An Ontological Analysis of Safety-Critical Software and Its Anomalies

Hezhen Liu[1,*], Zhi Jin[2,3], Zheng Zheng[4], Chengqiang Huang[1], and Xun Zhang[1]
[1]Reliability Technology Lab, Huawei Technologies Co., Ltd., Shenzhen 518129, China
[2]Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, 100871, China
[3]Institute of Software, School of Computer Science, Peking University, Beijing, 100871, China
[4]School of Automation Science and Electrical Engineering, Beihang University, Beijing 100191, China
{liuhezhen2, huangchengqiang, zhangxun}@huawei.com, zhijin@pku.edu.cn, zhengz@buaa.edu.cn
*corresponding author

***Abstract*—The progressively dominant role of software in safety-critical systems raise concerns about the software dependability. There are limited mature practices and guides for assessing software dependability and analyzing system-level hazards triggered by software anomalies. A problem is that faults, errors, and failures that represent software anomalies, albeit with different natures, are usually used indistinctly to predict software dependability, leading to unsolid results. The lack of such consensual conceptualization also leads to poor interoperability between supporting tools, and, consequently, difficulties in anomaly management and software maintenance. Anomaly analysis and management is more tough for safety-critical software due to its higher complexity and the safety-critical nature. The complex context of safety-critical software causes difficulties in determining the evolution/propagation path of software anomalies and the impact on system safety. To capture the nature of safety-critical software and support an understanding of mechanisms of software anomalies and associated hazards, we propose three reference ontologies: Safety-critical Software Ontology, Software Fault Ontology and Software-failure-induced Hazard Ontology, which are built based on international standards, guides, and relevant conceptual models. We also discuss the relationships among them. That will facilitate a better understanding of the software anomaly mechanisms and the design of intervening/mitigation solutions. We demonstrate how these ontologies can help analyze software problems of real-world safety-critical systems.**

***Keywords*—safety-critical software, dependability**

I. Introduction

Software is constantly growing with size and complexity and even dominant in modern safety-critical systems such as cars, power plants, medical devices, and flight control systems [1]. In some scenarios, software is safety- and mission-critical, and its errors may propagate and surface as dangerous behaviors of the system, resulting in loss of life and property and damage to the environment. The essential role of safety-critical software motivates a large amount of efforts into the assessment and improvement of its dependability.

As a type of artifact, software inevitably contains imperfections, flaws, or limits, which might cause failed operation or unexpected behaviors. The complex and abstract nature of software makes it often difficult to find the root causes when symptoms (i.e., failures or service terminations) occur. Some standards and guides [2]–[4] emphasize that it is important to manage various software anomalies (defects, faults, errors and failures etc.) in a structured manner; a well-established classification scheme can be used for various purposes including failure causal analysis and software process improvement (e.g., avoid defect insertion and/or support early defect detection).

However, many standards and reference models use different vocabularies to characterize these phenomena [4]–[6], resulting in inconsistent understandings of these concepts and ambiguous uses, causing many issues. For example, the software reliability growth model [7], [8], an usual methodology of software reliability assessment and prediction, is used based on the failure data. But misuses of defect/fault (i.e., flaws/risk causes) rather than failure data can lead to unreliable predictions [9]. The lack of consensual conceptualization also leads to interoperability issues between supporting tools in practice, resulting in poor usages of reference standards/models and difficulties in software maintenance [10].

Anomaly management becomes more tough for safety-critical software due to its higher complexity and the safety-critical nature. As suggested in [11], safety is not a property of the software itself but a combination of the software design and the environment where the software is located; safety is specific to the application, environment, and system. Software does nothing unsafe while could be unsafe only in the context of a safety-related system. NASA's software safety standard [12] provides more specific criteria to classify safety-critical software, such as the potential to cause a system hazard/condition/event or provide control or mitigation for a system hazards/condition/event.

From this viewpoint, we can infer that the safety-critical nature of software is determined by the assigned safety-related requirements and the corresponding functionality. To handle functionality related to system safety, software needs to incorporate operations of electrical/electronic and/or mechanical equipment such as sensors, controllers, and actuators. Safety-critical software is treated as an implementation element at a relatively low level of abstraction in a hierarchical safety-critical system, and its anomalies may propagate to the upper levels and finally trigger a dangerous condition (hazard) of the system. The hazard, together with a set of environmental conditions, may lead to an accident [11], [13]. To determine the impacts of unintended software behaviors from the safety perspective, a conceptual model is required to characterize artifacts related to safety-critical software and the complex context in which it exists and operates.

In this paper, we aim to address the issues mentioned

2693-9177/22/$31.00 ©2022 IEEE
DOI 10.1109/QRS57517.2022.00040

above by an ontological analysis. An ontological analysis can produce a set of formal descriptions (i.e., reference ontologies and associated operational versions) of domain knowledge. A reference ontology is a solution-dependent specification (i.e., conceptual model) with the purpose of providing a clear and precise description of domain entities that can be shared and reused across different communities; once consensual conceptualization is established, operational (machine-readable) versions of a reference ontology can be created [14], [15]. This study will focus on the development of reference ontologies related to safety-critical software and its anomalies. Our analysis will bring about three reference ontologies as main contributions – the Safety-critical Software Ontology (SCSO), the Software Fault Ontology (SFO), and the Software-failure-induced Hazard Ontology (SFIHO):

(1) SCSO captures the nature of safety-critical software from a requirements engineering perspective, which depicts how safety-critical software and other artifacts are organized at different levels of abstraction to achieve the top-level goals/requirements (including safety goals) of the system.
(2) SFO clarifies the different natures of fault, error, and failure, and explains the mechanisms by which the software fault activates and evolves into the system failure.
(3) SFIHO captures the entities (concepts and relations) that form a causal chain from a software failure to a hazard/accident.
(4) The combination of the three ontologies provides a conceptual framework for formalizing and managing hazard/accident experience.

These ontologies will be developed following the processes defined in the Systematic Approach for Building Ontologies (SABiO) [15]. To elicit consensual information, relevant standards, guides and literatures in the domain of systems and software engineering are analyzed. Particularly, these ontologies will be built based on a set of reference ontologies [10], [13], [16], by extending their concepts and relations while incorporating new entities. Furthermore, they will be grounded on the Unified Foundational Ontology (UFO) [17]–[20] to obtain real-world semantics.

The rest of the paper is organized as follows. Section II introduces the relevant ontology foundations and the SABiO method. Section III presents SCSO; Section IV presents SFO; Section V presents SFIHO. In Section VI, we evaluate the proposed ontologies through verification and validation processes. Finally, Section VII summarizes this work and presents some future prospects.

II. Background Knowledge

A. The Unified Foundational Ontology

A foundational ontology defines very common concepts across all domains, and it can support semantic interoperability to the domain-specific ontologies built upon it, by providing a common starting point for the formulation definition [21]. In this study, we adopt UFO as the foundation ontology, considering that it can provide a complete set of concepts that cover all the aspects of the three proposed ontologies. Besides, all the reused reference ontologies (see sections below) are grounded on UFO, and thus they can be reused directly without a reconstruction. A set of UFO concepts that are germane for the purpose of this study, including *Event*, *Situation*, *Disposition*, *Object*, *Function Universal* and *Function*, will be used. Detailed descriptions of these concepts have been presented in the literatures [10], [17]–[20]. Considering the limited space, we do not attempt to duplicate these descriptions here.

