

The Orthogonal Defect Classification-based Software Error Pattern Ontology Construction Researches

Xuan Hu, Jie Liu*, Bo Wu

CEPREI, Guangzhou, China, 510610

huxuan@ceprei.com, liujie@ceprei.com, wub@ceprei.com

*Corresponding author, liujie@ceprei.com

Abstract—Orthogonal defect classification is a multi-dimensional measurement system with both qualitative and quantitative features, which is widely used in the software industry. However, the level of abstraction is high, which leads to limited semantic information. Therefore, it has a limited effect in the process of software engineering of complex software systems. To solve this problem, this paper first analyses the software error lifetime from the perspective of knowledge-based software engineering. Furthermore, based on the orthogonal defect classification, the concept of software error pattern is proposed, and its constituent elements and value sets are given. Moreover construct the software error pattern ontology in the requirements analysis phase, focusing on the elements such as “scenario”, “error-manifestation” and “solution”. The example verification part takes unmanned aerial vehicle flight control and management system software as an example to carry out the software requirements error pattern ontology representation and conducts software verification activities based on this and measures the development quality. The results show that the software error pattern can effectively guide the verification of complex software systems and measure the development quality. Therefore, the proposal of software error pattern is of great significance to improve the quality of software development and verification.

Keywords—orthogonal defect classification; software error; pattern; ontology; software-intensive systems

I. INTRODUCTION

How to provide developers with fast and effective feedback in a controlled and measurable way is a problem that has long plagued the software industry [1]. Two common forms of software defect analysis, statistical defect models and root cause analysis, the former is a quantitative method with good repeatability, but cannot provide feedback to developers in a timely manner in the way of available process control; the latter is a qualitative method that focuses 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

multidimensional measurement system that transforms qualitative information into quantitative measures [1].

There are many researches on ODC recently, e.g., some cases of usage of error trigger during the testing process [3] and the researches of Bohr-Mandel error using error trigger [4], experimental researches on application of ODC on safety, critical error of spacecraft system [5, 6], how to model Y2K error through ODC [7], defect tracing system based on ODC [8], automated generation of ODC [9, 10], in-process usability problem classification, analysis and improvement based on ODC [11]. In all, the popular researches on ODC tend to focus on its macroscopic characteristic as a measurement system. Although ODC characteristics reflect some semantics information of error, it is too abstractive to reflect individual gap because of the consistency across phases and uniformity across products [2]. Moreover it is difficult to apply ODC to the application scenario of complex software system which includes specific documents and error types. This situation is especially notable in the Software-Intensive Systems (SISs) community. SISs have become the developing trend of large-scale system, its applications range from embedded systems for automotive and aerospace applications to dedicated systems for wireless communications [12]. However the state of art of SISs application increasingly reflects two problems, i.e., system complexity and emergent behavior induced by several factors, e.g., the open and non-deterministic environment, interaction complexities, unpredictable operations or circumstances, etc. ODC is on a high level and has limited semantics information. Thus it is difficult to describe the complicated scenario of SISs via ODC. And it hinders the researches on the above problems through ODC.

This paper analyzes the life cycle of software error from the viewpoint of knowledge-based software engineering firstly. Furthermore it presents the definition of software error pattern (SEP) based on ODC. Thirdly it constructs the software requirements error pattern (SREP) ontology by object-oriented ontology representation method. An ontology is an explicit specification of a conceptualization [13, 14]. This step unifies the related concepts in the framework of ontology. Finally this paper introduces the application of above researches.

II. THE LIFE CYCLE OF SOFTWARE ERROR

Software is a sequence of computer knowledge that

expresses human knowledge and the correct use of knowledge.^[15] From the viewpoint of philosophy, it can be considered that the software defect is the reflection of the deviation of the knowledge generated in the process of creating the object of the software product based on the knowledge and this reflection will be regarded as some necessary adjustment^[15]. Figure 1 shows the software error lifetime throughout the developing, verification and field use. The reliability test in verification appeared in the hardware field. It is a necessary means of design improvement and quality evaluation during the development and production. There are several forms of incorrect software knowledge, implicit software error induced by man-made mistake, incorrect knowledge point or, using the correct knowledge wrongly^[15].

