Detecting context-dependent defects and measuring review quality for software reviews

Michiyo Wakimoto

Abstract

This thesis focuses on two issues preventing software reviews from providing the expected effect. The first issue is low review quality, which results in that software review materials include overlooked defects. The second issue is context-dependent defects, which could not be considered as defects and turn out to be defects in the subsequent software development activities.

For the first issue, this thesis proposes a metric, the number of questions and discussions, which identifies concerns in software reviews. First, I defined an effective question, which identifies concerns. Then, I defined detailed software review processes (identifying, sharing, and recording processes), which capture how concerns identified by effective questions were shared and defects were documented. I conducted a case study with 25 projects in industry to investigate the impact of the number of effective questions, which identified concerns, on the number of detected defects in subsequent testing. The results of a multiple regression analysis showed that the number of effective questions predicted the number of defects in subsequent testing at the significance level of 0.05.

For the second issue (context-dependent issue), this thesis conducted a case study to investigate which type of defects could be regarded as context-dependent defects. Specifically, I analyzed defects that required significant correction effort in a simulation control software system development. The results of the case study showed that the defects were ambiguity defects (context-dependent defects) injected by misunderstandings and inconsistencies among stakeholders during interpreting requirements and specifying design documents. The ambiguities of the specifications are

found in the definitions of distance, time (time zone), and calculation accuracy. These cause inconsistencies among the implementations and errors in the control simulation execution results. Based on the analysis, I propose a low-effort defect prevention approach clearly defining the units to avoid such ambiguities. I evaluated the approach and estimated the expected effort reduction in the target control simulation software system development.

Additionally, this thesis proposes a software review method to detect context-dependent defects by generalizing the ambiguity defects identified in the simulation control software system case study. The proposed method can help reviewers detect omissions or ambiguities in requirements caused by design context. Some software requirements are omitted or ambiguous depending on the design context, although these requirements would not necessarily be regarded as omitted or ambiguous when viewed as requirements alone. The design context sometimes causes inconsistencies among implementations that realize the same requirement. The proposed method defines goal-oriented check items for design review using a goal tree obtained by goal-oriented requirements analysis. Reviewers use the goal-oriented check items to detect inconsistent implementations that realize the same requirement. This thesis also evaluates the proposed method through a case study. The results of the case study showed that the proposed method defined five goal-oriented check items and that reviewers detected 24 context-dependent defects with goal-oriented check items. The results also showed that the sum of the estimated additional effort to define goal-oriented check items and perform design reviews with goal-oriented check items was 19.6 person-hours. Furthermore, the results showed that an engineer with general skills and knowledge of software development but without system-specific skills and knowledge could define a goal tree and the corresponding goal-oriented check items.

Contents

List of Figures

List of Tables

1. Introduction

As software products have become an essential part of everyday life, defects in released software are becoming a more serious issue. To prevent such defects in released software, software development teams should detect and correct as many defects as possible within the software development process. Software development processes comprise several operations to deliver a product, which includes: requirements analysis (eliciting requirements), software architectural design (developing top-level structure, organization of the software, and identifying components), coding (writing source code), testing (evaluating the program and fixing detected defects), and maintenance (modifying existing software while preserving its integrity) [1], [2]. Software defects should be detected and corrected within the injected software development process or the process after the injection as earlier as possible because detecting and correcting software defects earlier can reduce correction efforts [3], [4]. A previous survey of 169 large software projects reported that changes in the maintenance process were roughly 100 times more costly than in the specification process [3], [5].

Software review is a static analysis technique aimed at early defect detection [6-11]. It is also one of the most effective evaluation quality assurance techniques [12-15]. Reviewers manually check materials (documents and source code) in this development activity to ensure no defects remain [16]. Specifically, reviewers point out and discuss potential defects, the authors and reviewers verify whether the identified defects are true defects, and decide which ones require action, including correction.

Early defect detection in software review reduces defect correction effort compared to defect detection techniques available after the coding process (e.g., source code

analysis, runtime checking, and testing). Software review can be performed on intermediate artifacts before the coding process. Fagan and Davis reported that while defects detected in software reviews formed 82% of all defects detected in entire development processes, software reviews consumed approximately 15% of development resources, thereby reducing defect correction efforts by more than 25% [6], [17].

Guided reviews provide one approach to enhance effectiveness in software reviews. Guided reviews help reviewers comprehensively detect severe defects, including omissions or ambiguities, by providing detailed instructions, procedures, and hints. Many studies have reported on the effectiveness of guided reviews [9], [11], [18-25]. Typical guided review techniques are checklist-based reading (CBR) [6], perspective-based reading (PBR) [21], defect-based reading (DBR) [20], usage-based reading (UBR) [22], and traceability-based reading [26]. CBR is a reading technique in which reviewers use a list of questions to help understand what defects to examine [23]. PBR [18], [27], [28] is a scenario-based reading (SBR) [20] that defines the perspectives of the stakeholders and assigns the perspectives to reviewers. DBR is an SBR that focuses on detecting specific types of defects [20], [23]. UBR prioritizes the use cases and detects the most critical defects in target materials along with prioritized use cases [23], [26].

Software review cannot always provide the expected effect. Although several software review metrics, including the number of detected defects and the effort to perform the review, have been proposed to judge whether reviews provide the expected effect, these metrics are insufficient to capture review quality. For example, although testing can evaluate target program quality by the number of failed and passed test cases from the test results, such clear metrics from software review results have not been proposed. Furthermore, existing models and techniques, which use existing review metrics, including the fault-prone module prediction [29] and the capture-recapture model [30-32], have been insufficient to capture review quality.

Some definitions or descriptions can be ambiguous depending on the decisions in subsequent development processes, although the definitions or descriptions would not necessarily be ambiguous without the decisions. Guided reviews do not refer to such ambiguities nor assume that reviewers detect such ambiguity defects with the guides. One example of ambiguity defects is the incident involving the Mars rover. The Mars Climate Orbiter lost communication just before arriving on Mars in 1999 [33]. According to the report, the lost communication was caused by a misunderstanding between the pound-force second and newton-second units among developers, which should have been detected during software requirements analysis reviews and software architectural design reviews. Nevertheless, it was not detected during the reviews and subsequent testing because the developers interpreted the same terminology differently.

This thesis focuses on two issues preventing software reviews from providing the expected effect. First, software review materials include overlooked defects when software reviews cannot detect defects sufficiently (low review quality). One reason for insufficient software reviews is that the discussions, questions, and answers during software reviews missed the point. Another reason is that the review time to detect appropriate defects is insufficient. Capturing such insufficient software reviews is difficult using existing software review metrics, including the number of detected defects and effort to perform the review. Second, context-dependent defects which are not considered as defects during reviews but turn out to be defects in subsequent software development activities. Although these requirements are not necessarily ambiguous when viewed as requirements alone, some are ambiguous depending on the design context. Hence, ambiguities in requirements may be identified as two or more different implementations realize the same requirement. Consequently, each implementation can be adequately implemented for the requirement, but the implementations are not always consistent with one another.

For the first issue (low review quality), this thesis proposes a new metric to assess whether software reviews are performed properly. Reviewers can evaluate software

review quality more precisely, using the proposed metric and the existing software review metrics. The proposed metric can also help a project manager (review leader) identify an insufficient software review, which has two benefits: first, it reveals that the project manager should plan additional software reviews with expert reviewers. Second, it highlights the need for more resources to conduct subsequent testing.

For the second issue (context-dependent defects), this thesis first conducts a case study to analyze context-dependent defects that require significant correction effort. The results of the case study found focusing on essential data definitions could have prevented a kind of context-dependent defects in the case study. Subsequently, by generalizing the focus on the essential data to the goals of the software, a software review method is proposed to detect context-dependent defects, referring to the goals of the software identified in the requirements. The proposed method selectively detects ambiguity defects requiring significant correction efforts among many ambiguity defects.

This thesis is structured as follows: Section 2 describes the new metric to assess whether software reviews are appropriately performed. Section 3 conducts the case study to analyze omissions or ambiguities in requirements depending on the design context. Section 3 also proposes an approach to prevent omission or ambiguity defects. The proposed method generalized from the approach for the case study is described in Section 4. Section 5 summarizes the thesis.

2. A Metric for Questions and Discussions Identifying Concerns in Software Reviews

2.1 Introduction

In software reviews, reviewers not only detect defects but also ensure that the software review material is free of concerns about potential defects by asking questions and engaging in discussions, because the concerns may cause defects [34], [35]. A study which analyzed utterances during software reviews [36] reported that 60–70% of the conversations consisted of “informing” and “clarification.” Other studies [37], [38] reported that reviewers spend 38% of the review time verifying, justifying, or rejecting potential defects and concerns. A study on code review effectiveness [39] reported that code review comments included questions, and these questions helped reviewers detect defects. In the case where a concern identified by a question is applicable to the software review material during the subsequent discussion, the concern and applicable locations are specified as a defect. On the other hand, in the case where a concern is not applicable to the software review material, it is discarded or recorded as a false-positive defect. For example, in a code review, a concern may be identified by the question, “Is it intentional that one of the parameters passed to the function is not used?” Then, the subsequent answers and discussions enable the reviewers and authors to find that the source code statements using the parameter passed to the function are omitted. In this case, the concern “the implementation using the parameter passed to the function may be omitted”

identified by the question and the discussion reveals a defect: “the implementation using the parameter passed to the function is omitted.” On the other hand, if the parameter passed to the function is designed for compatibility with older versions and is not used intentionally, the concern is not applicable and will not identify a defect. Although this kind of question and discussion may lead to defect detections, its effectiveness and the detailed process have yet to be investigated.

Concerns identified by questions and discussions cannot be directly extracted from defects in a defect list after software reviews because the defects include defects directly detected by reviewers and defects found by examining concerns. Furthermore, some concerns are discarded or recorded as false-positive defects if they are not applicable to the software review materials. Although some studies have used objective indicators such as the number of detected defects to assess whether software reviews are performed properly [31], [32], [40], such metrics only include the number of defects directly detected by reviewers and defects found by concerns, which are applicable to the software review material.

The number of questions identifying concerns can be an indicator of effective software reviews. Some studies have demonstrated that the number of questions identifying concerns is an indicator of effective software reviews. One study evaluated the percentage of interrogative sentences in each review comment as a metric for code review quality [41]. Another study defined a new metric, Issue Density, to estimate the code review quality [39]. However, neither study evaluated the relationship between the quality of review and the quality in subsequent testing.

This section proposes a metric, the number of effective questions which identify concerns in software reviews. First, I defined effective questions and the processes by which effective questions are recognized and recorded as defects as well as the categories for true and false-positive defects. Then, I surveyed previous studies according to the defined process and defect categories to investigate whether defects are

distinguished defects directly detected from those found by concerns according to the defined processes and categories. Furthermore, I implemented a case study, which involved 25 projects in industry, to investigate the effectiveness of effective questions in software reviews. The metrics in the case study include the number of effective questions that identify concerns, number of defects detected, and number of defects detected in subsequent testing. I performed multiple regression analysis for these metrics to evaluate the effectiveness of questions and discussions in software reviews. The research question is formulated as the following.

RQ: Does the number of effective questions in a software review affect the quality of subsequent testing?

2.2 Effective Questions in Software Reviews

2.2.1 Definition

I define effective questions to distinguish between questions, which identify concerns about potential defects from those that clarify and understand the software review material, because questions and subsequent discussions in software reviews cover diverse topics such as exchanging opinions on defects, evaluating the value, clarifying solutions, and rejecting hypotheses [37], [38]. Software reviews can be categorized as synchronous, such as a face-to-face meeting, or asynchronous, such as sending and receiving defect descriptions via a review support tool [42-44]. In synchronous reviews, reviewers share potential defects and ask effective questions in a review meeting. In asynchronous reviews, reviewers share potential defects and ask effective questions using review support tools. Figure 2.1 shows the relationships among concern, question, and defect. Table 2.1 shows the definitions of the terms.

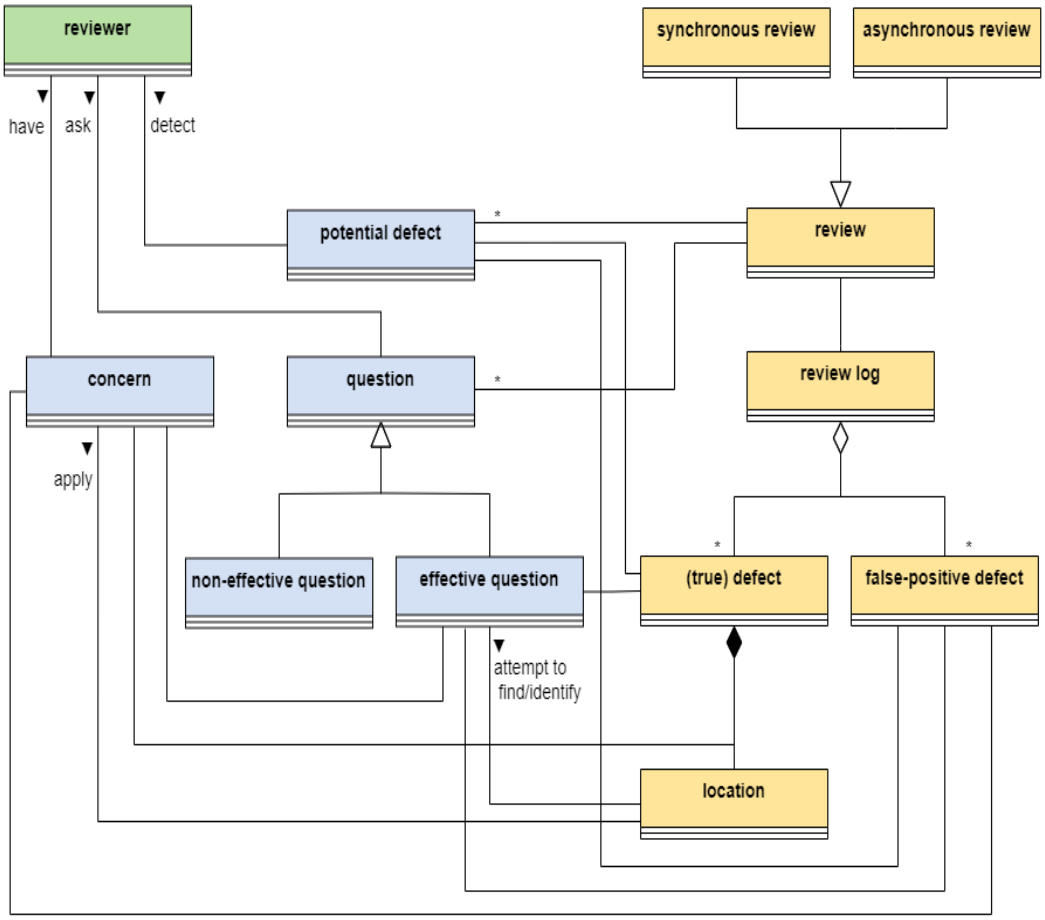

Figure 2.1 Class diagram of concern, question, and defect.

Table 2.1 Definitions of terms.

Category and term		Description
Reviewer		A reviewer is a person who checks materials (documents and source code) manually in a software review, asks questions, or points out potential defects.
Concern		A concern is a potential cause of a defect. If the concern applies to the review material, the defect will be identified with its location. In software reviews, reviewers ensure that the software review material is free of concerns about potential defects by asking questions and engaging in discussions, because the concerns may cause defects.
Question	Effective question	An effective question is a question clarifying whether the concern applies to the review materials. If a reviewer asks an effective question, the other reviewers verify the concern and try to determine the locations applicable to the concern.
	Non-effective question	A non-effective question is a question without any concern.
Defect	Potential defect	A potential defect is a defect identified and pointed out by a reviewer without effective questions. The other reviewers verify the potential defect to judge whether it is a true defect or a false-positive defect.
	True defect	A true defect is a defect that was judged to require action, including correction. A true defect can be found by an effective question or a potential defect.
	False-positive defect	A false-positive defect is an incorrect defect (mistakenly regarded as a defect) or is a concern identified by an effective question and not applying to any part of the review materials.