B. The Software Ontology

There are rich researches trying to clarify the natures of software and other similar entities like program, code, copy and algorithm [22]–[26]. [25] suggests that software (synonymous with program) is different from code since code may change while software maintains its identity; this study proposes that the identity criteria of software is derived from the artifactual nature. [26] later extends this discussion and adopts the notion of artifact proposed by [27], which treats artifacts as the results of intentional processes, and the identity of an artifact is connected to its proper function that it is (intentionally) designed to perform. [26] proposes an ontology – Ontology of Software Artifacts (OSA), which is built upon [28]'s requirements engineering model as follows: *Requirement* is defined as expected effects of a software system in an environment that surrounds a machine where the software system operates. A *Requirement* can be translated into part of a *Specification*, based on a set of domain assumptions. In contrast, a *Specification* describes the desired behaviors at the interface between the machine and the environment or that inside the machine. According to the requirements at different levels and the corresponding intended functions, OSA makes a distinction between four kinds of artifacts (i.e., software product, software system, program and code).

The Software Ontology (SwO) proposed in [16] is built based on OSA, and it extends part of the Software Artifact sub-ontology of the Software Process Ontology (SPO) [29]. It is grounded on UFO. This ontology also presents *Software System*, *Program*, *Code*, and the corresponding *System Requirements Specification* and *Program Requirements Specification* (see Figure 1). To explain how software is executed, SwO introduces the concept *Loaded Program Copy*, as materialization of *Program*. *Loaded Program Copy* is a subtype of *Disposition* that is manifested by a kind of *Event* [18], i.e., the execution of the *Loaded Program Copy* by a *Controller*. The *Loaded Program Copy* is constituted by software *Functions* that are instances of *Software Function Universals* described by the *Program Requirements Specification*. Thus, a *Program* runs as a *Program Copy Execution*, which produces correct results in conformance with the *Program Requirements Specification* if implementing the same set of *Software Function Universals* described by it.

SwO works well for characterizing the nature of software and how it is materialized and executed in a computer-driven system. However, SwO is a common solution but not specific

to safety-critical software. Safety-critical software is essentially software while critical to system safety. We still need an ontology model that captures both the software nature and the safety-critical property. SCSO will be built upon SwO while incorporating other concepts necessary for characterizing the safety-critical property. Some concepts of SwO will also be reused for developing SFO and SFIHO.

C. Ontological Interpretations of Software Anomalies and Hazard

In the literature, there are a few ontological interpretations of anomalies in engineering artifacts. [30] analyzes the notion of failure for engineering artifacts and defines three types of failures. For each type of failure, a failure is considered as an event that, once happens, will bring about a fault as a state of the artifact. Here the terminology "fault" has a notion different from that is conceived in our study as the initiator of anomalies that software may present. In [31], the authors perform an ontological analysis of fault process and propose an ontology of faults. This work focuses on the fault process and a complete vocabulary of faults, the other concepts (e.g., error, failure) have not been discussed.

[10] is the first ontological research focusing on software anomalies. The research delivers the Ontology of Software Defects, Errors and Failures (OSDEF). OSDEF provides an ontological interpretation for the two important concepts, vulnerability and failure, which represent, respectively, the cause and effect of software anomalies. However, it misses a concept that represents an incorrect intermediate state that may be bought about by the activation of a vulnerability. The notion of incorrect state is generally mentioned in widely accepted standards/guides/literatures (see Section IV). We will address this by adding the concept *Error* in SFO.

On the other hand, there are rich studies dedicating to conceptualization of *Hazard* and ontology-driven methodology for hazard/risk analysis [13], [32]–[36]. However, the conceptual models proposed in these works define only the universal notions that constitute *Hazard* and/or its cause/consequence. The associations between these notions and the anomalies and related entities of software have not been discussed. To target this problem, we propose SFIHO. The Hazard Ontology (HO; [13]) is selected as a core ontology to construct SFIHO because it provides a relatively complete concepts that constitute *Hazard* and its cause/consequence. Furthermore, it is also grounded on UFO so that it can be reused directly. The main concepts and relations of HO will be presented in Section V.

As discussed above, there are substantial reference conceptual models regarding the two aspects – software anomalies and system hazard. However, these models are not specific to the context of the safety-critical software/system, and thus the individual models have not been organized to form a systematic framework that can be utilized to manage and formalize *Hazard* experience resulting from software issues.

D. Systematic Approach for Building Ontologies

We follow the SABiO method [15] to develop the proposed ontologies. SABiO is an ontology engineering method that incorporates practices from software engineering. It provides activities that apply to develop domain reference ontologies, as well as the design and coding of operational ontologies. Several reference ontologies of the software engineering domain, such as SwO discussed above, were built using this method.

SABiO's development process comprises five main phases: (1) purpose identification and requirements elicitation; (2) ontology capture and formalization; (3) design; (4) implementation; and (5) test. These phases are supported by a set of processes, i.e., knowledge acquisition, reuse, configuration management, evaluation, and documentation. Since our goal is to build reference ontologies (conceptual models), we accomplish only the first two phases as suggested. We first talk with domain experts and review the existing standards, guides and literatures, to elicit functional requirements of each ontology. These requirements will be formulated as a set of Competency Questions (CQs; i.e., questions that each ontology should be able to answer). Guided by these CQs, we then identify and organize the concepts, relations, properties and axioms. Relevant knowledge are acquired from experts, as well as from standards, guides and reference models. The identified concepts and relations will be analyzed semantically and grounded on UFO. For illustration and communication purposes, Unified Modeling Language (UML) class diagrams are used to represent the conceptual models. Finally, the ontologies are evaluated by verification and validation processes.

III. Ontological Analysis of Safety-critical Software

As aforementioned, safety-critical software is essentially software while possessing the safety-critical property. Software becomes safety-critical only if it is in a safety-related system context. From the relevant literature [11], standards [12], [37]–[40] and guides [2], [3], [41], we know that safety-critical software is treated as an implementation element of a safety-critical system, and it is usually designed to handle safety-related functionality. The safety-critical nature is thus determined by the assigned safety requirements and safety-related functionality. We will perform an ontological analysis toward a safety-critical system that contains safety-critical software elements, from a requirements engineering perspective.

Following the SABiO method, we first formulate a set of CQs as the functional requirements of the ontology. To elicit the CQs, we have analyzed widely adopted guides [2], [3], [41] and international standards [37], [38] of systems and software engineering, and functional safety standards [39], [40]. The CQs are listed below:

- CQ1: What are the requirements specifications that exist for a project of a *Safety-critical System*?
- CQ2: How are these requirements specifications related?
- CQ3: What is a *Safety-critical System*? Which concepts indicate its safety-critical nature?

