An Ontological Analysis of Software System Anomalies and their associated Risks

Bruno Borlini Duarte[a,*], Ricardo de Almeida Falbo[a], Giancarlo Guizzardi[b,c], Renata Guizzardi[c], Vítor E. Silva Souza[a]

[a] *Universidade Federal do Espírito Santo, Vitória-ES, Brazil*
[b] *Free University of Bozen-Bolzano, Bolzano, Italy*
[c] *University of Twente, The Netherlands*

Abstract

Software systems have an increasing value in our lives, as our society relies on them for the numerous services they provide. However, as our need for larger and more complex software systems grows, the risks involved in their operation also grows, with possible consequences in terms of significant material and social losses. The rational management of software defects and possible failures is a fundamental requirement for a mature software industry. Standards, professional guides and capability models directly emphasize how important it is for an organization to know and to have a well-established history of failures, errors and defects as they occur in software activities. The problem is that each of these reference models employs its own vocabulary to deal with these phenomena, which can lead to a deficiency in the understanding of these notions by software engineers, causing potential interoperability problems between supporting tools, and, consequently, a poorer adoption of these standards and tools in practice. In this paper, we address this problem of the lack of a consensual conceptualization in this area by proposing two reference conceptual models: an Ontology of Software Defects, Errors and Failures (OSDEF), which takes into account an ecosystem of software artifacts, and a Reference Ontology of Software Systems (ROSS), which characterizes software systems and related artifacts at

*Corresponding author
Email address: brunoborlini@gmail.com (Bruno Borlini Duarte)

different levels of abstraction. Moreover, we use OSDEF and ROSS to perform an ontological analysis of the impact of defects, errors and failures of software systems from a risk analysis perspective. To do that, we employee an existing core ontology, namely, the Common Ontology of Value and Risk (COVR). The ontologies presented here are grounded on the Unified Foundational Ontology (UFO) and based on well-known and widely-accepted standards, professional and scientific guides and capability models. We demonstrate how this approach can suitably promote conceptual clarification and terminological harmonization in this area.

Keywords: Software Defects, Errors and Failures, Ontological Foundations of Software Systems, Conceptual Modeling, Methods and Methodologies, Software System Risk, Unified Foundational Ontology (UFO)

---

1. Introduction

Software plays an essential role in modern society and it has become indispensable in many contexts of our lives, such as social, business, and personal contexts. This essential role motivates a number of research initiatives aimed at understanding the nature of software, and its relation to us. A shared conception in those initiatives is that software is a complex (social) artifact [1, 2, 3]. This notion comes from the fact that a modern software system can be understood as the combination of a series of interacting elements, specifically organized to provide a set of functionalities to fulfill particular human purposes [4, 5]. Moreover, software is constantly growing, not only in simple measures such as the number of lines of code, but also according to other factors, like complexity, criticality and degree of heterogeneity [6]. This makes it harder and more costly to maintain and evolve software, and this may be the starting point for many problems in the software life-cycle.

Besides its importance in our society, software is special also because it is capable of existing through time, being replicated millions of times and having dozens of different versions while still maintaining its identity [2]. A classic

example of these intrinsic 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, susceptible to failures, defects and faults that can range from having a small impact to being critical, thus, potentially causing significant material and social losses. Concepts such as *problem*, *anomaly*, *bug* and *glitch* are usually treated indistinctly, while potentially having different ontological semantics. This informal use, as common and practical as it may be in our daily conversations, can be the source of ambiguity and false-agreement problems, since the concept *anomaly* is frequently overloaded, thus, referring to entities with distinct ontological natures. In a more formal environment, this construct overload may lead to communication problems and losses. Because of that, and as defended in scientific literature, international standards and maturity models, it is important to have a precise way of classifying different types of software anomalies.

For example, the Guide to Software Engineering Body of Knowledge (SWEBoK) [5] emphasizes the need of a consensus about anomaly characterization, and discusses how a well-founded classification could be used in audits and product reviews. Moreover, the CMMI [7] model advocates that organizations should create or reuse some form of classification method for defects and failures. It also suggests the use of a defect density index for many work products that are part of the software development process.

A proper classification scheme can enable the development of different types of anomaly profiles that can be produced as an indicator of product quality. Also, systematically classifying software anomalies that may occur at design-time or runtime is a rich source of data that can be used to improve processes and avoid the occurrence of anomalies in future projects [8]. Finally, defects, faults and failures have a negative impact on important aspects of software, such as reliability, efficiency, overall cost and, ultimately, lifespan. Hence, a better understanding of the ontological nature of these concepts and how they relate

to other software artifacts (e.g. requirements, change requests, reports and tests cases) can improve the way an organization deals with these issues, ultimately reducing costs with activities such as configuration management and software maintenance.

Although there are some proposals for classifying different *terms* for software anomalies, there is no reference model or theory that elaborates on the *nature* of different software anomalies. In other words, to the best of our knowledge, there is no proper reference ontology [9] focused on representing software defects, errors and failures. In order to address this gap, we propose a reference *Ontology of Software Defects, Errors and Failures (OSDEF*). This ontology takes into account different types of anomalies that may exist in software-related artifacts and that are recurrently mentioned in the set of the most relevant standards in the area. Furthermore, we recognize the importance of analyzing such anomalies in terms of the *risk* their presence ensues to software systems and, in particular, to these systems as bearers of *value* to some agent. In order to do that, we needed to elaborate on the relation between software systems and other software artifacts at different levels of abstraction. For this, we developed a *Reference Ontology on Software Systems (ROSS)*. OSDEF and ROSS are then analyzed from this *risk analysis* perspective by leveraging on the *Common Ontology of Value and Risk (COVR)* [10].

OSDEF and ROSS are developed following the process defined by the Systematic Approach for Building Ontologies (SABiO) [11] and grounded on the Unified Foundational Ontology (UFO) [12, 13], including UFO's Ontology of Events (UFO-B) [14, 15]. In order to elicit consensual information about the domain, we analyze relevant standards, guides and capability models such as CMMI [7], SWEBoK [5], IEEE Standard Classification for Software Anomalies [8], IEEE Standard for System, Software, and Hardware Verification and Validation [16], as well as complementary current Software Engineering literature. Finally, the ontologies are evaluated by verification and validation techniques recommended by SABiO.

This paper is an extended version of [17]. In this version, we present as orig-

inal contributions: an extension of the original OSDEF ontology to incorporate the notion of run-time vulnerabilities, which inhere in loaded program copies as opposed to programs; the Reference Ontology on Software Systems (ROSS); an ontological analysis of three famous cases of software failures. The latter is done by instantiating them with the concepts from OSDEF, ROSS and the *risk analysis* perspective provided by COVR.

The remainder of this paper is structured as follows. Section 2 briefly presents the foundations used for developing the ontologies proposed in this work. In that section, we briefly introduce the reader to: the SABiO method, the foundational ontology UFO, and three core ontologies, namely, COVR, but also the *Software Process Ontology (SPO)* [18] and the *Software Ontology (SwO)* [19]). The last two ontologies have been reused to create OSDEF and ROSS. OSDEF and ROSS are presented in Sections 3 and 4, respectively. Section 5 evaluates the proposed ontologies That section also presents the instantiation of ROSS and OSDEF from a risk analysis perspective by reusing them in combination with COVR. Section 6 discusses related work. Finally, Section 7 concludes the paper by presenting some final considerations.