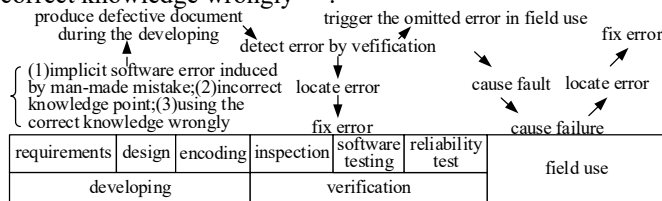


Figure 1. The software error lifetime

III. THE SOFTWARE ERROR PATTERN

ODC is an error classification method that helps to obtain error information at a higher level. Its elements include *Trigger*, *Defect-Type*, *Target*, etc. They are independent and orthogonal with each other. The *Trigger* describes the environment and condition which cause the error (*What*). The *Defect-Type* reflects the nature of error and guides how to amend it (*How*). These two elements are more reflective of the nature of the error than the rest of the elements. Meanwhile, experiences show that, (1) the context in which the error is generated is similar, and the effect is similar to the *Trigger* attribute; (2) the error has a similar performance and can be modified, i.e., there is a software implementation that satisfies the correct intent. The effect is similar to the *Defect-Type* and *Target* attributes. This inspires the paper to study software error with the help of the concept *Pattern*^[16, 17].

Definition1 SEP refers to the error produced in the software development stage, which occurs repeatedly in a certain scenario of error lifetime, and may cause the system (component) to fail to perform the expected function or affect the system maintainability. Such errors are general and common in a particular scenario and can be corrected by some means.

Therefore $SEP = \langle \text{Scenario}, \text{Error-manifestation}, \text{Solution}, \text{Severity}, \text{Detection Stage}, \text{Generation Stage}, \text{Instance} \rangle$.

The above definition distinguishes the generation and survival stages of error. Furthermore it is multi-viewpoint. Its cores are *Scenario*, *Error-manifestation* and *Solution*. The *Scenario* reflects the developer, tester and user viewpoints and is the basis for the test case and the catalyst that causes the failure (or failure). The *Error-manifestation* reflects the user's point of view and the *Solution* reflects the developer's point of view. Both of them express the nature of error and give a way to eliminate the error. An instance is the result of mapping a higher level of abstraction SEP concept to a specific case.

Most error classification systems use 5 to 15 defect attributes^[1]. The number of SEP components is in this range, they are independent of each other, and the two are orthogonal.

Figure 2 presents the SEP and its components value sets obtained based on the experiences and related literatures in the form of classes^[1, 18]. Both SEP and its constituent elements can be regarded as classes. The relationship between SEP and the constituent elements is an aggregation relationship. The *Scenario* class has six subclasses. The *SEP* class has three operations of *Measurement*, *Software Development* and *Software Verification*. The common operation of *Scenario*, *Error-manifestation*, *Detection Stage*, *Generation Stage* and *Severity* classes is *One Dimension Measurement*. The subclasses of *Scenario* class have the common operation of *Measure data supply*. The values of elements are the attributes of the corresponding classes. The above classes can be instantiated, corresponding to the "instance" of the constituent elements of SEP.

IV. THE SEP ONTOLOGY

A. The SEP Ontology Structure

A follow-up survey of 8,000 software projects across the United States also shows that 45% of the issues related to the demand process, and the lack of end-user participation and incomplete demand are the two main reasons, each 13% and 12%^[19]. A large amount of engineering experience indicates that demand issues are one of the main sources of engineering problems. Therefore, the research in this paper is limited to the requirements analysis stage and the following SEP is replaced by SREP. Demand analysis mainly addresses the problem of what the product should do. The main cause of requirements problems is the problem of knowledge sharing and reuse, and ontology is an effective means to solve these two problems. Besides, objects (classes) do not always play a primary role in the field, and sometimes the relationship between objects (classes) is more important^[20]. Therefore, other associations between SREP objects (classes) need to be characterized. Thus, this paper uses the object-based ontology method to characterize SREP and define its semantics.

1) The elements set of SREP ontology

The set of ontology elements is, *Ontology Elements Set* = {*Concepts*, *Object-property*, *Data-property*, P^R , P^C , *Inherit-hierarchies*, *Relations*, *Instances*, *Map*, *Rules*}. The *Concepts*, *Inherit-hierarchies*, *Relations* and *Instances* constitute the basic skeleton of the ontology. "Object-property" and "Data-property" are two types of properties. "Object-property" is the bridge of the associated instances and "Data-property" is the bridge between the associated instance and the value. P^R and P^C represent the restriction of property and characteristic of property. The *Map* is the mapping between different levels of ontology. The *Rules* includes axioms and custom rules.