A set of questions (U) consists of a set of effective questions (D_{iq}) and a set of non-effective questions (N). I assume that a defect can be specified with a concern and its locations in the software review material. The discussions following the effective questions (D_{iq}) verify the concern and determine the locations applicable to the concern. Thus, if a concern identified by an effective question is judged to apply to the software review material through discussions, the locations of concern can be determined. On the other hand, if a concern is not judged to apply to the software review material through discussions, no location is determined. Thus, the concern does not lead to specifying

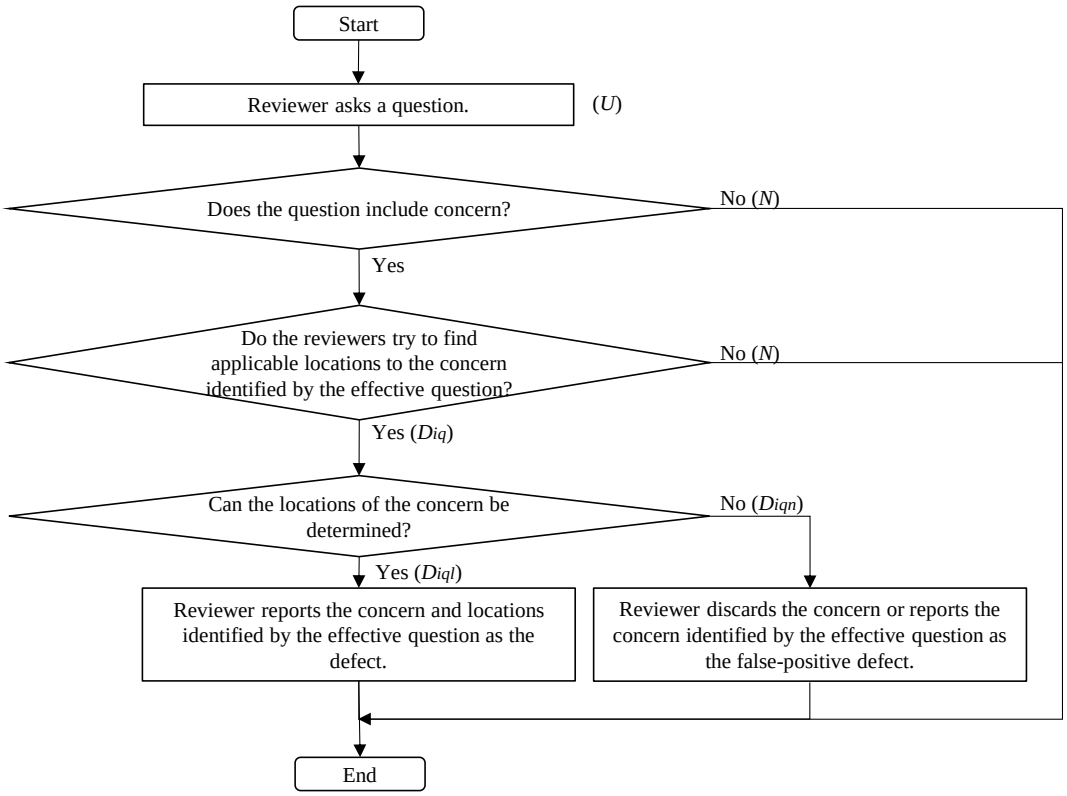

Figure 2.2 Flowchart to categorize effective questions and distinguish between true and false-positive defects identified and specified by effective questions.

(finding) a defect. Namely, a set of effective questions (D_{iq}) consists of a set of effective questions identifying concerns with locations that apply to the concerns (D_{iql}) and a set of effective questions without applicable locations that apply to the concerns (D_{iqn}). Figure 2.2 shows the flowchart for categorizing effective questions and determining true or false-positive defects identified and specified by effective questions. The first and second branches categorize effective questions. As indicated by the second branch in Figure 2.2, if reviewers do not attempt to find the applicable locations for the concern identified by the question, then it is considered to be a non-effective question. An example of a non-effective question without a concern is “What time does this review

meeting end?" An example of a non-effective question for the reviewer's self-understanding is "Which chapter defines the glossary?"

Reviewers ask an effective question when they cannot specify locations for concerns or when they are unable to expend effort to check and find locations of concerns. Figure 2.3 shows an example of an effective question. An example is the question ($\in U$), "Does the interrupt program change the value of the global variable x? If yes, the assigned value and reference value are not consistent." This identifies a concern that the global variable x can be overwritten by the interrupt program. If the interrupt program, which changes the global variable x, can be executed during the assignment and reference, the concern applies to the software review material (source code A in Figure 2.3). The locations are where the interrupt program changes the value of the global variable x or the omitted place (description), disabling the interrupt. Then, the defect ($\in D_{st}$) "The value of the global variable x may not be consistent because the interrupt program can change the value, and disabling the interrupt programs is omitted." is detected by the effective question, identifying the concern applicable to locations ($\in D_{iql}$). In the case where a concern identified by an effective question applies to the software review material, the defect is recorded as a true defect ($\in D_{rt}$). If a concern does not apply to the software review material (source code B in Figure 2.3), the concern is discarded or recorded as a false-positive defect ($\in D_{rf}$) detected by the effective question identifying the concern without applicable locations ($\in D_{iqn}$), depending on the recording policy.

The number of effective questions identifying concerns can be an indicator of effective software reviews. Furthermore, reviewers are expected to directly detect defects, and they then have confidence to ask effective questions. The proportion of the number of effective questions to the number of directly detected defects should be measured because available time and effort for the software reviews are limited. If the proportion of the number of detected defects to the number of effective questions is larger, time and effort for asking questions are likely to be limited.

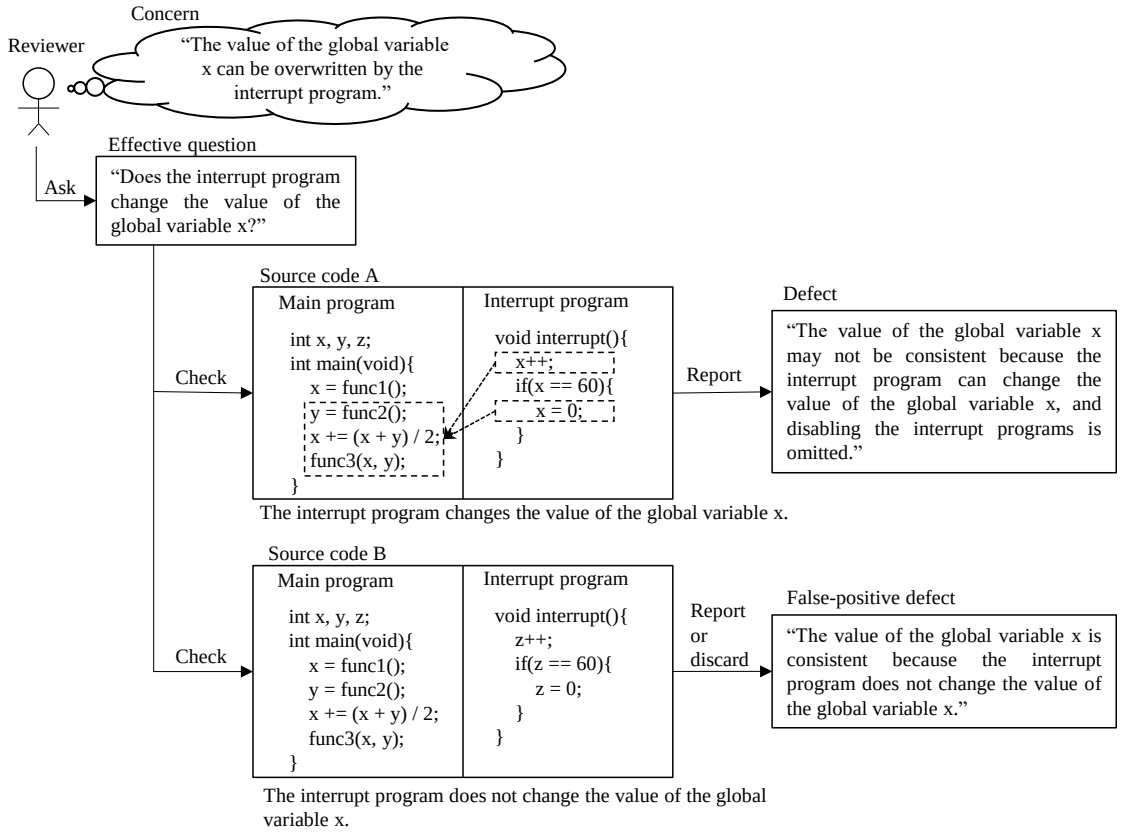

Figure 2.3 An example of an effective question, which shows true or false-positive defects from an effective question.

2.2.2 Defect Category and Effective Questions in Software Review Process

Although reviewers present potential defects and ask effective questions in both synchronous and asynchronous reviews, their processes differ.

In synchronous reviews, reviewers present potential defects verbally and ask effective questions during a software review. According to Fagan, a software review consists of overview, preparation, review, rework, and follow-up processes [6]. In the software review process, a reviewer presents potential defects and asks effective

questions. Then, the authors respond. If necessary, the potential defects and concerns identified by the effective questions are further discussed [6], [45].

In asynchronous reviews (non-meeting-based approaches [44]), a reviewer denotes potential defects and effective questions, which are sent to the authors and other reviewers via a review support tool. After the authors answer the effective questions, the authors and reviewers discuss the concerns identified by the effective questions and answers using the tool. In asynchronous patch reviews, a fix proposal (a code patch) may be sent with a potential defect [46], [47].

Both synchronous and asynchronous reviews include the following identifying, sharing, and recording processes.

- Identifying

  A reviewer checks the material and identifies potential defects. It is assumed that the reviewer thinks that the potential defects are true defects, as the reviewer does not want to share false-positive defects in software reviews. If the reviewer has a concern, they prepare effective questions, which will be asked in the sharing process. In asynchronous reviews, the reviewer inputs the potential defects and effective questions into the review support tool.

- Sharing

  Each potential defect identified by the reviewers is shared, and whether it is a true or false-positive defect is evaluated. The authors and other reviewers answer the effective questions and discuss identified concerns to find applicable locations and ensure that no defect remains. Each potential defect or concern identified by effective questions is subsequently categorized as either a true defect or a false-positive defect.

- Recording

In synchronous reviews, the true defects judged in the sharing process are recorded. In some reviews, defects judged to be false positives in the sharing process are recorded, whereas in other reviews, they are discarded. In asynchronous reviews, potential defects and effective questions are already recorded in the identification process. Hence, potential defects and effective questions are categorized into true or false-positive defects. In addition, the defect descriptions may be updated, depending on the discussions in the sharing process.

Figure 2.4 overviews the process to categorize potential defects and effective questions in the sharing process and how true and false-positive defects are recorded in the recording process. For synchronous reviews, reviewers identify potential defects (D_{id}) and effective questions (D_{iq}) in the identification process. Potential defects and effective questions are treated as the same type because both are identified by reviewers when checking the software review material. For synchronous reviews, reviewers explain potential defects and effective questions verbally. For asynchronous reviews, reviewers submit potential defects and effective questions in text. In the sharing process, the reviewers present D_{id} and ask D_{iq}, and then the authors and the other reviewers examine defects (D_{id}) and concerns identified the by D_{iq}. Finally, based on their discussion, the defects and concerns are categorized into true defects (D_{st}) and false-positive defects (D_{sf}). In the sharing process, new potential defects and effective questions may be found. In this case, they are added to D_{id} and D_{iq}. In the recording process, each defect in D_{st} is recorded as true defects (D_{rt}). Depending on the recording policy, some defects in false-positive defects (D_{sf}) are recorded as false-positive defects (D_{rf}). After the recording process, defects in true defects (D_{rt}) are corrected.

For asynchronous reviews, reviewers identify potential defects (D_{id}) and effective questions (D_{iq}). Then, they input D_{id} and D_{iq} into a review support tool. In the sharing process, the reviewers send D_{id} and D_{iq} to the authors and other reviewers. After the authors and other reviewers understand D_{id} and D_{iq}, they answer the effective questions

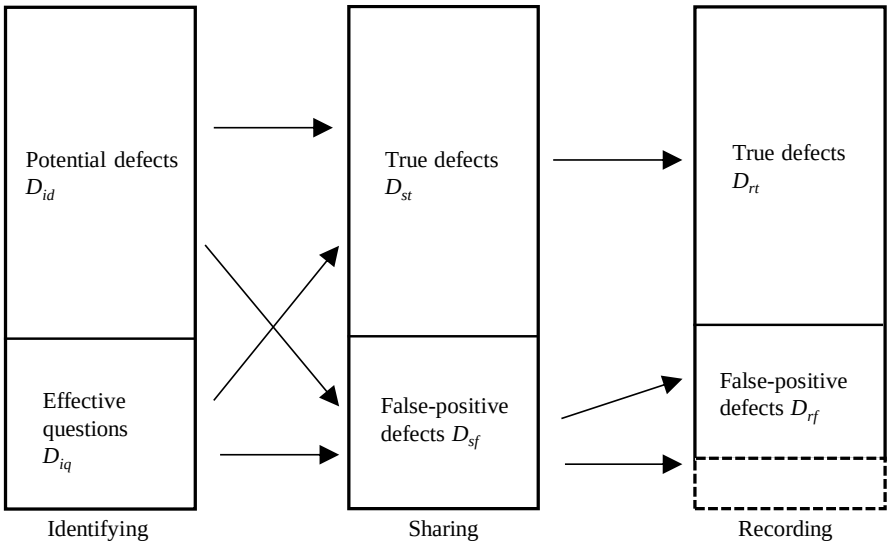

Figure 2.4 Defect categories in identifying, sharing, and recording process.

(D_{iq}) and discuss the concerns identified by D_{iq}. In addition, they examine and judge whether the potential defects (D_{id}) are true defects.

Finally, the authors and the reviewers categorize D_{id} and D_{iq} into true defects (D_{st}) and false-positive defects (D_{sf}) based on the discussions. Effective questions (D_{iq}) are categorized into effective questions identifying concerns with applicable locations (D_{iql}) and effective questions identifying concerns without applicable locations (D_{iqn}) depending on whether the concerns are applicable or not. In the recording process, true defects (D_{st}) are labeled or categorized as true defects (D_{rt}). False-positive defects (D_{sf}) are labeled or categorized as false-positive defects D_{rf}. Some of the defects in the false-positive defects (D_{sf}) may be discarded in the recording process. After the recording process, true defects (D_{rt}) are corrected. In the case where a code patch is attached to the true defects (D_{rt}), the patches are merged.

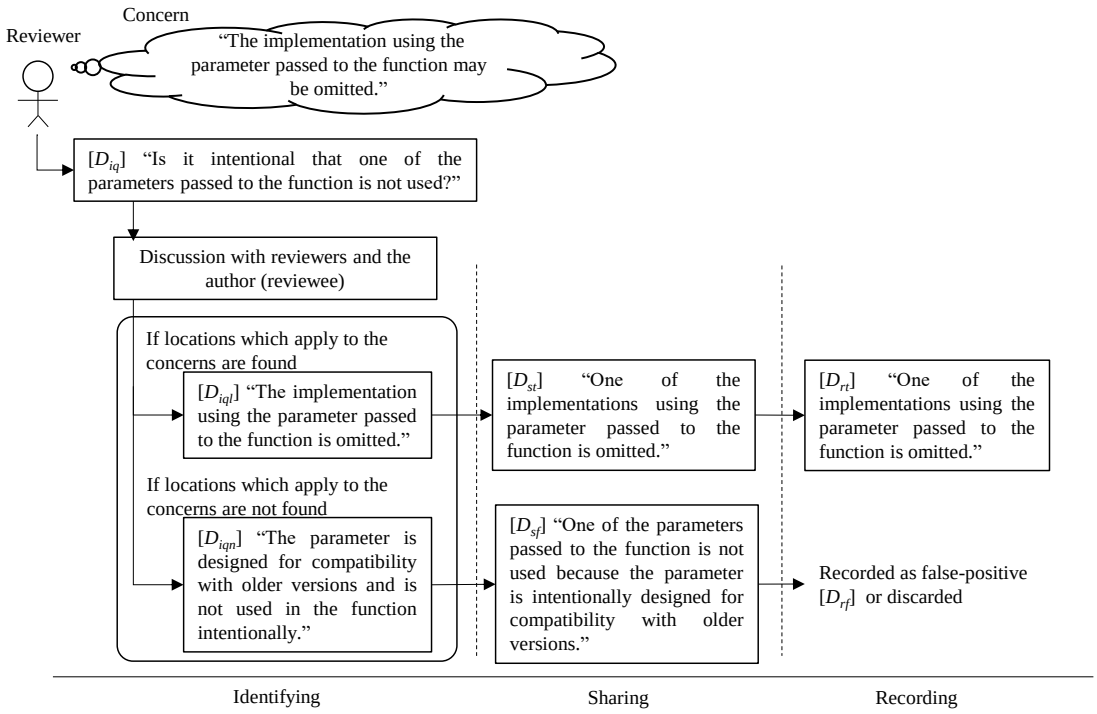

Figure 2.5 An example for the process and categories which shows the categorization of effective questions according to the flowchart in Figure 2.2 and the process in Figure 2.4.

Figure 2.5 shows an example for the process and categories. In Figure 2.5, if the locations of concern are found through discussion, the author (reviewee) who knows the design intention answers, "The implementation using the parameter passed to the function is omitted." On the other hand, if the locations of concern are not found through discussion, the author (reviewee) answers, "The parameter is designed for compatibility with older versions and is not used in the function intentionally."

2.2.3 Literature Review

I conducted a literature review to identify articles that describe concerns raised by effective questions, subsequent discussions, and categorization of true and false-positive defects according to the applicable concerns. Many studies categorized true defects [48-50], but few studies categorized false-positive defects. Previous studies referring to software reviews in which false-positive defects were detected [16], [45], [50], [51] did not refer to the categories or details of false-positive defects. Articles [16], [52] excluded false-positive defects prior to defect analysis. Moreover, one study described that analysis and retrospect of software reviews used information of true defects as inputs [53]. Only the article [54] referred to the use of a question list. However, it did not describe the concerns identified by the questions.

I investigated which processes and categories defined in Subsections 2.2.1 and 2.2.2 are referred to when true defects and false-positive defects are judged and recorded. Table 2.2 shows the result. The arrow (->) means the right side arises from the left side. True defects and false-positive defects were judged and recorded in different processes. Three articles [54-56] referred to defects in D_{id} -> D_{st}. These articles described that the participants of the software reviews discussed whether the presented defects were true defects or not, and they categorized defects as true defects. One article [57] referred to defects categorized as D_{id} -> D_{sf}. It described that the participants in the software reviews discussed whether the presented defects were false-positive or not prior to deciding they were false-positive defects. One article [58] referred to defects categorized as D_{id} -> D_{st}, D_{sf}. The participants of the software reviews discussed whether the presented defects were false-positive defects or not, and they found both true and false-positive defects. One article [55] referred to defects categorized as D_{st} and D_{sf}. Another article [54] referred to defects categorized as D_{id} -> D_{rf}. One article [50] referred to defects categorized as D_{rf}.

