

Orthogonal Defect Classification-based Ontology Construction and Application of Software-hardware Integrated Error Pattern of Software-intensive Systems

Xuan Hu, Jie Liu

Information Security Research Center, CEPREI, No. 110, Dongguanhuang Road, Guangzhou, China

Key laboratory of Ministry of Industry and Information Technology, No. 110, Dongguanhuang Road, Guangzhou, China

huxuan@ceprei.com, 47507346@qq.com

Abstract—Orthogonal defect classification (ODC) is a multi-dimensional measurement system with both qualitative and quantitative characteristics. And it is currently widely used in the software industry. However, its high level of abstraction leads to limited semantic information. Therefore, it seems to have a limited role in the process of software engineering of software-intensive systems (SISs). To solve this problem, this paper first analyzes software error lifetime from the perspective of knowledge-based software engineering and proposes an error generation model. Then, the paper proposes the concepts of software error pattern (SEP) and software requirements error pattern (SREP) based on the ODC. Then, according to an error generation mechanism, four types of software-hardware integrated error pattern (SHIEP) in the requirement stage, which is a sub-category of SREP, and corresponding ontology representation are given, focusing on “scenario”, “error manifestation” and “solution”. Finally, this paper takes a certain type of airborne radar software system as an example, uses protégé to edit the SHIEPs and instances, and further introduces the application of software FMEA based on the above work. The results show that the prior information based on the SHIEPs is helpful to discover potential failures and failure modes that may adversely affect the function or performance of SISs. Therefore, the proposed SHIEP is of great significance for improving the quality of software development and verification.

Keyword—orthogonal defect classification, ontology, pattern, software error

I. INTRODUCTION

HOW to provide developers with fast and effective feedback in a controlled and measurable way is a

problem that plagues the software industry [1]. There are two common forms of software defect analysis: statistical defect models and root cause analysis. The first is a quantitative method with good repeatability, but cannot provide feedback to developers timely in the way of available process control. The second is a qualitative method focusing on process details and therefore cannot be applied as an engineering method to full process control [1]. Orthogonal defect classification (ODC) is a technique that can bridge the gap between quantitative methods and qualitative analysis [2]. It extracts semantic information in defects via classification, converting what is semantically very rich into a few vital measurements on the product and process. Thus, essentially it is a multi-dimensional measurement system that transforms qualitative information into quantitative measures [1].

There are some researches on ODC recently, e.g., the applications of “defect triggers” during a testing process [3] and the researches on Bohr-Mandel errors using the “defect triggers” [4], the researches on ODC applications of safety, critical errors of spacecraft systems [5], [6], the researches on modeling Y2K errors through ODC [7], the researches on a defect tracing system based on ODC [8], the researches on automated generation of ODC [9], [10], the researches on in-process usability problem classification, analysis and improvement based on ODC [11]. In all, the representative researches on ODC tend to focus on its macroscopic characteristic as a measurement system. However, it is too abstractive to reflect individual gaps because of the consistency across phases and the uniformity across products [2]. Therefore, it can measure software processes only and cannot guide verification directly. In addition, it is difficult to apply ODC to the application scenario of software-intensive systems (SISs). Nowadays, SISs have become a developmental direction of large-scale systems, and the application ranges of SISs involve the embedded systems for automotive industry and aerospace applications, the dedicated systems for wireless communications [12]. A software product always stems from a defined requirements set. For SISs, the formation of a requirements set is a long-term and complicated process, and a main reason is: two problems of system complexity and sudden behavior are increasingly highlighted in the application process of SISs.

Manuscript received on Dec. 2, 2019. This work was supported by the National Key Research and Development Program of China (Grant ID: 2019YFC0118800), and a follow-up of the invited journal to the accepted & presented paper entitled “The Orthogonal Defect Classification-based Software Error Pattern Ontology Construction Researches” of the 22th International Conference on Advanced Communication Technology (ICACT2020).

Xuan Hu, she is a senior engineer of CEPREI Laboratory, China. (E-mail: huxuan@ceperi.com)

Jie Liu, he is the director of the Information Security Research Center of CEPREI Laboratory, China. (corresponding author, email: 47507346@qq.com)

The root cause of these two problems lies in the uncertainty of environment and the unpredictability of operation and scenario. ODC is on a higher level and has limited semantic information. Thus, it is difficult to describe the complicated scenarios of SISs via ODC. And it hinders the researches on the above problems through ODC.

Experience has shown that some abnormal events that occur under certain extreme conditions, although they have a low probability of occurrence, are extremely harmful. These problems need to be collected through long-term accumulation. In addition, there is a kind of errors depending on both operating time of hardware and operating state of software. Thus, one important source of an SISs requirements set is the error set obtained according to experience.

The rest of this paper is divided into the following sections: section 2 analyzes the lifetime of software errors and proposes an error generation model from the viewpoint of knowledge-based software engineering. Section 3 and section 4 present the definitions of software error pattern (SEP) and software requirements error pattern (SREP) based on ODC, respectively. Section 4 also describes four types of software-hardware integrated error pattern (SHIEP) according to an error generation mechanism, and constructs an SHIEP ontology (SHIEPO) by an object-oriented ontology representation method focusing on “scenario”, “error manifestation” and “solution”. In the knowledge engineering community, an ontology is a formal and explicit specification of a shared conceptualization [13]. This work unifies the SHIEP concept in the ontology framework. Section 5 presents a case study, taking an airborne radar software system as an example to introduce the application of above researches on software failure mode and effects analysis (SFMEA). Finally, section 6 concludes the study.

II. LIFETIME AND GENERATION MODEL OF SOFTWARE ERRORS

Software is a sequence of computer knowledge that describes human knowledge and the correct usage of human knowledge [14]. And it can be regarded that software errors are the reflection of inconsistency between the knowledge produced in the process of creating a software product and the original knowledge, and such reflection is regarded as necessary adjustments [15]. Fig. 1 shows that software error lifetime runs throughout developing, verification and field usage. The reliability test at the verification stage refers to a test type for hardware systems. It is a necessary means to improve product design, and evaluate and assess product quality characteristics during product development and production. There are several forms of incorrect software knowledge, (1) incorrect knowledge points or implicit software errors induced by man-made mistakes; (2) using correct knowledge in a wrong way [14].

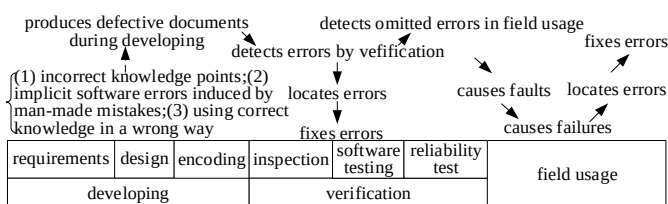


Fig. 1. Software error lifetime

Fig. 2 shows an error generation model. It is noted that these errors do not include system software errors and operator errors. This figure shows that a software error is generated during the process of software developing. Knowledge problems include domain knowledge problems and software engineering knowledge problems. Actions and artifacts vary from one stage to another. In horizontal direction, in-process errors may occur at various stages of software development. In vertical direction, knowledge inconsistencies between adjacent and non-adjacent layers may lead to inter-process errors. Software errors exist in artifacts and are expressed in a certain form. The intra-layer and inter-layer causes of errors are the scenarios of SEPs. The scenarios can be generalized, such as in-layer or inter-layer inconsistencies; they can also be relatively specific in combination with specific patterns.

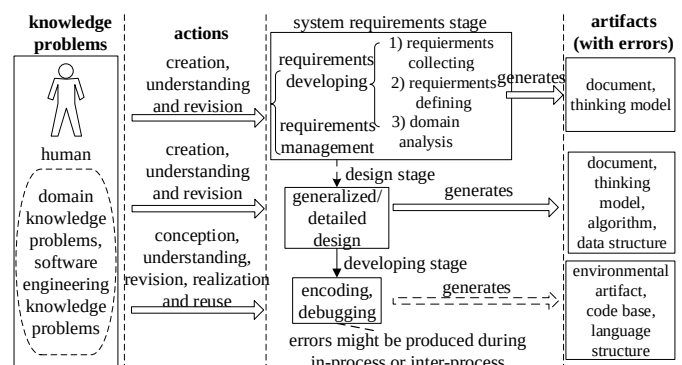


Fig. 2. Software error generation model

III. SOFTWARE ERROR PATTERN

The above researches show the software error lifetime throughout the software developing, verification and field usage. Therefore, the related researches should consider the multiple viewpoints of developers, testers, and users.