2) The construction of SREP ontology

Definition2 SREP ontology := $\langle \text{Concepts}, \text{Object-property}, \text{Data-property}, P^R, P^C, \text{Inherit-hierarchies}, \text{Relations}, \text{Rules}, \text{Instances} \rangle$.

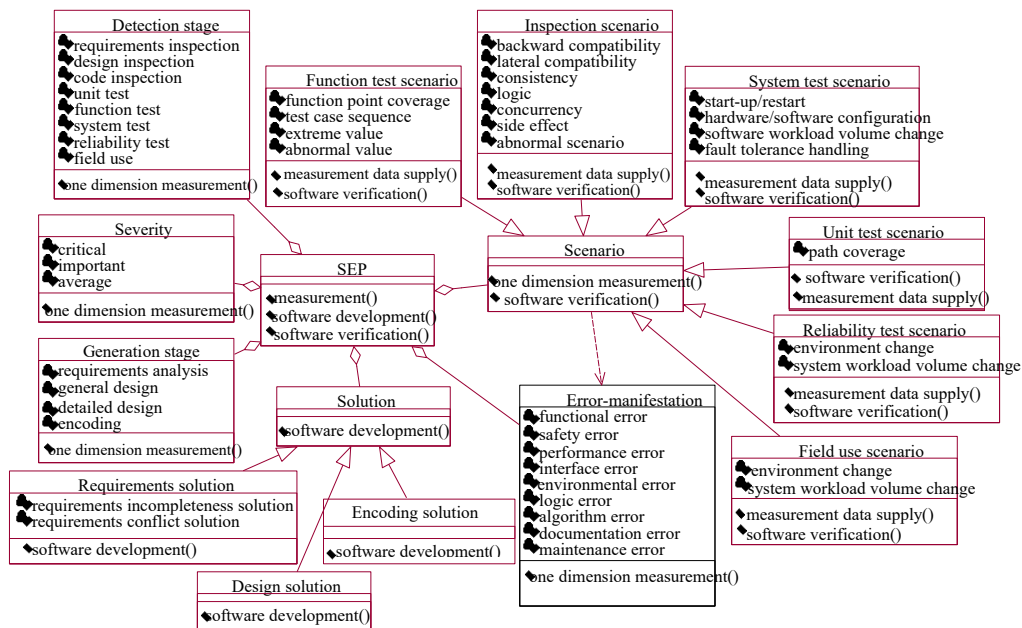


Figure 2. The class representation of SEP and its elements

Figure 3 shows the concept class of SREP ontology. The non-terminal concept classes are represented by “*”. The rest are terminal concept classes.

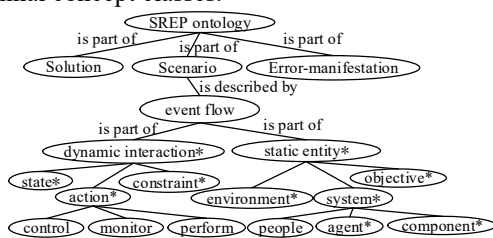


Figure 3. The hierarchies of SREP ontology concept class

Definition 3 The inheritance relationship is the mechanism in which the subclass in the SREP concept class hierarchy automatically shares the properties and structures of its parent class.

The definition shows that a new class can be obtained based on existing concept class. A part of concept relationship is shown in Table 1.

TABLE 1. A PART OF CONCEPT RELATIONSHIP OF SREP ONTOLOGY

Relationship type	Relationship	Property characteristic
subObjective	objective→objective	partial order
interact	system→environments	symmetry
has	system→goal	-

The left arrow is the source concept node and the right is the destination concept node. Take an example of *subObjective*. Its property characteristic is partial order. That is because it satisfies the reflexivity, symmetry and transitivity simultaneously, and it can be represented as $a \leq b$.

Furthermore, a part of the concept space of SREP ontology is shown as Figure 4.

B. The Representation of SREP Ontology

The software requirements (SRs) can be divided into

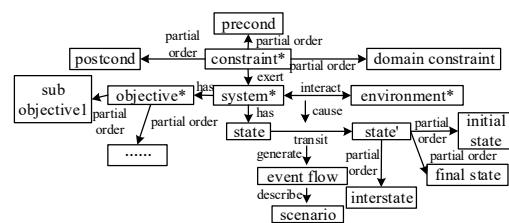


Figure 4. A part of the concept space of SREP ontology

internal requirements and external requirements. The external requirements can be further divided into explicit and implicit requirements. Thus SRs can be represented as, $SRs := \langle \text{Internal Requirements}, \text{External Requirements} \rangle$. Moreover the environments ontology can be constructed as, $\text{Environments Ontology} := \langle \text{Explicit Environments}, \text{Implicit Environments} \rangle$. The implicit environments can be represented as, $\text{implicit environments} = \{\text{stress environments} \{ \text{temperature environments}, \text{vibration environments}, \text{humidity environments}, \text{electricity environments}, \text{combined stress environments} \}, \text{EM environments}, \text{geography environments}, \text{climate environments} \}, \text{combined stress environments} = \text{temperature environments} \cup \text{vibration environments} \cup \text{electricity environments} \cup \text{humidity environments}$.