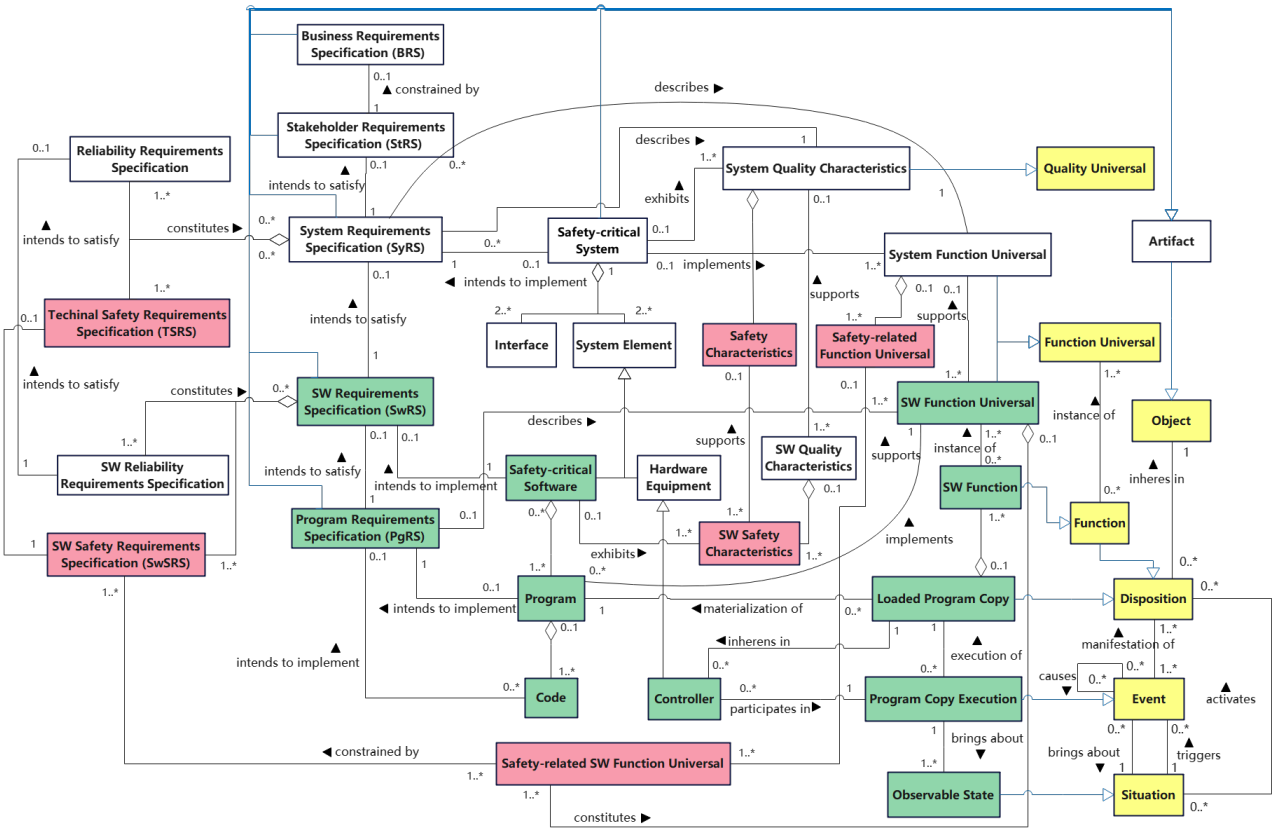

Figure 1. Conceptual model of SCSO. Each rectangle represents a concept. A single solid line represents an association with a labeled notion, and a nearby filled arrowhead indicates the direction in which the relationship works. A solid line with a hollow arrowhead represents an inheritance between two concepts; the arrowhead points from the child element (subclass) to the parent element (superclass). A solid line with a hollow diamond represents an aggregation where the diamond connects the containing class. Multiplicity labeled on each end of a relation is an indication of how many objects may participate in the given relationship. These representations also apply to the following figures. The concepts of UFO [17]–[20] are shown in yellow; the green ones show the concepts adopted from SwO [16]. The concepts in light red are the indicators of the safety-critical property of system/software (see the text for more details).

- CQ4: What is a *Safety-critical System* composed of?
- CQ5: What is *Safety-critical Software*? Which concepts indicate its safety-critical nature?
- CQ6: What is the relation between the requirements specification/functions of *Safety-critical Software* and those of the *Safety-critical System*?

The different natures of *Requirement* and *Requirement Specification* (as suggested by [28]) have been introduced in Section II-B. For simplicity, SCSO presents only a set of *Requirements Specifications* that exist for the domain of safety-critical system. A *Requirements Specification* represents a set of documents or models (*Artifacts*) that elaborate needs or formal requirements of a system/software artifact. Here *Artifacts* refer to objects intentionally developed to serve a given purpose in the context of a system development project [26], [27], meanwhile relevant domain assumptions are accounted for. From this viewpoint, a *Requirements Specification* is description of requirements based on a set of domain assumptions. A *Requirement Specification* may not be able to properly describe the requirements if making incorrect or incomplete assumptions [10].

The conceptual model of SCSO, represented as UML class diagrams, is shown in Figure 1. SCSO is centered on the concept of *Safety-critical System* (hereafter *System* for short). A *System* refers to an implementation solution of System-of-Interest (SOI); the latter exists in the problem space while the former exists in the corresponding solution space [41]. As presented in [37], *System* is generally defined as "a combination of interacting elements organized to achieve one or more stated purposes". We also consider Zave and Jackson' interpretation [28] that treats *System* as an interface between the machine and the environment; it is an artifact with manual, automated and even abstract (data) components, separating it from the concept of machine. From an ontological point of view, a *System* is a complex and heterogeneous artifact composed by many other artifacts that exist in different levels of abstraction. A *System* can be decomposed into several subsystems that can be further decomposed into smaller units until non-decomposable technological elements. For simplicity, SCSO presents only *System Element* and *Interface*; the instances of *Interface* are located between different *System Elements*. As a type of artifact, each *System Element* has its own identity and intended purposes.

A *System* is developed based on system requirements, which are generally classified into functional requirements and non-functional requirements. The functional requirements refer to *System Function Universals* that denote the types of functions expected for the *System*. The non-functional requirements define the criteria or capabilities of the *System* [16]; they can be quality requirements that refer to *Quality Characteristics* (e.g., reliability, safety, efficiency) that the *System* shall exhibit in some degree [16], [42]. All these system requirements are translated into models or textual requirements that are documented as a *System Requirements*

Specification (SyRS), taking into account assumptions about the machine and the environment. Therefore, the *SyRS* describes *System Function Universals* that the *System* should implement. It also defines *Quality Characteristics* that should be exhibited. A *System* is in conformance to its *SyRS* only if it implements (and exhibits) *System Function Universals* (and *Quality Characteristics*) the same as described by the *SyRS*.

There is a *Technical Safety Requirements Specification (TSRS)* that constitutes part of the *SyRS*. This concept is the same as *Technical Safety Concept* in ISO 26262 [40], i.e., "specification of the technical safety requirements and their allocation to system elements with associated information providing a rationale for functional safety at the system level". Therefore, a *TSRS* is specific to a system implementation scheme. The *TSRS* describes a set of *Safety-related Function Universals* and *Safety Characteristics*. For clarity, the *describes* relation is not presented in Figure 1. *Safety-related Function Universals* refer to *Function Universals* that (1) enable a *System* to achieve or maintain a safe state or degraded state or (2) may cause hazards once improperly implemented. *Safety Characteristic* is generally defined as safety integrity level (i.e., probability of a *System* satisfactorily performing the specified *Safety-related Function Universals*; see [39], [40]). It should be noted that, besides *TSRS*, there are other specifications constituting a complete *SyRS*.

A *SyRS* intends to satisfy a *Stakeholder Requirements Specification (StRS)*, which in turn, intends to satisfy a *Business Requirements Specification (BRS)*. *StRS* and *BRS* record high-level requirements of the SOI [41]. Note that both *BRS* and *StRS* subsume safety goals/requirements; for brevity, these notions are not presented in the conceptual model shown in Figure 1.

