reducing conceptual ambiguities and false agreements, improving knowledge representation, supporting system interoperability, etc., and are represented in ontology-based languages (GUIZZARDI, 2007), which are specifically created to provide a appropriate representation of a conceptualization.

Figure 7 depicts the relations between Conceptualization, Modeling Language and their instances, Abstraction and Model, respectively. Conceptualization and Abstractions

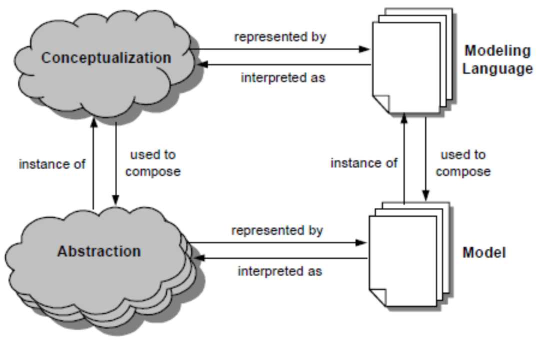


Figure 7 – Relations between Conceptualization, Modeling Language, Abstraction and Model (GUIZZARDI, 2007)

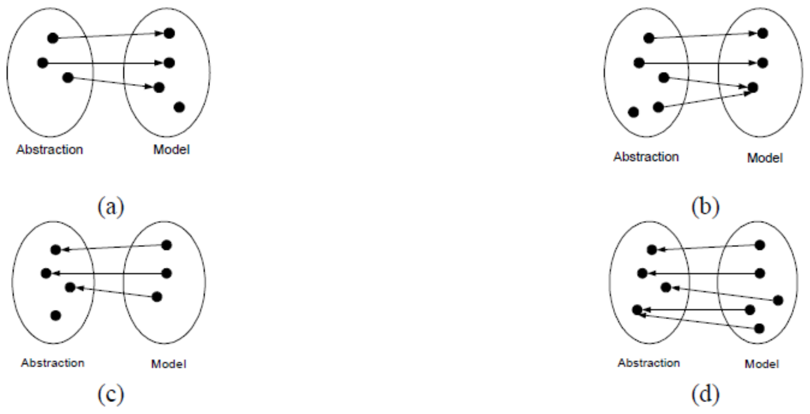


Figure 8 – Examples of (a) Lucid, (b) Sound, (c)Laconic and (d) Complete mappings adapted from (GUIZZARDI, 2005).

are represented with clouds because they are immaterial entities that exist in the mind of an user or a group of users of an language. This means that for a communication do be precise and unambiguous, it is necessary that the language represents the same conceptualization for the participants of the conversation (GUIZZARDI, 2007).[1]

Figure 8 presents these properties in examples of mappings between elements of an Abstraction and a Model. A *Lucidity*: A language L is lucid to a domain D iff every modeling primitive in the language represents at most one domain concept in the ontology O; *Soundness*: A language L is sound to a domain D iff every modeling primitive in the language has an interpretation in terms of a domain concept; *Laconicity*: A language L is laconic to a domain conceptualization iff every concept in Abstraction of that domain conceptualization is represented at most once in the meta-model of that language; *Completeness*: A language L is complete to a domain D iff every concept in the ontology O of that domain is represented in a modeling primitive of that language;

Based on these properties, a language based on an existing ontology, as its ontolog-

[1] Guizzardi (2005) deeply discusses a set o properties that should be guaranteed for an isomorphic mapping between an abstraction and a model.

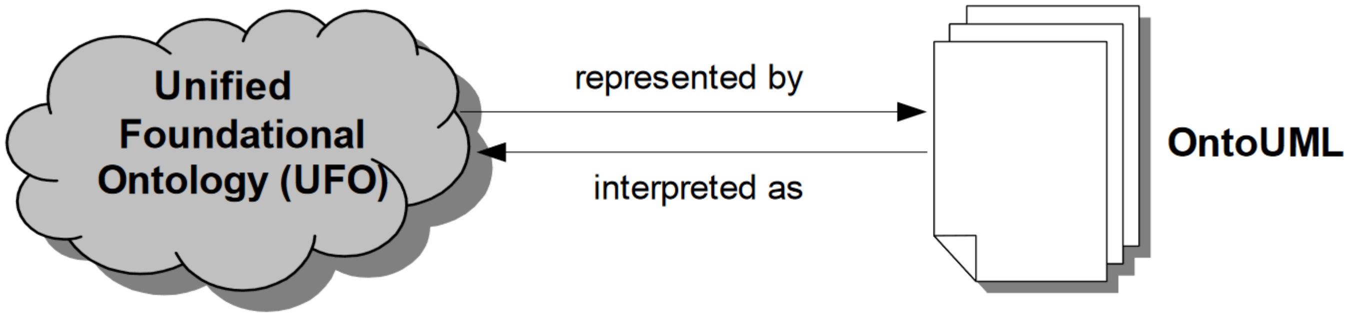


Figure 9 – Relation between UFO, as a conceptualization and OntoUML, as a modeling language

ical commitment,[2] describes a set of constructs that aim to approximate the language to its intended meaning (GUARINO, 1998), through the formal axioms that exists in the ontology underlying it. In other words, ontologies are specially useful to provide *real-world semantics* for conceptual modeling languages as they provide a worldview for the language abstract semantic to follow. For example, OntoUML is an ontology-based conceptual modeling language that is grounded on the UFO ontology (GUIZZARDI, 2005), in the sense that OntoUML commits to the world view and to the formal axioms that are defined in UFO.

Figure 9 is adapted from Figure 7 to make explicit the relation between UFO and OntoUML

Considering the information presented above, one can deduce that different types of ontologies should exist, based on the types of domains which they represent and on level of formalism adopted by them. There are several different classifications of ontologies in the scientific literature. These classifications are usually based on their specificity level or in their building purpose. Figure 10 presents the classification proposed by Guarino (1998), which separates ontologies based on their degree of specificity and is one of the most widely-accepted classification by the scientific community.

Top Level Ontologies (or *Foundational Ontologies*) are the most general ontologies, that describe concepts that are domain-independent such as objects, time, space, matter, events and the fundamental types of relations between these concepts. *Domain Ontologies* describe concepts that are related to a specific domain, such as Requirements Engineering or Economics. Domain Ontologies are usually created by specializing/reusing concepts from a pre-existing top-level ontology. *Task Ontologies*, as the name suggests, represent ontologies that are created over generic tasks or activities, such as selling or building, that

[2] The ontological commitment K, of Language L with vocabulary V, represents that L *commits* to a *intensional* interpretation of a conceptualization C that relates a world structure with the elements of L. In other words, it denotes the *result* of the commitment (of the language) with an underlying conceptualization.

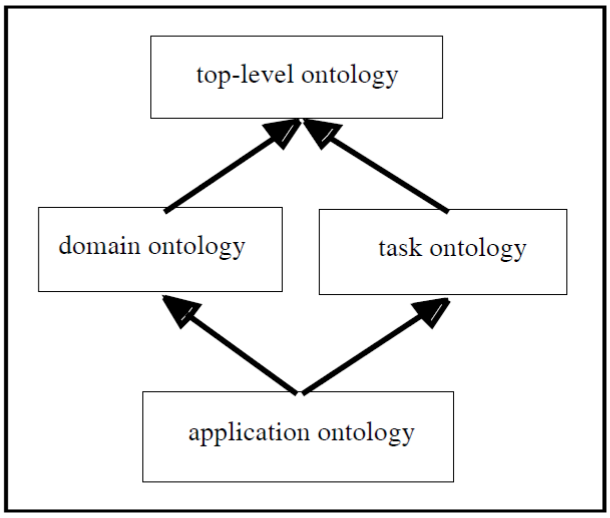


Figure 10 – Types of Ontologies according to their to level of generality, presented in (GUARINO, 1998). Arrows represents specialization relationships.

traverse multiple domains. *Application Ontologies* describe concepts about both a specific domain and the tasks and activities related to it. Because of that, *Application Ontologies* usually specialize both domain and task ontologies.

In a didactic representation of Guarino's classification, Figure 11 depicts the ontologies that are proposed and reused in this work. At the beginning of the continuum that represents the foundational level, we have UFO-A (GUIZZARDI, 2005), UFO-B (GUIZZARDI et al., 2013) and DOLCE (MASOLO et al., 2003), ontologies that are very generalist, in the sense that they are focused on concepts about the world and that are created to be reused by other ontologies. Following the continuum to the right, UFO-C is also considered a foundational ontology, although more specific, as it focuses on social aspects of the world, and reuses concepts that are originally part of UFO-A and UFO-B. Core Ontologies are the ones that are not domain independent but they tend to focus on macro-domain, for example, the Software Process Ontology (SPO) (BRINGUENTE; FALBO; GUIZZARDI, 2011a) is considered a core ontology of the software domain, as it was reused for the development many other ontologies, however it is still less generic than an ontology about processes in a domain-independent way. Besides, core ontologies are usually developed grounded on a foundational ontology with the specific purpose of being reused by domain ontologies. ROSS and OSDEF are considered domain ontologies because they are much more specific that the other ones and also reuse concepts from foundational (UFO) and core ontologies (SPO).

Based on these classification, the fact is that Figure 11 is represented as a continuum because the boundaries between the categories are fuzzy and it is difficult to drawn an strict line to separate when an ontology is too generic to be a domain ontology or, too specific to be a core ontology. For example, on one side, ontologies like UFO-A, UFO-B

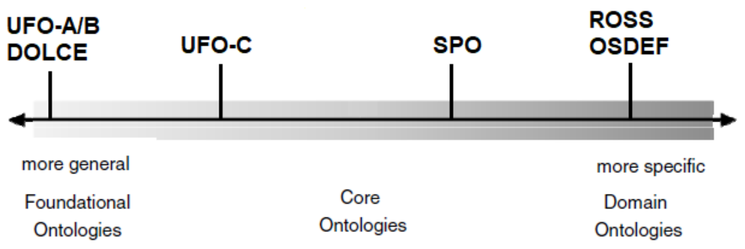


Figure 11 – Representation of ontologies generality level as a continuum. Adapted from (FALBO et al., 2013).

and DOLCE are clearly foundational ontologies, as they are genuinely domain independent. On the other side, an ontology like SPO can be understood as a domain ontology since it is focused on the software process however, since many more specific domain ontologies about parts of the software process have been built reusing SPO, it can also be seem as a core ontology. In practice, a domain ontology is not, necessarily at the end of the continuum, as sometimes a even more specific domain ontology can be created reusing the concepts of the first one and both, core and domain should be grounded on foundational ontologies.

Another widely accepted classification that is orthogonal to the one proposed by Guarino classifies ontologies based on their use: *Reference Ontologies* are heavily axiomatizated ontologies that are created to represent the intended domain as best as possible. They are usually created with languages that are focused on expressiveness and adequacy to the domain. Moreover, reference ontologies are also created to be used by humans, assisting them in their tasks. On the opposite side, *Operational* or *Lightweight ontologies* are based on reference ontologies, however, they are created to be machine processed, with languages that focus on computational properties, such as OWL or RDF (GUIZZARDI, 2007). For this work, we adopt both classifications that were discussed, as we intend to propose a domain reference ontology about requirements of software-based systems that is based on the foundational ontology UFO and also to implement it using an operational ontology language.

3.2 UFO

The Unified Foundational Ontology – UFO (GUIZZARDI, 2005; GUIZZARDI, 2006; GUIZZARDI, 2007; GUIZZARDI et al., 2013) is a foundational ontology that was developed based on a number of theories from Formal Ontologies, Modal Logic, Linguistics and Cognitive Psychology.

UFO is composed of three main parts: UFO-A, an ontology of endurants, that discusses objects, their types, compositions and relations (GUIZZARDI, 2005); UFO-B, an ontology of perdurants, that discusses events, their compositions, their participants and

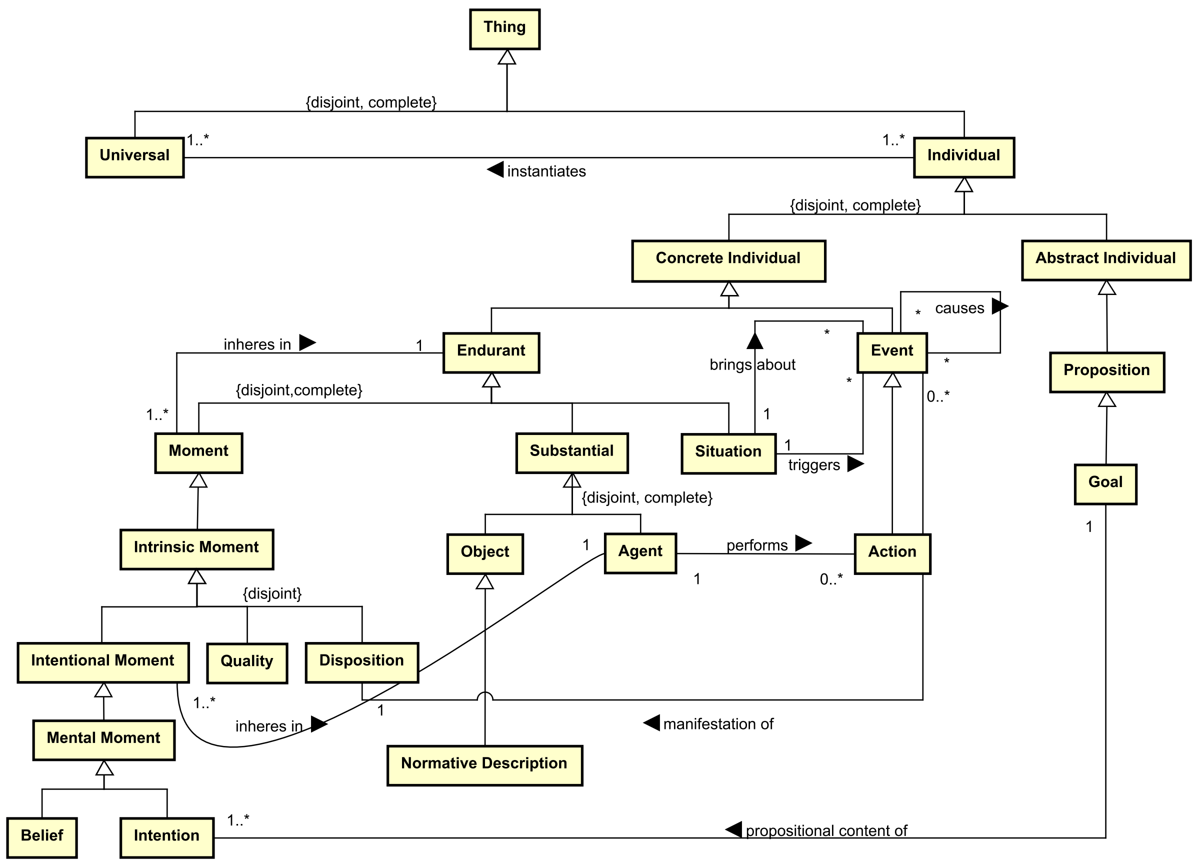


Figure 12 – A fragment of UFO presenting the concepts that are used in this work.

relations (GUIZZARDI et al., 2013; BENEVIDES et al., 2019); and UFO-C (GUIZZARDI, 2006; GUIZZARDI; FALBO; GUIZZARDI, 2008; BRINGUENTE; FALBO; GUIZZARDI, 2011b), an ontology of social entities, that is built on top of UFO-A and UFO-B (based on Endurants and Perdurants) and discusses agents, their actions, goals, intentions, and commitments. Moreover, other parts of UFO have been developed, such as UFO-S (NARDI et al., 2015a) that extends UFO-A, B and C to discuss the concept of services.

For brevity reasons, Figure 12 shows the concepts of UFO that are reused in this thesis.[3] The concepts presented are originally defined in the three main parts of UFO: UFO-A, UFO-B and UFO-C.

In UFO, things in the world can be classified as Universals, which are types of things, entities created to represent the general properties and aspects of something, for example, a chair as type of object on which someone can sit. On the opposite side, Individuals are entities that exist instantiating Universals and possessing a unique identity. For example, Queen Elizabeth's throne is an individual chair with unique properties. Individuals can be concrete or abstract, depending on their nature. Concrete Individuals can be further

[3] The complete model of UFO, combining UFO-A UFO-B and UFO-C has around 100 concepts, so it becomes impracticable to represent the entire model in this format.

classified as Perdurants/Events and Endurants.[4]

Events are entities with temporal parts, they happen in time accumulating temporal parts, for example, a football game is a complex event that is composed by many other atomic events. Besides that, Events can cause other Events (in the sense of chains of events) and can bring about Situations, which are portions of reality that exist as a whole. Moreover, Events are manifestations of Dispositions that inhere in Objects. For example, a magnet has the intrinsic property (a Disposition) to attract a piece of iron that is placed nearby.

On the other side, Endurants are entities that exist in time, they do not have temporal parts and can be existentially-independent: a Substantial, as Queen Elizabeth (an Agent), or her throne (an Object); or existentially-dependent: a Moment (sometimes also called a Trope), as the headache that the queen might have in a very hot day. Moments can be seen as properties that can only exist by inheriting in other Endurants. For example, the red color of the leather (a Quality of the leather) on the throne, or they can have the Disposition to be manifested because of an Event. For example, the crown has the Disposition to fall if the queen shakes her head.

Agents are Substantials capable of performing Actions, which are intentional Events. Goals are the propositional content of the Intentional Moments of an Agent.[5] For example, Queen Elizabeth might have the Goal that William becomes king, instead of his father, Charles, however, Goals require self-commitment from the Agent, which means that in order to achieve this Goal, the Queen would have to convince Charles and the other Lords that William will be a better King. In other words, she would have to commit to perform actions to achieve her Goal. Finally, Normative Descriptions are rules/norms that are recognized by an Agent. For example, the British people, as a Social Agent, accept the United Kingdom's constitution and the laws that are created in the parliament.

Finally, UFO was chosen as the foundational ontology to ground the ontologies presented in this thesis because it addresses many essential aspects for conceptual modeling, providing a complete set of categories to tackle the specificity of the software domain. Further, it has been employed, successfully, as foundational ontology for the creation of several other software-related ontologies (BRINGUENTE; FALBO; GUIZZARDI, 2011a; BARCELLOS; FALBO; MORO, 2010; DUARTE et al., 2018; SOUZA et al., 2013). Besides that, UFO is the foundational ontology suggested by SABiO, the ontology engineering method that was used to create the ontologies presented in this thesis.

[4] UFO-A is focused on Endurants and UFO-B on Perdurants.

[5] An Intentional Moment *inheres in* an Agent, in the sense that the Intentional Moment is existentially dependent of the Agent who bears it.

3.3 Operational Ontologies, Linked Data and the Semantic Web

The term *Semantic Web*, was originally created by the inventor of the World Wide Web (WWW), Tim Berners-Lee. In fact, the development of the Semantic Web has always been related to the traditional World Wide Web (HITZLER; KROTZSCH; RUDOLPH, 2009). Similarly to the WWW, the foundations of the Semantic Web are the technologies that are responsible for defining the data formats that are adopted in it.

3.3.1 RDF framework and SPARQL

The Resource Description Framework (RDF) is a formal language for representing information on the Web. RDF is a W3C recommendation,[6] developed to represent data in a flexible way, in addition to allow the automated processing of this data. Differently from common HTML pages, which was created to be human-readable, RDF was created to be a common language for the machine-readable Semantic Web (W3C, 2004). However, that is not the only difference between RDF and other well-know Web languages such as HTML and XML, as they are created in a tree structure, while RDF documents organize and store data in a graph data model (HITZLER; KROTZSCH; RUDOLPH, 2009) of triples. In other words, RDF data model is a graph, that can be represented by several syntax (even XML) while XML is a syntax with a tree data model. The direct consequence of this distinction is that XML is too permissive, an assertion can be made in many different ways, while RDF will always follow the structure format of a triple.

RDF *triples* are composed by a *subject*, that represents the resource of which we are talking about; a *predicate/property*, that denotes a relationship and defined the information that is being expressed about the *subject* and an *object* that defines the value of the predicate. Figure 13 presents this structure in a graph example about the birthplace of Tim Berners-Lee, where the nodes of the graph are the subject (Berners-Lee) and the object (London), and the arc is the predicate (birthplace). Moreover, it is important to notice that the direction of the predicate in the graph is relevant, as it will always point out to the object.

RDF is considered one of the foundations for the Semantic Web (HITZLER; KROTZSCH; RUDOLPH, 2009), as it is the base for many other technologies such as the RDF-Schema (W3C, 2014a), which is an extension of the basic RDF vocabulary, Turtle (W3C, 2014b), a textual syntax that allows an RDF graph to be written in a more user-friendly and compact way, adding the possibility for creation of abbreviations for data-types. Turtle provides compatibility with the triple pattern syntax of SPARQL.

[6] A W3C Recommendation is a specification that, after extensive consensus-building, has received the endorsement of W3C Members and the Director. In other words, W3C Recommendations are similar to the standards published by other organizations, such as ISO.

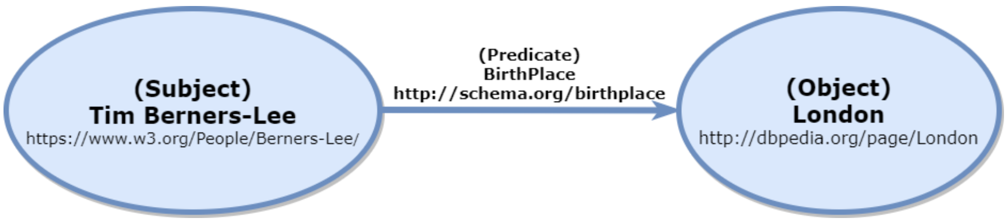

(Subject)
Tim Berners-Lee
https://www.w3.org/People/Berners-Lee/
(Predicate)
BirthPlace
http://schema.org/birthplace
(Object)
London
http://dbpedia.org/page/London

Figure 13 – Graph representation of a triple that represents birthplace information of Tim Berners-Lee. Adapted from (W3C, 2004).

SPARQL (W3C, 2013), is a query language for RDF that is used as an evaluation tool for the ontologies presented in this thesis.

SPARQL can be used to express queries across diverse data sources, whether the data is stored natively as RDF or viewed as RDF via middleware. SPARQL contains capabilities for querying required and optional graph patterns along with their conjunctions and disjunctions.

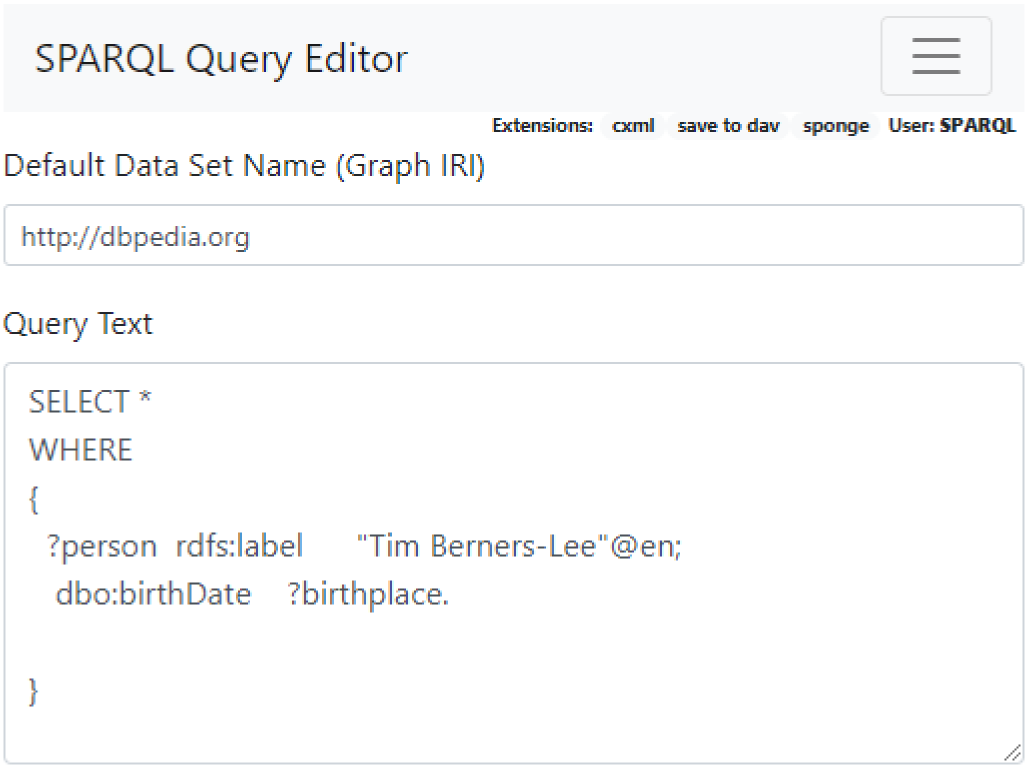
The SPARQL query language can be used to query data from different data sources that follow the RDF data model. SPARQL queries are composed of a series of triples that follow the *<Subject, Predicate, Object>* format. The query works by searching in the data sources for triples that obey the format presented in the query (usually after the WHERE clause). The results of SPARQL queries can be result sets or RDF graphs. SPARQL v1.1, its most recent version, contains capabilities for querying required and optional graph patterns along with their conjunctions and disjunctions, as the language provides a complete set of logical and numerical operators, aggregate functions, sub-query support and INSERT/DELETE commands.

For illustration purposes, Figure 14 presents a simple SPARQL query based on the triple depicted in Figure 13 that searches for the birthplace of Sir Tim Berners-Lee. The query searches for the information in the datastet of <dbpedia.org>.

3.3.2 gUFO – UFO‘s gentle implementation

gentle UFO (gUFO) (ALMEIDA et al., 2019) is a lightweight operational version of UFO. gUFO was created to provide an implementation of UFO that is suitable for the development of Linked Data approaches. gUFO is intended to be used by designers of lightweight/operational ontologies.Moreover, gUFO can be used through the specialization and instantiation of its concepts, facilitating the implementation of reference ontologies that are originally grounded in UFO.

The key feature of gUFO is that it includes both taxonomies originally presented in UFO. The first one with classes whose instances are individuals, for example, the concepts of gufo:Object and gufo:Event. The second one with classes whose instances are types/Universals, for example, the concepts of gufo:Kind, gufo:Phase, gufo:Category.


SPARQL Query Editor
Extensions: cxml save to dav sponge User: SPARQL
Default Data Set Name (Graph IRI)
Query Text

```
http://dbpedia.org
```

```
SELECT *
WHERE
{
 ?person  rdfs:label     "Tim Berners-Lee"@en;
  dbo:birthDate    ?birthplace.


}
```

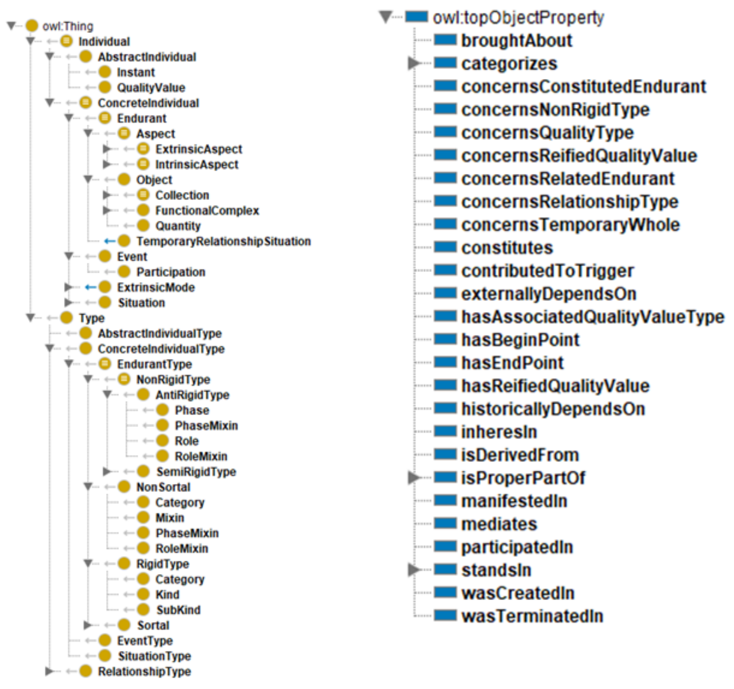
Figure 14 – A SPARQL query example executed in <https://dbpedia.org/sparql/>

Figure 15 depicts classes (left) and object properties (right) hierarchy of gUFO in Protégé. Classes are the concepts of the ontology and object properties are the relations that exist between the concepts.[7]

gUFO presents the implementation of a part of UFO-A (GUIZZARDI, 2005), which defines the concepts of Objects, Aspects, Situations and the distinction between Individuals and Types (Universals). gUFO also implements the definition of Events and the notion of object Participation in events, which is defined in UFO-B (GUIZZARDI et al., 2013).

Lastly, although gUFO does not present the definition of concepts of UFO-C, such as Agents and Actions, it was designed so the user can define new concepts with less effort. For example, an Action can be defined as a sub-type of Event, as it is originally defined in UFO-C. In this case, the new concept will automatically inherit all the properties that are already implemented for the concept of Event. gUFO was used as the base for the development of the operational ontologies that were used to evaluate the reference models of ROSS and OSDEF.

[7] The complete description of all gUFO elements and its source code can be found in <https://purl.org/nemo/doc/gufo>.



```
owl:Thing
  Individual
    AbstractIndividual
      Instant
      QualityValue
    ConcreteIndividual
      Endurant
        Aspect
          ExtrinsicAspect
          IntrinsicAspect
        Object
          Collection
          FunctionalComplex
          Quantity
        TemporaryRelationshipSituation
      Event
        Participation
      ExtrinsicMode
      Situation
  Type
    AbstractIndividualType
    ConcreteIndividualType
      EndurantType
        NonRigidType
          AntiRigidType
            Phase
            PhaseMixin
            Role
            RoleMixin
          SemiRigidType
        NonSortal
          Category
          Mixin
          PhaseMixin
          RoleMixin
        RigidType
          Category
          Kind
          SubKind
        Sortal
      EventType
      SituationType
    RelationshipType
```

```
owl:topObjectProperty
  broughtAbout
  categorizes
  concernsConstitutedEndurant
  concernsNonRigidType
  concernsQualityType
  concernsReifiedQualityValue
  concernsRelatedEndurant
  concernsRelationshipType
  concernsTemporaryWhole
  constitutes
  contributedToTrigger
  externallyDependsOn
  hasAssociatedQualityValueType
  hasBeginPoint
  hasEndPoint
  hasReifiedQualityValue
  historicallyDependsOn
  inheresIn
  isDerivedFrom
  isProperPartOf
  manifestedIn
  mediates
  participatedIn
  standsIn
  wasCreatedIn
  wasTerminatedIn
```

Figure 15 – Visualization of Classes and Object Properties (Relations) of gUFO in Protégé

3.4 SABiO – A Systematic Approach for Building Ontologies

This section presents SABiO, the Systematic Approach for Building Ontologies (FALBO, 2014), the ontology engineering method chosen for this thesis. SABiO was chosen over other methods, such as the NEON Method (SUÁREZ-FIGUEROA et al., 2012), On-to-Knowledge (SURE; STUDER, 2002) and METHONTOLOGY (FERNÁNDEZ-LÓPEZ; GÓMEZ-PÉREZ; JURISTO, 1997) because it is focused on the development of domain ontologies. Besides that, SABiO explicitly recognizes the importance of using foundational ontologies in the ontology development process to improve the ontology quality, representativity and formality. Finally, it has been successfully used on the development of several domain ontologies in Software Engineering, such as the Runtime Requirements Ontology (RRO) (DUARTE et al., 2016; DUARTE et al., 2018), the Reference Ontology on Software Testing (ROoST) (SOUZA; FALBO; VIJAYKUMAR, 2017), the Reference Software Requirements Ontology (RSRO) (DUARTE et al., 2018) and others. SABiO provides support and facilitates the reuse of those ontologies, which is very beneficial, as we intended to reuse concepts of already established ontologies for the software domain.

Figure 16 presents the five phases of SABiO and the support activities that are proposed by the method. SABiO's development process is composed of five main phases, which are supported by well-known activities in the Requirements Engineering life-cycle, such as knowledge acquisition, reuse and documentation.

The first phase, named Purpose Identification and Requirements Elicitation, as the name suggests, is about defining the purpose and the intend use of the ontology and

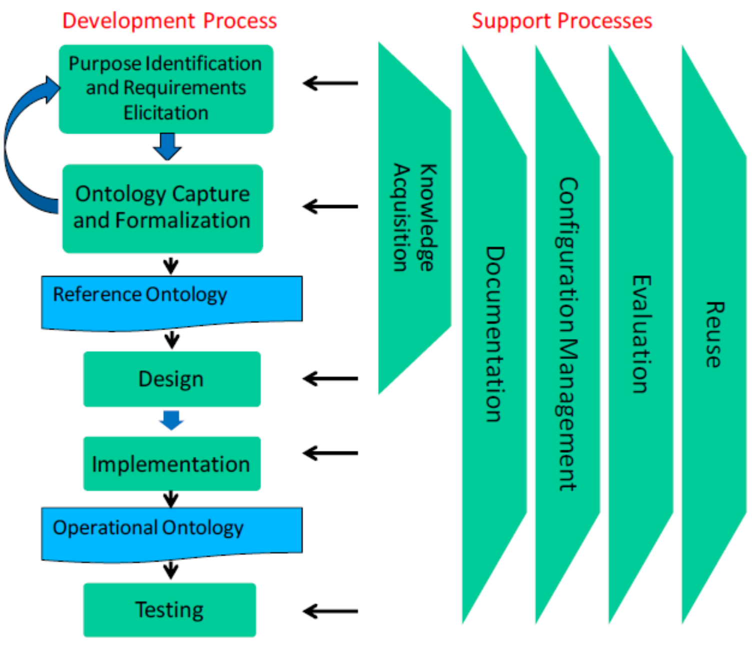


Figure 16 – SABiO's Development Process and Support activities (FALBO, 2014).

eliciting its requirements. Analogously to software requirements, ontology requirements can be divided in functional requirements, which define the content to be represented by the ontology, and non-functional requirements, which represent quality properties and general aspects of the ontology. SABiO defines that the functional requirements of an ontology should be written in the form of competency questions (CQs) (GRÜNINGER; FOX, 1995), which are questions that the ontology should be able to answer. Moreover, CQs help to refine the scope of the ontology and can also be used in the ontology verification process.

The main objective of the second phase, Ontology Capture and Formalization, is to define the domain conceptualization based on the CQs that were developed in the first phase. The concepts and relations that are part of the domain and that will be represented in the ontology should be identified, organized and analyzed in the light of a foundational ontology, to improve its capability to represent the domain. It is important to understand that this phase is very iterative, in the sense that new CQs may arise in the ontology development process. Besides that, SABiO suggests that high-expressiveness graphical languages, such as OntoUML should be used to represent the ontology, in a *reference ontology*, as they support communication and can be used as reference models.

The Design phase starts after the reference ontology is produced. Its objective is to transform the specification of the reference ontology, produced in second phase, in the specification of an operational ontology. In the design phase, ontology engineers have to tackle problems of lack of expressiveness of operational languages, when compared to languages focused on the creation of reference ontologies, such as OntoUML. Moreover, the specification of a reference ontology can be used as conceptual base to the development of many operational ontology designs, as the designers and ontology engineers may decide to design only specific parts of the reference ontology as an operational ontology.

Analogously to Software Engineering, the ontology implementation phase consists on the development of an operational ontology based on the design specification produced in the previous phase, using the adopted operational language.

Ontology Testing is the last phase of SABiO, it consists of dynamically verifying and validating the behavior of the operational ontology through a set of test cases and comparing the results with the expected behavior based on the competency questions. SABiO suggests that the tests be done initially starting from the sub-ontologies that make up the domain ontology and rising as the sub-ontologies are being integrated until the final ontology is properly integrated and can be tested working as a whole.

Regarding the supporting process, ontology reuse is probably the most important one. Reusing existing ontologies is a crucial part in an ontology engineering process and should be applied whenever possible, because it helps to keep consistency with previously built ontologies, reduces workload and avoids concept redundancy. SABiO explains that ontologies can be partially or totally reused, in distinct ways. Foundational and core ontologies are usually reused by means of specializations, where the concept defined in the foundational ontology is specialized in a more specific domain-related concept. For example, Artifact, a concept originally defined in the Software Process Ontology (SPO), is a sub-type of Object, defined in UFO. Ontologies can also be reused by Analogy. In this sense, concepts and relations are not explicitly extended in a domain ontology, instead, they are implicitly used to define the structure of a porting of a domain ontology. For example, the relations that exist between the concepts of Program, Loaded Program Copy, Program Copy Execution and Machine presented in RRO follow the structure pattern of Events, Dispositions and Objects, defined in UFO.[8] For this thesis, we reused concepts that are part of SPO, SwO, the ontology of assumptions (ASMP) and also specialize concepts from UFO. The reuse is facilitated because all ontologies are part of the same domain and are grounded on UFO.

3.5 SEON - the Software Engineering Ontology Network

In the Ontology Engineering literature, an ontology network is defined as a collection of ontologies related together through a variety of relationships, such as alignment, modularization and dependency. A networked ontology, in turn, is an ontology included in such a network, sharing concepts and relations with other ontologies (SUÁREZ-FIGUEROA et al., 2012). The Software Engineering Ontology Network (SEON) (RUY et al., 2016) was designed seeking to: (i) take advantage of well-founded ontologies (all of its ontologies are

[8] This ontology pattern can be see in Figure 12 and is extensively discussed in (GUIZZARDI et al., 2013).