The software requirements engineering concerns the goals that the software system expects to achieve and the scenarios in which it happens [21]. Thus it can be represented as, $SRs := \langle \text{System} \langle \text{People}, \text{Agent}, \text{Component} \rangle, \text{Environments} \langle \text{Explicit Environments}, \text{Implicit Environments} \rangle, \text{Interact} \langle \text{Action} \langle \text{Control}, \text{Monitor}, \text{Perform} \rangle, \text{Constraint}, \text{State} \rangle, \text{Objective} \langle \text{Avoid}, \text{Cease}, \text{Achieve}, \text{Maintain} \rangle \rangle$.

The state transition model can be adopted to describe state transition and event flow, i.e., the formation of scenario, and the environments framework is obtained as,

$$\text{State} \langle \text{Initial state}, \text{Interstate}, \text{Final state} \rangle \neq \text{NULL}$$

Interact → *Transition*: $T(\text{state}, \text{state}')$

Transition: $T(\text{state}, \text{state}') \rightarrow \text{Event flow}$

Constraints <*Precond*, *Postcond*, *Domain Constraints*
{*Trigger*, *Pred*, *Synch*, *Conseq*}>

The *Constraints* act on *Interact*, *Transition* and *Event flow* respectively. The *Domain Constraints* include *Trigger*_{domain}, *Pred*_{domain}, *Synch*_{domain} and *Conseq*_{domain}. And the domain knowledge can be described by state transition of environment.

Construct the set of environments, *Environments* = {*explicit environments*, *implicit environments*}. The set of implicit environments is as shown before. Moreover, *temperature environments* = {*t-envi*₁, ..., *t-envi*_{s1}}, *vibration environments* = {*v-envi*₁, ..., *v-envi*_{s2}}, *humidity environments* = {*h-envi*₁, ..., *h-envi*_{s3}}, *electricity environments* = {*e-envi*₁, ..., *e-envi*_{s4}}, *combined stress environments* = {*c-envi*₁, ..., *c-envi*_{s5}}, *EM environments* = {*EM-envi*₁, ..., *EM-envi*_{s6}}, *geography environments* = {*g-envi*₁, ..., *g-envi*_{s7}}, *climate environments* = {*a-envi*₁, ..., *a-envi*_{s8}}. $s1, \dots, s8 \in \mathbb{N}$. *Interact* = {*interact*₁, ..., *interact*_n}, $n \in \mathbb{N}$. *State* = {*state*₁, ..., *state*_t}, $t \in \mathbb{N}$. *Constraints* = {*constraints*₁, *constraints*₂, ..., *constraints*_l}, $l \in \mathbb{N}$. All of them are obtained by inference of domain ontology.

1) Scenario I: A single sub-function of requirements decomposition graph (incomplete requirements scenario)

After describing the corresponding sub-function of the leaf node in the requirements decomposition graph as an instance of a certain granularity of the domain ontology, the node requirements can be represented as, $SRs_c := \langle \text{Agent}, \text{Environments}, \text{Interact}, \text{Objective} \rangle$, the corresponding SREP ontology can be represented as follows,

a) Scenario 1

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

sub scenario 1

$\exists (t\text{-envi}_s \notin \text{implicit environments}) \rightarrow ((t\text{-envi}_s \cup \text{implicit environments}) \not\subset \text{Environments}) (s_1 \leq s, s \in \mathbb{N})$

sub scenario 2

$\exists (v\text{-envi}_s \notin \text{implicit environments}) \rightarrow ((v\text{-envi}_s \cup \text{implicit environments}) \not\subset \text{Environments}) (s_2 \leq s, s \in \mathbb{N})$

sub scenario 3

$\exists (h\text{-envi}_s \notin \text{implicit environments}) \rightarrow ((h\text{-envi}_s \cup \text{implicit environments}) \not\subset \text{Environments}) (s_3 \leq s, s \in \mathbb{N})$