As a *System* is decomposed into a set of *System Elements* and *Interfaces*, the system requirements are also decomposed and allocated to a set of system element requirements. *Safety-critical Software* (hereafter *Software* for short) is also a complex artifact that intends to implement a *Software Requirements Specification (SwRS)* that formally defines Software (SW) requirements. As presented in SwO [16], *Software* (i.e., *Software System*) is constituted by *Programs* that implement *SW Function Universals*, which are described by a *Program Requirements Specification (PgRS)*. As a distinction, the *SwRS* determines the desired behaviors (including *Quality Characteristics*) at the *Interface* between the *Software* and other *System Elements*, while the *PgRS* determines the desired behaviors inside the *Software*. Due to the safety-critical property, the *Software* shall implement an *SW Safety Requirements Specification* (*SwSRS*; as part of the *SwRS*) that is refined from the *TSRS*. The *SwSRS* defines a set of *SW Safety Characteristics* and constraints on *Safety-related SW Function Universals*. The *Safety-related SW Function Universals*, as part of the *SW Function Universals* implemented by the *Program*, in turn support the *Safety-related Function Universals* of the *System*.

From the discussions above we know that the safety-critical property of *Software*/*System* is indicated by the concepts shown as light red in Figure 1. It would be a common system/software (not necessarily related to safety) if these concepts are removed from the conceptual model.

IV. Ontological Analysis of Software Fault, Error and Failure

From SCSO we learn that the construction of safety-critical software is related to a lot of artifacts. The correctness of safety-critical software depends on whether these artifacts are created based on correct assumptions and in a correct manner. Artifacts inevitably contain imperfections, flaws or limits because humans can make mistakes; even if they do not, software behaves correctly (i.e., properly implements the *SwRS*) only under nominal conditions. For safety-critical software, the underlying risks may be activated under out-of-nominal conditions and finally evolve into threats for system safety. Software Fault Ontology (SFO) is developed to facilitate an understanding of software anomaly mechanisms and the design of intervening/mitigation solutions towards safety issues of software.

To elicit relevant concepts, besides the aforementioned standards, guides and literatures, we have analyzed other materials including a guidebook of software dependability engineering [7], a paper that proposes concepts and taxonomy in the context of dependability [43], and relevant standards [4], [5], [44]. The vocabularies in these references have been closely investigated. As discussed in Section I, there is still lack of an agreement about the terminologies/concepts that represent software anomalies. The three terms, *Fault*, *Error* and *Failure*, are most frequently mentioned, however, with different notions. We select them as the core concepts of SFO. We will illustrate how the three concepts and their relations are organized to form the causal chain from a design flaw to the symptom for software. The requirements of this ontology are expressed as CQS listed below:

- CQ7: What is *Fault*?
- CQ8: What is *Error*?
- CQ9: What are the events that result in *Errors*?
- CQ10: What is *Failure*?
- CQ11: In which type of situations can a *Failure* occur?
- CQ12: What is the error propagation process?

The conceptual model of SFO is shown in Figure 2. Some concepts from SwO [16] and OSDEF [10] are reused. *Fault* is defined as a flaw that if executed/activated potentially results in an *Error*. *Faults* represent *Dispositions* that exist in *Loaded Program Copies* or *Hardware Equipment* (e.g., *Controllers*) that participate in *Program Copy Executions*. *Hardware Faults*, usually known as random *Faults*, can be activated at any time. In contrast, *Loaded Program Faults* are manifested only if the part of *Loaded Program Copy* containing *Faults* is executed. *Fault* can also be classified into *Development Fault* and *Usage-limit Fault*. A *Development Fault* refers to a flaw that tied to any software artifact (e.g., requirements specification, source code and architecture model); for example, it refers to a bug (i.e., *Code Fault*) if presented in *Code*. As mentioned in Section III, artifacts are built upon a set of domain assumptions. Throughout a software life cycle any incorrect assumptions

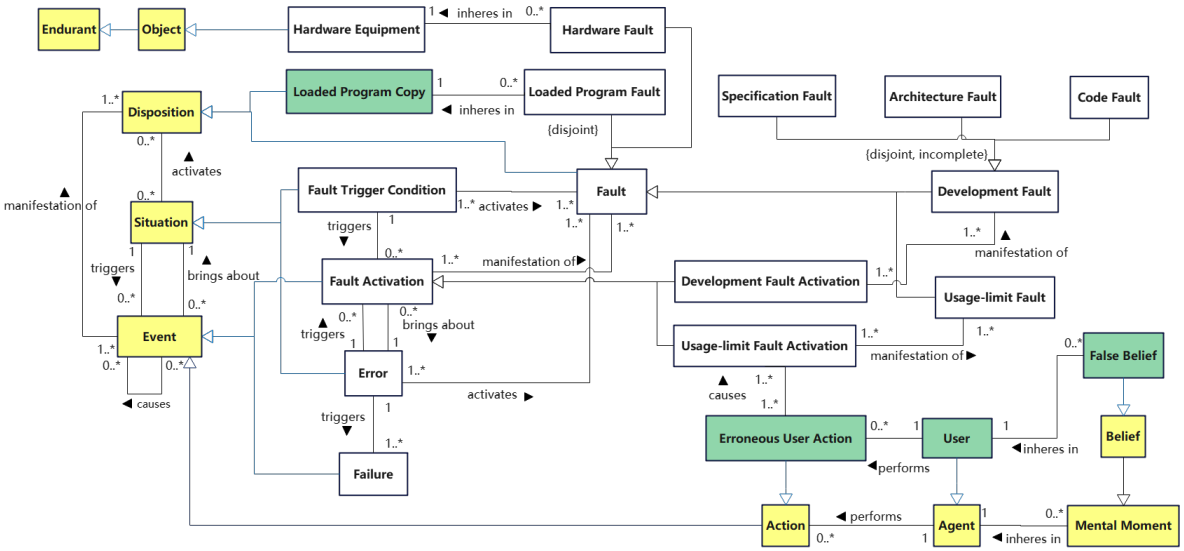

Figure 2. Conceptual model of SFO. The concepts of UFO are shown in yellow; the concepts shown in green are adopted from SwO [16] and OSDEF [10].

may result in *Development Faults*. However, in some cases when the assumptions are explicitly defined as disclaimers and usage guidelines while neglected by *Users* of system/software, *Usage-limit Faults* will be manifested [10].

A *Fault* is activated by a *Fault Trigger Condition* and manifested as a *Fault Activation*. *Fault Activation* has two subtypes – *Development Fault Activation* and *Usage-limit Fault Activation*, corresponding to the two types of *Faults*. Particularly, *Usage-limit Fault Activations* are caused by *Erroneous User Actions* performed by *Users* who have *User Malicious Intentions* or *False Beliefs* about the domain assumptions. We note that intentional malicious *Faults* are generally the focus of security but not of dependability [43]. To present a complete set of *Mental Moments* that cause *Erroneous User Actions*, we thus still include *User Malicious Intentions* in SFO.

A *Fault Activation* further brings about another *Situation*, i.e., *Error*. *Error* is defined as an incorrect state of an executed program that shows discrepancy between a computed, observed or measured value/condition and the true, specified or theoretically correct value/condition. As presented in SwO (see Figure 1), an *Event* of *Program Copy Execution* may bring about an *Observable State*. The *Fault Activation Event* refers to the execution of a fragment of program copy with *Faults*, which brings about an erroneous *Observable State* (*Error*). Such a *Situation* may work as another *Fault Trigger Condition* that activates other *Faults*, which is in turn manifested as another *Fault Activation*. It is the error propagation process. A *Failure* occurs only if an *Error* is observed to be far beyond a specified threshold.

