Towards an Ontology of Software Defects, Errors and Failures

Bruno Borlini Duarte[1], Ricardo A. Falbo[1], Giancarlo Guizzardi[1,2], Renata S. S. Guizzardi[1], Vítor E. S. Souza[1]

[1] Ontology and Conceptual Modeling Research Group (NEMO)
Federal University of Espírito Santo, Brazil
`bruno.b.duarte@ufes.br`
`{falbo,rguizzardi,vitorsouza}@inf.ufes.br`
[2] Conceptual and Cognitive Modeling Research Group (CORE)
Free University of Bolzano-Bozen, Italy
`gguizzardi@unibz.it`

Abstract. The rational management of software defects is a fundamental requirement for a mature software industry. Standards, 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, potential interoperability problems between supporting tools, and, consequently, to a poorer adoption of these standards and tools in practice. We address this problem of the lack of a consensual conceptualization in this area by proposing a reference conceptual model (domain ontology) of Software Defects, Errors and Failures, which takes into account an ecosystem of software artifacts. The ontology is grounded on the Unified Foundational Ontology (UFO) and is based on well-known standards, guides and capability models. We demonstrate how this approach can suitably promote conceptual clarification and terminological harmonization in this area.

Keywords: Ontologies in Software Engineering · UFO · Software Anomaly.

1 Introduction

In software and systems engineering, the term *anomaly* denotes a condition that deviates from expectations, based on requirements specifications, design documents, user documents, standards as well as user's and/or modeler's perceptions and experiences [1]. A software anomaly (usually loosely referred by terms such as *bug*, *glitch*, *error* or *defect*) is a situation that suggests a potential problem in a software artifact [2]. In other words, these concepts are used to denote that an artifact is not behaving as expected or is not producing the desired results.

This informal use, as common and practical as it is used in our daily conversations, may be the source of ambiguity and false-agreement problems, since the concept anomaly is constantly overloaded, refering to many entities with distinct nature. In a more formal environment, this construct overload may lead to communication problems and material losses. Because of that, having a way to properly classify software anomalies is important. Proper classification enables the development of different types of anomaly profiles that could be produced as one indicator of product quality. Also, the information that is generated when an organization understands and systematically classifies 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 [3].

Defects, errors and failures have a negative impact on important aspects of software, such as reliability, efficiency, overall cost and even lifespan. A software with a fairly large "defect density" may go through heavy reconstruction, since it does not meet a minimal acceptance criteria [4]. In more extreme cases, it may be abandoned/discontinued in its early years of usage.

The Guide to Software Engineering Body of Knowledge (SWEBOK) [5] emphasizes the need of a consensus about anomaly characterization and how a well-founded classification could be used in audits and product reviews. A proper defect characterization policy can also provide a better understanding and facilitate corrections in the product or in the process. For an example of this necessity, CMMI [6] defines that organizations should create or reuse some form of defect classification method. It also suggests the use of a defect density index for many work products that are part of the software development process.

The most recent version of the Standard Classification of Software Anomalies [3] provides a classification for different types of anomalies, including information about how they are related. Other standards are more concerned about how to deal with anomalies in different perspectives. For instance, IEEE 1012 [7] is focused on the Verification & Validation phase of the software life-cycle. On the other hand, IEEE 1028 [4] focuses on anomalies in a software audit context, which affects clients and users of a (software) product.

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* of software anomalies. 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. OSDEF was developed following the process defined by the Systematic Approach for Building Ontologies (SABiO) [8] and grounded in the Unified Foundational Ontology (UFO) [9], including UFO's Ontology of Events [10]. In order to extract consensual information about the domain, we analyzed relevant standards, guides and capability models such as CMMI [6], SWEBOK [5], IEEE Standard Classification for Software Anomalies [3], IEEE Standard for System, Software, and Hardware Verification and Validation [7] as well as complementary current Software Engineering literature. Finally, the ontology was evaluated by verification and validation techniques.

