Checklist-Based Intelligent Human-Machine Pair Inspection

Yujun Dai[1,a)] Shaoying Liu[1,b)]

Abstract: With the rapid development of computer technology, the scale of the software system is increasing, the quality of the software is getting more attention. Software inspection is a widely known practice of software quality assurance. However, most of the existing software inspection are handed over to other reviewers for inspection after the program is completed. In this paper, we study a process of software construction that machine can automatically and dynamically comprehend program and guide programmers to inspect the code in the current version of software as it is being constructed. The aim of the method is to use cyclomatic complexity to determine which function or method may have the most defects, and then use static analysis technology to extract its intermediate representation, and automatically generate the inspection syntax tree and the corresponding questions in checklist for programmer to inspect. We present an example to show how the method can be used and prove its accuracy and usefulness. The result shows that our method can be automatic and effective in assisting programmers to detect defects during programming.

1. Introduction

As the rapid development of computer and Internet technology, computer has been applied to all sectors of the society, and the software is a very important part of the computer system. At present, the scale of the software system is increasing, the software development technology is becoming more complex, the quality of the software is getting more attention. Especially now computer systems have been widely used in many safety-related systems related to national economy and people's livelihood, such as high-speed train control system, aerospace control system, nuclear reactor control systems, medical equipment systems, etc. Any errors in these systems can lead to catastrophic consequences. We have been longing for a process of software construction that can timely comprehend program and indicate faults in the current version of software as it is being constructed, simply because it can significantly improve software quality, reduce construction cost, and enhance software productivity.

Software inspection is a static analysis technique that has widespread practical use for identifying defects, assuring and improving reliability and quality of software [1–3]. In different publications, the term software inspection has a number of synonyms, including code review, code inspection, formal inspection, etc.The goal of software inspection is to look for defects or improvement opportunities without the software execution and before the product delivery, thus reducing the costs of fixing them later [4].Fagan pointed out in his theoretical work [5] that formal design and code inspections and face-to-face meetings reduced the number of errors detected during the testing phase in small development teams.However, this early form of software inspection mandate a number of formal requirements that do not adapt well to agile development methods, most notably a fixed, formal, waterfall process. Due to the adoption of agile methods and distributed software development, code inspection are currently done in a less formal way than in the past, reducing the inefficiencies of its early form. The lightweight variant of code inspection has been referred to as modern code review (MCR) [6], which is a flexible, tool-based, and asynchronous process. The general idea of MCR is that developers (i.e., reviewers) other than the author evaluate code changes to identify defects and quality issues, and decide whether to discard or integrate the changes into the main project repository.Nowadays, it is a common practice to use MCR to improve the quality of code in many development projects.But understanding code changes, their purpose, and motivations has always been the main challenge faced by developers(reviewers) when reviewing code changes [7]. Obviously, the author of the code comprehends his or her code most clearly. Therefore, adopting intelligent Human-Machine Pair Programming (HMPP) [8] technology, there is an opportunity to solve the above problems.HMPP means that a programmer and computer work together to construct a program, where the programmer plays the role of driver and the computer plays the role of observer. In other words, as the software is constructed by a human developer, its current version is always being observed and checked by a software tool, and faults are reported timely if any. The targeted faults are primarily semantic faults, not syntactical faults which can be found by a complier.

In this paper, we describe an approach to using cyclomatic complexity and static analysis technology to improve the process and quality of software inspection. We present a new checklist-

[1] Graduate School of Advanced Science and Engineering, Hiroshima University
[a)] d201609@hiroshima-u.ac.jp
[b)] sliu@hiroshima-u.ac.jp

© 2021 Information Processing Society of Japan

based human-machine pair inspection(HMPI) method which is intended to mitigate the above-mentioned deficiencies of existing inspection techniques, to increase the benefits of machine, that is, tool-based self-code-inspection through the automatic guidance by the machine, and to support HMPP. The aim of the method is to automatically and dynamically observing and verifying the process of software construction, thereby guiding the programmer to inspect code in the current version of the software to handle implementation-related bugs.The method includes six steps:

- Record the code being constructed by developer in real time. Preprocess the recorded data, once the number of lines of code other than blank lines and comment lines reaches 20 lines, perform static analysis.
- Generate the Intermediate Representation(IR) of these 20-lines-codes in the form of 3 Address Code(3AC).
- Calculate the cyclomatic complexity of each function.
- Generate the inspection syntax graph according to the IR of the codes.
- Establish the checklist according to each component in the graph.
- Analyze the corresponding code for each question to detect defects.