ODC is an error classification method that helps to obtain error information at a higher level. Its elements include “defect triggers”, “defect types”, “targets”, etc. They are independent and orthogonal with each other. A “defect trigger” is a condition that allows a defect to surface (**What**). The “defect type” reflects the nature of errors and guides how to amend them (**How**). Experience shows that, (1) the contexts in which errors are generated are similar, and the effect is similar to the “defect triggers” attribute; (2) errors have similar manifestations and can be modified, i.e., there is a software implementation that satisfies the correct intent. The effect is similar to the “defect types” and “targets” attributes. These inspire the paper to study software errors with the help of the concept “Pattern” [16].

Definition 1 The SEP refers to the error produced in the software development stage, which occurs repeatedly in a specific error lifetime scenario, and may cause a system (component) to fail to perform the expected function or affect the maintainability of the system. Such errors are general and common in a specific scenario and can be corrected by various means. SEP’s components can be instantiated.

Therefore, SEP = < Scenario, ErrorManifestation, Solution, Severity, DetectionStage, GenerationStage, Instance >.

This definition distinguishes the generation and survival

stages of errors. Furthermore, it is multi-viewpoint. Its core components are: “Scenario”, “ErrorManifestation” and “Solution”. “Scenario” reflects tester and user viewpoints, and is the basis for test cases and the catalyst that causes a failure. “ErrorManifestation” and “Solution” reflect developer viewpoint, express the nature of errors, and give a way to eliminate the errors. SEP also has other components, such as, “Severity”, “DetectionStage”, “GenerationStage” and “Instance”. An “instance” is the result of mapping a SEP concept at a higher level of abstraction to a specific case. The components of SEP are shown in Fig. 3. Most error classification systems use 5 to 15 defect attributes [1]. The number of SEP components is in this range. The SEP components are independent and orthogonal with each other. Thus, SEP can be regarded as a point in Cartesian space.

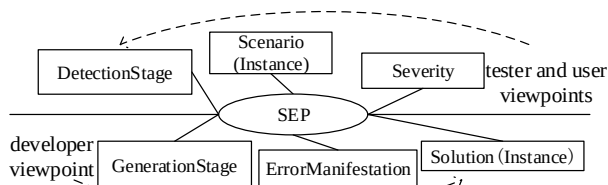


Fig. 3. SEP Components

Fig. 4 presents SEP and its components value sets in the form of classes based on experience and related references [1], [17]. Both SEP and its components can be regarded as classes. The relationships between SEP and the components are aggregation relationships. The “Scenario” class has 6 sub-classes. The “SEP” class has 3 operations of “measurement”, “software development”, and “software

verification”. The operation of “Scenario”, “ErrorManifestation”, “DetectionStage”, “GenerationStage”, and “Severity” classes is “1-dimension measurement” in common. The sub-classes of “Scenario” class have the operation of “measure data supply” in common. The values of components are the attributes of the corresponding classes. These classes can be instantiated, corresponding to the “Instance” of the components of SEP. The role of the operation “measurement” is similar to the metric of ODC attributes, i.e., the semantic information in errors is extracted by classification and converted into a metric. For example, the distribution and trends of various types of errors at each stage can be analyzed, and the deviation from an expected situation can indicate problems in a development process. This provides timely feedback to developers and can guide bug fixes. The operation “software development” is given from the developer viewpoint; while the operation “software verification” is given from the tester and user viewpoints. In addition, the difference between the attribute “software workload change” and the attribute “system workload change” is that the “software workload” directly affects the software product itself, exposing software product defects by imposing the behaviors that may result in exceeding software product resource limits or capacity limits, whereas the “system workload” directly affects the entire system, and generally has an indirect effect on software, such as the vibrations generated during an aircraft flight. The “safety error” of “ErrorManifestation” refers to the error that may cause failures of casualties, property damage or environmental damage. Such errors are very important in the SISs.

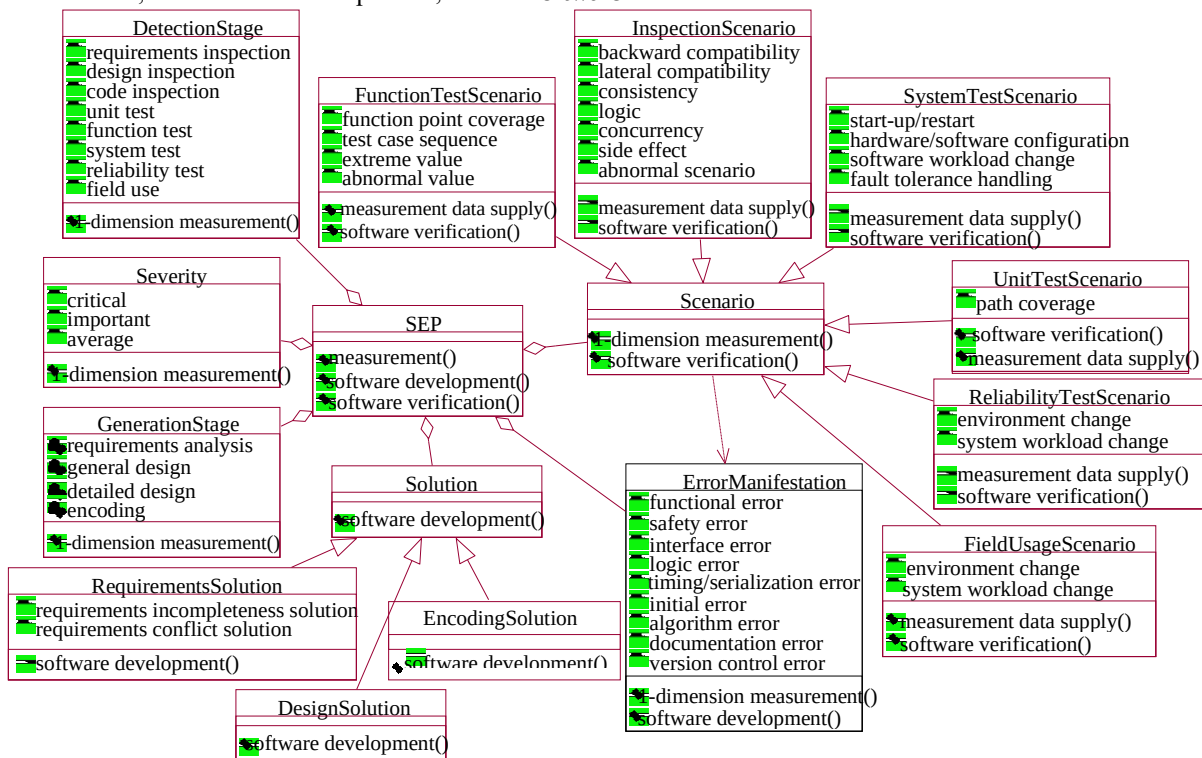


Fig. 4. Class representation of SEP and its components

IV. SREP ONTOLOGY

4.1 Ontology structure of SREP

SEP-related domains include the corresponding domains in the stages of requirements, design, and coding according to the “error generation stage”. And the results of a follow-up survey of 8,000 software projects across the United States showed that the problems related to requirements processes account for 45%, and the lack of end user participation and incomplete requirements are two main reasons, each accounting for 13% and 12% [18]. In addition, extensive engineering experience has shown that requirements problems are one of the main sources of engineering problems; therefore, the research in this paper is limited to the stage of requirements analysis, and below, SEP (s) are replaced by SREP (s). Requirements analysis mainly solves the problem of what a product should do [15]. The main reason for requirements problems is the problems of knowledge sharing and reuse, and ontology is an effective means to solve these two problems. In addition, objects (classes) do not always play a primary role in a domain, and sometimes it is more important to identify relationships between objects (classes) [19]. Therefore, it is necessary to characterize other associations between SREP objects (classes). This paper uses an object-based ontology method to characterize the SREP and define its semantics.

4.1.1 Definition of SREP

SREP is a summary of experience often making mistakes in requirements engineering. Thus, this paper proposes the definition of SREP according to the definition of SEP.

Definition 2: The SREP refers to the error produced in the software requirements development stage, which occurs repeatedly in a specific error lifetime scenario, spreads in the subsequent design and implementation, and may cause a system (component) to fail to perform the expected function or affect the maintainability of the system. Such errors are general and common in a specific scenario and can be corrected by various means.

The definition shows that the core components of an SREP are “Scenario”, “ErrorManifestation” and “Solution”. In addition, “Severity” should also be included. There is a class inheritance relationship between SEP and SREP as shown in Fig. 5.