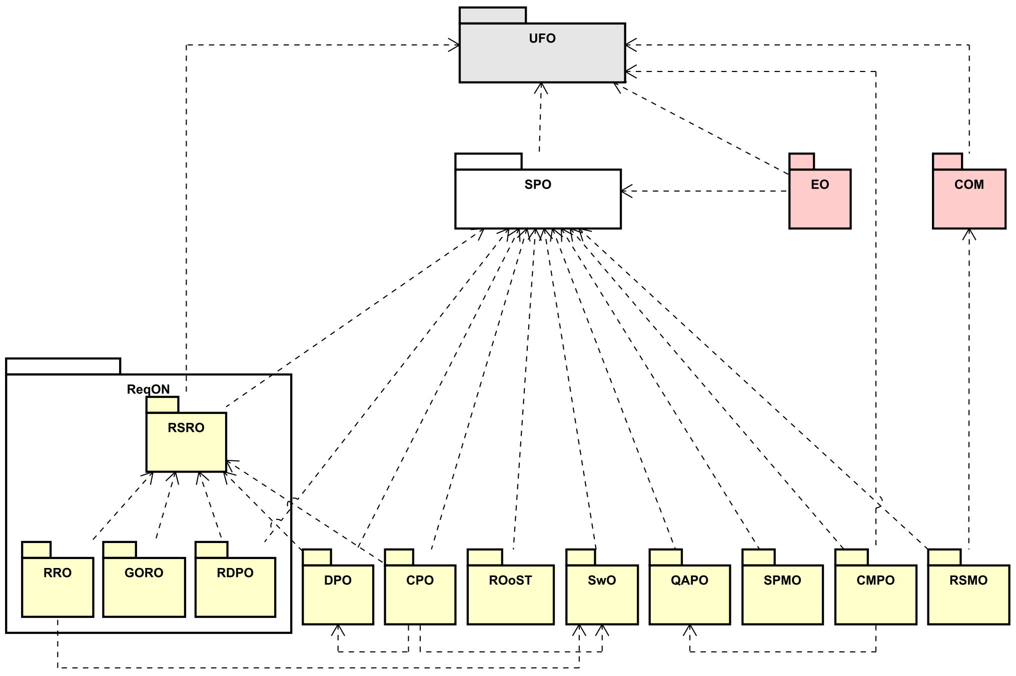


Figure 17 – Graphical representation of the Software Engineering Ontology Network (SEON). Adapted from (RUY et al., 2016).

ultimately grounded in UFO); (ii) provide ontology reusability and increase productivity, supported by core ontologies organized as Ontology Pattern Languages (FALBO et al., 2016); and (iii) solve ontology integration problems by providing integration mechanisms (RUY et al., 2016).

In this version, SEON's architecture is composed by: (i) UFO, as the foundational ontology that grounds all other ontologies in the network; (ii) the Software Processes Ontology (SPO) (BRINGUENTE; FALBO; GUIZZARDI, 2011a) as a core ontology that has a broader scope of the Software Engineering domain and is designed as a central node for the other ontologies; and (iii) the domain ontologies for the main technical Software Engineering sub-domains, e.g., design, coding and testing, and for some management sub-domains, e.g., software measurement, project management, configuration management, and quality assurance.

Figure 17 presents an overview of SEON. Each package represents an ontology and dashed lines represents the dependencies between ontologies. This dependency can be by concept reuse or by specialization. UFO, as a foundational ontology is represented in light gray. SPO, as a core ontology is represented in white. Networked domain ontologies are represented in light yellow and external ontologies are represented in light red. SEON complete specification is available at <https://nemo.inf.ufes.br/projects/seon/>.

Concerning software requirements, ReqON is SEON's ontology sub-network devoted to this topic. The Reference Software Requirements Ontology (RSRO) (DUARTE et al.,

2018) is the main ontology in the requirements domain. It captures the most general notions regarding requirements, which are valid for many Requirements Engineering approaches. The Goal-Oriented Requirements Ontology (GORO) (BERNABÉ et al., 2019) focuses on the basic notions of Goal-Oriented Requirements Engineering. The Requirements Development Process Ontology (RDPO) (RUY et al., 2016) aims at representing the activities, artifacts and stakeholders involved in the software requirements development process. Finally, the Runtime Requirements Ontology (RRO) (DUARTE et al., 2016; DUARTE et al., 2018), addresses the use of requirements at runtime. ReqON is depicted on the left side of Figure 17.

For this thesis, we adopted UFO as the foundational ontology and reused three ontologies that are part of SEON for the development of ROSS and OSDEF. The first one, SPO, as a core ontology that provides general concepts about the software system domain. The second one is the Software Ontology (SwO) (DUARTE et al., 2018), a domain ontology that discusses software artifacts and is based on Zave & Jackson's work. The third one is the Reference Software Requirements Ontology (RSRO) (DUARTE et al., 2018), an ontology that discusses software requirements in the context of the software process. SPO, RSRO and SwO are presented and discussed in the next subsections.

Besides, later on this work, in Chapter 7, we also reuse concepts of other ontologies that are part of SEON, such as the Configuration Management Ontology (CMPO) (RUY et al., 2016) and the Reference Ontology of Software Testing (ROoST) (SOUZA; FALBO; VIJAYKUMAR, 2013). These ontologies are reused to extend the operational versions of ROSS and OSDEF with concepts such as Configuration Item, Change Request and Test Case, which are Artifacts that are part of the to the software system life cycle but are not specifically discussed in either ROSS or OSDEF.

We believe that associating the different types of requirements presented in ROSS and the concepts of defects, faults and failures of OSDEF with the other concepts presented in the ontologies that are part of SEON can improve the quality and coverage of traceability information that is retrieved, since new traceability links are generated and can be queried through the relations that exist between requirements and the software Artifacts mentioned earlier. These links can easily be recovered because the domain ontologies presented in this work were originally designed to be reused with other related ontologies. Besides, this type of coverage of the domain can be beneficial for software organizations that want to improve software systems management. For example, a software engineer can list all the change requests and requirements associated with a Program that is part of a software system with a single query.

Finally, three distinct ways to integrate ontologies into SEON are defined. The first one, which is the one adopted for ROSS and OSDEF, considers that the new ontology is grounded in UFO and also reuses or specializes concepts from SPO or other ontologies in

SEON. In this case, the integration is facilitated, since the ontologies are designed following SEON's recommendations. The second case considers ontologies that are grounded on UFO but that do not reuse or specialize any concept from one of SEON's networked ontologies. In this case, it is necessary to adapt the ontology to be integrated so that it shares the representation pattern and aligns with the existing networked ontologies. The third case considers external ontologies, which are not grounded on UFO. In this case, an ontological analysis of the domain must be performed and the ontology to be integrated must be re-engineered before the integration. The knowledge presented in the ontology must be preserved but the representation strategy of this knowledge might be changed or adjusted, in order to allow for a better integration.

3.5.1 Software Process Ontology - SPO

The Software Process Ontology (SPO) is a core ontology originally developed in (FALBO; BERTOLLO, 2009) and re-engineered based on UFO in (BRINGUENTE; FALBO; GUIZZARDI, 2011a). SPO was built to establish a common conceptualization about the software process. Besides, as a core ontology, SPO provides the general concepts for software processes, to be specialized and reused in domain-specific ontologies. SPO is modularized in 5 sub-ontologies: Process/Activities sub-ontology, Artifacts sub-ontology, Procedures sub-ontology, Resources sub-ontology and Stakeholders sub-ontology. SPO was designed so that its five sub-ontologies could be easily reused during the development of software-related domain reference ontologies. SPO is also the central core ontology of SEON.

Besides, SPO reuses concepts from the Enterprise Ontology, such as Organization and Team. Both concepts are depicted with the prefix *EO* in Figure 19.

For the development of ROSS, we reuse the concepts Artifact, Stakeholder and Hardware Equipment, defined in the SPO. Software Artifacts are objects intentionally made to serve a given purpose in the context of a software project or organization. Moreover, Artifacts can be simple or composite, depending on their mereological structure. A Composite Artifact is an Artifact that is composed by two or more Artifacts. On the other hand, a Simple Artifact is an atomic one, an Artifact that cannot be decomposed in others.

Stakeholders are Agents (a single person, a group or an organization) interested or affected by the software process activities or their results, eventually being responsible for them (e.g., a user or a development team); Hardware Equipment (including Machine), which are physical objects used for running software programs or to support some related action (e.g., a computer or a tablet). Moreover, Stakeholders, can have different roles as part of a software process and can also be understood as Resources, from a project perspective.

Figures 18 and 19 respectively shows the Artifact and Stakeholder sub-ontologies

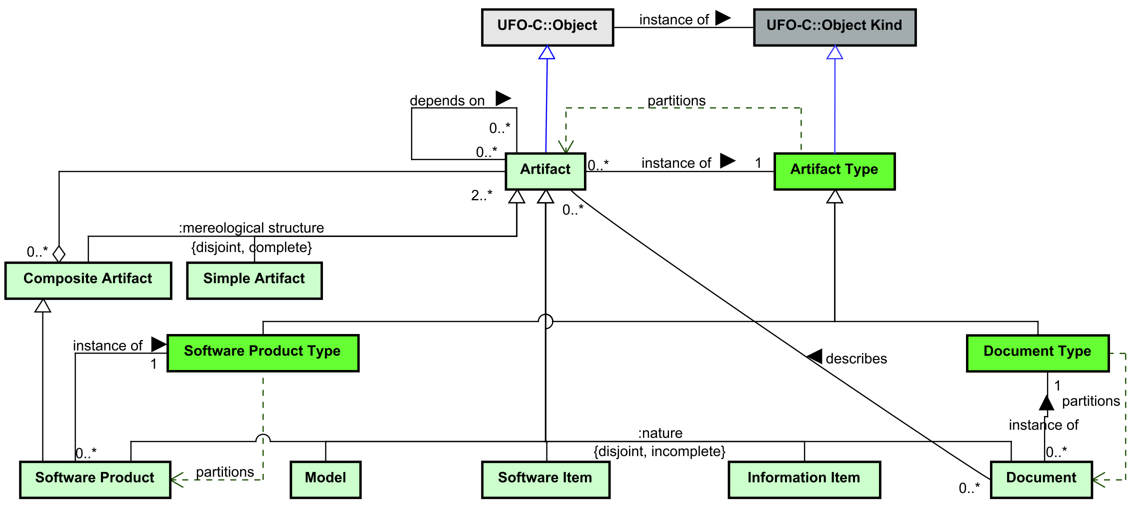


Figure 18 – SPO Artifact sub-ontology (BRINGUENTE; FALBO; GUIZZARDI, 2011a).

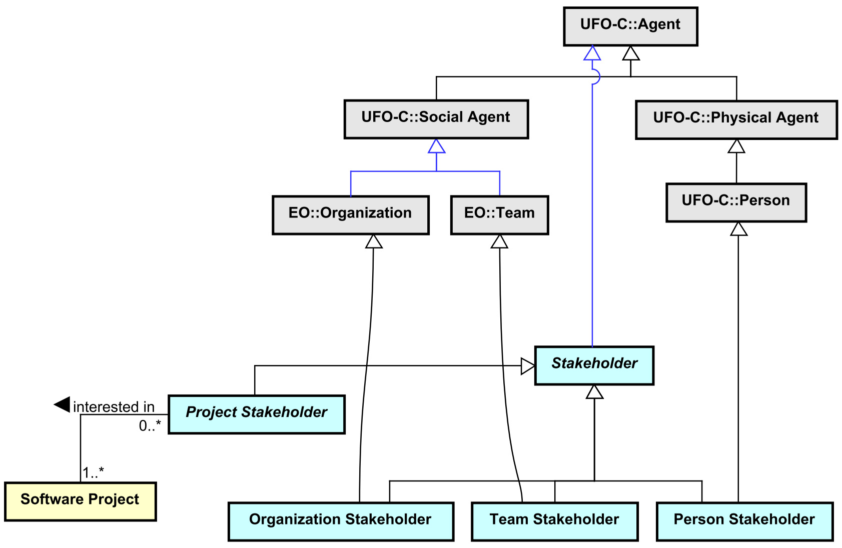


Figure 19 – SPO Stakeholder sub-ontology (BRINGUENTE; FALBO; GUIZZARDI, 2011a).

that are part of SPO.

3.5.2 Software Process Ontology - SwO

The Software Ontology (SwO) (DUARTE et al., 2018), depicted in Figure 20, is a domain ontology created to represent the complex nature of the Artifact "Software". SwO aims to clarify and to establish a common conceptualization about the notion of software, which is in fact composed by several Artifacts, with distinct natures and purposes.

SwO reuses the artifact sub-ontology of SPO (the classes depicted in green in Figure 20). SwO was based on the theory of UFO-B about Events, Dispositions and Objects

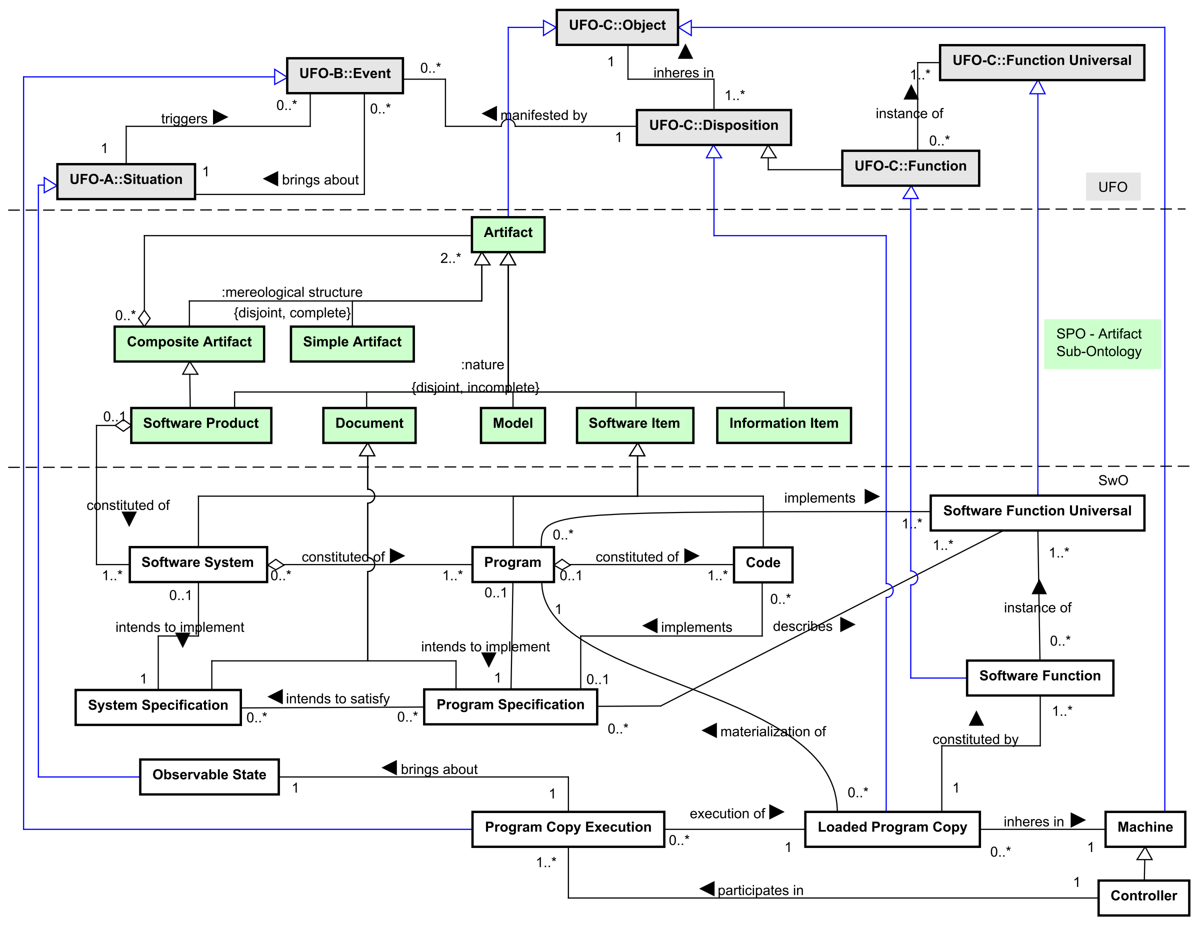


Figure 20 – SwO Conceptual Model (DUARTE et al., 2018).

to extend the Ontology of Software Artifacts (OSA), proposed by Wang et al. (2014b)[9] and Zave and Jackson's work. The Loaded Program Copy (Disposition) represents the Program loaded inside the memory of the Machine (Object) that participates in the Program Copy Execution (Event) as a Controller.

These concepts were not addressed by previous works and are important because they are responsible to "materialize" a Program, connecting it to the Machine and generating an Event that is capable to transform the state-of-affairs in which the Machine exists. This relations needed to be addressed for the development of the Runtime Requirements Ontology (RRO) (DUARTE et al., 2018).

Regarding ROSS, SwO is almost fully reused in the System and Program sub-ontologies of ROSS. It defines several concepts that were very important for the development of this work.

[9] OSA was discussed in Section 2.1.3.

3.5.3 Reference Software Requirements Ontology - RSRO

The Reference Software Requirements Ontology (RSRO) (DUARTE et al., 2018), depicted in Figure 21, is a domain ontology created to represent the ontological nature of the Requirement. Like SwO, RSRO is grounded on UFO and reuses part of the artifact sub-ontology of SPO. RSRO also reuses concepts from the NFR Ontology (GUIZZARDI et al., 2014), which are represented in the model with the *NFR* prefix. A Requirement is defined as a goal that represents the users' needs and expectations (Stakeholder Intention) to be achieved as result of the system development (GUIZZARDI et al., 2014).

Requirements can be functional or non-functional, depending on their nature. Functional Requirements are the ones defining a Function to be available from the target system (e.g., the need for providing a client register or controlling an order). They refer to Software Function Types, i.e., types of functions that the software must provide (e.g., providing a client register, controlling an order). Non-Functional Requirements define criteria or capabilities for the system (e.g., being easy to operate, being in conformance with a standard). A special type of Non-Functional Requirement is Product Quality Requirement, which refers to Quality Characteristics that the product shall present in some degree, such as reliability, usability, efficiency.

Requirements are documented by Requirement Artifacts, an Information Item that describes the Requirement in a proper way. The Requirements Document is composed of Requirement Artifacts and related information (such as models, information sources and varied descriptions). This Document is under the responsibility of the Requirements Engineer, a Stakeholder that conducts the requirements development activities. The Requirements Stakeholder represents the Stakeholders from whom the Requirements are collected and, consequently, are the ones interested in the Requirement Artifacts.

3.5.4 Ontology of Assumptions

Besides the concepts reused from SPO and SwO, we also reused the concepts of World and Machine Assumptions, depicted in Figure 22, that are presented by Wang et al. (2016) in their sub-ontology of assumptions. A World assumption is an assumption about world phenomena, that is invisible to the machine. For example, a meeting scheduling system marks a reserved room for a meeting as occupied even if the meeting participants decided to move the meeting to the cafeteria. In other words, from the perspective of the data persisted in the software system, the room is occupied for the meeting, even if it is empty in the real world and could be allocated for another meeting. Such assumptions constrain the environment in which software system exists. Machine Assumptions are assumptions about a machine's internal behavior, which is only visible to the machine. For example a user may assume that his daily-working backup software system is persisting his work in the cloud every time he clicks the save button, however, if the backup system

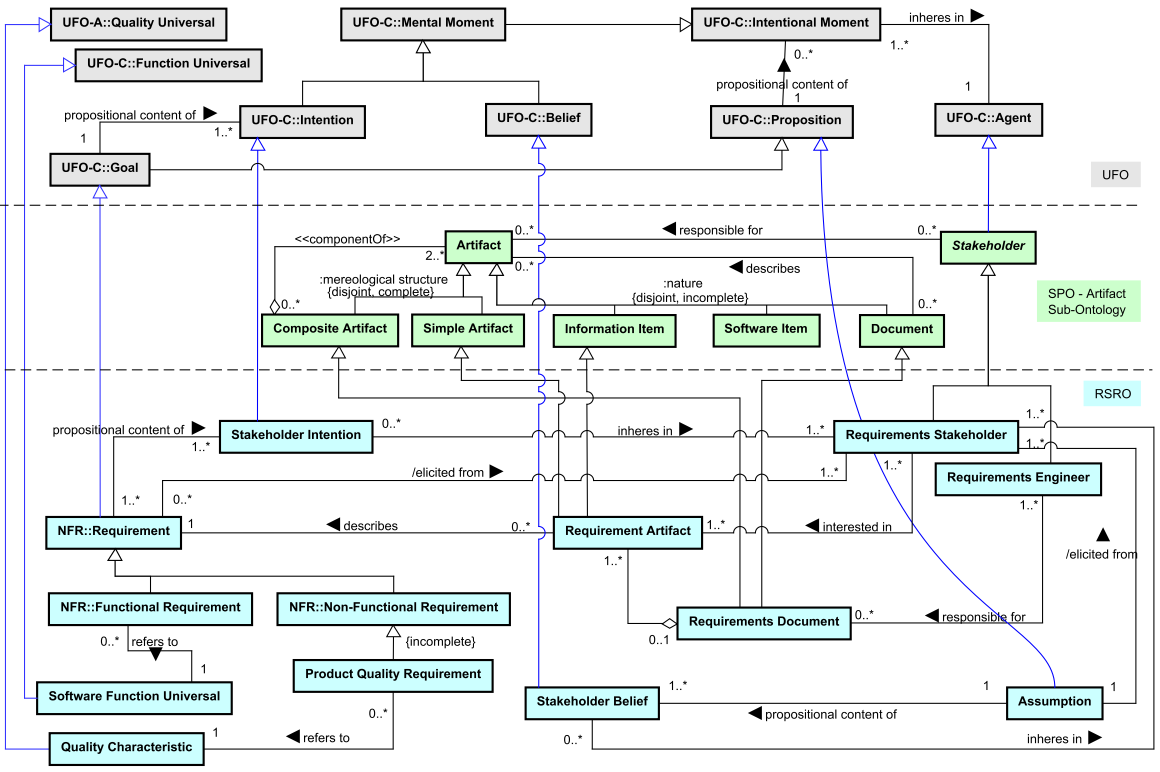


Figure 21 – RSRO Conceptual Model (DUARTE et al., 2018).

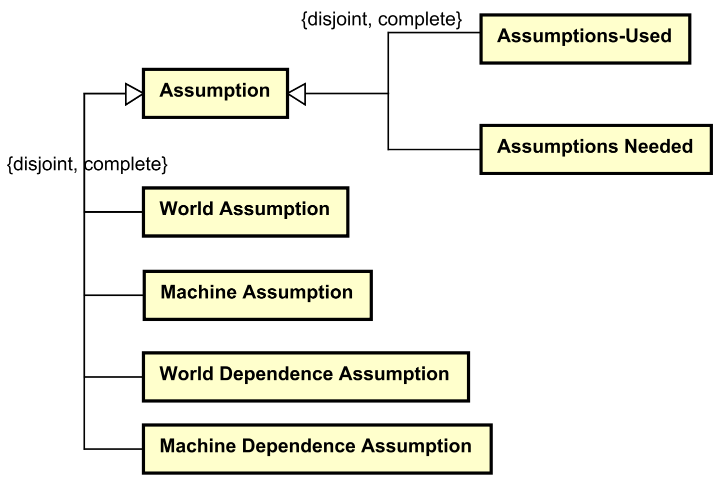


Figure 22 – Representation of the sub-ontology of assumptions proposed by Wang et al. (2016).

is not properly configured, his progress will not be saved.

The other two types of assumptions are created to represent the impact that the world and the machine have on each other. A Machine Dependence Assumption states that an external world phenomenon depends on some machine phenomena while the World Dependence Assumption represents the opposite, a machine phenomenon that depends on a world phenomena. Wang et al. present these new interpretations of the concept of

Assumption as a preliminary ontology of assumptions which, until now, has not been further advanced. Their contribution towards Zave and Jackson's work was heavily focused on the further understanding and development of the concept of Assumption and its implications for RE. Further, they also contributed by proposing a new notion for Specification.

Wang and his colleagues also use the distinction between Assumptions-Used and Assumptions-Needed, in a classification that is orthogonal to the previous one. In other words, all four types of assumptions proposed can be Assumptions-Used or Needed, depending on the situation. However, these assumptions are disjoint, as an assumption cannot be used and needed by the same individual at the same time.

3.6 Chapter Summary

This chapter discussed the ontological foundations that were adopted for the development of this thesis. Firstly we explained what ontologies are, how they are developed and used, the different types of ontologies that exist in the scientific literature and how foundational ontologies, the most general type of ontology, are important for the development of domain ontologies, the type of ontology presented in this thesis.

Secondly we presented UFO and gUFO, respectively, the foundational ontology adopted in this thesis and its 'gentle' (operational) version, which is used to support the evaluation process of ROSS and OSDEF.

Thirdly, we briefly explained the technologies that are used to create and use operational ontologies and that were used to evaluate the ontologies proposed in this thesis.

Lastly, we explained SABiO, the ontology engineering method that was adopted for the development of ROSS and OSDEF and SEON, the Software Engineering Ontology Network that relates SPO, SwO and CMPO the ontologies that were reused for the development of ROSS and OSDEF.

4 A Reference Ontology of Software-based Systems

This chapter presents the first contribution of this work: the Reference Ontology on Software Systems (ROSS). ROSS is a domain reference ontology (GUIZZARDI, 2007), a conceptual artifact that is intended to represent the software systems domain in the best possible way. ROSS was designed to be used as a backbone ontology, into which other software-related more specific ontologies can be merged and reused together. ROSS is based on international standards like ISO 29148 (ISO, 2018), ISO 12207 (ISO, 2017) and ISO 15288 (ISO, 2015). ROSS conceptualize on the different types of software systems requirements, their connections with the external (real-world) environment and with the machines where they are executed. ROSS also discusses the assumptions that are made during the software process. Because of theses characteristics, the operational version of ROSS is intended to be used as a tool for ontology-based requirements traceability. The work presented in this chapter was published as part of (DUARTE et al., 2021).

Section 4.1 briefly discusses the motivations and requirements for ROSS. Sections 4.2, 4.3 and 4.4 present the conceptual model of ROSS, divided in three layers that intend to represent the three parts of the Software Systems domain. Section 4.5 presents other ontologies that are directly related to ROSS. Section 4.6 concludes and summarizes the chapter.

4.1 Motivation and Requirements

As discussed in Chapter 2, in their seminal work (JACKSON; ZAVE, 1995; ZAVE; JACKSON, 1997), Zave and Jackson clarified the nature of RE and demonstrated the importance of certain information items that is often neglected. Their conclusion was the well-known formula: $S, K \vdash R$. Later on (GUNTER et al., 2000), the formula was improved to be used as a reference model for the requirements engineering process, taking into account other relevant software artifacts, such as the Machine (M), as the programming platform and the Program (P), as the unity that is intended to implement the specification.

However, this reference model does not take into account the fact that some of these software artifacts may exist in different levels of abstraction through the software process. For example, in Zave and Jackson's work, requirements are considered as entities that exist only in the external environment, far from the notions of program and machine. Based on Zave, Jackson and Wang's work we have developed SwO and RSRO, domain reference ontologies that, extends these earlier works based on an ontological pattern of

Events defined in UFO-B[1].

Nowadays, there is a clear consensus in modern standards, such as ISO 29148 (ISO, 2018) and ISO 12207 (ISO, 2017), and capabilities models, such as CMMI (CMMI Institute, 2018), that requirements exist in many formats during the software process, being refined from high-level goals for a system of interest to solution-oriented artifacts. In another example, the concept of specification is originally described as an artifact that exists in the interface between the world and the machine. Hence, if we are assuming that requirements exist in many abstraction levels, so should their specifications. For that reason, we intend to revisit and extend and ours previous works by building ROSS on top of these earlier contributions.

Moreover, ROSS is developed following SABiO specification and, because of that, it is based on a set of Competency Questions (CQs) as its functional requirements. The CQs raised for ROSS are listed below:

- CQ1: What are software systems?
- CQ2: How are software systems composed?
- CQ3: What are the types of requirements that exist in a software system domain?
- CQ4: How are these requirements related?
- CQ5: How are these requirements managed?
- CQ6: What are the types of assumptions that are relevant in the software system context?
- CQ7: How are requirements related to assumptions in the software systems domain?
- CQ8: What are the constrains on the software systems domain? How do they impact the organization and the requirements?

For ROSS, the CQs were focused on the gaps that existed in Zave, Jackson and Wang's research and that are intended to be covered by ROSS. CQs 1 and 2 are focused on expanding upon the concept of Software System, that is originally used by Wang in OSA. CQs 3, 4 and 5 are focused on Requirements and based on the knowledge presented in ISO 29148 and ISO 12207. What are the types of requirements? How are these different types of requirements are related with each other? And with the other artifacts. We believe that these questions are relevant for the software systems domain and consequently, for the development of ROSS. However, they are not addressed by Zave and Jackson or by Wang, in their research, since Requirements are only considered as top-level goals. Finally,

[1] SwO and RSRO are properly discussed in Section 4.5

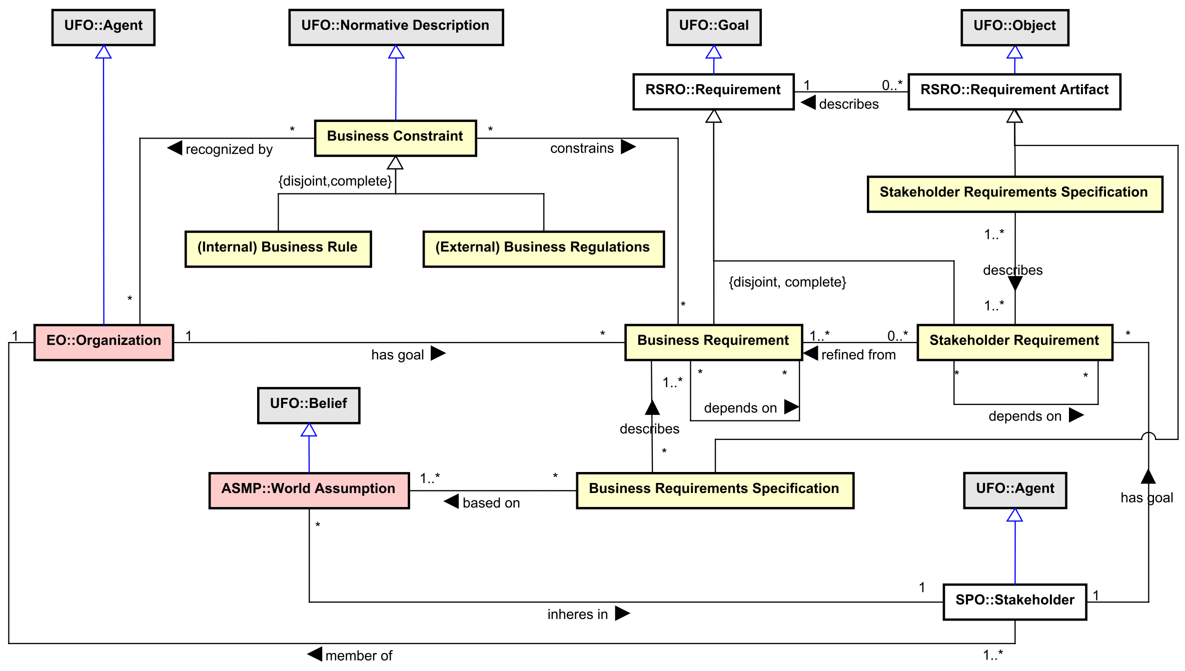


Figure 23 – The Business sub-ontology of ROSS.

CQs 6, 7 and 8 are focused on the assumptions and constraints that exist in the domain. Assumptions are discussed in Wang's work and his definitions are reused in ROSS.

To present the ontology, we adopted SABiO's normative on ontology modularization and divided it in three sub-ontologies. This decision was taken because the software systems domain itself can be divided in three parts, namely: (i) the business environment, in which Agents like Organizations and its members, take Actions and demand Services; (ii) the software system environment, that has the purpose of providing the services that are needed and to connect business and machines; and (iii) the machine environment, in which the machine is able to execute a translation of the specifications (source code) to create a pure logical result, which form the base for the services provided. This division of the domain in layers is also used in ISO standards 29148 (ISO, 2018) and 12207 (ISO, 2017).

4.2 ROSS Business Sub-ontology

The first part of the ontology, depicted in Figure 23, represents the business/organization environment in which the software system exists.

The Organization is a (Social) Agent involving people and other Agents and facilities with an arrangement of responsibilities, authorities and relationships. The concept Organization in ROSS and in other ontologies that are part of SEON are reused from the Enterprise Ontology (prefix EO), an ontology that represents the Enterprise domain and that is external (not part of) SEON. Business Requirements are high-level Goals an Organization has towards the system-to-be. They represent the main reason why a project is initiated, what the software system is intended to achieve, which metrics can be used to measure the project's success or failure (ISO, 2018). To represent the relation between Goals and Agents in ROSS, we created the *has goal* relation. In UFO, Goals are Propositions, in particular, they are the propositional content of an Intention that inheres in an Agent (in our domain, either an Organization or a Stakeholder). The *has goal* relation is then a derived relation associated to the following derivation rule: *given a goal G, agent A, and intention I, we have that has-goal(A,G) iff G is the propositional content of an intention I inhering in A.* This relation also appears in Figure 24, between Stakeholder Requirement and Stakeholder.

Additionally, a specific Business Requirement may *depend on* other Business Requirements. *Depends on* defines the coupling that may exist between Requirements with the same level of abstraction. Furthermore, the *Depends on* relation is individually represented on each of the four types of Requirements presented in ROSS, in order to support and enable horizontal requirements traceability (i.e the type of requirements traceability between artifacts at the same level of abstraction). For example, achieving a Stakeholder Requirement STREQ001 may be directly necessary for achieving another Stakeholder Requirement, STREQ002. In this context, the two Stakeholder Requirements are coupled and this relationship can be captured in the reference model, and traced. The coupling level between the requirements of software system is an intrinsic property and is heavily dependent to the conceptual modeling activities performed for software system development.

Business Requirements, as goals of an Agent, are usually described as some type of Artifact to be properly used, traced and maintained by the Organization. This Information Item[2] is named Business Requirements Specification (BRS) and, as a product of the system development process, it is created very early and will exist during the entire life of the system.

Moreover, it is important to understand that although specifications are usually defined (mainly in textbooks) as document-type artifacts, they are not, necessarily, formal, documented descriptions of requirements. For example, the description of the daily routine of an organization's office is a powerful source of requirements, with the support of proper World Assumptions, which are ontologically defined as Propositions about the domain of the system-to-be are part of the organization environment. Since World Assumptions are Propositions, they represent the propositional context of Intentions of the Organization over the Software System. These assumptions, and the other that will be further discussed in this work, may or may not be explicit during the system development process.

Zave and Jackson (1997) defended that Assumptions should be treated as first class citizens in every software system project, being documented and managed as any

[2] Information Items are part of the Artifact sub-ontology of SPO and are defined as a relevant piece of information created for human use in an performed process (see Figure 20).

other configuration item. Based on this, Wang et al. (2016) proposed a small ontology of assumptions (cf. Section 3.5.4), which is being reused in ROSS with the prefix ASMP. In other words, a specification is a possible description of requirements *based on* a set of assumptions, i.e., different assumptions used will result in different specifications and, if the assumptions used are incorrect or incomplete, the specification will not be able to properly describe the requirements.[3] This phenomenon is also true when requirements are in its lower levels of abstraction, as they are properly refined towards the solution of the problem that gave origin to the software process.[4]

Business Requirements, are constrained by Business Constraints, which are Normative Descriptions that are recognized by the Organization. Business Constraints describe conditions to be imposed in conducting the business process. In ROSS, we define two types of Business Constraints that are extremely relevant for an Organization: Business Rules and External Regulations. Business Rules are Normative Descriptions that define a policy, guideline or practice that constrains some aspects of a business project and its intended results. Moreover, Business Rules can be the origin/act as constraints of several types of requirements (WIEGERS; BEATTY, 2013). For example, a software-factory organization that has the internal policy of producing applications that are optimized for a certain type of platform will have to create and implement specific requirements to satisfy this rule.

On the other side, External Regulations are Normative Descriptions that exist outside of the organizational environment and that cannot be controlled by it. As examples we can mention laws, business and engineering standards, market trends and even external interface requirements. Moreover, alongside with Business Rules, External Regulations are important for any type of Organization because they are capable of constraining the Business Requirements. In line with CMMI (CMMI Institute, 2018), we argue that the relationships between higher-level Business Requirements and these Normative Descriptions are extremely important for the software process, since they define constraints for the software system and for the Organization developing it. Because of that both Business Rules and External Regulations must be managed and traced.

As Business Requirements are high-level Goals that exist in an organization-level of a business (ISO, 2018), they tend to exist far from the solution that is desired. For example, an Organization that desires to improve team productivity by reducing their dependence on spreadsheets may decide to build their own team management tool, or to buy one, such as Monday.com. At this point, the Organization has intentions towards acquiring a team management tool, but they do not know what solution to implement. If they decide to create their own tool, a software system project can be initiated. Because of this "distance" that exists between the initial need and the domain of the solution, a

[3] One can say that specifications are derived from requirements, as goals, with the support of assumptions.
[4] This will be further discussed in the last part of the ontology.

new type of requirement that is directly related to them but that is closer to the solution domain must be captured.

Stakeholder Requirements are statements of the needs of the Stakeholders. Stakeholder is a concept reused from SPO, it is defined as an Agent interested or affected by the software process activities. A Stakeholder may be a member of the Organization where the system will be implemented or an analyst that is responsible for the process. Stakeholder Requirements are *refined from* Business Requirements and can be understood as the stakeholder's point of view towards the existing Business Requirements. In their work, Zave and Jackson stated that requirements refinement is concerned with identifying the aspects of a requirement that cannot be guaranteed or effected by a computer alone and augmenting or replacing them until they are fully implementable. Moreover, goal refinement is directly related to the requirements traceability process, since it provides a natural mechanism for structuring connections between high-level requirements to low-level requirements (LAMSWEERDE, 2001).