As discussed in Section III, *Software* is a complex artifact designed to fulfill a certain goal, that is, to implement an *SwRS*. From the viewpoint of a *System*, a *Failure* of embedded *Software* indicates that the assigned *SwRS* is not satisfied, which has two unfolded interpretations: (1) the *Software* does not implement the *SW Function Universals* as described in the *PgRS*; (2) the *Software* properly implements the *PgRS*, which, however, does not satisfy the *SwRS*. The first type of *Failure* is caused by a wrong construction of the *Program* due to such as *Code Faults* and *Architecture Faults*. These *Faults* lead to the incorrect implementations of the (supposed-to-be-correct) *PgRS*. The latter type of *Failure* may result from *Specification Faults* that are introduced when inadequate or incorrect requirements are defined in the *PgRS*; alternatively, the *PgRS* is based on incorrect or incomplete machine assumptions that the developers have towards the programming platform [10]. Therefore, *Failure* in SFO refers to inability of *Software* to provide correct services (i.e., undertake a part of *System Function Universals*) specified by an *SwRS*. For safety-critical software, once it fails to properly implement the *SwSRS*, the *Failure* is likely to be a threat for system safety.

V. Ontological Analysis of Software-failure-induced Hazard and Accident

In the above sections, we first present SCSO that explains the nature of *Safety-critical Software*. We later build SFO that clarifies different natures of anomalies inside *Software*, so that the conceptual model can be used to explain why and how *Software* goes wrong. In this section, based on these two ontologies, we will develop a conceptual model that facilitates the analysis of safety issues induced by software *Failures*. Generally, safety analysis is performed through identifying hazards and potential causes/consequences [39], [40]. We will first analyze the entities (including *Situations*, *Events*, *Objects* etc.) that contribute to a hazard, from an ontological point of view. These entities will be later linked to the concepts of SCSO and SFO, to characterize how software anomalies contribute to system hazards.

The concept of *Hazard* has been extensively discussed in the literature [11], [39], [40] (also see Section II-C). Particularly, HO [13] treats *Hazard* as a combination of system and environment states (*Situations*) comprised by a set of necessary objects and harm factors. From the definitions we know that a *Hazard* can result from software failures, while environmental factors necessarily participate in the state. SFIHO aims to capture the concepts and relations that form a causal chain from a software failure to harm to the environment and people.

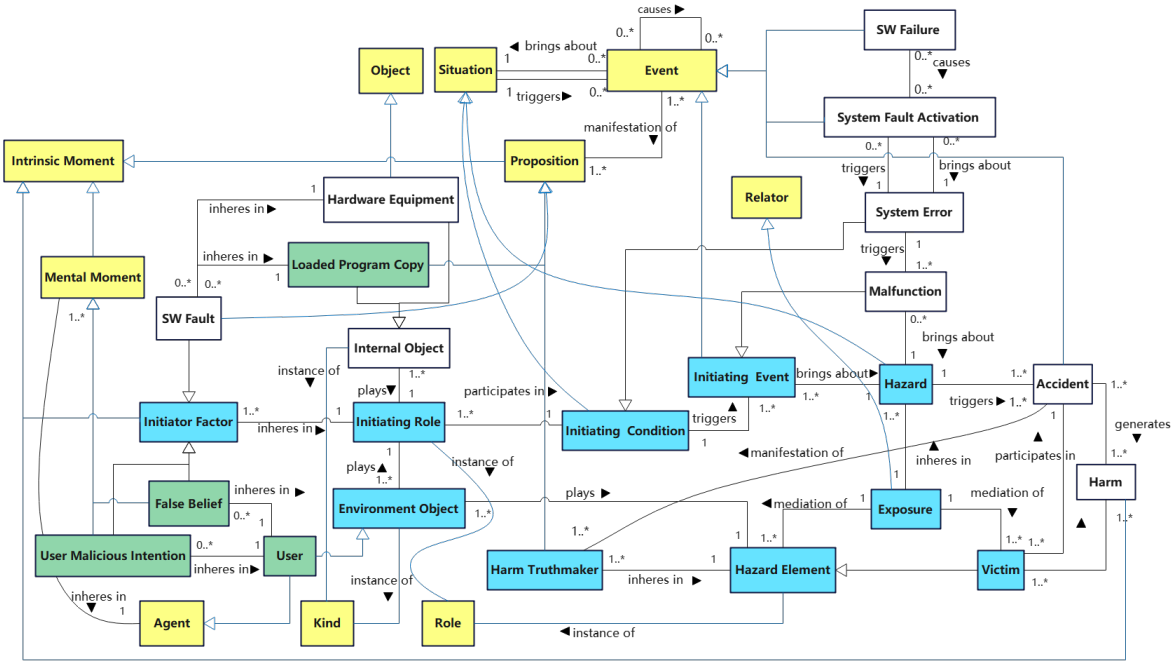

Figure 3. Conceptual Model of SFIHO. The concepts of UFO are shown in yellow; the concepts in green are adopted from SwO [16] and OSDEF [10]; the concepts in blue are from HO [13].

SFIHO will be built upon HO by extending its concepts and relations. SFIHO aims to answer the following CQs:

- CQ13: What is *Hazard*?
- CQ14: Which type of events can result in *Hazards*?
- CQ15: What is *Accident*? In which type of *Situations* can an *Accident* occur?
- CQ16: How does a *Software Failure* trigger a *Hazard*?
- CQ17: In the case of a software-failure-induced *Hazard*, which of the entities in SFO contribute to the *Hazard*?

The conceptual model of SFIHO is shown in Figure 3, where the classes in blue and their relations are mainly adopted from HO [13] with a few adjustments. *Hazard* is treated as a kind of *Situation*, and it comprises four types of *Entities*: (1) *Hazard Element* (as an instance of *Role*; e.g., a car); (2) *Harm TruthMaker* (as a subtype of *Disposition*; e.g., kinetic energy of a car) that inheres in *Hazard Elements*; (3) *Victim* (as a subtype of *Hazard Element*; e.g., a driver, a passenger); (4) *Exposure Relator* that represents a relation through which at least a *Victim* is exposed to the safety threats posed by *Hazard Elements* (e.g., "a man crossing a road is exposed to the threats of an out-of-control car"). When these necessary *Entities* exist and constitute a *Hazard*, the *Hazard* will trigger an *Accident* that brings about harm to people and the environment. In order to bring about such a *Hazard*, there must be an *Initiating Event* that is triggered by a prior *Initiating Condition*; the *Initiating Condition* comprises necessarily the *Initiating Roles* and *Initiating Factors*.

An *SW Failure* (i.e., *Failure* in SFO) is defined as an *Event* when *Software* losses the ability to provide specified services to a *System*. From the view of a *System* that contains *Software*, the *SW Failure* corresponds to an activation of a fault at the system level (i.e., *System Fault Activation*), which brings about an incorrect state (i.e., *System Error*) of the *System*. The *System Error* may further trigger a *Malfunction* of the *System*; alternatively, it activates another fault of the *System*, which is manifested as another *System Fault Activation*. As defined in [40], *Malfunction* is failure or unintended behavior of an *Item* with respect to its design intent, where *Item* refers to system or combination of systems that implements a function or part of a function that is observable by the customer (*User*). *Malfunctions* are thus associated with an item-level *System* that perform functions/behaviors in a real environment surrounding the machine where the *System* locates. For simplicity, SFIHO presents only limited classes. However, it should be kept in mind that a real item-level *System* may be decomposed to several levels of abstraction of *Systems*; an *SW Failure* probably propagates to the item-level *System* via several iterations of the *Fault Activation-Error-Failure* process at different levels.

