2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP) | 979-8-3503-0037-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSE-SEIP58684.2023.00044

Identifying Defect Injection Risks from Analysis and Design Diagrams: An Industrial Case Study at Sony

Yoji Imanishi
Sony Global Manufacturing & Operations
Tokyo, Japan
https://orcid.org/0000-0002-1379-3084

Kazuhiro Kumon
Sony Global Manufacturing & Operations
Tokyo, Japan
https://orcid.org/0000-0001-6707-3093

Shuji Morisaki
Graduate School of Informatics Nagoya University
Nagoya, Japan
https://orcid.org/0000-0002-8290-0584

Abstract*—Identifying the origins of potentially injected defects in implementation activities during requirement analysis and design activities is challenging, but leads to the prevention of defects, which are burdensome to correct. This study investigates whether such defect injection risks can be generalized and defined by risk occurrence conditions of the objects in the existing analysis and design diagrams and whether the defined defect injection risks are applicable to other analysis and design diagrams. Specifically, we identify defect categories, which are injected after analysis and design activities and subsequently detected during system testing of commercial products developed at Sony. Then, regarding the defect categories as the exposed defect injection risks, we define defect injection risks with the objects defined in the analysis and design diagrams of the products. Each defect injection risk consists of risk description, diagram type, and occurrence conditions of objects in the diagrams. Afterwards, we evaluate whether the defined defect injection risks appear in the analysis and design diagrams of three different products under development and five publicly available analysis and design diagrams. The results showed that three defect injection risks were defined and that the two risks appear in the analysis and design diagrams of the three products and the remaining one appears in the analysis and design diagrams of two products. The results also showed that one defect injection risk is present in all five publicly available analysis and design diagrams, and two risks appear in four of the diagrams. The three defect injection risks are general enough to identify risks in analysis and design diagrams from other domains. Developers can be more cautious about the risks and prevent defect injections with the defect injection risks.

Keywords*— *defect prevention, use case driven development, model-based development, risk ontology

I. Introduction

Software quality has become more important as the number of devices and appliances relying on software increases. A promising quality improvement approach is validation using requirement analysis and design artifacts. Validating and correcting analysis and design artifacts and identifying the origins of potentially injected defects can prevent defects in subsequent coding and testing activities. Raghuraman et al. compared the number of issue reports describing bugs between GitHub repositories with and without UML diagrams [1]. On average, repositories without UML diagrams contain more issues on bugs than those with UML diagrams.

Many studies have proposed analysis and design validation approaches [2][3][4][5][6][7]. Conradi et al. devised a method to detect omissions and inconsistencies by comparing different kinds of UML diagrams [2]. Egyed developed a method to detect inconsistencies by predefined rules comparing sequence diagrams and class diagrams [3]. Lange et al. investigated detected defects in UML diagrams and categorized the defects into defect types [4]. The results showed that defect types include "method not called in sequence diagram" and "use case without sequence diagram."

Some studies have proposed methods to verify consistencies in analysis and design diagrams with formalized conditions. David et al. developed a formal verification method to verify UML diagrams using a series of events and pre- and post-conditions defined in the object constrained language [6]. Menher et al. devised a method to detect a critical pair of use cases by applying graph transformation rules on activity diagrams [7]. Although these studies aimed at detecting omissions and inconsistencies in analysis and design diagrams and pairs of conflicting use cases, they did not detect nor refer to potential defect injection risks in subsequent development activities. Moreover, most of these studies required a large effort because exhaustive verification or formalization of the analysis and design diagrams was assumed.

Other studies have identified risks from design diagrams [8][9][10][11]. These risks include not only defects in the design diagrams such as omissions and inconsistencies but also the runtime risks and origins of potentially injected defects in subsequent development activities without careful considerations. UML HAZOP [8] detects risks by identifying deviations in UML diagrams by applying guidewords to use case diagrams, sequence diagrams, and state machine diagrams. CORAS [9] identifies security risks by applying guidewords to UML diagrams. VIKOR [10] detects reliability risks by performing FMEA (Failure Mode and Effect Analysis) on events included in a use case. Specifically, VIKOR extracts events from a use case, performs FMEA on each event, and identifies reliability risks such as noise immunity of the communication channel. Oveisi et al. proposed a method to identify reliability risks of objects and events extracted from sequence diagrams by performing FTA (Fault Tree Analysis) [11]. These studies identified risks of potential defects, which were not defects when the design diagrams were created, but could be injected in subsequent development activities. Although these studies did not require exhaustive formalization or evaluation, the deviation analyses used brute force approach using guidewords, failure modes, or fault trees to exhaustively enumerate potential deviations.

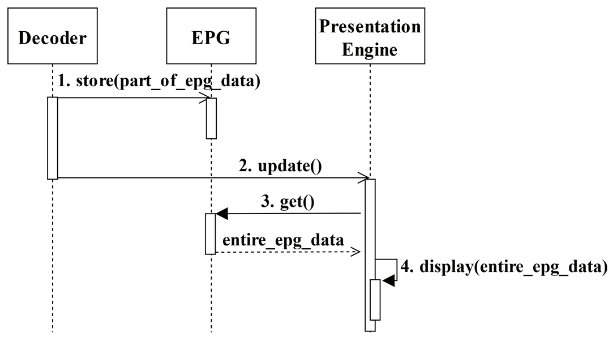

Fig. 1. Example of a sequence diagram with a defect injection risk

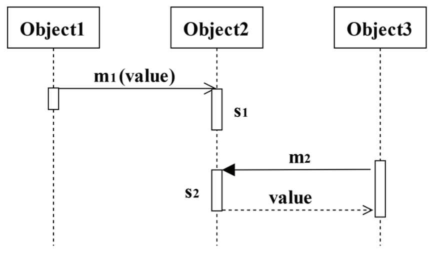

Fig. 2. Generalized example of a sequence diagram with a defect injection risk

TABLE I. Example of a Defect Injection Risk I_A

Risk description d_a	Type of diagram t_a	Occurrence conditions of objects u_a
Incorrect execution results due to an execution dependency	*S*	• Execution specifications s_1 and s_2 of Object2 have an execution dependency. (Ex. Execution specification s_1 should finish before execution specification s_2 is started) • Execution specification s_1 is asynchronously started.

Thus, analysts must determine whether each deviation may occur. Such analyses are limited to high-cost absorption domains such as safety- and security-critical domains because they are burdensome.

Some studies have constructed ontologies for early risk identification [12][13][14]. The risk ontologies consist of general risks along with the software development lifecycle such as a larger size of the software, requirement stability, and domain knowledge of the development team. These studies do not require exhaustive formalization or evaluation or brute force deviation analysis.

No studies referred to the identification of general, not specific to safety or security, defect injection risks in subsequent development activities to design activities from analysis or design diagrams. The defect injection risks are risks of omitted or incorrect implementation. The risks can be predicted from analysis and design diagrams but are not defects in the diagrams at the point of analysis and design activities. For example, a sequence diagram can include (a) a message between objects in the same computer and (b) a message between objects in different computers joined by an unstable network connection. Although the messages require different implementations, the sequence diagram represents the messages as the same type of elements (arrows). Programmers in subsequent development activities cannot always pay attention to such implicit difference from the sequence diagram. More specifically, the programmers must consider resend and timeout for message (b) due to the unstable network, whereas resend and timeout for message (a) are not always necessary. Note that the sequence diagram does not include a "missing resend and timeout" defect in the design activity because detailed implementations (message within the same computer and message via an unstable network) are usually not determined at this point. However, once the detailed implementations for the sequence diagram are determined, the defect injection risk for missing the resend and timeout in subsequent coding activity and failing to validate the resend and timeout in subsequent testing can be identified.