2. Ontological Foundations

This section presents the ontological foundations used by the reference models proposed in this article. Subsection 2.1 presents a fragment of UFO that is germane to purposes of this work. Subsection 2.2 presents SPO and SwO, ontologies focused on the software domain that were reused. Subsection 2.3, presents COVR, the Common Ontology of Value and Risk, a domain ontology that allows us to analyze the impact of software defects, errors and failures under a value and risk perspective. Finally, Subsection 2.4 presents SABiO, the ontology engineering method adopted for the development of this work.

2.1. The Unified Foundational Ontology (UFO)

We ground the ontologies presented in this article on UFO [14, 12, 9]. This choice is motivated by the following: (i) UFO's foundational categories address

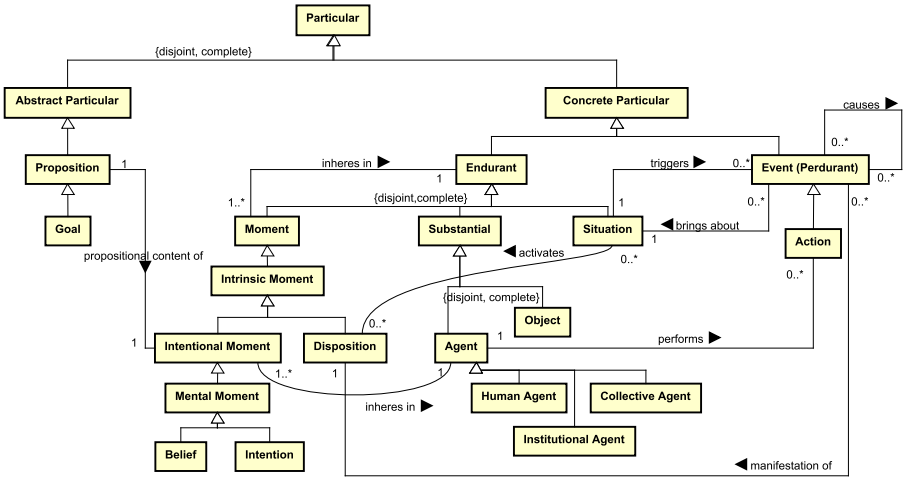

Figure 1: Fragment of UFO showing Events, Agents and Objects.

many essential aspects for the conceptual modeling of the intended domain, including concepts like events, dispositions and situations; (ii) UFO has a positive track record in being able to successfully address different phenomena in Software Engineering [20, 21, 22]; (iii) a recent study shows that UFO is among the most used Foundational Ontologies in Conceptual Modeling and the one with a fastest growing rate of adoption [23]. By adopting a foundational ontology that is frequently used, we increase the reusability of this work, also facilitating its future integration in *ontology networks* in software engineering [24].

UFO is composed of three main parts: UFO-A, an ontology of endurants [12]; UFO-B, an ontology of perdurants/events [14]; and UFO-C, an ontology of social entities (both endurants and perdurants) built on top of UFO-A and UFO-B [21]. Figure 1 presents a fragment of UFO that contains the categories that are essential for the purpose of this article. UFO has been formally characterized in [12, 25, 26, 15]. Its representation as UML diagrams is used here for illustration/communication purposes only.

Particulars are (unique) Individuals that can be abstract or concrete, depending on their nature. Endurants and Perdurants are Concrete Particulars, i.e., entities that exist in time and space possessing a unique identity. Endurants do not

have temporal parts, but are able to change in a qualitative manner while keeping their identity (e.g., a person). Perdurants (or Events, *occurrences*, *processes*), are composed by temporal parts (e.g., a trip): they exist in time, accumulating temporal parts and, unlike Endurants, they are immutable, i.e., cannot change any of their properties; cannot be different from what they are [14, 27]. Moreover, Events are transformations from a portion of reality to another, which means that when a Situation S triggers an Event E, E can *bring about* another Situation S'. Finally, Events can *cause* other Events. This causality relation is a strict partial order (irreflexive, asymmetric and transitive) relation [15].

Actions are Events that are *performed* by Agents (persons, organizations or teams) with the specific purpose of satisfying *intentions* of that Agent. However, if the agent's intentions are based on the wrong assumptions, they can lead to problems, i.e., they can bring about situations that do not satisfy (or that even dent) the goals (i.e., propositional content) of the intention that motivated that action. Moments (also called *aspects*, or *reified properties*) are existentially dependent entities. This means that they need to *inhere in* other *Concrete Particulars* in order to exist. For example, if a person (as an Agent) or a chair (Object) ceases to exist, their Moments (e.g., the Beliefs and Intentions of that person, the texture, a bump or a scratch on that chair) will also disappear. Dispositions are a special type of Moment that are only manifested in certain Situations and that can fail to be manifested. When Dispositions are *manifested*, they do so via the occurrence of an Event [14]. Examples of Dispositions are the capacity of a magnet to attract metallic material, or John's competence for playing guitar. Situations are complex Endurants that are constituted by possibly many Endurants (including other situations). Situations are portions of reality that can be comprehended as a whole. See [28, 14] for a deeper discussion about Events, Situations and Moments (including Dispositions).

2.2. The Software Process Ontology (SPO) and the Software Ontology (SwO)

For OSDEF and ROSS, we reuse the following concepts defined in the *Software Process Ontology (SPO)* [18]: Software Artifacts, which are objects in-

tentionally made to serve a given purpose in the context of a software project or organization; Stakeholders, which 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).

From the Software Ontology (SwO), which is depicted in Figure 2 we reuse the concept of Program [19].[1] A Program is an (Abstract) Artifact that is constituted by code but which is not identical to a code. In contrast, a Program owes its identity principle to a Program Specification, which the Program intends to implement. Besides, 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. A Program, when loaded inside a Machine, as a Loaded Program Copy, can be executed, a Program Copy Execution, in order to produce an expected result and to fulfill its given purpose. The Program Copy Execution is defined an Event that brings about Observable State result inside the Machine. Finally, Programs can be aggregated to constitute Software Systems.

2.3. Common Ontology of Value and Risk (COVR)

The Common Ontology of Value and Risk (COVR) [10] 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. COVR sees risk as intrinsically connected to the notion of value, in

[1] SwO reuses a fragment of SPO. The classes depicted in green in this figure are the SPO classes specialized by that ontology.

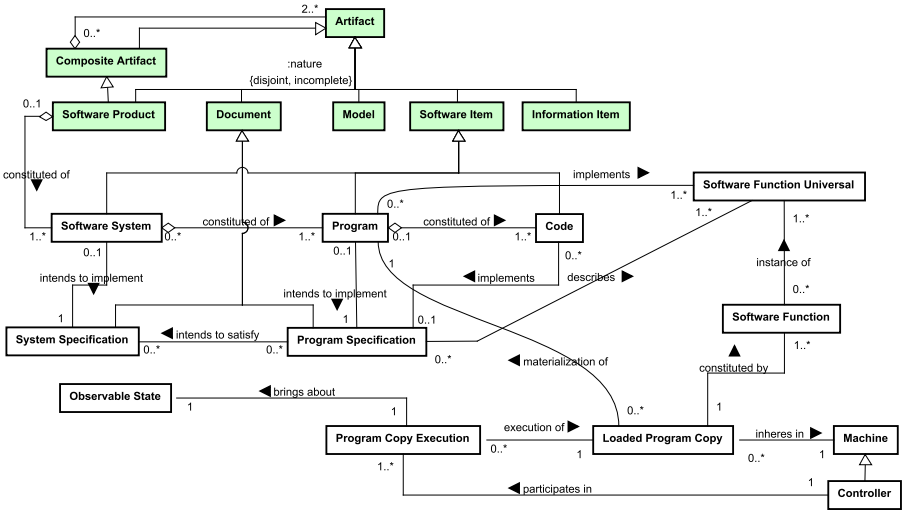

Figure 2: SwO Conceptual Model [19].

a way that *risk assessment* is seen as a particular case of *value ascription*. In other words, the authors conceptualize value and risk as two sides of the same coin, thus, sharing intrinsic properties such as goal dependency[2] and relativity.[3] Furthermore, the authors present and discuss different types of value and risk based on these intrinsic properties. Finally, it is important to mention that since COVR also is grounded in UFO, we can reuse it with a reduced effort.