The remainder of this paper is structured as follows. Section 2 introduces the ontological foundations used for developing OSDEF. Section 3 presents the Ontology of Software Defects, Errors and Failures. Section 4 evaluates the proposed ontology. Section 5 discusses related work. Finally, Section 6 concludes the paper.

2 Foundations: UFO and the Software Process Ontology

We ground the Ontology of Software Defects, Errors and Failures (OSDEF) in UFO [9]. This choice is motivated by the following: (i) UFO's foundational categories address 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 [11–13]; (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 [14]. By using an ontology that is frequently used, we increase the reusability of this work, also facilitating its future integration in *ontology networks* in software engineering [15].

UFO is originally composed of three main parts: UFO-A, an ontology of endurants [9]; UFO-B, an ontology of perdurants/events [10]; and UFO-C, an ontology of social entities (both endurants and perdurants) built on top of UFO-A and UFO-B [12]. However, for brevity, Figure 1 presents only a fragment of UFO that contains the categories that are essential for the purpose of this article. Moreover, we illustrate these categories and their relations using UML diagrams that express typed relations connecting categories, cardinality constraints for these relations, subsumption constraints, as well as disjointness constraints relating sub-categories with the same super-category. UFO has been formally characterized in [9, 10, 16]. Thus, it is important to emphasize that the following UML diagrams are used here for illustration purposes only.

Endurants and Perdurants are Concrete Particulars (also called *concrete individuals*), 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 [10, 17]. 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.

Actions are Events that are *performed* by Agents (persons, organizations or teams) with the specific purpose of satisfying *intentions* of that Agent. However, it is important to realize that although all Actions are based on intentions of Agents, if those 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

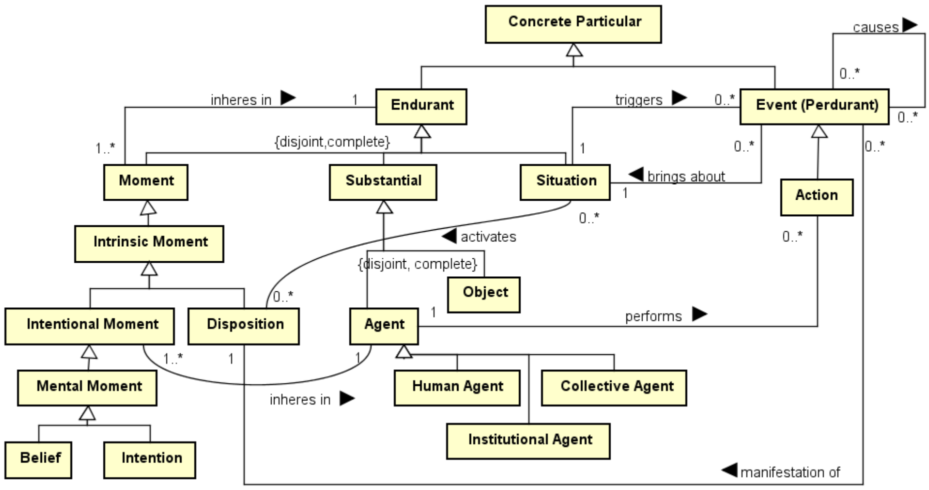

Fig. 1. Fragment of UFO showing Events, Agents and Objects.

even dent) the goals (propositional content of the intention) that motivated that action. Moments (also called *aspects*, *particularized properties* or *tropes*) 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) cease 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, but when they are *manifested*, it is by the occurrence of an Event [10]. Examples of Disposition 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 [10] for a deeper discussion about Events, Situations and Moments (including Dispositions).

For OSDEF, we reuse the concept of Software Artifact that is presented in the Software Process Ontology (SPO) [18]: objects intentionally 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, may be responsible for them (e.g., a user or a development team). We also reuse the concept of Hardware Equipment, which are physical objects used for running software programs or to support some related action (e.g., a computer or a tablet). Moreover, Hardware Equipment are not considered Software Artifacts because they are not created in the context of a software project, although they can be considered resources in a software project. We also reused the Program concept that is present in the Software Ontology (SwO) [19]. According to SwO, a Program is an 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. A complex aggregation of Programs can constitute a software system.