This paper aims to generalize the defect injection risks using elements in the analysis and design diagrams. First, we categorized defects detected in the testing of commercial products at Sony to identify the candidates of defect injection risks. Second, we generalized the categories of defects as defect injection risks by defining risk occurrence conditions using elements of the analysis and design diagrams developed in the products. Finally, we evaluated whether the defect injection risks are present in the analysis and design diagrams of other commercial products at Sony and publicly available ones. This study aims to answer the following research questions:

RQ1: Can defect injection risks be identified and generalized from analysis and design diagrams?

RQ2: Are defect injection risks applicable to other analysis and design diagrams?

II. Background and Related Research

A. Defect Injection Risk

Defect injection risks indicate behaviors and structure that are defined in analysis and design diagrams and need careful implementation and validation in subsequent development activities. Reviews with analysis and design diagrams can detect defect injection risks, which predict the behaviors and structure require careful implementation and validation in subsequent development activities. After the reviews, being cognizant of defect injection risks can prevent injecting defects in subsequent coding activity and failing to detect such defects in subsequent testing activity. Prevention should eliminate the rework effort to correct defects.

Defect injection risks differ from defects, including inconsistent and omitted objects, noted in analysis and design diagrams. Although analysis and design diagrams are accurate when they are created, they can be vulnerable to defect injection risks because they cannot identify future defects generated in subsequent development activities. Fig. 1 shows an example of defect injection risk in a sequence diagram for EPG (Electronic Programming Guide) for TV products. First, Decoder sends "1. store(part_of_epg_data)" asynchronous message to EPG. EPG receives and updates data in EPG. Second, Decoder sends "2. update()" asynchronous message to PresentationEngine. Third, PresentationEngine sends "3. get()" synchronous message to

TABLE II. EXAMPLE OF ELEMENTS OF ANALYSIS AND DESIGN DIAGRAMS

Diagram type	Element	Description
C: Class diagram	Class	Class is a container of classifier whose features are attributes and operations. A class describes a set of objects that share the same specifications of features, constraints, and semantics.
	Relationship	Relationship represents an abstract interaction, including Association, Aggregation, Dependency, and Generalization, between classes.
D: domain diagram	Class	Class is a kind of classifier whose features are attributes and operations. A class describes a set of objects that share the same specifications of features, constraints, and semantics.
	Relationship	Relationship represents an abstract interaction, including Aggregation and Generalization, between classes.
U: Use case diagram	Actor	Actor specifies a role played by a user or any other system that interacts with the subject.
	Use case	Use case specifies a set of actions performed by its subjects, which yields an observable result for one or more actors or other stakeholders of the system.
	Relationship	Relationship represents an abstract interaction, including Include and Extend, between use cases.
	Extension Point	Extension Point identifies a point in the behavior of a use case where the behavior can be extended by the behavior of some other (extending) use case, as specified by an Extend relationship.
R: Robustness diagram	Boundary	Boundary is a type of object and interfaces with system actors
	Entity	Entity is a type of object and represents system persistent data.
	Controller	Controller is a type of object and mediates between boundaries and entities.
	Relationship	Relationship represents an abstract interaction including communication associations, data flow, and control flow among objects.
S: Sequence diagram	Lifeline	Lifeline represents that an individual object participates in the interaction.
	Message	Message represents a flow from sender to receiver. Message type includes Asynchronous, Synchronous, Reply, Object creation, and Object deletion.
	Execution Specification	Execution Specification represents the execution of a unit of behavior or action within the Lifeline.
	Combined Fragment	Combined Fragment is defined by an interaction operator and corresponding interaction operands including opt, loop, and break.

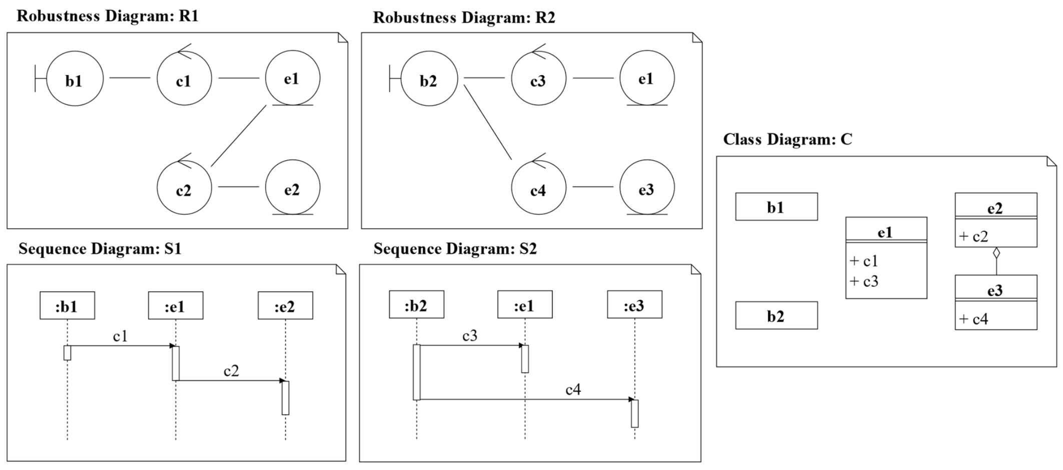

Fig. 3. Relationships between analysis and design diagrams

EPG and EPG sends "entire_epg_data" message to PresentationEngine. Fourth, PresentationEngine invokes "4. display(entire_epg_data)" of PresentationEngine. In this sequence diagram, if the execution specification after receiving "1. store(part_of_epg_data)" message does not finish before EPG receives "3. get()" message, the results "entire_epg_data" may be incorrect. If the asynchronous message "1. store(part_of_epg_data)" cannot be replaced with a synchronous one, developers must pay attention to the implementation and validation of "3. get()" and "1. store(part_of_epg_data)" messages. If the asynchronous message can be replaced with a synchronous one, developers can detect the asynchronous message as a defect and correct the message type in the sequence diagram. However, if the message type cannot be changed due to other constraints such that Decoder must send and notify other messages, developers must pay attention to the dependency between the execution specification and the message.

Defect injection risk I_k is defined as tuple $I_k = (d_k, t_k, u_k)$, where d_k denotes the description of defect injection risk I_k, t_k denotes the types of diagram, and u_k denotes the occurrence conditions of elements in the diagrams. Fig. 2 shows a generalized example of a sequence diagram with the defect injection risk I_a shown in Fig. 1. In Fig. 2, Object2 receives message m_1 and starts execution specification s_1. If Object2 receives message m_2 and starts execution specification s_2 before Object2 finishes execution specification s_1, return value "value" may be incorrect. Table I defines risk I_a for this example. Developers will be able to prevent injecting defects indicated by defect injection risk I_a, if they consider all of the execution dependencies of each message. However, this will be difficult in real use software because the number of asynchronous message and execution specifications can be much larger. The defect injection risk for a given sequence diagram limits the consideration to specific area, whereas without such a defect injection risk, developers must consider all dependencies.