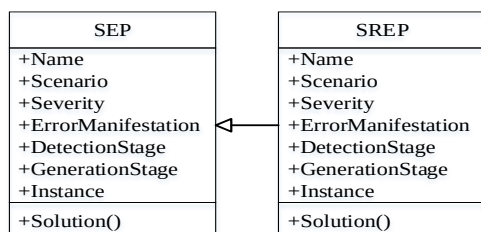


Fig. 5. Class inheritance relationship between SEP and SREP

4.1.2 Constituent components of SREPO

The universe of SREPO constituent components U is: $U = \{\text{Concepts, Object properties, Data properties, } P^R, P^C, \text{Inherit-hierarchies, Relationships, Instances, Mappings, Rules}\}$. Concepts, Inherit-hierarchies, Relationships, and Instances build the basic skeleton of the SREPO. All other components are attached to the basic skeleton and refine the basic skeleton. Object properties and Data properties are two types of properties. The Object properties connect instances together, and the Data properties connect instances and values together. P^R indicates the restrictions on properties, including the restrictions on the type, range, and maximum/minimum number of property values. P^C indicates property characteristics. Mappings represent the mappings between different levels of ontology. Rules include axioms and custom rules. The Rules can be used to constrain information, prove correctness, or derive new information. The Rules can also be used to express richer relationships between concepts.

4.1.3 Construction of SREPO

This paper builds the SREPO according to the “error generation stage”.

Definition 3 $\text{SREPO} := \langle \text{Concepts, Object properties, Data properties, } P^R, P^C, \text{Inherit-hierarchies, Relationships, Rules, Instances} \rangle$.

The concept classes of SREPO is shown in Fig. 6. The concept classes marked with an “*” are non-terminal concept classes, and the rest are terminal concept classes.

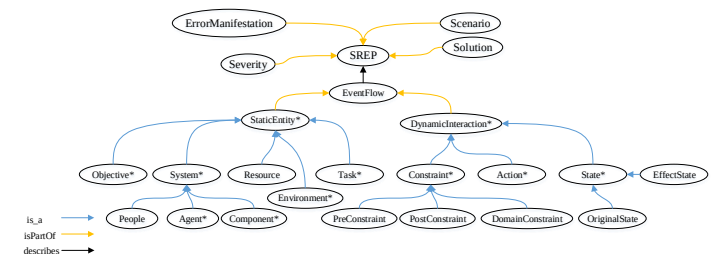


Fig. 6. Concept class hierarchies of SREPO

Define an inheritance relationship as:

Definition 4 An inheritance relationship is a mechanism of a sub-class automatically sharing the properties and structure of a parent class in the SREPO concept class hierarchy.

Then, the sub-class and the parent class of non-terminal concept classes form an inheritance relationship. Therefore, a new class can be created based on an existing concept class by taking the content defined by the existing class as its own content and adding new content. A portion of the concept relationships of SREPO is obtained as shown in Table. I. The left side of arrow is a source concept node, and the right side is a destination concept node. Take an example of “subObjective”. Its property characteristic is partial order. That is because it satisfies

“reflexivity”, “symmetry” and “transitivity” simultaneously, i.e.,

$$(1) \forall a \in A, (a, a) \in I$$

$$(2) \forall a, b \in A, ((a, b) \in I) \cap ((b, a) \in I) \rightarrow (a = b)$$

$$(3) \forall a, b, c \in A, ((a, b) \in I) \cap ((b, c) \in I) \rightarrow (a, c) \in I$$

and it can be represented as $a \prec b$.

TABLE I
A PORTION OF CONCEPT RELATIONSHIPS OF SREPO

Relationship types	Relationships	Property characteristics
subObjective	Objective $\rightarrow$ Sub-objective	partial order
interact	System $\rightarrow$ Environment	symmetry
has	System $\rightarrow$ Objective	-

Furthermore, a portion of the concept space of SREPO is shown as Fig. 7.

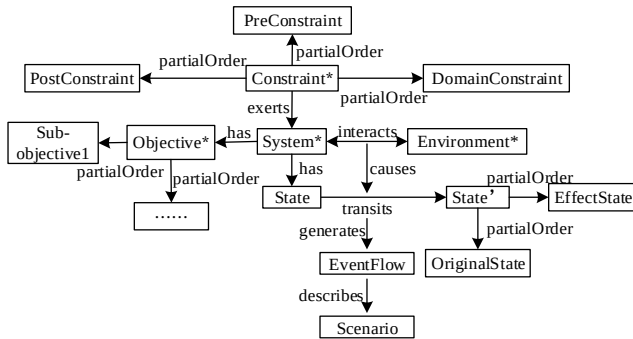


Fig. 7. A portion of concept space of SREPO

4.2 Ontology representation of SHIEP

4.2.1 Ontology representation framework of SHIEP

For the SISs, software requirements (SRs) can be divided into two categories: internal requirements and external requirements from the perspectives of macro function and internal structure. The internal requirements describe the connotation of requirements, i.e., the internal structure of a software system; the external requirements describe the extension of requirements, reflecting the interaction between a software system and its environment. The environment here includes the entities of people or other systems in contact with the software system. The external requirements determine the internal requirements. The external requirements can be further divided into explicit requirements and implicit requirements. The explicit requirements describe the direct interaction and constraint between a software system and its environment; the implicit requirements describe the indirect interaction and constraint between a software system and its environment. Thus, SRs := < internal requirements, external requirements < explicit requirements, implicit requirements > >. Then, Environment Ontology := < ExplicitEnvironment, ImplicitEnvironment>. Moreover, the set of implicit environment can be obtained based on experience:

ImplicitEnvironment = <OtherEnvironment < StressEnvironment<TemperatureEnvironment, VibrationEnvironment, HumidityEnvironment, ElectricityEnvironment, CombinedStressEnvironment>, EMEnvironment, GeographyEnvironment, AtmosphereEnvironment>, OperationEnvironment>, where CombinedStressEnvironment = TemperatureEnvironment $\cup$ VibrationEnvironment $\cup$ ElectricityEnvironment $\cup$ HumidityEnvironment.

Requirements engineering focuses on the expected goals and corresponding scenarios of software systems [20]; therefore, it can be considered that the SRs are the environmental state changes and goal realization caused by the interaction between a software system and its environment. Then, SRs:=<System <Agent, Entity>, Environment<ExplicitEnvironment, ImplicitEnvironment>, Interact<Control, Monitor, Perform>, Objective<Avoid, Cease, Achieve, Maintain>>, where “Agent” represents a special object, not a person, but only a machine or other mechanism, and must be autonomy, initiative and intelligence. “Entity” refers to any static thing that constitutes the basic structure of a software system. “Interact” refers to the interaction between a software system and its environment.

An event flow describes the sequence of interactive actions, including how to start, end, and how to interact. The semantics of SRs also describe the sequence of interactive actions changing the states of environment. Therefore, state transition models can be adopted to describe state transitions and event flows, i.e., the formation of scenario. Then, an environment framework can be defined as follows,

State< OriginalState, EffectState > $\neq$ NULL

interact $\rightarrow$ transit: T (State, State')

transit: T (State, State') $\rightarrow$ EventFlow

Constraint< PreConstraint, PostConstraint, DomainConstraint {Trigger, Pred, Synch, Conseq}>

“Constraint” acts on “interact”, “transit” and “EventFlow”, respectively. “DomainConstraint” includes “Trigger_{domain}”, “Pred_{domain}”, “Synch_{domain}” and “Conseq_{domain}”. And domain knowledge can be described by the state transition of environment.

“Environment” can be described as,

Environment = < ExplicitEnvironment, ImplicitEnvironment < OperationEnvironment, OtherEnvironment <StressEnvironment < TemperatureEnvironment, VibrationEnvironment, HumidityEnvironment, ElectricityEnvironment, CombinedStressEnvironment >, EMEnvironment, GeographyEnvironment, AtmosphereEnvironment >> >, where TemperatureEnvironment = {t-Env₁, ..., t-Env₁}, VibrationEnvironment = {v-Env₁, ..., v-Env₂}, HumidityEnvironment = {h-Env₁, ..., h-Env₃}, ElectricityEnvironment = {e-Env₁, ..., e-Env₄}, CombinedStressEnvironment = {c-Env₁, ..., c-Env₅}, EMEnvironment = {EM-Env₁, ..., EM-Env₆},

GeographyEnvironment = {g-Envi₁, ..., g-Envi₇},
 AtmosphereEnvironment = {a-Envi₁, ..., a-Envi₈}, s₁, ..., s₈ ∈ N . Interact = {interact₁, ..., interact_n}, n ∈ N . State={state₁, ..., state_i}, t ∈ N .
 Constraint = {constraint₁, constraint₂, ..., constraint_l}, l ∈ N . They are all obtained based on the inference of domain ontology.