A *Malfunction* could be an *Initiating Event* that brings about a *Software-Failure-Induced Hazard* (*SFI Hazard*). In this case, the *System Error* works as an *Initiating Condition*, and the *Initiating Role* is played by the *User* (as an *Environment Object*) of the *Software/System*. Considering there must be other objects (*Hardware Equipment* and/or *Loaded Program Copies*) that exist in the *System* and contain *SW Faults* participating in the *Initiating Condition*, we add the concept of *Internal Object*. The *SW Faults* are an *Initiator Factor*. When *Fault Activations* are caused by *Erroneous User Actions*, the *User*'s *False Beliefs* or *User Malicious Intentions* are another *Initiator Factor*. The *Environment Objects* including the *User* also play the roles of *Hazard Elements*. Sometimes the *User* is also a *Victim* when it is exposed to the *Hazard Element* that possesses the *Harm TruthMaker*. The *Hazard* can further trigger an *Accident*, as a manifestation of *Harm TruthMaker*, which will generate *Harm* to the *Victims*. *Harm* is *Intrinsic Moment* that can be physical/mental injury or death to people or damage to a system, equipment, or property.

The ontological interpretations of *Hazard*, *Accident* and associated concepts shed light on the basic principle for designing safety mechanisms. SFIHO indicates that it may

go through a set of transitions between *Situations* (i.e., *SW Error*, *Initiating Condition* and *Hazard*) and *Events* (i.e., *SW Failure*, *Initiating Event*) before an *Accident* occurs. From an ontological point of view, a certain prior *Situation* is necessary to trigger an *Event* [18]. One can thus prevent an *Event* from happening by breaking its prior *Situation*. To avoid an *Accident*, a *System* or its *User* can perform intervening actions when a *Hazard* is caught; alternatively, technical solutions are designed to transit the *Initiating Condition* to a safe *Situation* so that it cannot trigger the *Initiating Event*. Correspondingly, there are two types of safety mechanisms proposed in [40]: (1) when a *Fault* of an item-level system is activated and an *Error* state is observed, the first type of safety mechanism is able to transition to, or maintain the item in a safe state. (2) otherwise, once a *Malfunction* of an item is unavoidable and it triggers a *Hazard*, the second type of safety mechanism is able to alert the driver such that the driver is expected to control the effects of the *Malfunction*.

VI. Ontology Evaluation

In this section, we will follow the SABiO method and evaluate the proposed ontologies by conducting ontology verification and ontology validation processes.

A. Ontology Verification

Concerning ontology verification, the main objective is to ensure that the ontology is being built correctly, in the sense that there is no coherence and consistency issue and the output artifacts meet the previously defined specifications [15]. To achieve this goal, the verification can be done in a competency-question-driven manner. A suggested method is to create a table that indicates which ontology elements (concepts, relations, and axioms) are able to answer each CQ. Tables I show the verifications of SCSO, SFO and SFIHO regarding the predefined CQs. All the three proposed ontologies are able to adequately respond to all the CQs. The verifications are thus considered successful.

TABLE I
Ontology verifications based on Competency Questions

CQ	Concepts and Relations
CQ1	Requirements specifications from high- to low-level are *BRS*, *StRS*, *SyRS*, *SwRS* and *PgRS*.
CQ2	An *StRS* is constrained by a *BRS*; an *SyRS* intends to satisfy an *StRS*; an *SwRS* intends to satisfy an *SyRS*; a *PgRS* intends to satisfy an *SwRS*.
CQ3	A *Safety-critical System* is an *Artifact* that intends to implement an *SyRS* including a *TSRS*. The concepts of *TSRS*, *Safety Characteristic* and *Safety-related Function Universal* indicate its safety-critical nature.
CQ4	A *Safety-critical System* is composed of a set of *System Elements* and *Interfaces* between them.
CQ5	*Safety-critical Software* is a *System Element* that constitutes a *Safety-critical System*. It is an *Artifact* that intends to implement an *SwRS* including an *SwSRS*. The concepts of *SwSRS*, *SW Safety Characteristic* and *Safety-related SW Function* indicate its *Safety-critical* nature.
CQ6	An *SwRS* intends to satisfy an *SyRS*, meanwhile a set of *SW Function Universals* (described by the *PgRS*) support the *Function Universals* (described by the *SyRS*) of the *System*.
CQ7	*Fault* is a subtype of *Disposition* that inheres in an *Endurant*. A *Loaded Program Fault* inheres in a *Loaded Program copy*. A *Hardware Fault* inheres in *Hardware Equipment*.
CQ8	*Error* is a subtype of *Situation*. An *Error* triggers a *Failure* or, alternatively, activates a *Fault* and triggers a *Fault Activation*.
CQ9	*Fault Activation*, as a manifestation of a *Fault*, brings about an *Error*. A *Development Fault Activation* is a manifestation of a *Development Fault*. A *Usage-limit Fault Activation* that is caused by an *Erroneous User Action*, is a manifestation of a *Usage-limit Fault*.
CQ10	*Failure* is a subtype of *Event*.
CQ11	A *Failure* is triggered by an *Error*.
CQ12	An *Error* brought by a *Fault Activation* activates another *Fault* somewhere and triggers another *Fault Activation*, which further brings about another *Error*.
CQ13	*Hazard* is a subtype of *Situation* that is constituted by a set of *Entities – Hazard Element*, *Harm TruthMaker*, *Victim*, and *Exposure*.
CQ14	*Initiating Event* is a subtype of *Event* that can bring about *Hazards*.
CQ15	*Accident* is a subtype of *Event* that generates *Harms* to *Victims*. When necessary *Entities* exist and constitute a *Hazard*, the *Hazard* triggers an *Accident*.
CQ16	A *Software Failure* corresponds to a *System Fault Activation*, which brings about a *System Error*. The *System Error* further triggers a *Malfunction*. The *Malfunction*, as a subtype of *Initiating Event*, brings about an *SFI Hazard*.
CQ17	*Hardware Equipment* or a *Loaded Program Copy*, as an *Internal Object* that contains *SW Faults*, plays an *Initiating Role* that participates in an *Initiating Condition*. The *SW Faults* work as an *Initiator Factor* that inheres in the Initiating Role. Meanwhile, the *User* of software/system, as an *Environment Object*, also plays an *Initiating Role*. The *False Believes* or *User Malicious Intentions* of the *User* can be another *Initiator Factor*.

B. Ontology Validation

In this section, we will validate that the right ontologies have been developed, that is, to demonstrate that the ontologies fulfill the intended purpose: (1) managing the domain knowledge of safety-critical software/system; (2) supporting the practical analysis of software anomalies and their impacts on safety. We follow the SABiO method and adopt a real-world case the same as one presented in [10], to investigate if the built ontologies can be instantiated to represent the selected case.