B. Analysis and Design Diagrams

Stepwise refinement approaches, including use case driven development, elaborate the structure and behavior of software using analysis and design diagrams. Specifically, the structure is elaborated describing use cases U, domain diagram D, and class diagram C. The behavior is elaborated in the order of use case $U = \{U_1, U_2, \dots, U_n\}$, the robustness diagram $R = \{R_1, R_2, \dots, R_n\}$, and then the sequence diagram $S = \{S_1, S_2, \dots, S_n\}$. Use case U_k corresponds to robustness diagram R_k and sequence diagram S_k. Consistencies between the structure and behaviors can be checked among these diagrams.

Fig. 3 shows the relationships among diagrams R, C, and S. Class diagram C includes classes corresponding to objects (b_1, c_1, c_2, e_1, and e_2) in robustness diagram R_1 and sequence diagram S_1 and classes corresponding to objects (b_2, c_3, c_4, e_1, and e_3) in R_2 and S_2. Some objects appear in multiple diagrams among the same type of diagrams. For example, object e_1 appears in diagrams R_1 and R_2. Additionally, as defined in class diagram C, the aggregation relationship between objects e_2 and e_3 hold in diagrams R_1, R_2, S_1, and S_2.

This study refers to the types of objects in a certain diagram as its elements. Table II shows an example of the elements of analysis and design diagrams. To address RQ1, we attempt to define the defect injection risk using analysis and design diagrams and their elements.

C. Related Research

1) Defect Detection in Design Diagrams

Previous studies have proposed methods to detect omissions and inconsistencies in design diagrams [2][3][5][15]. Conradi et al. proposed a method to detect inconsistencies by comparing two or more UML diagrams [2]. The results of the evaluation showed that their proposed method detects defects other than those in usual UML diagram reviews. Egyed proposed a method to detect inconsistencies among UML sequence diagrams and class diagrams using 24 predefined rules, including "Name of message must match an operation in receiver's class." [3] Rao et al. proposed a method to detect inconsistencies among design diagrams by 13 predefined rules [16]. The redefined rules check for inconsistencies among class and sequence diagrams, sequence and collaboration diagrams, class and state machine diagrams, sequence and state machine diagrams, use case and class diagrams, and activity and class diagrams.

Kamalrudin et al. proposed a method to detect defects from use case scenarios written in a natural language employing a tool to match the templates and the written use case scenarios [17]. The templates are common abstracted interactions extracted from use cases. Hausmann et al. proposed a method and implemented a tool to detect conflicts in functional requirements by predefined graph transformation rules. The graph transformation rules check for dependencies and constraints among the objects described in use case scenarios [18].

Jurkiewicz et al. proposed a method called H4U to detect omissions of events in use cases [19]. H4U uses the eleven guidewords defined by Redmill [20] to detect omissions of events for alternative flows in the use case. An evaluation with 18 students and 82 practitioners showed that H4U can detect numerous event omissions. Srivatanakul et al. proposed a method to elicit security requirements using HAZOP [21]. Specifically, their method applies HAZOP guidewords to elements of use case scenarios and use case diagrams. As an example, they applied the guideword "more" to a use case "purchase" and identified a security requirement "Excessive order cannot be performed."

Bazyan and Krashuak proposed a metric to measure the number of UML document assessments by investigating the update histories of UML documents, consistencies among UML documents, and consistencies of the operator names among UML diagrams [22]. They also implemented a tool for the proposed method and evaluated their method via usability interviews. David et al. implemented a tool to perform FMEA on software components defined by UML or SysML [6]. Their tool applies failure modes categorized by software component types to the software components in UML or SysML using pattern matching. Mens et al. proposed a method to detect inconsistencies among UML diagrams by applying graph transformation rules [23]. For example, the graph transformation rules detect inconsistencies, including the names of elements in the UML diagrams. Jurack et al. proposed a method to indicate omissions and inconsistencies in the activity diagrams by extracting pre- and postcondition rules of the elements in the activity diagrams [24].

The above studies aimed to detect defects in use cases and design diagrams themselves. In contrast, this study aims to identify defect injection risks in subsequent development activities from the design diagrams.

2) Risk Identification in Design Diagrams

Some studies detected risks in runtime or defect injection risks in subsequent development activities after design activities from design diagrams instead of detecting omissions and inconsistencies in design diagrams. Such studies detected security risks [9][21], safety risks [8], and reliability risks [10][11]. CORAS [9] identifies security risks by applying guidewords to the elements in the given UML diagrams. For example, CORAS identified a risk of "unauthorized transfer of money" by applying the guideword "other than" to a sequence diagram specified from the use case "payment" for an online shopping service.

Guiochet proposed UML HAZOP for human-robot interactions [8]. It identified potential safety risk behaviors described in the UML use case, sequence, and state diagrams by applying six guidewords: "no/none," "other than," "as well as," "part of," "early," and "late." For example, a safety risk "the patient tries to stand up while the robot is not properly positioned" was identified with a use case "the robot is in front of the patient" and the guideword "no/none."

Studies have identified risks in runtime [10][11]. VIKOR identifies risks by performing FMEA for events extracted from a use case. For example, identified risks from an extracted event "data transmission from a sender to a receiver" were: "A device sending data may malfunction," "A device receiving data may malfunction," "Sent data may be incorrect," and "Communication bandwidth may be narrow due to noise." Oveisi et al. proposed a method to identify risks by performing FTA on events and objects described in sequence diagrams [11]. An example of their risks was missing considerations for the startup order of the sub-components in the runtime. Mehner et al. proposed a method to detect critical pairs among use cases by applying graph transformation rules to activity diagrams [7]. As an example, a use case conflict was detected in the use cases "paying flight tickets" and "redeeming flight tickets."

3) Code Smell

Code smell is a set of design anti-patterns (design risks) identified from the source code. Refactoring the identified design anti-patterns allows the source code to be easily evolved or modified. Fowler defined refactoring as "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure." The main idea is to improve the internal structure (design), while avoiding future problems, especially for maintenance [25]. Kim et al. investigated the benefit of eliminating code smells [26]. Mofa et al. proposed a tool to detect code smell patterns to recommend refactoring [27]. Some studies prioritized identified code smells for refactoring [28][29]. While these studies focused on change risks, our study aims at identifying defect injection risks from design diagrams.

4) Ontology

Some studies have constructed ontologies for early risk identification [12][13][14]. Menezes et al. identified general software development project risks [12]. The project risks include the size of the software, requirement stability, and development experience in the same domain. Abioye et al. proposed an approach to estimate software development risks along with their risk ontology. The risk ontology consists of potential risks along with the software development lifecycle, including requirement elicitation and requirement analysis [13]. Tsoumas and Grizalis proposed a security risk management method with risk ontology [14]. Because these studies employed risk ontology, they did not require comprehensive defect detection or risk identification. However, none of them identified defect injection risks in subsequent development activities using analysis or design diagrams.

III. Case Study

A. Goal and Research Questions

The goal of this study is to investigate whether defect injection risks can be identified from analysis and design diagrams and whether the identified risks are applicable to other analysis and design diagrams. Specifically, this study answers the following research questions:

- RQ1: Can defect injection risks be identified and generalized from analysis and design diagrams?
- RQ2: Are defect injection risks applicable to other analysis and design diagrams?