4.2.2 Types and ontology representations of SHIEPs

Most of the current software requirements elicitation (SRE) methods do not consider prior information such as SEPs at the beginning of SRE, resulting in incomplete knowledge. SIS software generally has the following characteristics: (1) SIS software is closely connected with hardware, and has strong requirements of dedicated external equipments; (2) SIS software is usually strong real-time; (3) SIS software usually operates in specific conditions, and relates to interactive environments; (4) SIS software has high safety requirements. The above characteristics also spawn the SHIEPs (SHIEs). The SHIEPs (SHIEs) arise from requirements analysis stage and are a sub-class of SREPs (SREs). They have gradually become an important part of error patterns (errors) during the field usage of SISs, and they are extremely harmful. Therefore, it is necessary to study the SHIEPs (SHIEs). Moreover, the collection of SHIEPs (SHIEs) mainly depends on the field usage of SISs. That is because the environment of field usage or field test of SISs is relatively real, and the exposed problems can reflect actual situations. The types of SHIEP according to an error generation mechanism are shown in Fig. 8. The core components of SHIEP are: “Scenario”, “ErrorManifestation” and “Solution”.

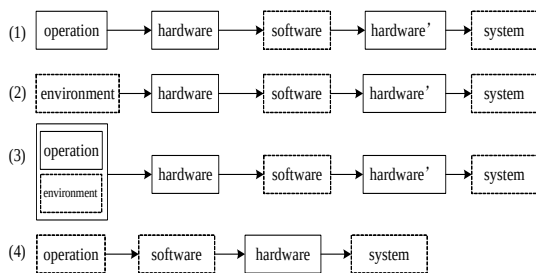


Fig. 8. Four types of SHIEP

1) Type 1 of SHIEP

The type 1 of SHIEP in Fig. 8 is described as, the unpredictable changes in operating conditions directly affect hardware and eventually lead to a system failure. The changes directly affect the hardware components of a system, accumulate over time, and gradually penetrate the internal units of the components, thereby affecting the characteristics of the components or circuits. Then, the generated abnormal electrical signals act as software inputs through software-hardware interfaces to software, resulting in abnormal software operations, and then react

to the components or the system, and in turn cause global or local failures of the system. The complete process is shown in Fig. 9. The “units of hardware components’ ” and “hardware components of system’ ” in this figure represent the state change of corresponding part (normal to abnormal), respectively. This process is a closed loop feedback process, and every step is accompanied by a state transition.

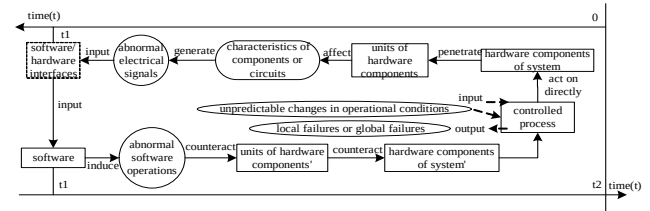


Fig. 9. Type 1 of SHIEP

The specific contents of the core part of type 1 of SHIEP are as follows,

Scenario of requirements incompleteness:

Scenario 1:

$\text{constraint}_k(\text{interact}_i) \in \text{Constraint}(1 \leq k \leq l, 1 \leq i \leq n, k \in N, l \in N, i \in N, n \in N),$

$\exists (\text{interact}_i \notin \text{Interact}) \rightarrow (\text{state}_i(\text{hardware}) \notin \text{State}(\text{hardware})) \rightarrow (\text{state}_i(\text{software}) \notin \text{State}(\text{software})) \rightarrow (\text{state}_j(\text{hardware}) \notin \text{State}(\text{hardware}))$

ErrorManifestation: safety error

Solution: A complete set of predefined operating conditions and a complete set of domain constraints are given to build a complete domain knowledge ontology.

2) Type 2 of SHIEP

The type 2 of SHIEP in Fig. 8 is described as, unpredictable environmental changes directly affect hardware and eventually lead to a system failure. The changes directly affect the hardware components of a system, accumulate over time, and gradually penetrate the internal units of the components, thereby affecting the characteristics of the components or circuits. Then, the generated abnormal electrical signals act as software inputs through software-hardware interfaces to software, resulting in abnormal software operations, and then react to the components or the system, and in turn cause global or local failures of the system. The complete process is shown in Fig. 10. The “units of hardware components’ ” and “hardware components of system’ ” in this figure represent the state change of corresponding part (normal to abnormal), respectively. This process is a closed loop feedback process, and every step is accompanied by a state transition.

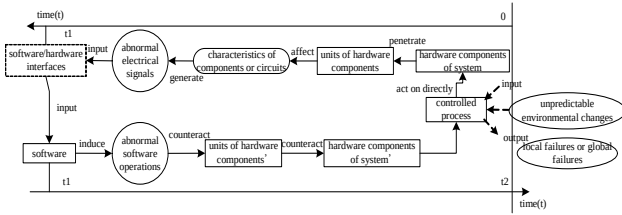


Fig. 10. Type 2 of SHIEP

The specific contents of the core part of type 2 of SHIEP are as follows,

Scenario of requirements incompleteness:

Scenario 2:

$\text{constraint}_k(\text{interact}_i) \in \text{Constraint}(1 \leq k \leq l, 1 \leq i \leq n, k \in \mathbb{N}, l \in \mathbb{N}, i \in \mathbb{N}, n \in \mathbb{N})$,

Sub-scenario 2-1:

$\exists (t\text{-Envi}_s \notin \text{ImplicitEnvironment}) \rightarrow ((t\text{-Envi}_s \cup \text{ImplicitEnvironment}) \not\subset \text{Environment}) \rightarrow (\text{state}_i(\text{hardware}) \notin \text{State}(\text{hardware})) \rightarrow (\text{state}_i(\text{software}) \notin \text{State}(\text{software})) \rightarrow (\text{state}_j(\text{hardware}) \notin \text{State}(\text{hardware})) (s1 \leq s, s \in \mathbb{N})$

Sub-scenario 2-2:

$\exists (v\text{-Envi}_s \notin \text{ImplicitEnvironment}) \rightarrow ((v\text{-Envi}_s \cup \text{ImplicitEnvironment}) \not\subset \text{Environment}) \rightarrow (\text{state}_i(\text{hardware}) \notin \text{State}(\text{hardware})) \rightarrow (\text{state}_i(\text{software}) \notin \text{State}(\text{software})) \rightarrow (\text{state}_j(\text{hardware}) \notin \text{State}(\text{hardware})) (s2 \leq s, s \in \mathbb{N})$

Sub-scenario 2-3:

$\exists (h\text{-Envi}_s \notin \text{ImplicitEnvironment}) \rightarrow ((h\text{-Envi}_s \cup \text{ImplicitEnvironment}) \not\subset \text{Environment}) \rightarrow (\text{state}_i(\text{hardware}) \notin \text{State}(\text{hardware})) \rightarrow (\text{state}_i(\text{software}) \notin \text{State}(\text{software})) \rightarrow (\text{state}_j(\text{hardware}) \notin \text{State}(\text{hardware})) (s3 \leq s, s \in \mathbb{N})$

Sub-scenario 2-4:

$\exists (e\text{-Envi}_s \notin \text{ImplicitEnvironment}) \rightarrow ((e\text{-Envi}_s \cup \text{ImplicitEnvironment}) \not\subset \text{Environment}) \rightarrow (\text{state}_i(\text{hardware}) \notin \text{State}(\text{hardware})) \rightarrow (\text{state}_i(\text{software}) \notin \text{State}(\text{software})) \rightarrow (\text{state}_j(\text{hardware}) \notin \text{State}(\text{hardware})) (s4 \leq s, s \in \mathbb{N})$

Sub-scenario 2-5:

$\exists (c\text{-Envi}_s \notin \text{ImplicitEnvironment}) \rightarrow ((c\text{-Envi}_s \cup \text{ImplicitEnvironment}) \not\subset \text{Environment}) \rightarrow (\text{state}_i(\text{hardware}) \notin \text{State}(\text{hardware})) \rightarrow (\text{state}_i(\text{software}) \notin \text{State}(\text{software})) \rightarrow (\text{state}_j(\text{hardware}) \notin \text{State}(\text{hardware})) (s5 \leq s, s \in \mathbb{N})$

Sub-scenario 2-6:

$\exists (EM\text{-Envi}_s \notin \text{ImplicitEnvironment}) \rightarrow ((EM\text{-Envi}_s \cup \text{ImplicitEnvironment}) \not\subset \text{Environment}) \rightarrow (\text{state}_i(\text{hardware}) \notin \text{State}(\text{hardware})) \rightarrow (\text{state}_i(\text{software}) \notin \text{State}(\text{software})) \rightarrow (\text{state}_j(\text{hardware}) \notin \text{State}(\text{hardware})) (s6 \leq s, s \in \mathbb{N})$

