The Detection of Faulty Code Violating Implicit Coding Rules

Tomoko MATSUMURA Akito MONDEN Ken-ichi MATSUMOTO
Graduate School of Information Science, Nara Institute of Science and Technology
E-mail: {tomoko-m, akito-m, matumoto}@is.aist-nara.ac.jp

Abstract

In the field of legacy software maintenance, there unexpectedly arises a large number of implicit coding rules, which we regard as a cancer in software evolution. Since such rules are usually undocumented and each of them is recognized only by a few members in a maintenance team, a person who is not aware of a rule often violates it while doing various maintenance activities such as adding a new functionality or repairing faults. The problem here is not only such a violation introduces a new fault but also the same kind of fault will be generated again and again in the future by different maintainers.

This paper proposes a method for detecting code fragments that violate implicit coding rules. In the method, an expert maintainer, firstly, investigates the cause of each failure, described in the past failure reports, and identifies all the implicit coding rules that lie behind the faults. Then, the code patterns violating the rules (which we call "faulty code patterns") are described in a pattern description language. Finally, the potential faulty code fragments are automatically detected by a pattern matching technique.

The result of a case study with large legacy software showed that 32.7% of the failures, which have been reported during a maintenance process, were due to the violation of implicit coding rules. Moreover, 152 faults existed in 772 code fragments detected by the prototype matching system, while 111 of them were not reported.

1. Introduction

Nowadays, many organizations have problems in keeping up the reliability and lowering the maintenance cost of large legacy software, which had been developed more than a decade ago and are still performing nucleus tasks for organizations [10][15]. During the long-term maintenance process of such old software, program code becomes more and more complicated than it used to be, and it becomes increasingly difficult to add a new functionality and repair its faults. This phenomenon is known as *code decay*, which is an inevitable part of software evolution [2]. In addition, the long-term maintenance also causes a lost of the maintainers' knowledge and experience of the aged software because many of today's engineers move rapidly between companies and job assignments[11][14].

We approach the phenomenon of code decay and the loss of maintenance knowledge from a point of *implicit coding rule[9]* –, which we regard as a cancer in software evolution. For example, the typical rules may informally be described as "In order to handle the function A, the global variable B should be initialized before calling the function C", "The function X and Y should not be used at the same time", and so on. These rules are different from the "Coding Standard" and "Coding Rules"(as normally said), and were generated unexpectedly and implicitly during a long-term maintenance process. So, usually these rules are not explicitly described in the specification documents and design documents. Moreover, even if someone tried to redesign the program for dissolving these rules, sometimes the new rules would be implicitly generated.

These implicit rules cause an increase of the maintenance cost and deterioration in the reliability of the software. If there are many implicit coding rules in the system, maintainers have to change the source code while paying attention to all the implicit coding rules, thus, the maintenance cost will increase. On the other side, maintainers who do not know these rules will inject faults to their changed code as a result of a violation of the rules as long as the rules exist. Moreover, it is difficult to detect these violations through a generic tool because the patterns of the rules and violations are usually highly dependent on each software system. Actually, we investigated certain legacy software and found that 32.7% of the failures were due to a violation of implicit coding rules and some of them have been caused by the same rule.

In this paper, we propose a method for detecting faulty code violating implicit coding rules. The method will be useful in lowering the maintenance cost because

7695-1796-X/02 $17.00 © 2002 IEEE

maintainers do not have to manually check all the implicit coding rules after changing code, and is useful in decreasing the number of faults made by maintainers who are not aware of the implicit coding rules. In addition, this method is very powerful in practical use because it can directly detect the line number of a faulty code area with detailed instructions, while previous methods for predicting fault-prone modules using software metrics only say which modules are faulty [4][7].

In the proposed method, the code patterns violating the rules (which we call "faulty code patterns") are described in a pattern description language. Then, the fault detecting system matches the source code of legacy software with the faulty code patterns, and it shows the matched code fragments to the maintainers. These faulty code patterns are extracted from past failure reports. In the failure reports, various information about the faults are written, such as the phenomena of the failures, causes of the failures, positions of the faults in program code, solutions for repairing faults, and so on. Many software companies make this kind of document during the development and maintenance process. The implicit coding rules can be mainly derived from the causes of the failures. Moreover, maintainers can check if the matched code fragments have a fault or not and repair it easily with the information from the past failure report.

This automatic detection of faulty code will be repeatedly performed by different maintainers once a pattern set is built. When maintainers change source code, they use this system in the changed source code, find the faulty code and can deal with them. By using this system, even maintainers who do not know the pattern, like a new member of the project, can keep the quality of their code.