Figures 3 and 4[4] show two fragments of COVR that constitute its expe-

[2]Roughly, 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. Moreover, risk is always the risk of the destruction of value.

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

[4]Figures 3 and 4 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 [12]. In this color code, light red classes represent Endurant types; blue classes represent Intrinsic Moment types; yellow classes represent Event types;

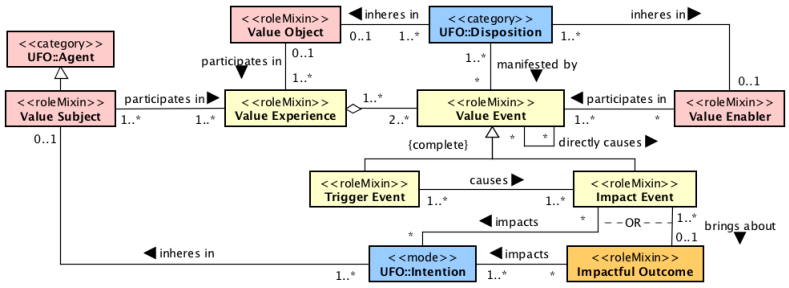

Figure 3: Fragment of COVR presenting the concept of Value Event and their relations [10].

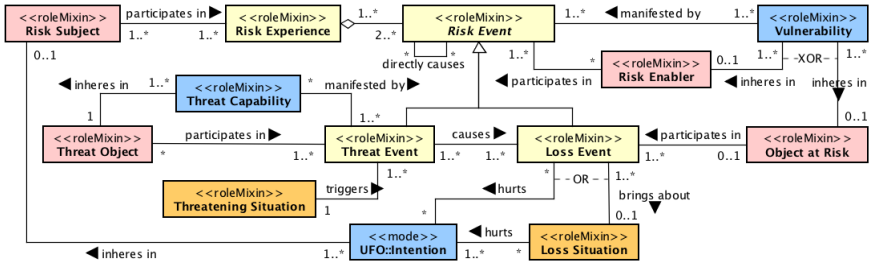

Figure 4: Fragment of COVR presenting the concept of Risk Event and their relations [10].

riential perspective. As these figures show, 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 focuses of these experiences can also participate in the Value and Risk Events as Value (Risk) Enablers. Finally, the central risk domain elements in COVR 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 [14].

finally, orange classes represent Situation types.

2.4. Systematic Approach to Building Ontologies (SABiO)

The *Systematic Approach to Building Ontologies (SABiO)* [11] is a method for building domain ontologies [9] that incorporates best practices from Software Engineering and Ontology Engineering. We chose SABiO as the ontology engineering method for the development of ROSS and OSDEF because it is focuses on the development of domain ontologies. Besides, SABiO explicitly recognizes the importance of using foundational ontologies in the ontology development process to improve the ontology quality. Additionally, SABiO has been successfully used in the development of several domain ontologies in Software Engineering, such as the Software Process Ontology (SPO) [20], the Software Ontology (SwO) [19] and the Reference Ontology of Software Testing (ROoST) [29], among other ontologies developed in the context of SEON, a Software Engineering Ontology Network [24]. SABiO also provides support and facilitates the reuse of those ontologies, which is very helpful, as we intend to reuse concepts of already established ontologies for the software domain.

Figure 5 depicts the five phases of SABiO and the support activities that comprise the method. SABiO's development process is composed of five main phases: (1) purpose identification and requirements elicitation; (2) ontology capture and formalization; (3) operational ontology design; (4) operational ontology implementation; and (5) testing. Furthermore, these phases are supported by well-known activities in the Requirements Engineering life-cycle, e.g., knowledge acquisition, reuse, configuration management, evaluation, and documentation.

Finally, since the main objective for this work is to produce domain reference ontologies as *conceptual models*, we focus on the first two phases of SABiO. The codification of the operational versions of OSDEF and ROSS shall be addressed in a complementary work. These operational ontologies, in turn, shall allow us to reason over requirements-data[5] produced during a software system life-cycle.

[5] E.g., software system requirements and other information items, such as test cases, faults and failures reports, and configuration management data produced during the development and the operation of a software system.

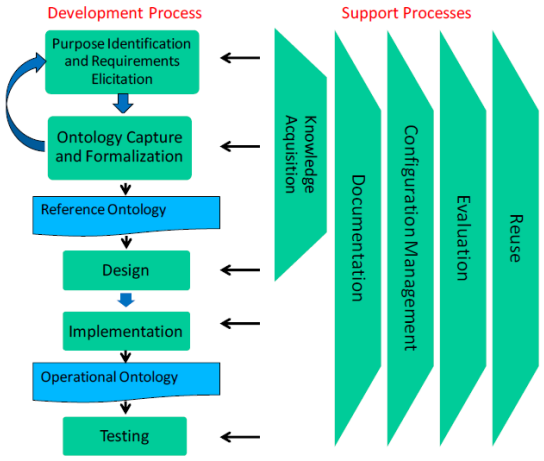

Figure 5: SABiO's process [11].

3. An Ontology of Software Defects, Errors and Failures

As previously mentioned, the term *anomaly* is commonly used to refer to a variety of notions of distinct ontological nature. To target this problem, OSDEF provides an ontological conceptualization of the different types of software anomalies that exist throughout the software life-cycle. To elaborate on these different types of anomalies, we formulate a set of *Competency Questions (CQ)*, i.e., questions that the ontology should be able to answer [30].

In a Requirements Engineering perspective, CQs are analogous to the functional requirements of the ontology [11]. Moreover, CQs help to refine the scope of the ontology and can also be used in the ontology verification process. For OSDEF, CQs were conceived and refined in a highly-interactive way, through analysis of the international standards mentioned in Section 1 and through several meetings with ontology experts. These CQs are listed below:

- **CQ1:** What is a failure?
- **CQ2:** What is a defect?
- **CQ3:** What is a fault?
- **CQ4:** What is an error?