Sub-scenario 2-7:

$\exists (g\text{-Envi}_s \notin \text{ImplicitEnvironment}) \rightarrow ((g\text{-Envi}_s \cup \text{ImplicitEnvironment}) \not\subset \text{Environment}) \rightarrow (\text{state}_i(\text{hardware}) \notin \text{State}(\text{hardware})) \rightarrow (\text{state}_i(\text{software}) \notin \text{State}(\text{software})) \rightarrow (\text{state}_j(\text{hardware}) \notin \text{State}(\text{hardware})) (s7 \leq s, s \in \mathbb{N})$

Sub-scenario 2-8:

$\exists (a\text{-Envi}_s \notin \text{ImplicitEnvironment}) \rightarrow ((a\text{-Envi}_s \cup \text{ImplicitEnvironment}) \not\subset \text{Environment}) \rightarrow (\text{state}_i(\text{hardware}) \notin \text{State}(\text{hardware})) \rightarrow (\text{state}_i(\text{software}) \notin \text{State}(\text{software})) \rightarrow (\text{state}_j(\text{hardware}) \notin \text{State}(\text{hardware})) (s8 \leq s, s \in \mathbb{N})$

ErrorManifestation: safety error

Solution: A complete set of predefined operating conditions and a complete set of domain constraints are given to build a complete domain knowledge ontology.

3) Type 3 of SHIEP

The type 3 of SHIEP in Fig. 8 is described as, unpredictable changes in operating conditions and unpredictable environmental changes directly affect hardware and eventually lead to a system failure. The changes directly affect the hardware components of a system, accumulate over time, and gradually penetrate the internal units of the components, thereby affecting the characteristics of the components or circuits. Then, the generated abnormal electrical signals act as software inputs through software-hardware interfaces to software, resulting in abnormal software operations, and then react to the components or the system, and in turn cause global or local failures of the system. The complete process is shown in Fig. 11. The “units of hardware components” and “hardware components of system” in this figure represent the state change of corresponding part (normal to abnormal), respectively. This process is a closed loop feedback process, and every step is accompanied by a state transition.

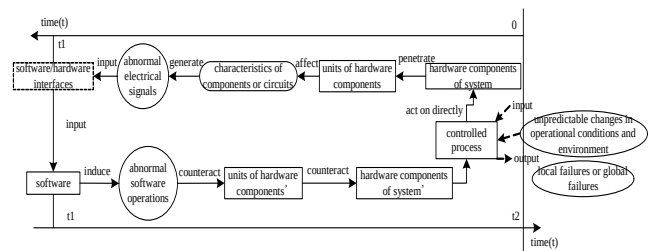


Fig. 11. Type 3 of SHIEP

The specific contents of the core part of type 3 of SHIEP are as follows,

Scenario of requirements incompleteness:

Scenario 3:

$\text{constraint}_k(\text{interact}_i) \in \text{Constraint}(1 \leq k \leq l, 1 \leq i \leq n, k \in \mathbb{N}, l \in \mathbb{N}, i \in \mathbb{N}, n \in \mathbb{N})$,

Sub-scenario 3-1:

$\exists ((\text{operation}_i \notin \text{Operation}) \cap ((t\text{-Envi}_s \notin \text{ImplicitEnvironment}) \rightarrow ((t\text{-Envi}_s \cup \text{ImplicitEnvironment}) \not\subset \text{Environment})))$

$\text{ImplicitEnvironment} \not\subseteq \text{Environment})) \rightarrow$
 $(\text{state}_i(\text{hardware}) \notin \text{State}(\text{hardware})) \rightarrow (\text{state}_i(\text{software}) \notin$
 $\text{State}(\text{software})) \rightarrow (\text{state}_j(\text{hardware}) \notin \text{State}(\text{hardware}))$
 $(s1 \leq s, s \in \mathbb{N})$

Sub-scenario 3-2:

$\exists ((\text{operation}_i \notin \text{Operation}) \cap ((v\text{-Envi}_s \notin$
 $\text{ImplicitEnvironment} \rightarrow ((v\text{-Envi}_s \cup$
 $\text{ImplicitEnvironment} \not\subseteq \text{Environment}))) \rightarrow$
 $(\text{state}_i(\text{hardware}) \notin \text{State}(\text{hardware})) \rightarrow (\text{state}_i(\text{software}) \notin$
 $\text{State}(\text{software})) \rightarrow (\text{state}_j(\text{hardware}) \notin \text{State}(\text{hardware}))$
 $(s2 \leq s, s \in \mathbb{N})$

Sub-scenario 3-3:

$\exists ((\text{operation}_i \notin \text{Operation}) \cap ((h\text{-Envi}_s \notin$
 $\text{ImplicitEnvironment} \rightarrow ((h\text{-Envi}_s \cup$
 $\text{ImplicitEnvironment} \not\subseteq \text{Environment}))) \rightarrow$
 $(\text{state}_i(\text{hardware}) \notin \text{State}(\text{hardware})) \rightarrow (\text{state}_i(\text{software}) \notin$
 $\text{State}(\text{software})) \rightarrow (\text{state}_j(\text{hardware}) \notin \text{State}(\text{hardware}))$
 $(s3 \leq s, s \in \mathbb{N})$

Sub-scenario 3-4:

$\exists ((\text{operation}_i \notin \text{Operation}) \cap ((e\text{-Envi}_s \notin$
 $\text{ImplicitEnvironment} \rightarrow ((e\text{-Envi}_s \cup$
 $\text{ImplicitEnvironment} \not\subseteq \text{Environment}))) \rightarrow$
 $(\text{state}_i(\text{hardware}) \notin \text{State}(\text{hardware})) \rightarrow (\text{state}_i(\text{software}) \notin$
 $\text{State}(\text{software})) \rightarrow (\text{state}_j(\text{hardware}) \notin \text{State}(\text{hardware}))$
 $(s4 \leq s, s \in \mathbb{N})$

Sub-scenario 3-5:

$\exists ((\text{operation}_i \notin \text{Operation}) \cap ((c\text{-Envi}_s \notin$
 $\text{ImplicitEnvironment} \rightarrow ((c\text{-Envi}_s \cup$
 $\text{ImplicitEnvironment} \not\subseteq \text{Environment}))) \rightarrow$
 $(\text{state}_i(\text{hardware}) \notin \text{State}(\text{hardware})) \rightarrow (\text{state}_i(\text{software}) \notin$
 $\text{State}(\text{software})) \rightarrow (\text{state}_j(\text{hardware}) \notin \text{State}(\text{hardware}))$
 $(s5 \leq s, s \in \mathbb{N})$

Sub-scenario 3-6:

$\exists ((\text{operation}_i \notin \text{Operation}) \cap ((EM\text{-Envi}_s \notin$
 $\text{ImplicitEnvironment} \rightarrow ((EM\text{-Envi}_s \cup$
 $\text{ImplicitEnvironment} \not\subseteq \text{Environment}))) \rightarrow$
 $(\text{state}_i(\text{hardware}) \notin \text{State}(\text{hardware})) \rightarrow (\text{state}_i(\text{software}) \notin$
 $\text{State}(\text{software})) \rightarrow (\text{state}_j(\text{hardware}) \notin \text{State}(\text{hardware}))$
 $(s6 \leq s, s \in \mathbb{N})$

Sub-scenario 3-7:

$\exists ((\text{operation}_i \notin \text{Operation}) \cap ((g\text{-Envi}_s \notin$
 $\text{ImplicitEnvironment} \rightarrow ((g\text{-Envi}_s \cup$
 $\text{ImplicitEnvironment} \not\subseteq \text{Environment}))) \rightarrow$
 $(\text{state}_i(\text{hardware}) \notin \text{State}(\text{hardware})) \rightarrow (\text{state}_i(\text{software}) \notin$
 $\text{State}(\text{software})) \rightarrow (\text{state}_j(\text{hardware}) \notin \text{State}(\text{hardware}))$
 $(s7 \leq s, s \in \mathbb{N})$

Sub-scenario 3-8:

$\exists ((\text{operation}_i \notin \text{Operation}) \cap ((a\text{-Envi}_s \notin$
 $\text{ImplicitEnvironment} \rightarrow ((a\text{-Envi}_s \cup$
 $\text{ImplicitEnvironment} \not\subseteq \text{Environment}))) \rightarrow$
 $(\text{state}_i(\text{hardware}) \notin \text{State}(\text{hardware})) \rightarrow (\text{state}_i(\text{software}) \notin$

$\text{State}(\text{software})) \rightarrow (\text{state}_j(\text{hardware}) \notin \text{State}(\text{hardware}))$
 $(s8 \leq s, s \in \mathbb{N})$

ErrorManifestation: safety error

Solution: A complete set of predefined operating conditions and a complete set of domain constraints are given to build a complete domain knowledge ontology.