The remainder of this paper is organized as follows: Section 2 formally presents and discusses the rationale for the proposed HMPI approach, paving the way for the discussion of the inspection process and tool support in later chapters.Section 3 explains the inspection process. Section 4 presents an example to illustrate the process. Finally,Section 5 concludes the paper and identifies future research directions.

2. Principle of Inspection

2.1 Checklist-based Inspection

Inspection in software engineering, refers to peer review of any work product by trained individuals who look for defects using a well-defined process.The goal of inspection is to identify defects, assure and improve reliability and quality of software, enable systematic software process improvement and preventive actions earlier than testing. The process of formal software inspection includes 6 stages:

(1) `Planning:`The plan of inspection is made by the moderator.
(2) `Overview meeting:` The background of the work product is described by the author.
(3) `Preparation:`The work product is examined by each reviewer to identify possible defects.
(4) `Inspection meeting:` During this meeting the reader reads through the work product, part by part and the reviewer point out the defects for every part.
(5) `Rework:` The author makes changes to the work product according to the action plans from the inspection meeting.
(6) `Follow-up:` The changes by the author are checked to make sure everything is correct.

In the preparation phase of formal inspection, the reviewers work alone on the product using the checklists provided. The aim of each reviewers is to find the maximum number of potential defects by answering all the questions in the checklists. The original formal inspection method (Fagan Inspection) [5] included the idea of using checklists in defect finding. The purpose of the checklist is to gather expertise concerning the most common defects to support the inspection. Chorkowski et al. (2003) [9] reported that about half of the respondents in their survey used checklists in peer review. Checklists are a fundamental part of the Inspection process. There are individual checklists for each type of documentation in Inspection. Sometimes individual checker specialist roles are defined by means of a special set of checklist questions. Checklists are built to the following rules [10]:They may contain suggestions for probable defect severity (e.g. Major, minor); the checklist does not need to contain every possible question; a checklist should concentrate on questions which will turn up Major defects. Checklists serve the following purposes:

- `Instruction:` teach reviewers about what is expected.
- `Stimulation:` provokes reviewers to look for more than they otherwise might do.
- Checklists should increase the number of bugs found by a reviewer.
- They define part of what is expected of the inspection process, in ”question form”.

In our method, we mainly focus on the checklist for code files.The author acts as an inspector, and the targeted faults are primarily semantic faults.

2.2 Intermediate Representation(IR)

In order to analyze the semantics of the code and generate appropriate checklists for the author to inspect. We can use static analysis methods, that is, intermediate representation(referred to as IR) to record every piece of information of the code in our file. 3-Address Code(3AC) is the most common form of IR.Note that IR mentioned in the rest of the paper are all in 3AC form. Common 3AC forms includes 3AC has two properties: (1)There is at most one operator on the right side of an instruction. For example,instruction $v_2 = a + b + 3$ includes two operator on the right side of the equation.When converted to 3AC form, it is expressed as: $v_1 = a + b; v_2 = v_1 + 3$.(2)Each 3AC contains at most 3 addresses, where address can be one of the following:name(e.g., a, b); constant(e.g., 3);compiler-generated temporary(e.g., v_1, v_2). Common 3ac forms include: $a = b \ \boldsymbol{bop} \ c$,where $\boldsymbol{bop}$ means binary arithmetic or logical operation;$a = \boldsymbol{uop} \ b$, where $\boldsymbol{uop}$ indicates unary operation (e.g.,minus,negation,casting);*goto L*, it is an unconditional jump, where $\boldsymbol{L}$ represents a label of a program location; if $a \ \boldsymbol{rop} \ b$ *goto L*, it is a conditional jump, where $\boldsymbol{rop}$ stands for relational operator(e.g.,>, <, ==, >=, <=, *etc*.) For example, the do-while loop:”$doi = i + 1; while(a[i] < v)$;”,its corresponding IR is shown as follows:

```
1:  i = i + 1
2:  v1 = a[i]
3:  if v1 < v goto 1
```

Obviously, IR has four characteristics:low-level and closed to machine code; usually language independent; compact and uniform; contains control flow information. Therefore, IR is usually considered as the basis for static analysis. After generating 3ac, it should be converted to a control flow graph(CFG), and then static analysis can be performed on the CFG. Given the 3AC of

© 2021 Information Processing Society of Japan

program P as follows:

```
1:   x = input
2:   y = x − 1
3:   z = x ∗ y
4:   if z < x goto 7
5:   p = x/y
6:   q = p + y
7:   a = q
8:   b = x + a
9:   c = 2a − b
10:  if p == q goto 12
11:  goto 3
12:  return
```