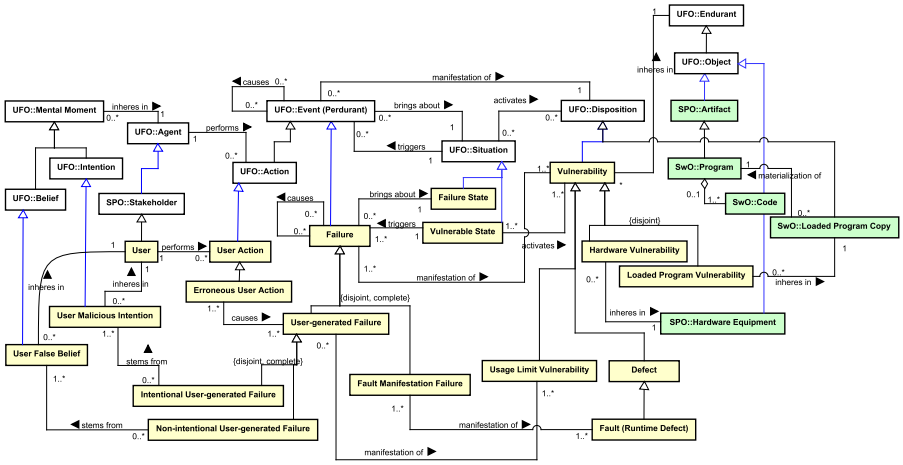

Figure 6: Conceptual Model of the Ontology of Software Defects, Errors and Failures.

- **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?

Figure 6 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 [8, 16, 4] and employed in general in the scientific literature [31], 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 [32]. As Events, Failures can cause other Failures in a chain of Events (e.g., a severe failure in a web server such as *Apache httpd* can make all of its hosted applications undergo subsequent failures). As defined in UFO [14], *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. 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..

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. This concepts are discussed in Section 4

Failures are further refined in two distinct subtypes: 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 [6] 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. The second one represents the types of entities in which those Dispositions inhere: a

[6] 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 [33, 10] of vulnerabilities as Dispositions that can be manifested by events that can hurt stakeholder's goals [32] or diminish something's value [10].

Hardware Vulnerability inheres in a Hardware Equipment, while a Loaded Program Vulnerability inheres in a Loaded Program Copy. 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 [7] a program-related Vulnerability can only be manifested if the program is loaded inside the memory of a Machine, namely, if it *inheres in* inside 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 [8] 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 [5] 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 cases, the "Dirty Copy on Write" [34] 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 life-cycle [35]. 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*

[7] The definition of Program as an Abstract Artifact is discussed in Section2.2

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 [36], software artifacts are designed taking into consideration *Domain Assumptions*. When a software artifact makes 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 [37], 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 a collective of users simultaneously accessing the system (e.g., as on Nike's website during the 2017 Black Friday).

4. A Reference Ontology of Software Systems

As mentioned in Section 1, Software Systems are complex abstract artifacts that can be composed by elements with distinct natures. In this section we present the Reference Ontology of Software Systems (ROSS). ROSS characterizes such software systems and the artifacts related to them at different levels of abstraction.

In their seminal work, Pamela Zave and Michael Jackson [38] discuss what they term "the four dark corners of Requirements Engineering (RE)". In doing so, they clarify certain aspects of nature of RE, and demonstrate the importance of certain information items that are often neglected in that discipline. In that paper, they proposed the following (by now, well-known) formula $S, A \vdash R$, which is meant to capture that in order to fulfill a set of requirements (R) are satisfied by a specification (S) associated with a set of domain assumptions (A). Later on [39], the formula was improved to take into account other software artifacts of relevance, such as the Machine (M), as the programming platform, and the Program (P), as the unity that is intended to implement the specification.

In a previous work [19], we adopted Zave and Jackson's contributions, together with the ontology of Software Artifacts proposed by Wang et al. [1, 2] to develop the Software Ontology (SwO) and the Reference Software Requirements Ontology (RSRO). SwO and RSRO are two reference ontologies created with the purpose of being reused for the development of a more specific ontology, the Runtime Requirements Ontology (RRO) [40, 19]. RRO, in its turn, aims at serving as a conceptual reference for the creation and the interoperability of requirements at runtime frameworks and methods. However, as prescribed in SABiO [11] (which was used in their development), SwO and RSRO were designed in a general manner, containing only the concepts necessary for supporting extensions like RRO. As such, they neither contemplate socio-technical aspects of software systems nor some of their aspects as multi-artifact entities [1].

However, these aspects are central for explaining how defects, errors and failures impact software systems from a risk analysis perspective, in particular,

due to the connection between (societal) risks and values. For example, they are needed to characterize the context in which software systems exists and operate as well as the impact caused by failure events (in the sense described in OSDEF). With this in mind, we developed the Reference Ontology of Software Systems (ROSS), a domain reference model and knowledge representation tool for software systems that reuses and complements these previous works. ROSS is based on widely accepted international standards, such as ISO 29148 [41], ISO 12207 [42] and SWEBoK [5] but also on Zave and Jackson's seminal work on the nature of software requirements [38, 39].

Once more, as prescribed by SABiO, the requirements for this ontology are expressed as a set of Competency Questions (CQs). The CQs for ROSS were produced by the same iterative process used for the CQs of OSDEF. In tandem with that process, the ontology itself evolved as a sequence of versions following the process of refining and answering these CQs. The latter are presented in the sequel:

- CQ1: What are Software Systems?
- CQ2: How are Software Systems composed?
- CQ3: What are the types of requirements that exist in the Software System domain?
- CQ4: How are these requirements related?
- CQ5: How are these requirements described?
- CQ6: What are the types of assumptions that are relevant in the Software System domain?
- CQ7: How are requirements related to assumptions in the Software System domain?
- CQ8: What are the constraints on the Software System domain? How they impact the Organization and their requirements?

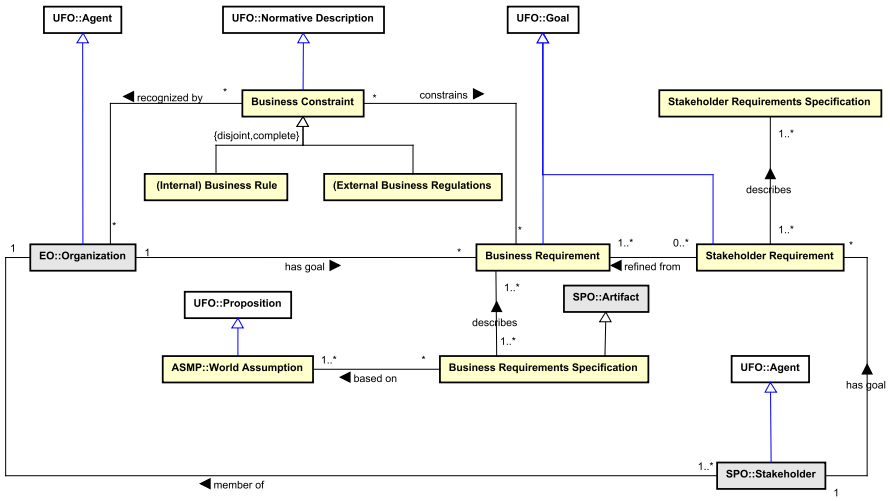

Figure 7: The Business Layer

To present the ontology, we adopted SABiO's guidelines on ontology modularization and divided ROSS in three modules. This decision was taken because the software systems domain itself can be divided in three layers, namely: (i) the business layer, in which Agents like Organizations and its members formulate goals and requirements for achieving these goals; (ii) the software system layer, that has the purpose of providing requirements for functions able to satisfy the goals of these business entities, as well as to connect these entities and machines; and (iii) the Machine layer, in which the machine is able to execute a translation of a program specification[8] to realize these requirements. This division of the domain in layers is also used in ISO standards 29148 [41] and 12207 [42].