Table 2.2 Defect processes in previous studies.

Categories	Text referring the defect categories in the previous studies
D_{id} -> D_{st}	To support decision making, discussants can also vote by rating any potential defect as true defect [54] Collated defects: the number of defects merged from individual findings to be discussed during the meeting. True defects: the number of defects for which consensus was reached during the meeting in considering them as true defects [55]. We used the information from the repair form and interviews with the author to classify each issue as a true defect (if the author was required to make an execution affecting change to resolve it) [56].
D_{id} -> D_{sf}	False positives are items reported by subjects as defects, when in fact no defect exists [57].
D_{id} -> D_{st}, D_{sf}	In addition to the instructions from the preparation phase, the instructions in the meeting phase were: use the individual inspection record and decide which are faults and which are false positives [58].
D_{st}	True defects: the number of defects for which consensus was reached during the meeting in considering them as true defects [55].
D_{sf}	Removed false positives: the number of defects for which consensus was reached during the meeting in considering them as not true defects, thus as false positives [55].
D_{id} -> D_{rf}	In the Discrimination stage, discussion takes place asynchronously as in a discussion forum. When a consensus has been reached, the moderator can mark potential defects as false positives, thus removing them from the list that will go to the author for rework [54].
D_{rf}	False positives were issues that were identified in the meeting but that were discovered not to be defects either during the meeting or after. The decision whether a defect was a false positive was done by the code review team [50].

I investigated the definitions of false-positive defects to survey the categories for false-positive defects. Table 2.3 shows the definitions of false-positive defects, percentages of false-positive defects, and definitions of true defects for each article. No article categorized false-positive defects into incorrectly detected defects and concerns that are not applicable to the software review material. Eighteen articles described the definitions of both true and false-positive defects. No article described effective questions. One article [54] presented the format of a question list in a software review, but it did not refer to concerns identified by the questions in the question list.

Table 2.3 shows that not only the definitions of false positives but also those of true defects were inconsistent among the articles. Table 2.3 also shows the percentages of false-positive defects to the sum of true and false-positive defects. The percentages varied from 20% to 80%. More than half of the articles did not indicate the percentages or refer to the false-positive defects. Ten articles referred to categorizing detected defects into true and false-positive defects but did not report the percentages of the false-positive defects. Eight articles reported the percentages of detected false-positive defects.

The results of the literature review showed that no article referred to questions and concerns that were categorized into true or false-positive defects. Additionally, no literature categorized true defects into defects directly detected and shared by a reviewer or those found by concerns identified by effective questions and subsequent discussions. Therefore, I investigated whether the number of questions identifying concerns leads to an indicator of effective software reviews and helps a project manager identify an insufficient software review.

Table 2.3 Definitions and percentages of false-positive defects.

Article	Definitions of false-positive defects	Percentages of false-positive defects	Definitions of true defects
[16]	False positives (issues raised as defects that are not actual defects) False positives, the number of invalid defects recorded by the group	22%	Defects, the total number of distinct, valid defects detected by a group
[45]	False positives (issues raised as defects that are not actual defects)	22%	Actual defects
[50]	False positives were issues that were identified in the meeting but that were discovered not to be defects either during the meeting or after	22%	If the code review team finds an issue and agrees that it is a deviation from quality, the issue is counted as a defect
[51]	False positives (no real usability problems)	43.10%	Real usability problem
[52]	False positives (reported defects that were not considered to be actual defects)	-	Actual defects
[54]	False positives (non-true defects) False positives (defects erroneously reported as such by inspectors)	46%	True defects
[55]	For which consensus was reached during the meeting in considering them as not true defects, thus as false positives	-	True defects: the number of defects for which consensus was reached during the meeting in considering them as true defects
[56]	False positive (any issue which required no action)	20%	True defect (if the author was required to make an execution affecting change to resolve it), soft maintenance issue (any other issue which the author fixed)
[57]	False positives are items reported by subjects as defects, when in fact no defect exists	-	Defects
[59]	A false positive is a description which is not a true defect, i.e., does not require rework	-	A true defect is a description of a positively identified defect which requires rework; it causes the program to fail, and violates the given specifications and design
[60]	False positives (erroneously identified defects) False positives are the non-true defects—defects that require no repair	42.62%	True defects
[61]	It classifies too many consistent designs as inconsistent (false positives)	-	True positive
[62]	False positive (FA)—defects that do not exist but were wrongly identified	-	True defects (TR)—defects that actually exist and have been successfully detected

[63]	False defect estimations, known as false positive	-	The number of true defect estimations, known as true positive
[64]	False positive rate: the percentage of issues reported by an inspector that turn out not to represent real quality problems in the artifact	80%	Defect detection rate: the percentage of known defects in a given software artifact that are found during the inspection
[65]	False positives (not identified from preparation)	-	True defects Net defects
[66]	A false positive—an obviously wrong statement of the document.	-	True defect
[67]	False positive—items pointed by the subjects that do not correspond to a defect of the RD RD: the Requirements Document	-	Defects—items that really are defects of the RD RD: the Requirements Document

2.3 Case Study

2.3.1 Goal

This evaluation investigated whether the number of effective questions in software reviews could predict software quality. Specifically, the metric defect detection rate in testing (*Q*) was used as the quality of the software, where *Q* = [number of defects detected in testing]/[lines of source code]. The evaluation examined whether the number of effective questions could predict the defect detection rate in testing *Q* by performing multiple regression analysis because multiple parameters may affect *Q*. The independent variables of the multiple regression analysis include the number of effective questions in the software reviews. This evaluation assumes that effective software reviews decrease the number of defects detected in testing because effective design and code reviews reduce defects overlooked in the software reviews. Consequently, defects detected in subsequent testing are reduced.

2.3.2 Projects

The data for the evaluation were collected from a Japanese software development Company *Sa*. The standard software development process in Company *Sa* is based on the waterfall model and follows the process areas Organizational Process Definition (OPD) and Integrated Project Management (IPM) defined in CMMI-DEV V.1.3 [68]. The standard process also defines software measurements and metrics. In each software development project in Company *Sa*, the standard development processes require that detected defects and review logs including review comments in software reviews should be recorded in a defect list and that the detected defects in testing should be recorded, too.

The standard software development process of Company *Sa* requires that each project performs design and source code reviews. The software reviews are performed in a synchronous (face-to-face meeting) or asynchronous (adding detected defects to defect lists on a defect tracking server) manner. The standard process of Company *Sa* also requires that each reviewer complete review training and have detailed knowledge on the product domain to participate in a software review.

The evaluation used metrics collected in 25 projects of Company *Sa*. First, I selected 33 completed projects between April 2010 and March 2016 in Company *Sa*. Second, for each of the 33 projects, I checked that the metrics did not have missing values for review metrics, review logs, and defect metrics in testing. Eight projects were excluded due to the missing values. Finally, I measured the number of effective questions categorized as false-positive defects from the review logs of the remaining 25 projects. The reviewers of the case study categorized the effective questions into true defects or false-positive defects. If the reviewers categorized the effective questions into true defects, they were recorded as true defects in the defect list. A quality assurance team in company *Sa* verified the categorizations.

The 25 projects were for the development of embedded systems software, including safety-critical systems software, specifically, communication control systems software, engine control systems software, and browsing systems software. The development types were new development from scratch, enhancement of the same product, and reuse from another product. The number of project members varied from 3 to 20. The number of years of software development experience of the project members varied from 1 to 25 years. The lines of source code varied from 3000 to 1,100,000 lines written in C, C++, or Java.

2.3.3 Metrics and Procedure

Table 2.4 shows the metrics, excluding the number of effective questions categorized as false-positive defects, collected for project management defined by the standard development process. The product size (SZ) was used to assess the project progress management defined in the standard software development process. In Table 2.4, SZ is equal to the lines of code developed in the project without reusing code. In the development of enhanced or evolved development projects reusing an existing code base, SZ is equal to the sum of the lines of newly developed code (nLOC), lines of changed code from the code base (cLOC), and lines of reused code (rLOC) with a coefficient. The standard development process determines the coefficient according to the project attributes, such as the product domain and development types, to assess effort consumption to the product size in the project management. The number of effective questions categorized as false-positive defects (NOQf), not the number of effective questions, was measured because the effective questions categorized as true defects and rNOD would be double-counted. The metric of NOQf was measured from the review logs. The standard development process defines that review logs should include questions, which affect the quality of the product because some of the products in Company *Sa* are embedded in safety-critical systems. Consequently, the review logs could be used as a part of accountability for safety, if needed.

Table 2.4 Measured metrics for project management.

Name		Description
Lines of code	New (nLOC)	Lines of code newly developed, excluding headers and comments
	Changed (cLOC)	Lines of code changed from the code base or reused source code, excluding headers and comments
	Reused (rLOC)	Lines of code reused from the code base or another product, excluding headers and comments
Product size (SZ)		Product size for assessing development effort consumption in the project management defined by the standard development process. SZ = nLOC + cLOC + rLOC × coefficient (where the coefficient is determined by the project attributes)
Number of defects and questions in reviewed	True defects (rNOD)	Sum of the number of defects detected in software architecture design, software detailed design, and code reviews
	Effective questions categorized as false-positive defects (NOQf)	Sum of the number of effective questions subsequently categorized as false-positive defects detected in software architecture design, software detailed design, and code reviews
Number of defects detected in the test (tNOD)		Sum of numbers of defects detected in the unit test, software integration test, and software qualification test

Table 2.5 shows the derived metrics from the metrics shown in Table 2.4 for this evaluation. The dependent variable was *Q*, which measured the software quality in the standard software development process, because it was an indicator of the software quality in Company *Sa*. The independent variables included the proportion of rLOC to the total lines of code (p_1), and proportion of the number of true defects detected in software reviews (rNOD) to SZ (p_2). These independent variables were used in the project management and were defined in the standard software development process. The denominator of p_1 was nLOC + cLOC + rLOC. The standard software development process included metric p_1 because it was an indicator to estimate the productivity and had a higher correlation with the number of detected defects in the past developments. The standard software development process included metric p_2 because it was used as an evaluation criterion to measure the effectiveness of software reviews. The remaining

Table 2.5 Derived metrics for the analysis.

	Name	Description
Q	Proportion of the number of defects detected in testing to the product size	tNOD/SZ
p_1	Proportion of the reused lines of code to the lines of code	rLOC/(nLOC + cLOC + rLOC)
p_2	Proportion of the number of true defects to the product size	rNOD/SZ
p_3	Proportion of the number of effective questions categorized as false-positive defects to the product size	NOQf/SZ
p_4	Proportion of the number of effective questions categorized as false-positive defects to the sum of the number of defects and effective questions categorized as false-positive defects	NOQf/(rNOD + NOQf)

independent variables, proportion of NOQf to SZ (p_3), and the proportion of NOQf to the sum of rNOD and NOQf (p_4) were measured. If NOQf was large (large p_3), true defects were likely to be overlooked in the software reviews because the discussions and concerns may have missed the point. If the proportion of NOQf to the sum of rNOD and NOQf was large (large p_4), true defects were likely to be overlooked. The review time to detect the true defects was insufficient when the value of p_4 was large and the review time was a constraint. For metric p_3, I normalized NOQf by SZ because they largely depended on SZ as well as the independent variable p_2. For metric p_4, as described in Subsection 2.2 , I normalized NOQf by the sum of rNOD and NOQf because rNOD may affect NOQf in the software reviews. Specifically, the sum of rNOD and NOQf was likely to be limited due to available time and effort for the software reviews.

In the multiple regression analysis, I selected significant independent variables using the stepwise method. The evaluation investigated whether metrics of NOQf (p_3 and p_4) could predict the metric of the number of detected defects in testing (Q).

Table 2.6 Distribution of the measured metrics.

	SZ	rNOD	NOQf	tNOD
max	110,5000	1871	384	711
min	1330	32	0	3
median	144,390	589	99	171

Table 2.7 Distribution of the independent and dependent variables.

	Q	p_1	p_2	p_3	p_4
max	6.50	0.97	46.29	7.90	0.29
min	0.54	0.00	0.55	0.00	0.00
median	2.61	0.75	12.03	1.42	0.17

Table 2.8 Results of the multiple regression analysis.

	Estimate (b)	Std. Error	t Value	Pr(>|t|)	VIF
p_1	2.53	0.88	2.89	0.01	1.07
p_3	0.42	0.15	2.91	0.01	1.38
p_4	−9.68	3.69	−2.63	0.02	1.46

2.3.4 Results

Table 2.6 shows the distribution of the measured metrics. Table 2.7 shows the distribution of the dependent and independent variables. Table 2.8 shows the results of the multiple regression analysis. Metrics p_1, p_3, and p_4 contained significant coefficients. The variance inflation factor (VIF) values indicated that there was no multicollinearity among the variables. The adjusted R^2 of the model was 0.45 (p = 0.0013).

From the coefficients in Table 2.8, the model is expressed as

$$Q = 1.71 + 2.53p_1 + 0.42p_3 - 9.68p_4$$

The metrics of NOQf (p_3 and p_4) affected Q. The proportion of NOQf to SZ (p_3) increased Q. The proportion of NOQf to the sum of rNOD and NOQf (p_4) decreased Q. Specifically, the metric p_3 (ranging from 0.00 to 7.90) increased Q (ranging from 0.54 to 6.50). The coefficient of p_3 was 0.42 ($p = 0.01$). The metric p_4 (ranging from 0.00 to 0.29) decreased Q. The coefficient of p_4 was −9.68 ($p = 0.02$). When the sum of the third and fourth terms of the model was zero or larger, the accuracy of Q was larger than when the sum was less than zero.

2.4 Discussion

2.4.1 RQ: Does the Number of Effective Questions in a Software Review Affect the Quality of Subsequent Testing?

The results of the case study indicated that the answer to RQ is yes. In the case study, p_3 (the proportion of NOQf to SZ) positively affected Q (tNOD to SZ). I did not assume that p_4 (the proportion of NOQf to the sum of rNOD and NOQf) negatively affected Q (the proportion of the number of defects detected in subsequent testing (tNOD) to SZ). Hence, I investigated the software review details. I found that the software review materials contained a small number of defects. Almost all defects were detected in the software reviews. The number of defects detected in subsequent testing was small. Furthermore, the reviewers took a shorter time to detect almost all the defects, indicating that reviewers had enough time to ask additional effective questions to ensure that they did not overlook the remaining defects. In the discussion with the reviewers, they indicated that the potential true defects were shared before they asked effective questions and discussed them in higher quality projects. This suggests that sharing potential true defects has a higher priority than asking the effective questions and subsequent

discussions due to the limited review time. Facilitating effective questions and discussions after sharing potential true defects directly detected by reviewers may improve the software review effectiveness. Furthermore, the results may imply that effective questions and discussions trigger the Phantom Inspector effect [69].

The case study suggests that NOQf and rNOD can be used as a metric to measure the effectiveness (quality) of software reviews because metrics p_3 and p_4 affected Q. The metric can help a project manager identify an insufficient software review. It reveals that the project manager should plan additional software reviews with expert reviewers and/or more resources for subsequent testing.

2.4.2 Implications for Practitioners

The number of effective questions categorized as false-positive defects can be used as a metric to measure the software quality required by process models. The reviewers commented that the proposed metric can meet the requirements in process areas QPM.SP.1.4 in the CMMI [68] and MAN.6.BP4 in the Automotive SPICE [70]. The proposed method has a high usability from two perspectives. First, it can be used in various types of reviews, including code reviews with support tools. Second, it can objectively determine whether a question is effective, and the number of effective questions categorized as false-positive defects can be measured easily. Moreover, the proposed method is efficient even for cost-sensitive software development because categorizing effective questions and measuring the number of them can be performed in a short time.

In an iterative development process including agile development [71], [72], the proposed method can predict the product quality in each iteration. Although design reviews might not be implicitly performed in some iterative development processes, the essence of the proposed method can be applied for architectural and implementation discussions or comments in code reviews.

2.4.3 Threats to Validity

In the case study, the criterion for distinguishing effective questions from other questions may be biased. However, the reviewers in the case study selected effective questions based on whether or not the question identified a concern. Furthermore, after the reviewers selected the effective questions, an assessor in the quality assurance department verified that each effective question identified a concern.

In the case study, the variance of difficulties for projects may affect *Q*. However, the standard development process should mitigate such variance. For projects with technical challenges such as deploying novel technologies, prior development and verification were conducted. For projects whose members did not have sufficient domain knowledge on the product, additional developers and reviewers with sufficient domain knowledge were invited to the software reviews.

Identifying effective questions is potentially difficult. However, in this case study, the reviewers asked questions and categorized effective questions. Furthermore, identifying effective questions has previously been reported. One study [39] showed that some of the review comments could be categorized as questions, and about half of the approximately 470 questions helped reviewers detect defects.

2.5 Conclusions

This section proposed a review metric measuring the number of effective questions which identifies concerns about potential defects. Effective questions and subsequent discussions lead to defect detections if concerns identified by the effective questions and discussions are applicable to review materials, whereas concerns that are not applicable are discarded or recorded as false-positive defects. I performed a literature review to investigate whether previous studies referred to such effective questions and concerns.

2.5 Conclusions

The results of the literature review showed that no article referred to questions and concerns that were categorized into true or false-positive defects. Additionally, no literature categorized true defects into defects directly detected and shared by a reviewer or those found by concerns identified by effective questions and subsequent discussions.