3 An Ontology of Software Defects, Errors and Failures

To build the Ontology of Software Defects, Errors and Failures (OSDEF), we apply SABiO [8], a method for building domain ontologies [20] that incorporates best practices from Software Engineering and Ontology Engineering. We chose SABiO because it is focused on the development of domain ontologies. Moreover, it has been successfully used on the development of several domain ontologies in Software Engineering, such as the Software Process Ontology (SPO) [11] and the Software Ontology [21] and other ontologies developed in the context of SEON, a Software Engineering Ontology Network [15]. Moreover, SABiO explicitly recognizes the importance of using foundational ontologies in the ontology development process to improve the ontology quality, representativity and formality.

SABiO's development process is composed of five phases: (1) purpose identification and requirements elicitation; (2) ontology capture and formalization; (3) operational ontology design; (4) operational ontology implementation; and (5) testing. These phases are supported by well-known activities in the Requirements Engineering life-cycle, such as knowledge acquisition, reuse, documentation, etc. Here, since our main goal is to produce a domain ontology as a *reference conceptual model*, we focus on the first two phases of SABiO, executed in an iterative way, refining the ontology at each iteration. As discussed in Section 4, we also conducted verification and validation of the proposed reference conceptual model. Phases (3) to (5), i.e., the design, implementation and testing of the reference ontology proposed here in a computational language (e.g., Common Logic, OWL, HOL-Isabelle, Alloy) are left for future work.

As previously mentioned, the term *anomaly* is commonly used to refer to a variety of notions of distinct ontological nature. Because of that, OSDEF was developed to provide an ontological conceptualization of the different types of software anomalies that exist throughout the software life-cycle. To elaborate on these different types of entities, we raised a set of Competency Questions (CQ), which are questions that the ontology should be able to answer [22]. In a Requirements Engineering perspective, CQs are analogous to the functional requirements of the ontology [8]. Moreover, CQs help to refine the scope of the ontology and can also be used in the ontology verification process. For OSDEF, CQs were raised and refined in a highly-interactive way, through analysis of the international standards mentioned in Section 1 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?

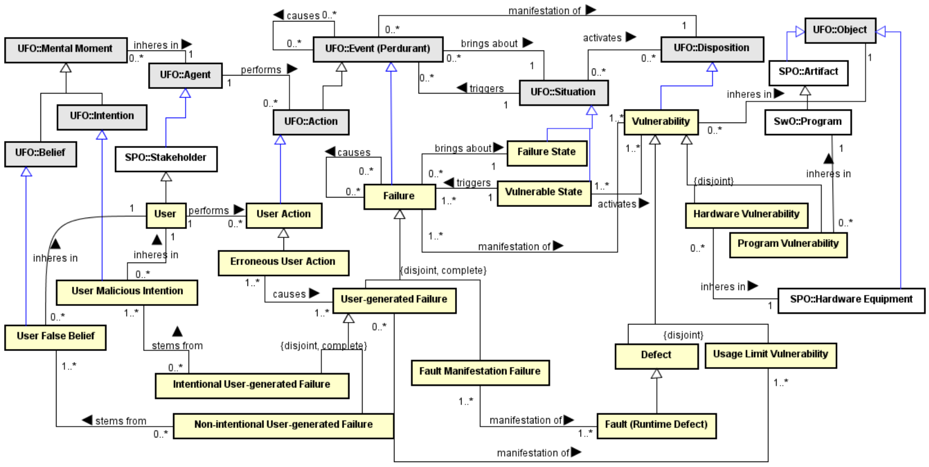

Fig. 2. Conceptual Model of the Ontology of Software Defects, Errors and Failures.

Figure 2 shows 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 [3, 7, 1] and as employed in scientific literature [23], 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 hurts the goals of stakeholders [24], which motivated the creation of that software. 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 experience failures as well). As defined in UFO [10], *causation* is a relation of *strict partial order* and, hence, failures cannot be their own causes or causes or their causes but failures can (perhaps indirectly) trigger other failures in a chain of causation.