4.1. Business Layer

The first part of the ontology, depicted in Figure 7, represents the business/organization environment in which a software system exists.

Business Requirements are high-level Goals of an Organization towards the

[8]The source code of a Program is a translation of a Program Specification to a machine-readable language.

system-to-be. 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 is an Agent (in our domain, either an Organization or an 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 inhering in A*. This relation also appears in Figure 8, between Stakeholder Requirement and Stakeholder.

In this contexts, goals represent the main reason for why a project is initiated, what the project intends to achieve, and indicate which metrics can be used to measure the project's success or failure [41]. Furthermore, as goals of an Agent, Business Requirements are usually described by a requirements specification or description (a type of Artifact) which is used, traced and maintained by the Organization). This Information Item (see Figure 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-cycle of the system.

Moreover, 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 frequent source of requirements, embedded with World Assumptions about the domain of the system-to-be. These assumptions will be further discussed in this work. They may or may not be explicit during the system development process. Zave and Jackson [38] defend that Assumptions should be treated as first class citizens in every software system project, being documented and managed as any other configuration item. Based on this, Wang et al. [36] proposed a small ontology of assumptions, which is being reused in ROSS with the prefix ASMP. In other words, a specification is a description of requirements *based on* a set of assumptions, i.e., different assumptions will result in different specifications and, if the assumptions are incorrect or incomplete, the specification might not be able to properly describe the requirements.

Business Requirements, are constrained by Business Constraints, which are Normative Descriptions recognized by the Organization. In our ontology, we define two types of Business Constraints: Business Rules and External Regulations. In this domain, Business Rules are Normative Descriptions that define a policy, guideline or practice that constrains some aspects of a business project and its intended results. Business Rules are not requirements themselves, but they can be the origin/act as constraints of several types of requirements [43]. For example, a software-factory organization that has the internal policy of producing applications that are optimized for certain type of platform will have to create and implement specific requirements to satisfy this rule.

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. In a way analogous to Business Rules, External Regulations are important to all types of Organization, because they are capable of constraining the Business Requirements. In line with CMMI [7], we argue that the relationships between higher-level Business Requirements and the Normative Descriptions that exist around (Business Rules) and over (External Regulations) them are extremely important and must be traced and maintained during the entire software life-cycle.

As Business Requirements are high-level goals [41], they tend to be far from particular implementable solutions. For example, an Organization that desires to improve team productivity by reducing their dependence on spreadsheets as a team management tool may decide to build their own team tool, or acquire one. At this point, the Organization is able to formulate a desired state of affairs, without necessarily committing to what what solution to implement. If they decide to create their own tool, a software system project will be initiated. Because of this "distance" that exists between the initial need and the domain of the solution, a more concrete (refined) requirement closer to the solution domain must be formulated.

Stakeholder Requirements are statements of the needs of a particular Stake-

holder or a group of stakeholders, which are members of the Organization over the system-to-be. They represent the needs of a stakeholder and they must be somehow aligned with business requirements. In other words, Stakeholder Requirements can be seen as a stakeholder's point of view towards an existing Business Requirements. Finally, as they are more concrete than Business Requirements but still exist in the business level, they serve as a bridge between Business Requirements and the other types of requirements that are solution-oriented. Stakeholder Requirements are Goals that are usually described in a specific Information Item called a Stakeholder Requirements Specification (StRS).

4.2. Systems Layer

The second part of the ontology, depicted in Figure 8, 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 [38] 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, Software System is defined by ISO 24765 [4] 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.

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

For example, Microsoft Windows 10 is an (operating) system composed by many 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, such as servers, sensors or peripherals.

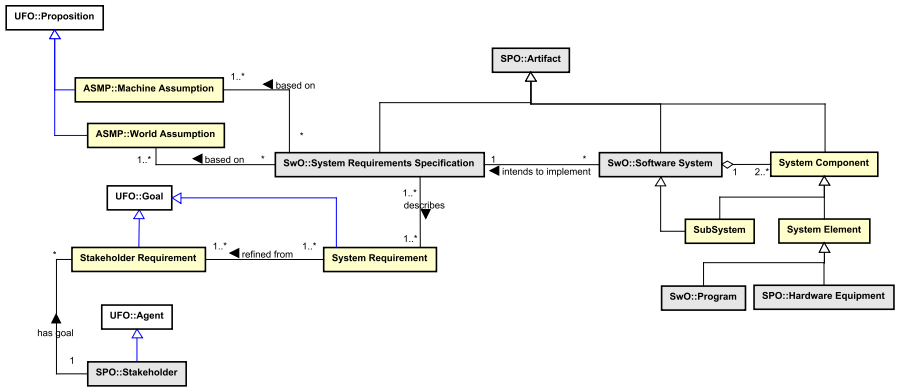

Figure 8: The System layer and the notion of Software System

Furthermore, the notion of software proposed here, in line with [2, 19], allows for a software systems 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, however, in a sense "behave like types" given that they can be made repeatable in various 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 [20], 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 [41]. 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

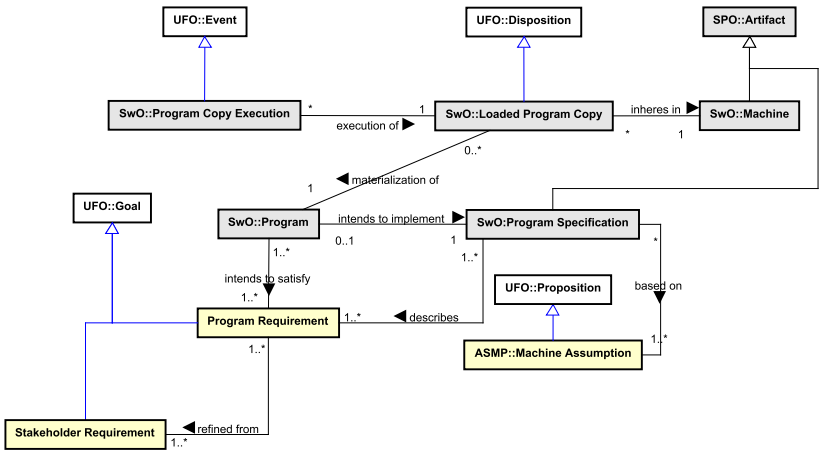

Figure 9: The Machine Layer

Stakeholder Requirements. This relation between both types of requirements provides the connection between the business layer and the system layer.

Furthermore, similarly from their higher-level counterparts, System Requirements are *described* in an Information Item called System Requirements Specification (SyRS) [41]. 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 were 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.

4.3. Machine Layer

Finally, the last part of the ontology, presented in Figure 9, represents the parts of a software system that exist inside a Machine and is focused on the concept of Program.

Wang et al. [2] promote an extensive discussion and a reference ontology of

software artifacts.[9] Based on Wang et al.'s work and in order to capture the complex nature of software, in SwO [19], 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 [36]. Moreover, Programs are artifacts constituted by source code, although not being identical to code. 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. They 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 [2].

Furthermore, as artifacts produced through a development process, a Program *intends to implement* a Program Requirements Specification that describes the Program Requirements related to such Program. In this context, we can say that the Program[10] *intends 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 [41, 5], and are focused on a possible solution for the computational part of the system-to-be. Figure 10 presents an adaptation of a figure presented in ISO 29148 [41], 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.

[9] 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.

[10] For a deeper discussion about the concept of Program, please see [19].