4) Type 4 of SHIEP

The type 4 of SHIEP in Fig. 8 is described as, unpredictable changes in operating conditions directly affect software and eventually lead to a system failure. The changes directly affect the software and cause abnormal software operations generating abnormal electrical signals through software-hardware interfaces. Then, the abnormal electrical signals directly act on the units of hardware components and then the hardware components of a system, and in turn cause global or local failures of the system. The complete process is shown in Fig. 12. This process is a closed loop feedback process, and every step is accompanied by a state transition.

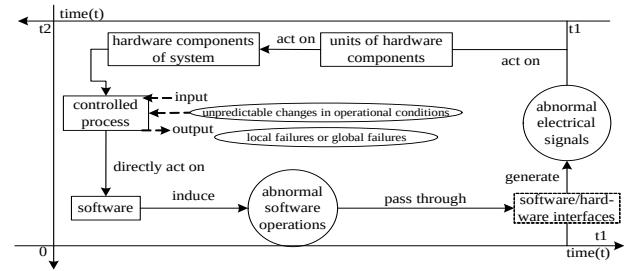


Fig. 12. Type 4 of SHIEP

The specific contents of the core part of type 4 of SHIEP are as follows,

Scenario of requirements incompleteness:

Scenario 4:

$\text{constraint}_k(\text{interact}_i) \in \text{Constraint}(1 \leq k \leq l, 1 \leq i \leq n,$
 $k \in \mathbb{N}, l \in \mathbb{N}, i \in \mathbb{N}, n \in \mathbb{N}),$

$\exists (\text{interact}_i \notin \text{Interact}) \rightarrow (\text{state}_i(\text{software}) \notin$
 $\text{State}(\text{software})) \rightarrow (\text{state}_j(\text{hardware}) \notin \text{State}(\text{hardware}))$

ErrorManifestation: safety error

Solution: A complete set of predefined operating conditions and a complete set of domain constraints are given to build a complete domain knowledge ontology.

V. CASE STUDY

Various applications can be implemented based on the SHIEP, e.g., using the instances of SHIEP to guide the applications of failure mode and effect analysis (FMEA) method to detect the potential failure modes that may adversely affect system functions or performance. This is because the three core components of SHIEP represented by the ontology method are conducive to FMEA analysts familiar with the actual usage of a system and the weaknesses of a design. Then, the followings can be

achieved: (1) identify known or potential failure modes in a system; (2) assess the effect of the failure modes; (3) propose improvement suggestions to reduce or eliminate these failures. The above effects are particularly significant under the current software-intensive developmental trend of large-scale systems. And the basic work is to summarize test experience and collect related SHIEP instances, and then to represent them by the ontology and store them in an ontology library.

This paper takes a certain type of airborne radar software system as an example, uses protégé to edit the SHIEPs and their instances, and further introduces the applications of software FMEA (SFMEA) based on the above work.

5.1 Introduction of radar systems

Because SHIEP instances need to use the domain concepts of radar systems, the functional principle of radar systems is first introduced. Radar is a system that uses radio waves to find the position and movement of objects. Its basic task is discovering the interested objects, measuring the state parameters of distance, direction, and speed, etc. And it is mainly composed of an antenna, a transmitter, a receiver, a signal processor, a data processor, an indicator, and a display, as shown in Fig. 13 [21].

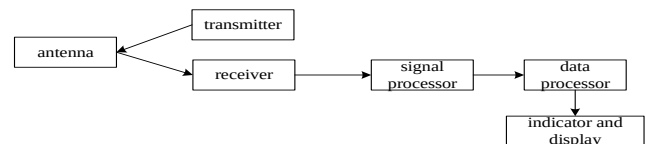


Fig. 13. Composition of radar systems

5.2 Ontology edit of SHIEP and SHIEP instances

Protégé is an extensible, cross-platform and open source development environment for generating and editing ontologies. It has been widely used and promoted in many countries. Protégé is one of the most popular ontology editing tools currently, capable of defining classes and class hierarchies, attribute relationships and attribute-value constraints, and the relationships between classes and attributes. Instance tags can be used to obtain the instances of classes defined in an ontology. This paper adopts Protégé to edit ontologies. Specifically, first establish an SHIEP class, and then establish its constituent component classes. For the sake of brevity, this paper only gives the three core component classes of SHIEPs: “Scenario” class, “ErrorManifestation” class and “Solution” class. Fig. 14 shows that an SHIEP class relates to its constituent component classes by an object property “isPartOf”. After the completion of ontology editing, the description logic (DL) query in Protégé can be used for class queries or simple reasoning, which is not repeated here.

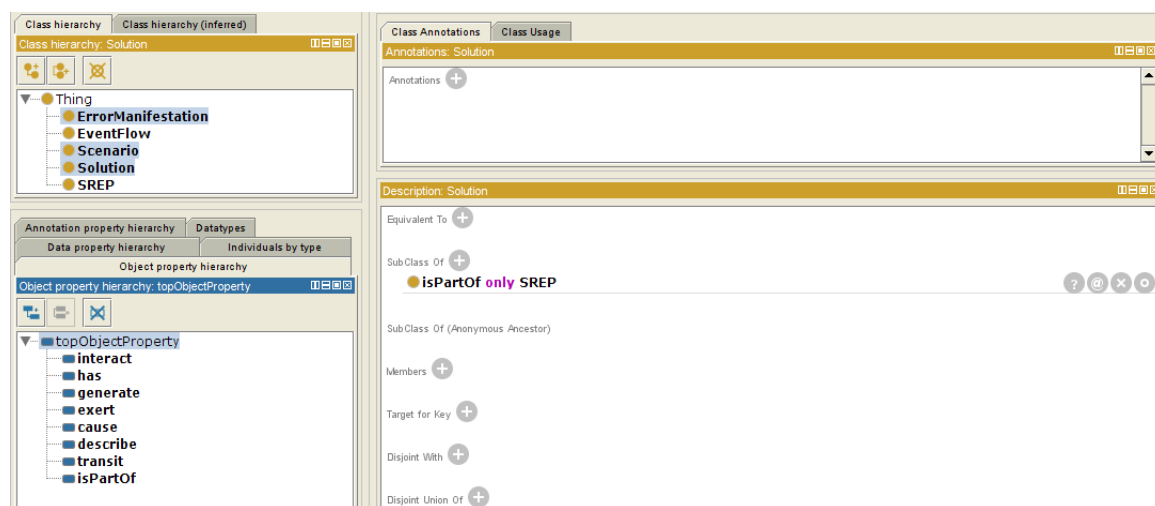


Fig. 14. Object property “isPartOf”

Experience has shown that the number of the failures based on SHIEs of radar systems found in reliability tests or in field usage is limited, far less than traditional failures. Therefore, the case study of SFMEA given below does not distinguish specific types of radar, but analyzes a specific type of error related to a certain type of stress, trying to reveal the effect of such stress on the SHIEs and the failures based on the SHIEs of radar systems.

5.3 FMEA based on SHIEP

1 System definition

1) Analyzing functions

The functions of the radar sub-system of a certain type of mission electronic system are to detect a target and measure the state parameters of distance, direction, and speed. A functional schematic diagram is shown in Fig. 15.

The dashed box of the “servo” in the figure indicates that this part is not available in all radar. For example, phased array radar (PAR) does not have a servo system.

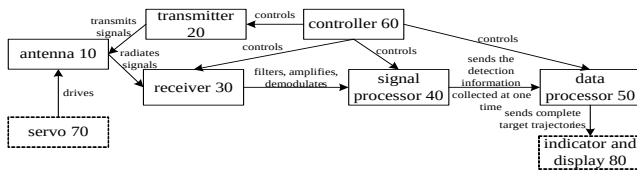


Fig. 15. Functional schematic diagram of radar sub-system of a certain type of mission electronic system

2) Drawing a block diagram

• Drawing a functional block diagram: the correspondence diagram of function level and structure level of a certain type of radar is shown in Fig. 16.

2 Indenture levels

As shown in Fig. 18, an initial indenture level is “aircraft”; an indenture level is “radar system”; the lowest indenture levels are “antenna (10)”, “transmitter (20)”, “.....”, “controller (60)”, etc.