The selected case is the Patriot Missile Battery Intercept Failure [45]. On February 25, 1991, during the Gulf War, a Patriot Missile Battery's incorrect tracking of an Iraqi Scud missile resulted in the death of 28 soldiers (*Harm*). A later investigation revealed that a software *Fault* is responsible for the failed intercept. The Patriot system was designed to track and shoot down Soviet missiles including Scuds (*System Function Universal*), and the embedded software is mainly used for missile tracking and intercept decision (*Software Function Universal*). The system was specified to operate for a few hours at a time to avoid detection. However, at the time of this accident, it had been operated for about 100 hours (*Incorrect User Action*). The radar had successfully detected the incoming Scud. However, a *Failure* occurred in determining the next location of the missile by the *Program*, for which the velocity of missile and the time of the last radar detection are necessary. The time was measured with a precision of tenths of a second, and it was truncated at 24 bits (*Fault Activation*) due to a 24-bit fixed-point register (*Usage-limit Fault*). The inaccuracy of the measure time and the subsequent track length (*Error*) grew with the increasing time. The program thus returned a wrong predicted location of

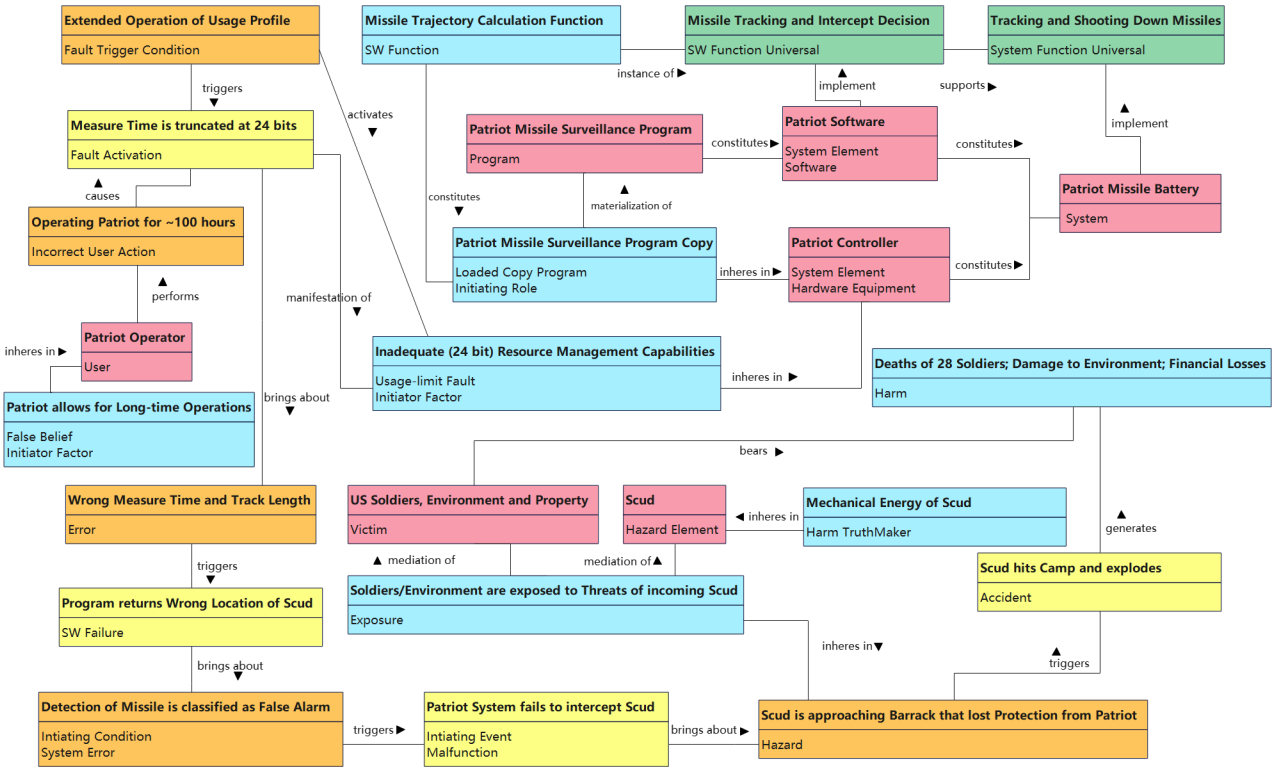

Figure 4. Patriot Missile Battery intercept failure as an instance of SCSO, SFO and SFIHO. Each box represents an instance; each solid line with an annotation represents a relation; in the lower partition of each box the classes from SCSO, SFO or SFIHO are presented. The UFO concepts inherited by the classes are represented with a color code: orange represents situations; yellow–events; blue–intrinsic moments; light red–agents/objects; green–function universals.

the missile (*SW Failure*). The initial detection of the missile is treated as a false alarm and removed (*System Error*), and no intercept was attempted (*Malfunction*). The US Army Barracks thus lost the protection of the Patriot system while the Scud was approaching (*Hazard*). The Scud finally hit the barracks and exploded (*Accident*).

As shown in Figure 4, the case introduced above is represented as an instantiation model of SCSO, SFO and SFIHO. By connecting the anomalies inherent in the *System* to the *Situations/Events* prior to the *Hazard*, the model establishes a causal chain from a root cause (software *Fault*) to symptoms (*Hazard/Accident*). The instantiation process thus provides a more exhaustive interpretation of the *Hazard* experience than that of an analysis based solely on HO [46]. Besides, compared to the instance model presented in [10], our model identifies the *Hazard/Accident* at the system level and associated entities. In a nutshell, our model captures a more complete set of entities that contribute to the final *Hazard/Accident*, including those inherent to the system and the necessary environmental objects. Such an instantiation process can work as an example template of a hazard analysis. The combination of the three ontologies thus provide a structured framework to formalize and manage the real-world *Hazard* experience.

VII. Conclusion and Future Work

In this study, we propose the three reference ontologies, SCSO, SFO and SFIHO. The first ontology characterizes the safety-critical software and associated artifacts in the safety-critical system, from a requirements engineering perspective. The second ontology is about various software anomalies. The last one is about *Hazard* and related entities of software that contribute to it. The concepts and relations involved in the ontologies are elicited from a set of international standards and scientific literatures. The ontologies are grounded on Unified Foundational Ontology so that they obtain real-world semantics. The contributions of the three ontologies proposed in this paper to conceptual modelling and the dependability analysis of safety-critical software are summarized as follows.

Firstly, the three ontologies clarify relevant concepts and relations and convey the fundamental principles and basic functionality of the safety-critical software and its anomaly mechanisms. The ontologies provide a common vocabulary that can be shared among different communities, improving communication and avoiding misunderstanding among engineers and stakeholders of safety-critical systems. Besides, the conceptual models provide a point of reference for the developers of safety-critical software/systems to systematically extract and analyze safety requirements specifications. Furthermore, the three ontologies, together as a framework, provide conceptual basis and a systematic guide to the dependability analysis of the safety-critical software. It has been confirmed in Section VI-B that through the ontology instantiation process, the software-related hazard causes and consequences and the other entities that participate in/contribute to the hazard are well identified. The ontologies can support the development of tools that apply to configuration and management of the safety-critical software, anomaly tracking and hazard experience management.

As future work, more industrial case studies are required to further evaluate and improve the ontologies. Besides, an important application of the conceptual models is to support

an automated hazard analysis at the early phase of the development of a safety-critical system. To do that, operational versions of these ontologies implemented in machine-readable language (e.g., OWL) will be established to provide concepts and annotations to system configuration, to further support anomaly data management. These operational ontologies will provide basis to an analysis framework that can be used for fault generation and error propagation inference (e.g., [47]); a further hazard analysis can be implemented once the environmental objects are modelled.

REFERENCES

[1] C. Hagen, S. Hurt, and J. Sorenson, "Effective approaches for delivering affordable military software," *Real-time Information Assurance*, vol. 26, pp. 26–32, 11 2013.

[2] P. Bourque and R. E. Fairley, *The guide to the Software Engineering Body of Knowledge (SWEBOK(R)): V.3.0*. IEEE Computer Society Press, January 2014.

[3] C. P. Team, "Cmmi for development, version 1.3," Tech. Rep. CMU/SEI-2010-TR-033, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2010.

[4] "Standard classification for software anomalies," *IEEE 1044:2010*, pp. 1–23.