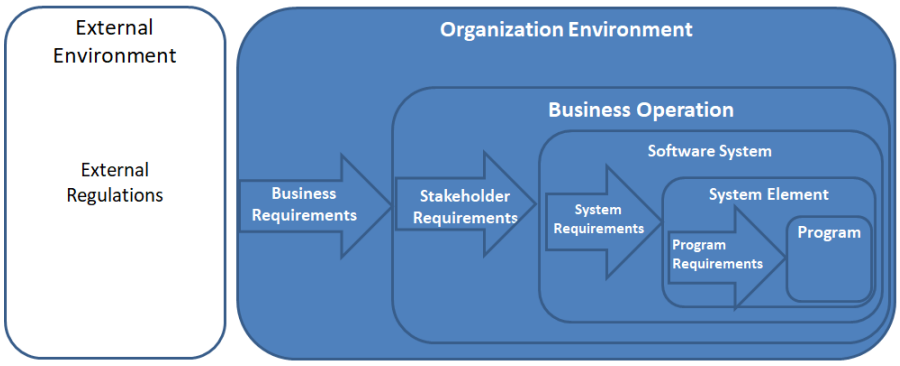

Figure 10: Adaptation of an example of requirements scope in a business context. Figure originally presented in ISO 29148 [41].

Moreover, as mentioned earlier in this section, a Program Specification is not necessarily a formal document in natural language about requirements. As in the original formula proposed by Zave and Jackson [38], Program Specification is heavily related to the assumptions that exist in the context of the Machine. For example, the source code of a Program is also a type of Program Specification, since it will be derived from the Program Requirements and, because of that, it will heavily depend on the assumptions (Machine Assumptions that the developers have towards the programming platform, i.e., the Machine).[11] More precisely, different developers, with different Machine Assumptions will produce different implementations, which can satisfactory or not achieve the same requirements.

5. Evaluation

For ontology evaluation, SABiO prescribes that ontologies need to go through ontology verification and validation techniques. These should be conducted in a particular manner as described in the sequel.

[11] 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.

5.1. 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 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 1: 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.

Following, table 2 presents the ROSS verification regarding its competency questions. Once more, since ROSS is able to adequately respond to all proposed CQs, the verification is considered a success.

Table 2: 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.

5.2. 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 suggests that a good and relatively simple validation technique is to check if the created reference ontology may be instantiated to represent real-world situations that are related to the domain of the ontology.

As previously discussed, we want to conduct here a particular type of evalu-

ation of these two ontologies in terms of their capacities to support the analysis of software risks associated with systems' anomalies. So, in order to do that, we employed here real-world scenarios of famous cases of software failures. Our aim is showing that the combination OSDEF, ROSS and COVR is capable of representing these real-world situations. We choose these specific cases because they are well-known and well-documented cases of failures of software systems that caused major damage. 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. These instance-level models have been used as an evaluation technique associated with OntoUML models (e.g., in [10, 44]). The color scheme used here is the same as the aforementioned OntoUML color convention[12]. Moreover, we here abuse the UML class diagram notation in the following manner: boxes represent instances of the elements of the ontology; arrows represent links; in the lower partition of each of these boxes represent the types from OSDEF, COVR and ROSS instantiated by each of these elements. This convention is also used in Figures 12 and Figure 13.

Case 1 (see Figure 11[13]): the Therac-25 disaster [45]. 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, condi-

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

[13]Please see [26] 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 [10].

tions that can be understood as a 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 (see Figure 12): in 2013, Spamhaus, a nonprofit professional protection service in 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

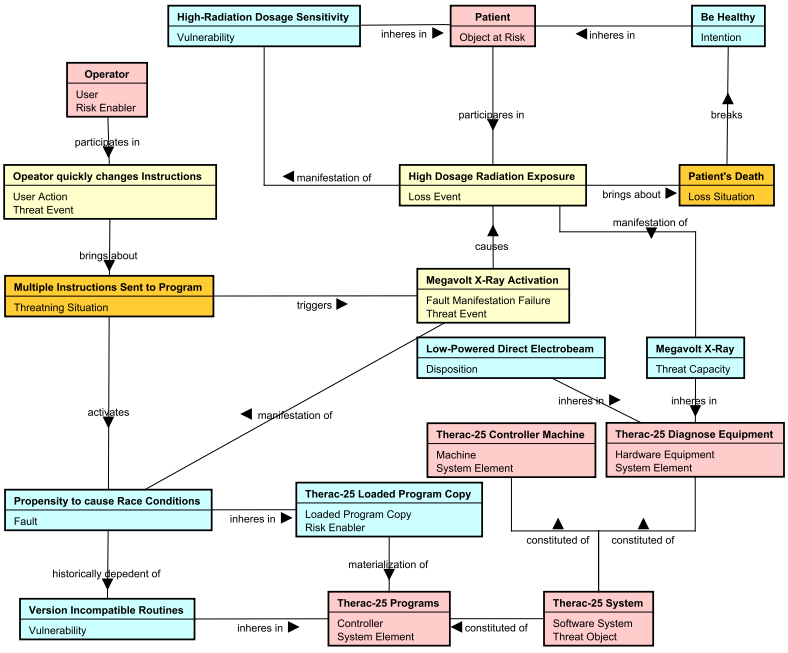

Figure 11: The Therac-25 Case as an instance of OSDEF, COVR and ROSS.

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 (see Figure 13): In 1991, during the Gulf War, the Patriot missile-system [46] 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,

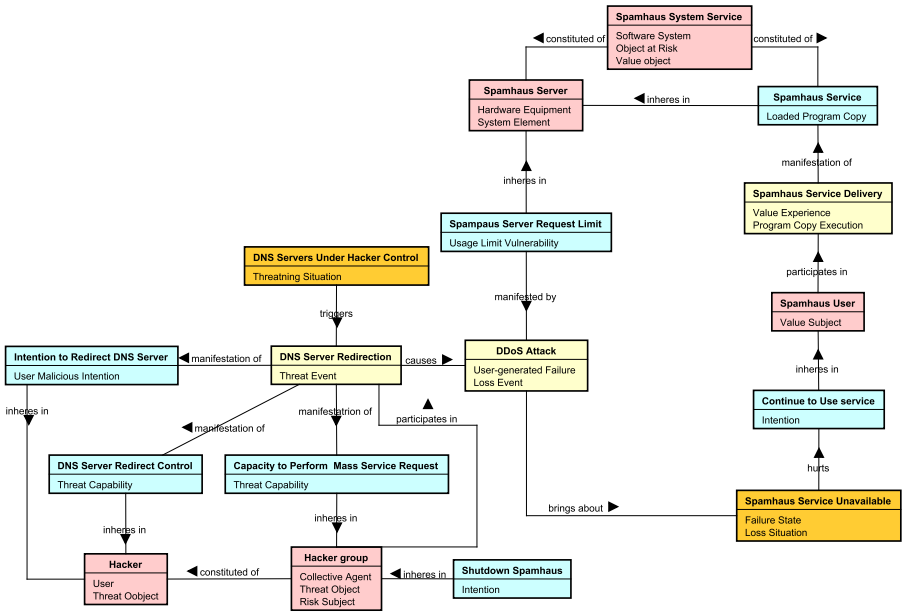

Figure 12: The Spamhaus System case as an instance of OSDEF, COVR and ROSS.