The concept of refining Requirements (Goals) exists in the scientific literature since the early 80s. Mostow's goal refinement (MOSTOW, 1983) is defined as an "operationalization" of goals, a process of converting a goal into an "executable" procedure (in our domain, a Program).[5] However, this concept of operationalization does not provide the semantics necessary for our *refined from* relation, because the refinement of Business Requirements to Stakeholder Requirements does not fit in the *converting a goal into an executable procedure* definition, proposed by Mostow. In this context, the *refined from* relation denotes a type *Reduction* relation (DARDENNE; LAMSWEERDE; FICKAS, 1993). *Reduction* denotes that to achieve a Goal g_1, possibly with other Goals $g_2, \ldots, g_n$, is among the alternative ways of achieving the higher-goal G. For example, the Business Requirement *Reducing operation costs by 10%* can be achieved by achieving a combination of several Stakeholder Requirements, that will also be refined into other "lower-level" requirements. *Refined from* also denotes a historical dependence (FONSECA et al., 2019)[6] that exists between Stakeholder Requirements and Business Requirements, in the sense that Stakeholder Requirements can only be specified after the Business Requirements are properly defined. In addition, *refined from* is not a parthood relation, since the sum of all Stakeholder Requirements $g_1, g_2 \ldots g_n$ is not the Business Requirement G. In other words, a Business Requirement can be refined in more than one set of Stakeholder Requirements.

Moreover, since all Business Requirements and Stakeholder Requirements are defined as Goals, the same relation is also used to represent the relations between Stakeholder, System and Program Requirements, that are respectively presented in the system and machine sub-ontologies of ROSS.

[5] Goal refinement through operationalization is also presented in KAOS (DARDENNE; LAMSWEERDE; FICKAS, 1993).

[6] A historical dependence is a non-descriptive relation where the truthmaker of the relation is an Event.

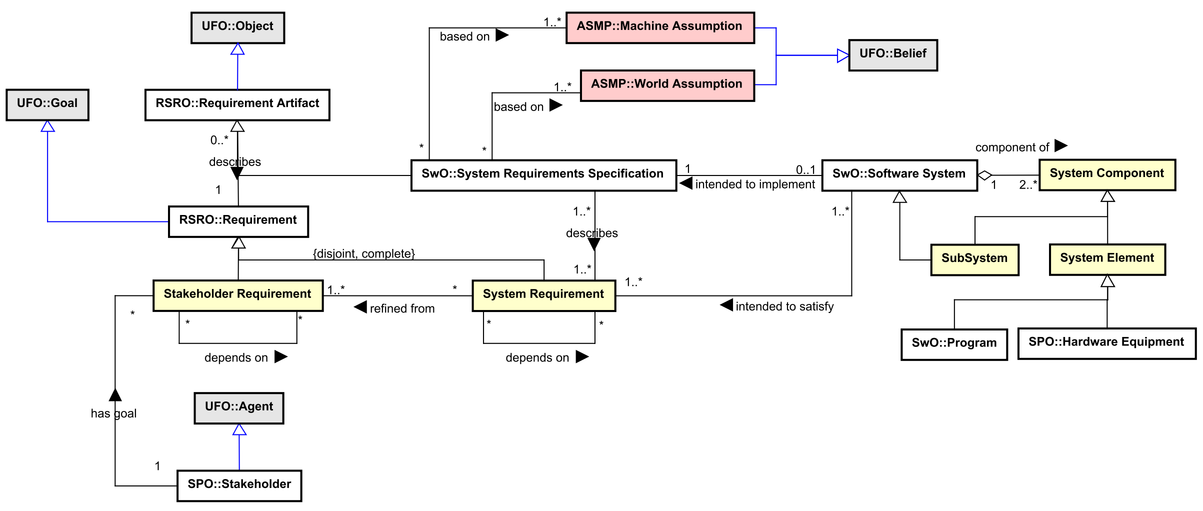


Figure 24 – The System sub-ontology of ROSS.

Furthermore, as Stakeholder Requirements are 'crispier' than Business Requirements but still exist in business level,[7] they serve as a bridge between Business Requirements and the other types of requirements that are solution-oriented. Finally, Stakeholder Requirements are also classified as Goals, which are described in specific Information Item, called Stakeholder Requirements Specification (StRS).

4.3 ROSS Systems Sub-ontology

The second part of the ontology, depicted in Figure 24, is centered around the concept of Software System, which acts as an interface between the Machine and the Environment. In their work, Zave and Jackson (1997) briefly define a Software System as a general artifact with manual, automatic and even abstract (data) components, separating it from the concept of Machine. In a more general definition, a Software System is defined by ISO 24765 (ISO, IEC, 2017b) as a *combination of interacting elements organized to achieve one or more stated purposes*. SWEBoK extends this definition by explaining the concept of Software System as a complex and heterogeneous artifact, as it can be composed by many System Elements, such as software, hardware, firmware, people, data and even other systems. For ROSS, we reused the concept of Software System that is presented in SwO, which is based on Zave and Jackson's definition.

From an ontological point of view, software systems are complex social Artifacts, composed of other artifacts as System Components. A System Component in turn, can be either a System Element or a (sub)system.

For example, Microsoft Windows 10 is an (operating) system composed by many

[7] ISO 29148 defines Stakeholder Requirements as Information Items that exist in the Business Operational Level of an Organization, while Business Requirements are part of the Business Organization Level.

subsystems, such as the memory-management system, the user interface and the security system.

System Elements are also artifacts that are used by Software Systems during their operation, such as Programs or Hardware Equipment, that are necessary for system operation, such as servers, sensors or peripherals.

Furthermore, the notion of software proposed here, in line with (WANG et al., 2014b; DUARTE et al., 2018), allows for a software system to be composed by many artifacts that exist in different levels of abstraction, each with its own identity and purpose. As discussed by these authors, the simple term "software" is heavily overloaded. Moreover, Software Systems and Programs are Individuals, which, in a sense "behave like types" given that they define properties that are repeatable in their copies. For example, Microsoft Outlook is Microsoft's well-known mailing software that, as an Individual, has properties and an unique identity, which makes it different from another individual of the same type (E-mail Client Software, e.g., Mozilla Thunderbird). Nonetheless, it can share a number of properties with their copies, in a way that is analogous to how individuals of the same type share the same properties, each of which, however, having a unique identity.

As a type of software Artifact (FALBO; BERTOLLO, 2009; BRINGUENTE; FALBO; GUIZZARDI, 2011a), Software Systems are also developed based upon a set of requirements. System Requirements are solution-oriented Goals for the system-of-interest, which are based on background information about the high-level objectives to be achieved by a solution (ISO, 2018). System Requirements are different from Business Requirements and Stakeholder Requirements since they exist in a solution perspective, whereas stakeholder and business requirements exist in a problem perspective. However, System Requirements are derived from Stakeholder Requirements. This relation between both types of requirements provides the connection between the business and the system sub-ontologies.

Furthermore, similarly from their higher-level counterparts, System Requirements are *described* in an Information Item called System Requirements Specification (SyRS) (ISO, 2018). In this context, Software Systems are *intended to implement* the SyRS and *intended to satisfy* the System Requirements, which are described in the SyRS. Both *intended to implement* and *intended to satisfy* relations are originally discussed in OSA. The first one, *intended to implement*, links the software system to its specification and is derived from someone's intention towards the software system being developed. More precisely, Stakeholders from an Organization have the Intention that the Software System (an Artifact) is capable of providing the functionalities and capabilities described in its specification. The second one, *intended to satisfy*, is also derived from these Intentions of Stakeholders and represent the fact the System Requirements denote the essential functions of the Software Systems, the goals that need to be fulfilled. This pattern of relations also occur for the concepts of Program, Program Specification and Program Requirements, in the machine

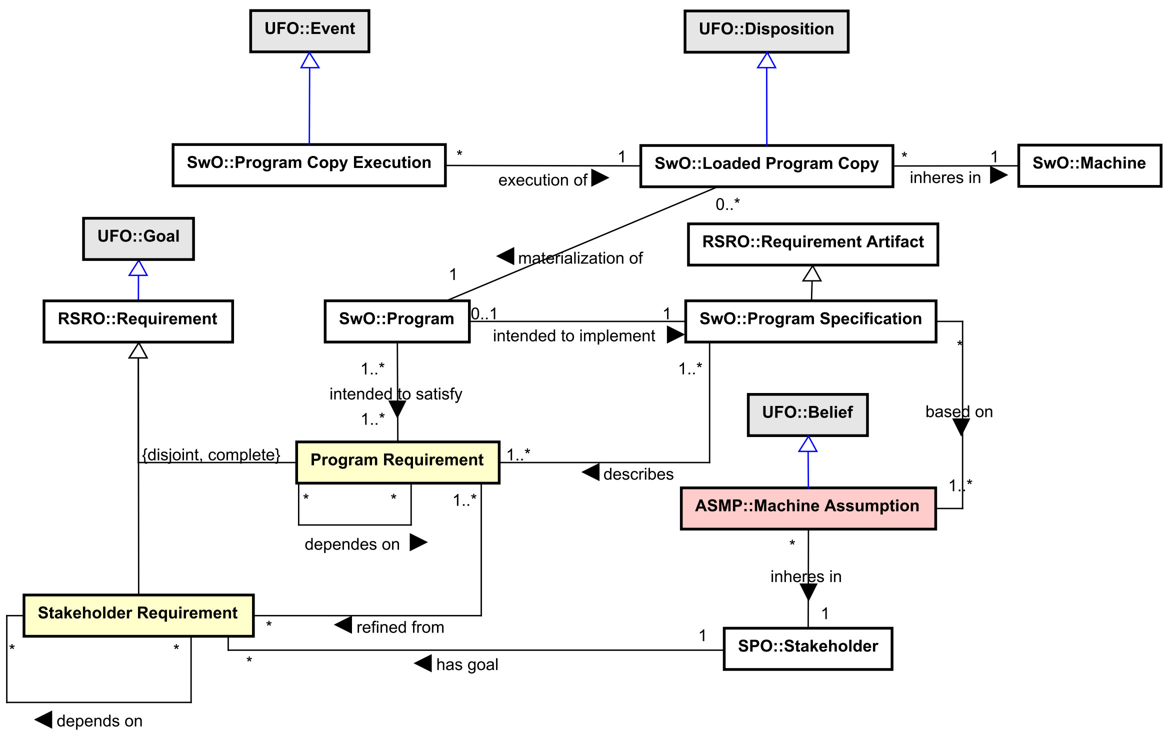


Figure 25 – The Machine sub-ontology of ROSS.

sub-ontology of ROSS.

Moreover, as a software system can be composed by distinct System Elements, the SyRS compiles, in a technical level, requirements, capabilities and constraints of the system-of-interest as a whole. Because of that, it *depends on* two types of assumptions, namely, the World Assumption, that was previously presented and the Machine Assumption, i.e., an assumption about the machine's internal operations, that are only visible to the machine. In other words, for the SyRS to be created, it depends on assumptions about the environment and about the machine.

4.4 ROSS Machine Sub-ontology

Finally, the last part of the ontology, presented in Figure 25, represents the parts of a software system that exist inside a Machine and is focused on the concept of Program.

Wang et al. (2014b) promote an extensive discussion and a reference ontology of software artifacts.[8] Based on Wang et al.'s work and in order to capture the complex nature of software, in SwO (DUARTE et al., 2018), we argue that a Program is defined as an Artifact produced during a software process and having the purpose of generating a result in the environment, through its execution in a Machine (WANG et al., 2016). Moreover, Programs are artifacts constituted by source code, although not being identical

[8] In line with Wang et al.'s work, we avoid to use the word *Software* and prefer to use a specific terms proper to each situation, e.g., Program, Machine or Software System.

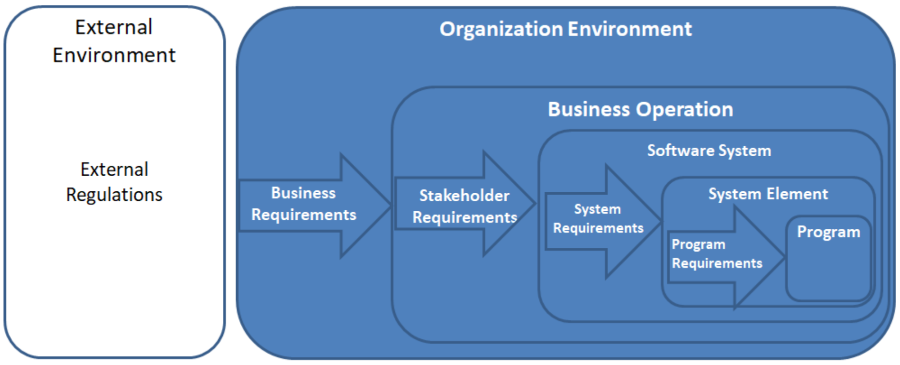


Figure 26 – Adaptation of an example of requirements scope in a business context. Figure originally presented in ISO 29148 (ISO, 2018).

to code. The consequence of that is that the source code, as a sequence of symbols, can be altered without changing the identity of the Program. In this context, Programs are System Elements related to the Machine. However, Programs are considered abstract because they are not physical objects, like a printer or a circuit board, although they are artifacts created for a specific purpose. Besides, despite of its abstract nature, Programs and other artifacts, like laws, are designed and capable of producing results (affecting) in the real (concrete) world.

Programs can only fulfill their purpose when loaded (as Loaded Program Copy) and executed as Events, called Program Copy Execution, which occur inside a Machine. Moreover, the purpose of the Program is directly related to its identity. In a very simple example: changing variable names changes the set of expressions (i.e. the code of the Program) and it may even change how code is loaded inside the Machine, yielding Loaded Program Copies with different characteristics. This type of change, however, does not affect the identity of the Program, since it does not affect its requirements (WANG et al., 2014b).

Furthermore, as artifacts produced through a development process, a Program *is intended to implement* a Program Requirements Specification that describes the Program Requirements related to such Program. In this context, we can say that the Program[9] *intended to satisfy* the Program Requirements. Program Requirements are the lower-level goals for the part of a system that is commonly understood as software. In other words, they are solution-oriented goals that are refined from higher-level requirements, such as stakeholder and system requirements (ISO, 2018; BOURQUE; FAIRLEY et al., 2014), and are focused on a possible solution for the computational part of the system-to-be. Figure 26 presents an adaptation of a figure presented in ISO 29148 (ISO, 2018), depicting how requirements exist in different levels of abstraction, inside an organization, and how they are derived from high-level organization needs to solution-specific goals.

Moreover, as mentioned earlier in this section, a Program Specification is not

[9] For a deeper discussion about the concept of Program, please see (DUARTE et al., 2018).

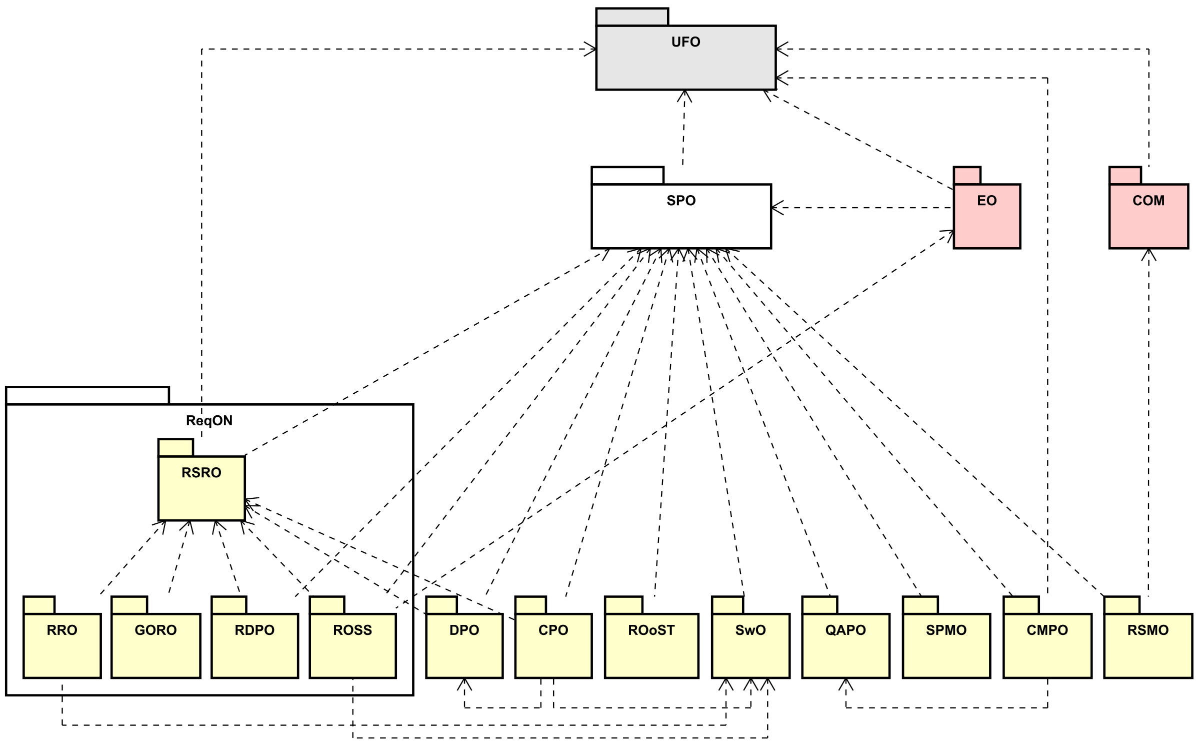


Figure 27 – Integration of ROSS to SEON's ReqON sub-network.

necessarily a formal document in natural language about requirements. As in the original formula proposed by Zave and Jackson (1997), Program Specification is heavily related to the assumptions that exist in the context of the Machine. For example, the source code of a Program could be considered a translation of a Program Specification to a machine-readable language that will be derived from the Program Requirements. Because of that, it will heavily depend on the assumptions (Machine Assumptions that the developers have towards the programming platform, i.e., the Machine).[10] More precisely, different developers, with different Machine Assumptions will produce different implementations, which can satisfactorily or not achieve the same requirements.

Finally, regarding SEON integration, ROSS is grounded on UFO and already specializes concepts from three networked ontologies, SPO, SwO and RSRO. Due to that, no further ontological analysis or re-engineering processes are necessary. ROSS specializes three concepts from SPO, seven from SwO and one from RSRO. Furthermore, as it is focused on requirements, ROSS must be integrated into the ReqON sub-network of SEON. Figure 27 depicts ROSS integrated to SEON as part of the ReqON sub-network. The color scheme adopted in Figure 27 is the same one used in Figure 17, in Section 3.5.

[10] Wang et al. defend that even the 'plan about a program' that exists in the mind of the developer can be considered a type of abstract specification.

4.5 Related Works

This Section presents other ontologies that are directed related and influenced de development of ROSS.

4.5.1 Goal Oriented Requirements Ontology (GORO)

GORO (NEGRI et al., 2017; BERNABÉ et al., 2019) is a reference ontology created to represent the Goal-Oriented Requirements Engineering (GORE) domain. GORE emerged as an alternative Requirements Engineering, it is concerned with the use of goals for eliciting, elaborating, structuring, specifying, analyzing, negotiating, documenting, and modifying requirements (LAMSWEERDE, 2001).

GORO was developed based on nine GORE languages presented in the literature, such as iStar (YU, 1995; YU, 1997), KAOS (DARDENNE; LAMSWEERDE; FICKAS, 1993), Tropos (BRESCIANI et al., 2004), Techne (BORGIDA et al., 2009) and others. GORO is composed of three modules/sub-ontologies. Figure 28 depicts the first one, which represents the concepts of Goals and Assumptions. Figure 29 depicts Tasks, as Plans and their relations to Goals. Finally, a third one includes Obstacles and Contribution links as Relators. In the context of GORE, Goals are a central concept of every GORE language, they represent desirable conditions and are associated to Agents that seek to achieve such Goals. Moreover, GORO reuses the concept of Goal from UFO-C and specializes it to define a Goal-Based Requirement. Tasks are Plans that operationalize Goals. In other words, Goals can be operationalized into one or many tasks. Finally, Obstacles are anti-goals, entities that captures undesirable conditions.

In comparison to ROSS, GORO is also grounded on UFO and also reuses concepts from SwO and RSRO, however, it is focused on representing Requirements Engineering with the GORE perspective. In other words, GORO reuses UFO and the ontologies from SEON to define concepts that are usually proposed by the GORE literature, such as Goals, Softgoals, Assumptions and Tasks. Furthermore, from a foundational perspective, GORO is heavily based on the concepts and ontological patterns discussed in UFO-C, the part of UFO that is focused on social aspects.

4.5.2 Core Ontology of Requirements

The Core Ontology for Requirements (CORE) (JURETA; MYLOPOULOS; FAULKNER, 2008; JURETA; MYLOPOULOS; FAULKNER, 2009) was created to conceptualize about stakeholders communication with the Software Engineer. CORE presents four modalities of the mental states underlying the communication acts between stakeholder and software engineer: belief (B), desire (D), intention (I), and attitude (A). The engineer associates a

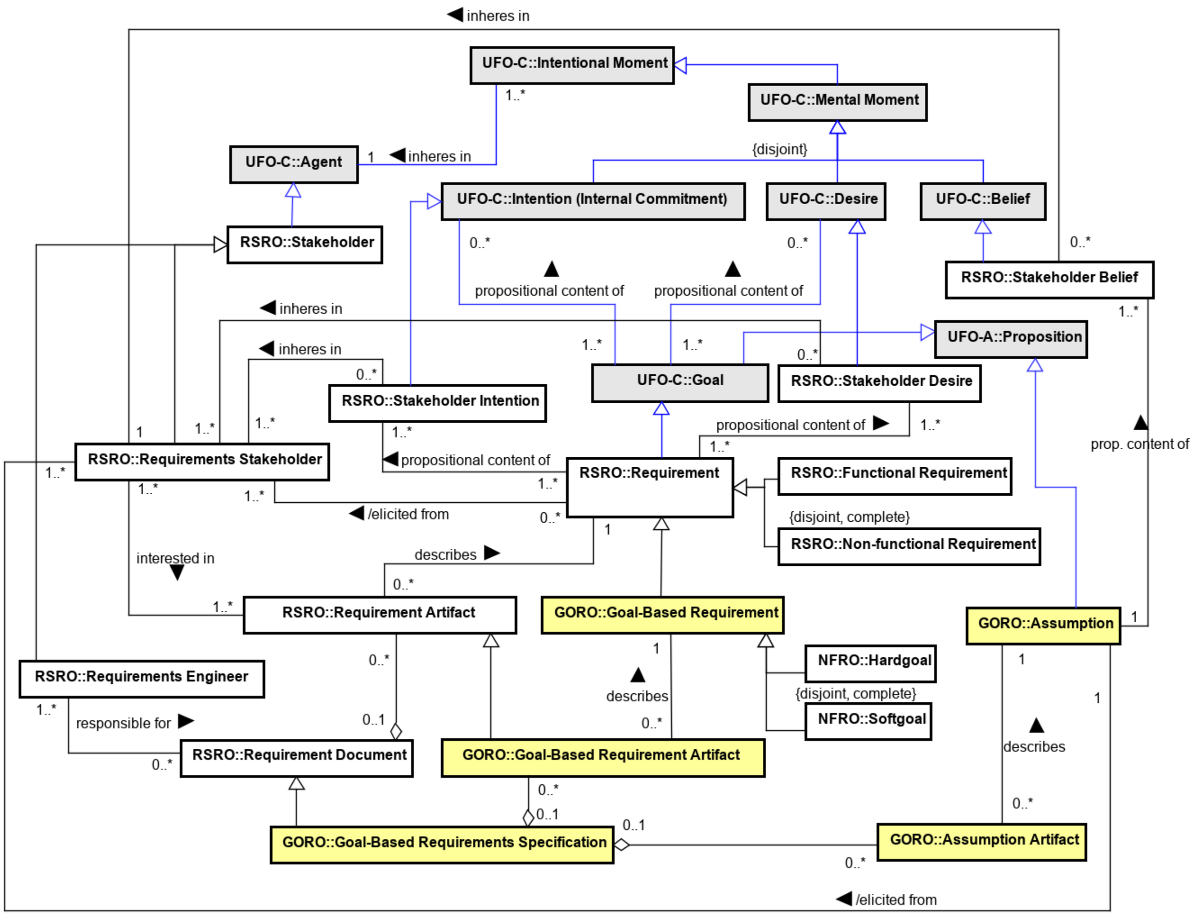


Figure 28 – GORO first sub-ontology, which depicts Goals and Assumptions.

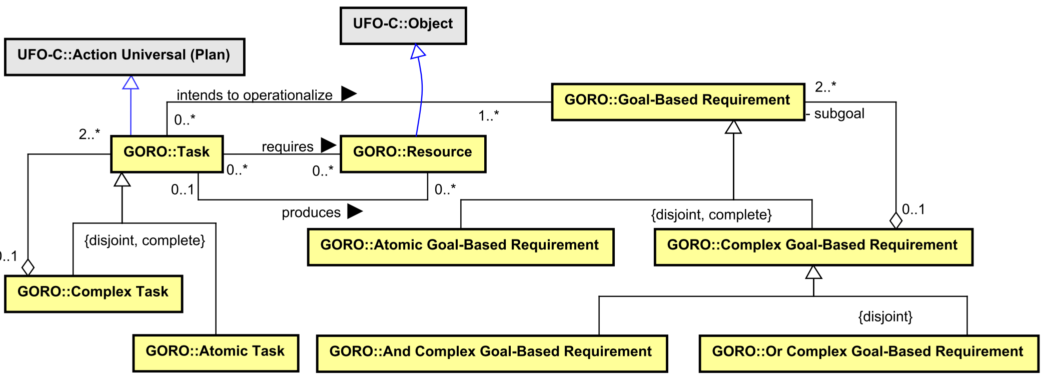

UFO-C::Object
UFO-C::Action Universal (Plan)
intends to operationalize
GORO::Goal-Based Requirement
subgoal
GORO::Task
requires
GORO::Resource
produces
{disjoint, complete}
{disjoint, complete}
GORO::Atomic Goal-Based Requirement
GORO::Complex Goal-Based Requirement
GORO::Complex Task
{disjoint}
GORO::Atomic Task
GORO::And Complex Goal-Based Requirement
GORO::Or Complex Goal-Based Requirement

Figure 29 – GORO second sub-ontology, which depicts Tasks and their relations to Goals and Resources.

modality to the content of a given communication act, and then proceeds to determine if the obtained result is an instance of a concept or relationship in the core ontology.

The concepts of the core ontology are Goal, Softgoal, Quality Constraint, Plan, and Domain Assumption. The relationships between these concepts are: *attitude-based optionality* and *preference*, *justified approximation*, and *non-monotonic consequence*. Moreover, the concepts of Goal and Quality Constraint are introduced into CORE to cover the classical taxonomic dimension for the requirement concept: the distinction between the notions of functional and nonfunctional requirement. Attitudes are given by expressive speech acts. Plans denote the content of communicated intentions of stakeholders in order to satisfy Goals. Domain Assumptions denote the content of an assertive speech act and are directly associated with the beliefs of the stakeholder. Softgoals are Goals that cannot be satisfied to the ideal extent, not only because of subjectivity, but also because the ideal level of satisfaction is beyond the resources available to (and including) the system. Due to its concepts that are intrinsically related to the Goal-Oriented Requirements Engineering domain and its support on DOLCE, CORE was adopted as a conceptualization for the creation of the GORE language Techne (BORGIDA et al., 2009).

In comparison to ROSS, the Core Ontology for Requirements (CORE) is also based on the knowledge presented in Zave and Jackson's work. However, CORE is grounded in DOLCE (MASOLO et al., 2003), although a part of DOLCE that is similar to the conceptualization presented in UFO. Furthermore, CORE conceptualizes about the communication between stakeholders and software engineers and not about the software systems domain.

4.6 Chapter Summary

This chapter presented the first contribution of the thesis, the Reference Ontology of Software Systems, ROSS. ROSS is a domain ontology grounded on UFO and implemented under gUFO which is divided into three sub-ontologies, that represents three parts of the Software System environment.

ROSS contributes to the software research field by representing the nature of a software system, and the entities that are directly and indirectly related to it. Software systems are not simple programs/computational artifact as it was thought decades ago (LEHMAN, 1980). They are socio-technical artifacts, capable of transforming our world, even if not in an agentive way,[11] but based on the assumptions that we have over them. ROSS was created over an ontological analysis of the scientific literature and modern industrial standards to improve Zave & Jackson's and Wang's works, as a complete reference model for software systems focused on the different types of requirements, their representations as specifications, the assumptions used in the process and the other Artifacts that exist

[11] In UFO, Agents are entities capable of performing Actions, based on their Goals and Intentions.

around them. Moreover, based on its modularized structured, designed to represent the three parts of the environment in which a software system exists, ROSS was intended to be used as a backbone ontology, into which other more domain-specific ontologies can be merged and reused, in order to represent more specific parts of the software system domain.

In Section 4.5 we discussed GORO and the Core Ontology for Requirements. Ontologies about the software requirements domain and that are related to ROSS through Zave and Jackson's work. GORO is a domain ontology about goals and the domain of Goal-Oriented Requirements Engineering. GORO is also grounded on UFO and is a networked ontology of SEON. The Core ontology for Requirements focus on the conceptualization of speech acts between stakeholders and software engineers. CORE is grounded on DOLCE and discusses concepts as intentions, goals and beliefs, which are also depicted in UFO-C. Besides, CORE was adopted for the development of the GORE modeling language Techne (BORGIDA et al., 2009).

5 An Ontology of Software Defects, Errors and Failures

This chapter presents the Ontology of Software Defects, Errors and Failures (OSDEF). OSDEF is a domain reference ontology that presents an ontological analysis of software systems failures, defects, faults and errors, which are often collapsed in the commonly used term *software anomaly*, that is usually used to denote a situation in which a Software System deviates from its expected behavior. OSDEF was based on the international standards such as the Standard Classification for Software Anomalies (IEEE, 2009) and the Standard for System, Software, and Hardware Verification and Validation (IEEE, 2016).

OSDEF is intended to represent Artifacts, Agents and Events related to Failures in Software Systems. This part of the domain is very specific and is not discussed in ROSS or any other ontology that is part of SEON. Due to that, OSDEF is also a very specific ontology, in the sense that it was designed to be used, in the requirements traceability context, together with ontologies like ROoST (an ontology about software testing) and CMPO (an ontology about configuration management), as part of our ontology-based reference model for requirements traceability. Moreover, OSDEF could also be reused with ontologies of the cybersecurity domain. The work presented in this chapter was published as part of (DUARTE et al., 2018).

Section 5.1 briefly discusses the motivations and requirements for OSDEF. Section 5.2 presents the reference conceptual model of OSDEF. Section 5.3 presents related works. Finally, Section 5.4 concludes and summarizes this chapter.

5.1 Motivation and Requirements

As previously mentioned, the term *anomaly* and others, such as *bug* or *glitch*, are commonly used within the Software Systems domain to represent a situation in which the Software System, or a part of it, behaves in an abnormal, irregular or even inconsistent way. This usually collapses a variety of concepts with distinct ontological natures in a single term that gives very few or even no context to the real problem. This kind of issue, is very common and it is know on the ontology-based conceptual modeling literature as the *construct overload problem* (GUIZZARDI, 2005).

Because of that, OSDEF was developed to provide an ontological conceptualization of the different types problems that exist throughout the software system operation.

To elaborate on these different types of entities we followed SABiO's directive and raised a set of Competency Questions (CQ) for OSDEF. CQs were raised and refined in a highly-interactive way, through analysis of the aforementioned international standards and through several meetings with ontology experts. The CQs raised for OSDEF are listed below:

- **CQ1:** What is a failure?
- **CQ2:** What is a defect?
- **CQ3:** What is a fault?
- **CQ4:** What is an error?
- **CQ5:** What is a usage limit?
- **CQ6:** In which type of situation can a failure occur?
- **CQ7:** What are the situations that result from failure?
- **CQ8:** What are the cases of failures?

CQs 1 to 5 are related to the concepts that usually are collapsed under the term "Software Anomaly". The ontology is intended to answer the ontological nature of each concept that are commonly used to denote problems and situations that deviate from expected in Software Systems. CQs 6 and 7 are about the state of affairs that is related to the occurrence of a software system failure, a loss event. Both CQs are based on the concept of Situation that are part of the ontological pattern of Events from UFO-B. Finally, CQ 8 was raised because of the necessity for the ontology to represent not only the sub-types of Software System failures, but also the other entities that are related to these sub-types.

5.2 OSDEF Reference Model

Figure 30 depicts the conceptual model of OSDEF. The central concept of our ontology is Failure, since it is the occurrence of a failure that is usually perceived by an agent operating the software system. As defined in standards (IEEE, 2009; IEEE, 2016; ISO, IEC, 2017b) and employed in general in the scientific literature (DELFRATE, 2012), Failures are Perdurants (Events). In that respect, the conceptual basis provided by UFO can help us to understand how failures occur as events during the execution of software. In a software context, a Failure is defined as an event in which a program does not perform as it is intended to, i.e., an event that negatively impacts these relevant goals of stakeholders that motivated the creation of that software (GUIZZARDI et al., 2013). As Events, Failures can cause other Failures in a chain of Events (e.g., a severe failure in a web server such

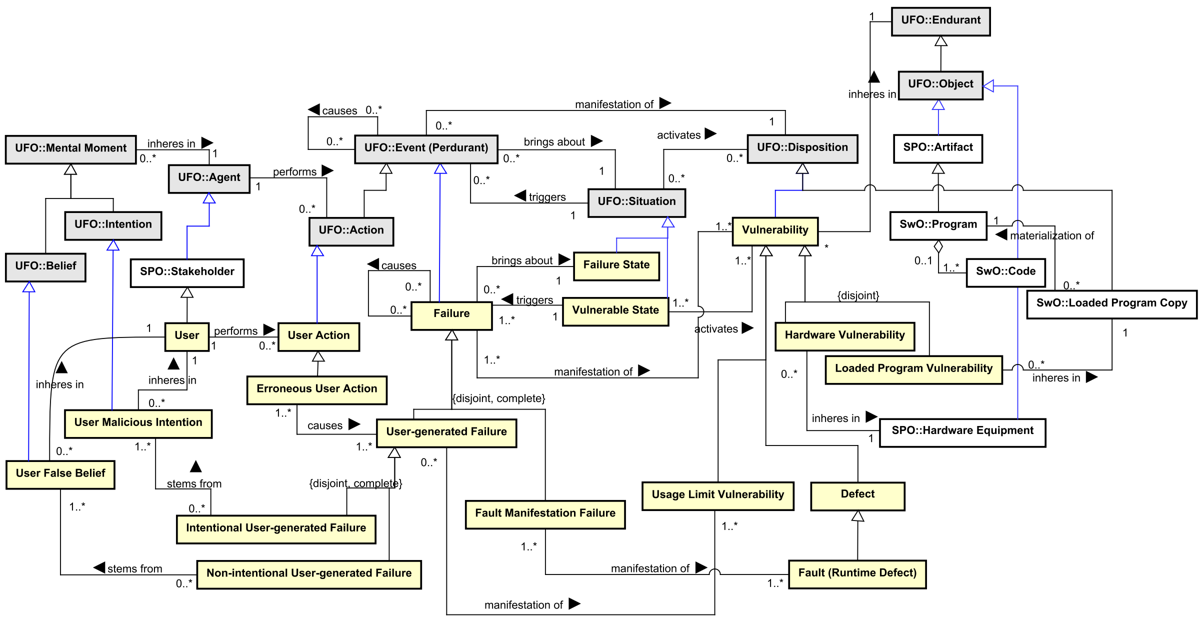


Figure 30 – Conceptual Model of the Ontology of Software Defects, Errors and Failures.

as *Apache httpd* can make all of its hosted applications undergo subsequent failures). As defined in UFO (GUIZZARDI et al., 2013), *causation* is a relation of *strict partial order* and, hence, failures cannot be their own causes or causes of their causes but failures can (perhaps, indirectly) trigger other failures in a chain of causation.

As Events, Failures are directly related to two distinct Situations. The first one is the Situation that exists prior to the occurrence of that Failure and that *triggers* the Failure. This Situation is represented in the ontology as a Vulnerable State and denotes the situation that *activates* the Disposition (i.e., a Vulnerability) that will be manifested in that Failure. For example a Software System with security issues that is exposed to the possibility of a hacker attack.

The second one is the situation that is *brought about* by the occurrence of the Failure, which is defined in the ontology as the Failure State, i.e., a situation that hurts the intentions of stakeholders.

In this context, the occurrence of the failure transforms a portion of reality to another: in its pre-situation, the software in execution has the disposition (i.e., a Vulnerability) to manifest the failure, but it has not occurred yet, since the disposition was not yet activated; in its post-situation, the (failure) event was triggered and reality was "transformed" to a situation in which the software is not executing its functions properly.

Although it is out of the scope of this ontology to provide vocabulary for the classification of post-failure situations, we note that Failure States can be: transient — when a failure happens but the software system is capable of recovering itself; continued — when after the occurrence of the failure the Failure State becomes permanent, or at least perduring until some action is taken in order to bring the software system back to a state in which it is capable to properly execute its functions. Failures can also be classified by other properties, such as severity, effect and how it is capable to affect a Software System.

Failures are further refined in two distinct sub-types: Fault Manifestation Failures and User-Generated Failures. The former are Failures that are manifestations of Faults; the latter are Failures that are directly *caused by* User Actions.

A Vulnerability[1] represents the Dispositions that can exist in software artifacts or in hardware equipment. This notion is then specialized in two distinct generalization sets. The first represents the types of Dispositions that can be activated and manifest Failures: Defects and Usage Limit Vulnerabilities. For example. not treating exceptions in code or not using tools to avoid SQL injections are very common Defects that can lead into Failures. The second one represents the types of entities in which those Dispositions inhere: a Hardware Vulnerability inheres in a Hardware Equipment, while a Loaded Program Vulnerability inheres in a Loaded Program Copy. For example, two well-known Hardware Vulnerabilities were Meltdown and Spectre, which affected Intel and ARM processors for almost an decade. These Hardware Vulnerabilities were abused by hackers and crackers for many years, without the knowledge of Intel or users, since at least four generations of processors were created with these Vulnerabilities in them.