As Events, Failures are directly related with 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 that will be manifested in that Failure. In other words, the state of being exposed to the possibility of a harm or an 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. The occurrence of the failure transforms a portion of reality to another:

in its pre-situation, the software is executing, it has the disposition to manifest the failure (i.e., a Vulnerability) but the failure 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 (at least not as intended by 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 until some action is taken in order to bring the software system back to a execution state in which it is capable to properly execute its functions. Failures can also be classified by other properties, such as severity or effect. We are also not addressing these finer-grained classification here because, once again, we are more interested in providing an ontological analysis of the nature of different software anomalies than in providing a terminological systematization. In our view, the former is a prerequisite for the latter.

Failures are classified in two distinct subtypes: Fault Manifestation Failures and User-Generated Failures. The former are Failures that are manifestations of Faults and are not *caused by* User Actions; the latter are Failures that are directly *caused by* User Actions. These two subtypes have sub-distinctions of their own.

A Vulnerability[3] represents the Dispositions that can exist in software artifacts or in hardware equipments. We thus refined this concept in two distinct generalization sets. The first represents the types of the Dispositions that can be activated and manifest Failures and is composed by Defects and Usage Limit Vulnerabilities. The second represents the types of entities in which those Dispositions inhere: a Hardware Vulnerability inheres in a Hardware Equipment, while a Program Vulnerability inheres in a Program. With that said, we have that both dispositions, Defect and Usage Limit Vulnerability can *inhere in* both types of Objects and thus, we have the following definitions: defects that inhere in Programs are Program Defects; defects that inhere in Hardware Equipment are Hardware Defects. Moreover, we have that a Program Usage Limit Vulnerability is a Usage Limit Vulnerability that inheres in a Program, and that a Hardware Usage Limit Vulnerability is a Usage Limit Vulnerability that inheres in a Hardware Equipment.

A Defect is a type of Vulnerability that can exist in Programs or Hardware Equipments. It is defined by the Standard Classification for Software Anomalies [3] 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 [27, 5] have in common is that Defects are understood as properties of Objects. However, differently from moments that are manifested all the time (e.g. the color of a wall), Defects may never be activated and, consequently, never be manifested into Failures. For example, suppose that a program

[3] 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 [25, 26] of vulnerabilities as dispositions that can be manifested by events that can hurt stakeholder's goals [24] or diminish something's *perceived value* [26].

has a bad implementation of the method **retrieveUsersByLastName**, that can cause a Failure in the software system, which will not be able to execute the functionalities that are associated to that Defect. If that method is never invoked during a program execution, the system may never experience the Failure that is a manifestation of that particular Defect. Given this characteristic, we take Defects (Vulnerabilities in general) to be Dispositions that inhere in Objects.

Defects can exist throughout the entire life-cycle of a software [28]. As previously mentioned, some Defects can (accidentally) refrain from being manifested across software executions. When a Defect is manifested in 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 [29], 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 [30], 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 DDoS (Distributed Denial of Service) attack. In this case, the server that is being attacked has no Defect (and, hence, no Fault), it 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 witnessed on Nike's website during the 2017 Black Friday).

4 Evaluation

In order to evaluate the Ontology of Software Defects, Errors and Failures (OSDEF), we applied verification and validation techniques, as prescribed by SABiO. Regarding ontology verification, SABiO suggests a table that shows the ontology elements that are able to answer the competency questions (CQs) that were

Table 1. Results of ontology verification.

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 that is *manifestation of* a Usage Limit Vulnerability and is *caused by* an Erroneous User Action *stemming from* a User False Belief (Non-intentional) or a User Malicious Intention (Intentional). 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 (which is a subtype of Disposition) that *inheres in* an Object. A Defect that *inheres in* a Program (i.e., a Program Vulnerability) is called a Program Defect; A Defect that *inheres in* a Hardware Equipment (i.e., a Hardware Vulnerability), is called a Hardware Defect.
CQ3	Fault is a *subtype of* Defect which is manifested at runtime via a Fault Manifestation Failure.
CQ4	An error or, more precisely, an 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 Object. Analogous to Defect, it can *inhere in* a Program (Program Usage Limit Vulnerability) or a Hardware Equipment (Hardware Usage Limit Vulnerability).
CQ6	Vulnerable State is a *subtype of* Situation that *activates* a Fault (which, in turn, is manifested by a Failure) 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 by a Fault Manifestation Failure. An Erroneous User Action can *cause* a User-generated Failure, which is a *manifestation of* a Usage Limit Vulnerability.

raised. For validation, the reference ontology should be instantiated to check if it is able to represent real-world situations.