To answer these research questions, this study conducted a case study at Sony. For RQ1, an analyst manually categorized defects detected during system testing to find exposed defect injection risks. Afterwards, the analyst investigated whether the defect injection risks could be identified in analysis or design activities. This investigation assumed that the defect injection risks can be identified in analysis or design activities when the risks are defined with occurrence (exposure) conditions using the elements in the analysis or design diagrams developed for the products. For RQ2, an analyst evaluated whether the defined risks are applicable to analysis or design diagrams developed in other products and whether the defined risks are applicable to publicly available analysis and design diagrams.

TABLE III. Products for RQ1

ID	Product	Components
1-1	Professional video camera A	User interface components
1-2	Professional video camera B	User interface components
1-3	Professional video camera C	User interface components
1-4	Professional video camera D	All components
1-5	Consumer camera	All components
1-6	Security camera	All components
1-7	Medical display	All components

TABLE IV. Products for RQ2

ID	Product	Components
2-1-1	Blu-ray disc player & recorder A	All components
2-1-2	Blu-ray disc player & recorder B	All components
2-1-3	Professional video camera E	All components

TABLE V. Publicly Available Diagrams for RQ2

ID	Domain	Diagram Type	Number of diagrams	Reference
2-2-1	Online bookstore	*R*	7	[30]
2-2-2	Automated teller machine	*S*	4	[31]
2-2-3	Address book	*S*	10	[32]
2-2-4	Library system	*S*	4	[33]
2-2-5	Employee attendance system	*R*	6	[34]

B. Case Study Selection

For RQ1, we selected analysis and design diagrams and defect repositories of seven commercial products developed at Sony. Each product contains analysis and design diagrams and a defect repository. Table III summarizes the selected products. Each software consists of sub-components with the corresponding development sub-teams. As shown in Table III, we did not select the entire software of a product but sub-components for software 1-1, 1-2, and 1-3 because we could easily ask questions and have discussions with the corresponding development teams. We considered that these conversations with the developers were more important than analyzing all the components.

For RQ2, we selected the analysis and design diagrams of three commercial products under development at Sony and five publicly available analysis and design diagrams. Here, products under development were used because the software was not exposed to defect injection risks and the developers also considered and recalled such risks. Table IV summarizes the selected products. Similar to the criterion for RQ1, we selected products that we can easily ask questions and have discussion with the development teams. Additionally, we selected five publicly available analysis or design diagrams via a search engine to investigate whether the identified risks are applicable to other software than that developed at Sony. Table V summarizes the selected analysis and design diagrams.

C. Data Collection

For defect injection risks in RQ1, we used the defects detected during system testing of products shown in Table III. Table VI shows the recorded items in the defect repositories. We selected defects, which were injected in subsequent activities after requirement analysis and were detected in

TABLE VI. RECORD ITEMS IN DEFECT REPOSITORIES FOR RQ1

Name	Type	Description
Defect description	Free text	Explanation of the defects, including phenomena, reproduction procedure, and expected results
Detected activity	Choice	One of the following: requirements, design, coding, integration testing, or system testing
Injected activity	Choice	
Cause	Free text	Cause of the defect
Correction	Free text	Explanation to fix the defect
Component	Choice	Name of sub-component of software

TABLE VII. DEFECT TYPES DEFINED IN [37]

Type	Description
Logic	Defects made with comparison operations, control flow, and computations and other types of logical mistakes
Interface	Mistakes made when interacting with other parts of the software such as an existing code library, a hardware device, a database, or an operating system
Timing	Defects that are possible only in multi-threaded applications where concurrently executing threads or processes use shared resources
Resource	Mistakes made with data, variables, or other resource initialization, manipulation, and release
Function	Functionality is missing or designed incorrectly

system testing using the "detected activity," "cause," and "injected activity" records defined in Table VI. Specifically, we selected defects detected in system testing using the "detected activity" record. Then, we narrowed down the defects to defects injected after requirement activity using the "injected activity" record. Finally, we further narrowed down the defects to defects that could be detected prior activities using the "cause" record. In Sony, some detected defects were analyzed in retrospective meetings (Kaizen meetings) after the development to investigate prevention or detection in prior activities to improve the practices and processes. The "cause" record included the analysis results of the meetings. Additionally, we limited the defects by the "component" record for products 1-1, 1-2, and 1-3. For RQ1 and RQ2, analysis and design diagrams were use case, robustness, sequence, domain, and class diagrams.

We selected publicly available analysis and design diagrams to answer RQ2 using the following procedure:

1) Web search with a search engine: We searched books and articles with search keywords "use case driven," "ICONIX," or "UML development" using Google scholar. If the books or literature in the search results referred to diagrams in other books or articles, we referred to these books or articles.

2) Sufficient diagrams: We checked whether the required diagrams (diagram type) defined by the defect injection risks t_k were included or not.

D. Analysis Procedure

For RQ1, Analyst A initially identified and generalized defect patterns, which can be regarded as exposed defect injection risks. The analyst used ODC (Orthogonal Defect Classification) [35], which is a major analysis technique to identify frequent defect patterns (Procedure 1-1). Then Analyst B validated the results. Afterwards, Analyst A generalized the defect patterns as defect injection risks using the analysis and design diagrams and their elements (Procedure 1-2). Analyst B validated the results of Procedure 1-2.

For RQ2, Analyst A evaluated whether the defect injection risks in RQ1 were applicable to the analysis and design diagrams in products in development at Sony (Procedure 2-1). Developers of each product validated the results of Procedure 2-1. Furthermore, we evaluated whether the defect injection risks were applicable to publicly available analysis and design diagrams (Procedure 2-2). Analyst C validated the results of Procedure 2-2. All analysts were practitioners with product domain knowledge and five or more model-based development experience. Analysts A and B are the authors of this paper. The developers of the products and Analyst C are not.

Procedure 1-1: Defect pattern identification Analyst A identified defect patterns using ODC in defects described in Section III.C for each product shown in Table III. The analysis used the ODC attribute "defect type" and recorded item "component," which are common attributes to reveal categories of defects in specific components [36]. We used the ODC attribute "defect type" defined in the literature [37] (Table VII). Analyst A selected defect categories, which were common among the products. Analyst B validated the identified defect patterns. If Analyst B had questions or concerns about the defect patterns, Analysts A and B discussed until a consensus was reached. If necessary, the defect pattern was changed.

Procedure 1-2: Defect injection risk definition Analyst A defined defect injection risks using the analysis and design diagrams and their elements for the defect patterns identified in Procedure 1-1. Analyst B validated the defined defect injection risks. If Analyst B had questions or concerns about the defect injection risks, Analysts A and B discussed until a consensus was reached. If necessary, the defect injection risk was changed.

Procedure 2-1: Evaluation with diagrams developed at Sony Analyst A evaluated whether the defect injection risks defined in Procedure 1-2 were present in the analysis and design diagrams developed for the products shown in Table IV. For each product, the results included which defect injection risk was applicable to the analysis and design diagrams and their elements. Two or more developers of the products validated the results for their own products. If the developers had questions about the results or disagreed, they discussed with Analyst A until a consensus was reached. If necessary, the result was changed.

Procedure 2-2: Evaluation with publicly available diagrams Analyst A evaluated whether the defect injection risks defined in Procedure 1-2 appeared in the publicly available analysis and design diagrams shown in Table V. If publicly available diagrams included omissions or ambiguity, Analyst A complemented or corrected the diagrams because we observed omitted and incorrect objects in publicly available analysis and design diagrams in our preliminary survey. The evaluation results employed the same format as that in Procedure 2-1. Analyst C validated the results. Analyst C also validated the complements and corrections, if applicable. If Analyst C had questions or concerns, Analysts A and C discussed until a consensus was reached. If necessary, the results, complements, and corrections were changed.