The remainder of this paper first describes what the implicit coding rules and faulty code patterns are (Section 2), and our fault detecting system based on implicit coding rules (Section 3). Then we will describe a case study to evaluate our system (Section 4). Afterwards, we will describe other related works (Section 5), and in the end, conclusions and future topics will be shown (Section 6).

2. Implicit coding rules and faulty code patterns

2.1 Characteristics of implicit coding rules

The "implicit coding rules" have the following characteristics:

- These rules are seldom described in the specification documents or design documents. They only exist in the developers' and maintainers' mind.
- These rules are different from the "Coding rules" which have been decided at the starting point of the development process. These rules were generated unexpectedly and implicitly during a long-term maintenance process.
- These rules are quite specific to the particular software. This means there are different implicit coding rules to the different software. Therefore, it is too difficult to detect code fragments violating implicit coding rules by the existing checking tools or checklists because the generic checking tools (such as 'lint' and 'purify') and generic checklists detect only the generic problems.
- It is difficult to redesign for eliminating these rules due to the risk and the cost. If we are to redesign software, it is necessary to exactly understand the source code; however, the source code in an old system is much too complicated and there are no precise documents and no expert maintainers who know the system very well. Hence, the redesigning usually takes too much time, and sometimes it makes a new rule. "Refactoring"[3], which is one of methods for redesigning, also has the same problems mentioned above.

The presence of implicit coding rules causes an increase of maintenance cost and deterioration in the reliability of the software. Violating implicit coding rules causes an injection of faults, so maintainers have to change the source code while paying attention to all the implicit coding rules. Thus, the more rules there are in the software, the more cost is needed. Furthermore, there is always a risk of repeatedly injecting faults in the newly changed code as a result of a violation of the rules as long as the rules exist.

2.2 Detecting faulty code pattern and faults

We call the patterns of code violating implicit coding rules, "faulty code patterns". Here we show examples of implicit coding rules and faulty code patterns in Table 1.

Formerly, faults based on faulty code patterns were detected by the source code review and test, which are often very expensive. There are two major ways for detecting faulty code in the common software development. One is finding a fault manually when an implicit coding rule was found by seeking the cause of the failures. After a rule was found, the engineers must manually find all the other potential faulty code violating the rule. However, some of the potential faulty code may be overlooked. Furthermore, in the future, the

Table 1. Implicit Coding Rules for Faulty Code Pattern

Implicit Coding Rules	Faulty Code Pattern
A page number must be set to the global variable A before calling the function B for recovering screens after interrupting with a specific key.	Call the function B without setting a value to the global variable A
Certain functionality does not work if we use the function X and the function Y at the same time.	Call the function X and the function Y continuously

maintainers will not try to find the faults from their newly changed code because they may not be aware of the previously found rules. The other one is a code review. In the code review process, the reviewers need knowledge on the faulty code patterns or a listing of these patterns, yet, these rules are rarely described in the specification documents or design documents. Moreover, the knowledge and experience of the maintainers often disappears because many of today's engineers move rapidly between companies and job assignments. In addition, in a large-software development, it is difficult to share the knowledge with multiple working groups. Besides, it is difficult to review a whole source code of large legacy software because it is just too great in content to completely read and understand. Furthermore, when some different groups or maintainers are updating different files at the same time, it is impossible to check the violations of rules completely.

In this paper, however, a code fragment that matches a faulty code pattern does not mean the fragment definitely has a fault, i.e., violating an implicit coding rule may introduce a fault, but not always. It is because the pattern in this paper is a static one and does not consider the dynamic relations between the code fragments. We regard that a faulty code pattern is a pattern of code that significantly contains a fault. Therefore, if a maintainer finds code that matches a faulty code pattern in their changed code, they must check if there is a fault or not in fact. For checking, the maintainer needs some information about the pattern, for example, what failure the pattern had caused in the past, how to reproduce the failure, and so on.