The corresponding CFG can be generated as shown in figure 1:

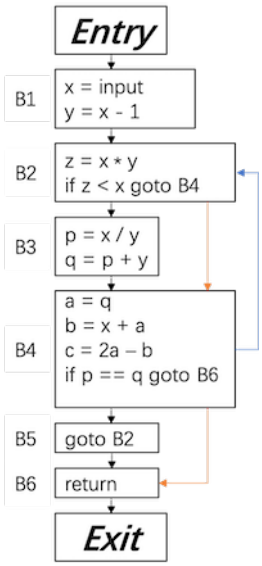

Fig. 1 Control flow graph

The node in CFG can be an individual 3-address instruction, or(usually) a Basic Block(BB). BBs are maximal sequences of consecutive 3-address instructions with the properties as follows: it can be entered only at the beginning,i.e., the first instruction in the block; it can be exited only at the end,i.e., the last instruction in the block.The edge between nodes indicate the order of execution. There is an edge from node x to node y if and only if: there is a conditional or unconditional jump from the end of x to the beginning of y; y immediately follows x in the original order of instructions and x does not end in an unconditional jump.

2.3 Cyclomatic Complexity(CC)

Software complexity is a key property that has been discussed widely in software literature. The complexity of the program is often proportional to the possibility of defects.Functions and methods with the highest complexity tend to also contain the most defects. Thus, reviewers should give priority to inspecting the code with higher software complexity. McCabe's Cyclomatic Complexity [11] is static code attributes that is commonly used for measuring code complexity [12]. It is a quantitative measure of the number of linearly independent paths through a program's source code. In basis path testing, the number of test cases will equal the cyclomatic complexity of the program. The high cyclomatic complexity means that the logic of the program code is complicated, the quality may be low, and it is difficult to test and maintain. In general, methods with cyclomatic complexity greater than 10 have a great risk of error.

The cyclomatic complexity can be obtained by analyzing the control flow graph. There are three main calculation formulas, respectively as follows:

$$V(G) = E - N + 2P \tag{1}$$

Where $V(G)$ represents cyclomatic complexity, E represents the number of edges in the control flow graph, and N represents the number of nodes in the control flow graph, and P stands for the connected parts in the control flow graph.Since the control flow graph is connected, P is equal to 1 here.

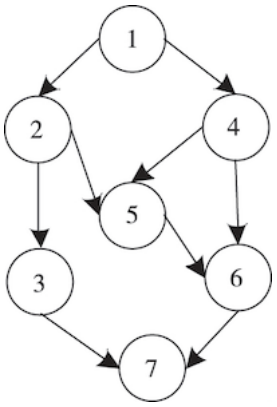

Fig. 2 Control flow graph G_1

Taking the control flow graph G_1 shown in Figure 2 as an example, according to formula (1), we can get: $V(G) = E-N+2P = 9 - 7 + 2 = 4$. The cyclomatic complexity of G_1 is 4.

$$V(G) = E - N + P \tag{2}$$

The difference between formula (2) and formula (1) is whether each exit node has an edge connecting the corresponding entry node in the control flow graph, that is, whether the control flow graph constitutes a strongly connected graph. The strongly connected graph refers to a graph that can reach all other nodes starting from any node in the graph. As shown in the control flow graph G_2 in Figure 3.

For the control flow graph G_2 shown in Figure 3, the formula (2) is calculated as: $V(G) = E - N + P = 10 - 7 + 1 = 4$. The cyclomatic complexity of G_2 is 4.

$$V(G) = P + 1 \tag{3}$$

Where P denotes the number of logical decision points. Logical decision point refers to that two or more outgoing edges are issued from each such node. This formula is the simplest and the easiest to use to calculate cyclomatic complexity, just find the decision points.Some well-known open source tools can be used to calculate cyclomatic complexity. For example, PMD, CheckStyle and JavaNCSS tools for Java programming language, OCLint,

© 2021 Information Processing Society of Japan

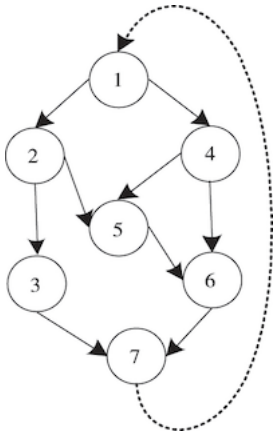

Fig. 3 Control flow graph G_2

Testwell CMT++, Understand, SourceMonitor and other tools for C programming language. For the Python programming language, there are also many cyclomatic complexity measurement tools, such as PyMetrics and Pygenie.