I conducted a case study to investigate the effectiveness of the metric. The case study measured the number of questions, which ensures that the authors and reviewers do not overlook defects in terms of the concerns identified by the questions. The case study evaluated the impact of the number of effective questions on the number of defects in subsequent testing by multiple regression analysis. The independent variables were the proportion of reused lines of code, proportion of true defects detected in software reviews to the product size, proportion of effective questions categorized as false-positive defects in software reviews to the product size, and proportion of effective questions categorized as false-positive defects to the sum of true defects and effective questions categorized as false-positive defects. The dependent variable was the proportion of the defects detected in testing to the product size. The evaluation used metrics collected in 25 projects in a company. As the proportion of the number of effective questions categorized as false-positive defects to the sum of the number of true defects and effective questions categorized as false-positive defects (ranging from 0.00 to 0.29) increased, the proportion of the number of defects detected in testing to the product size decreased (ranging from 0.54 to 6.50) ($b = -9.68$, $p = 0.02$). Additionally, as the proportion of the number of effective questions categorized as false-positive defects to the product size (ranging from 0.00 to 7.90) slightly increased, the proportion of the number of defects detected in testing to the product size increased ($b = 0.42$, $p = 0.01$).

3. An Analysis on a Case Study of Requirements Ambiguities

3.1 Introduction

Software defects from requirements and specification ambiguities are harder to identify compared to other types of defects. Defects from ambiguities cause misunderstandings because each developer interprets the requirements and creates incorrect materials. Hence, the subsequent software development activities proceed under incorrect assumptions. In this situation, because checklists used in software reviews and test cases are created under incorrect assumptions, misunderstandings may not be discovered during software reviews and testing.

One approach that reduces such ambiguity defects is guided software reviews. Value-based review [73], a guided review technique, reduces effort for tailoring review scenarios. Value-based reviews aim at detecting high-priority defects efficiently decreased risk of overlooking. Priority is determined by artifact priority and defect criticality. However, in the situations in the Mars Climate Orbiter lost communication just before arriving on Mars in 1999 [33], determining the appropriate artifact priority and defect criticality are rather difficult.

I conducted a case study that analyzes defects that required significant correction effort during a commercial control simulation software development. The case study also analyzed the causes of the defects. Based on the results of the case study, this section proposes a goal-oriented approach for specifying requirements. I conducted a simple

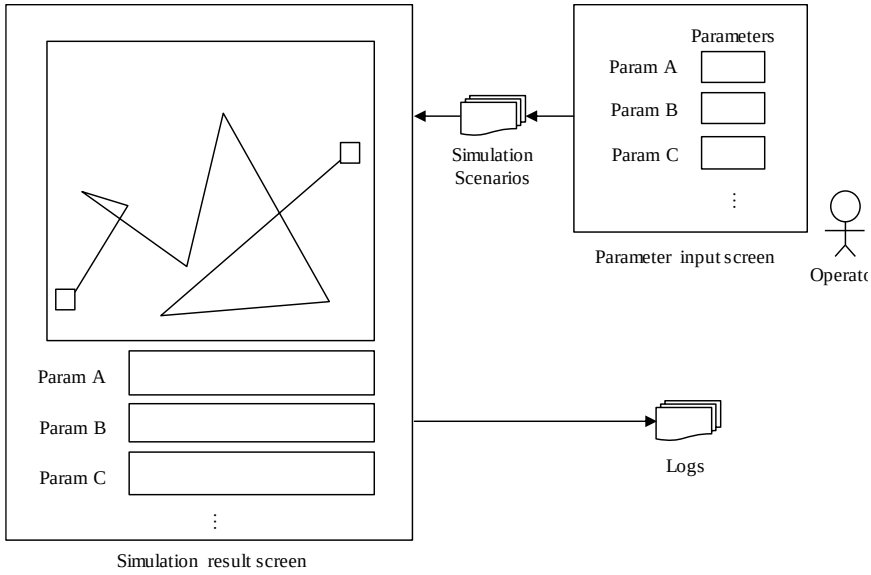

Figure 3.1 Data flow and user interfaces of system Sb-A.

evaluation for the efficiency of the approach in the case study by estimating its expected effort reduction.

3.2 Case Study

3.2.1 Target System

The target software system is the control simulation software developed by Company *Sb*. Figure 3.1 shows the data flow and user interfaces of the system Sb-A. System Sb-A consists of several subsystems including control simulator subsystem and data creation subsystem. System Sb-A receives parameters from the operator as parameters for simulation. System Sb-A calculates simulation results from predefined simulation scenarios and given parameters. Finally, system Sb-A stores the simulation results to the simulation logs.

Table 3.1 summarizes target system Sb-A and development project *P*. This project involved a development team of ten developers. They had enough knowledge of OS and programming languages, but none had experience related to developing control simulations. The development period was nine months. Target processes are the processes from software design to software integration testing. The requirements were defined by other members than the ten developers.

Table 3.1 Summary of the target system and development project.

Type of software	Control simulation software
Development period	Nine months
Developers	Ten software developers in Company *Sb*
Size (the number of lines of code)	220K lines
OS	Windows
Programming languages	C++ and C#
Target processes	From software design to software integration testing

Table 3.2 Number of defects injected in processes.

Injected process	Number of defects	Percentage
Software requirements analysis	29	24.2%
Software architectural design	52	43.3%
Software detailed design	12	10.0%
Coding	15	12.5%
Others	12	10.0%
Total	120	100.0%

Table 3.2 shows the number of defects injected in software development processes. The defects in Table 3.2 are detected in the system testing. During system testing, 120 defects were detected. Of these, 29 were injected in software requirements analysis process and 52 in software architectural design process.

3.2.2 Defects Cause Analysis

I analyzed defects that required large correction effort and were injected in software requirements analysis and software architectural design processes. The results of the analysis revealed that there were three defects causes in requirements ambiguities and inconsistent understanding among the developers. All three were categorized as definitions of the important terms. The details of each situation are described below:

1) Ambiguity and misunderstanding of units of distance

This is attributed to the fact that the architectural design developers were unaware that multiple interpretations existed for the unit “mile.” Figure 3.2 shows this defect. As described in Figure 3.2, one nautical mile is 1852.00 meters. One (international) mile is 1609.344 meters.

(a) Incident

The distance value displayed on the screen differed from the expected one.

(b) Cause

The defect was due to the misinterpretation of the unit “mile.” The requirements defined clearly that “the unit used is ‘nautical mile.’” However, the software design developers were unaware of multiple interpretations of “mile” and assumed “nautical mile” could be expressed as “mile.” Therefore,

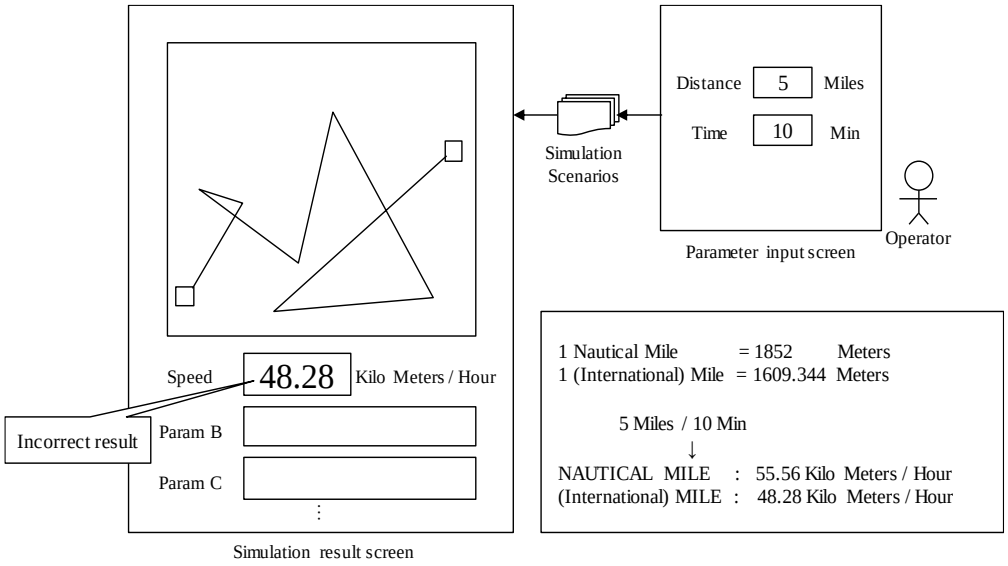

Figure 3.2 Ambiguity and misunderstanding of units of distance.

the units differed between requirements and design, causing a significant defect, which was identified during system testing.

2) Undefined time standard

This defect resulted from an undefined time standard in software design because of the absence of the definition in software requirements.

(a) Incident

In system Sb-A, the displayed time in screens in some subsystems were inconsistent and different from the expected one. Figure 3.3 shows this defect.

(b) Cause

System Sb-A consisted of multiple subsystems. Requirements for each subsystem had its own separate requirements specification document. The majority of the requirements specifications stated the time standard was

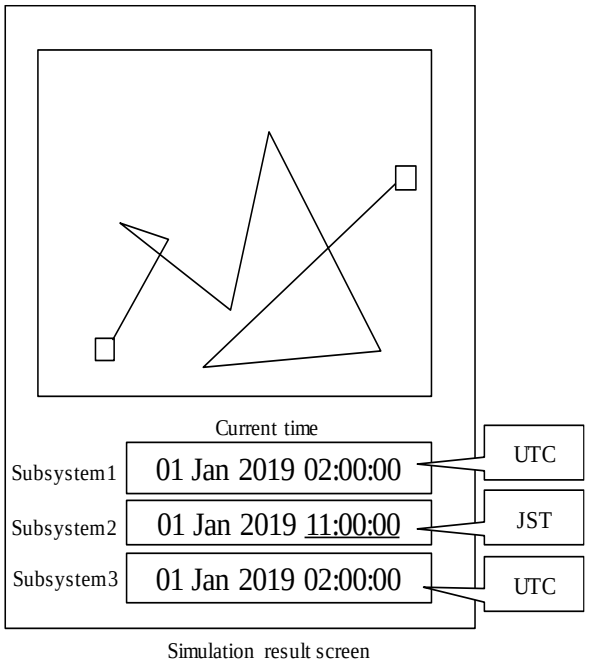

Figure 3.3 Time standard inconsistency among the subsystems.

Coordinated Universal Time (UTC), but some specifications did not have a clear definition of the time standard. As a result, the architectural design developers who worked on the specification without a clear definition of the time standard assumed Japan Standard Time (JST). Each subsystem were developed by a different architectural design developer and the developers did not have opportunities to communicate. Consequently, they did not realize the time standard inconsistency among the subsystems.

3) Inconsistent definitions of significant digits

This defect resulted from not defining the number of significant digits after the decimal point for the values dealt in system Sb-A. Figure 3.4 shows insufficient

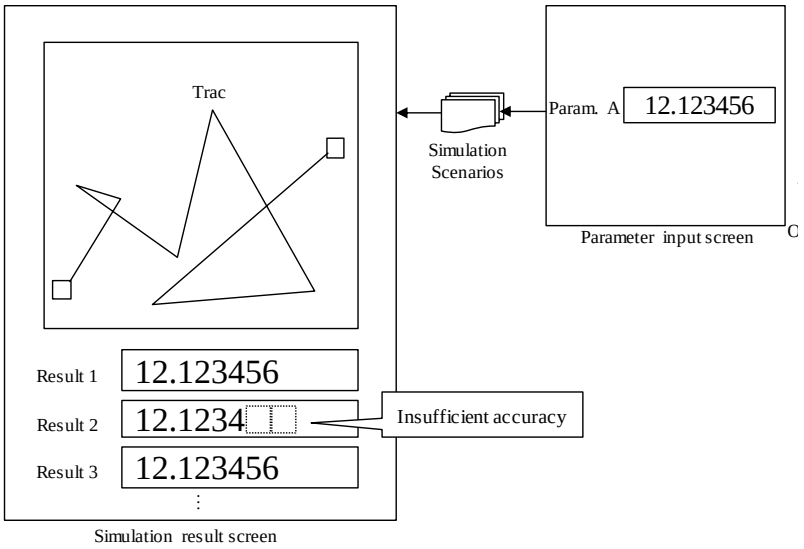

Figure 3.4 Insufficient accuracy.

accuracy in some of the simulation results. In Figure 3.4, results 1, 2, and 3 were calculated by different subsystems.

(a) Incident

For the system input parameters, after the fifth decimal point, it was not treated as a significant digit.

(b) Cause

Each subsystem had its own separate requirements specification document, and the number of significant digits varied between four or six. Architectural design developers defined the significant digits according to the requirements specification for the subsystems. Therefore, the developers assigned to the subsystems where the requirement specification defined significant digits as four digits assumed that there was no need to deal with the values more than four digits after the decimal point.

3.2.3 Defects Analysis

The three defects shown in the previous subsection are caused by requirement ambiguity and misunderstandings and inconsistencies among the developers. In this subsection, I analyze the root cause of such defects. Below are the factors that lead to misunderstandings and inconsistencies.

1) Communicating Tacit Knowledge

One of the failure factors of project *P* was tacit knowledge was not shared properly among the developers. The developers started the software development without opportunities to share information and knowledge required by the control simulation software system development.

When including tacit knowledge into the glossary, not only the description of the keywords that cause frequent defects, but also additional content, including the background information, is necessary [74]. In project *P*, a glossary was organized. However, project *P* was the first control simulation software system development for the developers. Hence, the developers were unfamiliar with the terminologies not included in the glossary because they were tacit knowledge.

2) Using Experts

To be effective, the software review process must include experts. In project *P*, the lack of experts was identified as a risk factor from the beginning of the project. As a countermeasure, the project leader assigned expert *E* as an advisor. Although expert *E* had plenty of knowledge in control simulation software system field, he was not a full-time member of project *P* and was assigned to another project. Because the project leader could not secure expert *E*, his knowledge was not shared and utilized in project *P*.

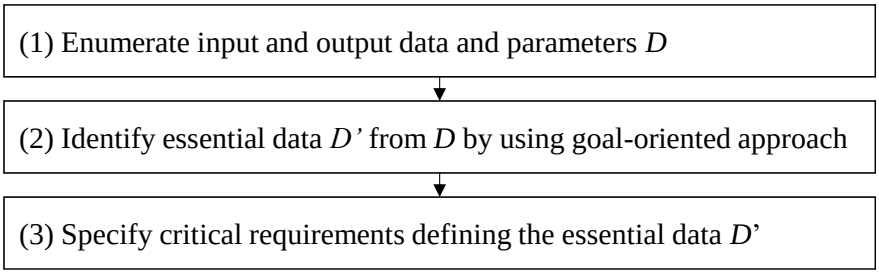

Figure 3.5 Procedure of the proposed approach.

3) Sharing information

Sharing defect information among developers can effectively increase the number of similar defects detection and increase the opportunities for defect prevention. The defects detected during the software reviews and testing processes can be shared and among the project members; however, there was no opportunity to share defect information which was not found as a defect.

3.3 A Prevention Approach for Focusing Essential Data

3.3.1 Overview

Under cost constraints, an approach for detecting all defects detected in project *P* is difficult to realize. It will also cost to list all background information and common knowledge required during requirements analysis or to involve an expert in all projects. To prevent injecting critical requirements defects that require large correction effort, I propose an approach for preventing such defects as the three significant defects detected in project *P*. Figure 3.5 shows the procedure of the proposed approach. The procedure

Table 3.3 Examples of critical requirement definitions.

Critical requirement	Definition	Supplemental definition
Distance unit	Nautical mile	1 nautical mile = 1,852 meters. Please note that there are multiple interpretation of mile.
Time standard	Coordinated Universal Time (UTC)	UTC = JST – 9 Japan Standard time (JST) must be converted to UTC.
Significant digits	Six digits after the decimal points	If there are more than seven digits after the decimal point, round the seventh digit after the decimal point.

consists of (1) enumerating the input and output data *D* for the target system *T*, (2) identifying essential data *D*', and (3) specifying critical requirements for the essential data *D*'. Essential data *D'* can be found by a goal-oriented approach [75].

3.3.2 Identifying Essential Data

The approach first enumerates a set of input and output data *D* of target system *T*. Then, the approach identifies a set of essential data *D*' from the set *D* along with the goal-oriented approach. The goal-oriented approach focuses on the goal of the target system *T*. For example, the goal of the system Sb-A is to obtain simulation results from given parameters. Therefore, essential data *D*' are the input parameters for simulation scenarios and the results of the simulation. Essential data *D*' included distance and time because incorrect parameters lead to incorrect simulation results.

3.3.3 Critical Requirements Definition

The approach identifies and specifies critical requirements of the target system *T*. The critical requirements define the set of essential data *D*'. For system Sb-A, the critical requirements definition include distance unit for distance parameters, time standard for time parameters, and calculation accuracy for simulation results. Table 3.3 shows an example of critical requirement definitions.

Table 3.4 Estimated effort.

Task	Effort
Correcting the three defects	36 person-hours
Identifying the essential data and critical requirements	7 person-hours
Defining the critical requirements	8.7 person-hours
Improved effort	20.3 person-hours

3.3.4 Estimated Effort Reduction

I conducted a simple evaluation for the proposed approach. The evaluation first enumerated a set of input and output data *D* of system Sb-A and identified a set of essential data *D*'. The evaluation identified critical requirements for *D*' and estimated effort for specifying the critical requirements based on the estimation by the project leader with development experience of the product domain.

The results of the evaluation showed that there are 52 critical requirements for essential data *D*'. If each of critical requirements had a clear definition, at least three defects described in the previous subsection could have been avoided. Simply adding definitions and notes may significantly reduce the likelihood of the defect injections.