Besides, it is also important to understand that hardware and software vulnerabilities are very different in nature. As a Hardware Equipment is essentially a physical Artifact, an existing Vulnerability can be manifested at any time. On the other side, as Programs are Abstract Artifacts,[2] a program-related Vulnerability can only be manifested if the program is loaded inside the memory of a Machine, namely, if it *inheres in* a Loaded Program Copy.

A Defect is a common type of Vulnerability that can exist in physical artifacts (e.g., Hardware Equipments), in the source code of a Program and even in the Loaded Programs Copies inhering in a Machine. It is defined by the Standard Classification for Software Anomalies (IEEE, 2009) as *an imperfection in a work product (WP) where that WP does not meet its specification and needs to be repaired or replaced*. What this and other definitions in the literature (BOURQUE; FAIRLEY et al., 2014) have in common is that Defects are understood as properties of Endurants. However, differently from intrinsic moments that are always manifest, i.e., qualities (e.g., the color of a wall), Defects, as Vulnerabilities may never be activated and, consequently, never be manifested into Failures. This means that a Vulnerability can exists inside a Loaded Program Copy for a long time, until it is activated and manifested. For example, in one of the most famous of theses

[1] The notion of vulnerability is frequently used in a way that is restricted to defects that can be exploited by attacks. We take a more general *Risk Management* view (HOGGANVIK; STØLEN, 2006; SALES et al., 2018) of vulnerabilities as Dispositions that can be manifested by events that can hurt stakeholder's goals (GUIZZARDI et al., 2013) or diminish something's value (SALES et al., 2018).

[2] The definition of Program as an Abstract Artifact is discussed in Section 3.5.2.

cases, the "Dirty Copy on Write" (ALAM et al., 2017) Vulnerability existed inside Linux Kernel for over nine years, until a researcher discovered that it could be exploited to grant root access to an attacker with malicious intentions.

Defects can exist throughout the entire software process (CHILLAREGE, 1996). As previously mentioned, some Defects can (contingently) refrain from being manifested across software executions. When a Defect is manifested as a Failure, we term that Defect a Fault (Runtime Defect). A Fault, hence, can be seen as a role played by a Defect in relation to a Failure. Furthermore, we countenance the occurrence of Failures that are directly caused by User actions. In this scenario, a User *performs* an Erroneous User Action that *causes* a User-Generated Failure. In other words, we name an Erroneous User Action a User action that *causes* such a Failure. As discussed in (WANG et al., 2016), software artifacts are designed taking into consideration *Domain Assumptions*. When a Program is created based on incorrect assumptions about the environment in which it will execute, we consider this a Program Defect. However, there are cases in which the software makes explicitly defined assumptions (disclaimers, usage guidelines), which are neglected by users in their actions. In this case, it is the Erroneous User Action itself that is the cause of the Failure.

As discussed in (FRICKER; SCHNEIDER, 2015), events (including Failures) are *polygenic* entities that can result from the interaction of multiple dispositions. For instance, we take that a User-Generated Failure can be caused by a combination of certain dispositions of a software system combined with certain Mental Moments of Agents. These mental moments include Beliefs (including User False Beliefs about domain assumptions) as well as Intentions (including User Malicious Intentions). A particular case of a User-Generated Failure, is one in which this Usage Limit Vulnerability is exploited in an intentional malicious manner, in what is termed an *attack* (e.g., a User with Malicious Intentions can make a Web server fail with a Distributed Denial of Service attack). In this case, the server that is being attacked has no Defect (and, hence, no Fault). This server just has a limited number of requests that it can answer in a period of time (a *capacity*, which is a type of disposition). If this number is exceeded for a long period, all system resources will be consumed and the server will experience an Intentional User-generated Failure. This failure can be as simple as a denial of service due to lack of resources, or as critical as a full system crash. In a different scenario, a Non-intentional User-generated Failure can stem from the User False Belief of unconsciously performing a wrong action in a software syste (e.g deleting necessary configuration files in Debian Linux).

Regarding SEON integration, OSDEF is grounded on UFO and also specializes concepts from SwO and RSRO. OSDEF specializes three concepts from SPO and three from SwO. Furthermore, since OSDEF is not focused on software requirements, it does not need to be integrated into ReqON sub-network. Figure 31 depicts OSDEF integrated

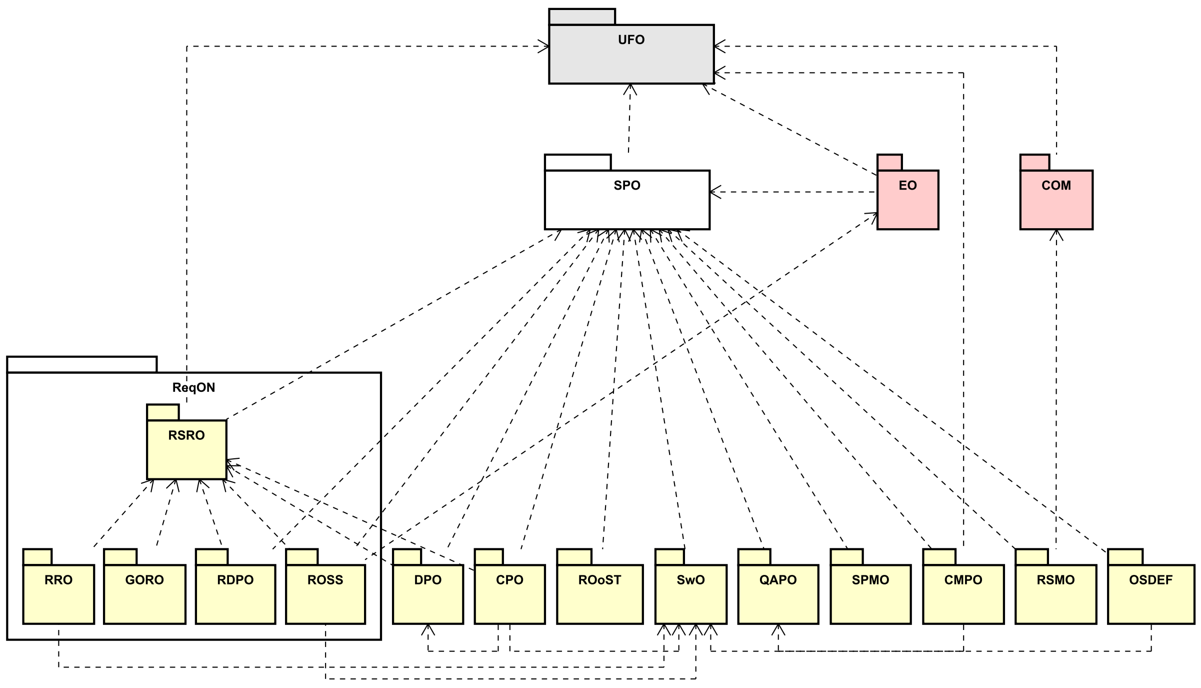


Figure 31 – Integration of OSDEF to SEON

to SEON. The color scheme adopted in Figure 31 is the same one used in Figure 17, in Section 3.5.

5.3 Related Works

This Section presents the works that are related to OSDEF. OSDEF is also related and reuses concepts from SPO and SwO. However, as those ontologies were discussed in Section 4.5, they will not be repeated here.

5.3.1 Del Frate's Ontology of Failure Engineering Artifacts

DelFrate (2012) provides an ontological analysis of the notion of failure in engineering artifacts. A theory that distinguishes between three types of failures is built: *function-based failures*, *specification-based failure* and *material-based failure*. Del Frate also discusses the relation between a Failure, as an Event that happens to an Artifact and a Fault, which is the state of the artifact after the Failure, for each of the three types of Failures that are proposed.

The ontological analysis provided by Del Frate shares with our work the interpretation of failures as events. However, honoring the terminology employed in Software Engineering standards, we conceive faults as processual roles of defects in an existing (occurred) failure. In contrast, Del Frate considers faults as states (Situations, in the sense of UFO) in a way that is similar to what we call a Failure State. Moreover, another important

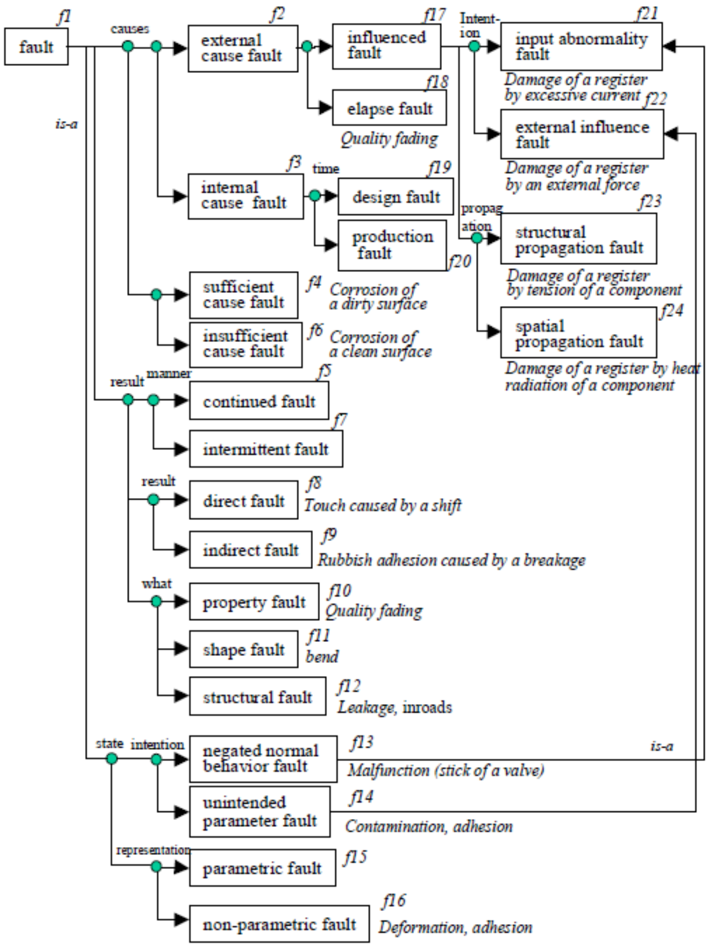


Figure 32 – Ontology of Faults (KITAMURA; MIZOGUCHI, 1999a).

difference is that we take into account other types of anomalies, such as defects and errors (even taking in consideration the direct participation of human agents in the occurrence of failures). Other distinctions worth mentioning is that our work is focused on software systems and grounded on a foundational ontology, whereas Del Frate's work is more generic (covering all engineering artifacts) and does not reuse any particular foundational ontology.

5.3.2 Kitamura and Mizoguchi's Ontology of Faults

Kitamura and Mizoguchi (1999b) propose an ontological analysis of the fault process and an ontology of faults, depicted in Figure 32, that provides a categorization of different types of Faults considering different properties for specifying the scope of a diagnostic activity. Distinct properties, relations and constraints of different types of faults are presented, e.g., faults are differentiated between: externally or internally caused; structural or property-related; or depending on their ontological nature. The ontology is intended to be used as a tool for characterization of model-based diagnostic systems and as a formal vocabulary, for human use, during the diagnostic activity.

In comparison to ours, this work has a different focus, which is centered on the fault process and on defining a glossary of faults by specifying different characteristics and constraints of Faults (events) and Failures, which are called Fault States. Besides that, although their ontology is not grounded on a foundational ontology, they recognize and discuss, at a minor level, concepts and ontological aspects that are presented in our work, such as the concepts of Event, State (as a Situation in UFO) and also *causality* and *parthood* relations.

5.3.3 Avizienis Taxonomy of Faults

Avizienis et al. (AVIZIENIS et al., 2004) proposes a taxonomy of faults, failures and errors in a context of dependability, reliability and security. In comparison with OSDEF, the taxonomy proposed there also understands Failures as Events and Faults and Vulnerabilities as properties of a system, composed of software, hardware and people. However, the concept of Error used by the taxonomy is different from the one that we used in OSDEF. Our notion of Error is the one of an Erroneous User Action, being based on the IEEE 1044 standard. This notion is similar to what is termed by Avizienis and colleagues as a Human Fault. Moreover the taxonomy presented by Avizienis et al. has a broader scope than OSDEF, presenting a larger vocabulary focused on properties such as criticality and consistency. On the other hand, OSDEF is more focused on defining the ontological nature of these concepts and the relations between then, using UFO as foundation.

5.3.4 Common Ontology of ValuE and Risk (COVER)

The Common Ontology of ValuE and Risk (COVER) (SALES et al., 2018) provides a rigorous ontological analysis of Events, Objects, Qualities, Situations and relations that can be used to characterize the notion of risk. The ontology is based on three domain-independent perspectives: (i) the experiential perspective, which represents both value and risk as events with their causes; (ii) the relational perspective, which presents the relational nature of value and risk; and (iii) the quantitative perspective, which presents value and risk in terms of mensurable qualities. COVER sees risk as intrinsically connected to the notion of value, in a way that *risk assessment* is seen as a particular case of *value ascription*. Value is related to the degree to which certain properties of the object can be enacted to satisfy one's goals. Analogously, the risk incurred to an object is roughly the degree to which its vulnerabilities together with the capacities (and possibly, intentions) of a threatening entity can be enacted to end one's goals.[3] Moreover, risk is always the risk of the destruction of value. In other words, the authors conceptualize value and risk as two sides of the same coin, thus, sharing intrinsic properties such as goal dependency

[3] Value and risk are always defined in relation to one's goals. As a consequence, they are always relative notions.

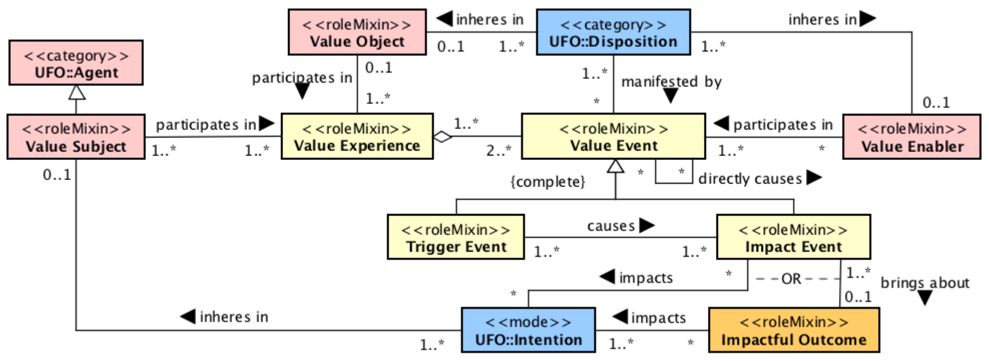

<<roleMixin>> Value Object
inheres in
<<category>> UFO::Disposition
inheres in
<<category>> UFO::Agent
participates in
manifested by
<<roleMixin>> Value Subject
participates in
<<roleMixin>> Value Experience
<<roleMixin>> Value Event
participates in
<<roleMixin>> Value Enabler
{complete}
directly causes
<<roleMixin>> Trigger Event
causes
<<roleMixin>> Impact Event
impacts
OR
brings about
inheres in
<<mode>> UFO::Intention
impacts
<<roleMixin>> Impactful Outcome

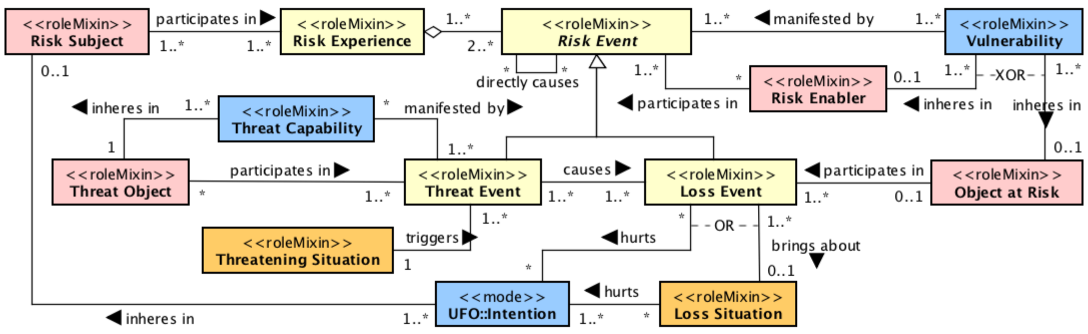
Figure 33 – Fragment of COVER presenting the concept of Value Event and their relations (SALES et al., 2018).

and relativity. Furthermore, the authors present and discuss different types of value and risk based on these intrinsic properties.

Figures 33 and 34 show two fragments of COVER that constitute its experiential perspective. Both figures are shown here exactly as in the original article, i.e., using the OntoUML language as well as a color code often used by that community. OntoUML is a UFO-based conceptual modeling language (GUIZZARDI, 2005). In this color code, light red classes represent Endurant types; blue classes represent Intrinsic Moment types; yellow classes represent Event types; finally, orange classes represent Situation types.

Value (also Risk) Events can be decomposed into "smaller" events, all of which constitute the Value (Risk) Experiences of an Agent. Value and Risk can be ascribed to Objects, which then play the roles of Value Objects and Objects at Risk, respectively. They can also be ascribed to experiences (Events) focused on the relevant qualities and dispositions of these Objects. Other Objects that are not the focus of these experiences can also participate in the Value and Risk Events as Value (Risk) Enablers. Finally, the central risk domain elements in COVER are specializations of the general categories of Events, Dispositions, Agents, Objects, and Situations organized around the same ontological pattern that is used here as a basis for OSDEF, namely, the *Events as Manifestations of Object Dispositions* pattern (GUIZZARDI et al., 2013).

In comparison to OSDEF, COVER is also grounded on UFO, it conceptualizes over the same ontological pattern of Events, Dispositions and Situations presented by UFO-B. In this context, the concepts of Failure and Failure State presented in OSDEF are, respectively, sub-types of of Loss Event and Loss Situation, defined in COVER. Besides that, both ontologies define Vulnerability as Dispositions that can be manifested in Artifacts. In a direct comparison, COVER is a more generalist ontology, one can understand COVER as a core ontology that conceptualizes about value and risk. OSDEF is domain ontology that conceptualizes about failures, defects and errors in the context of software systems.



<<roleMixin>> Risk Subject
<<roleMixin>> Risk Experience
<<roleMixin>> Risk Event
<<roleMixin>> Vulnerability
<<roleMixin>> Risk Enabler
<<roleMixin>> Threat Capability
<<roleMixin>> Threat Object
<<roleMixin>> Threat Event
<<roleMixin>> Loss Event
<<roleMixin>> Object at Risk
<<roleMixin>> Threatening Situation
<<mode>> UFO::Intention
<<roleMixin>> Loss Situation
participates in
manifested by
directly causes
inheres in
causes
triggers
hurts
brings about
OR
XOR

Figure 34 – Fragment of COVER presenting the concept of Risk Event and their relations (SALES et al., 2018).

5.4 Chapter Summary

This chapter presented the second contribution of the thesis, the Ontology of Software Defects, Errors and Failures, OSDEF. Like ROSS, OSDEF was grounded on UFO and developed using SABiO, following an ontological analysis process that was based over the ontological pattern (FALBO et al., 2016) of Events, presented in UFO-B (GUIZZARDI et al., 2013).

However, differently from ROSS, it was created to represent a very specific part of the Software System domain, which is the understanding of the ontological nature of the problems that affect Software Systems and that are erroneously condensed into the term *anomaly*. Differently from works like the *Orthogonal Defect Classification* (CHILLAREGE, 1996), OSDEF was not created to define a vocabulary about types of Failures and Faults, but to represent the ontological nature of each of these concepts, and how they are related to each other and to elements of the software systems domain.

In the requirements traceability context, OSDEF was not created to be used on its own, but to be reused with other ontologies, such as ROSS; ROoST, the Reference Ontology of Software Testing (SOUZA; FALBO; VIJAYKUMAR, 2013; SOUZA; FALBO; VIJAYKUMAR, 2017); and CMPO (RUY et al., 2016). OSDEF represents concepts like Failures, Defects and Vulnerabilities that: (i) are directly related to a software system operation cycle, (ii) are not represented or discussed in any other ontology that are part of SEON and (iii) that provide a wide perspective for the requirements traceability domain when reused together with the concepts of the previously mentioned domain ontologies.[4]

Finally, we ended the chapter by presenting and briefly discussing proposals from the scientific literature that are directly related to OSDEF.

[4] Concepts of OSDEF are reused with concepts of ROSS and CMPO in the queries presented in Chapter 7.

6 Ontology Evaluation and Implementation

This chapter presents the implementation and evaluation processes of ROSS and OSDEF. SABiO prescribes that ontologies need to go through ontology verification and validation techniques, in a process analogous to the verification and validation of software systems.

Based on this normative, in the next sections we discuss the verification and validation techniques applied for ROSS and OSDEF. Section 6.1 presents the ontology verification technique, based on competency questions. Section 6.2 presents the validation of OSDEF and ROSS, based on ontology instantiation. Section 6.3 discusses the design and implementation process for both ROSS and OSDEF. Section 6.4 summarizes the chapter.

6.1 Ontology Verification

For ontology verification, SABiO states the primary objective is to ensure that the ontology is being built correctly, in the sense that it has no major consistency and coherence problems, and that the output artifacts meet the previously defined specifications. To achieve that, ontology verification should be *Competency Question-driven*, as such questions are used as the requirements of the ontology. More precisely, the method suggests the creation of a table that shows that the ontology elements are able to answer all raised competency questions (CQs).

Table 1 presents ROSS' verification regarding its competency questions. Once more, since ROSS is able to adequately respond to all proposed CQs, the verification is considered successful.

Table 1 – ROSS verification table based on its CQs.

CQ	**Concepts and *Relations***
CQ1	Software System is a *subtype of* Artifact.
CQ2	Software System is *composed by* many System Components, which is also *subtype of* Artifact). A Software System can be developed as with SubSystems or as an simple system (no Subsystem).
CQ3	Business Requirements, Stakeholder Requirements, System Requirements and Program Requirements, are *subtypes of* Goal.
CQ4	Stakeholder Requirements are *derived from* Business Requirements. System Requirements are derived from Stakeholder Requirements.

	Program Requirements are derived from Stakeholder Requirements and from System Requirements.
CQ5	Business Requirements are *described* in a Business Requirements Specification, which is a *subtype of* Information Item. Stakeholder Requirements are *described* in a Stakeholder Requirements Specification, which is a *subtype of* Information Item System Requirements are *described* in a System Requirements Specification, which is a *subtype of* Information Item. Program Requirements are *described* in a Program Requirements Specification, which is a *subtype of* Information Item.
CQ6	World Assumptions and Machine Assumptions are *subtypes of* Dispositions that are part of the Software System domain.
CQ7	Business Requirements Specification *describes* a set of Business Requirements based on World Assumptions, which are Propositions about the World Behavior. System Requirements Specification *describes* a set of System Requirements based on World Assumptions and Machine Assumptions, which are, respectively, Propositions about the World and the Machine Behaviors. Program Requirements Specification *describes* a set of Program Requirements based on Machine Assumptions, which are Propositions about the Machine Behavior.
CQ8	Business Rules and External Regulations are *subtypes of* Business Constraints, which are *recognized by* the Organization. Business Constraints *constrains* Business Requirements.

Analogously, Table 2 illustrates the results of the OSDEF verification regarding the predefined CQs. Moreover, the table can also be used as a traceability tool, supporting ontology change management. The table shows that the ontology can answer all CQs appropriately.

Table 2 – OSDEF verification table based on its CQs.

CQ	**Concepts and Relations**
CQ1	Failure is a *subtype of* Event that *brings about* a Failure State. A User-generated Failure is a *subtype of* Failure is *caused by* an Erroneous User Action *stemming from* a User False Belief or a User Malicious Intention. A Fault Manifestation Failure is a *subtype of* Failure that is *manifestation of* a Fault (a Runtime Defect).
CQ2	Defect is a *subtype of* Vulnerability Defect *inheres in* an Endurant

CQ3	Fault is a *subtype of* Defect which is manifested at runtime via a Fault Manifestation Failure.
CQ4	Erroneous User Action is a *subtype of* User Action (Action) that is *performed by* a User, which is a *subtype of* Stakeholder (Agent).
CQ5	Usage Limit Vulnerability is a *subtype of* Vulnerability that *inheres in* an Endurant. Program Usage Limit Vulnerability) *inheres in* a Loaded Program Copy. Hardware Usage Limit Vulnerability *inheres in* a Hardware Equipment.
CQ6	Vulnerable State is a *subtype of* Situation that *activates* a Fault and *triggers* a Failure.
CQ7	Failure State is a *subtype of* Situation that is *brought by* a Failure.
CQ8	A Failure can be *caused by* another Failure, in a chain of Events. A Vulnerable State can activate a Fault that is manifested into a Fault Manifestation Failure. An Erroneous User Action can *cause* a User-generated Failure, which is a *manifestation of* a Usage Limit Vulnerability.

6.2 Ontology Validation

For ontology validation, SABiO states that its primary objective is to ensure that the right ontology is being built. In other words, the ontology must fulfill its intended purpose. The method indicates that a good and relatively simple validation technique is to check whether the created reference ontology may be instantiated to represent real-world situations that are directly related to the domain of the ontology.

In this context, we conducted a particular type of validation, of ROSS and OSDEF in terms of their capacities to support the analysis of software risks associated with systems' anomalies. So, in order to do that, we reused COVER, the Common Ontology of Value and RISK (SALES et al., 2018) (cf. Section 5.3.4) and employed it in combination with ROSS and OSDEF to instantiate real-world scenarios of famous cases of software failures. Our objective is showing that the combination OSDEF, ROSS and COVER is capable of representing these real-world situations in the best possible way, as we believe that the risk analysis perspective given by COVER complements the representation provided by the combination (reuse) of ROSS and OSDEF. Furthermore, we choose these specific cases because they are well-known and well-documented cases of failures of software systems that caused major damage, in our society. Besides, these cases are also good candidates because they are not based on software-only systems. More precisely, the described human actions, hardware-based defects and value-risk situations exemplify scenarios that are appropriate for the validation of our proposed domain ontologies.

In what follows, we describe each case and present an *instantiation model* for each of them, based on the UML object diagram. These instance-level models have been used as an evaluation technique associated with OntoUML models (e.g., in (SALES et al., 2018; AMARAL et al., 2019)). The color scheme used here is the same as the aforementioned OntoUML color convention.[1]

Moreover, we here abuse the object diagram notation in the following manner: boxes (object names) represent instances of the domain elements; arrows represent links; base class notation represents the concepts from OSDEF, COVER and ROSS instantiated by each of these elements. This convention is used in figures 35, 36 and 37.

Case 1 (Figure 35[2]): the Therac-25 disaster (LEVESON; TURNER, 1993). Therac-25 was a medical equipment that handled two types of therapy: a low-powered direct electron beam and a megavolt X-ray mode. The core of the incident was that the Software System that was responsible for controlling the equipment was reused from a previous model of the Diagnose Equipment (Hardware Equipment), in such way that it was missing important upgrades to the existing routines (parts of Programs that constituted the system) and adequate testing, conditions that can be understood as Vulnerabilities that inhered in those Programs. The propensity of the system to cause race conditions (Fault), was manifested into a critical Failure when an operator (Risk Enabler), unconsciously, changed the therapy mode of the equipment too quickly, causing, instructions for both treatments to be simultaneously sent to the diagnose equipment. The first instruction to arrive would set the mode for the treatment to be applied (a kind of fault known as *race condition*). The consequences were devastating, as patients (Objects at Risk) expecting to receive an electron-beam, could ended up receiving the X-ray and because of that, ended up getting sick or even dying from radiation poisoning. This was an example of a Fault Manifestation Failure happening as the manifestation of Fault that caused patients to be exposed to high doses of radiation (Loss Event). Besides, although the Fault Manifestation Failure was brought about by a User Action, as the operator quickly changed the mode of the equipment (this action created a Threatening Situation), it cannot be considered an Erroneous User Action, since this cannot be considered a user's negligence of stated assumptions. In other words, the operator, as an User of the Therac-25 Software System, even if unknowingly, participated as a Risk Enabler for the Failure of the system and being responsible for creating the Mega-volt X-Ray Activation (a Threat Event for the patient), which in its turn, caused the Loss Event and brought about Loss Situations where patients ended up dying.

Case 2 (Figure 36): in 2013, Spamhaus, a nonprofit professional protection

[1] Orange is used to represent situations; yellow - events; blue - intrinsic moments; light red - objects.

[2] Please see (FONSECA et al., 2019) for the semantics of *historical dependence*. Moreover, following the goal-oriented requirements engineering tradition, we use the relation of *break* here as an extreme case of the *hurt* relation as in (SALES et al., 2018).

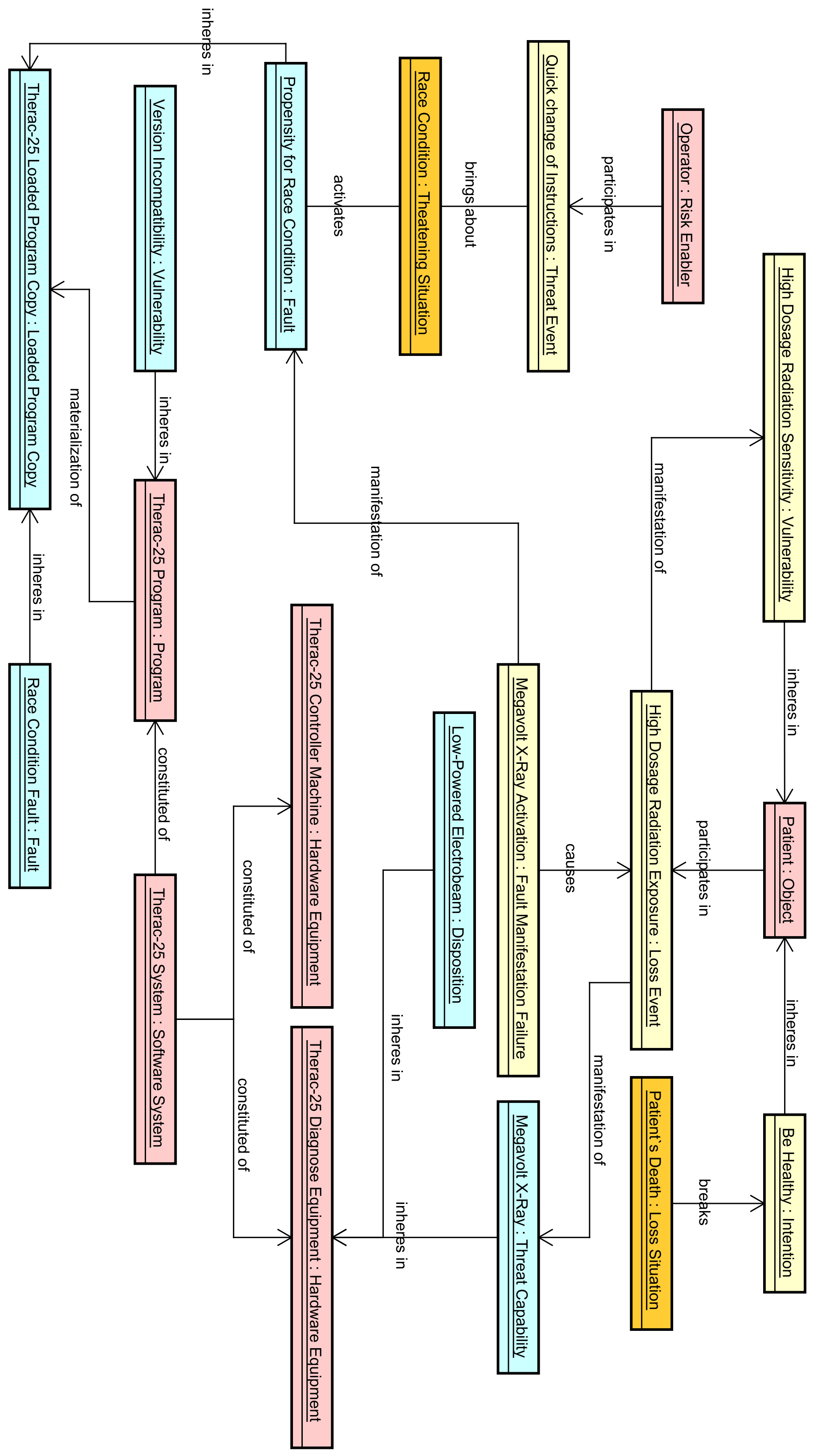

Operator : Risk Enabler
participates in
Quick change of Instructions : Threat Event
brings about
Race Condition : Theatening Situation
activates
Propensity for Race Condition : Fault
inheres in
Therac-25 Loaded Program Copy : Loaded Program Copy
Version Incompatibility : Vulnerability
inheres in
materialization of
Therac-25 Program : Program
inheres in
Race Condition Fault : Fault
constituted of
Therac-25 System : Software System
constituted of
constituted of
Therac-25 Controller Machine : Hardware Equipment
Therac-25 Diagnose Equipment : Hardware Equipment
manifestation of
Low-Powered Electrobeam : Disposition
inheres in
Megavolt X-Ray Activation : Fault Manifestation Failure
causes
inheres in
Megavolt X-Ray : Threat Capability
manifestation of
High Dosage Radiation Exposure : Loss Event
manifestation of
Patient's Death : Loss Situation
breaks
High Dosage Radiation Sensitivity : Vulnerability
inheres in
participates in
Patient : Object
inheres in
Be Healthy : Intention

Figure 35 – The Therac-25 System instantiation with OSDEF, COVER and ROSS.

service on the Web (a Web-based Software System), was the target of what might have been the largest DDoS attack (Loss Event) in history. Hackers redirected hundreds of controlled DNS servers (Threat Event), to send up to 300 gigabits of flood data to each server (Hardware Equipment) of the Spamhaus Network, with the Intention to suspend the service provided. In this case, the occurrence of the User-generated Failure is directly related with deliberate Actions of a group of hackers, acting as Risk Subjects, with User malicious intentions, to cause a Loss Event and bring about a Loss Situation where the service becomes unavailable.

For this case, there was no particular Defect, nor any Fault was activated that could end up be manifested into a Failure in the system. As an Artifact, the Spamhaus Software System had a Usage Limit concerning the number of service requests to which it could respond. When this limit was far surpassed by hundreds of hacker-controlled DNS servers, the Spamhaus Service Loaded Program Copy was compromised, because of a natural Usage Limit Vulnerability that inheres in the servers of the network. Consequently, users of the system (the Value Subjects for the owners of the Spamhaus project), had their Intention to continue to use the system, also compromised.

Case 3 (Figure 37): In 1991, during the Gulf War, the Patriot missile-system (DEFENSE, 1992) failed to protect US Army Barracks from an incoming Scud missile, resulting in the death of 28 soldiers, which were the Value Subjects for the system. The heart of the patriot defensive system was the computer that controlled the radar, responsible for detecting incoming threats. This computer was based on a 1970s design, with a limited capability to perform high-precision calculations, as it was based on a 24-bit architecture. This outdated architecture ended up being a Vulnerability for the Patriot system. The system worked based on communications between a radar, a computer, the missile turret component and the Program that was responsible to calculate the trajectory of incoming Threat Objects (usually SCUD Missiles). After the radar detected the incoming projectile, with electric pulses, the loaded missile surveillance program (Loaded Program Copy) was responsible for calculating the next area where the incoming object might be, in order to track down its trajectory (Software Function) and trigger the launch of a patriot missile to intercept the incoming Threat Object, before it hits base camp. To do that, the computer measured time, with the precision of tenths of a second, in an integer that could be 24-bits long.

The system lost precision over time, as the calculations were not precise enough due to the outdated architecture (Israel army reported that the system was operating with considerable deviation in the calculations, after only 8 hours of run-time). The specific Patriot unit of the incident was online for over 100 hours (Loaded Program Copy), contributing to the loss of precision in the calculations, a Defect that propagated and escalated over time. At the end, the system was looking for the incoming Scud (Threat

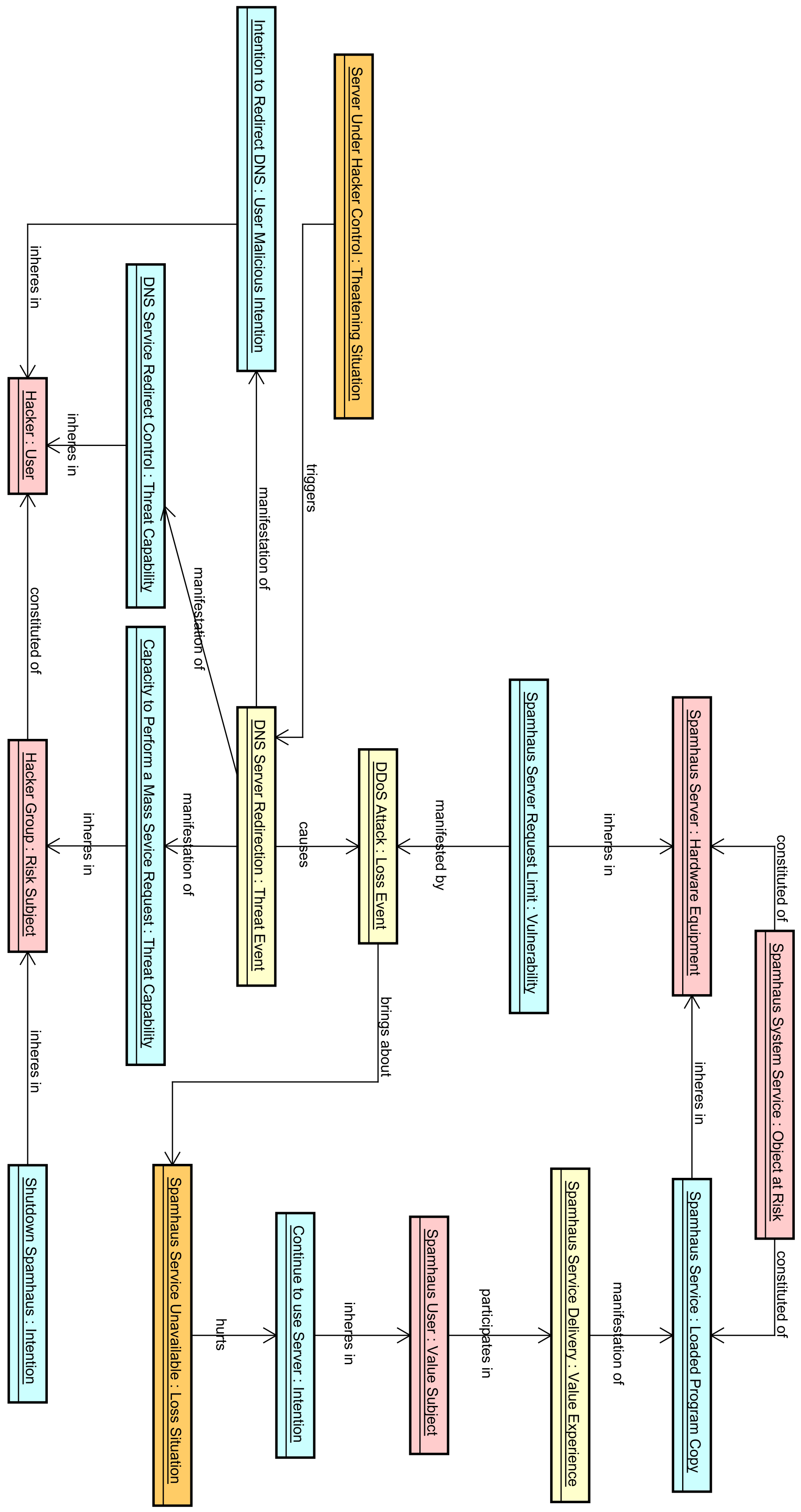

Intention to Redirect DNS : User Malicious Intention
Server Under Hacker Control : Theatening Situation
inheres in
DNS Service Redirect Control : Threat Capability
Hacker : User
inheres in
triggers
manifestation of
manifestation of
constituted of
Capacity to Perform a Mass Sevice Request : Threat Capability
DNS Server Redirection : Threat Event
DDoS Attack : Loss Event
Spamhaus Server Request Limit : Vulnerability
Spamhaus Server : Hardware Equipment
Hacker Group : Risk Subject
inheres in
manifestation of
causes
manifested by
inheres in
constituted of
Spamhaus System Service : Object at Risk
inheres in
brings about
inheres in
Shutdown Spamhaus : Intention
Spamhaus Service Unavailable : Loss Situation
hurts
Continue to use Server : Intention
inheres in
Spamhaus User : Value Subject
participates in
Spamhaus Service Delivery : Value Experience
manifestation of
Spamhaus Service : Loaded Program Copy
constituted of

Figure 36 – The Spamhaus System case instantiation with of OSDEF, COVER and ROSS.