Table 1 illustrates the results of verification of OSDEF 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 answers all of the appropriate CQs.

For a brief validation, we took real-world scenarios of famous cases of software failures and used the ontology to analyze them, showing that OSDEF is capable of representing and analyzing these situations.

Case 1: the Therac-25 disaster [31]. Therac-25 is a medical equipment that handled two types of therapy: a low-powered direct electron beam and a megavolt X-ray mode. The issue was that the software that was responsible for controlling the equipment was reused from a previous model, missing important upgrades and adequate testing. The Fault was manifested into a critical Failure when an operator changed the therapy mode of the equipment too quickly, causing, instructions for both treatments to be simultaneously sent to the machine. 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 expecting an electro beam ended up receiving the X-ray and, because of that, died from radiation poisoning. This was an example of a Fault caused by a Program Vulnerability. Although the Fault Manifestation Failure was brought about by a User Action, however, this action cannot be considered an Erroneous User Action (since this cannot be considered a user's negligence of stated assumptions).

Case 2: in 1994, an entire line of Pentium processors could not calculate floating point operations precisely after the eighth decimal case [32]. No matter what software was executing the calculations, the Failure could be manifested since the Defect was intrinsic to the CPU of the computer. We can analyze this case based on OSDEF and on the reports that the Failures happened independently of which software was being executed. We can start our analysis by assuming that the whole Failure State started with a Hardware Vulnerability. The Vulnerability in this case was a Defect inhering in the chip that would prevent it from correctly process arithmetic operation with more than eight decimal cases. As a result, whenever a software execution would trigger the manifestation of that Vulnerability, aFault Manifestation Failure would occur.

Case 3: in 2013, Spamhaus, a nonprofit professional protection service, was the target of what might have been the largest DDoS attack in history. Hackers redirected hundreds of controlled DNS servers to send up to 300 gigabits of flood data to each server of the network, in order to stop them. For this type of situation, when the occurrence of a Failure is directly related with deliberate Actions of an Agent, OSDEF proposes the representation of the event as an Intentional User-generated Failure, since in this particular case the Agents that were responsible for the Failure were basing their Actions in a set of User Malicious Intentions.

5 Related Work

Del Frate [23] 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 — 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 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 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 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 [33] propose an ontological analysis of the fault process and an ontology of faults that provides a categorization of different types of Faults considering different facts, providing a vocabulary for specifying the scope of a diagnostic activity. Characteristics, ontological aspects (e.g., causality and parthood 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. It is also used in a diagnostic system that aims to enumerate deeper causes of Failures, providing "depth analysis" to diagnostic systems. In comparison with our ontology, this work has a different focus, which is centered in the fault process and in specifying different characteristics and constraints of Faults and Failures.

Avizienis et al. [34] 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.

Finally, we should emphasize that, unlike these efforts, OSDEF has been conceived in connection with other UFO-based Software Engineering domain ontologies [12, 11] 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.

6 Conclusions

The main contribution of this paper is proposing an Ontology of Software Defects, Errors and Failures (OSDEF), developed using the SABiO approach, based on a series of standards and capability models, and grounded in UFO. This ontology contributes to the conceptual modeling and management of software anomalies in a number of ways that are summarized as follows.

Firstly, by making use of UFO's foundational categories, OSDEF provides 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, this ontology 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 itself or in a Hardware Equipment that is manifested at runtime, in which manifestation this Defect plays the role of a Fault. An Error (Error Erroneous Action) is an Action (an Event brought about by an Agent) that neglects the assumptions under which a Program was designed.