The evaluation estimated effort for specifying 52 critical requirements and compared effort for correcting the three defects. Table 3.4 shows the results of the estimation. It took 36 person-hours correcting the three defects. On the other hand, it took seven person-hours to identify essential data *D'* from *D*, and 8.7 person-hours to specify critical requirements defining the essential data *D'*. The total was 15.7 person-hours, and the proposed approach improved 20.3 person-hours. The results indicated that a 43.5% effort improvement can be achieved by the proposed approach compared to the case without clear requirements definitions.

3.4 Conclusions

This section conducted a case study in a simulation control software system for analysis on defects that required significant correction effort to investigate context-dependent issues. The results of the case study indicated that defects that required correction effort were ambiguity defects injected by misunderstandings and inconsistencies among stakeholders during interpreting requirements and specifying design documents. In the target system, ambiguities are found in the definitions of distance, time (time zone), and calculation accuracy. These cause inconsistencies among the implementations and incorrect simulation results of the control simulation software systems.

Based on the analysis, as a low-effort defect prevention approach, I propose an approach for identifying essential data and specifying critical requirements for the essential data. Ambiguities of the critical requirements potentially lead to defect injections that may require significant correction effort when detected in later development processes. In addition, I conducted a simple evaluation for the proposed approach. The results of the evaluation showed that the essential data for the control simulation software system in the case study were distance, time, and simulation result (accuracy). The evaluation estimates the effort for defect corrections in the case study and the effort for specifying the definitions of the essential data in the case study. The results of the evaluation indicated that the proposed approach could achieve 43.5% effort reduction.

4. Goal-Oriented Software Design Reviews

4.1 Introduction

Inappropriate requirements can consume substantial rework effort in subsequent development activities. In particular, as described in Section 3, omissions and ambiguities in requirements may lead to extensive changes and corrections in the subsequent development activities. The causes of requirement omissions include missing functionality, missing performance, missing interface, and missing environment [19]. Ambiguity enables multiple interpretations of the requirements document [76]. Various approaches and methods to reduce omissions and ambiguities in requirements have been proposed. Software review is one such static analysis technique for the early detection of defects, including omissions and ambiguities, and does not require program execution [6], [77]. PBR [18] is designed to reduce omissions and ambiguities in requirements through multiple perspectives. The perspectives provide reviewers guides for finding defects from the viewpoint of stakeholders such as project managers, users, and testers. Goal-oriented requirements analysis [78] prevents missing requirements and facilitates requirements decomposition [79], [80]. Requirements decomposition increases the requirements coverage by defining goals at various levels of abstraction [79].

This section refers to omissions or ambiguities in requirements caused by design context as context-dependent requirement issues (CDRIs). A CDRI occurs when two or more different implementations realize the same requirement because the requirements

are defined before the implementations are defined. For example, the requirement “The data are exchanged with files. The fields in the file must be separated by a line break” can be realized by two implementations: one implementation for writing a data file and another implementation for reading the data file. If the two implementations are realized on the same operating system, no CDRI occurs because the line-break characters are the same between the implementations. However, a CDRI occurs if the two implementations are realized on different operating systems and do not consider the line-break characters for the other operating system. Specifically, if the implementation for writing a file is realized with UNIX (LF for a line-break character) and the implementation for reading the file is realized with Windows (CR and LF for a line-break character), the fields will not be separated properly despite each of the two implementations realize the requirement accurately. In this case, the design context is the line-break characters for the operating systems.

Feasibility or impact analysis which can help reviewers find such design contexts during the requirement process, requires extensive effort because the analyses check all implementations: not only two or more implementations which realize the same requirement, but also a single implementation. Identifying two or more implementations which are supported by a single requirement and checking consistencies among them in the design review can help detect inconsistencies caused by CDRIs.

To the best of our knowledge, no specific approach or method to detect CDRIs or context-dependent requirement (CDR) defects caused by CDRIs has been proposed. This section proposes a design review method to identify such inconsistencies among implementations realizing the same requirement by using a goal tree obtained by goal-oriented requirements analysis. The proposed method defines check items to find inconsistencies in the implementations, where the check items are created from the goal tree. This section also evaluates the proposed method through a case study with two criteria. First, the evaluation investigates whether the check items for design review can be defined from the goal tree and then whether the check items can detect CDR defects.

Second, the evaluation investigates whether the proposed method reduces the estimated rework effort to correct defects.

4.2 Related Research

Guided reviews are one approach to detecting defects caused by omissions or ambiguities in requirements. As described in Section 1, many studies have reported on the effectiveness of guided reviews. However, these reading techniques do not require that guides including checklists and scenarios verify inconsistencies among different implementations for the same requirement.

Goal-oriented requirements analysis [78], [81] is one of the methods to reduce omissions or ambiguities in requirements. Goal-oriented requirements analysis defines software requirements by clarifying the structured goals of the software. Goal-oriented requirements analysis also clarifies the background and necessities for requirements, facilitates requirements analysis discussions, and enhances the validation and tracking of changes to the requirements. Many goal-oriented requirements analysis methods have been studied, including the KAOS method [82-86], the i* framework [87-91], and the NFR framework [92-95]. However, detecting omissions or ambiguities in requirements caused by the design context determined in the design process is difficult because goal-oriented requirements analysis is performed during the requirements processes.

Traceability between requirements and design elements can verify that the requirements have been implemented as design elements in the design document [96-101]. Traceability studies have strongly focused on requirements traceability, with the objective of studying how to describe and follow requirements in both the forward and backward directions [97], [100]. Traceability is an effective guide to detect CDRIs; however, it is unclear whether traceability can examine consistencies among implementations. Although two methods [102], [103] check consistencies between

requirements and design documents, both of them check consistencies between different types of UML documents. Thus, they cannot always detect inconsistencies among the implementations in the same UML document.

Change impact analysis identifies where the changes affect [104-106], estimates the effort for implementing a change request [107], [108], and predicts necessary regression tests according to the set of changes [109]. However, change impact analysis cannot always detect inconsistencies among implementations. Automotive-SPICE [70] recommends analyzing the operational (execution) environment, including platforms, to analyze the feasibility of the requirements. Such operational environment analysis can detect implementation inconsistencies. However, it does not explicitly refer to detecting inconsistencies among implementations.

4.3 Proposed Method

4.3.1 Prerequisite

In the proposed method, reviewers attempt to detect inconsistencies among design implementations and ensure that the implementations satisfy the goal using goal-oriented check items. This section refers to the inconsistencies as CDR defects. CDR defects are caused by CDRIs. The proposed method defines goal-oriented check items taking a goal tree as input. The goal tree provides traceability links from high-level strategic objectives to low-level technical requirements [79]. The proposed method adds goal-oriented check items to the leaf nodes of the goal tree, which is created by goal-oriented requirements analysis. The goal tree consists of the top goal and subgoals. The top goal is the root node of the goal tree. The root node has a label describing the objective or state that the system should achieve. The top goal is decomposed into one or more subgoals (child nodes) because, without the decomposition, a goal tree may not provide technical requirements. The goal-oriented check items defined from the

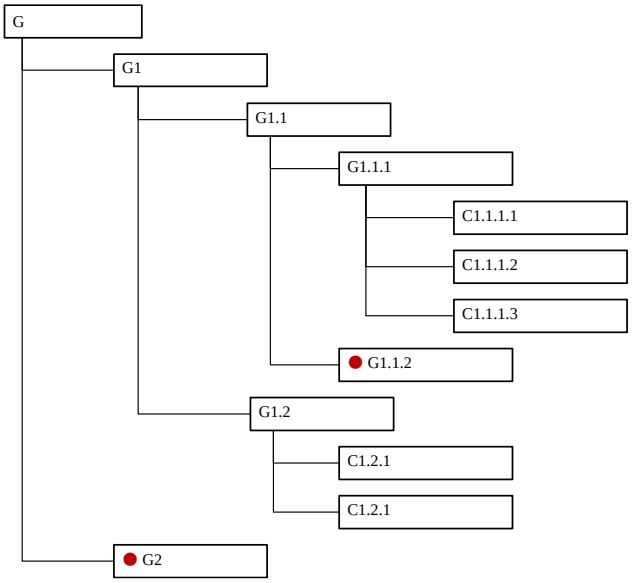

Figure 4.1 An example of a goal tree. Red circles represent pruned subgoals.

subgoals that do not satisfy the desired requirement cannot detect inconsistencies among context-dependent implementations. Thus, the decomposition should be performed carefully. The goal-oriented requirements analysis [84], [93] categorizes goals into three categories: functional requirements, non-functional requirements, and external constraints. Thus, the top goal and subgoals for the proposed method can be categorized into three categories. The node has a label describing the purpose or the status required to achieve the parent goal (parent node). Subgoals are recursively decomposed into sub-subgoals. The label description is a prescriptive statement of intent that the system should satisfy [82].

4.3.2 Procedure

(a) Identify the goal tree (top goal and subgoals). If goal-oriented requirements analysis has created a goal tree in advance (e.g., requirements analysis), the goal tree is reused. If the goal tree does not exist, the analyst

describes the goal of the system as the label of the top goal G. The analyst then decomposes the top goal G into subgoals G1, G2,…, G*m*. The analyst creates a simple label for each subgoal and adds the subgoals as child nodes of the top goal. The analyst decomposes the subgoals (G1, G2, …, G*m*) into sub-subgoals (G1.1, G1.2, G1.3, …, G2.1, G2.2, …, G*m*.1, G *m*.2, …) and continues decomposing the subgoals until the subgoals are complete, consistent, and minimal.

(b) Prune unnecessary subgoal nodes. The analyst selects and prunes unnecessary subgoal nodes, which do not need to be broken down further for consideration, such as duplicated subgoals. After pruning, each leaf node of the goal tree is marked as a leaf subgoal node.

(c) Define goal-oriented check items. Goal-oriented check items verify whether the design implementations are consistent and satisfy the corresponding subgoal. Goal-oriented check items are determined by two or more implementations that realize the same subgoal. The analyst defines one or more goal-oriented check items for each of the leaf subgoal nodes except pruned subgoals. The analyst then adds goal-oriented check items as child nodes of the leaf subgoal nodes. Figure 4.1 shows an example goal tree. Goal nodes and goal-oriented check-item nodes are labeled with symbols G and C, respectively. The pruned subgoals are indicated by red circles, as shown in G1.1.2 and G2.

(d) Perform goal-oriented software design review. Reviewers use the goal-oriented check items to attempt to detect inconsistencies among implementations realizing the same requirement.

Table 4.1 Overview of an example system.

Name	Electric kettle
Goal	The electric kettle heats water and keeps the water temperature constant.
Architecture and developers	The system consists of a sensor unit, control unit, and heating unit. Each unit was developed by different developers (no developer developed two or more units).

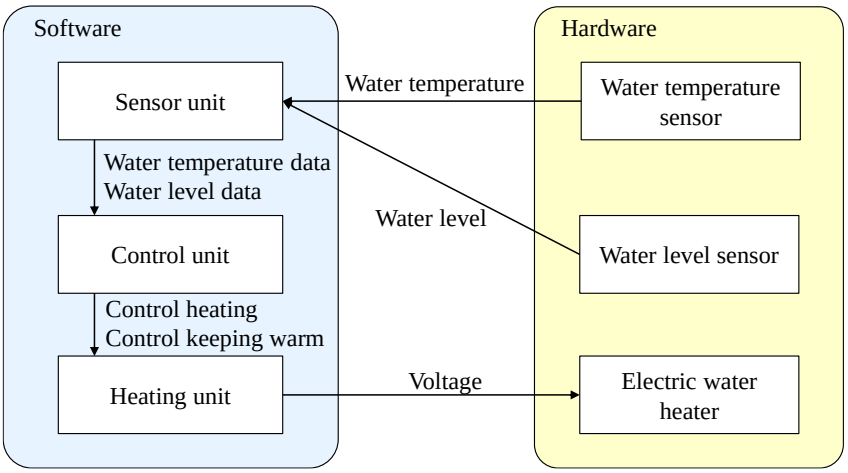

Figure 4.2 Architecture of an example system.

4.3.3 Example of A Goal Tree and Check Items

1) Overview of an Example System

This subsection presents a goal tree and the corresponding goal-oriented check items for an example system. Figure 4.2 shows an overview of the example system. The design implementations for the same subgoal can differ among the three units because the units had different developers.

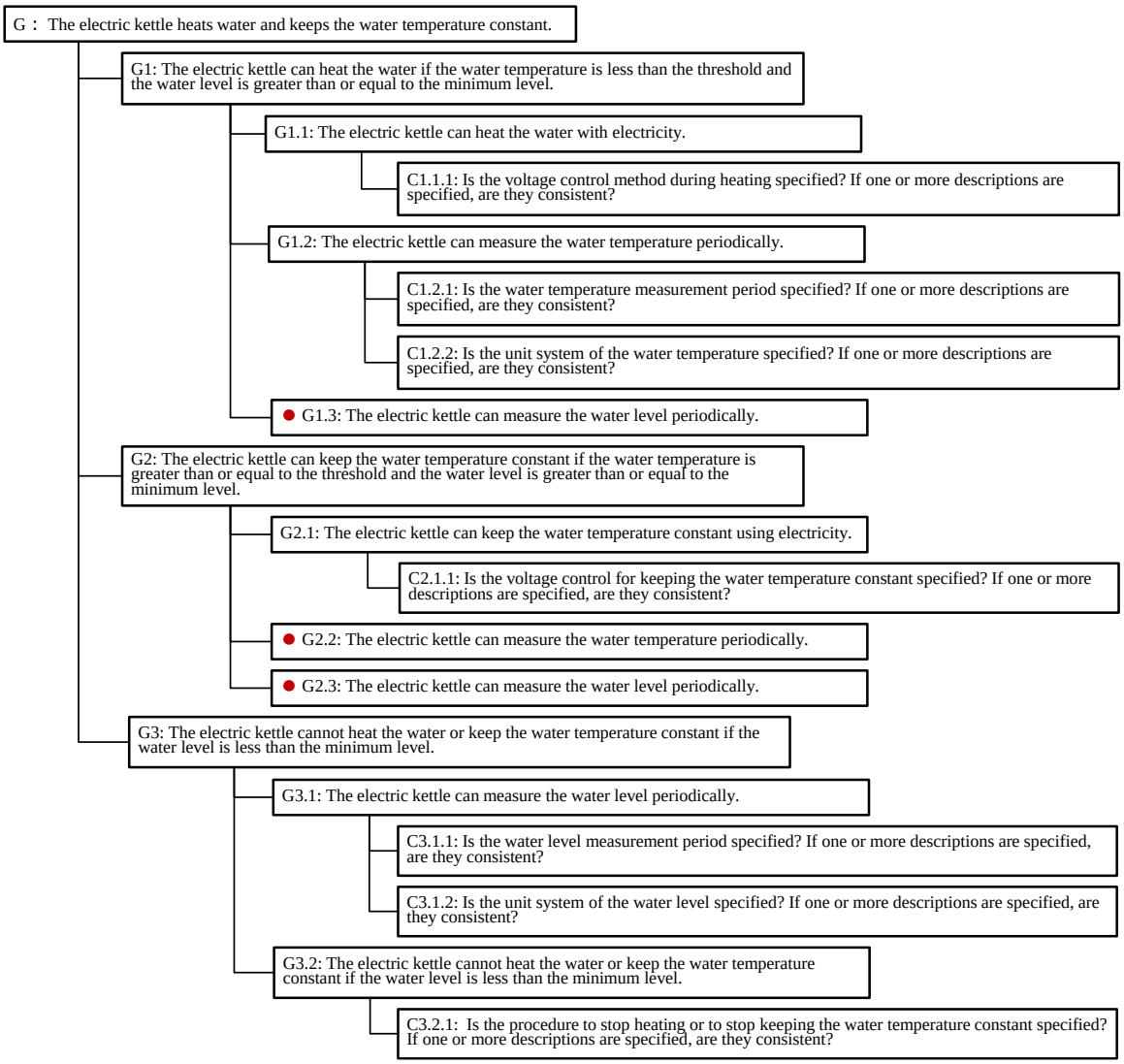

Figure 4.3 A goal tree for the example system.

2) Procedure

Figure 4.3 shows the goal tree and goal-oriented check items. The procedure is detailed as follows.

Step (1). The analyst identifies the goal tree. The identified goal tree consists of the following:

Top goal G: The electric kettle heats water and keeps the water temperature constant.

The analyst decomposes the top goal G into the following subgoals:

G1: The electric kettle can heat the water if the water temperature is less than the threshold and the water level is greater than or equal to the minimum level.

G2: The electric kettle can keep the water temperature constant if the water temperature is greater than or equal to the threshold and the water level is greater than or equal to the minimum level.

G3: The electric kettle cannot heat the water or keep the water temperature constant if the water level is less than the minimum level.

The analyst decomposes subgoal G1 into the following subgoals:

G1.1: The electric kettle can heat the water with electricity.

G1.2: The electric kettle can measure the water temperature periodically.

G1.3: The electric kettle can measure the water level periodically.

The analyst decomposes subgoal G2 into the following subgoals:

G2.1: The electric kettle can keep the water temperature constant using electricity.

G2.2: The electric kettle can measure the water temperature periodically.

G2.3: The electric kettle can measure the water level periodically.

The analyst decomposes subgoal G3 into the following subgoals:

G3.1: The electric kettle can measure the water level periodically.

G3.2: The electric kettle cannot heat the water or keep the water temperature constant if the water level is less than the minimum level.

Step (2). The analyst prunes unnecessary subgoal node G2.2 because G2.2 duplicates G1.2. The analyst also prunes unnecessary subgoal nodes G1.3 and G2.3 because G1.3 and G2.3 duplicate G3.1. The reason for keeping G3.1 instead of G1.3 or G2.3 is because G3 is the goal referring to the water level.