TABLE VIII. IDENTIFIED DEFECT PATTERNS

ID	Description	Product ID						
		1-1	1-2	1-3	1-4	1-5	1-6	1-7
D_1	Data access confliction	15%	9%	20%	14%	23%	9%	0%
D_2	Insufficient performance due to resource shortages	0%	0%	0%	12%	12%	11%	14%
D_3	Insufficient exceptional or alternative implementations for specific parameter values	26%	27%	26%	10%	0%	7%	38%

TABLE IX. DEFECT INJECTION RISKS

ID	Risk description d_k	Type of diagrams t_k	Occurrence condition u_k
I_1	Insufficient implementation to prevent data access confliction (Implementation of an object or controller should consider the possibility that the value is changed by another implementation of the object or controller)	*U, R* and either *C* or *D*	• Two use cases U_1 and U_2 in use case diagram *U* can be concurrently executed. • (1) or (2) is applicable. (1) Controller c_1 in robustness diagram R_1 (corresponding to U_1) and controller c_2 in robustness diagram R_2 (corresponding to U_2) have connection to the same entity e_1. (2) Controller c_1 has connection to entity e_1 in robustness diagram R_1 and controller c_2 has connection to entity e_2 in robustness diagram R_2. Class diagram *C* or domain diagram *D* defines a relationship between entities e_1 and e_2.
		U, S and either *C* or *D*	• Two use cases U_1 and U_2 in use case diagram *U* can be concurrently executed. • (1) or (2) is applicable. (1) Message m_1 in sequence diagram S_1 (corresponding to U_1) and message m_2 in sequence diagram S_2 (corresponding to U_2) are sent to the same object o_1. (2) Message m_1 in sequence diagram S_1 is sent to object o_1 and message m_2 in sequence diagram S_2 is sent to object o_2. Class diagram *C* or domain diagram *D* defines a relationship between objects o_1 and o_2.
I_2	Insufficient implementation for performance or resource shortage (Implementation of receiver should consider the variance of the time required by the sender, depending on data size)	*R* and either *C* or *D*	• Controller c_1 has a connection to entity e_1 in robustness diagram R_1. • Class diagram *C* or domain diagram *D* defines that entity e_1 has various sizes of data.
		S and either *C* or *D*	• Object o_1 in sequence diagram S_1 has execution specification s_1 for data manipulation. • Class diagram *C* or domain diagram *D* defines that the object o_1 has various sizes of data.
I_3	Insufficient exceptional or alternative implementations for specific values of parameters (Implementations of a controller or object should consider alternative or exceptional cases, depending on the specific values)	*R* and either *C* or *D*	• Controller c_1 has a connection to entity e_1 in robustness diagram R_1. • Domain diagram *D* or class diagram *C* defines the value range of entity e_1. • Controller c_1 defines the operation requiring the consideration for the value of entity e_1.*
		S and either *C* or *D*	• Execution specification s_1 of object o_1 receives message m_1 in sequence diagram S_1. • Domain diagram *D* or class diagram *C* defines the value range of attribute a_1 of object o_1. • Execution specification s_1 defines the operation requiring the consideration for the value of attribute a_1. **

* If the operation defined in controller c_1 refers to the value of entity e_1, the operation must consider exceptional or alternative implementations depending on the value. If the operation updates the value of entity e_1, the operation c_1 must verify whether the value is appropriate or not.
** If the implementation of execution specification s_1 refers to the value of attribute a_1, the implementation must consider exceptional or alternative implementations depending on the value. If the implementation updates the value of attribute a_1, the implementation must verify whether the value is appropriate or not.

IV. RESULTS

A. Defect Injection Risk (RQ1)

The analysts identified eleven defect categories and selected three defect categories among the eleven categories in Procedure 1-1. The analysts categorized twenty-seven percent of the defects for ODC (see Section III.C) into one of the three defect categories. Table VIII shows the three defect categories. The column "Description" indicates the description of the identified defect patterns. The column "Product ID" indicates the product ID shown in Table III. The values in the columns "Product ID" are the percentages of the number of defects categorized into the corresponding defect pattern to the number of defects categorized into one of the identified eleven defect categories.

Table IX shows the result for Procedure 1-2. Defect injection risk I_k corresponds to defect category D_k ($k = 1, 2, 3$).

TABLE X. IDENTIFIED DEFECT INJECTION RISKS

Risk	Product ID		
	2-1-1	2-1-2	2-1-3
I_1	7.5	13.5	1.0
I_2	NA	7.5	11.5
I_3	3.0	12.5	2.5

TABLE XI. IDENTIFIED DEFECT INJECTION RISKS

Risk	Diagram ID				
	2-2-1	2-2-2	2-2-3	2-2-4	2-2-5
I_1	2	0	2	3	3
I_2	3	0	3	2	2
I_3	4	2	2	1	1