Secondly, as a domain reference model, OSDEF can be used for the development of issue trackers or other types of configuration management-related tools, since it is based on widely accepted standards. Moreover, it can also be used for enabling interoperability between existing tools developed for these purposes.

Thirdly, the ontology establishes a common vocabulary for improving communication among software engineers and stakeholders, avoiding construct overloads and other types of communication problems.

Fourth, in addition to these uses as a reference model, an operational version of OSDEF (for instance, implemented in a logical language such as Common Logic or OWL) can be used to semantically annotate configuration management and software testing data that are directly related to the occurrence of software anomalies. In fact, as future work, we intend to connect OSDEF to our Software Engineering Ontology Network (SEON) [15]. In particular, we intend to develop an ontology of configuration management artifacts and combined it with OSDEF and related ontologies. This, in turn, will enable the development of a traceability tool to relate requirements and stakeholders goals with change requests and issue reports that are tracked during configuration management.

We also intend to strengthen the connection between the work developed here and a common ontology of Value and Risk [26]. After all, the management of anomalies in software artifacts is a special case of Risk Management applied to software. Also, we pretend to investigate further properties of Event types, such as regularity and consistency failures in the Failure context, (e.g., the case of the Therac-25). Finally, we intend to improve the evaluation of OSDEF by comparing (instantiating) the ontology with data produced by development tools such as, e.g., static analysis tools.

Acknowledgments

NEMO (`http://nemo.inf.ufes.br`) is currently supported by Brazilian research funding agencies CNPq (process 407235/2017-5), CAPES (process 23038.028816/2016-41), and FAPES (process 69382549/2015).

References

1. ISO: ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary. Technical report, International Organization for Standardization (Aug 2017)

2. Guimaraes, E., Garcia, A., Figueiredo, E., Cai, Y.: Prioritizing software anomalies with software metrics and architecture blueprints. In: Modeling in Software Engineering (MiSE), 2013 5th International Workshop on, IEEE (2013) 82–88
3. IEEE: IEEE 1044: Standard Classification for Software Anomalies. Technical report, Technical report, Institute of Electrical and Electronics Engineers, Inc (2009)
4. IEEE: IEEE 1028: Standard for Software Reviews and Autis. Technical report, Technical report, Institute of Electrical and Electronics Engineers, Inc (2008)
5. 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)
6. 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 (2010)
7. IEEE: IEEE 1012: Standard for System, Software, and Hardware Verification and Validation. Technical report, Technical report, Institute of Electrical and Electronics Engineers, Inc (2016)
8. Falbo, R.A.: SABiO: Systematic Approach for Building Ontologies. In Guizzardi, G., Pastor, O., Wand, Y., de Cesare, S., Gailly, F., Lycett, M., Partridge, C., eds.: 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 (sep 2014)
9. Guizzardi, G.: Ontological Foundations for Structural Conceptual Models. Phd thesis, University of Twente, The Netherlands (2005)
10. Guizzardi, G., Wagner, G., Falbo, R.d.A., Guizzardi, R.S.S., Almeida, J.P.A.: Towards Ontological Foundations for the Conceptual Modeling of Events. In: Proc. of the 32th International Conference on Conceptual Modeling, Springer (2013) 327–341
11. 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 **4**(4) (2009) 239–250
12. Guizzardi, G., de Almeida Falbo, R., 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) 127–140
13. Guizzardi, R.S.S., Li, F.L., Borgida, A., Guizzardi, G., Horkoff, J., Mylopoulos, J.: An Ontological Interpretation of Non-Functional Requirements. In Garbacz, P., Kutz, O., eds.: Proc. of the 8th International Conference on Formal Ontology in Information Systems. Volume 267., Rio de Janeiro, RJ, Brasil, IOS Press (sep 2014) 344–357
14. Verdonck, M., Gailly, F.: Insights on the use and application of ontology and conceptual modeling languages in ontology-driven conceptual modeling. In: Conceptual Modeling - 35th International Conference, ER 2016, Gifu, Japan, November 14-17, 2016, Proceedings. (2016) 83–97
15. Ruy, F.B., Falbo, R.d.A., Barcellos, M.P., Costa, S.D., Guizzardi, G.: Seon: A software engineering ontology network. In: Knowledge Engineering and Knowledge Management: 20th International Conference, EKAW 2016, Bologna, Italy, November 19-23, 2016, Proceedings 20, Springer (2016) 527–542
16. Benevides, A.B., Bourguet, J., Guizzardi, G., Peñaloza, R.: Representing the UFO-B foundational ontology of events in SROIQ. In: Proceedings of the Joint Ontology Workshops 2017 Episode 3: The Tyrolean Autumn of Ontology, Bozen-Bolzano, Italy, September 21-23, 2017. (2017)