Step (3). The analyst defines goal-oriented check items as child nodes of the subgoals. The analyst defines the following goal-oriented check items from G1.1:

C1.1.1: Is the voltage control method during heating specified? If one or more descriptions are specified, are they consistent?

The analyst defines the following goal-oriented check items from G1.2:

C1.2.1: Is the water temperature measurement period specified? If one or more descriptions are specified, are they consistent?

C1.2.2: Is the unit system of the water temperature specified? If one or more descriptions are specified, are they consistent?

The analyst defines the following goal-oriented check item from G2.1:

C2.1.1: Is the voltage control for keeping the water temperature constant specified? If one or more descriptions are specified, are they consistent?

The analyst defines the following goal-oriented check items from G3.1:

C3.1.1: Is the water level measurement period specified? If one or more descriptions are specified, are they consistent?

C3.1.2: Is the unit system of the water level specified? If one or more descriptions are specified, are they consistent?

The analyst defines the following goal-oriented check item from G3.2:

C3.2.1: Is the procedure to stop heating or to stop keeping the water temperature constant specified? If one or more descriptions are specified, are they consistent?

Step (4). The reviewer performs goal-oriented software design review using the goal-oriented check items. For example, in G1.2, the developers of the sensor unit considered and defined the water temperature in Fahrenheit, whereas the developers of the control unit considered and defined the temperature in Celsius. The reviewer can detect this inconsistency between the definitions and implementation with goal-oriented check item C1.2.2.

4.4 Case Study

4.4.1 Goal

The goal of the case study is to investigate the effectiveness and efficiency of the proposed method. The case study was conducted with a commercial software system. An overview of the commercial software system is described in Subsection 4.4.2 . The case study evaluated whether the proposed method could define goal-oriented check items, whether the proposed method could detect CDR defects, and whether detecting CDR defects in design reviews contributed to a reduction of the rework effort for correcting the defects.

4.4.2 System Context

I selected the subsystems of System Sc-A developed in a Japanese software development Company *Sc* for this case study. System Sc-A was a communication network control system. Table 4.2 and Figure 4.4 shows the details. The development period was from April 2017 to March 2019. System Sc-A consisted of 12 subsystems. Each subsystem was developed from scratch. The number of developers for each subsystem varied from

Table 4.2 Overview of system Sc-A.

Name	Communication network control system
Goal	The system controls emergency communication for human safety. The system enables every terminal in the network to communicate with other terminals in the network.
Architecture	System Sc-A consisted of monitoring control unit, communication control unit, and data transmission unit. The monitoring control unit reused another software whose reliability was proved in another system in operation. No severe defects had not been found in the original software.

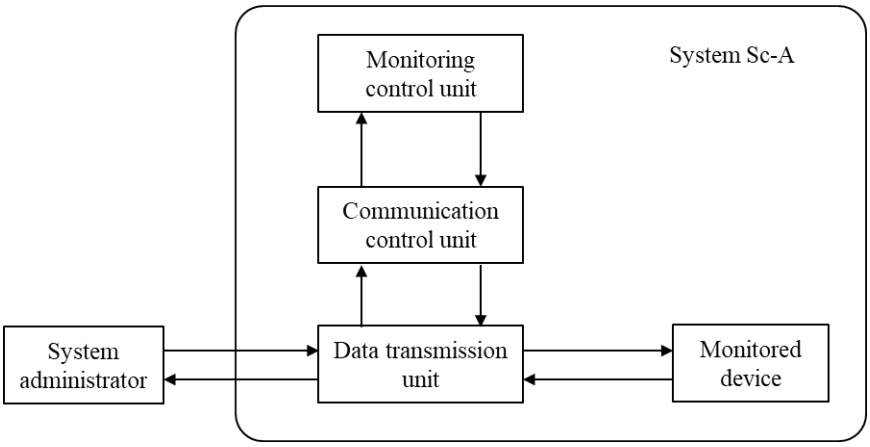

Figure 4.4 Architecture of System Sc-A.

three to seven. The number of years of software development experience of the developers varied from 2 to 25 years. The lines of source code of the subsystems varied from 3100 to 8400 lines in C language.

The standard software development process was based on the waterfall model and followed the process areas Organizational Process Definition (OPD) and Integrated Project Management (IPM) defined in CMMI-DEV V.1.3 [68]. The standard process

Table 4.3 Metrics for the evaluation.

	Name	Description
cH	Effort to define goal-oriented check items	Person-hours to identify a goal tree and define goal-oriented check items
grH	Effort for goal-oriented software design reviews	Person-hours to perform design reviews with the goal-oriented check items
gwH	Estimated additional rework effort	Difference between the sum of the estimated effort (person-hours) for investigating, fixing, and regression testing and the sum of the effort for fixing defects detected in the goal-oriented software design reviews
grD	Number of CDR defects detected in goal-oriented software design reviews	Number of CDR defects detected in design reviews with goal-oriented check items
srD	Number of CDR defects detected in standard design reviews	Number of CDR defects among defects detected in the standard design reviews
gtD	Number of CDR defects detected in subsequent testing	Number of CDR defects, which were overlooked in design reviews and detected in subsequent software testing

also defined software measurements and metrics. For each software development, the standard process required that the defects detected in software reviews should be recorded in a defect list and that the defects detected in testing should also be recorded. The standard software development process required that each project perform design reviews (standard design reviews), record the design review logs (meeting minutes), and use the standard design review checklists. The standard software development process of Company *Sc* also required that each reviewer complete the software review training and have detailed knowledge of the system domain to participate in the software review.

4.4.3 Metrics

The metrics cH, grH, gwH, grD, srD, and gtD (Table 4.3) were measured for this evaluation in addition to the metrics in the standard software development process of

Company *Sc*. These metrics are defined in Table 4.3. The metrics cH and grH are efforts for the preparation of the proposed method; metrics grD, srD, and gtD are the number of detected defects. Note that the metric gwH was the estimated additional rework effort (person-hours) that would be needed if the CDR defects detected in the proposed method were overlooked in goal-oriented software design reviews and detected and corrected in subsequent software testing. The metric gwH was analogously estimated according to the reviewers' related experiences and verified by the analyst.

4.4.4 Evaluation and Procedure

I selected four subsystems from the 12 subsystems of System Sc-A. I selected two subsystems 1a and 2a from the four subsystems for Evaluations 1 and 2. For Evaluation 3, from the remaining subsystems, I selected subsystem 1b with a goal similar to that of subsystem 1a. Similarly, I selected subsystem 2b with a goal similar to that of subsystem 2a.

1) Evaluation 1: Can an Analyst Define Goal-Oriented Check Items Using the Proposed Method?

Evaluation 1 evaluated whether an analyst (engineer) could identify a goal tree and define the corresponding goal-oriented check items. Following the steps in Subsection 4.3.2 , the analyst identified goal trees and defined goal-oriented check items for subsystems 1a and 2a. The analyst is a quality assurance engineer and one of the authors.

2) Evaluation 2: Can the Proposed Method Detect CDR Defects and Reduce the Defect Correction Effort?

Evaluation 2 consisted of the following evaluations:

Evaluation 2.1: Can reviewers detect CDR defects in goal-oriented software design reviews?

Evaluation 2.2: Can the proposed method reduce the estimated additional rework effort to correct CDR defects?

Evaluation 2.1 measured the number of CDR defects detected in goal-oriented software design reviews (grD). In addition, Evaluation 2.1 measured the number of CDR defects detected in subsequent testing (gtD) because overlooked CDR defects in goal-oriented software design reviews could be detected in subsequent software testing. Evaluation 2.2 measured the effort to define goal-oriented check items (cH), the effort for goal-oriented software design reviews (grH), and the estimated additional rework effort (gwH) to investigate whether the proposed method required less effort than the standard design reviews and the subsequent testing defined by the standard process. Specifically, Evaluation 2.1 deemed that the proposed method was feasible if CDR defects were detected in goal-oriented software design reviews (grD > 0) and the number of CDR defects detected in subsequent testing (gtD) was sufficiently small. In Evaluation 2.2, if the sum of the effort to define goal-oriented check items and the effort for goal-oriented software design reviews was smaller than the estimated additional rework effort (cH + grH < gwH), the proposed method was efficient.

Reviewers performed goal-oriented software design reviews for subsystems 1a and 2a using the goal-oriented check items defined in Evaluation 1. The goal-oriented software design reviews were performed in addition to the standard design reviews. After the goal-oriented software design reviews, the subsequent development activities, including software testing, were performed according to the standard software development process. The analyst measured and recorded the metrics in Table 4.3 (excluding srD). The analyst then categorized the CDR defects detected in goal-oriented software design reviews and the subsequent testing into defect groups corresponding to goal-oriented check items defined in Evaluation 1.

3) Evaluation 3: Are CDR Defects Detected in Other Similar Subsystems?

Evaluation 3 measured the number of CDR defects detected in the standard design reviews (srD) and subsequent testing (gtD) for subsystems 1b and 2b to investigate the applicability of the proposed method in other subsystems. Specifically, Evaluation 3

measured srD to investigate whether the standard design reviews detected CDR defects and gtD to investigate whether the standard design reviews overlooked CDR defects.

Evaluation 3 deemed that CDR defects existed in design documents for subsystems 1b and 2b if CDR defects were detected in the standard design reviews and/or subsequent testing (srD + gtD > 0). If CDR defects were present and the proposed method was carried out, goal-oriented software design reviews could possibly detect CDR defects. If CDR defects were detected in the standard design reviews and the number of detected CDR defects in the subsequent testing was sufficiently small (srD > 0 and srD >> gtD), the standard design reviews were considered to be able to detect most of CDR defects. If CDR defects were not detected in the standard design reviews or in subsequent testing (srD = gtD = 0), CDR defects were not considered to have been injected in the design documents.

Reviewers performed the standard design reviews for subsystems 1b and 2b. After the standard design reviews, the subsequent development activities, including software testing, were performed according to the standard software development process. The analyst categorized the CDR defects detected in the standard design reviews and subsequent testing into defect groups corresponding to the goal-oriented check items defined in Evaluation 1.

4.5 Results

4.5.1 Results of Evaluation 1

Figure 4.5 shows the goal tree and the goal-oriented check items which the analyst identified and defined. In Figure 4.5, subgoals G1 and G2 were realized by subsystems 1a and 2a, respectively. Notably, subgoals G2.1.2 and G2.2 were not broken down

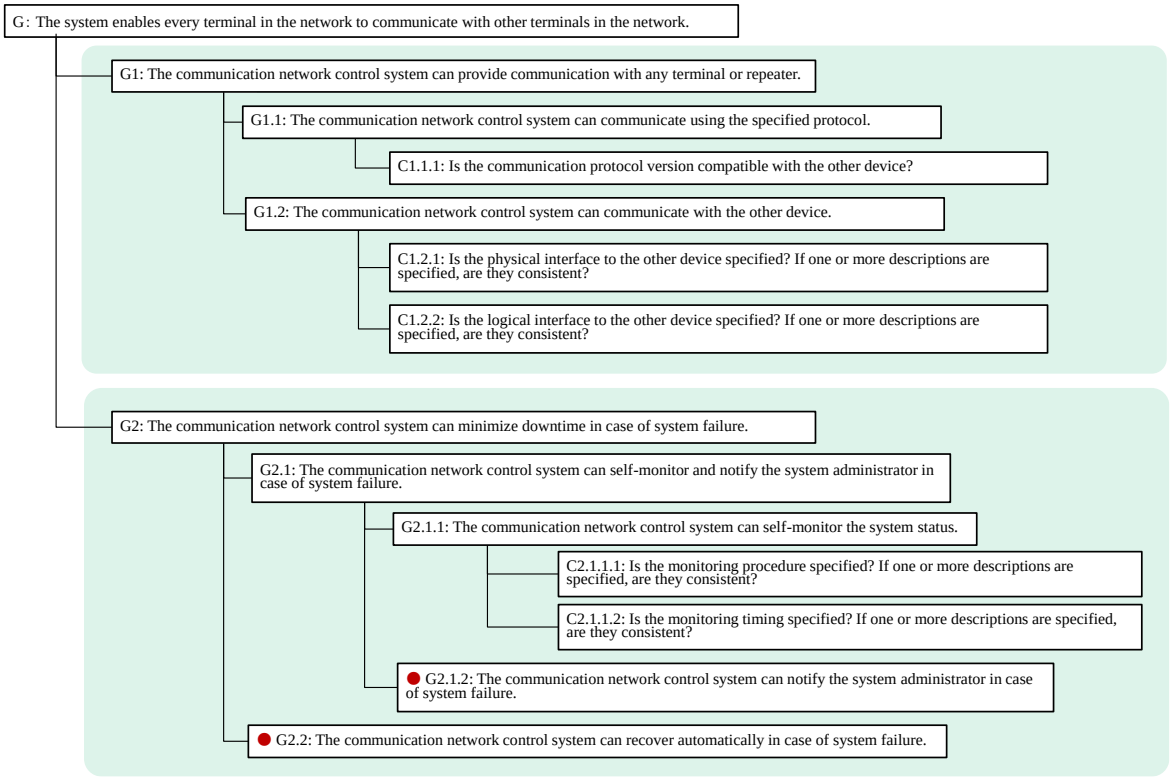

Figure 4.5 The goal tree and goal-oriented check items for System Sc-A.

further because these subgoals were realized by the reused software whose reliability was already proven in another system in operation with the same design contexts.

4.5.2 Results of Evaluation 2

As shown in Table 4.4, the value of grD was six for subsystem 1a and eighteen for subsystem 2a. For subsystem 1a, the sum of cH and grH was 8.3 and the value of gwH was 42.0. The sum of cH and grH was 19.6% of the value of gwH. For subsystem 2a, the sum of cH and grH was 9.0 and the value of gwH was 54.0. The sum of cH and grH was 16.7% of the value of gwH.

Table 4.4 Results for Evaluation 2.

Subsystem	cH	grH	gwH	grD	gtD
1a	1.0	7.3	42.0	6	0
2a	1.0	8.0	54.0	18	0

Table 4.5 Number of defects for goal-oriented check items for Subsystem 1a.

	Goal-oriented check item	grD	gtD
C1.1.1	Is the communication protocol version compatible with the other device?	4	0
C1.2.1	Is the physical interface to the other device specified? If one or more descriptions are specified, are they consistent?	0	0
C1.2.2	Is the logical interface to the other device specified? If one or more descriptions are specified, are they consistent?	2	0
Total		6	0

Table 4.6 Number of defects for goal-oriented check items for Subsystem 2a.

	Goal-oriented check item	grD	gtD
C2.1.1.1	Is the monitoring method specified? If one or more descriptions are specified, are they consistent?	13	0
C2.1.1.2	Is the monitoring timing specified? If one or more descriptions are specified, are they consistent?	5	0
Total		18	0

Table 4.5 shows the goal-oriented check items and the number of defects categorized as the defect groups which could be detected with the goal-oriented check items for subsystem 1a. For subsystem 1a, the reviewer detected four defects for C1.1.1

Table 4.7 Results for Evaluation 3.

Subsystem	srD	gtD
1b	2	1
2b	5	2

and two defects for C1.2.2. No CDR defect was detected in subsequent software testing for subsystem 1a.

Table 4.6 shows the goal-oriented check items and the number of defects in defect groups corresponding to the goal-oriented check items for subsystem 2a. For subsystem 2a, the reviewer detected thirteen defects for C2.1.1.1 and five defects for C2.1.1.2. No CDR defects were detected in subsequent software testing for subsystem 2a.

4.5.3 Results of Evaluation 3

As shown in Table 4.7, the value of srD was two for subsystem 1b and five for subsystem 2b. Table 4.7 also shows that the value of gtD was one for subsystem 1b and two for subsystem 2b. Table 4.8 shows the goal-oriented check items for subsystem 1b and the number of detected defects for the check items. For subsystem 1b, the reviewer detected two CDR defects for C1.2.1 in the standard design reviews. Subsequent testing detected one CDR defect for C1.1.1.

Table 4.9 shows the goal-oriented check items for subsystem 2b and the number of detected defects for the check items. For subsystem 2b, the reviewer detected two CDR defects for C2.1.1.1 and three CDR defects for C2.1.1.2 in the standard design reviews. Subsequent software testing detected one CDR defect for C2.1.1.1 and one CDR defect for C2.1.1.2.

Table 4.8 Number of defects for goal-oriented check items for Subsystem 1b.

	Goal-oriented check item	srD	gtD
C1.1.1	Is the communication protocol version compatible with the other device?	0	1
C1.2.1	Is the physical interface to the other device specified? If one or more descriptions are specified, are they consistent?	2	0
C1.2.2	Is the logical interface to the other device specified? If one or more descriptions are specified, are they consistent?	0	0
Total		2	1

Table 4.9 Number of defects for goal-oriented check items for Subsystem 2b.

	Goal-oriented check item	srD	gtD
C2.1.1.1	Is the monitoring method specified? If one or more descriptions are specified, are they consistent?	2	1
C2.1.1.2	Is the monitoring timing specified? If one or more descriptions are specified, are they consistent?	3	1
Total		5	2

4.6 Discussion

4.6.1 Evaluation Results

In Evaluation 1, a quality assurance engineer defined both a goal tree and the corresponding goal-oriented check items without additional explanations for System Sc-A. This indicated that the proposed method does not require a domain expert as an

analyst. In discussion, another engineer of the case study said, “Although the quality assurance engineer is not a member of the development team, the engineer could define the goal tree and the corresponding goal-oriented check items. For future development, I suppose that engineers with software quality assurance skills can define them.”