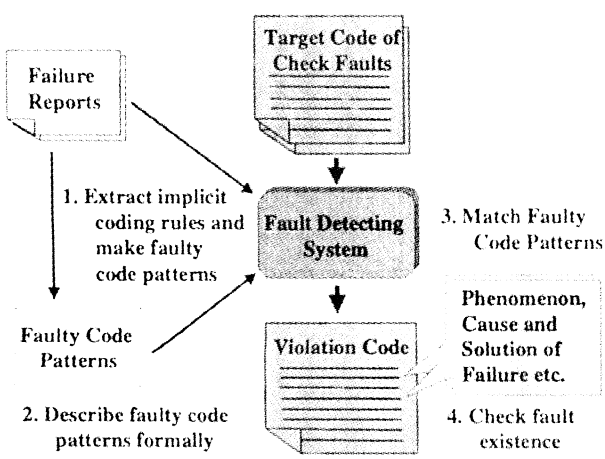

Figure 1. Procedure of Fault Detecting System

2.3 Faulty code patterns in industry software

We have investigated implicit coding rules and faulty code patterns in a subsystem of large legacy software (see detailed information in Section 4.2). In this investigation, we have found that 54 failures out of 165 failures described in failure reports were due to the violation of implicit coding rules. We could extract 45 implicit coding rules from these 54 failures. 9 failures have been caused by the same rule. 2 of these 45 rules were dissolved in the maintenance process, but 43 still remained in this software, thus, these rules may cause fault injections by other maintainers in the future.

Some faults have been introduced by oversight during the code review, and another caused by violating the same implicit coding rule have been introduced by different working groups. Moreover, one rule caused a failure in another version of the system. We realized that since it is quite difficult to manually detect all the faults based on the implicit coding rules, the faults were repeatedly generated in the long-term maintenance.

3. Fault detecting system

3.1 Procedure of the fault detecting system

In this paper, we propose a method for detecting code fragments that match faulty code patterns. We call the implementation of this method, "Fault Detecting System". The procedure of our method is the following (see Figure 1):

1. Experts investigate failure reports and extract implicit coding rules that have caused the faults. Next, they extract the faulty code patterns based on these rules.
2. Experts formally describe the faulty code patterns in a pattern description language (explained in Section 3.3), and store them in the system.
3. The system matches the faulty code patterns with the input program code. If the system finds some

matched code fragments, it outputs detailed information about them.

4. Maintainers check the output code fragments with the failure information to discover whether they are faults or not.

In addition, the set of faulty code patterns will be updated every time someone finds a new implicit coding rule in the system or if he/she dissolves one of the rules through reengineering work.

We will discuss about each phase in detail from next subsection.

3.2 Extracting implicit coding rules and make faulty code patterns

Implicit coding rules can be extracted from the failure reports, which are one aspect of process data. Many organizations make such documents when failures occur, faults are repaired, or the fixed code is released.

In this paper, we used 3 types of failure reports; failure occurrence reports, fault solution reports, and file update reports. In these reports, there are information about the situation in which the failures have occurred, works and results of the solution process, and releases. Some important information in this method is a phenomenon, cause, and solution of the faults (see Figure 2). Implicit coding rules are mainly extracted from the causes of the failures. However, they are very useful not only for extracting implicit coding rules but also for confirming the failure occurrences, to check for the faults existence, and to repair them.

Conditions for extracting implicit coding rules are the following:

- A cause of failure is clear on the program code.
- There is a possibility of injecting other faults caused by the same rules in the future.

The global knowledge and prospect of the target software are necessary to make the faulty code patterns from implicit coding rules. The maintainers have to judge whether the patterns are important or not and also have to extract the "appropriate" faulty code pattern. In case the derived faulty code patterns are inappropriate, we cannot detect the proper code fragments that should be checked, or we may detect too many code fragments that do not need to be checked.

From a failure described in Figure 2, we can extract a rule like "The page number must be set to *CtrlNo* before calling *ChangeView()* ", and make a faulty code pattern like "Absence of a statement setting a value to *CtrlNo* before calling *ChangeView()* ".

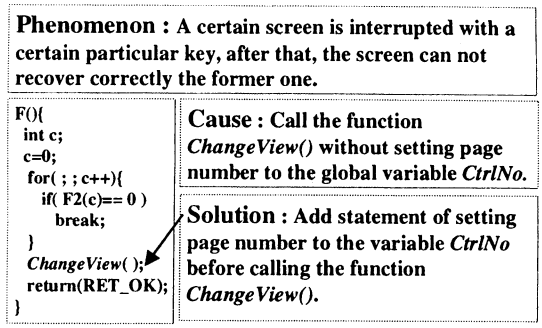

Figure 2. Example of Failure Report

declaration	$d	$*d	$d_{name}
type	$t		$t_{name}
variable	$v	$*v	$v_{name}
function	$f		$f_{name}
expression	#	#*	#_{name}
statement	@	@*	@_{name}