[5] "Iso/iec/ieee international standards - systems and software engineering–vocabulary," *ISO/IEC/IEEE 24765:2017*, pp. 1–541.

[6] "Systems and software engineering–systems and software assurance–part 1: Concepts and vocabulary," *IEEE 15026:2014*, pp. 1–34.

[7] M. R. Lyu, *Handbook of Software Reliability Engineering*. IEEE Computer Society Press, 1996.

[8] N. Ullah, M. Morisio, and A. Vetro, "A comparative analysis of software reliability growth models using defects data of closed and open source software," in *2012 35th Annual IEEE Software Engineering Workshop*, pp. 187–192, 2012.

[9] R. Butler and G. Finelli, "The infeasibility of quantifying the reliability of life-critical real-time software," *IEEE Transactions on Software Engineering*, vol. 19, no. 1, pp. 3–12, 1993.

[10] B. B. Duarte, R. de Almeida Falbo, G. Guizzardi, R. Guizzardi, and V. E. S. Souza, "An ontological analysis of software system anomalies and their associated risks," *Data & Knowledge Engineering*, vol. 134, p. 101892, 2021.

[11] N. Leverson, *Safeware: System Safety and Computers*. Boston: Addison-Wesley, 1995.

[12] NASA-STD-8719.13, "Nasa software safety guidebook: Nasa technical standard," *Department of Defense*, 2004.

[13] J. Zhou, K. Hänninen, K. Lundqvist, and L. Provenzano, "An ontological interpretation of the hazard concept for safety-critical systems," in *The 27th European Safety and Reliability Conference*, June 2017.

[14] G. Guizzardi, "On ontology, ontologies, conceptualizations, modeling languages, and (meta)models," in *DB&IS*, 2006.

[15] R. de Almeida Falbo, "Sabio: Systematic approach for building ontologies," in *ONTO.COM/ODISE@FOIS*, vol. 1301 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2014.

[16] B. Duarte, A. Leal, R. Falbo, G. Guizzardi, R. Guizzardi, and V. Silva Souza, "Ontological foundations for software requirements with a focus on requirements at runtime," *Applied Ontology*, vol. 13, pp. 1–33, April 2018.

[17] G. Guizzardi, *Ontological foundations for structural conceptual models*. PhD thesis, University of Twente, Oct. 2005.

[18] G. Guizzardi, G. Wagner, R. de Almeida Falbo, R. S. S. Guizzardi, and J. P. A. Almeida, "Towards ontological foundations for the conceptual modeling of events," in *Conceptual Modeling*, pp. 327–341, Springer Berlin Heidelberg, 2013.

[19] A. Benevides, J.-R. Bourguet, G. Guizzardi, R. Peñaloza, and J. Almeida, "Representing a reference foundational ontology of events in sroiq," *Applied Ontology*, vol. 14, pp. 1–42, 07 2019.

[20] G. Guizzardi, A. Benevides, C. Fonseca, D. Porello, J. Almeida, and T. Prince Sales, "Ufo: Unified foundational ontology," *Applied Ontology*, vol. 17, pp. 167–210, January 2022.

[21] Wikipedia contributors, "Upper ontology — Wikipedia, the free encyclopedia," 2022. [Online; accessed 21-June-2022].

[22] P. Suber, "What is software," *J. Specul. Philos.*, vol. 2, no. 2, pp. 89–119, 1988.

[23] A. H. Eden and R. Turner, "Problems in the ontology of computer programs," *Appl. Ontol.*, vol. 2, no. 1, pp. 13–36, 2007.

[24] D. Oberle, S. Grimm, and S. Staab, *An Ontology for Software*, pp. 383–402. 05 2009.

[25] N. Irmak, "Software is an abstract artifact," *Grazer Philos. Stud.*, vol. 86, no. 1, pp. 55–72, 2013.

[26] X. Wang, N. Guarino, G. Guizzardi, and J. Mylopoulos, "Towards an ontology of software: a requirements engineering perspective," in *Formal Ontology in Information Systems*, vol. 267, pp. 317–329, September 2014.

[27] L. R. Baker, "The ontology of artifacts," *Philos. Explor.*, vol. 7, pp. 99–111, Jun 2004.

[28] P. Zave and M. Jackson, "Four dark corners of requirements engineering," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, pp. 1–30, jan 1997.

[29] A. Bringuente, R. Falbo, and G. Guizzardi, "Using a foundational ontology for reengineering a software process ontology.," *Journal of Information and Data Management*, vol. 2, pp. 511–526, January 2011.

[30] L. D. Frate, "Preliminaries to a formal ontology of failure of engineering artifacts," *FOIS*, pp. 117–130, 2012.

[31] Y. Kitamura and R. Mizoguchi, "An ontological analysis of fault process and category of faults," *Tenth International Workshop on Principles of Diagnosis (DX-99)*, pp. 118–128, 1999.

[32] T. Mahmood, E. Kazmierczak, T. Kelly, and D. Plunkett, "Modeling and learning interaction-based accidents for safety-critical software systems," in *14th Asia-Pacific Software Engineering Conference (APSEC'07)*, pp. 175–182, 2007.

[33] O. Daramola, T. Stålhane, G. Sindre, and I. Omoronyia, "Enabling hazard identification from requirements and reuse-oriented hazop analysis," in *2011 4th International Workshop on Managing Requirements Knowledge*, pp. 3–11, 2011.

[34] R. Winther and W. Marsh, "Hazards, accidents and events—a land of confusing terms," 2013.

[35] A. Lawrynowicz and I. Lawniczak, "The hazardous situation ontology design pattern," in *WOP*, 2015.

[36] A. Parisaca Vargas and R. Bloomfield, "Using ontologies to support model-based exploration of the dependencies between causes and consequences of hazards," in *Proceedings of the International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*, p. 316–327, SCITEPRESS, 2015.

[37] "Iso/iec/ieee international standards - systems and software engineering – system life cycle processes," *ISO/IEC/IEEE 15288:2015*, pp. 1–118.

[38] "Iso/iec/ieee international standard - systems and software engineering – life cycle processes – requirements engineering," *ISO/IEC/IEEE 29148:2018*, pp. 1–104.

[39] "Functional safety of electrical/electronic/programmable electronic safety-related systems," *IEC 61508:2010*.

[40] "Road vehicles — functional safety — part 1: Vocabulary," *ISO 26262:2018*.

[41] SEBoK, *Guide to the Systems Engineering Body of Knowledge (SEBoK) v.2.5*. Hoboken, NJ: The Trustees of the Stevens Institute of Technology, 15 October 2021.

[42] E. Y. Lawrence Chung, Brian A. Nixon and J. Mylopoulos, "Non-functional requirements in software engineering," Springer New York, NY, 2000.

[43] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[44] "Iso/iec/ieee international standard - systems and software engineering–systems and software assurance –part 1:concepts and vocabulary," *ISO/IEC/IEEE 15026-1:2019(E)*, pp. 1–38.

[45] P. M. Defense, "Software problem led to system failure at dhahran, saudi arabia," *US GAO Reports, report no. GAO/IMTEC-92-26*, 1992.

[46] J. Zhou, K. Hänninen, K. Lundqvist, and L. Provenzano, "An ontological approach to identify the causes of hazards for safety-critical systems," in *2017 2nd International Conference on System Reliability and Safety (ICSRS)*, pp. 405–413, 2017.

[47] X. Diao, M. Pietrykowski, F. Huang, C. Mutha, and C. Smidts, "An ontology-based fault generation and fault propagation analysis approach for safety-critical computer systems at the design stage," *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, vol. 36, p. e1, 2022.