Object) meters away from its precise location and, hence, never activated the defensive patriot missile. As in the Therac-25 incident, the Object at Risk is not the software system by itself, but human lives, as the Patriot System was critical to support the lives of the soldiers in the battlefield, which had the Intention to remain protected while in base camp. Such Intention was broken as a SCUD missile goes undetected by the radar (Threat Event) and ended up hitting the barracks, bringing about a Loss Situation where 28 soldiers lives were lost.

Besides, for this particular case, the Defect was not manifested in a split of a second, resulting in a Failure as soon as a *defective* part of the system was accessed during program execution. Instead, the Defect occurs because after some hours at run-time, the system calculations were not correct anymore. Consequently, the Fault manifests in the system. In other words, the software that controlled the Patriot Defense System entered in a Threatening Situation of a high accumulation of calculation errors a few hours after being online. However, this situation is not easily perceived, as in an ordinary Web-based system. At this point, the system can suffer a critical failure at any time, as it is no longer capable of fulfilling its most important requirement: protecting the soldiers in the camp from attacks.

6.3 Ontology Design and Implementation

Ontology Design and Implementation are respectively the third and fourth phases of SABiO ontology development process. SABiO defines that in order to create operational ontologies, it is necessary to design and implement the reference ontology in a machine-readable language (e.g OWL).

Regarding Ontology Design, the operational versions of ROSS and OSDEF are based on gUFO (ALMEIDA et al., 2019), the lightweight/gentle version of UFO created with the specific purpose of providing support for the development of operational ontologies.[3] This decision was taken because we were aiming to produce a lightweight version of ROSS and OSDEF without losing all the properties that were present on the reference ontologies and gUFO was originally designed to tackle this aspect.

Besides, both ROSS and OSDEF were implemented with Protégé (NOY et al., 2001; NOY et al., 2003) and linked data technologies: RDF, OWL and SPARQL. These design decisions were based on the fact that the graph structure of RDF (subject, predicate and object) is a very good option to represent the requirements traceability data, which is composed by many instances of concepts of ROSS and OSDEF and the relations that exists between them. The concepts of the ontologies are the nodes and the relations are the edges

[3] gUFO was presented in Section 3.3.2.

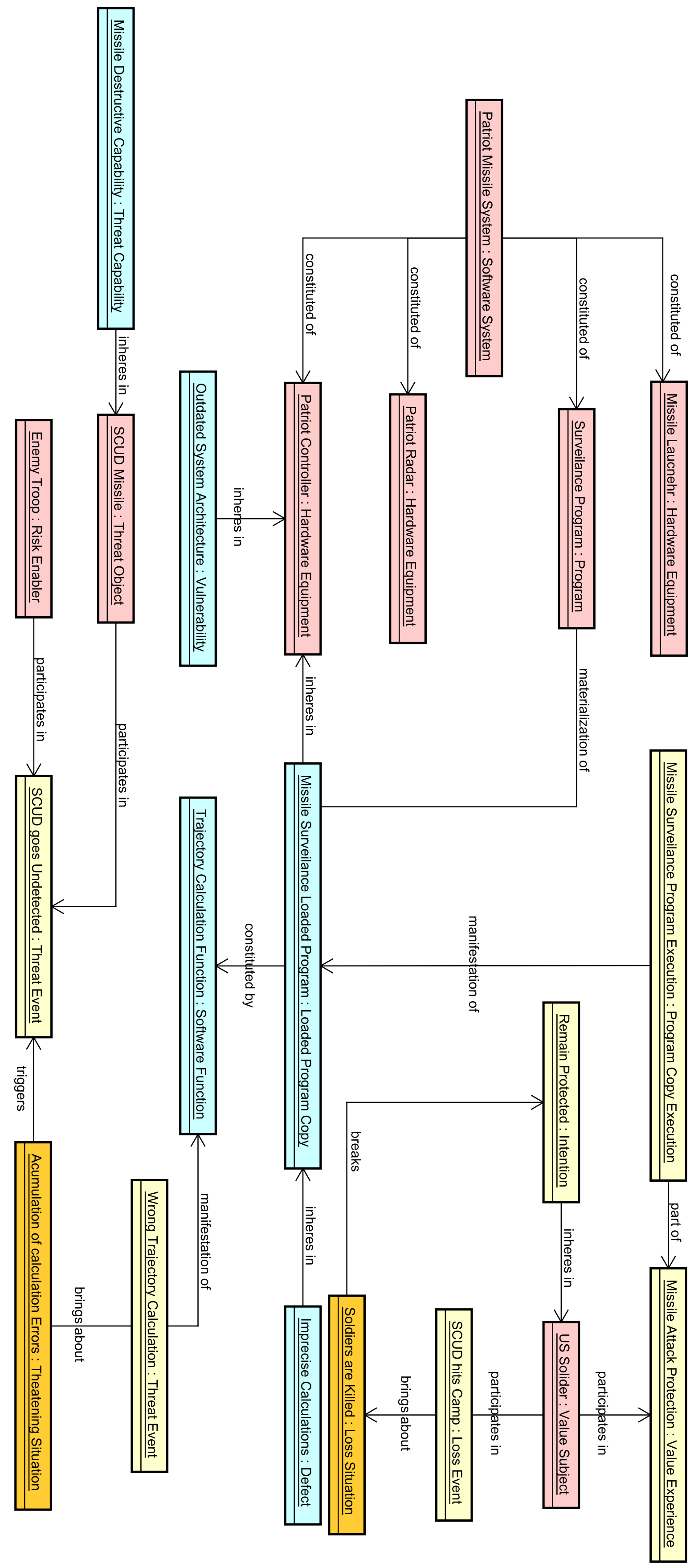
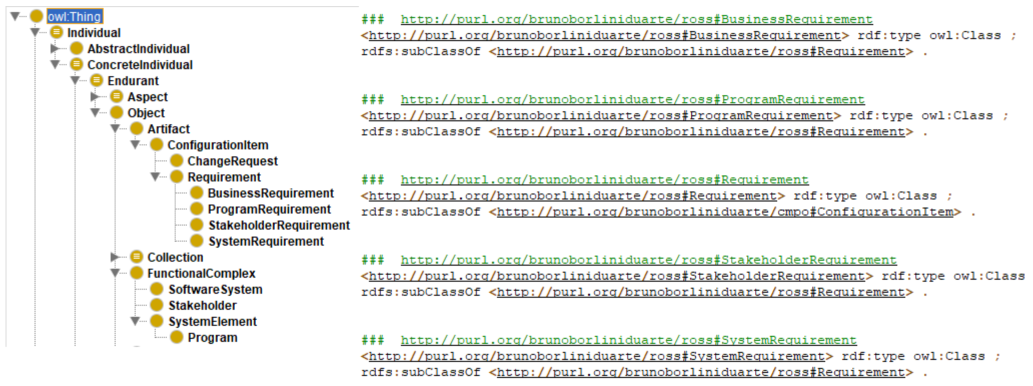

Missile Destructive Capability : Threat Capability
inheres in
SCUD Missile : Threat Object
Enemy Troop : Risk Enabler
participates in
participates in
SCUD goes Undetected : Threat Event
triggers
Acumulation of calculation Errors : Theatening Situation
brings about
Wrong Trajectory Calculation : Threat Event
manifestation of
Trajectory Calculation Function : Software Function
constituted by
Outdated System Architecture : Vulnerability
inheres in
Patriot Missile System : Software System
constituted of
constituted of
constituted of
constituted of
Patriot Controller : Hardware Equipment
Patriot Radar : Hardware Equipment
Surveilance Program : Program
Missile Laucnehr : Hardware Equipment
inheres in
materialization of
Missile Surveilance Loaded Program : Loaded Program Copy
manifestation of
Missile Surveilance Program Execution : Program Copy Execution
part of
Missile Attack Protection : Value Experience
Remain Protected : Intention
breaks
inheres in
inheres in
Imprecise Calculations : Defect
Soldiers are Killed : Loss Situation
brings about
SCUD hits Camp : Loss Event
participates in
US Solider : Value Subject
participates in

Figure 37 – The Patriot Missile System case instantiation with of OSDEF, COVER and ROSS.



```
###  http://purl.org/brunoborliniduarte/ross#BusinessRequirement
<http://purl.org/brunoborliniduarte/ross#BusinessRequirement> rdf:type owl:Class ;
rdfs:subClassOf <http://purl.org/brunoborliniduarte/ross#Requirement> .

###  http://purl.org/brunoborliniduarte/ross#ProgramRequirement
<http://purl.org/brunoborliniduarte/ross#ProgramRequirement> rdf:type owl:Class ;
rdfs:subClassOf <http://purl.org/brunoborliniduarte/ross#Requirement> .

###  http://purl.org/brunoborliniduarte/ross#Requirement
<http://purl.org/brunoborliniduarte/ross#Requirement> rdf:type owl:Class ;
rdfs:subClassOf <http://purl.org/brunoborliniduarte/cmpo#ConfigurationItem> .

###  http://purl.org/brunoborliniduarte/ross#StakeholderRequirement
<http://purl.org/brunoborliniduarte/ross#StakeholderRequirement> rdf:type owl:Class
rdfs:subClassOf <http://purl.org/brunoborliniduarte/ross#Requirement> .

###  http://purl.org/brunoborliniduarte/ross#SystemRequirement
<http://purl.org/brunoborliniduarte/ross#SystemRequirement> rdf:type owl:Class ;
rdfs:subClassOf <http://purl.org/brunoborliniduarte/ross#Requirement> .
```

Figure 38 – Concepts of ROSS as sub-types of concepts of gUFO being represented in Protégé (left) and in Turtle syntax (right)

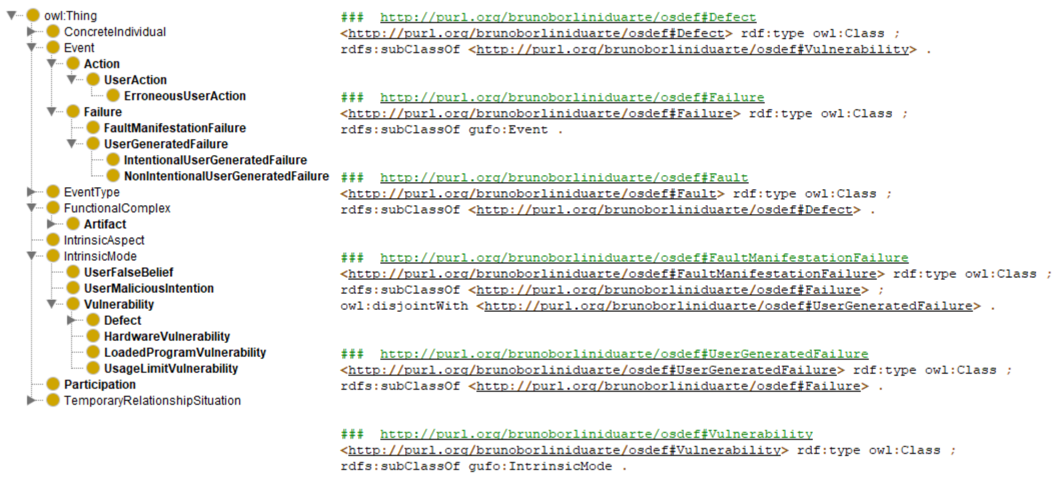
in the graph structure. Moreover, RDF format is necessary to query data with SPARQL.[4] Besides that, the concepts defined in OWL and RDF could be used to semantic annotate resources in a configuration management context. Finally, Protégé is a free, open-source ontology editor that is supported by a strong and active academic community, with full support for the technologies mentioned above.

The development process was thus conducted using the Protégé tool. Basically, the file that contains the implementation of gUFO, written in Turtle syntax (W3C, 2014b), is imported into Protégé and the concepts of ROSS and OSDEF are created as sub-classes of the concepts of gUFO. This process is, to a certain extent, analogous to the one of reusing UFO as a foundational ontology, when developing a domain reference ontology. However, it is important to understand that the artifact (the operational ontology) that is being created is not a perfect representation of the reference ontology. Differently from reference ontologies, operational ontologies are not focused on representation adequacy, but are designed with the focus on guaranteeing desirable computational properties (FALBO, 2014). Both operational ontologies can be found in <purl.org/brunoborliniduarte>.

Figure 38 depicts the concepts of Program Requirement, System Requirement, Stakeholder Requirement and Business Requirement, all sub-types of Requirement in Protégé (left) and in the original Turtle syntax (right). Moreover, it is important to explain that the requirements being represented in the operational ontologies are, in fact, Requirements Artifacts, specifications of the Goals that are depicted in the reference ontologies. This design decision is based on the fact that, in the context of a software process, Requirements (as UFO::Goals) are described into Requirements Artifacts (specifications) which are implemented into Programs. These Requirements Artifacts are traceable to Programs and to other Artifacts, during the Requirements Traceability process.

Analogously the implementation of ROSS, Figure 39 depicts part of the concepts of OSDEF in the Protégé graphical tool (left) and in the original Turtle syntax (right).

[4] SPARQL is used as a tool to support Requirements Traceability in Chapter 7.



```
owl:Thing
    ConcreteIndividual
    Event
        Action
            UserAction
                ErroneousUserAction
        Failure
            FaultManifestationFailure
            UserGeneratedFailure
                IntentionalUserGeneratedFailure
                NonIntentionalUserGeneratedFailure
    EventType
    FunctionalComplex
        Artifact
    IntrinsicAspect
    IntrinsicMode
        UserFalseBelief
        UserMaliciousIntention
        Vulnerability
            Defect
            HardwareVulnerability
            LoadedProgramVulnerability
            UsageLimitVulnerability
    Participation
    TemporaryRelationshipSituation
```

```
###  http://purl.org/brunoborliniduarte/osdef#Defect
<http://purl.org/brunoborliniduarte/osdef#Defect> rdf:type owl:Class ;
rdfs:subClassOf <http://purl.org/brunoborliniduarte/osdef#Vulnerability> .


###  http://purl.org/brunoborliniduarte/osdef#Failure
<http://purl.org/brunoborliniduarte/osdef#Failure> rdf:type owl:Class ;
rdfs:subClassOf gufo:Event .


###  http://purl.org/brunoborliniduarte/osdef#Fault
<http://purl.org/brunoborliniduarte/osdef#Fault> rdf:type owl:Class ;
rdfs:subClassOf <http://purl.org/brunoborliniduarte/osdef#Defect> .


###  http://purl.org/brunoborliniduarte/osdef#FaultManifestationFailure
<http://purl.org/brunoborliniduarte/osdef#FaultManifestationFailure> rdf:type owl:Class ;
rdfs:subClassOf <http://purl.org/brunoborliniduarte/osdef#Failure> ;
owl:disjointWith <http://purl.org/brunoborliniduarte/osdef#UserGeneratedFailure> .


###  http://purl.org/brunoborliniduarte/osdef#UserGeneratedFailure
<http://purl.org/brunoborliniduarte/osdef#UserGeneratedFailure> rdf:type owl:Class ;
rdfs:subClassOf <http://purl.org/brunoborliniduarte/osdef#Failure> .


###  http://purl.org/brunoborliniduarte/osdef#Vulnerability
<http://purl.org/brunoborliniduarte/osdef#Vulnerability> rdf:type owl:Class ;
rdfs:subClassOf gufo:IntrinsicMode .
```

Figure 39 – Concepts of OSDEF as sub-types of concepts of gUFO being represented in Protégé (left) and in Turtle syntax (right).

Moreover, although it is not shown in Figures 38 and 39, we also imported and reused the object and data properties that are defined in gUFO.

6.4 Chapter Summary

This chapter presented the evaluation and implementation processes performed for ROSS and OSDEF. Ontology implementation process was based on gUFO, the "gentle" version of UFO which was created for the development of operational ontologies. Moreover, SABiO, the ontology engineering method adopted for the development of ROSS and OSDEF states that the ontology evaluation process should be performed based on verification and validation techniques.

For ontology verification, we answered the competency questions that were raised as the ontology requirements. This activity aims to demonstrate that the ontologies are build correctly, since they need to be capable to answer the competency questions that were raised for them to answer.

For ontology validation, we used real world-scenarios of well-known cases of catastrophic failures in software systems to demonstrate that the ontology developed is capable to represent such situations.

7 Ontology-based Requirements Traceability

In chapters 4 and 5, we proposed two domain reference ontologies about software systems and about the types of failures and defects that can happen during the software system operation. In this chapter, we present our approach for the use of operational ontologies and SPARQL queries (cf. Section 3.3) as tools for recovering traceability links and reasoning over the requirements data with the ontologies proposed in this thesis.

Section 7.1 starts the discussion of our approach for Ontology-based Requirements Traceability. Section 7.2 presents the data of the ATM System that will be used to demonstrate our approach. Section 7.3 presents our approach using the data previously mentioned and discusses the implications, possibilities and limitations of it. Section 7.4 presents a prototype tool created to provide graphical visualization of the SPARQL queries performed in our approach. Section 7.5 discusses proposals for requirements traceability that are related to this thesis. Finally, Section 7.6 summarizes this chapter.

7.1 From Ontologies to Traceability Reference Models

The current literature has many works that intend to create real-semantic or fully-automated requirements traceability approaches, as it is considered to be a very important but, at the same time, very difficult activity to be performed in a continuous way.[1] Many of these works are based on very complex formal approaches, that would require a considerable effort and resources to be properly executed. Due to that, they become almost prohibitive for low-end traceability users (RAMESH; JARKE, 2001). Besides, some other proposals are dependent on prototype tools that may never be fully developed or are just not available anymore.

Due to that, we do not advocate for or intend to propose an approach for automatic or “effortless” requirements traceability. Instead, we believe that requirements traceability and the requirements management process, as the name implies, are management processes, in the sense that they probably will not be conducted in a fully automatic mode. Moreover, our proposal is based on knowledge and tools that can be easily found in the scientific literature, such as domain reference ontologies and linked data technologies.[2]

Furthermore, differently from some proposals in the literature, our reference models (ontologies) are not focused on the definition of the traceability process. More precisely, we are not focused on defining concepts like Trace or Trace Artifact and associating them with

[1] Requirements Traceability benefits and major problems were discussed in Section 2.2.
[2] Ontologies and linked data technologies used in this work were presented in Chapter 2.

elements of the software process. Our proposal is focused on using reference ontologies for the software domain and implementing them to support ontology-based requirements traceability. In other words, our proposal is focused on defining conceptual models about the software system domain and implementing them to support semantic-based requirements traceability. Due to that, the concept of Trace is implicit in the relations between concepts that are part of the reference models. In other words, every relation that exists between Requirements and other Artifacts that is represented in the reference models is a possible trace that can be retrieved.

The benefit of this approach is that the semantics of the relationships is maintained. For example, the Semantic Traceability model, discussed in Section 7.5.3, defines the concept of Trace as a relation between a TraceObject, an Operation and a Client. In this case, the concept of trace becomes generic, as a *relevant relation between a ClientAgent C,an Operation O and a TraceObject T*. We argue that, in this case, the knowledge that may exist between the specific relationships between each specialization of TraceObject, Operation and Client is lost. The alternative is to explicitly define each relationship, between each type of concept in the reference model (as done in this work). However, this alternative raises the complexity of the model. In order to avoid the increase of complexity, some strategies can be executed, for example, divide the reference model in modules (or sub-ontologies, as in our case), that can be used according to the user needs. Another possible strategy is to create multiple models, in such a way that all models will share the same basic knowledge, but each one will have its own intrinsic complexity. This approach is used by Ramesh and Jarke, in their Semantic Traceability Model, discussed in Section 7.5.1.

Furthermore, on other types of proposals, such as the Traceability Meta-Model proposed by Espinoza and Garbajosa (2011), a simpler model is defined and the user is responsible to extend the model by specializing its concepts. For example, the authors define the concept of Traceability link and instruct the user to specialize this concept with the sub-types of Traceability links that are relevant for their use case. We did not chose this modeling approach for our reference model because it places a big responsibility in the users hands, which is being responsible for completing modeling activity for their own reference model, a task that can be complex if the users have no experience in conceptual modeling. Instead, we suggest to the users of our reference model to reuse the concepts of the ontologies that are part SEON, since both ROSS and OSDEF were designed with this type of reuse in mind. In other words, our user is instructed to reuse the ontologies in SEON to represent concepts that are not discussed in ROSS and OSDEF. We believe that this was the best case for our approach since all ontologies are grounded on UFO and based on SPO and, because of that, the ontology reuse process should be facilitated. The difference of this two approaches lies in the fact that, for our reference model, the final users do not need to finish the modeling activity of their reference model, by specializing concepts of the base model and defining the existing relations between them. Instead, the user only needs to connect the parts of the software systems domain (represented by the ontologies in SEON plus ROSS and OSDEF) that are already fully modeled, according to their needs.

Analogously to ROSS and OSDEF, we also define CQs for the queries, presented here. This CQs are intended to be answered by the SPARQL queries presented below and to depict some of the traceability information that can be retrieved by reusing the networked ontologies of SEON (SwO, RSRO, ROSS, OSDEF, CMPO and ROoST), together.

- **CQ1:** How are Business Requirements traced to Programs?
- **CQ2:** How are Change Requests related to Programs and to the Requirements that they implement?
- **CQ3:** How are Defects related to Requirements?
- **CQ4:** How are Stakeholder Requirements refined into Program Requirements and implemented into Programs?
- **CQ5:** How are Test Cases related to Requirements?

In this context, Figure 40 depicts a graphical representation of the ontologies used in the SPARQL queries reported in this chapter. SPO is represented in the center and in a different color because it is a core ontology. The numbers inside each circle correspond to the number of concepts of the ontology. The lines connecting ontologies represent the number of concepts that the two connected ontologies share. It is important to emphasize that this figure only represents the ontologies used for this particular example of the ATM Machine. In another situation, other networked ontologies from SEON, such as GORO or CPO also could be used.

Moreover, the ontologies depicted in Figure 40 are designed to be reused together through this concepts in common or through specialization. For example, ROSS and OSDEF are reused together through the concept of Program, which is common for both ontologies. On another example, CMPO does not have a concept in common with ROSS or OSDEF, however, it also adopts SPO as its core ontology, defining that an Change Request (the concept that we intend to reuse) *addresses* an Configuration Item, which is a specialization of Artifact (from SPO). Since the requirements described into specifications are Artifacts in the same context, we can reuse both ontologies together, through the concept of Artifact. In other words, requirements artifacts (specifications) are specializations of a concept described in CMPO, due to that, we can reuse these ontologies together. External ontologies can also be reused in our traceability model, however, the external ontologies will need to be adapted or re-engineered, if they are not grounded on UFO.

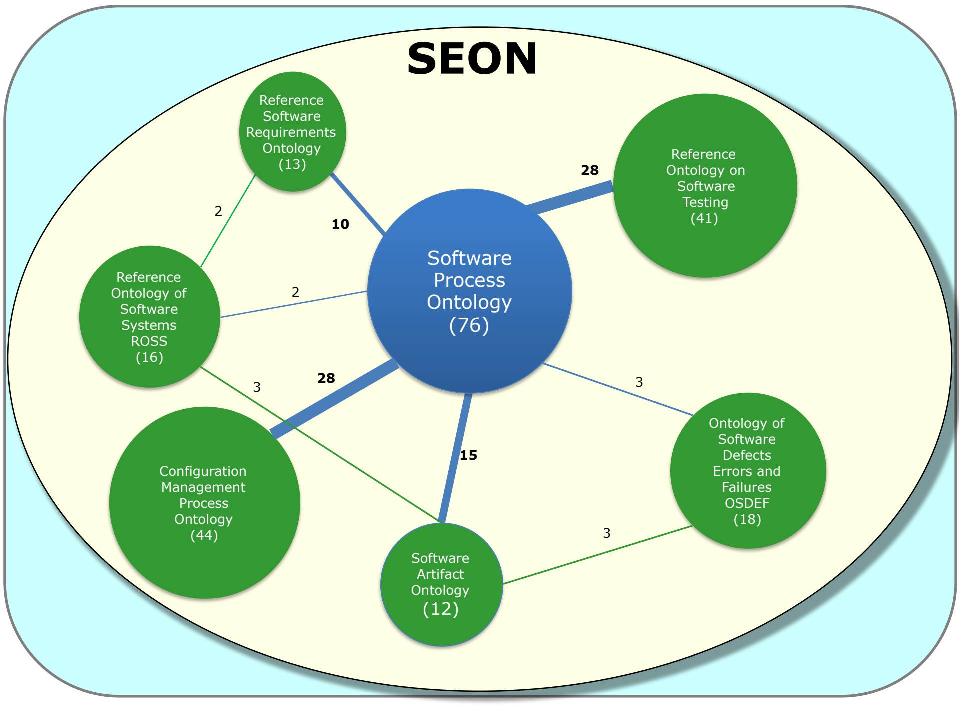

SEON
Reference Software Requirements Ontology (13)
Reference Ontology on Software Testing (41)
Software Process Ontology (76)
Reference Ontology of Software Systems ROSS (16)
Configuration Management Process Ontology (44)
Software Artifact Ontology (12)
Ontology of Software Defects Errors and Failures OSDEF (18)
28
10
2
2
3
28
15
3
3

Figure 40 – Graphical Representation of SEON ontologies used for SPARQL queries performed

From the operational ontology perspective, the concepts of Test Case from ROoST and Change Request from CMPO are included into the same RDF file that contains ROSS and OSDEF, which were developed on top of gUFO.[3] Relations between concepts from two different ontologies should be defined as object properties in the operational ontology. The reused concepts and relations should be defined as specializations of concepts from gUFO, as it was done with ROSS and OSDEF.

7.2 ATM System Scenario

In order to demonstrate our ontology-based requirements traceability approach, we adopted the ATM Software System as the scenario for our proof of concept. The ATM System was chosen because it is a well-know case inside the Software Engineering/Requirements Engineering literature, as it has been discussed and used as a proof of concept tool by many authors (DALPIAZ et al., 2013; TALLABACI; SOUZA, 2013; WANG et al., 2009). Further, a publicly available complete implementation of an ATM System, with detailed requirements, design models, implemented classes and test cases are provided by Bjork (2009). These common software development information items, can be used as base for the development of the different types of requirements that are proposed by our ontologies.

[3] This hierarchy is presented in Section 6.3.

Finally, the ATM system provides a viable scenario for a proof of concept because it is not purely software-based, since it has hardware-parts, external components, and connections to other systems (bank system), which are vital for the correct operation of the system, but that have to be controlled and managed by the embedded software system. This type of heterogeneity adds an amount of complexity to the scenario and approximate it to a real-world software system demand, which is beneficial for our objective, which is demonstrate that we can use domain ontologies without making it too complex, to a point that would make it impracticable.

7.2.1 ATM System Scope and Objective

It is important to clarify that a real ATM system is very complex, mainly because of the security protocols that are adopted in order to avoid frauds. For this proof of concept, we focus the scope on the core elements and features of an ATM System prototype and develop the proof of concept over the requirements, design elements and contextual information that are originally provided by Bjork (2009).

The main objectives of this proof of concept is to: (i) demonstrate that ROSS and OSDEF, as reference models, are able to properly represent the Software System domain, with a focus on the different types of requirements and software anomalies, with different levels of abstraction that exist throughout its implementation; (ii) demonstrate that an operational version of ROSS and OSDEF, properly built upon a series of axioms and rules that are provided by the operational version of UFO, gUFO, can be used as a linked data tool to properly classify requirements-based data and to provide traceability and reasoning capabilities over this data.

In order to achieve these objectives, we used the ATM System data provided by Bjork, in the format of textual requirements, design models and implementation classes, test cases and reported issues to instantiate the concepts defined in ROSS and OSDEF. However, as the data available in Bjork's artifacts were not enough to instantiate the entire ontology, some concepts had to be derived from them. For example, Bjork provides a set of textual Stakeholders Requirements, a series of design elements and Java-based classes that are based on these requirements. However, properly described System and Program Requirements are not provided. Furthermore, in order to obtain the complete list of system and program requirements for this experiment, we adopted the definition provided by ROSS that System Requirements and Program Requirements are solution-oriented refinements of the Stakeholder Requirements and reverse-engineered the Design Elements provided, based on each individual Stakeholder Requirement to obtain a list of program and system requirements that are consistent with this particular version of the ATM System.

Besides that, it is important to clarify that the data used in this proof of concept is based on the different types of requirements of the ATM System and on the relations that

Table 3 – Business Requirements for the ATM System.

Requirements	**Description**
BREQ001	Reduce service time for customer
BREQ002	Reduce costs with human personnel
BREQ003	Allow customer service outside of business hours
BREQ004	Improve reliability through process automation

exist between them, as it is proposed in ROSS' reference model. Moreover, for simplicity reasons, data was manually written in OWL files using the Protégé tool (NOY et al., 2001; NOY et al., 2003), as it is out of the scope of this thesis to provide or discuss semantic annotation methods and data retrieval approaches from documents or other types of Information Items that exist during Software System operation. However, we understand and point out to the fact that for large and very-large projects, some form of automation in the data retrieval process may be required. Finally, all queries were executed in Protégé using its native SPARQL plugin.

7.2.2 ATM System Requirements Data

As explained in the previous section, the data used for this proof of concept was directly reused from the data originally made available by Bjork (2009) and refined to fit ROSS' reference model.

This section presents the *requirements data* of Bjork's ATM system adapted to our proof of concept. The data consists of all the requirements for the ATM system and the other (Software) Artifacts that are related to them. For example, Change requests, Test cases and Programs of the ATM System.

The first step is to define the Business Requirements, which, as it was discussed in Chapter 4, represent the top-level requirements and the organization's goals towards the system-of-interest. As Bjork's prototype does not directly provide this specific type of requirement, we defined four Business Requirements based on the original ATM Software System description provided by Bjork and by the ones presented in the literature by other authors (TALLABACI; SOUZA, 2013; WANG et al., 2009) that used the same example. The four Business Requirements defined for this ATM System are presented in Table 3.

Following, the Stakeholder Requirements proposed in (BJORK, 2009) are presented in Table 4.

Table 4 – Stakeholder Requirements for the ATM System.

Requirements	**Description**
STREQ001	The ATM will communicate with the bank's computer over an appropriate communication link.
STREQ002	The ATM will service one customer at a time. A customer will be required to insert an ATM card and enter a personal identification number (PIN) – both of which will be sent to the bank for validation as part of each transaction. The customer will then be able to perform one or more transactions. The card will be retained in the machine until the customer indicates that he/she desires no further transactions, at which point it will be returned – except as noted below.
STREQ003	A customer must be able to make a cash withdrawal from any suitable account linked to the card, in multiples of $20.00. Approval must be obtained from the bank before cash is dispensed.
STREQ004	A customer must be able to make a deposit to any account linked to the card, consisting of cash and/or checks in an envelope. The customer will enter the amount of the deposit into the ATM, subject to manual verification when the envelope is removed from the machine by an operator. Approval must be obtained from the bank before physically accepting the envelope.
STREQ005	A customer must be able to make a transfer of money between any two accounts linked to the card.
STREQ006	A customer must be able to make a balance inquiry of any account linked to the card.
STREQ007	A customer must be able to abort a transaction in progress by pressing the Cancel key instead of responding to a request from the machine.
STREQ008	The ATM will communicate each transaction to the bank and obtain verification that it was allowed by the bank. Ordinarily, a transaction will be considered complete by the bank once it has been approved.
STREQ009	If the bank determines that the customer's PIN is invalid, the customer will be required to re-enter the PIN before a transaction can proceed. If the customer is unable to successfully enter the PIN after three tries, the card will be permanently retained by the machine, and the customer will have to contact the bank to get it back.
STREQ010	If a transaction fails for any reason other than an invalid PIN, the ATM will display an explanation of the problem, and will then ask the customer whether he/she wants to do another transaction.

Table 4 – Stakeholder Requirements for the ATM System.

Requirements	**Description**
STREQ011	The ATM will provide the customer with a printed receipt for each successful transaction, showing the date, time, machine location, type of transaction, account(s), amount, and ending and available balance(s) of the affected account ("to" account for transfers).
STREQ012	The ATM will also maintain an internal log of transactions to facilitate resolving ambiguities arising from a hardware failure in the middle of a transaction. Entries will be made in the log when the ATM is started up and shut down, for each message sent to the Bank (along with the response back, if one is expected), for the dispensing of cash, and for the receiving of an envelope. Log entries may contain card numbers and dollar amounts, but for security will never contain a PIN.

As proposed by ROSS, the Stakeholder Requirements must be further refined into System Requirements and Program Requirements, as they are the solution-oriented goals towards the system-of-interest. Tables 5 and 6, respectively, present the System Requirements and the Program Requirements for the ATM Software System and the Stakeholder Requirement from which they were derived.

Table 5 – System Requirements for the ATM Software System.

Requirements	**Description**	**Derived from**
SYSREQ001	The ATM must have a proper internet connection in order to communicate with the bank system.	STREQ01
SYSREQ002	Internet Connection must be encrypted with AES encryption all the time.	STREQ01
SYSREQ003	System must be implemented with Java EE technologies, in order to maintain consistency with other systems in the bank.	STREQ01
SYSREQ004	The ATM must have a Card reader installed. The Card reader driver must be natively compatible with the OS installed in the ATM Machine.	STREQ02
SYSREQ005	The ATM must have a Cash Dispenser peripheral installed. The Cash Dispenser driver must be natively compatible with the OS installed in the ATM Machine.	STREQ04

Table 5 – System Requirements for the ATM Software System.

Requirements	**Description**	**Derived from**
SYSREQ006	The ATM must have an Envelope Acceptor peripheral installed. The peripheral driver must be natively compatible with the OS installed in the ATM Machine.	STREQ05
SYSREQ07	The ATM must have a printer installed. Printer driver must be natively compatible with the OS installed in the ATM machine.	STREQ12

Table 6 – Program Requirements for the ATM Software System.

Requirements	**Description**	**Derived From**
PROGREQ001	Following the organization's pattern, class *NetworkToBank* must implement methods for opening and closing a connection with the banking system.	STREQ001
PROGREQ002	Operations for reading, ejecting and retaining a card must be implemented in the system.	STREQ002
PROGREQ003	The withdrawal screen must confirm with the customer the amount of cash to be withdrawn.	STREQ003
PROGREQ004	Before a withdrawal transaction starts, the ATM System must confirm if the customer has funds (money plus account limits) to perform the withdrawal. If the customer has not enough funds a message shall be displayed to the customer: "Not enough funds to perform this operation". A log must be recorded.	STREQ003
PROGREQ005	An operation for dispensing cash must be implemented in the software. The amount of cash dispensed must be checked by the peripheral and persisted in the system.	STREQ003

Table 6 – Program Requirements for the ATM Software System.

Requirements	**Description**	**Derived From**
PROGREQ006	The ATM must confirm that the envelope was properly deposited by reading a bar code printed in the envelope. A message shall be displayed in for the customer: “Deposit is now subject of bank analysis”. If the customer fails to deposit the envelope within the timeout period, or presses cancel, the transaction will be considered canceled and a message shall be displayed to the customer: “Deposit not concluded”. A log must be recorded.	STREQ004
PROGREQ007	Before a transfer transaction starts, the ATM System must confirm with the bank system if the customer has funds (money plus account limits) to perform the transfer. If the customer has not enough funds a message shall be displayed to the customer: “Not enough funds to perform this operation”. A log must be recorded.	STREQ005
PROGREQ008	The transfer screen must ask the customer to confirm both accounts (number, owner and agency), before the transaction starts.	STREQ005
PROGREQ09	Balance Inquiry must be primarily presented at the screen. The balance screen shall also present a button where the customer can choose to print the balance.	STREQ006
PROGREQ010	A constraint must be implemented to prevent the customer to print the balance more than 2 times in the same day.	STREQ006
PROGREQ011	The Cancel button present in the customer console shall throw an exception that needs to be captured and for the session to cancel the ongoing transaction.	STREQ007
PROGREQ012	Every major service provided by the ATM (withdrawal, transfer, deposit and balance inquiry) is considered a transaction and must be communicated to the bank. The implementations of these services shall verify if the transaction to be executed is available to the customer.	STREQ008

Table 7 – ATM Software System Programs.