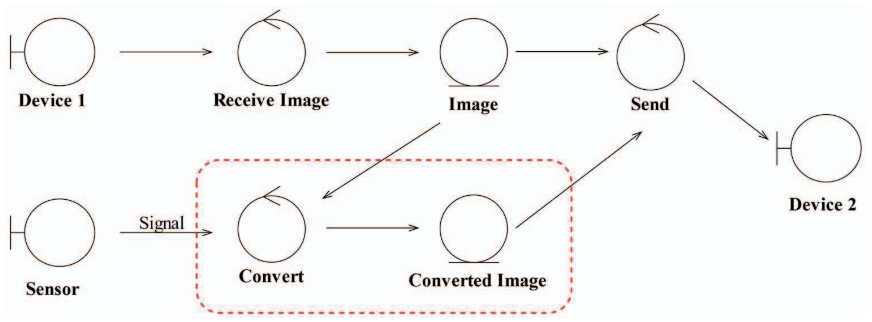

Fig. 4. Robustness diagram applicable to risk I_2

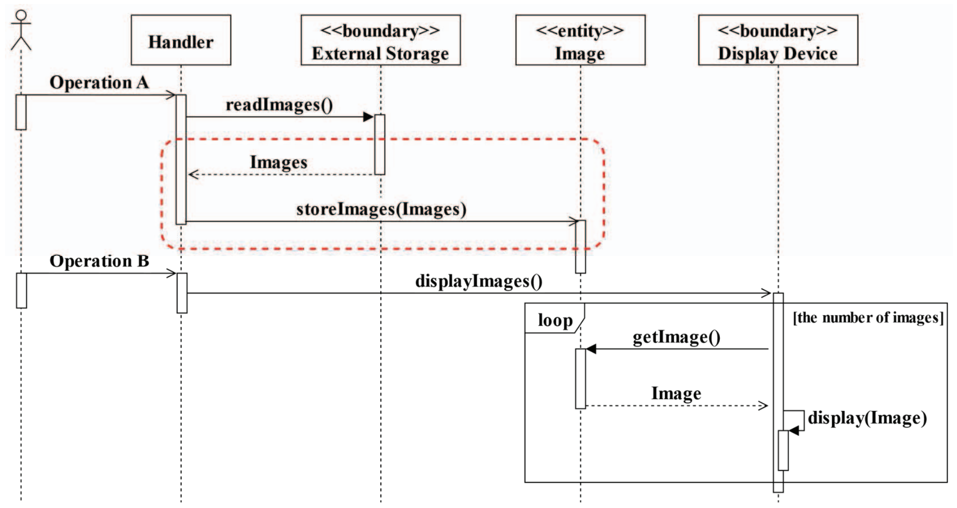

Fig. 5. Sequence diagram applicable to risk I_3

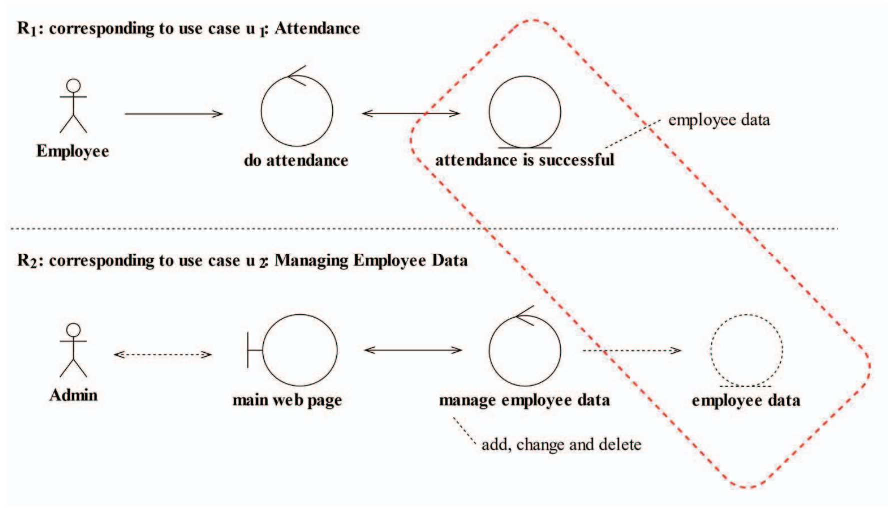

Fig. 6. Robustness diagram applicable to identified risk I_1

Defect injection risk I_1 is found in two ways: a combination of use cases, robustness diagrams, and either a domain diagram or class diagram or a combination of use cases, sequence diagrams, and either a domain diagram or class diagram. Defect injection risks I_2 and I_3 are found in two ways: a combination of robustness diagrams and either a domain diagram or class diagram or a combination of sequence diagrams and either a domain diagram or class diagram.

B. Application of Defect Injection Risk (RQ2)

Table X shows the results for Procedure 2-1, where the values are the ratios of the numbers of identified risks to minimum number of identified risk (the number of identified risk I_1 for product 2-1-3). For product 2-1-1, the analyst did not evaluate defect injection risk I_2 because the data size in domain or class diagram (C or D) may have changed. Thus, the value is NA for I_2 in product 2-2-1.

Table XI shows the result for Procedure 2-2. The values in the table represent the number of risks identified in the analysis and design diagrams.

Fig. 4 and 5 show the parts applicable to defect injection risk I_2 for product 2-1-3, and I_3 for 2-1-1. Fig. 6 show the part applicable to I_1 for diagram 2-2-5. In Fig. 4, the dotted red line indicates defect injection risk I_2. In the conditions defined in Table IX, controller c_1 corresponds to the "Convert" controller in Fig. 4, and entity e_1 corresponds to the "Converted Image" entity. In product 2-2-3, class diagram that corresponds to C for I_2 in Table IX refers to the variance of the size of the "Converted Image" entity.

The actual defect injection risk in the robustness diagram in Fig. 4 is the following. Device2 usually displays an image sent from Device1. Device2 occasionally displays a converted image to protect Device2 from hardware degradation. The "Device1" boundary continuously sends the "Image" entity to the "Device2" boundary. The "Sensor" boundary occasionally sends a trigger to generate the "Converted Image" entity to the "Convert" controller. The "Convert" controller sends the "Converted Image" entity to the "Device2" boundary. Then the "Converted Image" entity is sent to the "Device2" boundary. Consequently, the execution time to generate the "Converted Image" depends on the implementation of the "Convert" controller and the size of the "Image" entity. If the execution time is longer than expected, Device2 may be damaged. If damaged, "Device2" hardware must be repaired. Thus, developers of subsequent development activities must pay attention to the execution time of the "Convert" controller in implementation and validation.

In Fig. 5, the dotted red line indicates defect injection risk I_3. Object o_1 in the defect injection risk I_3 defined in Table IX corresponds to the "Image" object in Fig. 5. Message m_1 corresponds to "storeImages(Images)" message. Sequence diagram S_1 is defined as follows: When a user executes operation A, the system reads the "Images" data in its external storage and stores the data. When a user executes operation B, the system reads the "Image" data and displays the data on the "Display Device." The format type of the "Image" object is defined in attribute a_1 of object o_1 in the class diagram C.

In the sequence diagram in Fig. 5, the size of the "Image" object must be in the value range defined in the class diagram to meet the performance specifications of the "Display Device." If the resizing function for the data stored in the "External Storage" object is omitted, the size of the "Images" object exceeds the value range. If this occurs, the "Display Device" may fail to execute "display(Image)." For example, "Display Device" that does not support displaying 4K images (exceeding the performance specification) cannot finish the "display" feature within the allotted time. Thus, developers of the subsequent development activities must pay attention to the size of the "Image" object within the value range in implementation and validation.

Fig. 6 shows the identified defect injection risk I_1 in robustness diagrams for diagram 2-2-5. It should be noted that the "employee data" entity and the arrow indicated by the dotted black line are newly added by Analyst A because the entity was lacking in the publicly available diagram. The upper side of the robustness diagram corresponds to the "Attendance" use case, while the lower side of the robustness diagram corresponds to the "Managing Employee Data" use case. In the robustness diagram corresponding to the "Attendance" use case, when the "Employee" actor taps their ID card on the card reader, their attendance is recorded in the "employee data" entity. In the robustness diagram corresponding to the "Managing Employee Data" use case, the "Admin" actor adds, changes, or deletes the "employee data." The dotted red line indicates risk I_1.

In the robustness diagram in Fig. 6, the "attendance is successful" entity and "employee data" entity correspond to entity E defined by I_1 in Table IX because the "Attendance is successful" entity is included in "employee data." "Do attendance" controller corresponds to c_1. "Manage employee data" controller corresponds to c_2. Two use cases correspond to the "Attendance" and "Managing Employee Data" use cases. Thus, developers in subsequent development activities must consider the case of the concurrent execution to update the "attendance is successful" entity and changing the "employee data" entity because "employee data" can cause data corruption.

V. Discussion

A. RQ1: Can defect injection risks be identified and generalized from the analysis and design diagrams?

The answer to RQ1 is yes. Three defect injection risks were defined using the analysis and design diagrams and their elements. Defect patterns D_1 Data access confliction and D_3 Insufficient exceptional or alternative implementations for specific parameter values were observed in the defect repositories for six products used in Procedure 1-1. Defect category D_2 Insufficient performance due to resource shortages appeared in the defect repositories for four products. Defect pattern D_2 was not observed in defect repositories for Products 1-1, 1-2, and 1-3 because the defects categorized into defect pattern D_2 were not common among user interface components. Defects categorized into these patterns were not immediately detected at injected activities because their "injected activity" was after "requirement analysis" and before "system testing" and their "detected activity" was system testing. This means that defects in these defect patterns were difficult to find across the products in Procedure 1-1. Thus, the defect injection risks efficiently can help developers prevent and validate defect categories D_1, D_2, and D_3.