3. Human-Machine Pair Inspection Process

The inspection process supporting our inspection method is illustrated in Fig.4.The process consists of six major steps:Preprocess the recorded data, Generate the IR and CFG, Calculate the cyclomatic complexity, Generate the inspection syntax graph, Establish the checklist, Analyze the code to detect defects. Each step that describes an operation is represented by a diamond in the figure,and each data item, such as ”IR, CFG”, is represented by a box. The black arrow from an operation to a box indicates that the data item of the box is an output of the operation. An arrow between operations shows a control flow. The blue arrow from operation to Source code means the link between the questions in the checklist and source code. The yellow arrow from ”Establish the checklist” to ”Preprocess the recorded data” will be the future work to make the inspection more intelligent.

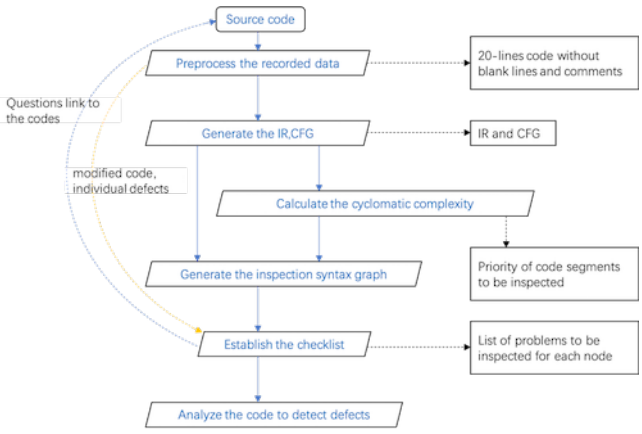

Fig. 4 Inspection process of our method

The **Preprocess the recorded data** step takes the source code being constructed by developer as input. Update the record synchronously to form a real-time mapping: $f : P \rightarrow P'$, where P is the code in the editor, P' is the code recorded by the inspection tool. Preprocess the recorded data, once the number of lines of code other than blank lines and comment lines reaches 20 lines, perform static analysis.It is set to 20 lines since that some researchers once did a survey [13]. They invited more than 400 volunteers to view 90 C++source code and indicate if they felt the program would be pleasant or unpleasant to review to determine typical characteristics of software that is pleasant to review. Their results show that the average function length of programs that were considered unpleasant to review is about 23 and the average length of programs that were considered pleasant to review is about 15. Therefore it is reasonable to inspect every 20 lines. Similarly,the average complexity of programs that were considered unpleasant to review is about 5.8and were considered pleasant to review is about 3.2. It is reasonable to set the minimum cyclomatic complexity of the code to be inspect to 4.

The **Generate the IR and CFG** step takes every 20-lines code as input and produce its IR and CFG. Because the goal is to inspect for semantic errors based on implementation, IR needs to be generated. In the process of programming, it is easy to ignore control flow information, such as calls between functions, and jumps between codes. The relationship between these codes is also crucial in the inspection process.Therefore, the generation of CFG is also essential.IR and CFG can be generated by existing tool.

The **Calculate the cyclomatic complexity** step takes IR and CFG as input to generate the priority of code segments to be inspected.As described in section 2.3, find out the decision node according to the control flow graph, and then the cyclomatic complexity can be obtained according to formula (3). According to the survey by Michael Dorin et al., code with a cyclomatic complexity not less than 4 is set as the high priority of inspection. The greater the cyclomatic complexity, the higher the priority of inspection.

The **Generate the inspection syntax graph** step takes IR, CFG of the codes and priority of inspection as input to show the components of the code that need to be inspected namely a set of inspection targets. Each target, denoted by a node in the graph. Retain all CFG components involved in functions with cyclomatic complexity not less than 4, sort these components in decreasing order of cyclomatic complexity, which means listing the most complex function components first. So as to guide the reviewer to inspect the code following this order.

The **Establish the checklist** step takes inspection syntax graph as input to generate questions in checklist.The checklist is related to the ISG: It contains a set of questions which can be derived automatically from the inspection targets in the ISG. For example, we can derive the following questions from the target ”while” node, which represents the while loop instruction:”Is each symbol in while loop correctly implemented?””Are the guard condition and defining condition in while loop correctly implemented?” These questions will facilitate a three-level analysis in an inspection, which is to be discussed next. Note that the phrase “correctly implemented” used above, and in the rest of the paper, means that the code without implementation-related bugs.