Evaluation 2.1 showed that the reviewers could perform goal-oriented software design reviews and detect CDR defects. In addition, Evaluation 2.2 showed that the estimated additional rework effort for the detected defects in subsequent testing was reduced by the defects detected in the goal-oriented software design reviews. The case study results showed that the CDR defects were detected by goal-oriented software design reviews in both subsystems 1a and 2a. In subsequent activities, including testing, releasing, operating, and maintenance, no CDR defect was detected.

Evaluation 3 suggests that non-expert reviewers can detect CDR defects in goal-oriented software design reviews. For example, in subsystem 2b, a defect was detected in the design review: “The definition of a port-level monitoring method for a certain device is omitted.” The defect could have been detected by goal-oriented check item C2.1.1.1: “Is the monitoring procedure specified? If one or more descriptions are specified, are they consistent?” Thus, even if the reviewers are not experts in the system, they might have noticed an omission in the definition of the monitoring procedure.

The results of the case study suggest that the proposed method can potentially detect CDR defects. In the case study, CDR defects were detected with goal-oriented check items. An example of a detected CDR defect is “Some devices could not communicate with other devices because of incompatible communication protocol versions.” This defect was detected with the goal-oriented check item C1.1.1: “Is the communication protocol version compatible between the devices?” In the requirement definition activity, the requirement explicitly specified the communication protocol name but did not specify the version of the communication protocol.

Sharing a goal tree before goal-oriented software design reviews can reduce the effort for goal-oriented software design reviews. In a discussion with an engineer of the case study, the engineer pointed out that sharing and discussing a goal tree in advance facilitates understanding the corresponding goal-oriented check items and prevents engineers from misunderstanding the specification. He also mentioned that sharing and discussing a goal tree ensure that all reviewers reach a consensus on the specifications before starting the design reviews.

4.6.2 Threats to Validity

1) Internal Validity

Defining a goal tree and the corresponding goal-oriented check items may require expert-level skills and knowledge in the domain, and personnel overhead. In the case study, the quality assurance engineer who was responsible for the verification of System Sc-A defined the goal tree and the corresponding goal-oriented check items. The engineer had general quality-assurance skills but did not have system-specific skills and knowledge and was not a member of the development team. Thus, an engineer with general skills and knowledge of software development and the target system can define a goal tree and the corresponding goal-oriented check items. In addition, the case study showed that the personnel overhead for the proposed method would be small. In the case study, the engineer took 1 hour to identify the goal tree and define the goal-oriented check items for each subsystem 1a and 2a.

If the goal-oriented check items and the check items in the checklist defined in the standard software development process of Company *Sc* overlap, the effectiveness of goal-oriented software design reviews will be insufficient. If each goal-oriented check item is included in the standard checklist, reviewers can detect all CDR defects with the standard checklist in standard design reviews. In this case study, the standard design review checklist did not include any goal-oriented check items because the standard design review check items were more general and comprehensive; they were intended

for use in the development of various software in the communication network domain, whereas the goal-oriented check items were system-specific.

2) External Validity

If the target system has various goals, such as in the case of a customer relationship management (CRM) system or enterprise resource planning (ERP) system, the effectiveness of the proposed method might be limited. In this case study, the top goal could be easily identified and defined because the communication system had a simple goal tree. By contrast, the goal tree may be more complex in other systems such as CRM and ERP systems. However, goal-oriented requirements analysis methods are not limited by the types of systems. Once a goal tree has been defined, goal-oriented software design reviews can be performed.

A larger number of subgoals may require a larger effort to define goal-oriented check items and perform goal-oriented software design reviews. An engineer who participated in the case study stated that the subgoals needed to be prioritized in case of a larger number of subgoals. Although the proposed method does not consider the priorities of subgoals, the proposed method can easily incorporate subgoal priority via decision-making methods such as an analytic hierarchy process (AHP) [110].

In an iterative development process including agile development process [71], [72], [111], [112], design and source code are updated in each iteration. Thus, CDR defects are potentially injected in each iteration. Although design reviews might not be explicitly performed in some iterative development processes, CDR defects are detected in activities such as architectural discussion, testing, and implementation of the test and product codes. The rework effort can be reduced if CDR defects are detected using the essence of the proposed method for architectural discussion, testing, and implementation of the test and product codes. For example, potential CDR defects can be identified in end-of-iteration reviews, one of the recommended practices for constant feedback on technical decisions and customer requirements in agile development process [71], [113].

Specifically, when a context-dependent requirement is implemented in two or more iterations, the implementation can be inconsistent in the iterations. The potential inconsistencies (potential CDR defects) in subsequent iterations can be identified in the end-of-iteration review of the first iteration, in which the context-dependent requirement is implemented. Further work is required to establish the viability of the proposed method to iterative development processes including agile development process.

To generalize the results of the case study, further evaluations in other systems are needed. Because of the limited analysis effort, I carried out lightweight evaluations for other systems developed in Company *Sc*. The results of the lightweight evaluations showed that context-dependent ambiguous requirements injected CDR defects and that the CDR defects were overlooked in design reviews and detected in subsequent testing. The CDR defects include inconsistencies among the unit of distance, the notations of time, and the significant digits of numbers. In the lightweight evaluation, I identified the subgoal “The speed of the moving object can be calculated from the distance moved and the elapsed time.” I also defined the goal-oriented check item “Are the measurement methods of the distance moved and the elapsed time correct?” These results imply that the proposed method can be applied to other systems.

4.7 Conclusions

This section proposed a method to detect CDR defects by design reviews using a goal tree created via goal-oriented requirements analysis based on the analysis on the case study of the simulation control software system. CDR defects are caused by inconsistencies among design implementations, which are supported by the same requirement (the same subgoal). First, the proposed method creates a goal tree of the target software via goal-oriented requirements analysis. Second, the proposed method defines goal-oriented check items to detect inconsistencies among implementations that

realize the same requirement and examine whether the goal and subgoals are satisfied. Third, reviewers perform goal-oriented software design reviews with the goal-oriented check items.

To evaluate the effectiveness of the proposed method, I conducted a case study. The case study evaluated whether the goal-oriented check items could detect CDR defects. The case study also evaluated whether the effort to create a goal tree, define the goal-oriented check items, and perform goal-oriented software design reviews was smaller than the estimated rework effort if the detected defects were overlooked in design review and corrected in subsequent testing. The estimated saved rework effort was calculated as the difference between the sum of the estimated effort for investigating, fixing, and regression testing and the sum of the effort for fixing defects detected in goal-oriented software design reviews assuming that the defect was overlooked in the goal-oriented software design review and detected in subsequent testing. The results of the case study showed that the proposed method detected CDR defects and that other CDR defects were not detected in subsequent testing. The results also showed that the estimated savings in additional rework effort for defects detected by the proposed method was larger than the sum of the effort for preparing and performing the proposed method. Furthermore, the case study investigated whether CDR defects were detected by design reviews without the proposed method and subsequent testing in other subsystems sharing the same goal tree of the target subsystems. The results showed that CDR defects were detected in the other subsystems.

5. Conclusions

Software review is a visual software-artifact evaluation technique to detect anomalies, defects, errors, or deviations from specifications or standards; however, the software review cannot always provide the expected effect. This thesis focuses on two issues preventing software reviews from providing the expected effect. The first issue is low review quality. This results in that software review materials include overlooked defects because software reviews could not detect defects sufficiently. The second issue is context-dependent defects which could not be considered as defects in the review time and turn out to be defects in the subsequent software development activities. For example, some requirements are omitted or ambiguous depending on the design context, although these requirements would not necessarily be omitted or ambiguous when viewed as requirements alone.

For the first issue (low review quality issue), this thesis proposed a new metric to assess whether software reviews were performed properly. Reviewers can evaluate the software review quality more precisely by the proposed metric in addition to the existing common software review metrics. Previous studies reported that reviewers asked questions and engaged in discussions during software reviews and that the concerns identified by the questions and discussions helped detect defects. Although such concerns about potential defects lead to finding defects, review metrics such as the number of defects detected do not always reflect the questions and discussions because concerns which are not applicable to the software review material are excluded from the number of defects. This thesis proposed a metric, the number of questions and discussions, which identifies concerns in software reviews. First, I defined an effective question, which identifies concerns. Then, I defined detailed software review processes (identifying, sharing, and recording processes), which capture how concerns identified

by effective questions are shared and defects are documented. I conducted a case study with 25 projects in industry to investigate the impact of the number of effective questions, which identified concerns, on the number of detected defects in subsequent testing. The results of a multiple regression analysis showed that the number of effective questions predicted the number of defects in subsequent testing at the significance level of 0.05.

For the second issue, this thesis conducted a case study to analyze context-dependent defects. Specifically, I analyzed defects that required significant correction effort in a simulation control software system. The context-dependent defects were ambiguity defects injected by misunderstandings and inconsistencies among stakeholders during interpreting requirements and specifying design documents. The ambiguities of the specifications were found in the definitions of distance, time (time zone), and calculation accuracy. These caused non-conformance in the implementations and errors in the control simulation execution results. Based on the analysis, I proposed a low-effort defect prevention approach defining the units to avoid such ambiguities and estimated the expected effort reduction by the definitions in the target control simulation software development. The results of the evaluation indicated that the proposed approach could achieve 43.5% effort reduction.

Additionally, this thesis proposed a method for detecting inconsistent implementations caused by context-dependent requirement omissions and ambiguities in design reviews. The proposed method could reduce rework efforts for such omissions or ambiguities in requirements caused by design context. Existing detection and analysis methods did not limit evaluation of software review materials to implementations of context-dependent design. The proposed method defines goal-oriented check items for design review using a goal tree obtained by goal-oriented requirements analysis. Reviewers use the goal-oriented check items to detect inconsistent implementations that realize the same requirement. This thesis also evaluated the proposed method through a case study. The results of the case study showed that the proposed method defined five goal-oriented check items and that reviewers detected 24 context-dependent defects with

goal-oriented check items. The results also showed that the sum of the estimated additional effort to define goal-oriented check items and perform design reviews with goal-oriented check items was 19.6 person-hours. Furthermore, the results showed that an engineer with general skills and knowledge of software development but without system-specific skills and knowledge could define a goal tree and the corresponding goal-oriented check items.

Future works include (semi-)automatic categorization for review comments to categorize effective questions that identify concerns for the proposed metric. Sentiment analysis is widely used in natural language processing research [114], [115]. Recent studies have shown that sentiment analysis can categorize review comments from certain perspectives. For example, the sentiment of a comment (i.e., whether or not a comment is formulated in a positive or negative tone) may relate to comment usefulness [34], a model algorithm founded to identify review comments expressing negative sentiments [116], and the emotionality of the comment reflecting conventional metrics such as typing duration and typing speed [117]. Applying these studies to review comments may categorize effective questions that identify concerns. Future works also include increasing case studies for generalizing the results of the case study for the proposed method.

Acknowledgment

I am profoundly indebted to my adviser, Associate Professor Shuji Morisaki, for his guidance, understanding, generosity, and most importantly, sincerity, during my studies at Nagoya University. His mentorship, uncompromising stance toward research and encouragement throughout this work helped me grow as a researcher and make personal progress. I am also very grateful to Professor Hiroyuki Seki and Professor Yuichi Kaji for their understanding and generosity. I wish to sincerely express my respect and appreciation for their insightful comments and advice on my studies.

I would also like to express my special thanks to the company members I work with for sharing their datasets and some ideas with me. They also provided valuable comments and insights during my studies. Their helpful comments and cooperation were encouraging, and the fulfilling discussions helped me develop new ideas concerning my studies. Outside the work environment, I wish to acknowledge and thank my excellent friends, who have always been cheerful and supportive. Last but not least, I thank my family for their steady and warm support, encouragement, and understanding of me.

References

[1] L. Nderu, “Framework for an effective formal technical review in software quality assurance,” Master of Science in Software Engineering, Jomo Kenyatta University, 2011.

[2] P. Bourque and R. E. Fairley, “Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0,” 2014. [Online]. Available: http://www.swebok.org/

[3] S. Thomke and T. Fujimoto, “Front-loading problem-solving: implications for development performance and capability,” in *PICMET '99: Portland International Conference on Management of Engineering and Technology*, Portland, OR, USA, 1999.

[4] J. D. Blackburn, G. Hoedemaker, and L. N. Van Wassenhove, “Concurrent software engineering: prospects and pitfalls,” *IEEE Transactions on Engineering Management*, vol. 43, no. 2, pp. 179-188, 1996.

[5] B. W. Boehm, “Software engineering economics,” *IEEE Transactions on Software Engineering*, vol. SE-10, no. 1, pp. 4-21, 1984.

[6] M. E. Fagan, “Design and code inspections to reduce errors in program development,” *IBM Systems Journal*, vol. 15, no. 3, pp. 182-211, 1976.

[7] IEEE standard for software reviews and audits, IEEE Std 1028-2008, 2008.

[8] B. Boehm and V. R. Basili, “Top 10 list [software development],” *IEEE Computer*, vol. 34, no. 1, pp. 135-137, 2001.

[9] B. P. De Souza, R. C. Motta, D. De O. Costa, and G. H. Travassos, “An IoT-based scenario description inspection technique,” in *Proceedings of the XVIII Brazilian Symposium on Software Quality*, Fortaleza, Brazil, 2019, pp. 20-29.

[10] T. Gilb and D. Graham, *Software inspection*. Boston, Masachusetts, USA: Addison-Wesley Reading, 1993.

[11] M. Ciolkowski, O. Laitenberger, and S. Biffl, “Software reviews: The state of the practice,” *IEEE Software*, vol. 20, no. 6, pp. 46-51, 2003.

[12] J. Tian, *Software quality engineering: testing, quality assurance, and quantifiable improvement*. Hoboken, NJ, USA: John Wiley & Sons, 2005.

[13] D. L. Parnas and M. Lawford, "The role of inspection in software quality assurance," *IEEE Transactions on Software Engineering*, vol. 29, no. 8, pp. 674-676, 2003.

[14] A. S. Olalekan and A. Osofisan, "Empirical study of factors affecting the effectiveness of software inspection: A preliminary report," *European Journal of Scientific Research*, vol. 19, pp. 614-627, 2008.

[15] V. Suma and T. R. G. Nair, "Four-step approach model of inspection (FAMI) for effective defect management in software development," *InterJRI Science and Technology*, vol. 3, no. 1, pp. 29-41, 2012.

[16] A. A. Porter and P. M. Johnson, "Assessing software review meetings: Results of a comparative analysis of two experimental studies," *IEEE Transactions on Software Engineering*, vol. 23, no. 3, pp. 129-145, 1997.

[17] A. M. Davis, *201 principles of software development*. Washington, DC, USA: IEEE Computer Society, 1995.

[18] V. R. Basili *et al.*, "The empirical investigation of perspective-based reading," *Empirical Software Engineering*, vol. 1, no. 2, pp. 133-164, 1996.

[19] A. Porter and L. Votta, "Comparing detection methods for software requirements inspections: A replication using professional subjects," *Empirical Software Engineering*, vol. 3, no. 4, pp. 355-379, 1998.

[20] A. A. Porter, L. G. Votta, and V. R. Basili, "Comparing detection methods for software requirements inspections: A replicated experiment," *IEEE Transactions on Software Engineering*, vol. 21, no. 6, pp. 563-575, 1995.

[21] F. Shull, I. Rus, and V. Basili, "How perspective-based reading can improve requirements inspections," *IEEE Computer*, vol. 33, no. 7, pp. 73-79, 2000.

[22] T. Thelin, P. Runeson, and B. Regnell, "Usage-based reading—an experiment to guide reviewers with use cases," *Information and Software Technology*, vol. 43, no. 15, pp. 925-938, 2001.

[23] T. Thelin, P. Runeson, and C. Wohlin, "An experimental comparison of usage-based and checklist-based reading," *IEEE Transactions on Software Engineering*, vol. 29, no. 8, pp. 687-704, 2003.

[24] S. A. Ebad, “Inspection reading techniques applied to software artifacts-a systematic review,” *Computer Systems Science and Engineering*, vol. 32, no. 3, pp. 213-226, 2017.

[25] B. P. De Souza, R. C. Motta, and G. H. Travassos, “The first version of SCENARIotCHECK,” in *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, Salvador, Brazil, 2019.

[26] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili, “Detecting defects in object-oriented designs: using reading techniques to increase software quality,” *ACM SIGPLAN Notices*, vol. 34, no. 10, pp. 47-56, 1999.

[27] O. Laitenberger, “Cost-effective detection of software defects through perspective-based Inspections,” *Empirical Software Engineering*, vol. 6, no. 1, pp. 81-84, 2001.

[28] F. Shull, “Developing techniques for using software documents: A series of empirical studies,” Ph.D. thesis, in *Department of Computer Science*, Univ. of Maryland, College Park, MD, USA, 1998.

[29] T. Gyimóthy, R. Ferenc, and I. Siket, “Empirical validation of object-oriented metrics on open source software for fault prediction,” *IEEE Transactions on Software engineering*, vol. 31, no. 10, pp. 897-910, 2005.

[30] S. G. Eickt, C. R. Loader, M. D. Long, L. G. Votta, and S. V. Wiel, “Estimating software fault content before coding,” in *International Conference on Software Engineering*, Melbourne, Australia, 1992, pp. 59-65.

[31] P. Runeson and C. Wohlin, “An experimental evaluation of an experience-based capture-recapture method in software code inspections,” *Empirical Software Engineering*, vol. 3, no. 4, pp. 381-406, 1998.

[32] L. C. Briand, K. El Emam, B. Frelmut, and O. Laitenberger, “Quantitative evaluation of capture-recapture models to control software inspections,” in *Proceedings The Eighth International Symposium on Software Reliability Engineering*, Albuquerque, NM, USA, 1997, pp. 234-244.