17. Guizzardi, G., Guarino, N., Almeida, J.P.A.: Ontological considerations about the representation of events and endurants in business models. In: International Conference on Business Process Management, Springer (2016) 20–36
18. de Oliveira Bringuente, A.C., de Almeida Falbo, R., Guizzardi, G.: Using a foundational ontology for reengineering a software process ontology. Journal of Information and Data Management **2**(3) (2011) 511
19. Duarte, B.B., Souza, V.E.S., Leal, A.L.d.C., Guizzardi, G., Falbo, R.d.A., Guizzardi, R.S.S.: Ontological foundations for software requirements with a focus on requirements at runtime. Applied Ontology (2018) 1–33
20. Guizzardi, G.: On ontology, ontologies, conceptualizations, modeling languages, and (meta) models. Frontiers in artificial intelligence and applications **155** (2007) 18
21. de Souza, É.F., Falbo, R.d.A., Vijaykumar, N.L.: ROoST: Reference Ontology on Software Testing. Applied Ontology (2017) 1–32
22. Grüninger, M., Fox, M.: Methodology for the Design and Evaluation of Ontologies. In: IJCAI'95 Workshop on Basic Ontological Issues in Knowledge Sharing. (1995)
23. Del Frate, L.: Preliminaries to a formal ontology of failure of engineering artifacts. In: FOIS. (2012) 117–130
24. Guizzardi, R.S.S., Franch, X., Guizzardi, G., Wieringa, R.: Ontological distinctions between means-end and contribution links in the i* framework. In: Conceptual Modeling - 32th International Conference, ER 2013, Hong-Kong, China, November 11-13, 2013. Proceedings. (2013) 463–470
25. Hogganvik, I., Stølen, K.: A graphical approach to risk identification, motivated by empirical investigations. In: International Conference on Model Driven Engineering Languages and Systems, Springer (2006) 574–588
26. Prince, T., et al.: The common ontology of value and risk. In: submitted to the 37th International Conference on Conceptual Modeling (ER 2018), Xi'an. (2018)
27. PMI: A guide to the project management body of knowledge (PMBOK guide). Technical report, Project Management Institute (2013)
28. Chillarege, R.: Orthogonal defect classification. Handbook of Software Reliability Engineering (1996) 359–399
29. Wang, X., Mylopoulos, J., Guizzardi, G., Guarino, N.: How software changes the world: The role of assumptions. In: Tenth IEEE International Conference on Research Challenges in Information Science, RCIS 2016, Grenoble, France, June 1-3, 2016. (2016) 1–12
30. Fricker, S.A., Schneider, K., eds.: Requirements Engineering: Foundation for Software Quality - 21st International Working Conference, REFSQ 2015, Essen, Germany, March 23-26, 2015. Proceedings. Volume 9013 of Lecture Notes in Computer Science., Springer (2015)
31. Leveson, N.G., Turner, C.S.: An investigation of the Therac-25 accidents. Computer **26**(7) (July 1993) 18–41
32. Williams, C.: Intel's Pentium chip crisis: an ethical analysis. IEEE Transactions on Professional Communication **40**(1) (Mar 1997) 13–19
33. 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) 118–128
34. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE transactions on dependable and secure computing **1**(1) (2004) 11–33