sub scenario 4

$\exists (e\text{-envi}_s \notin \text{implicit environments}) \rightarrow ((e\text{-envi}_s \cup \text{implicit environments}) \not\subset \text{Environments}) (s_4 \leq s, s \in \mathbb{N})$

sub scenario 5

$\exists (c\text{-envi}_s \notin \text{implicit environments}) \rightarrow ((c\text{-envi}_s \cup \text{implicit environments}) \not\subset \text{Environments}) (s_5 \leq s, s \in \mathbb{N})$

sub scenario 6

$\exists (EM\text{-envi}_s \notin \text{implicit environments}) \rightarrow ((EM\text{-envi}_s \cup \text{implicit environments}) \not\subset \text{Environments}) (s_6 \leq s, s \in \mathbb{N})$

sub scenario 7

$\exists (g\text{-envi}_s \notin \text{implicit environments}) \rightarrow ((g\text{-envi}_s \cup \text{implicit environments}) \not\subset \text{Environments}) (s_7 \leq s, s \in \mathbb{N})$

sub scenario 8

$\exists (a\text{-envi}_s \notin \text{implicit environments}) \rightarrow ((a\text{-envi}_s \cup \text{implicit environments}) \not\subset \text{Environments}) (s_8 \leq s, s \in \mathbb{N})$

b) Scenario 2

$\text{constraints}_k(\text{interact}_i, t\text{-envi}_j) \in \text{Constraints} (1 \leq k \leq l, 1 \leq i \leq n, s_1 \leq j, k \in \mathbb{N}, l \in \mathbb{N}, i \in \mathbb{N}, n \in \mathbb{N}, j \in \mathbb{N}, s_1 \in \mathbb{N})$

$\exists (\text{interact}_i \in \text{Interact}) \cap ((t\text{-envi}_j \notin \text{implicit environments}) \rightarrow ((t\text{-envi}_j \cup \text{implicit environments}) \not\subset \text{Environments}))$

c) Scenario 3

$\text{constraints}_k(\text{interact}_i, v\text{-envi}_j) \in \text{Constraints} (1 \leq k \leq l, 1 \leq i \leq n, s_2 \leq j, k \in \mathbb{N}, l \in \mathbb{N}, i \in \mathbb{N}, n \in \mathbb{N}, j \in \mathbb{N}, s_2 \in \mathbb{N})$

$\exists (\text{interact}_i \in \text{Interact}) \cap ((v\text{-envi}_j \notin \text{implicit environments}) \rightarrow ((v\text{-envi}_j \cup \text{implicit environments}) \not\subset \text{Environments}))$

d) Scenario 4

$\text{constraints}_k(\text{interact}_i, h\text{-envi}_j) \in \text{Constraints} (1 \leq k \leq l, 1 \leq i \leq n, s_3 \leq j, k \in \mathbb{N}, l \in \mathbb{N}, i \in \mathbb{N}, n \in \mathbb{N}, j \in \mathbb{N}, s_3 \in \mathbb{N})$

$\exists (\text{interact}_i \in \text{Interact}) \cap ((h\text{-envi}_j \notin \text{implicit environments}) \rightarrow ((h\text{-envi}_j \cup \text{implicit environments}) \not\subset \text{Environments}))$

e) Scenario 5

$\text{constraints}_k(\text{interact}_i, e\text{-envi}_j) \in \text{Constraints} (1 \leq k \leq l, 1 \leq i \leq n, s_4 \leq j, k \in \mathbb{N}, l \in \mathbb{N}, i \in \mathbb{N}, n \in \mathbb{N}, j \in \mathbb{N}, s_4 \in \mathbb{N})$

$\exists (\text{interact}_i \in \text{Interact}) \cap ((e\text{-envi}_j \notin \text{implicit environments}) \rightarrow ((e\text{-envi}_j \cup \text{implicit environments}) \not\subset \text{Environments}))$

f) Scenario 6

$\text{constraints}_k(\text{interact}_i, c\text{-envi}_j) \in \text{Constraints} (1 \leq k \leq l, 1 \leq i \leq n, s_5 \leq j, k \in \mathbb{N}, l \in \mathbb{N}, i \in \mathbb{N}, n \in \mathbb{N}, j \in \mathbb{N}, s_5 \in \mathbb{N})$