The **Analyze the code to detect defects** step uses the checklist and ISG as guidelines to analyze the related codes. Since ISG is generated by the IR and CFG of the code, it is easy to link

© 2021 Information Processing Society of Japan

the checklist to the source code. So the tool can conduct that when the mouse selects the checklist, the corresponding source code will be highlighted to guide reviewer inspect the code. The analysis of paths is aimed at answering all the questions on the checklist. For example, to answer the above questions derived from the target ”while” node, the analysis can be done at three levels:

(1) the symbol level,
(2) the condition level, and
(3) the relation level.

The questions on symbol and condition level can be automatically formed based on IR. The questions on the relation between codes can be automatically formed based on control flow analysis.

4. An Example

We use part of a stock reservation and purchase system as an example to illustrate the inspection process. The stock system can provide the following functions:Customer registration, Cancel a customer registration,Stock registration,Cancel a stock information,Purchase a stock,Sell a stock. Since the inspection of every operation follows the same process and uses the same principles, we only choose the calculate stock price function as an example here. The operation is implemented in Python. After preprocessing the recorded data,generating IR and CFG, We calculate that cyclomatic complexity of the calculate stock price function is 4, which needs to be inspected.For symbol-level analysis, we detected that the size of the dictionary changed due to the deletion during the iteration process when canceling delisting stocks.For relationship level analysis, through control flow information we found the infinite loop defect caused by mutual calls between functions when calculating stock prices.

5. Conclusion and Future Research

We have described a checklist-based human-machine pair inspection method to assure and improve reliability and quality of software. The underlying principle of the method is based upon the concept of cyclomatic complexity and IR: defects are expected to be found during the inspection to check all component of IR with high complexity. The process includes six major steps:Preprocess the recorded data, Generate the IR and CFG, Calculate the cyclomatic complexity, Generate the inspection syntax graph, Establish the checklist, Analyze the code to detect defects.Furthermore, we have conducted a case study to validate the method and presented our findings. The results show that our method may effectively reduce.

Three things will be the focus of our future research.First, we will continue to make efforts to build prototype tools to support more intelligent inspection.Second,make the problems in the checklist more specific and add possible implementation-related bugs in questions, such as stack overflow, etc. Finally, make the inspection process more intelligent, record the individual mistakes that programmers often make, recognize similar codes, and raise related questions in checklist.

Acknowledgments This work is supported by the Hiroshima University Graduate School Research Fellowship.

References

[1] Gregory, F.: Software Formal Inspections Standard, Technical report, Technical Report NASA-STD-2202-93,, NASA Office of, Safety and Mission . . . (1993).
[2] Parnas, D. L. and Lawford, M.: The role of inspection in software quality assurance, *IEEE Transactions on Software engineering*, Vol. 29, No. 8, pp. 674–676 (2003).
[3] Liu, S., Chen, Y., Nagoya, F. and McDermid, J. A.: Formal specification-based inspection for verification of programs, *IEEE Transactions on software engineering*, Vol. 38, No. 5, pp. 1100–1122 (2011).
[4] Baum, T., Liskin, O., Niklas, K. and Schneider, K.: A faceted classification scheme for change-based industrial code review processes, *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, IEEE, pp. 74–85 (2016).
[5] Fagan, M.: Design and code inspections to reduce errors in program development, *Software pioneers*, Springer, pp. 575–607 (2002).
[6] Bacchelli, A. and Bird, C.: Expectations, outcomes, and challenges of modern code review, *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, pp. 712–721 (2013).
[7] Davila, N. and Nunes, I.: A systematic literature review and taxonomy of modern code review, *Journal of Systems and Software*, p. 110951 (2021).
[8] Liu, S.: Software Construction Monitoring and Predicting for Human-Machine Pair Programming, *International Workshop on Structured Object-Oriented Formal Language and Method*, Springer, pp. 3–20 (2018).
[9] Ciolkowski, M., Laitenberger, O. and Biffl, S.: Software reviews, the state of the practice, *IEEE software*, Vol. 20, No. 6, pp. 46–51 (2003).
[10] Gilb, T. and Graham, D.: *Software inspections*, Addison-Wesley Reading, Masachusetts (1993).
[11] McCabe, T. J.: A complexity measure, *IEEE Transactions on software Engineering*, No. 4, pp. 308–320 (1976).
[12] Zuse, H.: *Software complexity: measures and methods*, Vol. 4, Walter de Gruyter GmbH & Co KG (2019).
[13] Dorin, M.: Coding for inspections and reviews, *Proceedings of the 19th International Conference on Agile Software Development: Companion*, pp. 1–3 (2018).

© 2021 Information Processing Society of Japan