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 runtime). 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 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 runtime, 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. Related Work

Del Frate [31] 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*. The author also discusses the relation between a failure — an event that happens to an artifact — and a fault — a 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 the work presented here the interpretation

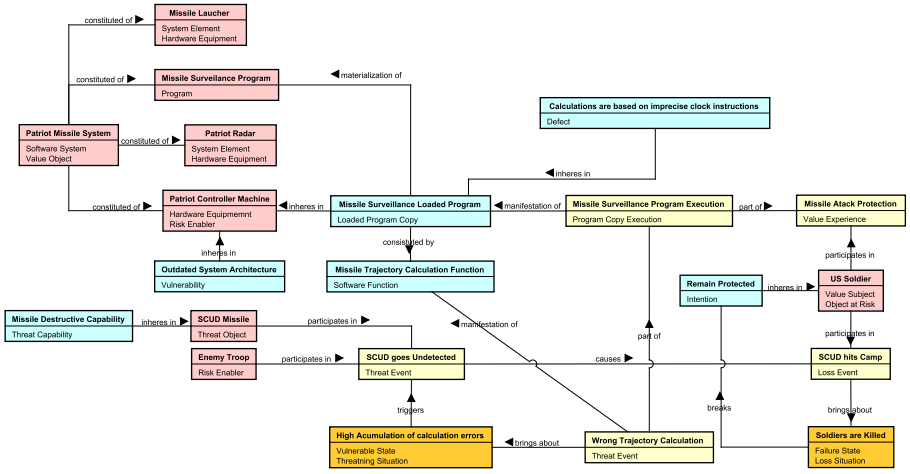

Figure 13: The Patriot Missile System case as an instance of OSDEF, COVR and ROSS.

of failures as events. However, honoring the terminology employed in software engineering standards, we conceive faults as *processual roles* [15] 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 difference is that we take into account other types of anomalies, such as defects and errors (even those caused by the direct participation of human agents). Other distinction worth mentioning is that our work is focused on software 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.

Kitamura & Mizoguchi [47] performed an ontological analysis over the fault process and proposed an ontology of faults. The ontology of faults provides a categorization of different types of Faults and relates them with other concepts that are part of the fault process [14]. Additionally, this ontology was used to provide a vocabulary for specifying the scope of a fault diagnostic activity and

[14] faults are differentiated between: externally or internally caused; structural or property-related; continued and intermittent or direct and indirect

as conceptual model for the development of a ontology-based fault diagnostic system. The idea is that this ontology-based system would be capable of performing a deeper analysis of a software system to uncover root causes of Faults. In comparison with OSDEF, Kitamura & Mizoguchi's ontology of faults has a different focus, which is centered in the fault process and in specifying a complete vocabulary centered in the concept of Fault. Besides, the ontology of faults is not grounded on any foundational ontology.

Avizienis et al. [48] 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 use 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.

Finally, we should emphasize that, unlike these efforts, OSDEF has been conceived in connection with other UFO-based Software Engineering domain ontologies [21], [20] and with the purpose of contributing to a Software Engineering Ontology Network (SEON) [19]. Although these previous works do not address aspects related to software anomalies, they provide context to our work.

7. Conclusions

In this extended version of [17], we present OSDEF and complement it with ROSS in order to provide an ontological analysis of defects, errors and failures that are part of the Software Systems life-cycle. Moreover, we analyze these

concepts from a risk analysis perspective, in light of the COVR ontology. In order to provide a more rigorous definition and a better representation of their real-world semantics, the ontologies presented are grounded in UFO and the definitions are based on international standards and on the scientific literature in the domain of software failure. OSDEF and ROSS contribute to the conceptual modeling and management of software systems and to the problems related to them in a number of ways that are summarized as follows.

Firstly, by making use of UFO's foundational categories, the ontologies provide a conceptual analysis of the *nature* of different types of anomalies, systematizing the overloaded use of the term *anomaly* in the Software Engineering literature. Furthermore, they can serve as a reference model for supporting the ontological analysis and conceptual clarification of real-world failure cases. For instance, although sometimes used almost interchangeably, we manage to show that notions such as Failure, Fault, Defect and (User) Error (Erroneous User Action) refer to different types of phenomena. In a nutshell, a Failure is an Event caused by a Vulnerability (a Disposition). A Defect is a Vulnerability inhering in the Program\Loaded Program Copy or in a Hardware Equipment that is manifested at runtime. In this manifestation, the Defect plays the role of a Fault. An Error (Erroneous User Action) is an Action (an Event brought about by an Agent) that neglects the assumptions under which a Program was designed.

Secondly, both OSDEF and ROSS, as reference ontologies, can be used to support the development of software systems management tools, such as issue trackers or knowledge and configuration management-related tools, since they have their base on widely accepted standards. Moreover, they can also be used to support interoperability of existing tools.

Thirdly, the ontologies establish a common vocabulary for their domains, improving communication among software engineers and stakeholders, avoiding construct overloads and other types of communication problems.

Fourth, in addition to these uses as reference models, operational versions of these ontologies, implemented in logical language (e.g., Common Logic or OWL) can be used to semantically annotate configuration management and

issue tracker data that are directly related to the occurrence of software anomalies. As result of this annotation, one may reason about anomalies in multiple ways, by navigating the ontological model.

Finally, as future work, we intend to connect OSDEF and ROSS to our Software Engineering Ontology Network (SEON) [24] and develop tools based on them to help organizations perform analyses of their software system assets, using a value and risk perspective.

8. Acknowledgements

This paper is dedicated to Ricardo Falbo (*in memoriam*) for all his uncountable contributions to our lives. B. Borlini and V. Souza are currently supported by CNPq (407235/2017-5, 433844/2018-3) and CAPES (23038.028816/2016-41).

References

[1] X. Wang, N. Guarino, G. Guizzardi, J. Mylopoulos, Software as a social artifact: a management and evolution perspective, in: International Conference on Conceptual Modeling, Springer, 2014, pp. 321–334.

[2] Wang, X. et al., Towards an Ontology of Software: a Requirements Engineering Perspective, in: Proc. of the 8th International Conference on Formal Ontology in Information Systems, Vol. 267, 2014, pp. 317–329.

[3] N. Irmak, Software is an abstract artifact, Grazer Philosophische Studien 86 (1) (2013) 55–72.

[4] ISO, IEC, ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary, Tech. rep. (Aug 2017). doi:10.1109/IEEESTD.2017.8016712.

[5] P. Bourque, R. E. Fairley, et al., Guide to the software engineering body of knowledge (SWEBOK (R)): v.3.0, IEEE Computer Society Press, 2014.

[6] B. H. Cheng, J. M. Atlee, Research directions in requirements engineering, in: 2007 Future of Software Engineering, IEEE Computer Society, 2007.

[7] SEI/CMU, CMMI® for Development, Version 1.3, Improving processes for developing better products and services, no. CMU/SEI-2010-TR-033. Software Engineering Institute.

[8] IEEE, IEEE 1044: Standard Classification for Software Anomalies, Tech. rep., Institute of Electrical and Electronics Engineers, Inc (2009).

[9] G. Guizzardi, On ontology, ontologies, conceptualizations, modeling languages, and (meta) models, Frontiers in artificial intelligence and applications 155 (2007) 18.