$\exists (\text{interact}_i \in \text{Interact}) \cap ((c\text{-envi}_j \notin \text{implicit environments}) \rightarrow ((c\text{-envi}_j \cup \text{implicit environments}) \not\subset \text{Environments}))$

g) Scenario 7

$\text{constraints}_k(\text{interact}_i, EM\text{-envi}_j) \in \text{Constraints} (1 \leq k \leq l, 1 \leq i \leq n, s_6 \leq j, k \in \mathbb{N}, l \in \mathbb{N}, i \in \mathbb{N}, n \in \mathbb{N}, j \in \mathbb{N}, s_6 \in \mathbb{N})$

$\exists (\text{interact}_i \in \text{Interact}) \cap ((EM\text{-envi}_j \notin \text{implicit environments}) \rightarrow ((EM\text{-envi}_j \cup \text{implicit environments}) \not\subset \text{Environments}))$

2) Scenario II: two sub-functions of requirements decomposition graph (conflict requirements scenario)

The requirements of sub-functions of any two leaf nodes in requirements decomposition graph are represented as, $SRs_{c1} := \langle \text{Agent}_1, \text{Environments}_1, \text{Interact}_1, \text{Objective}_1 \rangle$, $SRs_{c2} := \langle \text{Agent}_2, \text{Environments}_2, \text{Interact}_2, \text{Objective}_2 \rangle$. The corresponding sets of state are as follows, $\text{State}_1 = \{\text{state}_{11}, \dots, \text{state}_{t1}\}$, $\text{State}_2 = \{\text{state}_{12}, \dots, \text{state}_{t2}\}$, $t_1, t_2 \in \mathbb{N}$.

$\forall \text{interact}_i \in \text{Interact}, \text{interact}_j \in \text{Interact} (1 \leq i, j \leq n, i \in \mathbb{N}, j \in \mathbb{N}, n \in \mathbb{N})$,

a) Scenario 1

$\forall (\text{interact}_i \rightarrow \text{transition}_i: T (\text{state}_r, \text{state}_r')) \cap (\text{interact}_j \rightarrow \text{transition}_j: T (\text{state}_p, \text{state}_p')) \cap (\text{state}_r \in \text{State}) \cap (\text{state}_p \in \text{State}) \cap (\text{state}_r' \in \text{State}) \cap (\text{state}_p' \in \text{State})$,
 $\exists (\text{restricts}_q \notin \text{Restricts}) (1 \leq r, r' \leq t_1, 1 \leq p, p' \leq t_2, q \in \mathbb{N})$

b) Scenario 2

$\forall (\text{restricts}_q \in \text{Restricts}), \exists (\text{interact}_i \rightarrow \text{transition}_i: T (\text{state}_r, \text{state}_r')) \cap (\text{interact}_j \rightarrow \text{transition}_j: T (\text{state}_p, \text{state}_p'))$

$state_p' \cap ((state_r' \notin State) \cup (state_p' \notin State)) \quad (1 \leq r, r' \leq t_1, 1 \leq p, p' \leq t_2, 1 \leq q \leq l, q \in \mathbb{N})$

Error-manifestation: functional error, safety error.

Solution: A complete set of predefined environments, a combination of interactions and environmental conditions, and a set of domain constraints should be given to construct complete domain ontology.

V. THE EXPERIMENT

SREP can be used for a variety of applications, such as software inspection and testing guided by *Scenario* instance of one of its core components. This paper takes an example of Flying Control & Management System (FCMS) of Unmanned Aerial Vehicles (UAV). It adopts protégé to edit the SREP and its instances. Moreover it will introduce the application experiment of software inspection and system test.

A. The edit of SREP and its instance using protégé

This paper uses Protégé for ontology editing. For the sake of simplicity, the examples in this section only present 3 core classes of SREP, i.e., scenario, error-manifestation and solution. Corresponding to Figure 2, 3 and 4, Figure 5 shows that the SREP class and its constituent classes are related by the object property *isPartOf*. The SREP ontology includes 147 concept classes and 10 concept relationships. The two main SREP classes are the functional and the safety error pattern.

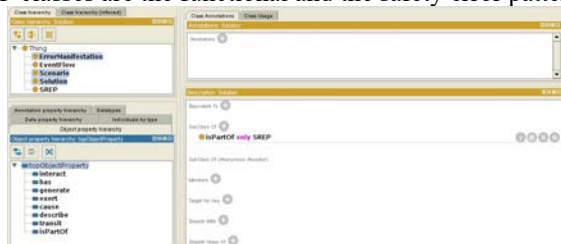


Figure 5. The object property "isPartOf"

B. The SREP-based software verification activity

This paper guides the verification process and measures the development process by using the *Scenario* element. Table 2 records the SREP error-manifestation, quantity distribution and corresponding scenario instances of UAV FCMS software detected through the requirements inspection. The software was developed in C language. The inspectors used the "inspection scenario" and its instances in the SREP *Scenario* during the inspection process. Table 2 also shows more safety errors were detected by requirements inspection. The "safety error" has certain similarities with "functional error" and is at a high level. Yet they differ in that failures caused by safety error might incur serious consequences, e.g., casualties or property damage, etc. Thus the severity level of most of the errors found in this phase is "critical".