Program	**Description**	**Program Requirement**
Prog001	Bank System Connection	ProgReq001
Prog002	ATM Session	ProgReq002, ProgReq014
Prog003	Withdrawal	ProgReq003, ProgReq004, ProgReq005
Prog004	Deposit	ProgReq006
Prog005	Transfer	ProgReq007, ProgReq008
Prog006	Balance Inquiry	ProgReq009, ProgReq010
Prog007	General Transaction	ProgReq011,ProgReq012
Prog008	Operator Terminal	ProgReq013
Prog009	Receipt Printing	ProgReq015
Prog010	Log Creation	ProgReq016

Table 6 – Program Requirements for the ATM Software System.

Requirements	**Description**	**Derived From**
PROGREQ013	An Operator Terminal shall be implemented and present the functionalities of: release retained card and consult/print system logs.	STREQ009
PROGREQ014	An option for using another service should be available for the customer after completing or canceling the ongoing transaction.	STREQ010
PROGREQ015	Deposit, transfer and withdrawal services shall print a receipt for the customer after transaction is completed. The receipt shall display the type of operation executed, the day, the value of the transaction and the accounts involved.	STREQ011
PROGREQ016	Transactions, envelope, cash-dispenser and error logs shall be implemented and should be accessible only through the operator terminal.	STREQ012

With all requirements for the experiment listed, we need to define the data relative to the Programs.[4] Furthermore, it is important to explain that in order to adapt Bjork's ATM System requirements data for this experiment, we used the source code and the design models provided by Bjork. Table 7 presents the list of the Programs for the ATM and the Program Requirements from which they implement.

Besides, Bjork also provides lists of Test Cases and Change Requests to the ATM Software System example, which we will use to capture a small list of Defects that may

[4] As discussed in Chapter 4, the concept of Program was reused from SwO, it denotes a System Element (a part of the Software System) that is responsible to produce a result when executed inside a machine.

Table 8 – List of Defects on the ATM Software System and the Programs in which they inhere.

Defect	**Description**	**Program**
Defect001	Withdrawal Screen is not double checking the amount of money to be withdrawn by the customer.	Prog003
Defect002	The constraint that prevents the customer from printing more than one receipt of balance inquiry in the same banking session is not working.	Prog006
Defect003	Receipt for transfer is not being properly printed.	Prog009
Defect004	ATM is closing the session after printing a receipt, without asking the client if she desires to perform another operation.	Prog002
Defect005	ATM is not persisting a log with information of retained cards.	Prog010

exist in the ATM Software System, shown in Table 8. This piece of data will be useful for querying data using the concepts defined in OSDEF.

Table 7.2.2 presents a small list of test cases proposed for the ATM Machine, with the related use cases, functionality being tested and expected output to the test.

Table 9 – Test Cases for ATM

Test Case	**Use Case**	**Function Being Tested**	**Expected Output**
TC001	Session	System reads a customer's ATM card	Card is accepted; System asks for entry of PIN
TC002	Session	System rejects an unreadable card	Card is ejected; System displays an error screen; System is ready to start a new session
TC003	Session	System accepts customer's PIN	System displays a menu of transaction types
TC004	Transaction	System handles an invalid PIN properly	The Invalid PIN Extension is performed
TC005	Withdrawal	System asks customer to choose an account to withdraw from	System displays a menu of account types
TC006	Withdrawal	System asks customer to choose a dollar amount to withdraw	System displays a menu of possible withdrawal amounts
TC007	Deposit	System asks customer to choose an account to deposit to	System displays a menu of account types
TC008	Deposit	System asks customer to enter a dollar amount to deposit	System displays a request for the customer to type a dollar amount
TC009	Deposit	System asks customer to insert an envelope	System requests that customer insert an envelope
TC010	Transfer	System asks customer to choose an account to transfer from	System displays a menu of account types specifying transfer from
TC011	Transfer	System asks customer to choose an account to transfer to	System displays a menu of account types specifying transfer to

TC012	Transfer	System asks customer to enter a dollar amount to transfer	System displays a request for the customer to type a dollar amount
TC013	Inquiry	System asks customer to choose an account to inquire about	System displays a menu of account types
TC014	Inquiry	System performs a legitimate inquiry transaction properly	System prints a correct receipt showing correct balance;
TC015	Transaction	Transaction Recording in the log	System records transaction correctly in the log.

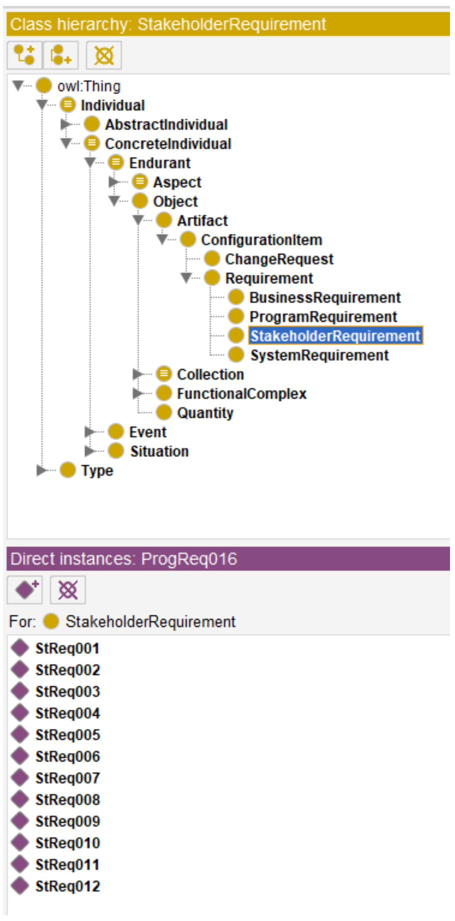


Figure 41 – Stakeholder Requirements represented as individuals in Protégé

7.3 Traceability and Information Recovery as SPARQL queries

From a Requirements Management perspective, it is very important for an Organization to be able to develop and maintain traceability over the requirements of Software Systems in operation. This type of capability is considered one of the bases for software system quality improvement, being repeatedly discussed and encouraged in standards (ISO, 2017; ISO, IEC, 2017a) and maturity models (CMMI Institute, 2018).

In this context, we propose that operational ontologies developed based on domain reference ontologies and populated with data generated during the software process can support requirements traceability through SPARQL queries. The queries are able to navigate the graph defined in the operational ontologies, using the relations and concepts of the ontology. Moreover, the data structured in a graph facilitates the visualization of indirect relations that exists between artifacts that are part of the software process.

Based on the presented list of software-related artifacts, we can use SPARQL for querying over the data of the ATM System that is registered as Individuals in the operational versions of ROSS and OSDEF. Figure 41 depicts the Stakeholder Requirements from the ATM System being represented as Individuals in Protégé.

Moreover, it is important to explain that SPARQL was chosen over SQL, as it was done in (DALL'OGLIO; SILVA; PINTO, 2010), because of the way that data is organized. Both SPARQL and SQL are languages created to query data, in RDF and in the relational model, respectively. The relational model structures data based on tables, columns and rows. Moreover, columns, as attributes, can be defined as keys, to create relations between tables. In general, the primary key of table A is used as foreign key in table B, thus creating an association between A and B. In other words, the relations between tables in the relational model are defined by values that are used to connect rows (one or many) in table A to rows in table B.

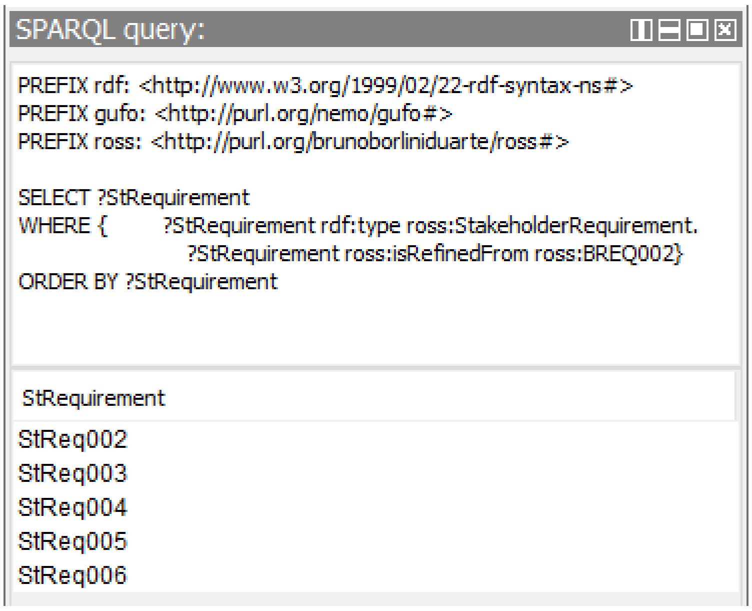
On the other side, RDF structures data in graphs, using the subject, predicate and object format. The relationship between two nodes of the graph is represented by the predicate, the edge between both nodes.

For many years, the ontologies/conceptual modeling community have been working under the assumption that the knowledge that exists in a conceptual model is within the relations between the concepts of the model, not in the concepts themselves. In this context, we have chosen SPARQL as the query language adopted for this work over SQL because of the way that relations (predicates) between concepts (subject and object) are represented in RDF/SPARQL. We believe that the graph format is naturally compatible with the structure that is defined by the ontologies. In other words, we believe that the knowledge in the Requirements Traceability process is in the relations (*traces*) between Requirements and other Software Artifacts, which are better represented in RDF/SPARQL than in the Relational Model/SQL.

Figure 42 presents a very simple example of direct utilization of the concepts and relations of the ontologies to recover a piece of information about the ATM Software System. The results are a list of the Stakeholder Requirements,[5] that are refined from Business Requirement[6] *BREQ002*.

Moreover, SPARQL also allows the user to query on the opposite direction of the relations (Object Properties) that are defined in the ontology. This can be very useful when the user wants to retrieve information from a trace that exists in the opposite direction of a relation in the ontology. It also helps to avoid overly-complicated queries, as all relations in a operational ontology will have a *Domain* and a *Range* that represents its intended direction. Figure 43 presents this case: the original relation presented in ROSS is that System Requirements[7] and Program Requirements[8] are *refined from* Stakeholder Requirements, however we can use the caret operator (^)[9] to create a much simpler query to discover

[5] Stakeholder Requirements for the ATM were presented in Table 4.
[6] Business Requirements for the ATM were presented in Table 3.
[7] System Requirements for the ATM were presented in Table 5.
[8] Program Requirements for the ATM were presented in Table 6.
[9] Caret operator semantic is *get inverse property*.



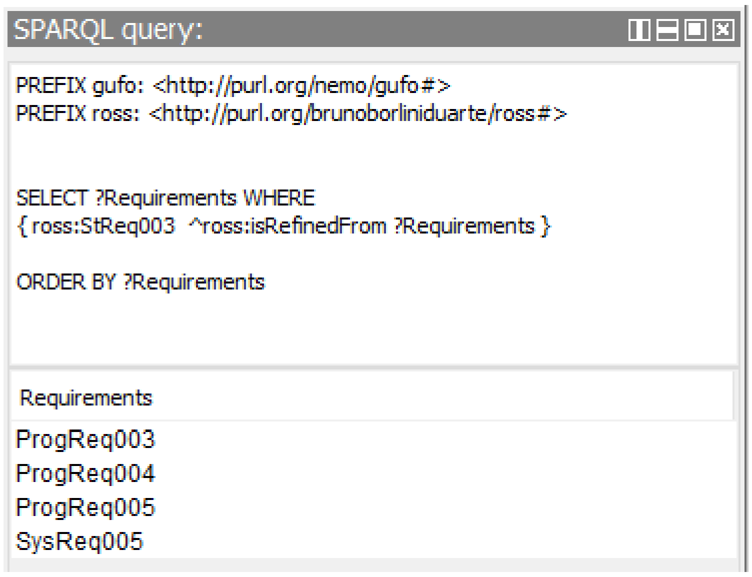
SPARQL query:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX gufo: <http://purl.org/nemo/gufo#>
PREFIX ross: <http://purl.org/brunoborliniduarte/ross#>

SELECT ?StRequirement
WHERE {     ?StRequirement rdf:type ross:StakeholderRequirement.
              ?StRequirement ross:isRefinedFrom ross:BREQ002}
ORDER BY ?StRequirement
```

```
StRequirement
StReq002
StReq003
StReq004
StReq005
StReq006
```

Figure 42 – Query 1 - Queries all Stakeholder Requirements which are refined from Business Requirement *BREQ002*.



SPARQL query:

```
PREFIX gufo: <http://purl.org/nemo/gufo#>
PREFIX ross: <http://purl.org/brunoborliniduarte/ross#>


SELECT ?Requirements WHERE
{ ross:StReq003  ^ross:isRefinedFrom ?Requirements }

ORDER BY ?Requirements
```

```
Requirements
ProgReq003
ProgReq004
ProgReq005
SysReq005
```

Figure 43 – Query 2 - Searches System Requirements and Program Requirements that are refined from *StReq003*.

all system and program requirements that are refined from *StReq003*. More precisely, the caret operator is used to get the inverse path of the object property *isRefinedFrom* that are associated with the entity *StReq003*.

Furthermore, as operational ontologies are designed as graphs, it is possible to navigate the entire graph based on the relations (object properties) defined in the ontology. For this particular case, supposing that a Requirements Engineer wants to discover if a Business Requirement is related to a Program, a part of the ATM system, he would need to navigate the whole ontology in order to retrieve such information, as Business Requirement is a concept that exists in the Business Layer and Program exists in the Machine Layer.

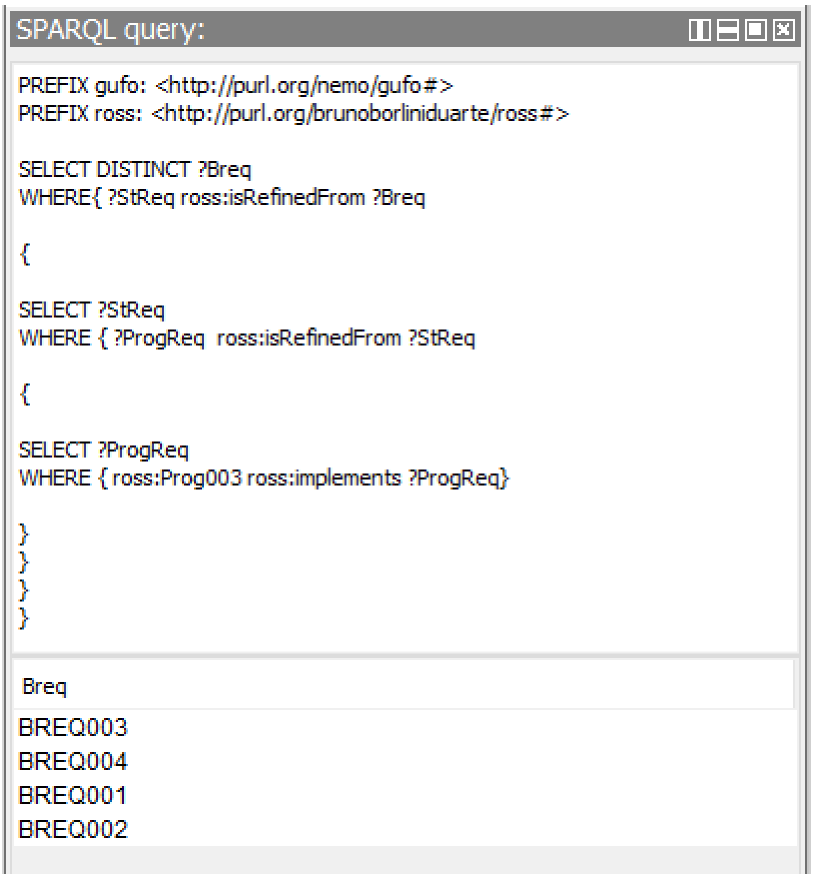
Figure 44 uses the concept of sub-query to navigate the graph to show that Program *Prog003* is indirectly related to Business Requirements *Breq001*, *Breq002*, *Breq003* and *Breq004*. The relation is retrieved by navigating the ontology between the concepts of Programs, Program Requirements, Stakeholder Requirements and Business Requirements. More precisely, the inner part of the query, which is always executed first, recovers the Program Requirements that are implemented by *Prog003*, this information is used as input data for the middle query, which presents the Stakeholder Requirements related to *Prog003* and finally, the outer query uses the result of the middle one as input to deliver the final answer. The *DISTINCT* clause in *SELECT* works exactly like in normal SQL, by returning only unique values in the result.

This type of query is particularly useful because it is capable to be used as a powerful requirements traceability tool, as it is able to “travel” through the ontology and use the results of each SPARQL query as input for another one, in other part of the ontology. For example, another sub-query could be added to retrieve data about change requests in the ATM Software System, as the engineer can easily associate Programs and Program Requirements with Change Requests, and use the same type of query to navigate through them. For illustration purposes, Figure 45 depicts the results of Query 3 in an instance diagram.

Besides, as mentioned earlier in this thesis, ROSS is intended to act as a backbone ontology in the Software System domain. This means that it is intended to be reused in combination with related ontologies to further extend the capability, to classify and reason over a dataset. For example, ontologies that are part of SEON (cf. Section 3.5) and grounded on UFO, like as the Reference ontology on Software Testing (ROoST) (SOUZA; FALBO; VIJAYKUMAR, 2013), the Goal-Oriented Requirements Ontology (GORO) (BERNABÉ et al., 2019) and the Configuration Management Process Ontology (CMPO) can be reused[10] in the model to enhance its capability to represent certain parts of the domain, as it allows that more information can be classified, retrieved and reasoned with. This type of ontology reuse is important and highly suggested by SABiO because, although ROSS' reference model was designed to be complete on its own, it obviously does not account for all Artifacts that are produced during the Software process. In this context, reusing ontologies that are all part of an ontology network, interlinked, created under the same design parameters and grounded on the same foundational ontology, becomes a much simpler task.

Figure 46 depicts the reuse of the concept of Change Request (CR) from the Configuration Management Process Ontology (CMPO). Change Request is a Information Item that describes a request for modification on other software artifacts that are under the

[10] Ontologies are usually reused together through existing common concepts or by directly specialization.



```
PREFIX gufo: <http://purl.org/nemo/gufo#>
PREFIX ross: <http://purl.org/brunoborliniduarte/ross#>

SELECT DISTINCT ?Breq
WHERE{ ?StReq ross:isRefinedFrom ?Breq

{

SELECT ?StReq
WHERE { ?ProgReq  ross:isRefinedFrom ?StReq

{

SELECT ?ProgReq
WHERE { ross:Prog003 ross:implements ?ProgReq}

}
}
}
}
```

Breq
BREQ003
BREQ004
BREQ001
BREQ002

Figure 44 – Query 3 - Association between Program *Prog003* with the Business Requirements that are directly impacted by it

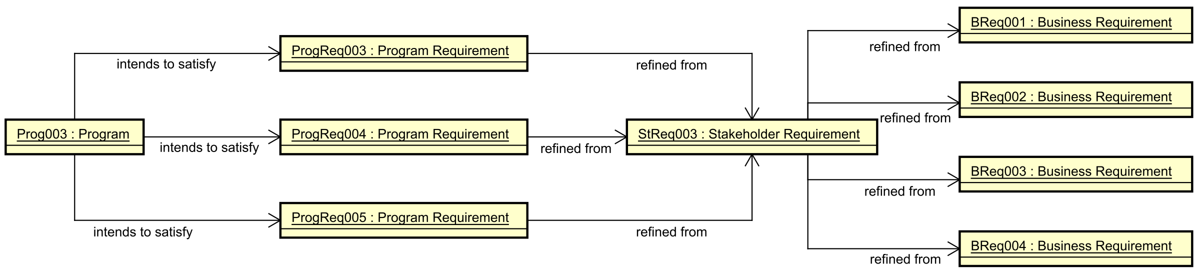
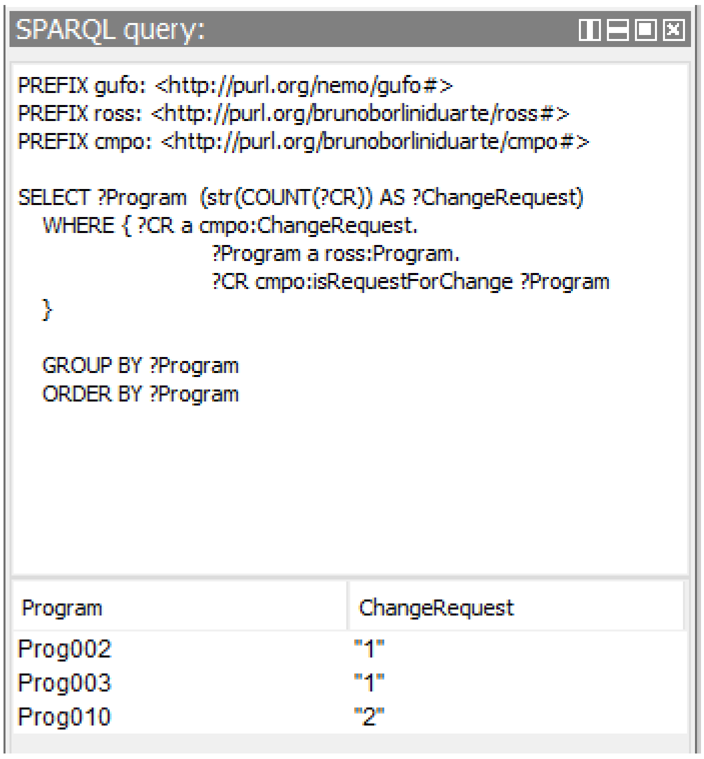

Prog003 : Program
intends to satisfy
intends to satisfy
intends to satisfy
ProgReq003 : Program Requirement
ProgReq004 : Program Requirement
ProgReq005 : Program Requirement
refined from
refined from
refined from
StReq003 : Stakeholder Requirement
refined from
refined from
refined from
refined from
BReq001 : Business Requirement
BReq002 : Business Requirement
BReq003 : Business Requirement
BReq004 : Business Requirement

Figure 45 – Instance Diagram representing Query 3 and its result.



SPARQL query:

```
PREFIX gufo: <http://purl.org/nemo/gufo#>
PREFIX ross: <http://purl.org/brunoborliniduarte/ross#>
PREFIX cmpo: <http://purl.org/brunoborliniduarte/cmpo#>

SELECT ?Program  (str(COUNT(?CR)) AS ?ChangeRequest)
   WHERE { ?CR a cmpo:ChangeRequest.
                     ?Program a ross:Program.
                     ?CR cmpo:isRequestForChange ?Program
   }

   GROUP BY ?Program
   ORDER BY ?Program
```

Program	ChangeRequest
Prog002	"1"
Prog003	"1"
Prog010	"2"

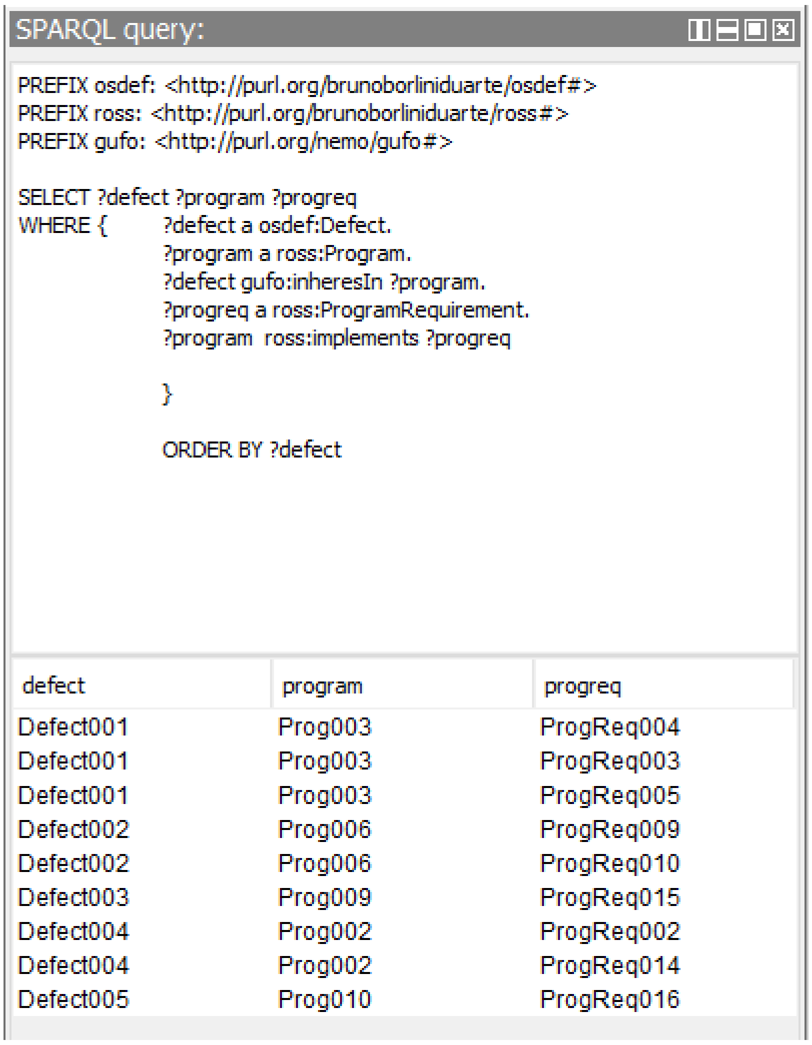
Figure 46 – Query 4 - Counts the number of Change Requests that are related with the Programs that are part of the ATM System.

Configuration Management process. The query searches for all registered CRs, counts and classifies them, based on which Program they are associated. This kind of information is important because it allows the engineer to clearly see which parts of the software system are generating more requests for changes.

Finally, as ROSS and OSDEF share the concept of Program, we can use it to connect both ontologies, in order to be able to represent a relation between Defects and the Program Requirements. Figure 47 depicts the execution of SPARQL query that associates the Defects[11] and Program Requirements, based on Programs which implement them.

Besides, it is important to explain that the variables in the query, which respectively represent Defects, Programs and Program Requirements may be repeated in the result multiple times because of the cardinality of the relations that exist between these concepts. For example, *defect001* is repeated three times in the results because as it *inheres in Prog003*, which is responsible to implement three Program Requirements and the existence of the Defect may have a negative impact on all Program Requirements that are implemented in the specific Program. On the other side, as *defect005* exists in *Prog010*, which only implements one Program Requirement, *ProgReq016*, it is only displayed on the results one time, as it is only able to directly impact on *ProgReq016*. Besides, if the the query was more complex and associated Defects with Business Requirements, the same phenomenon would be accentuated, since a Business Requirement can be associated with a very large

[11] A list of known defects of the ATM is presented in Table 8.



```
PREFIX osdef: <http://purl.org/brunoborliniduarte/osdef#>
PREFIX ross: <http://purl.org/brunoborliniduarte/ross#>
PREFIX gufo: <http://purl.org/nemo/gufo#>

SELECT ?defect ?program ?progreq
WHERE {        ?defect a osdef:Defect.
               ?program a ross:Program.
               ?defect gufo:inheresIn ?program.
               ?progreq a ross:ProgramRequirement.
               ?program  ross:implements ?progreq

               }

               ORDER BY ?defect
```

defect	program	progreq
Defect001	Prog003	ProgReq004
Defect001	Prog003	ProgReq003
Defect001	Prog003	ProgReq005
Defect002	Prog006	ProgReq009
Defect002	Prog006	ProgReq010
Defect003	Prog009	ProgReq015
Defect004	Prog002	ProgReq002
Defect004	Prog002	ProgReq014
Defect005	Prog010	ProgReq016

Figure 47 – Query 5 - Relates Defects from OSDEF with Program Requirements in ROSS based on concept of Program that exist in both ontologies

amount of Defects that may occur in a Software System over the years.

Obviously, the decision of how many Programs will be responsible to implement each Program Requirement that is raised for a project is based on the design created for the software system by the responsible Organization. For example, the Organization may decide that each Program Requirement must be implemented in only one Program in a one-to-one relation, which could be done, based on the reference model of ROSS. On the other side, the complete opposite can also be done, and the relation between Programs and Program Requirement could be from m-to-n cardinality. Figure 48 depicts a part of the results of Query 5 in an instance diagram.

A variation of Query 5 is depicted in Figure 49. Query 6 lists and associates all Stakeholder Requirements with the Programs that are part of the ATM System, using the concept of Program Requirement that exists between them. This query structure can also be used to created a Traceability Matrix (RAMESH et al., 1995), associating all types of requirements and even other Artifacts that are part of the Software System. Moreover,

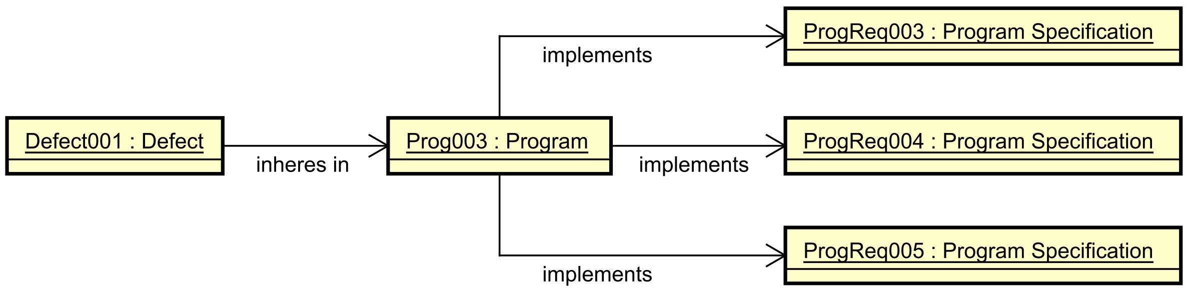

Defect001 : Defect
inheres in
Prog003 : Program
implements
implements
implements
ProgReq003 : Program Specification
ProgReq004 : Program Specification
ProgReq005 : Program Specification

Figure 48 – Instance Diagram representing a part of Query 5's result.

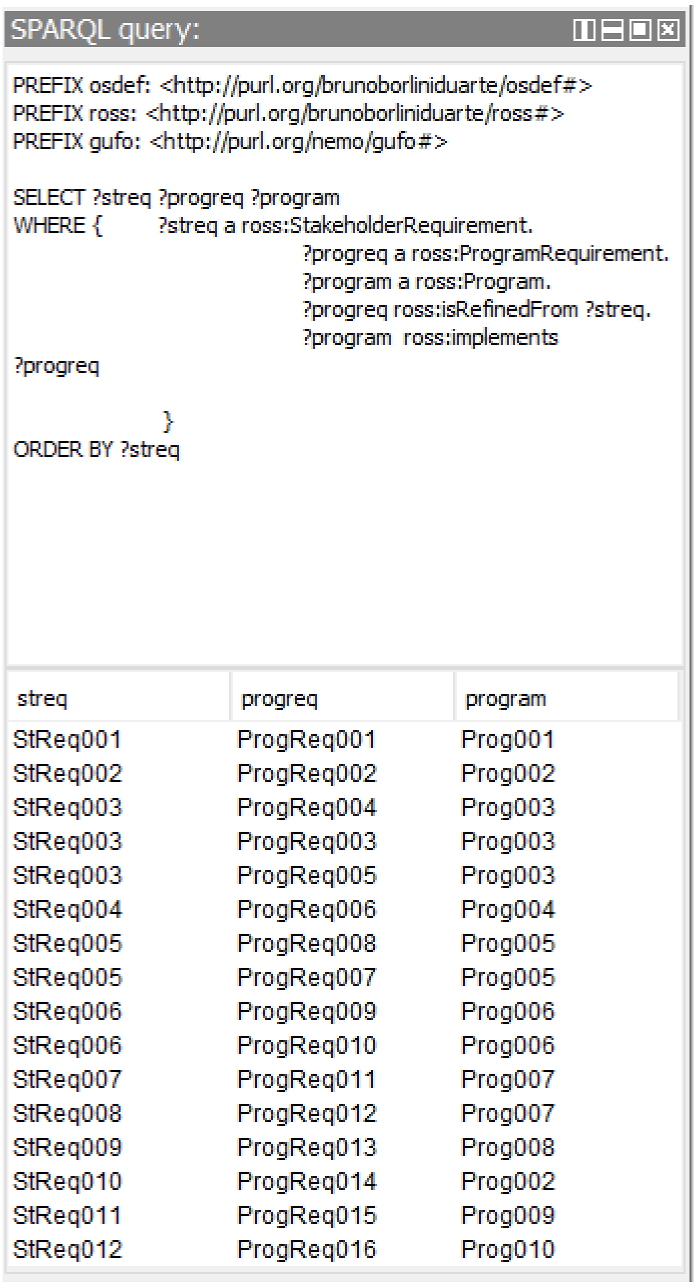
the query structure can be further extended to associate more elements of the domain, by adding other variables in the query (concepts of the domain). For example, it is possible to create a variation to include the concepts Business Requirements and Defects as variables in the query. However, as more variables are added, the query becomes more complex, which can make the query unpractical or at least, require some kind of front-end system interface to support the software engineer with the results that are retrieved.

Finally, Query 7, depicted in Figure 50, represents the relation between Test Cases and Program Requirements. To do that, we reuse the concept of Test Case presented in ROoST, in our operational ontology. The Test Case is associated with the concept of Program that exists in ROSS. According to the definition in ROoST, a Text Case *tests* a specific part of the code (Programs are constituted by code) to verify if the expected behavior of that code is capable to fulfill what was specified in the Requirements. Figure 51 depicts a part of the results of Query 7 in an instance diagram.

7.4 Prototype Tool: Requirements Tracker

The ontologies and queries presented in this thesis were used to create a user-friendly web-based prototype tool for requirements traceability. The development of the tool was conducted as a Computer Science undergraduate thesis (DUARTE, 2021), supervised by us. For this work, we used the Apache Jena framework (MCBRIDE, 2002) and Java EE technologies, such as JavaServer Faces (JSF) (BURNS; KITAIN, 2006) and PrimeFaces (PRIMEFACES, 2021), to create a tool that provides a friendly, web-based, user interface for the SPARQL queries presented in this thesis. The tool used components provided by JSF and by PrimeFaces to create graphs that represent the relations between the requirements of the ATM Machine, presented in Section 7.2. Moreover, the tool also removed the necessity of using Protégé to perform queries.

Figures 52 and 53 are screen captures of the tool being used. Figure 52 represents the execution of query 1, depicted earlier in Figure 42, and executed with Business Requirement *BREQ004* as input. The graph is generated using the *mindmap* component


SPARQL query:

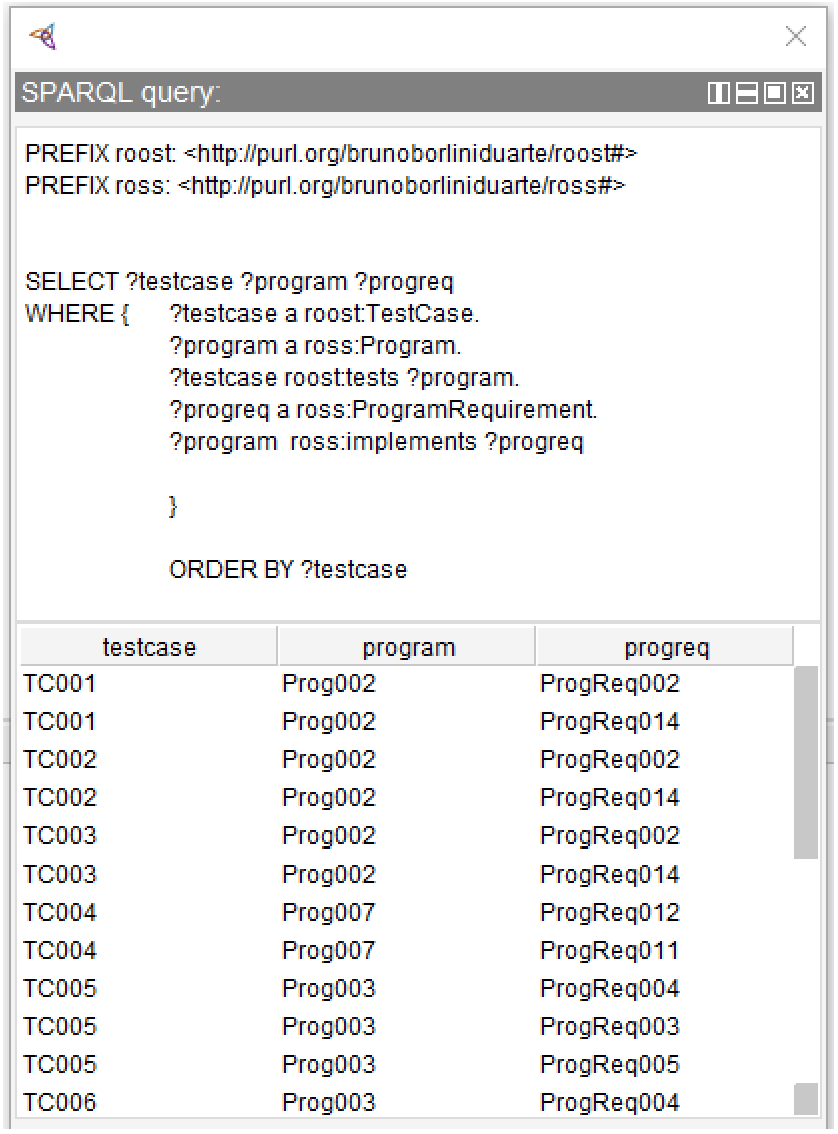
```
PREFIX osdef: <http://purl.org/brunoborliniduarte/osdef#>
PREFIX ross: <http://purl.org/brunoborliniduarte/ross#>
PREFIX gufo: <http://purl.org/nemo/gufo#>