3 Severity definition

The systems on an aircraft can be divided into two categories: (1) the systems that perform basic flight functions, e.g., power supplies, flying control (FC) systems, etc.; (2) the systems that perform specific tasks, e.g., mission electronic systems including radar systems and electronic countermeasures, etc. Although systems

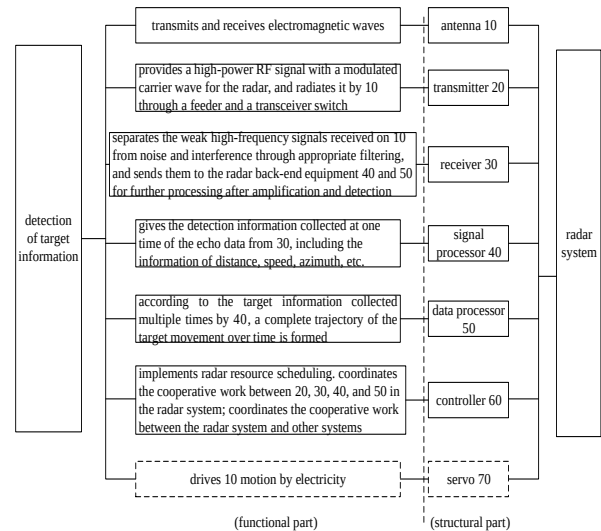


Fig. 16. Correspondence diagram of function level and structure level of a certain type of radar

• Drawing a mission reliability block diagram: the mission reliability block diagram of a certain type of radar system is shown in Fig. 17.

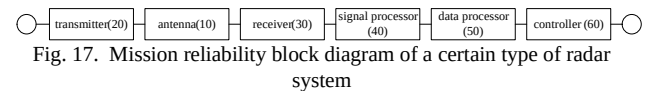


Fig. 17. Mission reliability block diagram of a certain type of radar system

that perform specific missions generally do not affect basic flight functions, they have a decisive effect on operational effectiveness and affect the reliability and safety of aircraft systems. Therefore, the definition of severity is given based on combat effectiveness and basic usage as shown in Table. II.

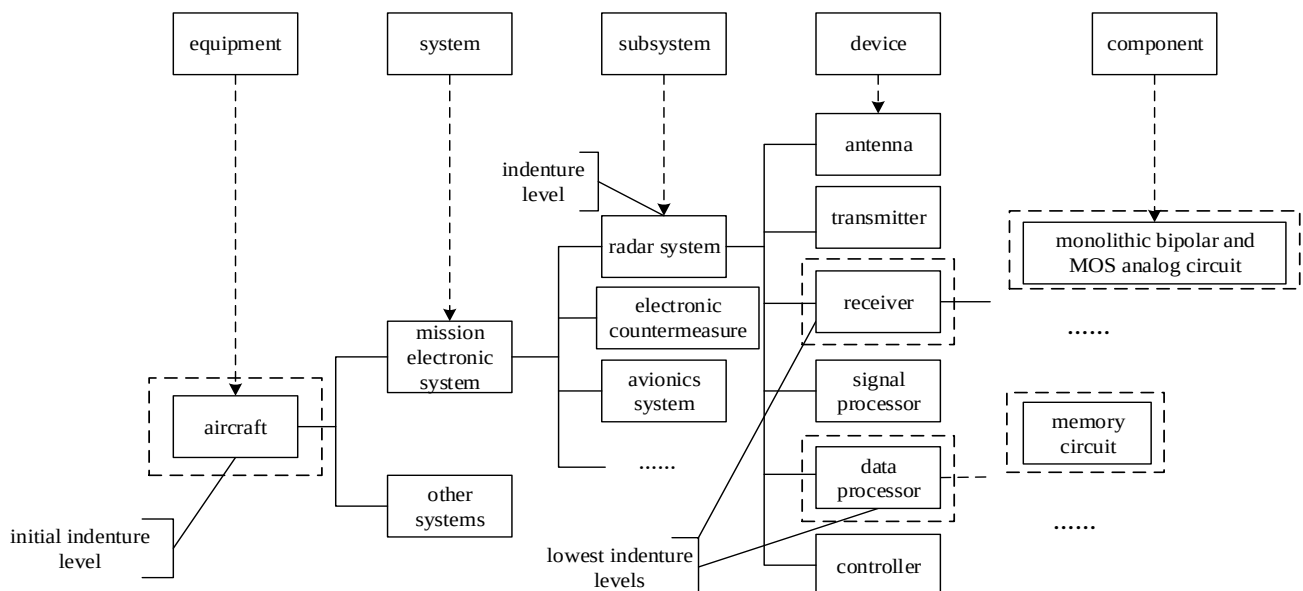


Fig. 18. An example of indenture level division of a certain type of mission electric system

TABLE II
 TYPES AND DEFINITIONS OF SEVERITY

Types	Definitions
Type I (hazardous)	Mission failures cause a complete loss of combat effectiveness, and may endanger the safety of personnel or aircraft.
Type II (moderate)	The decrease in mission completion effect causes the decline in operational effectiveness (such as flight delay, suspension or cancellation of flight, declined flight quality, increased landing difficulty, moderate environmental damage, etc.), and may cause moderate injury to personnel or moderate damage to systems.
Type III (minor)	No effect or little effect on operational effectiveness, but may increase unplanned maintenance or repair.

4 Failure mode analysis

The failure modes based on the SHIEPs of a certain type of radar system are mainly obtained from relevant information analysis. The occurrence probability levels of failure modes based on SHIEPs are divided into five levels: A, B, C, D, and E. Table. III shows the specific definitions.

 TABLE III
 OCCURRENCE PROBABILITY LEVELS OF FAILURE MODES

Occurrence probability levels	Severity levels*		
	I (catastrophic)	II (moderate)	III(slight)
A(frequently)	1	5	13
B(sometimes)	2	7	16
C(occasionally)	4	8	18
D(rarely)	8	12	19
E(nearly impossible)	12	16	20

Note: the shadow part of table should be paid special attention to. The smaller the value is, the greater the risk.

5 FMEA table filling

The software/hardware integrated FMEA method is an extension of traditional FMEA methods. A risk index table and a risk level table are shown in Table. IV and Table. V, respectively.

 TABLE IV
 DEFINITIONS OF RISK INDEX

Levels	Definitions	Occurrence probability characteristics	Occurrence probability P (during product usage time)
A	frequently	very high probability	$\geq 20\%$
B	sometimes	medium probability	$10\% \leq P < 20\%$
C	occasionally	relatively low	$1\% \leq P < 10\%$
D	rarely	low	$0.1\% \leq P < 1\%$
E	nearly impossible	very low	$\leq 0.1\%$

 TABLE V
 TYPES OF RISK LEVEL

Risk evaluation indexes	Risk levels	Evaluation criteria
1-5	very high	unacceptable
6-9	high	unexpected (generally not accepted)

 10-17 moderate acceptable (need to review)
 18-20 low acceptable

According to the specific analysis of this case, the software/hardware integrated FMEA table of a certain type of radar system is shown in Table. VI.

Traditional software testing methods have limitations in detecting the SHIEPs. For type 1, type 2 and type 3 of SHIEPs given in section 4, software testing usually assumes that a hardware platform on which it operates has always been in an ideal state, without considering the actual environmental conditions or workload of the product, which may change the hardware characteristics or performance of the product, and eventually affect software operations. Software problems caused indirectly by environmental stresses, e.g., vibration stresses or temperature stresses, need to be detected by the hardware environment on which reliability tests operate. This also shows the limitations of software as a downstream product that must be attached to hardware.

Take a failure instance detected in domain usage as an example: this failure is caused by the error that the delay error of a delay device increases at a low temperature. The delay error causes the timing signals unmatched, resulting in the wrong signal value received by software, and further causes the operation error of the software, which acts on system hardware and eventually causes the system to malfunction. Although the direct cause is a design problem, the root cause is that the changes in component characteristics under extreme temperature conditions are not considered in the requirements analysis stage. After the detailed analysis of the problem, the problem-related knowledge can be incorporated into the ontology library as new knowledge. The scenario where this error occurs is: "Environment $\rightarrow$ AtmosphereEnvironment $\rightarrow$ ExtremeTemperature" + "interaction between software and hardware". Similarly, there are errors that occur under extreme vibration conditions or external interference represented by EMI, which are not listed here.

In short, based on the collected SHIEPs and their instances, we can elicit the corresponding scenarios and error manifestations, analyze solutions, further guide the implementation of FMEA method, identify all possible SHIEPs, analyze the causes and possible effects of SHIEPs, determine the severity and risk of SHIEPs, eliminate or control the dangerous failures of safety-critical products, and formulate effective improvement measures to improve product reliability and quality.