TABLE 2. THE ERROR REPRESENTATION AND DISTRIBUTION OF SREP

Scenario instance type	Error-manifestation	Error No.
Geography environments error scenario	Functional error	1
Geography environments error scenario	Safety error	2
Pre-instruction error scenario	Safety error	3

-	Document description error	5
---	----------------------------	---

Table 3 presents a detailed description of some errors detected during the requirements inspection phase. The *Scenario* is the basis of test case and the inspection conducted includes a wealth of experiences and more complete knowledge, such as many external conditions and factors in the actual implementation of the test case, including the geography/climate environment. Therefore, it is beneficial for the inspectors to check the field use and the actual system behavior to detect problems such as incomplete requirements, etc. If the requirements items in the requirements document cannot cover all the scenarios, it may lead to the omission of requirements items. The instances of *Scenario* also provide the inspectors with a perceptual understanding. Table 2 shows that the number of errors found through the "geographic environment error scenario" are relatively large. This shows that the environment the UAV is performing its tasks in has not been considered adequately.

TABLE 3. A PART OF TYPICAL ERRORS DETECTED DURING REQUIREMENTS INSPECTION

Error-manifestation	Error description in detail	Severity
Functional error	Bind a line which has two adjacent waypoints locating on both sides of eastern longitude of 180°. When pass through eastern longitude of 180°, the flight path of UAV might be wrong.	critical
	Binding a line which has the target waypoint of latitude of 90° might cause the flight path of UAV to be wrong.	critical
Safety error	Fixed flight on the plateau might cause an unsafe flying state of UAV during the process of speculative navigation.	critical
	The omission of the flight switch instruction before the parachute landing fall might cause an unsafe instruction operation.	critical

Table 4 presents a detailed description of some errors detected during the system test phase. The type of error-manifestation here focuses on "safety error". Several instances of such scenario exist in the ontology library in advance and the related requirements for fault handling also exist in the requirements specifications. However the types of priority of the fault handling were omitted during the requirements inspection. This is because there are many types of failures that may occur, and the determination of their priority depends on the degree of understanding in the UAV flight control field. In view of the importance of the fault handling function and the severity of possible consequences, the function was still considered as a key item in the system test. Thus, the system test also accumulated new knowledge for SREP's ontology knowledge library. This also reflects the construction of the SREP ontology library is a process of continuous improvement and iterative iteration.

TABLE 4. A PART OF TYPICAL ERRORS DETECTED DURING SYSTEM TEST

Error-manifestation	Error description in detail	Severity
Safety error	Wrong handling sequence of the communication link failure and height fault when landing	critical
	Wrong handling sequence of the height fault and	critical

engine permanent fault when landing

VI. CONCLUSIONS

This paper presents the software error lifetime and proposes the SEP concept based on ODC. Furthermore, the SREP ontology in the requirements analysis phase is constructed, and the SREP ontology representation is studied. The experiment demonstrates the validity of the SEP-based verification activity and the feasibility of measuring the quality of development. However, the current use of this method is limited, and the experimental results obtained are still relatively lacking in quantity. This is because the construction of the ontology itself is a time-consuming and laborious task, which requires the participation of many experts such as domain experts and software engineers. Besides, the collection of error data is also a long-term work, especially in today's increasingly complex software systems. Software errors are caused by a variety of reasons, and with the development of new technologies, new errors and their causes continue to emerge. However, once the ontology is built, it will contain a wealth of experiences and domain knowledge, which makes it possible to reuse data resources and improve efficiency. In addition, the growing enrichment of error pattern library also lays the foundation for a comprehensive understanding and the quality assurance of software system.

ODC's defect analysis method 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 current large, open, dynamic and non-deterministic development trend of the software system will be an ongoing research topic.