SELECT ?streq ?progreq ?program
WHERE {        ?streq a ross:StakeholderRequirement.
                        ?progreq a ross:ProgramRequirement.
                        ?program a ross:Program.
                        ?progreq ross:isRefinedFrom ?streq.
                        ?program  ross:implements
?progreq

                }
ORDER BY ?streq
```

streq	progreq	program
StReq001	ProgReq001	Prog001
StReq002	ProgReq002	Prog002
StReq003	ProgReq004	Prog003
StReq003	ProgReq003	Prog003
StReq003	ProgReq005	Prog003
StReq004	ProgReq006	Prog004
StReq005	ProgReq008	Prog005
StReq005	ProgReq007	Prog005
StReq006	ProgReq009	Prog006
StReq006	ProgReq010	Prog006
StReq007	ProgReq011	Prog007
StReq008	ProgReq012	Prog007
StReq009	ProgReq013	Prog008
StReq010	ProgReq014	Prog002
StReq011	ProgReq015	Prog009
StReq012	ProgReq016	Prog010

Figure 49 – Query 6 - Lists all Stakeholder Requirements and relates them with Programs that are part of the ATM System



```
PREFIX roost: <http://purl.org/brunoborliniduarte/roost#>
PREFIX ross: <http://purl.org/brunoborliniduarte/ross#>


SELECT ?testcase ?program ?progreq
WHERE {     ?testcase a roost:TestCase.
            ?program a ross:Program.
            ?testcase roost:tests ?program.
            ?progreq a ross:ProgramRequirement.
            ?program  ross:implements ?progreq