[10] T. P. Sales, F. Baião, G. Guizzardi, J. P. A. Almeida, N. Guarino, J. Mylopoulos, The common ontology of value and risk, in: International Conference on Conceptual Modeling, Springer, 2018, pp. 121–135.

[11] R. de Almeida Falbo, Sabio: Systematic approach for building ontologies., in: ONTO. COM/ODISE@ FOIS, 2014.

[12] G. Guizzardi, Ontological Foundations for Structural Conceptual Models, Phd thesis, University of Twente, The Netherlands (2005).

[13] G. Guizzardi, G. Wagner, J. P. A. Almeida, R. S. Guizzardi, Towards ontological foundations for conceptual modeling: The unified foundational ontology (ufo) story, Applied ontology 10 (3-4) (2015) 259–271.

[14] 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, pp. 327–341.

[15] Benevides, A.B. et al., Representing a reference foundational ontology of events in sroiq, Applied Ontology 14 (3) (2019) 293–334.

[16] IEEE, IEEE 1012: Standard for System, Software, and Hardware Verification and Validation, Tech. rep., Technical report, Institute of Electrical and Electronics Engineers, Inc (2016).

[17] B. B. Duarte, R. d. A. Falbo, G. Guizzardi, R. S. S. Guizzardi, V. E. S. Souza, Towards an ontology of software defects, errors and failures, in: Intl. Conference on Conceptual Modeling, Springer, 2018, pp. 349–362.

[18] A. C. d. O. Bringuente, R. d. A. Falbo, G. Guizzardi, Using a foundational ontology for reengineering a software process ontology, Journal of Information and Data Management 2 (3) (2011) 511.

[19] B. B. Duarte, V. E. S. Souza, A. L. d. C. Leal, G. Guizzardi, R. d. A. Falbo, R. S. S. Guizzardi, Ontological foundations for software requirements with a focus on requirements at runtime, Applied Ontology (2018) 1–33.

[20] R. D. A. Falbo, G. Bertollo, A software process ontology as a common vocabulary about software processes, International Journal of Business Process Integration and Management 4 (4) (2009) 239–250.

[21] Guizzardi, G. at al., Grounding Software Domain Ontologies in the Unified Foundational Ontology (UFO): The case of the ODE Software Process Ontology, in: 11th Iberoamerican Conf. on Soft. Engineering (CIbSE), 2008.

[22] Guizzardi, R.S.S. et al., An Ontological Interpretation of Non-Functional Requirements, in: Proc. of the 8th International Conference on Formal Ontology in Information Systems, Vol. 267, IOS Press, 2014, pp. 344–357.

[23] M. Verdonck, F. Gailly, Insights on the use and application of ontology and conceptual modeling languages in ontology-driven conceptual modeling, in: 35th Intl. Conf. on Conceptual Modeling, ER, Gifu, Japan, 2016, pp. 83–97.

[24] F. B. Ruy, R. d. A. Falbo, M. P. Barcellos, S. D. Costa, G. Guizzardi, Seon: A software engineering ontology network, in: 20th Knowledge Engineering and Knowledge Management(EKAW), Springer, 2016, pp. 527–542.

[25] Guizzardi, G. et al., Endurant types in ontology-driven conceptual modeling: Towards ontouml 2.0, in: International Conference on Conceptual Modeling, Springer, 2018, pp. 136–150.

[26] C. M. Fonseca, D. Porello, G. Guizzardi, J. P. A. Almeida, N. Guarino, Relations in ontology-driven conceptual modeling, in: International Conference on Conceptual Modeling, Springer, 2019, pp. 28–42.

[27] G. Guizzardi, N. Guarino, J. P. A. Almeida, Ontological considerations about the representation of events and endurants in business models, in: International Conference on Business Process Management, Springer, 2016.

[28] J. P. A. Almeida, P. D. Costa, G. Guizzardi, Towards an ontology of scenes and situations, in: 2018 IEEE Conference on Cognitive and Computational Aspects of Situation Management (CogSIMA), IEEE, 2018, pp. 29–35.

[29] É. F. de Souza, R. d. A. Falbo, N. L. Vijaykumar, ROoST: Reference Ontology on Software Testing, Applied Ontology (2017) 1–32.

[30] M. Grüninger, M. Fox, Methodology for the Design and Evaluation of Ontologies (1995).

[31] L. Del Frate, Preliminaries to a formal ontology of failure of engineering artifacts., in: FOIS, 2012, pp. 117–130.

[32] Guizzardi, R.S.S. et al., Ontological distinctions between means-end and contribution links in the i* framework, in: Proc. of the 32th International Conference on Conceptual Modeling, ER, Hong-Kong, China, 2013.

[33] I. Hogganvik, K. Stølen, A graphical approach to risk identification, motivated by empirical investigations, in: International Conference on Model Driven Engineering Languages and Systems, Springer, 2006, pp. 574–588.

[34] D. Alam, M. Zaman, T. Farah, R. Rahman, M. S. Hosain, Study of the dirty copy on write, a linux kernel memory allocation vulnerability, in: Intl. Conf. on Consumer Electronics and Devices (ICCED), IEEE, 2017.

[35] R. Chillarege, Orthogonal defect classification, Handbook of Software Reliability Engineering (1996) 359–399.

[36] Wang, X. et al., How software changes the world: The role of assumptions, in: 10th IEEE Intl. Conf. on Research Challenges in Information Science, RCIS, Grenoble, France, 2016, pp. 1–12. doi:10.1109/RCIS.2016.7549327.

[37] S. A. Fricker, K. Schneider (Eds.), Requirements Engineering: Foundation for Software Quality - 21st International Working Conference, REFSQ 2015, Essen, Germany, Vol. 9013 of LNCS, Springer, 2015.

[38] P. Zave, M. Jackson, Four Dark Corners of Requirements Engineering, ACM Trans. on Software Engineering and Methodology 6 (1) (1997) 1–30.

[39] C. A. Gunter, E. L. Gunter, M. Jackson, P. Zave, A reference model for requirements and specifications, IEEE Software 17 (3) (2000) 37–43.

[40] Duarte, B.B. et al., Towards an ontology of requirements at runtime, in: Proc. of the 9th International Conference on Formal Ontology in Information Systems (FOIS 2016), Vol. 283, IOS Press, 2016, p. 255.

[41] I. ISO, Ieee. 29148: Systems and software engineering-requirements engineering, Tech. rep., International Organization for Standardization (2018).

[42] I. ISO, ISO/IEC International Standard - Systems and software engineering – Software life cycle process, Tech. rep. (2017).

[43] K. Wiegers, J. Beatty, Software requirements, Pearson Education, 2013.

[44] G. Amaral, T. P. Sales, G. Guizzardi, D. Porello, Towards a reference ontology of trust, in: OTM Confederated International Conferences" On the Move to Meaningful Internet Systems", Springer, 2019, pp. 3–21.

[45] N. G. Leveson, C. S. Turner, An investigation of the Therac-25 accidents, Computer 26 (7) (1993) 18–41. doi:10.1109/MC.1993.274940.

[46] P. M. Defense, Software problem led to system failure at dhahran, saudi arabia, US GAO Reports, report no. GAO/IMTEC-92-26.

[47] Y. Kitamura, R. Mizoguchi, An ontological analysis of fault process and category of faults, in: Proceedings of tenth international workshop on principles of diagnosis (DX-99), 1999, pp. 118–128.

[48] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE transactions on dependable and secure computing 1 (1) (2004) 11–33.