[33] NASA, *Mars Climate Orbiter mishap investigation board phase I report*. Washington, DC, USA, 1999.

[34] A. Bosu, M. Greiler, and C. Bird, “Characteristics of useful code reviews: An empirical study at Microsoft,” in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, Florence, Italy, 2015, pp. 146-156.

[35] F. Ebert, F. Castor, N. Novielli, and A. Serebrenik, “Communicative intention in code review questions,” in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Madrid, Spain, 2018, pp. 519-523.

[36] G. Huet, S. J. Culley, C. A. McMahon, and C. Fortin, “Making sense of engineering design review activities,” *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, vol. 21, no. 3, pp. 243-266, 2007.

[37] P. N. Robillard, P. d’Astous, F. Détienne, and W. Visser, “An empirical method based on protocol analysis to analyze technical review meetings,” in *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*, Toronto, Ontario, Canada, 1998, pp. 1-12.

[38] P. d'Astous, F. Détienne, W. Visser, and P. Robillard, “On the use of functional and interactional approaches for the analysis of technical review meetings,” in *Proceedings of the 12th Annual Workshop of the Psychology of Programming Interest Group*, Cosenza, Italy, 2000, pp. 155-170.

[39] M. Hasan, A. Iqbal, M. R. U. Islam, A. J. M. I. Rahman, and A. Bosu, “Using a balanced scorecard to identify opportunities to improve code review effectiveness: an industrial experience report,” *Empirical Software Engineering*, vol. 26, no. 6, pp. 129-163, 2021.

[40] N. H. Taba and S. H. Ow, “A web-based model for inspection inconsistencies resolution: a new approach with two case studies,” *Malaysian Journal of Computer Science*, vol. 32, no. 1, pp. 1-17, 2019.

[41] M. M. Rahman, C. K. Roy, and R. G. Kula, “Predicting usefulness of code review comments using textual features and developer experience,” in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, Buenos Aires, Argentina, 2017, pp. 215-226.

[42] L. G. Votta, “Does every inspection need a meeting?” *ACM SIGSOFT Software Engineering Notes*, vol. 18, no. 5, pp. 107-114, 1993.

[43] P. Murphy and J. Miller, “A process for asynchronous software inspection,” in *Proceedings Eighth IEEE International Workshop on Software Technology and Engineering Practice incorporating Computer Aided Software Engineering*, London, UK, 1997, pp. 96-104.

[44] O. Laitenberger and J.-M. DeBaud, “An encompassing life cycle centric survey of software inspection,” *Journal of Systems and Software*, vol. 50, no. 1, pp. 5-31, 2000.

[45] P. M. Johnson and D. Tjahjono, “Does every inspection really need a meeting?,” *Empirical Software Engineering*, vol. 3, no. 1, pp. 9-35, 1998.

[46] Y. Yu, H. Wang, G. Yin, and T. Wang, “Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment?,” *Information and Software Technology*, vol. 74, pp. 204-218, 2016.

[47] P. Thongtanunam and A. E. Hassan, “Review dynamics and their impact on software quality,” *IEEE Transactions on Software Engineering*, pp. 2698-2712, 2020.

[48] R. Chillarege *et al.*, “Orthogonal defect classification-a concept for in-process measurements,” *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 943-956, 1992.

[49] IBM, *Orthogonal defect classification v 5.2 for software design and code*. Armonk, NY, USA, 2013.

[50] M. V. Mantyla and C. Lassenius, “What types of defects are really discovered in code reviews?” *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 430-448, 2009.

[51] A. Fernandez, S. Abrahao, and E. Insfran, “Empirical validation of a usability inspection method for model-driven Web development,” *Journal of Systems and Software*, vol. 86, no. 1, pp. 161-186, 2013.

[52] B. Regnell, P. Runeson, and T. Thelin, “Are the perspectives really different?–further experimentation on scenario-based reading of requirements,” *Empirical Software Engineering*, vol. 5, no. 4, pp. 331-356, 2000.

[53] M. Ciolkowski, O. Laitenberger, and S. Biffl, “Software reviews, the state of the practice,” *IEEE Software*, vol. 20, no. 6, pp. 46-51, 2003.

[54] F. Lanubile and T. Mallardo, “An empirical study of web-based inspection meetings,” in *2003 International Symposium on Empirical Software Engineering*, Rome, Italy, 2003, pp. 244-251.

[55] F. Calefato, F. Lanubile, and T. Mallardo, “A controlled experiment on the effects of synchronicity in remote inspection meetings,” in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, Madrid, Spain, 2007, pp. 473-475.

[56] A. Porter, H. Siy, A. Mockus, and L. Votta, "Understanding the sources of variation in software inspections," *ACM Transactions on Software Engineering and Methodology*, vol. 7, no. 1, pp. 41-79, 1998.

[57] F. Macdonald and J. Miller, "A comparison of tool-based and paper-based software inspection," *Empirical Software Engineering*, vol. 3, no. 3, pp. 233-253, 1998.

[58] T. Thelin, P. Runeson, C. Wohlin, T. Olsson, and C. Andersson, "Evaluation of usage-based reading—conclusions after three experiments," *Empirical Software Engineering*, vol. 9, pp. 77-110, 2004.

[59] L. P. W. Land, B. Tan, and L. Bin, "Investigating training effects on software reviews: a controlled experiment," in *2005 International Symposium on Empirical Software Engineering*, Noosa Heads, Australia, 2005, pp. 356-366.

[60] G. Sabaliauskaite, S. Kusumoto, and K. Inoue, "Assessing defect detection performance of interacting teams in object-oriented design inspection," *Information and Software Technology*, vol. 46, no. 13, pp. 875-886, 2004.

[61] L. Briand, D. Falessi, S. Nejati, M. Sabetzadeh, and T. Yue, "Traceability and SysML design slices to support safety inspections," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 1, pp. 1-43, 2014.

[62] Y. Wong and D. Wilson, "An empirical investigation of the important relationship between software review meetings process and outcomes," in *IASTED International Conference on Software Engineering*, Innsbruck, Austria, 2004, pp. 422-427.

[63] B. Soltanifar, A. Erdem, and A. Bener, "Predicting defectiveness of software patches," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, Ciudad Real, Spain, 2016, pp. 1-10.

[64] J. Carver, F. Shull, and V. Basili, "Can observational techniques help novices overcome the software inspection learning curve? An empirical investigation," *Empirical Software Engineering*, vol. 11, no. 4, pp. 523-539, 2006.

[65] L. P. W. Land, "Software group reviews and the impact of procedural roles on defect detection performance," *Empirical Software Engineering*, vol. 7, no. 1, pp. 77-79, 2002.

[66] K. Sandahl, O. Blomkvist, J. Karlsson, C. Krysander, M. Lindvall, and N. Ohlsson, "An extended replication of an experiment for assessing methods for software

requirements inspections," *Empirical Software Engineering*, vol. 3, no. 4, pp. 327-354, 1998.

[67] Ö. Albayrak and J. C. Carver, "Investigation of individual factors impacting the effectiveness of requirements inspections: a replicated experiment," *Empirical Software Engineering*, vol. 19, no. 1, pp. 241-266, 2014.

[68] Software Engineering Institute, Carnegie Mellon University, CMMI for development, version 1.3, 2010.

[69] M. E. Fagan, "Advances in software inspections," *IEEE Transactions on Software Engineering*, no. 7, pp. 744-751, 1986.

[70] VDA QMC Working Group 13, Automotive SIG, Automotive SPICE process assessment and reference model version 3.1, 2017.

[71] J. Highsmith and A. Cockburn, "Agile software development: the business of innovation," *Computer*, vol. 34, no. 9, pp. 120-127, 2001.

[72] P. Abrahamsson, J. Warsta, M. T. Siponen, and J. Ronkainen, "New directions on agile methods: a comparative analysis," in *25th International Conference on Software Engineering, 2003*, Portland, OR, USA, 2003.

[73] K. Lee and B. Boehm, "Empirical results from an experiment on value-based review (VBR) processes," in *2005 International Symposium on Empirical Software Engineering*, Noosa Heads, QLD, Australia, 2005.

[74] F. Shull *et al.*, "Replicating software engineering experiments: Addressing the tacit knowledge problem," in *Proceedings International Symposium on Empirical Software Engineering*, 2002, pp. 7-16.

[75] A. Dardenne, A. Van Lamsweerde, and S. Fickas, "Goal-directed requirements acquisition," *Science of computer programming*, vol. 20, no. 1-2, pp. 3-50, 1993.

[76] D. M. Berry and E. Kamsties, "Ambiguity in requirements specification." in *Perspectives on software requirements*: Kluwer Academic Publishers, 2004, pp. 7-44.

[77] B. Dhanalaxmi, G. A. Naidu, and K. Anuradha, "A fault prediction approach based on the probabilistic model for improvising software inspection," *Indian Journal of Science and Technology*, vol. 9, no. 45, 2016.

[78] A. Van Lamsweerde and E. Letier, “Integrating obstacles in goal-driven requirements engineering,” in *Proceedings of the 20th international conference on Software engineering*, Kyoto, Japan, 1998, pp. 53-62.

[79] A. Van Lamsweerde, “Goal-oriented requirements engineering: A guided tour,” in *Proceedings Fifth IEEE International Symposium on Requirements Engineering*, Toronto, ON, Canada, 2001, pp. 249-262.

[80] A. Lapouchnian, *Goal-oriented requirements engineering: An overview of the current research*. Toronto, Ontario, Canada, 2005.

[81] A. I. Anton, “Goal-based requirements analysis,” in *Proceedings of the Second International Conference on Requirements Engineering*, Colorado Springs, CO, USA, 1996, pp. 136-144.

[82] A. Van Lamsweerde, *Requirements engineering: from system goals to UML models to software specifications*. New York, NY, USA: Wiley, 2009.

[83] A. Van Lamsweerde, “Elaborating security requirements by construction of intentional anti-models,” in *26th International Conference on Software Engineering*, Edinburgh, UK, 2004, pp. 148-157.

[84] A. Van Lamsweerde, R. Darimont, and P. Massonet, “Goal-directed elaboration of requirements for a meeting scheduler: Problems and lessons learnt,” in *Proceedings of 1995 IEEE International Symposium on Requirements Engineering*, York, England, 1995, pp. 194-203.

[85] S. Tueno, R. Laleau, A. Mammar, and M. Frappier, “Integrating domain modeling within a formal requirements engineering method.” in *Implicit and Explicit Semantics Integration in Proof-Based Developments of Discrete Systems*: Springer Singapore, 2021, pp. 39-58.

[86] C. Kartiko, A. C. Wardhana, and W. A. Saputra, “Requirements engineering of village innovation application using goal-oriented requirements engineering (GORE),” *Jurnal Infotel*, vol. 13, no. 2, pp. 38-46, 2021.

[87] X. Franch, L. López, C. Cares, and D. Colomer, “The i* framework for goal-oriented modeling.” in *Domain-specific conceptual modeling*. New York, NY, USA: Springer, 2016, pp. 485-506.

[88] E. S. Yu, “Towards modelling and reasoning support for early-phase requirements engineering,” in *Proceedings of ISRE '97: 3rd IEEE International Symposium on Requirements Engineering*, Annapolis, MD, USA, 1997, pp. 226-235.

[89] E. S. Yu, "Modeling organizations for information systems requirements engineering," in *[1993] Proceedings of the IEEE International Symposium on Requirements Engineering*, San Diego, CA, USA, 1993.

[90] Y. Wang, T. Li, Q. Zhou, and J. Du, "Toward practical adoption of i* framework: An automatic two-level layout approach," *Requirements Engineering*, vol. 26, no. 3, pp. 301-323, 2021.

[91] A. S. Vingerhoets, S. Heng, and Y. Wautelet, "Using i* and UML for blockchain oriented software engineering: Strengths, weaknesses, lacks and complementarity," *Complex Systems Informatics and Modeling Quarterly*, no. 26, pp. 26-45, 2021.

[92] J. Mylopoulos, L. Chung, and B. Nixon, "Representing and using nonfunctional requirements: A process-oriented approach," *IEEE Transactions on software engineering*, vol. 18, no. 6, pp. 483-497, 1992.

[93] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos, *Non-functional requirements in software engineering*. Berlin, Germany: Springer Science & Business Media, 2012.

[94] J. Mylopoulos, L. Chung, and E. Yu, "From object-oriented to goal-oriented requirements analysis," *Communications of the ACM*, vol. 42, no. 1, pp. 31-37, 1999.

[95] S. S. Paradkar, "A framework for modeling non-functional requirements for business-critical systems," *International Journal of Innovative Research in Computer Science & Technology*, vol. 9, no. 1, pp. 15-19, 2021.

[96] J. Cleland-Huang, O. Gotel, and A. Zisman, *Software and systems traceability*. New York, NY, USA: Springer, 2012.

[97] O. C. Z. Gotel and C. W. Finkelstein, "An analysis of the requirements traceability problem," in *Proceedings of IEEE International Conference on Requirements Engineering*, Colorado Springs, CO, USA, 1994.

[98] G. Spanoudakis and A. Zisman, "Software traceability: A roadmap." in *Handbook Of Software Engineering And Knowledge Engineering*. Singapore: World Scientific, 2005, pp. 395-428.

[99] R. Torkar, T. Gorschek, R. Feldt, M. Svahnberg, U. A. Raja, and K. Kamran, "Requirements traceability: A systematic review and industry case study," *International Journal of Software Engineering and Knowledge Engineering*, vol. 22, no. 03, pp. 385-433, 2012.

[100] S. Nair, J. L. De La Vara, and S. Sen, “A review of traceability research at the requirements engineering conference,” in *2013 21st IEEE International Requirements Engineering Conference (RE)*, Rio de Janeiro, Brazil, 2013.

[101] J. Lin, Y. Liu, Q. Zeng, M. Jiang, and J. Cleland-Huang, “Traceability transformed: generating more accurate links with pre-trained BERT models,” in *ICSE '21: Proceedings of the 43rd International Conference on Software Engineering*, Madrid, Spain, 2021.

[102] A. Zisman and A. Kozlenkov, “Knowledge base approach to consistency management of UML specifications,” in *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, San Diego, CA, USA, 2001.

[103] A. Kozlenkov and A. Zisman, “Are their design specifications consistent with our requirements?,” in *Proceedings IEEE Joint International Conference on Requirements Engineering*, Essen, Germany, 2002.

[104] J. Hassine, J. Rilling, and J. Hewitt, “Change impact analysis for requirement evolution using use case maps,” in *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, Lisbon, Portugal, 2005.

[105] S. Lehnert, *A review of software change impact analysis*. Ilmenau, Germany: Ilmenau University of Technology, 2011.

[106] T. Jalaja, T. Adilakshmi, and P. Abhishek, “Automation of change impact analysis for Python applications.” in *Smart Computing Techniques and Applications*. New York, NY, USA: Springer, 2021, pp. 259-267.

[107] S. B. Robert Arnold, *Software change impact analysis*. Los Alamitos, CA, USA: Wiley-IEEE Computer Society Publications Tutorial Series, 1996.

[108] Bohner, “Impact analysis in the software change process: A year 2000 perspective,” in *Proceedings of International Conference on Software Maintenance ICSM-96*, Monterey, CA, USA, 1996.

[109] B. G. Ryder and F. Tip, “Change impact analysis for object-oriented programs,” in *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, Snowbird, Utah, USA, 2001, pp. 46–53.

[110] T. L. Saaty, *The analytic hierarchy process: Planning setting priorities, resource allocation*. New York, NY, USA: McGraw-Hill, 1980.

[111] A. Lopez Lorca, R. Burrows, and L. Sterling, "Teaching motivational models in agile requirements engineering," in *2018 IEEE 8th International Workshop on Requirements Engineering Education and Training (REET)*, Banff, AB, Canada, 2018.

[112] N. Potter and M. Sakry, "Implementing SCRUM (agile) and CMMI together," *The Process Group-Post newsletter*, vol. 16, no. 2, pp. 1-6, 2009.

[113] E. Rubin and H. Rubin, "Supporting agile software development through active documentation," *Requirements Engineering*, vol. 16, no. 2, pp. 117-132, 2010.

[114] A. Agarwal, B. Xie, I. Vovsha, O. Rambow, and R. J. Passonneau, "Sentiment analysis of Twitter data," in *Proceedings of the workshop on language in social media (LSM 2011)*, Portland, OR, USA, 2011, pp. 30-38.

[115] R. Feldman, "Techniques and applications for sentiment analysis," *Communications of the ACM*, vol. 56, no. 4, pp. 82-89, 2013.

[116] T. Ahmed, A. Bosu, A. Iqbal, and S. Rahimi, "SentiCR: A customized sentiment analysis tool for code review interactions," in *ASE 2017: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, Urbana, IL, USA, 2017, pp. 106-111.

[117] H. Vrzakova, A. Begel, L. Mehtätalo, and R. Bednarik, "Affect recognition in code review: An in-situ biometric study of reviewer's affect," *Journal of Systems and Software*, vol. 159, no. 110434, pp. 1-12, 2020.

List of Publications

Research Achievements Related To The Dissertation

I. Journal Papers

1 M. Wakimoto and S. Morisaki, “Goal-oriented software design reviews,” IEEE Access, vol. 10, pp. 32584-32594, 2022.

2 M. Wakimoto and S. Morisaki, “A metric for questions and discussions identifying concerns in software reviews,” Software, vol. 1, no. 3, pp. 364-380, 2022.

II. International Conference

1 M. Wakimoto, S. Morisaki, and S. Yamamoto, “A case study of requirements ambiguities and goal-oriented focused requirements specification,” 8th International Congress on Advanced Applied Informatics (IIAI-AAI 2019), pp. 908-913.