REFERENCES

- [1] M. R. Lyu, Handbook of software reliability engineering, Los Alamitos, California: McGraw-Hill Companies, 1996, pp. 359, 364-367, 394.
- [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, 1992, 18(11), pp. 943-956.
- [3] R. Chillarege, Kothanda Ram Prasad, "Test and development process retrospective: a case study using ODC triggers," Proceedings of the International Conference on DSN'02. Washington, DC, USA: IEEE Computer Society, pp. 669-678.
- [4] R. Chillarege, "Understanding Bohr-Mandel bugs through ODC Triggers and a case study with empirical estimations of their field proportion," Proceedings of the 3rd International WoSAR'11. Washington, DC, USA: IEEE Computer Society, pp. 7-13.
- [5] Lutz, R.R., Mikulski, I.C., "Empirical analysis of safety-critical anomalies during operations," IEEE Transactions on Software Engineering, 2004, 30(3), pp. 172-180.
- [6] Nuno Silva, Marco Vieira, "Experience Report: Orthogonal Classification of Safety Critical Issues," Proceedings of IEEE 25th International Symposium on Software Reliability Engineering, 2014, pp. 156-166.
- [7] R.A. Paul, F. Bastani, I-Ling Yen, V.U.B. Challagulla, "Defect-based reliability analysis for mission-critical software," Proceedings of the 24th Annual International Computer Software and Applications Conference, 2000, pp. 439-444.
- [8] Tiejun Pan, Zheng Leina, Fang Chengbin, "Defect Tracing System Based on Orthogonal Defect Classification," Proceedings of the 2008 International Conference on Computer Science and Software Engineering, 2008, pp. 574-577.

- [9] LiGuo Huang, Vincent Ng, Isaac Persing, Mingrui Chen, Zeheng Li, Ruili Geng, Jeff Tian, "AutoODC: Automated generation of orthogonal defect classifications," Automated Software Engineering, Springer, 2015, 22(1), pp. 3-46.
- [10] Ferdian Thung, David Lo, Lingxiao Jiang, "Automatic Defect Categorization," Proceedings of the 19th Working Conference on Reverse Engineering, 2012, pp. 205-214.
- [11] Ruili Geng, Mingrui Chen, Jeff Tian, "In-Process Usability Problem Classification, Analysis and Improvement," Proceedings of the 14th International Conference on Quality Software, 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] Cf. T. R. Gruber., "A translation approach to portable ontologies," Knowledge Acquisition, 1993, 5(2), pp. 199-220.
- [14] Thomas R. Gruber., "Toward Principles for the Design of Ontologies Used for Knowledge Sharing," International Journal of Human-computer Studies, 1995, 43(5-6), pp. 907-928.
- [15] Renzuo Xu, "The testing method based on software knowledge," J. Wuhan Univ. (Nat. Sci. Ed.), 2000, 46(1), pp. 61-62.
- [16] C. Alexander, The Timeless Way of Building, UK: Oxford University Press, 1979, pp. 5-20.
- [17] 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, 2009, 35(6), pp. 723-727.
- [18] Xizi Huang, Software reliability, safety and quality assurance, Beijing: Publishing House of Electronics Industry, 2002.
- [19] Zhijin, Linliu, Yingjin, Software requirements engineering method, Beijing: Science Press, 2008.
- [20] LU Ruqian, JIN Zhi, CHEN Gang, "Ontology-oriented requirements analysis," Journal of Software, 2000, 11(8), 1009-1017.
- [21] Jon G. Hall, Ivan Mistrik, Bashar Nuseibeh, Dr Andrés Silva, "Editorial relating software requirements and architectures," IEE Proc.-Softw. 2005, 152(4), pp. 141-142.



Xuan Hu received her bachelor's degree in mathematics and the doctor's degree in Aerospace Science and Technology both from Beihang University, China. She entered the postdoctoral workstation of CEPREI Laboratory of Ministry of Industry and Information Technology of People's Republic of China in 2011. She is currently a senior engineer in Information Security Research Center of CEPREI Laboratory. Her main research interests include ontology modeling, software requirements engineering, software reliability, self-adaptive software and highly reliable fault tolerant technology.



Jie Liu received his bachelor's degree in computer science from Xidian University, China. He is currently the director of the Information Security Research Center (Industrial Control System Information Security Evaluation Laboratory) of CEPREI Laboratory Ministry of Industry and Information Technology of People's Republic of China. His main research interests include information security, software reliability and security, and software evaluation technology.



Wu Bo received his Bachelor degree in printing engineering in Wuhan University, China, in 2007, the Ph.D. degree in pulping and papermaking engineering from South China University of Technology, China, in 2012. He is currently a senior engineer in the information security research center, China CEPREI Laboratory. His research interests include industrial process control and computer simulation, information security of industrial control systems, information test and vulnerability discovery.