In the discussions between the analysts provided the following opinions. For defect pattern D_1, developers should pay attention to an object, whose behaviors are referred to by two or more use cases. If the design and implementations for one use case are assigned to one developer and the design and implementations for another use case are assigned to a different developer, defects categorized into D_1 are rarely found. Thus, defect injection risk I_1 was helpful to detect these defects especially in such situations, which are common among software for large products. For defect pattern D_2, although it is not easy for developers to estimate the execution time depending on the data size of the entities in analysis and design activities, they can identify the entities and carefully consider the execution time with defect injection risk I_2 Insufficient implementation for performance or resource shortage.

The defect injection risks were identified from defects detected in system testing, implying that these defects and risks have larger impacts on the correction effort compared to defects detected in earlier development activities. Furthermore, the effort for considering such risks should be small compared to existing comprehensive defect or risk detection methods because our risks do not always require exhaustive validation as described in occurrence conditions u_k.

Defect injection risks can be identified from other artifacts than analysis and design diagrams. If elements defined in the occurrence conditions u_k can be identified from other notations including free descriptions in natural language, defect injection risks can be identified. As a free description example for Fig. 4, the behavior can be described as follows. (a) Device 1 sends images to Device 2. (b) Device 2 displays a sent image from Device 1. (c) Sensor sends a signal to Device 2 when the display time limit is reached. (d) Device 2 converts and displays another image sent from Device 1 when Device 2 receives the signal. (e) The image size is either 640 * 360 pixels, 1920 * 1080 pixels, or 7680 * 4320 pixels. In this example, the image data described in (a), (b), (d), and (e) corresponds to the entity e_1 defined in defect injection risk I_2. Controller c_1 performs the conversion described in (d). The image sizes described in (e) correspond to the "various sizes of data" defined in defect injection risk I_2.

B. RQ2: Are defect injection risks applicable to other analysis or design diagrams?

The answer to RQ2 is yes. Defect injection risks were applicable to the analysis and design diagrams developed in both the commercial products and publicly available diagrams, even though the development teams of the diagrams of commercial products in RQ1 and those in RQ2 differed. Furthermore, publicly available analysis and design diagrams were those in the different domains including ATM and library systems. Thus, the defect injection risks are expected to be general risks for other software. Specifically, the results showed that defect injection risk I_3 was applicable to the all eight diagrams and that defect injection risk I_1 and I_2 were applicable to six or more diagrams. Although more replications and investigations are necessary to generalize the results, this study suggests that the risks are not specific to the product domain of Sony.

The discussion with the analysts and developers provided the following opinions. First, the risks are general enough to identify risks in diagrams from other domains because the defined conditions u_k for the existence of potential risks are common to various types of software. Second, if the risks are identified in analysis or design activities, developers may be more cautious about the risks because they can easily track concerns by localized potential defect injections indicated by the occurring conditions. Third, although the publicly available diagrams included incomplete and incorrect objects, the analysts were able to identify risks because the occurring conditions u_k were clear and localized. Fourth, the impacts of risks (the effort to correct the defects, if injected) should be smaller than those in real-use software because the publicly available ones in our evaluation were educational materials or samples.

Diagram 2-2-2 was part of a banking system for education material. The publicly available diagrams were limited to an ATM subsystem. Conditions u_1 and u_2 for risks I_1 and I_2 did not apply to any part of the diagrams. Thus, no defect injection risks were identified. If the diagrams included other parts of the banking system such as back office subsystems, the number of the applicable defect injection risks may increase.

C. Threats to validity

1) Internal validity

The analyst may affect the results for RQ1 and RQ2. In this study, one analyst assessed the results, and the other confirmed them. If necessary, the results were changed. Specifically, the results by Analyst A were verified by Analyst B for Procedures 1-1 and 1-2. For Procedure 2-1, the analyst evaluated the defect injection risks. Then at least two developers verified the evaluation. The number of identified risks by the analyst in the results was the number of risks that all the developers agreed upon. For Procedure 2-2 evaluation with publicly available diagrams, two analysts evaluated the defect injection risks separately. More than 90% of the results were consistent between the analysts. For the inconsistent results, the analysts discussed until a consensus was reached.

The identified defect injection risks might be risks for specific products. The overlaps of the products for Procedures 1-1 and 1-2 and the products for Procedures 2-1 were small. The distribution of the applicable defect injection risks was not skewed (Table X). Moreover, conditions u_k was represented with combinations of analysis and design diagrams and their elements. Additionally, the defect injection risks were applied to the publicly available diagrams for a different kind of software.

2) External validity

The identified defect injection risks may not be applicable to diagrams for other software. For risk I_1, condition u_1 is that two or more use cases refer to the same or aggregated objects or entities. Such conditions apply to various situations, including access to data in a database management system. For risk I_2, condition u_2 is that the attribute size of the entities or objects varies. Such attributes are observed in various implementations, including content delivery servers. For risk I_3, condition u_3 is that certain values of an attribute require alternative and exceptional handling. Such attributes are common in many software systems, including server-side systems, which change the response depending on the submitted parameters.

The defect injection risks may depend on the specific development style or practice. However, the defect injection risks only require analysis and design diagrams. The defect injection risks can be used in various stepwise refinement developments with analysis and design diagrams, including upfront plan and iterative approach. Moreover, defect injection risk can easily incorporate with existing practices and processes including test-driven development (TDD) practices and the ICONIX development process [30][38].

VI. Conclusion

This paper defined the defect injection risks from requirement analysis and design diagrams to prevent defects in implementation and validation activities. Defect injection risks indicate specific criterion or parts where developers must carefully consider in the implementation and validation activities. A defect injection risk consists of a risk description, diagram type, and occurrence conditions of objects in the diagram. First, we identified defect patterns, which were injected after the requirement analysis and design activities and detected during system testing for the development of commercial products at Sony. Second, regarding the defect patterns as exposed defect injection risks, we defined three defect injection risks with the analysis and design diagrams of the products. Finally, we evaluated whether these three defect injection risks were observed in three other sets of analysis and design diagrams developed at Sony. The results of the evaluation showed that two risks were found in the analysis and design diagrams for the three products. The remaining one risk was found in the analysis and design diagrams for two products and could not be evaluated in the analysis and design diagrams for one product due to requirement volatility. Moreover, we evaluated whether the risks were applicable to five publicly available analysis and design diagrams for different software in other domains. The result showed that one risk was found in all sets of the analysis and design diagrams, and the other two risks were identified in four sets of the diagrams.

Future works include defining the procedures to identify defect injection risks. Such procedures will aid novice and intermediate software engineers in identifying defect injection risks. Investigating the defect prevention effectiveness by the defect injection risks is also an important future work. In discussions with the developers of the products in this evaluation, expert engineers have knowledge and experience on other defect injection risks. Defining such defect injection risks will realize an ontology of defect injection risks. Future works include constructing a defect injection risk ontology.

Acknowledgment

The authors thank the product developers for their cooperation and Dr. Taro Kurita for his insightful reviews.

References

[1] Raghuraman, A., Ho-Quang, T., Chaudron, M. R., Serebrenik, A., Vasilescu, B. (2019, May). Does UML modeling associate with lower defect proneness?: a preliminary empirical investigation. In 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pp. 101-104. IEEE.