            }

            ORDER BY ?testcase
```

testcase	program	progreq
TC001	Prog002	ProgReq002
TC001	Prog002	ProgReq014
TC002	Prog002	ProgReq002
TC002	Prog002	ProgReq014
TC003	Prog002	ProgReq002
TC003	Prog002	ProgReq014
TC004	Prog007	ProgReq012
TC004	Prog007	ProgReq011
TC005	Prog003	ProgReq004
TC005	Prog003	ProgReq003
TC005	Prog003	ProgReq005
TC006	Prog003	ProgReq004

Figure 50 – Query 7 - Relates Test Cases to the Program Requirements based on the association between Test Case-Program

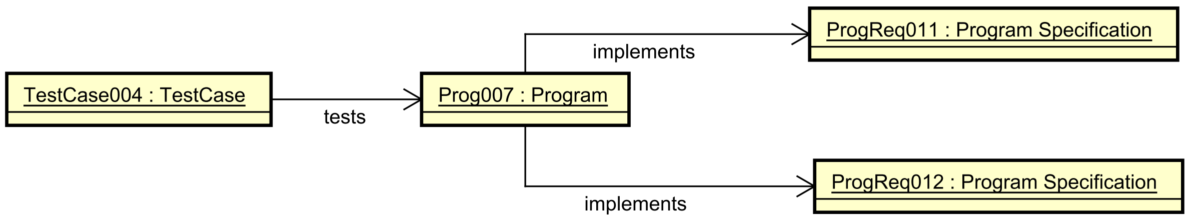

ProgReq011 : Program Specification
implements
TestCase004 : TestCase
tests
Prog007 : Program
ProgReq012 : Program Specification
implements

Figure 51 – Instance Diagram representing a part of Query 7's result.

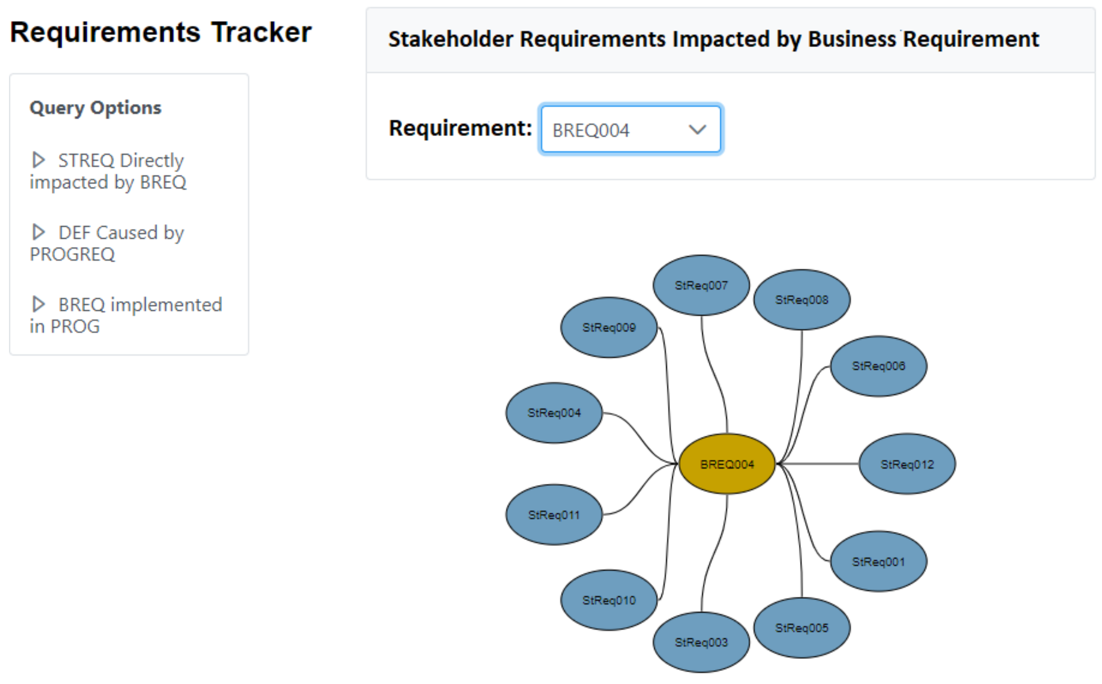


Figure 52 – Execution of query 1 in the Requirements Tracker tool.

of the PrimeFaces library. The edges in the graph represents the relation *refined from* that exists between Business and Stakeholder Requirements.

In turn, Figure 53 shows the result of executing query 3, depicted earlier in Figure 44, and executed with Program *Prog003* as input. The edges in the graph represent the indirect relation that exists between Business Requirements and Programs.

The components used for the creation of the graphs support Asynchronous JavaScript and XML (AJAX) technology. This means that the graphs generated are not static, they are objects of the system, with which the user can interact. In this case, when the user interacts with any of the requirements depicted in the graph, the system will present additional information about the requirement.

7.5 Related Works

This section presents other proposals about the use of reference models for requirements traceability that are related to the work that is developed in this thesis.

7.5.1 Traceability Reference Model

As mentioned earlier in Section 2.2, the traceability reference model proposed by Ramesh and Jarke (2001) is one of the most referenced studies in the requirements

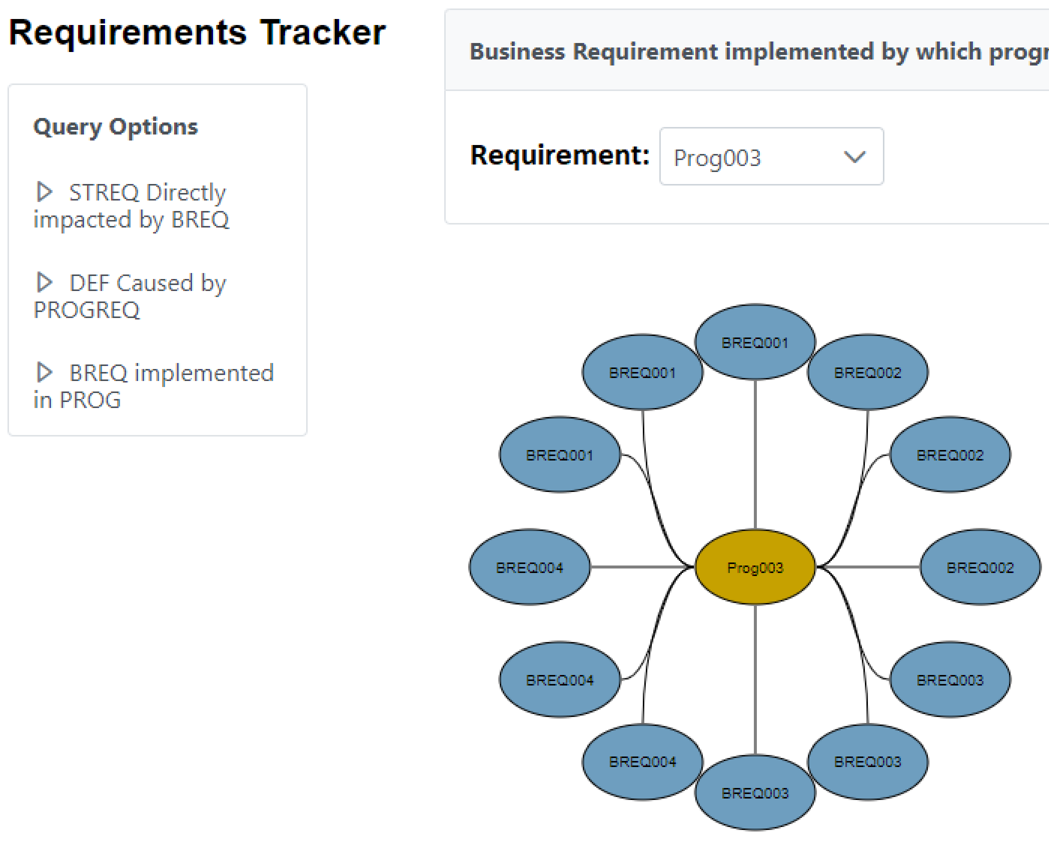


Figure 53 – Execution of query 3 in the Requirements Tracker tool.

traceability research field. They focus their model in the analysis and the representation of the objects (artifacts) and trace link types that are commonly used and produced in Software Systems projects to create a reference model for traceability around them. Moreover, they distinguish their models according to its use. The low-end traceability model (Figure 54) is suitable to users that do not require a complex traceability system. As it is shown in the present work, the traceability links are the relations that exist between system elements. In other words, low-end traceability can be successfully applied to model requirements dependencies or the relation between a requirement and a system component.

The authors defend that organizations that choose to adopt a more advanced traceability model do so because of the necessity for much richer traceability schemes than low-end users. Due to that, authors decided to divide the high-end model in four sub-models: Requirements Management, Design Allocation, Compliance Verification and Rationale Management, so that each sub-model is focused on a specific part of the of the process. Figure 55 depicts the Requirements Management sub-model. Moreover, it is important to explain that we choose to show the requirements management sub-model instead of the other models that are proposed because it is focused on the relations that

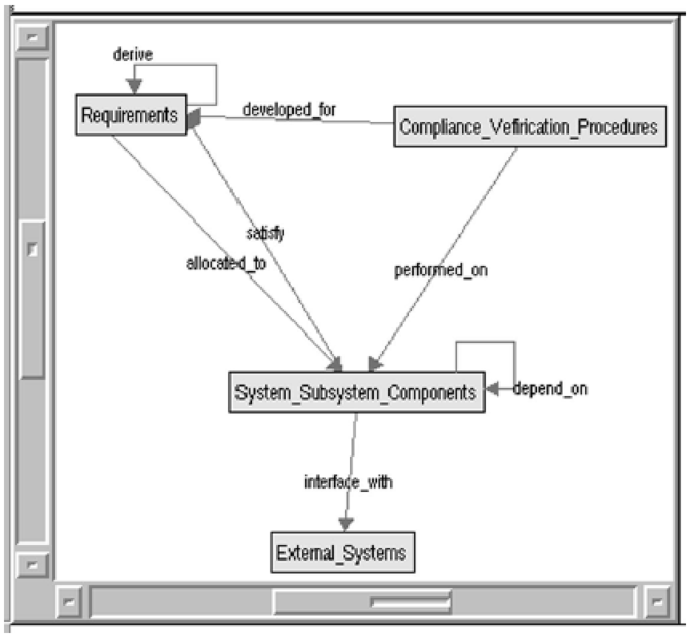


Figure 54 – Low-end traceability model (RAMESH; JARKE, 2001).

exist around the concept of requirement, which is one of the focuses of the work being presented here.

Further, they also propose the utilization of four types of traceability links, to support the semantics between the information items that are being managed: (i) dependency links, to represent dependency between elements; (ii) satisfies links, to represent which elements are created to satisfy a requirement; (iii) evolution links to identify the origin of an element; and (iv) rationale links to identify the reasons behind the creation of an element.

7.5.2 Traceability Meta-Model

The traceability meta-model (GOKNIL; KURTEV; BERG, 2014), depicted in Figure 56, is an approach for trace generation and validation of traces between requirements and system architecture elements. The authors' argument is that establishing traces between requirements and architecture elements can be beneficial for change management, at source code level, as software architects base their architectural design decisions on the essential system requirements, thus making direct relations between those requirements and the architectural elements of the system, which in its turn can be used as a reference model for traceability. Authors also provide a tool that supports the approach, based on the Eclipse platform. The traces for the approach can be generated manually into the provided tool or with a degree of automation, when the tool is combined with an Architecture Description Language with support for code generation.

The proposed meta-model is divided in three parts: a requirements and an architec-

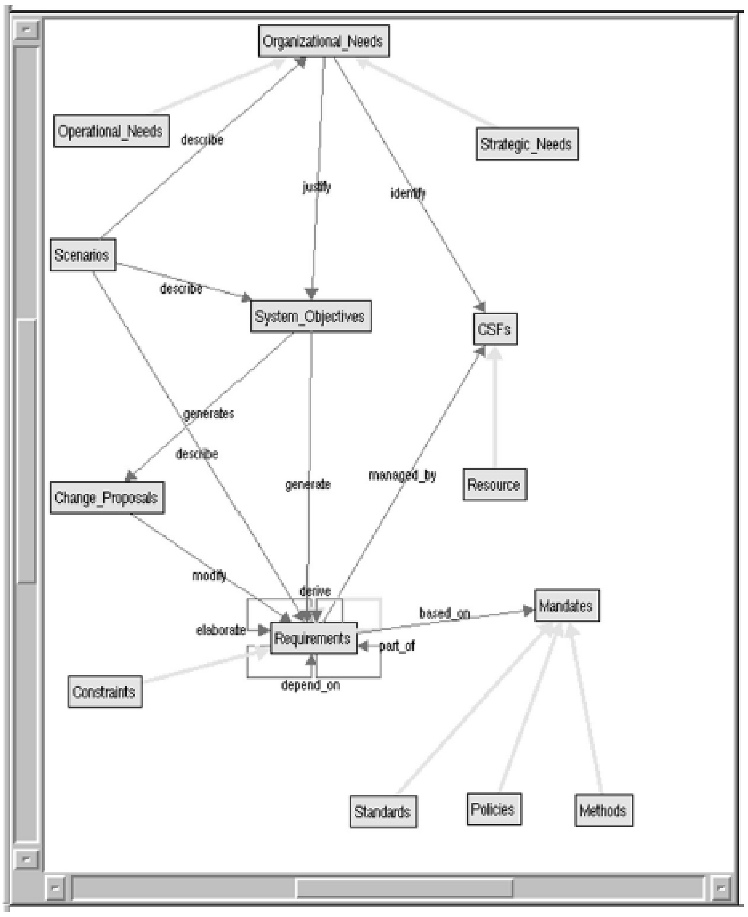


Figure 55 – Requirements Management Traceability sub-model (RAMESH; JARKE, 2001).

ture meta-model and a trace meta-model that is responsible for connecting the previous two and defines two types of traces: *Satisfies* and *AllocatedTo*, which are reused from the work of Ramesh and Jarke (2001).

In comparison with our work, the meta-model proposed by Goknil and colleagues is fairly simple, as it does not take into consideration other artifacts that are part of the software process. Besides, their meta-model focuses only on the definition of a single type of requirement that is connected to a single type of architecture component by the *Satisfies* and *AllocatedTo* traces. Furthermore, in our approach we do not specifically define a Trace super-type for all the relations that exist in the ontology or a concept the defines Trace as piece of information that must be retrieved/recovered, such as a Configuration Item. Instead, we consider that all elements that are directly or indirectly connected through the ontology network can be traced, based on the the utilization of inference tools and SPARQL queries with the operational ontology.

7.5.3 Semantic Traceability Model

The Semantic Traceability Model (ALONSO-RORÍS et al., 2016) is part of a platform that is composed by three major models: a technical business model, the Semantic

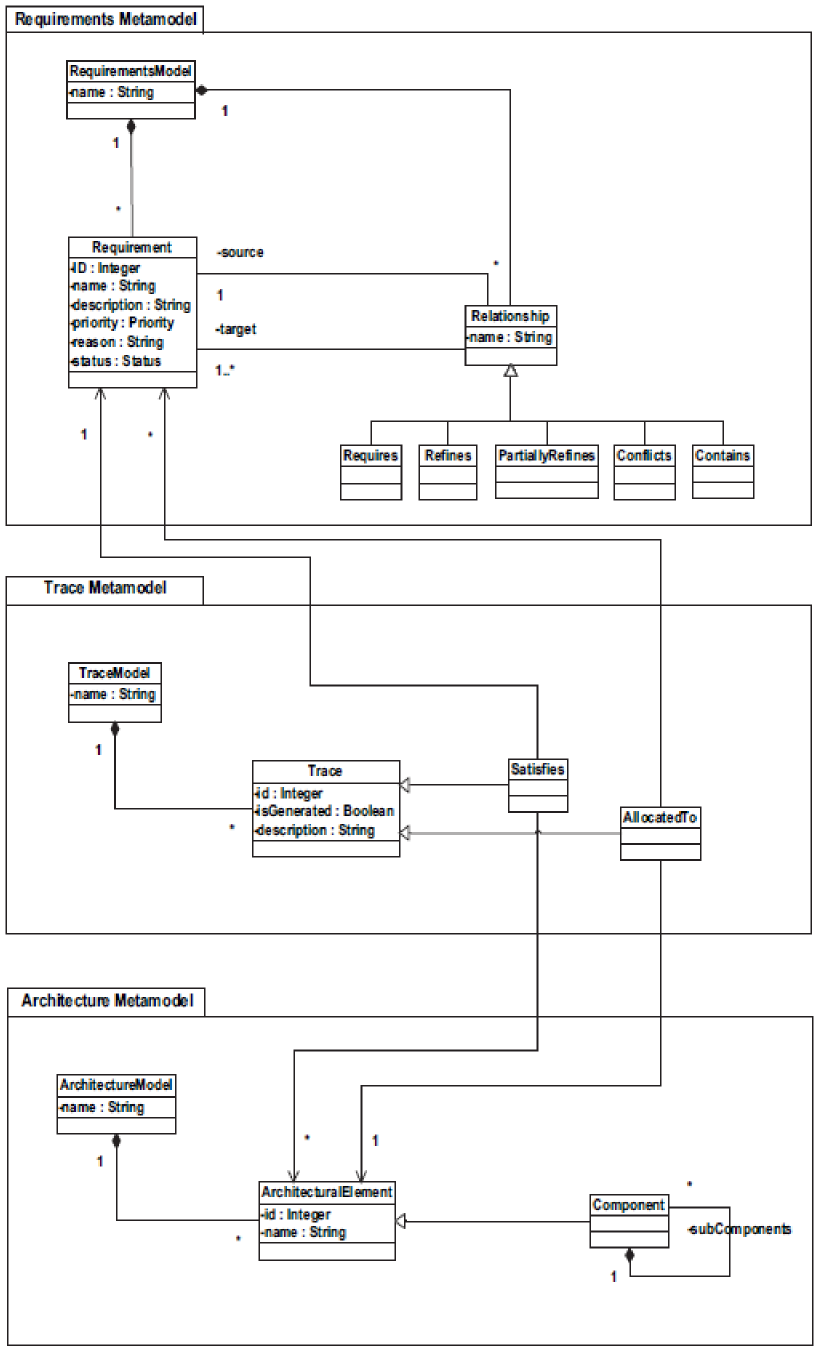


Figure 56 – Trace meta-model for requirements and architecture elements (GOKNIL; KURTEV; BERG, 2014).

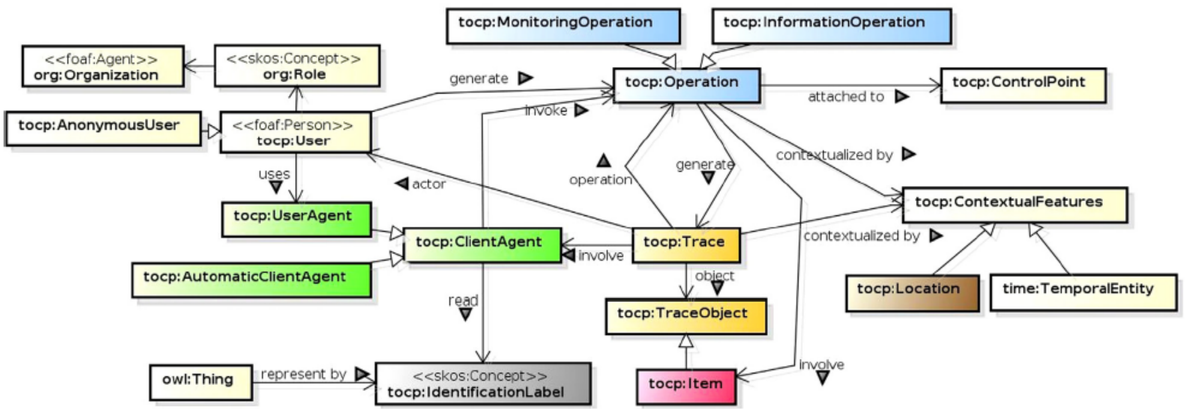


Figure 57 – Semantic Traceability Model concepts and trace characterization (ALONSO-RORÍS et al., 2016).

Model and a reference architecture. Figure 57 depicts concepts and relations defined for the traceability domain by Alonso-Rorís and colleagues. Trace and TraceObject are the main concepts of the model, they are responsible to represent the core information retrieved in a operation and also are connected to the other concepts of the model, such as User and Item.

The model is composed by an ontology and a set of inference rules that were built under the principles of the NeOn methodology (SUÁREZ-FIGUEROA; GÓMEZ-PÉREZ; FERNÁNDEZ-LÓPEZ, 2012), an ontology engineering method focused on the development of operational ontologies. Since the model was built over the NeOn methodology, it is based on Semantic Web technologies, such as the RDF framework (W3C, 2004; W3C, 2014a) and also reuses other well-known Semantic Web ontologies, such as the friend-of-a-friend (FOAF) vocabulary (BRICKLEY; MILLER, 2015). The Semantic Traceability Model is intended to provide explicit ground knowledge about the traceability domain. Authors also argue that the model can be extended to fit the particular needs of a user, however, they do not explicit how this extension should be performed by users.

The semantic model is probably the approach that most resembles ours, as it is based on an ontology specifically developed for it and it intends to use operational ontologies as a tool to provide support for the traceability process. However, different from our model, the work of Rorís and colleagues is focused on the operational ontology, in the sense that their ontology is not grounded on a foundational ontology. Besides that, their approach is focused on the traceability domain, where they specifically present the concepts of Trace as a process and TraceObject as an object. However, these concepts are not further developed and there is no support from an foundational ontology. In other words, the concepts used in the model are not formally defined and supported by a well-grounded theory.

In comparison, our model is grounded on a foundational ontology and is based on the artifacts and relations that are part of software systems domain, specially the different types of software requirements that were discussed in Chapter 4.

Because of that, we do not specifically define the aforementioned concepts in our ontologies. Instead, we believe that the traces between software artifacts are already defined as the relations between ontology concepts. In other words, every software system artifact that exists in the software process is a possible TraceObject in our model, even the ones that are not specifically defined in ROSS or in OSDEF, as we design our model in modules, with the objective of supporting reuse with other ontologies on the software system domain.

Table 10 lists the concepts of ROSS, OSDEF and the ones that are reused from the ontologies in SEON and compares them with the concepts proposed by the works presented in this section. The table provides an easy visualization of all concepts that are part of each reference model for requirements traceability discussed in this thesis.

Moreover, it is important to point out that none of the proposals presented in this section nor ours claim to represent all possible concepts and relations of the domain in their models. Furthermore, we believe that it is not possible to provide a solution/proposal that is capable of mapping and representing all the concepts and relations of the software domain, since every Organization is unique, with their own processes, daily activities and artifacts. In this context, we propose conceptual models about software systems and the capability to use a Software Engineering ontology network (SEON) as a tool for implementing ontology-based requirements traceability based on the relations that exist between domain concepts.

7.5.4 Infrastructure for Semantic Document Management

The Infrastructure for Semantic Document Management (ISDM) (ARANTES; FALBO, 2010; MACHADO; ARANTES; FALBO, 2011; FALBO; BRAGA; MACHADO, 2014) is an ontology-based architecture and tool, developed to provide requirements traceability based on the semantic annotation of text documents. The semantic annotation is done based on the conceptualization of the Software Requirements Ontology (NARDI; FALBO, 2008). The authors' argument is that despite the current advances in electronic documentation along with the boom of collaborative management tools (such as wiki engines), desktop text editors are still the most frequently solution used by software organizations when it comes to electronic documentation.

They explain that Requirements Documents hold a considerable amount of information that are to be mainly interpreted by human readers, such as requirements statements and use case descriptions. Their hypothesis is that requirements traceability can be more easily achieved if the semantic content of the requirements documents could be exposed in

Table 10 – Comparison between the concepts of ROSS and OSDEF with the Approaches presented in Section 7.5.

ROSS/OSDEF/Reused	**Traceability Reference Model**	**Traceability Metamodel**	**Semantic Traceability Model**
Organization	Operational Need	Requirements Model	Organization
Stakeholder	Strategic Need	Requirement	User
Business Rule	Resource	Relation	Trace
Business Regulation	Change Proposal	Trace Model	Location
Business Requirement	Scenario	Satisfies Trace	Temporal Entity
Business Requirement Specification	Requirement	Allocatedto Trace	Item
World Assumption	Constraint	Architecture Model	Control Point
Machine Assumption	Standard	Architectural Element	Control Param
Stakeholder Requirement	Policy	Component	Monitoring Operation
Stakeholder Requirement Specification	Method		Information Operation
System Requirement	Component		
System Requirement Specification	Design Element		
System Component	Assumption		
Program	Argument		
Hardware Equipment	Function		
Program Requirement	External System		
Program Requirement Specification	Critical Sucess Factor		
Loaded Program Copy	Test		
Machine	Prototype		
User-Generated Failure	Simulation		
User			
Fault Manifestation Failure			
Defect			
Fault			
Change Request			

order to allow visibility of the data and the relationships embedded in the document, and if the semantic content of each document version is extracted and registered into a version control system.

Their architecture is composed of three parts: (i) the Semantic Annotation Module is responsible for allowing users to semantically enrich an Open Document Format (ODF) template; (ii) the Data Extraction and Versioning Module, which is responsible for extracting the semantic content from an annotated document whenever a new version of that document is checked into the Semantic Data Repository (SDR); (iii) the Search and Traceability Interface Module is responsible for providing an API (Application Programming Interface) that allows users and other systems to perform ontology-based searches and data traceability towards the Data Repository.

The approach for requirements traceability presented by the authors is based on searching for information into requirements documents and specifications. In comparison to our work, they use a software requirements ontology as the conceptualization behind the proposed architecture, however, the ontology is implicit inside the documentation structure and tools provided, in the sense that it is not explicitly presented as a reference model. Moreover, our work could benefit from this proposal by using their approach with our models, in order to import requirements data from documents into our operational ontologies.

7.6 Chapter Summary

This chapter presented the last contribution of our work, an ontology-based approach for requirements traceability that uses domain ontologies as both reference models and operational tools for recovering traceability information.

To demonstrate our approach in an experimental way, we adapted a prototype of an ATM System (BJORK, 2009) that is well-know in the requirements literature and imported the prototype data provided into our operational versions of ROSS and OSDEF, using the RDF language (W3C, 2014b) and the Protégé tool (NOY et al., 2001).

After that, we queried the ontology data using the SPARQL language (W3C, 2013) to recover traceability links information and to enable an easy visualization of relations that exist between the software system requirements and the other Artifacts that are part of the domain. We used relations that exist between the concepts in the ontologies act as the traceability links that are recovered, instead of defining a concept of trace directly in the reference model, as other proposals did to represent a generic relation that exist between multiple information items. In other words, every relation that exists in the ontologies is a traceability link that can be used to connect two or more concepts, even if they are not directly related. This situation can be clearly seen in Query 3, which relates two concepts that are not directly connected in ROSS, (Business Requirement and Program), but that can be associated using the other concepts that exist around them (Stakeholder and Program Requirements).

Although the ATM prototype system is a fairly simple, it is big enough to demonstrate our approach, as it provides useful data about many artifacts produced during the software process, such as requirements, programs, test cases, change requests, known defects, among others. For time reasons, in this proof of concept we created the operational ontologies and entered the data of the ATM system manually. However, we do understand that for larger datasets, manual input of data becomes impractical. To work around this limitation, frameworks and tools for extracting data from text-based documents, such as requirements specifications, change requests and bug reports could be found in the literature. After that, the user only needs to adapt the data into Turtle Syntax (W3C, 2014b) using some type of string manipulation.

Unfortunately, the data of the prototype ATM system provided by Bjork is limited in some aspects. For example, it does not provide information about the business constraints and external business regulations that may exists towards the implementation of a real ATM system, by a bank. Because of that, we could not create queries using all concepts presented in ROSS and in OSDEF, such as the concept of Business Constraint, that is part of the Business Layer of ROSS (cf. Section 4.2). However, as mentioned earlier in this work, both ontologies are created to be as complete and to represent their respective domains as best as possible, so if a user of our ontologies has this type of data, it can be used in new queries very easily.

In fact, a very important aspect of this work is that it can be adapted based on the needs of the user with little effort. Starting with the operational implementation of ROSS and OSDEF, a user can reuse concepts and relations of other existing ontologies in the model. For example, in Query 4 the concept of Change Request was not originally part of the models of ROSS and OSDEF. However, as it originally defined in CMPO, the reuse was simplified. That happens because: (i) CMPO is also grounded in UFO; (ii) operational ROSS and OSDEF, built on top gUFO, already present several concepts and relations from UFO that can be specialized, if needed; and (iii) since CMPO is a part of SEON, it also reuses SPO as its core ontology. More precisely, it reuses concepts of the artifact sub-ontology of SPO, the same sub-ontology that ROSS and OSDEF also reuse. The direct consequence of these facts is that the connection between CMPO and our ontologies already exists, through SPO and UFO, it only needs to be properly defined in the operational level, with the support of gUFO.

Besides, the queries presented in this work can be altered and others can be created by the user, as the knowledge and the necessary technologies are free, open-source and widely available on the Web. For example, a user may want to create a query that relates different Stakeholders (sales and financial departments), with the Stakeholder and System Requirements of a new software system. As these concepts already exist in ROSS, the user would only need to create a new query that respects the relations between the concepts.

In section 7.4 we have discussed the implementation of a requirements tracker system prototype, based on the ontologies presented in this thesis. The prototype uses Jena, JSF and PrimeFaces to depict the results of the SPARQL queries presented in Section 7.3 as graphs. This prototype is our first attempt at developing an ontology-based requirements management system that can support the visualization of the relations that exist between requirements and other artifacts that are produced during the software process. We believe that such visualization can improve requirements traceability activity and enhance change impact management.

Finally, the chapter ended by summarizing and discussing different proposals that are part of the requirements traceability literature which are related to the work developed for this thesis.

8 Conclusions

This final chapter summarizes the main contributions of this thesis to the state-of-the-art in the conceptual modeling of the Software System domain in Section 8.1. Section 8.2 discusses research limitations of this work and Section 8.3 presents our ideas for future research, involving ROSS, OSDEF, and the Software Systems domain.

8.1 Research Contributions

In Section 1.2, we defined the research hypothesis for this thesis as follows:

Well-grounded domain ontologies can support semantic requirements traceability. Reference ontologies about the software systems domain can define the important domain concepts (entities) and the relations that exists around them. Operational ontologies act as machine-readable tools that provide the support for querying over the system artifacts data produce during the software process.

Based on this hypothesis, we develop the following general and specific objectives for this work:

The general objective was to: *provide a reference model for the software systems domain that focuses particularly on requirements traceability, that allows us to trace from low-level concepts such as programs at runtime all the way to high-level concepts such as business requirements.*

which was further decomposed into the following general objectives:

- *Develop a set of reference ontologies that are able to properly represent and cover the software system domain, by proposing new reference models and reusing existing ones;*
- *Apply the knowledge from the reference models into operational ontologies, in order to allow the implementation of ontology-based traceability;*
- *Demonstrate, through a proof of concept, the feasibility of a requirements traceability based on operational ontologies;*

In order to tackle these objectives, we performed a research in the scientific literature and in software-related international standards, in search of the base knowledge that would be the foundation for the development of this work and for the tools that would enable us to test our hypothesis. Throughout the research, it became clear that requirements were a crucial part for the development of the reference models that already exist in the literature.

Because of that, we would need to develop a reference model that was able to improve over the existing ones, expanding over the concept of Requirement itself and relating it, with higher level business objectives, rules, internal policies and lower level program-related information, such as change requests and defect/failure reports. This part is, in particular, is very important because the lack of connection between all stages of the software process was a limitation encountered on most of the proposals found in our research.

In this context, the first two contributions for this work were two domain reference ontologies that could be reused together. The first one, ROSS (Chapter 4), is an ontology about the software systems domain focused on requirements and that inherits the contributions of the seminal work from Zave and Jackson (1997). ROSS extends our previous work (DUARTE et al., 2018) since it reuses and improves the notion of *software* presented in SwO by associating it with other relevant concepts that were not represented in SwO and also by focusing on the figure of the software system as an entity that connects the machine with our world. ROSS also extends the concept of Requirement presented in RSRO to a much broader sense, since it define four types of Requirements, with distinct properties, that are a relevant part to the software system domain.

The second one, OSDEF (Chapter 5), is an ontology built over a well-know ontological design-pattern of events, proposed by Guizzardi et al. (2013), that was created to represent how *software anomalies* such as Defects, Errors and Failures are related to system elements.

Both ROSS and OSDEF were created using SABiO (Section 3.4) and grounded on UFO (Section 3.2), respectively, the ontology engineering method and the foundational ontology that were adopted.

Following, for the second and third general objectives, we developed operational versions of both, ROSS and OSDEF, using gUFO, the "gentle" version of UFO and used them with linked data technologies, through a proof of concept, to demonstrate that domain ontologies can be used as traceability and management tools for the data produced during the software process.

In this context, the last contribution is an approach to enable ontology-based requirements traceability over the artifacts produced during the software process (Chapter 7) using domain ontologies (reference and operational) and linked data technologies. This data was classified according to the concepts defined in ROSS and OSDEF. In addition, SPARQL queries were used to recover the relations between the data as traceability links. For example, this type of query enables an extended perspective of the data, in such a way that a user can see the relationship (traceability link) that exists between a defect that inheres in a component of the system (a program) and higher level stakeholder or business requirements. This perspective is possible because of the way that the model was designed and all types of requirements defined are related to other elements of the domain, following and expanding the original formula created by Zave and Jackson.

In other words, our first contribution are conceptual models about the software systems domain that compliment the theory already presented in SEON. The second contribution is the capability to use these conceptual models as a tool for implementing ontology-based requirements traceability based on the relations that exist between domain concepts.

In Section 7.4 we presented a prototype requirements tracker tool created using the knowledge presented in this thesis, in combination with Jena, JSF and PrimeFaces frameworks. The prototype provides a web-based user interface and a graph-visualization of the query results. The tool also eliminates the necessity of loading the operational ontologies files in Protégé to execute the queries.

Finally, we believe that the objectives defined in Chapter 1 were reached, although we also believe that the models produced can be further improved.

8.2 Research Limitations

ROSS and OSDEF were built following SABiO's recommendations for verification and validation, they systematically answer all competency questions that were raised and both ontologies were instantiated using real-world scenarios. However, a significant limitation for both ontologies is that we were not able to empirically evaluate them using data from software systems that are in current operation. The direct consequence of that was that we were not able to conclude the Design Science Methodology, since it requires that the solution (the artifacts) being designed are used in a real-world context. To contour this limitation and to perform an empirical evaluation, we used data from a well-known exemplar in the literature (BJORK, 2009).

This particular limitation brings about another potential one, that we could not evaluate the performance of the queries when executed with data from very large systems, i.e., we could not perform a test with thousands of requirements, change requests, relations and some degree of dependency between the elements of the software system. However, it is important to highlight that this potential limitation does not, necessarily, bring about a make-or-break type of situation, since, in the last few years, OWL and SPARQL computational performance greatly improved with the release of OWL 2 and SPARQL 1.1.

Another possible limitation is related to the fact that SABiO does not prescribe any activities or support processes for anti-pattern detection and simulation in the models created based on it. Anti-patterns (GUIZZARDI; SALES, 2014) are modeling structures that are syntactically valid parts of conceptual models, but are prone to result in unintended domain representations. A study has shown (SALES; GUIZZARDI, 2015) that anti-patterns are recurrent even in models produced by experienced researchers, because of the increasing complexity of the models being created. In this context, the addition of a support process for anti-pattern detection and treatment after phase 2 of SABiO, Ontology Capture and Formalization, could improve the overall quality of the models created based on the method.

Besides that, as we mentioned in Section 7.6, we could not create queries using many concepts that are presented and discussed in both ontologies because of the limited data used during the proof of concept. That lack of a robust dataset is not a major concern on its own, as we believe that the hypothesis behind the experiment could be demonstrated without a “perfect” dataset. However, with a more complete data we could have created other types of queries and explored other parts of the ontologies, focusing more on OSDEF and even on reusing external ontologies.

This work does not provide a technique or tool to capture requirements data from text documents or from project management tools and import them into the operational ontologies used in Chapter 7. Due to that, a user would need to create such a tool for herself or to input the data manually into the operational ontology, which becomes unfeasible for very large software projects.

Finally, our work does not provide any process or technique for the users to develop and maintain their requirements management process. However, we do indicate that the user needs to have some degree of management over the data produced during software projects. We believe, based on earlier studies and on the fact that both ontologies were built using the knowledge presented in the scientific literature and on international standards, that organizations that follows ISO 12207 or that implements the early levels of CMMI can use and benefit from our work.

8.3 Future Work

As mentioned in Section 8.2, we do not provide a technique or tool to import requirements data into the operational ontologies. In this context, a possible future work is the development of a tool capable of importing data from management tools, such as Jira,[1] Trello[2] or Monday.com.[3] This importing tool can be developed without much effort, since project management tools usually provide the functionality of exporting data to

[1] <https://www.atlassian.com/software/jira>
[2] <https://trello.com/>
[3] <https://monday.com/>

widely supported formats, such as CSV and JSON. After the data is properly exported, the proposed tool can covert it to the RDF/Turtle syntax, obeying the concepts and relations defined in the operational ontologies.

Another possible tool-related future work for this thesis is to improve the prototype tool presented in Section 7.4. The requirements tracker could be improved by the utilization of other Web-based technologies focused on the development of rich user interfaces. For example, traceability matrices could be easily derived from the queries presented Section 7.3. Moreover, a dashboard that associates all the important artifacts of the software system could be built to depict the relations between these software-based artifacts. Furthermore, although PrimeFaces does not provide "Dashboard" or "Data Matrix" components natively, they can be implemented in the prototype system by using existing JavaScript libraries, specifically designed for the development of software systems management tools. In this context, traceability matrices and dashboards would be used as knowledge representation tools for the relations between requirements and other software artifacts, a representation that complements the graph-based one presented in the prototype. The impact analysis capability provided by such management tools can be a very useful business resource for an organization, specially for the ones that have software systems as their main product. Because of that, we believe that a professional tool can be developed, using the knowledge presented in this thesis.

An important future work for this thesis is applying the artifacts produced herein in the context of a real-world software system project. A design science project iterates over the activities of designing and investigating a problem. This project must exist in a larger Engineering Cycle, in which the result of the Design Cycle made, is transferred to the real world. For this thesis, we conducted the Design Cycle; however, we did not conclude the Engineering Cycle, due to the lack of data on a fully operational software system or an engineering sample that could be used. This kind of data is usually sensible data in the industry, being out of our scope for this thesis.

Another possible future work is to expand ROSS to discuss Software Systems in the context of their operation and to the services they provide in our world. These System Services are the entities that are responsible for creating the Value Experience[4] over the utilization of operating software systems. In order to pursue this part of the research, we intend to reuse and to adopt the definitions and concepts presented in UFO-S (NARDI et al., 2015b; GRIFFO et al., 2017; GRIFFO et al., 2019; SALES et al., 2018), a core ontology of services that is grounded on UFO. Our preliminary hypothesis is that as Programs need to be loaded and executed inside a Machine to fulfill their ultimate function and produce value, Software Systems must be in constant operation, in order to provide the System

[4] A Value Experience is defined in COVER as a complex Event that is related to the achievement of the Goals of an Agent, that *participates in* the Value Experience as a Value Subject.

Services that are needed for our modern society. These System Services are responsible for creating the Value Experience that we perceive, for instance, when we listen to a music on Spotify or search for a product on Amazon.

Additionally, the queries presented in our proof of concept (cf. Chapter 7) are only demonstrating of what could be achieved when we associate reference ontologies of the software domain, like ROSS and OSDEF, with linked data technologies. In this context, another future work is based on the use of inference engines in the proposed models. Inference Engines (or Semantic Reasoners) are software applications that derive new facts or associations from existing information, based on inference rules (SINGH; KARWAYUN, 2010). This research is relevant because the use of semantic reasoners allow ontology engineers to derive new data from data that is already known. In other words, new information can be discovered based on the data that already exists in the operational ontology (SUÁREZ-FIGUEROA et al., 2012). Furthermore, inference engines like Pellet (SIRIN et al., 2007), Hermit (GLIMM et al., 2014) and Fact++ (TSARKOV; HORROCKS, 2006) are available as plugins inside Protégé. Besides that, linked data technologies are constantly evolving, supported by scholars and practitioners. Since this part of our research is related to this technologies, as the field evolves, our research will evolve as well.

Another important research that is strongly related to the work developed here is the one that relates requirements with software evolution (LEHMAN, 1980; LEHMAN, 1996). We intend to pursue this research directive as we believe that the domain ontologies discussed here and in previous works (DUARTE et al., 2018) are able to bring new arguments to the discussion and to improve the work that was already published by Wang et al. (2016).

Furthermore, we introduced COVER in this work to reuse it with the objective validate our ontologies based on its well-founded theory about value and risk, in order to demonstrate that our ontologies are capable to represent real-world situations. However, as we go further on the discussions about *Software Systems as Service Providers* and about *Software System Evolution*, the theory presented in COVER can be very useful for us, as the following questions emerge: *How changes in the software system affect the service and the value perspective of the users over it?* or *Since a Software System exists in partitions, is the risk experience caused by changes the same in all partitions?* and *What are the reasons for a software system to evolve? Does the value perspective and the value produced by the system also evolve with it?*

In this context, we can also reuse ROSS, OSDEF and COVER together to support the development of a (software) risk management ontology, as part of a more extended research in the software cybersecurity domain. ROSS defines the software artifacts and the requirements that are their essential properties. OSDEF ontologically defines the concepts of Failures, Errors and Faults, their relations based on the theory of Events presented in UFO and is associated to COVER through the concept of vulnerability, a disposition that in a cybersecurity domain must be discovered and treated, in order to avoid its manifestation as a (software) failure. COVER defines the concepts of value and risk as properties that are directly related to each other. COVER also associates value and risk in a complete model that represents the objects and events in a situation of lost of value over unmanaged risks. We believe that a software risk management reference ontology used together with other conceptual modeling works in the software cybersecurity area can support the development of behavior-based intrusion prevent systems (IPS).

Moreover, the assumptions used in this work, are originally part of a preliminary ontology of assumptions presented by Wang et al. (2016) that was not fully developed. Based on that, a possible future work lies in the proper definition of a (core) reference ontology of assumptions, based on an ontological analysis over the scientific literature, grounded on existing foundational ontology and that extends Wang and colleagues' work by fully characterizing the uncovered concepts and relations that exist around the concept of assumption. We believe that the development this ontology of assumptions can be very positive for our work, as it will allow us to further extend ROSS and OSDEF, by understanding the relations that exist between the assumptions reused in it and consequentially, improving the presented reference model to take full advantage of it. These relations are not defined by Wang. Our hypothesis here is that the assumptions that exist in the superior layers of the software systems act as constraints for the requirements and system components in the lower layers. For example, the assumptions made for the development of a software product in a Business environment will constrain the definition of System and Program Requirements. However this relations are not fully discovered and are objects for future research.

Technical Debt is defined as the obligation that a software organization incurs when it chooses a design or construction approach that is expedient in the short term but that increases complexity and is more costly in the long term. The ability to track and manage the technical debt is crucial for organizations, since the decision of cutting costs in the short term can lead to increasing complexity and costs in the long term (MCCONNELL, 2008). In this context, a possible future work is to associate techniques for calculating and managing the technical debt with our approach for requirements traceability based on reference models. We believe that the work presented here can be useful for the technical debt managing process, since it can capture the relations between high-level business requirements and the programs that are part of the software system.

Finally, we believe that many concepts and techniques related to domain ontologies presented in this thesis could be applied outside the specific area of software systems, in other domains such as governmental, medical and economic. We intend to research the possible applications of the techniques used here outside of the software system domain.

Bibliography

AHN, S.; CHONG, K. A feature-oriented requirements tracing method: A study of cost-benefit analysis. In: IEEE. *2006 International Conference on Hybrid Information Technology*. 2006. v. 2, p. 611–616. Cited in page 37.

ALAM, D. et al. Study of the dirty copy on write, a linux kernel memory allocation vulnerability. In: IEEE. *Intl. Conf. on Consumer Electronics and Devices (ICCED)*. 2017. Cited in page 79.

ALMEIDA, J. P. A. et al. *gUFO: A Lightweight Implementation of the Unified Foundational Ontology (UFO)*. 2019. <http://purl.org/nemo/doc/gufo>. Accessed: 2020-07-25. Cited 2 times in pages 47 and 92.

ALONSO-RORÍS, V. M. et al. Towards a cost-effective and reusable traceability system. a semantic approach. *Computers in Industry*, Elsevier, v. 83, p. 1–11, 2016. Cited 4 times in pages 11, 35, 123, and 125.

AMARAL, G. et al. Towards a reference ontology of trust. In: SPRINGER. *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. 2019. p. 3–21. Cited in page 88.

ARANTES, L. d. O.; FALBO, R. d. A. An infrastructure for managing semantic documents. In: *Proceedings of the 2010 14th IEEE International Enterprise Distributed Object Computing Conference Workshops*. 2010. p. 235–244. Cited in page 126.

AVIZIENIS, A. et al. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, IEEE, v. 1, n. 1, p. 11–33, 2004. Cited in page 82.

BARCELLOS, M. P.; FALBO, R. de A.; MORO, R. D. A well-founded software measurement ontology. In: *Proc. of the 6th International Conference on Formal Ontology in Information Systems (FOIS 2010)*. 2010. p. 213–226. Cited in page 45.

BELL, T. E.; THAYER, T. A. Software requirements: Are they really a problem? In: *Proceedings of the 2nd international conference on Software engineering*. 1976. p. 61–68. Cited in page 27.

BENEVIDES, A. B. et al. Representing a reference foundational ontology of events in sroiq. *Applied Ontology*, IOS Press, v. 14, n. 3, p. 293–334, 2019. Cited in page 44.

BERNABÉ, C. H. et al. Goro 2.0: Evolving an ontology for goal-oriented requirements engineering. In: SPRINGER. *International Conference on Conceptual Modeling*. 2019. p. 169–179. Cited 3 times in pages 53, 71, and 113.

BERNERS-LEE, T.; HENDLER, J.; LASSILA, O. The semantic web. *Scientific american*, JSTOR, v. 284, n. 5, p. 34–43, 2001. Cited in page 35.

BJORK, R. C. *ATM Simulation*. 2009. <http://www.cs.gordon.edu/courses/cs211/ATMExample/>. Accessed: 2020-07-05. Cited 5 times in pages 99, 100, 101, 128, and 133.

BORGIDA, A. et al. Techne: A (nother) requirements modeling language. *Computer Systems Research Group. Toronto, Canada: University of Toronto*, 2009. Cited 3 times in pages 71, 73, and 74.

BORST, W. N.; BORST, W. Construction of engineering ontologies for knowledge sharing and reuse. 1997. Cited in page 39.

BOURQUE, P.; FAIRLEY, R. E. et al. *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0.* : IEEE Computer Society Press, 2014. Cited 8 times in pages 17, 18, 20, 27, 28, 32, 69, and 78.

BRESCIANI, P. et al. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, Springer, v. 8, n. 3, p. 203–236, 2004. Cited in page 71.

BRICKLEY, D.; MILLER, L. Foaf vocabulary specification 0.99 (2014). *Namespace Document. Available online: http://xmlns. com/foaf/spec/(accessed on 23 November 2018)*, 2015. Cited in page 125.

BRINGUENTE, A. C. de O.; FALBO, R. de A.; GUIZZARDI, G. Using a foundational ontology for reengineering a software process ontology. *Journal of Information and Data Management*, v. 2, n. 3, p. 511, 2011. Cited 8 times in pages 9, 10, 42, 45, 52, 54, 55, and 67.

BRINGUENTE, A. C. O.; FALBO, R. d. A.; GUIZZARDI, G. Using a Foundational Ontology for Reengineering a Software Process Ontology. *Journal of Information and Data Management*, Brazilian Computer Society, v. 2, n. 3, p. 511–526, 2011. Cited in page 44.

BUDGEN, D. et al. Using mapping studies in software engineering. In: *Ppig*. 2008. v. 8, p. 195–204. Cited in page 36.

BURNS, E.; KITAIN, R. *JavaServer™ Faces Specification.* : Version, 2006. Cited in page 117.

CHENG, B. H.; ATLEE, J. M. Research directions in requirements engineering. In: IEEE COMPUTER SOCIETY. *2007 Future of Software Engineering.* 2007. p. 285–303. Cited in page 18.

CHILLAREGE, R. Orthogonal defect classification. *Handbook of Software Reliability Engineering*, IEEE CS Press, p. 359–399, 1996. Cited 2 times in pages 79 and 84.

CLELAND-HUANG, J. et al. Best practices for automated traceability. *Computer*, IEEE, v. 40, n. 6, 2007. Cited in page 34.

CMMI Institute. CMMI® for Development, Version 2.0, Improving processes for developing better products and services. 2018. Cited 5 times in pages 18, 33, 61, 64, and 110.

DAHLSTEDT, Å. G.; PERSSON, A. Requirements interdependencies: state of the art and future challenges. In: *Engineering and managing software requirements.* : Springer, 2005. p. 95–116. Cited in page 36.

DALL'OGLIO, P.; SILVA, J.; PINTO, S. Um modelo de rastreabilidade com suporte ao gerenciamento de mudanças e análise de impacto. In: *Workshop de Engenharia de Requisitos (WER 2010)*. 2010. Cited in page 111.

DALPIAZ, F. et al. Runtime goal models. In: *Proc. of the IEEE 7th International Conference on Research Challenges in Information Science*. Paris, France: IEEE, 2013. p. 1–11. ISBN 978-1-4673-2914-9. Cited in page 99.

DARDENNE, A.; LAMSWEERDE, A. V.; FICKAS, S. Goal-directed requirements acquisition. *Science of computer programming*, Elsevier, v. 20, n. 1-2, p. 3–50, 1993. Cited 2 times in pages 65 and 71.

DEFENSE, P. M. Software problem led to system failure at dhahran, saudi arabia. *US GAO Reports, report no. GAO/IMTEC-92-26*, 1992. Cited in page 90.

DELFRATE, L. Preliminaries to a formal ontology of failure of engineering artifacts. In: *FOIS*. 2012. p. 117–130. Cited 2 times in pages 76 and 80.

DUARTE, B. B. et al. Towards an ontology of software defects, errors and failures. In: SPRINGER. *International Conference on Conceptual Modeling*. 2018. p. 349–362. Cited in page 75.

DUARTE, B. B. et al. An ontological analysis of software system anomalies and their associated risks. *Data & Knowledge Engineering*, Elsevier, v. 134, p. 101892, 2021. Cited in page 60.

DUARTE, B. B. et al. Towards an ontology of requirements at runtime. In: IOS PRESS. *Proc. of the 9th International Conference on Formal Ontology in Information Systems (FOIS 2016)*. 2016. v. 283, p. 255. Cited 2 times in pages 49 and 53.

DUARTE, B. B. et al. Ontological foundations for software requirements with a focus on requirements at runtime. *Applied Ontology*, p. 73–105, 2018. Cited 14 times in pages 10, 24, 45, 49, 53, 55, 56, 57, 58, 67, 68, 69, 132, and 136.

DUARTE, G. C. *Uma Ferramenta Web Baseada em Ontologias Operacionais para Visualização da Relação entre Requisitos*. 79 p. Monografia (Monography) — Universidade Federal do Espirito Santo, Av. Fernando Ferrari, 514, Vitória, ES, 2021. Cited in page 117.

ESPINOZA, A.; GARBAJOSA, J. A study to support agile methods more effectively through traceability. *Innovations in Systems and Software Engineering*, Springer, v. 7, n. 1, p. 53–69, 2011. Cited 2 times in pages 35 and 97.

FALBO, R. d. A. SABiO: Systematic Approach for Building Ontologies. In: GUIZZARDI, G. et al. (Ed.). *Proc. of the Proceedings of the 1st Joint Workshop ONTO.COM / ODISE on Ontologies in Conceptual Modeling and Information Systems Engineering*. Rio de Janeiro, RJ, Brasil: CEUR, 2014. Cited 6 times in pages 9, 24, 26, 49, 50, and 94.

FALBO, R. d. A. et al. Organizing ontology design patterns as ontology pattern languages. In: SPRINGER. *Extended Semantic Web Conference*. 2013. p. 61–75. Cited 2 times in pages 9 and 43.

FALBO, R. d. A. et al. Ontology pattern languages. In: *Ontology Engineering with Ontology Design Patterns: Foundations and Applications.* : IOS Press, 2016. Cited 2 times in pages 52 and 84.

FALBO, R. d. A.; BERTOLLO, G. A software process ontology as a common vocabulary about software processes. *International Journal of Business Process Integration and Management*, Inderscience Publishers, v. 4, n. 4, p. 239–250, 2009. Cited 2 times in pages 54 and 67.

FALBO, R. de A.; BRAGA, C. E. C.; MACHADO, B. N. Semantic documentation in requirements engineering. In: *CIbSE*. 2014. p. 506–519. Cited in page 126.

FERNÁNDEZ-LÓPEZ, M.; GÓMEZ-PÉREZ, A.; JURISTO, N. Methontology: from ontological art towards ontological engineering. American Asociation for Artificial Intelligence, 1997. Cited in page 49.

FONSECA, C. M. et al. Relations in ontology-driven conceptual modeling. In: SPRINGER. *International Conference on Conceptual Modeling*. 2019. p. 28–42. Cited 2 times in pages 65 and 88.

FRICKER, S. A.; SCHNEIDER, K. (Ed.). *Requirements Engineering: Foundation for Software Quality - 21st International Working Conference, REFSQ 2015, Essen, Germany, March 23-26, 2015. Proceedings*, v. 9013 de *Lecture Notes in Computer Science*, (Lecture Notes in Computer Science, v. 9013). : Springer, 2015. ISBN 978-3-319-16100-6. Cited in page 79.

GLIMM, B. et al. Hermit: an owl 2 reasoner. *Journal of Automated Reasoning*, Springer, v. 53, n. 3, p. 245–269, 2014. Cited in page 136.

GOKNIL, A.; KURTEV, I.; BERG, K. V. D. Generation and validation of traces between requirements and architecture based on formal trace semantics. *Journal of Systems and Software*, Elsevier, v. 88, p. 112–137, 2014. Cited 3 times in pages 11, 122, and 124.

GOTEL, O. C.; FINKELSTEIN, C. An analysis of the requirements traceability problem. In: IEEE. *Requirements Engineering, 1994., Proceedings of the First International Conference on*. 1994. p. 94–101. Cited 4 times in pages 18, 19, 32, and 33.

GRIFFO, C. et al. From an ontology of service contracts to contract modeling in enterprise architecture. In: IEEE. *2017 IEEE 21st International Enterprise Distributed Object Computing Conference (EDOC)*. 2017. p. 40–49. Cited in page 135.

GRIFFO, C. et al. Service contract modeling in enterprise architecture: An ontology-based approach. *Information Systems*, Elsevier, p. 101454, 2019. Cited in page 135.

GRÜNINGER, M.; FOX, M. Methodology for the Design and Evaluation of Ontologies. In: *Workshop on Basic Ontological Issues in Knowledge Sharing, IJCAI'95*. 1995. Cited in page 50.

GUARINO, N. *Formal ontology in information systems: Proceedings of the first international conference (FOIS'98), June 6-8, Trento, Italy.* : IOS press, 1998. v. 46. Cited 5 times in pages 9, 20, 39, 41, and 42.

GUIZZARDI, G. *Ontological Foundations for Structural Conceptual Models*. Tese (PhD Thesis) — University of Twente, The Netherlands, 2005. Cited 10 times in pages 9, 25, 26, 40, 41, 42, 43, 48, 75, and 83.

GUIZZARDI, G. On ontology, ontologies, conceptualizations, modeling languages, and (meta) models. *Frontiers in artificial intelligence and applications*, IOS Press, v. 155, p. 18, 2007. Cited 8 times in pages 9, 20, 25, 26, 39, 40, 43, and 60.

GUIZZARDI, G.; FALBO, R. de A.; GUIZZARDI, R. S. Grounding Software Domain Ontologies in the Unified Foundational Ontology (UFO): The case of the ODE Software Process Ontology. In: *Proc. of the 11th Iberoamerican Conference on Software Engineering (CIbSE)*. 2008. p. 127–140. Cited 3 times in pages 25, 31, and 44.

GUIZZARDI, G.; SALES, T. P. Detection, simulation and elimination of semantic anti-patterns in ontology-driven conceptual models. In: *Conceptual Modeling*. : Springer, 2014. p. 363–376. Cited in page 134.

GUIZZARDI, G. et al. Towards Ontological Foundations for the Conceptual Modeling of Events. In: *Proc. of the 32th International Conference on Conceptual Modeling*. : Springer, 2013. p. 327–341. Cited 11 times in pages 25, 26, 42, 43, 44, 48, 51, 77, 83, 84, and 132.

GUIZZARDI, R. S. S. *Agent-oriented Constructivist Knowledge Management*. Tese (PhD Thesis) — University of Twente, The Netherlands, 2006. Cited 2 times in pages 43 and 44.

GUIZZARDI, R. S. S. et al. Ontological distinctions between means-end and contribution links in the i* framework. In: *Proceedings of the 32th International Conference on Conceptual Modeling, ER*. Hong-Kong, China: , 2013. p. 463–470. Cited 2 times in pages 76 and 78.

GUIZZARDI, R. S. S. et al. An Ontological Interpretation of Non-Functional Requirements. In: GARBACZ, P.; KUTZ, O. (Ed.). *Proc. of the 8th International Conference on Formal Ontology in Information Systems*. Rio de Janeiro, RJ, Brasil: IOS Press, 2014. v. 267, p. 344–357. Cited in page 57.

GUNTER, C. A. et al. A reference model for requirements and specifications. *IEEE Software*, IEEE, v. 17, n. 3, p. 37–43, 2000. Cited 7 times in pages 9, 19, 28, 29, 30, 31, and 60.

HEVNER, A. R. A three cycle view of design science research. *Scandinavian journal of information systems*, v. 19, n. 2, p. 4, 2007. Cited 2 times in pages 9 and 23.

HITZLER, P.; KROTZSCH, M.; RUDOLPH, S. *Foundations of semantic web technologies*. : CRC press, 2009. Cited in page 46.

HOGGANVIK, I.; STØLEN, K. A graphical approach to risk identification, motivated by empirical investigations. In: SPRINGER. *International Conference on Model Driven Engineering Languages and Systems*. 2006. p. 574–588. Cited in page 78.

IEEE. *IEEE 1044: Standard Classification for Software Anomalies*. 2009. Cited 3 times in pages 75, 76, and 78.

IEEE. *IEEE 1012: Standard for System, Software, and Hardware Verification and Validation*. 2016. Cited 2 times in pages 75 and 76.

IRMAK, N. Software is an abstract artifact. *Grazer Philosophische Studien*, Rodopi, v. 86, n. 1, p. 55–72, 2013. Cited in page 17.

ISO, I. *ISO/IEC International Standard - Systems and software engineering – System life cycle process*. 2015. 1-118 p. Cited in page 60.

ISO, I. *ISO/IEC International Standard - Systems and software engineering – Software life cycle process*. 2017. 1-157 p. Cited 7 times in pages 20, 24, 27, 60, 61, 62, and 110.

ISO, I. *ISO/IEC International Standard - Systems and software engineering – Requirements Engineering*. 2018. 1-104 p. Cited 14 times in pages 10, 18, 20, 24, 27, 28, 33, 60, 61, 62, 63, 64, 67, and 69.

ISO, IEC. *ISO/IEC International Standard - Software and systems engineering – Methods and tools for variability traceability in software and systems product line*. 2017. 35 p. Cited 4 times in pages 32, 33, 38, and 110.

ISO, IEC. *ISO/IEC International Standard - Systems and software engineering – Vocabulary*. 2017. 1-541 p. Cited 3 times in pages 17, 66, and 76.

JACKSON, M.; ZAVE, P. Deriving specifications from requirements: an example. In: *Proceedings of the 17th international conference on Software engineering*. 1995. p. 15–24. Cited 4 times in pages 28, 29, 31, and 60.

JURETA, I.; MYLOPOULOS, J.; FAULKNER, S. Revisiting the Core Ontology and Problem in Requirements Engineering. In: *Proc. of the 16th IEEE International Requirements Engineering Conference*. : IEEE, 2008. p. 71–80. Cited in page 71.

JURETA, I. J.; MYLOPOULOS, J.; FAULKNER, S. A core ontology for requirements. *Applied Ontology*, IOS Press, v. 4, n. 3-4, p. 169–244, 2009. Cited in page 71.

KANNENBERG, A.; SAIEDIAN, H. Why software requirements traceability remains a challenge. *CrossTalk The Journal of Defense Software Engineering*, v. 22, n. 5, p. 14–19, 2009. Cited 4 times in pages 9, 18, 33, and 34.

KITAMURA, Y.; MIZOGUCHI, R. An ontological analysis of fault process and category of faults. In: *Proceedings of tenth international workshop on principles of diagnosis (DX-99)*. 1999. p. 118–128. Cited 2 times in pages 10 and 81.

KITAMURA, Y.; MIZOGUCHI, R. An ontological analysis of fault process and category of faults. In: *Proceedings of tenth international workshop on principles of diagnosis (DX-99)*. 1999. p. 118–128. Cited in page 81.

KITCHENHAM, B. A.; BUDGEN, D.; BRERETON, O. P. The value of mapping studies: A participantobserver case study. In: *Proc. of the 14th International Conference on Evaluation and Assessment in Software Engineering*. Swinton, UK, UK: British Computer Society, 2010. (EASE'10), p. 25–33. Cited in page 36.

KITCHENHAM, B. A.; CHARTERS, S. *Guidelines for performing Systematic Literature Reviews in Software Engineering*. 2007. Cited 2 times in pages 24 and 36.

LAMSWEERDE, A. V. Requirements engineering in the year 00: a research perspective. In: ACM. *Proceedings of the 22nd international conference on Software engineering*. 2000. p. 5–19. Cited 2 times in pages 27 and 28.

LAMSWEERDE, A. V. Goal-oriented requirements engineering: A guided tour. In: IEEE. *Proceedings fifth ieee international symposium on requirements engineering*. 2001. p. 249–262. Cited 2 times in pages 65 and 71.

LAMSWEERDE, A. V. From worlds to machines. *A Tribute to Michael Jackson*, Lulu Press, 2009. Cited 4 times in pages 9, 27, 30, and 31.

LEHMAN, M. M. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, IEEE, v. 68, n. 9, p. 1060–1076, 1980. Cited 2 times in pages 73 and 136.

LEHMAN, M. M. Uncertainty in computer application and its control through the engineering of software. *Journal of Software Maintenance: Research and Practice*, Wiley Online Library, v. 1, n. 1, p. 3–27, 1989. Cited in page 29.

LEHMAN, M. M. Laws of software evolution revisited. In: SPRINGER. *European Workshop on Software Process Technology*. 1996. p. 108–124. Cited 2 times in pages 29 and 136.

LEHMAN, M. M.; RAMIL, J. F. Rules and tools for software evolution planning and management. *Annals of software engineering*, Springer, v. 11, n. 1, p. 15–44, 2001. Cited in page 28.

LEVESON, N. G.; TURNER, C. S. An investigation of the Therac-25 accidents. *Computer*, v. 26, n. 7, p. 18–41, July 1993. ISSN 0018-9162. Cited in page 88.

LINDVALL, M.; SANDAHL, K. Practical implications of traceability. *Software: Practice and Experience*, Wiley Online Library, v. 26, n. 10, p. 1161–1180, 1996. Cited in page 33.

MACHADO, B. N.; ARANTES, L. de O.; FALBO, R. de A. Using semantic annotations for supporting requirements evolution. In: *SEKE*. 2011. p. 185–190. Cited in page 126.

MÄDER, P.; GOTEL, O. Towards automated traceability maintenance. *Journal of Systems and Software*, Elsevier, v. 85, n. 10, p. 2205–2227, 2012. Cited in page 19.

MASOLO, C. et al. Dolce: a descriptive ontology for linguistic and cognitive engineering. *WonderWeb Project, Deliverable D17 v2*, v. 1, 2003. Cited 3 times in pages 31, 42, and 73.

MCBRIDE, B. Jena: A semantic web toolkit. *IEEE Internet computing*, IEEE, v. 6, n. 6, p. 55–59, 2002. Cited in page 117.

MCCONNELL, S. Managing technical debt. *Construx Software Builders, Inc*, p. 1–14, 2008. Cited in page 137.

MOSTOW, J. A problem-solver for making advice operational. In: *AAAI*. 1983. p. 279–283. Cited in page 65.

NAIR, S.; VARA, J. L. D. L.; SEN, S. A review of traceability research at the requirements engineering conference re@ 21. In: IEEE. *2013 21st IEEE International Requirements Engineering Conference (RE)*. 2013. p. 222–229. Cited 3 times in pages 18, 24, and 36.

NARDI, J. C.; FALBO, R. d. A. Evolving a Software Requirements Ontology. In: *Proc. of the 34th Conferencia Latinoamericana de Informatica (CLEI 08)*. Santa Fe, Argentina: , 2008. Cited in page 126.

NARDI, J. C. et al. A commitment-based reference ontology for services. *Information systems*, Elsevier, v. 54, p. 263–288, 2015. Cited in page 44.

NARDI, J. C. et al. A commitment-based reference ontology for services. *Information systems*, Elsevier, v. 54, p. 263–288, 2015. Cited in page 135.

NEGRI, P. P. et al. Towards an ontology of goal-oriented requirements. In: *Proc. of the 20th Workshop on Requirements Engineering (WER) at the 20th Ibero-American Conference on Software Engineering (CIbSE)*. 2017. Cited in page 71.

NOY, N. F. et al. Protégé-2000: an open-source ontology-development and knowledge-acquisition environment. In: *AMIA... Annual Symposium proceedings. AMIA Symposium*. 2003. p. 953–953. Cited 2 times in pages 92 and 101.

NOY, N. F. et al. Creating semantic web contents with protege-2000. *IEEE intelligent systems*, IEEE, v. 16, n. 2, p. 60–71, 2001. Cited 3 times in pages 92, 101, and 128.

PANDANABOYANA, S. et al. Requirements tracing on target (retro) enhanced with an automated thesaurus builder: An empirical study. In: *2013 7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)*. 2013. p. 61–67. Cited in page 32.

PRIMEFACES. *PrimeFaces for JSF*. 2021. Disponível em: <https://www.primefaces.org/>. Cited in page 117.

RAMESH, B. Factors influencing requirements traceability practice. *Communications of the ACM*, ACM New York, NY, USA, v. 41, n. 12, p. 37–44, 1998. Cited 4 times in pages 9, 18, 34, and 35.

RAMESH, B.; JARKE, M. Toward reference models for requirements traceability. *IEEE transactions on software engineering*, IEEE, n. 1, p. 58–93, 2001. Cited 8 times in pages 11, 19, 35, 36, 96, 120, 122, and 123.

RAMESH, B. et al. Implementing requirements traceability: a case study. In: IEEE. *Proceedings of 1995 IEEE International Symposium on Requirements Engineering (RE'95)*. 1995. p. 89–95. Cited 2 times in pages 32 and 116.

RAMESH, B. et al. Requirements traceability: Theory and practice. *Annals of software engineering*, Springer, v. 3, n. 1, p. 397–415, 1997. Cited in page 32.

REGAN, G. et al. The barriers to traceability and their potential solutions: Towards a reference framework. In: IEEE. *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*. 2012. p. 319–322. Cited in page 33.

RUY, F. B. et al. Seon: A software engineering ontology network. In: SPRINGER. *Knowledge Engineering and Knowledge Management: 20th International Conference, EKAW 2016, Bologna, Italy, November 19-23, 2016, Proceedings 20*. 2016. p. 527–542. Cited 7 times in pages 9, 22, 26, 51, 52, 53, and 84.

SALES, T. P. et al. The common ontology of value and risk. In: SPRINGER. *International Conference on Conceptual Modeling*. 2018. p. 121–135. Cited 9 times in pages 10, 17, 78, 82, 83, 84, 87, 88, and 135.

SALES, T. P.; GUIZZARDI, G. Ontological anti-patterns: Empirically uncovered error-prone structures in ontology-driven conceptual models. *Data & Knowledge Engineering*, Elsevier, v. 99, p. 72–104, 2015. Cited in page 134.

SERRANO, M.; LEITE, J. C. S. do P. A rich traceability model for social interactions. In: *Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering*. 2011. p. 63–66. Cited in page 37.

SINGH, S.; KARWAYUN, R. A comparative study of inference engines. In: IEEE. *2010 seventh international conference on information technology: New generations*. 2010. p. 53–57. Cited in page 136.

SIRIN, E. et al. Pellet: A practical owl-dl reasoner. *Journal of Web Semantics*, Elsevier, v. 5, n. 2, p. 51–53, 2007. Cited in page 136.

SOMMERVILLE, I. *Software engineering 10th edition.* : Pearson Education, 2016. ISBN 1-292-09613-6. Cited 2 times in pages 27 and 28.

SOUZA, E.; FALBO, R.; VIJAYKUMAR, N. L. Using ontology patterns for building a reference software testing ontology. In: IEEE. *2013 17th IEEE International Enterprise Distributed Object Computing Conference Workshops*. 2013. p. 21–30. Cited 4 times in pages 22, 53, 84, and 113.

SOUZA, É. F. de; FALBO, R. d. A.; VIJAYKUMAR, N. L. ROoST: Reference Ontology on Software Testing. *Applied Ontology*, IOS Press, p. 1–32, 2017. ISSN 18758533. Cited 2 times in pages 49 and 84.

SOUZA, V. E. S. et al. Requirements-driven software evolution. *Computer Science - Research and Development*, Springer, v. 28, n. 4, p. 311–329, nov 2013. Cited in page 45.

SUÁREZ-FIGUEROA, M. C.; GÓMEZ-PÉREZ, A.; FERNÁNDEZ-LÓPEZ, M. The neon methodology for ontology engineering. In: *Ontology engineering in a networked world.* : Springer, 2012. p. 9–34. Cited in page 125.

SUÁREZ-FIGUEROA, M. C. et al. *Ontology engineering in a networked world.* : Springer Science & Business Media, 2012. Cited 3 times in pages 49, 51, and 136.

SURE, Y.; STUDER, R. On-to-knowledge methodology-final version. Citeseer, 2002. Cited in page 49.

TALLABACI, G.; SOUZA, V. E. S. Engineering adaptation with zanshin: an experience report. In: IEEE. *2013 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2013. p. 93–102. Cited 2 times in pages 99 and 101.

TSARKOV, D.; HORROCKS, I. Fact++ description logic reasoner: System description. In: SPRINGER. *International joint conference on automated reasoning*. 2006. p. 292–297. Cited in page 136.

TUFAIL, H. et al. A systematic review of requirement traceability techniques and tools. In: IEEE. *System Reliability and Safety (ICSRS), 2017 2nd International Conference on*. 2017. p. 450–454. Cited 4 times in pages 19, 24, 33, and 36.

W3C. *Resource Description Framework*. 2004. <https://www.w3.org/TR/rdf-concepts/>. Accessed: 2020-07-05. Cited 4 times in pages 9, 46, 47, and 125.

W3C. *SPARQL query Language for RDF version 1.1*. 2013. <https://www.w3.org/TR/sparql11-query/>. Accessed: 2020-07-05. Cited 3 times in pages 26, 47, and 128.

W3C. *RDF Schema*. 2014. <https://www.w3.org/TR/rdf-schema/>. Accessed: 2020-07-05. Cited 2 times in pages 46 and 125.

W3C. *Terse RDF Triple Language*. 2014. <https://www.w3.org/TR/turtle/>. Accessed: 2020-07-05. Cited 4 times in pages 46, 94, 128, and 129.

WANG, X. *Towards an Ontology of Software*. Tese (PhD Thesis) — University of Trento, 2016. Cited in page 21.

WANG, X. et al. Software as a social artifact: a management and evolution perspective. In: *International Conference on Conceptual Modeling*. : Springer, 2014. p. 321–334. Cited in page 17.

WANG, X. et al. Towards an Ontology of Software: a Requirements Engineering Perspective. In: *Proceedings of the 8th International Conference on Formal Ontology in Information Systems*. Rio de Janeiro, RJ, Brasil: IOS Press, 2014. v. 267, p. 317–329. Cited 9 times in pages 9, 17, 20, 31, 32, 56, 67, 68, and 69.

WANG, X. et al. How software changes the world: The role of assumptions. In: *Tenth IEEE International Conference on Research Challenges in Information Science, RCIS*. Grenoble, France: , 2016. p. 1–12. Cited 10 times in pages 10, 20, 31, 57, 58, 64, 68, 79, 136, and 137.

WANG, Y. et al. Monitoring and diagnosing software requirements. *Automated Software Engineering*, Springer, v. 16, n. 1, p. 3, 2009. Cited 2 times in pages 99 and 101.

WIEGERS, K.; BEATTY, J. *Software requirements*. : Pearson Education, 2013. Cited in page 64.

WIERINGA, R. J. *Design science methodology for information systems and software engineering*. : Springer, 2014. Cited in page 23.

YU, E. S. Towards modelling and reasoning support for early-phase requirements engineering. In: IEEE. *Requirements Engineering, 1997., Proceedings of the Third IEEE International Symposium on*. 1997. p. 226–235. Cited in page 71.

YU, E. S.-K. *Modelling Strategic Relationships for Process Reengineering*. Tese (Doutorado) — University of Toronto, 1995. Cited in page 71.

ZAVE, P. Classification of research efforts in requirements engineering. In: IEEE. *Proceedings of 1995 IEEE International Symposium on Requirements Engineering (RE'95)*. 1995. p. 214–216. Cited 2 times in pages 19 and 27.

ZAVE, P.; JACKSON, M. Four Dark Corners of Requirements Engineering. *ACM Transactions on Software Engineering and Methodology*, v. 6, n. 1, p. 1–30, jan 1997. Cited 8 times in pages 19, 28, 29, 60, 63, 66, 70, and 132.

ZHANG, H. et al. Investigating dependencies in software requirements for change propagation analysis. *Information and Software Technology*, Elsevier, v. 56, n. 1, p. 40–53, 2014. Cited in page 36.