VI. CONCLUSION

This paper first analyzed software error lifetime from the perspective of knowledge-based software engineering and proposed an error generation model. Then, it proposed the concepts of SEP and SREP based on the ODC. According to an error generation mechanism, four types of SHIEP in the requirement stage, and

TABLE VI
SHIEP-BASED FMEA TABLE OF A CERTAIN TYPE OF RADAR SYSTEM

Initial indenture level: aircraft

Analyze: XXX

Approve: XXX

Indenture level: radar system

Check: XXX

Date: XX, XX, XXXX

Codes	Identifiers of product (function)	Functions	SHIEPs	Failure causes		Mission phases and work modes	Failure effects			Severity types	OP levels of SHIEPs	REI	Failure detection methods	Improvement measures
				Mechanisms	Sensors		Local	Higher level	Final					
10	antenna	transmits/receives electromagnetic waves	type I	the chip of central antenna enters a power saving mode under low temperature conditions			a chip enters the power-saving mode under low temperature conditions and does not communicate with a servo system		no pitch display on the screen	type III	C	18	vision	add a pull-up resistor to the chip's read-write end
20	transmitter	provides a high-power RF signal with a modulated carrier wave for the radar	-	-	-	-	-	-	-	-	-	-	-	-
30	receiver	Separates the weak high-frequency signals from noise and interference through appropriate filtering, and sends them to radar back-end equipment after amplification and detection	type I	the error of a delay component (a linear integrated circuit) increases under low temperature conditions			causes mismatch between related timing pulses, resulting in timing waveform distortion	using the timing signal to control the data transmission of receiving channel causes transmission dislocation	causes the radar cannot track the target under low temperature conditions	type I	C	2	vision	uses a software delay to replace a hardware delay
40	signal processor	gives the detection information collected at one time of the echo data from the receiver, including the information of distance, speed, azimuth, etc.	type I	the delay time of a delay component (a linear integrated circuit) changes with temperature rise.	temperature stress	detection of target information	the failure of an external component causes the other component to dissipate insufficient heat, and the temperature rise changes the delay time of the delay device	causes a data conflict	causes the software to fail and further causes the mission to fail	type I	C	4	vision	strengthens component screening and improves component reliability
50	data processor	according to the target information collected multiple times by the signal processor, a complete trajectory of the target movement over time is formed.	type I	data processing bus arbitration timing is closer to the critical value under low temperature conditions.			reports a flight failure, a maintenance failure, no echo signal on the picture under an imaging state	reports a data processor failure, a transmitter failure, a receiver failure	cannot intercept a target during an aerial combat	type I	B	2	vision	improves the algorithm design of data processing bus arbitration timing
60	controller	implements radar resource scheduling, coordinates the cooperative work between the transmitter, the receiver, the signal processor, and the data processor in the radar system, coordinates the cooperative work between the radar system and other systems.	-	-	-	-	-	-	-	-	-	-	-	-

Notes:

REI: Risk evaluation indexes; OP levels of SHIEPs: Occurrence probability levels of SHIEPs

corresponding ontology representation were given. The results showed that the prior information based on the SHIEPs was helpful to discover potential failures and failure modes that may adversely affect the function or performance of SISs. Therefore, the proposed SHIEP was of great significance for improving the quality of software development and verification. However, the current application scope of this method is relatively limited, and the experimental results obtained are still lacking in quantity. This is because the construction of ontology is a time-consuming and laborious task, requiring the participation of domain experts, software engineers, etc. On the other hand, the collection of error data is also a long-term task, especially for today's increasingly complex software systems, there are many reasons for software errors. Moreover, there are constantly new errors and their causes with the development of new technologies. However, once the ontology is built, it will contain rich experience and domain knowledge, which provides the possibility for the reuse of data resources, as well as the possibility of resource conservation and efficiency improvement. In addition, an increasingly complete SEP set also lays the foundation for a comprehensive understanding of software systems to ensure their quality.

The defect analysis method of ODC is of great significance and has a profound impact on the software industry. How to use the ODC method more reasonably and effectively under the development trend of large, open, dynamic and non-deterministic software systems will be a continuous research topic.

REFERENCES

- [1] M. R. Lyu, "Handbook of software reliability engineering," *Los Alamitos, California: McGraw-Hill Companies*, 1996.
- [2] R. Chillarege, Bhandari, I.S., Chaar, J.K., Halliday, M.J., Moebus, D.S., Ray, B.K., Wong, M.-Y., "Orthogonal defect classification: a concept for inprocess measurements," *IEEE Transactions on Software Engineering*, 18(11), pp. 943-956, 1992.
- [3] R. Chillarege, "Test and development process retrospective: a case study using ODC triggers," in *Proc. International Conference on DSN'02*. Washington, DC, USA, 2002, pp. 669-678.
- [4] R. Chillarege, "Understanding Bohr-Mandel bugs through ODC Triggers and a case study with empirical estimations of their field proportion," in *Proc. 3th International WoSAR'11*. Washington, DC, USA, 2011, pp. 7-13.
- [5] Lutz, R.R., Mikulski, I.C., "Empirical analysis of safety-critical anomalies during operations," *IEEE Transactions on Software Engineering*, 30(3), pp. 172-180, 2004.
- [6] Nuno Silva etc., "Experience report: orthogonal classification of safety critical issues," in *Proc. IEEE 25th International Symposium on Software Reliability Engineering*, Naples, Italy, 2014, pp. 156-166.
- [7] Raymond A. Paul, et al., "Defect-based reliability analysis for mission-critical software," in *Proc. 24th Annual International Computer Software and Applications Conference*, Taipei, Taiwan, 2000, pp. 439-444.
- [8] Pan Tiejun etc., "Defect tracing system based on orthogonal defect classification," in *Proc. 2008 International Conference on Computer Science and Software Engineering*, Wuhan, China, 2008, pp. 574-577.
- [9] LiGuo Huang, et al., "AutoODC: automated generation of orthogonal defect classifications," *Automated Software Engineering*, Springer, 22(1), pp. 3-46, 2015.
- [10] Ferdian Thung, et al., "Automatic defect categorization," in *Proc. 19th Working Conference on Reverse Engineering*, Kingston, Ontario, Canada, 2012, pp. 205-214.
- [11] Ruili Geng, et al., "In-process usability problem classification, analysis and improvement," in *Proc. 14th International Conference on Quality Software*, Dallas, TX, USA, 2014, pp. 240-245.
- [12] Martin Wirsing, "Beyond the horizon-final report, thematic group 6 on software intensive systems," June 2006, pp: 1-39.

- [13] T. R. Gruber, "Toward principles for the design of ontologies used for knowledge sharing?" *Int. J. Hum.-Comput. Stud.*, vol. 43, no. 5-6, pp. 907-928, Nov. 1995.
- [14] Renzuo Xu, "The testing method based on software knowledge," *J. Wuhan Univ. (Nat. Sci. Ed.)*, 46(1), pp. 61-62, 2000.
- [15] Xuan Hu, Bin Liu, Yichen Wang and Chongwu Jiang, "Application of software requirement error pattern based on ontology," *Journal of Beijing University of Aeronautics and Astronautics*, vol. 35, no. 6, pp. 723-727, 2009.
- [16] C. Alexander, "The timeless way of building," UK: Oxford University Press, 1979, pp. 5-20.
- [17] Xizi Huang, "Software reliability, safety and quality assurance," Beijing: Publishing House of Electronics Industry, 2002.
- [18] Zhijin, Linliu, Yingjin, "Software requirements engineering method," Beijing: Science Press, 2008.
- [19] Ruqian Lu, et al., "Ontology-oriented requirements analysis," *Journal of Software*, 11(8), pp. 1009-1017, 2000.
- [20] Jon G. Hall, Ivan Mistrík, Bashar Nuseibeh, Dr Andrés Silva, "Editorial relating software requirements and architectures," *IEE Proc.-Softw.*, 152(4), pp. 141-142, 2005.
- [21] Lufei Ding, Fulu Geng, Jianchun Chen, "Radar Principles," Beijing: Publishing House of Electronics Industry, 2009.



Xuan Hu was born in 1983. She received the Ph.D. degree in aerospace science and technology from Beihang University, China, in 2010. She is currently a senior engineer of CEPREI Laboratory. Her main research interests include ontology modeling, software requirement engineering, and software reliability.



Jie Liu was born in 1963. He received the B.S. degree in computer science from Xidian University, China. He is currently the director of the Information Security Research Center of CEPREI Laboratory. His main research interests include information security, software reliability and security, and software evaluation technology.