[2] Conradi, R., Mohagheghi, P., Arif, T., Hegde, L. C., Bunde, G. A., Pedersen, A. (2003, July). Object-oriented reading techniques for inspection of UML models–an industrial experiment. In European Conference on Object-Oriented Programming, pp. 483-500. Springer, Berlin, Heidelberg.

[3] Egyed, A. (2010). Automatically detecting and tracking inconsistencies in software design models. IEEE Transactions on Software Engineering, 37(2), pp. 188-204.

[4] Lange, C. F., Chaudron, M. R. (2006, May). Effects of defects in UML models: an experimental investigation. In Proceedings of the 28th international conference on Software engineering, pp. 401-411.

[5] Zisman, A., Kozlenkov, A. (2001, November). Knowledge base approach to consistency management of UML specifications. In Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001), pp. 359-363. IEEE.

[6] David, P., Idasiak, V., Kratz, F. (2008). Towards a better interaction between design and dependability analysis: FMEA derived from UML/SysML models. Safety, reliability and risk Analysis: theory, methods and applications, pp. 2259-2266.

[7] Mehner, K., Monga, M., Taentzer, G. (2006, September). Interaction analysis in aspect-oriented models. In 14th IEEE International Requirements Engineering Conference (RE'06), pp. 69-78. IEEE.

[8] Guiochet, J. (2016). Hazard analysis of human–robot interactions with HAZOP–UML. Safety science, 84, pp. 225-237.

[9] Houmb, S. H., Den Braber, F., Lund, M. S., Stølen, K. (2002, September). Towards a UML profile for model-based risk assessment. In Critical systems development with UML-Proceedings of the UML'02 workshop, pp. 79-91.

[10] Oveisi, S., Farsi, M. A. (2018). Software safety analysis with UML-Based SRBD and fuzzy VIKOR-Based FMEA. International Journal of Reliability, Risk and Safety: Theory and Application, 1(1), pp. 35-44.

[11] Oveisi, S., Farsi, M. A., Kamandi, A. (2020). Design Safe Software via UML-based SFTA in Cyber Physical Systems. Journal of Applied Intelligent Systems and Information Sciences, 1(1), pp. 11-23.

[12] Menezes Jr, J., Gusmão, C., Moura, H. (2013). Defining indicators for risk assessment in software development projects. Clei electronic journal, 16(1), pp. 11-11.

[13] Abioye, T. E., Arogundade, O. T., Misra, S., Akinwale, A. T., Adeniran, O. J. (2020). Toward ontology-based risk management framework for software projects: an empirical study. Journal of Software: Evolution and Process, 32(12), e2269.

[14] Tsoumas, B., Gritzalis, D. (2006, April). Towards an ontology-based security management. In 20th International Conference on Advanced Information Networking and Applications-Volume 1 (AINA'06), 1, pp. 985-992. IEEE.

[15] Kozlenkov, A., Zisman, A. (2002, September). Are their design specifications consistent with our requirements?. In Proceedings IEEE Joint International Conference on Requirements Engineering, pp. 145-154. IEEE.

[16] Rao, A. A., RajiniKanth, T. V., Ramesh, G. (2016). A Model Driven Framework for Automatic Detection and Tracking Inconsistencies. J. Softw., 11(6), pp. 538-553.

[17] Kamalrudin, M., Grundy, J., Hosking, J. (2010, September). Tool support for essential use cases to better capture software requirements. In Proceedings of the IEEE/ACM international conference on Automated software engineering, pp. 255-264.

[18] Hausmann, J. H., Heckel, R., Taentzer, G. (2002). Detection of conflicting functional requirements in a use case-driven approach. In Proceedings of the 24th International Conference on Software Engineering. ICSE 2002, pp. 105-115. IEEE.

[19] Jurkiewicz, J., Nawrocki, J., Ochodek, M., Głowacki, T. (2015). HAZOP-based identification of events in use cases. Empirical Software Engineering, 20(1), pp. 82-109.

[20] Redmill, F., Chudleigh, M., Catmur, J. (1999). System safety: HAZOP and software HAZOP. Wiley.

[21] Srivatanakul, T., Clark, J. A., Polack, F. (2004, September). Effective security requirements analysis: Hazop and use cases. In International Conference on Information Security, pp. 416-427. Springer, Berlin, Heidelberg.

[22] Bazyan, A., Krashak, N. (2019). UML Ninja: Automatic Assessment of Documentation and UML Practices in Open Source Projects, University of Gothenburg/Department of Computer Science and Engineering, pp. 1-13. http://hdl.handle.net/2077/62455

[23] Mens, T., Van Der Straeten, R., D'Hondt, M. (2006, April). Dependency Analysis of Model Inconsistency Resolutions. In ERCIM Workshop on Software Evolution, pp. 127-136.

[24] Jurack, S., Lambers, L., Mehner, K., Taentzer, G., Wierse, G. (2009, March). Object flow definition for refined activity diagrams. In International Conference on Fundamental Approaches to Software Engineering, pp. 49-63. Springer, Berlin, Heidelberg.

[25] Fowler, M. (1999). Refactoring: improving the design of existing code. Addison-Wesley Professional.

[26] Kim, M., Zimmermann, T., Nagappan, N. (2012, November). A field study of refactoring challenges and benefits. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, pp. 1-11.

[27] Moha, N., Guéhéneuc, Y. G., Duchien, L., Le Meur, A. F. (2009). Decor: A method for the specification and detection of code and design smells. IEEE Transactions on Software Engineering, 36(1), pp. 20-36.

[28] Vidal, S. A., Marcos, C., Díaz-Pace, J. A. (2016). An approach to prioritize code smells for refactoring. Automated Software Engineering, 23(3), pp. 501-532.

[29] Chug, A., Tarwani, S. (2017, September). Determination of optimum refactoring sequence using A* algorithm after prioritization of classes. In 2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI), pp. 1624-1630. IEEE.

[30] Rosenberg, D., Scott, K. (1999). Use case driven object modeling with UML. Reading: Addison-Wesley Professional.

[31] R. Bjork. 2005. An ATM Simulation. http://www.math-cs.gordon.edu/local/courses/cs211/ATMExample

[32] R. Bjork. 2005. A Simple Address Book. http://www.math-cs.gordon.edu/local/courses/cs211/AddressBookExample

[33] Briand, L., Labiche, Y. (2002). A UML-based approach to system testing. Software and systems modeling, 1(1), pp. 10-42.

[34] Farizd, M., Pradana, B. P., Shahita, D., Wati, S. F. A. (2022). Analysis and Design of Employee Attendance Application System Using RFID E-KTP Technology with ICONIX Process Method. Inform: Jurnal

Ilmiah Bidang Teknologi Informasi dan Komunikasi, 7(2), pp. 132-142.

[35] Chillarege, R., Bhandari, I. S., Chaar, J. K., Halliday, M. J., Moebus, D. S., Ray, B. K., , Wong, M. Y. (1992). Orthogonal defect classification-a concept for in-process measurements. IEEE Transactions on software Engineering, 18(11), pp. 943-956.

[36] Butcher, M., Munro, H., Kratschmer, T. (2002). Improving software testing via ODC: Three case studies. IBM Systems Journal, 41(1), 31-44.

[37] Mäntylä, M. V., Lassenius, C. (2008). What types of defects are really discovered in code reviews?. IEEE Transactions on Software Engineering, 35(3), 430-448.

[38] Collins-Cope, M., Stephens, M., Rosenberg, D. (2005). Agile development with the ICONIX process: people, process and pragmatism. Springer.