$v	Wildcard for Variables
$*v	Wildcard for Collection of variables
$v_{name}	Named Wildcard for a variable
@[stmt1| stmt2]	Any of the specified statement
@<id_1>	refers to/uses identifiers
%%	Symbol for separating two sections

Figure 3. Pattern Description Symbols (from [13])

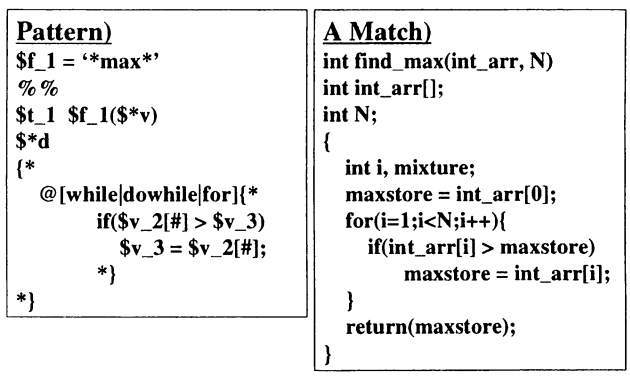

Figure 4. Pattern for finding the maximum in an array of integers. (from [13])

3.3 Describing patterns in formal description

In this system, we use a pattern description language proposed by S. Paul and A. Prakash[13]. This pattern description language has been developed for reengineering code, understanding code, and so on. Users can describe patterns using symbols that have syntactic meanings on the source code, like "function", "variable", and so on (see Figure 3 and 4), instead of a string search (such as "grep"). So, it is possible to write code patterns that the users exactly want. This language is an extended

version of the underlying programming language, thus, learning to write patterns is easy and the scalability is good. Figure 4 shows main syntactic entities for describing patterns. Paul and Prakash selected these entities "based on our perceptions of what maintainers typically look for", and they also say "if queries requiring pattern matching on other syntactic entities were required, such syntactic entities could be added easily to the pattern language without changing the basic design of the system"[13]. An example of a pattern in this language is shown in Figure 4.

In this research, we will describe the faulty code patterns in this language. Considering the actual faulty code patterns extracted from real legacy software, we extended their pattern description language (See Figure 5). Below we describe our extensions:

- Many faults are caused by the absence of a vital statement. In turn, we need a symbol that expresses the absence of a statement (this will be an expression of the existence of any statement excluding the target statement).
- Global variables are more risky in causing faults than local ones. For example, if one global variable that is used in multiple modules has changed in a certain module, this change often affects other modules and thus causes faults. Therefore, it is important to distinguish the "local" and "global" variable in the faulty code pattern, so we added a symbol of the "global variable".
- In large software, there is a strict coding rule for deciding the names of the functions and variables, so if we can describe the naming rules in the pattern language, it will be useful to describe the patterns that the maintainers exactly want. We believe that regular expressions (e.g. used in "grep" command) are very useful to express these rules.
- Some of the faults in the complicated software are caused by multiple code fragments, which exist in multiple functions and/or multiple files. Therefore, in order to represent one faulty code pattern in one pattern description, it is necessary to match multiple chain patterns.

3.4 Matching with faulty code patterns

The matching system architecture SCRUPLE from the paper[13], is shown in the Figure 6. The source code is transformed into an AST (Attributed Syntax Trees; see Figure 7) by a code parser, and the faulty code patterns written by users are transformed into a CPA (Code Pattern Automata; see Figure 8). The CPA interpreter runs the CPA with the AST as inputs. As the CPA reaches the final state (e.g. *q6* in Figure 8), the

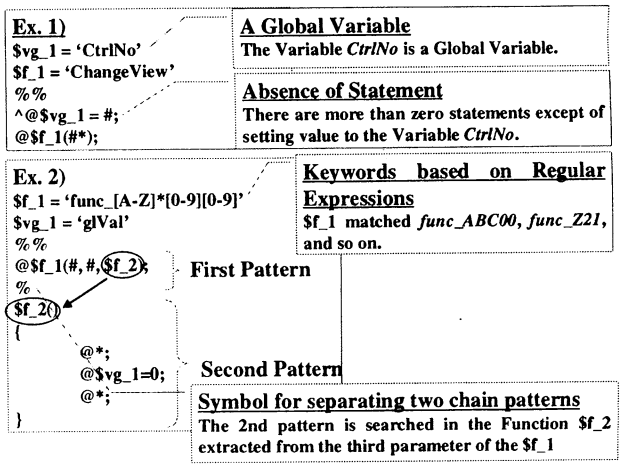

Figure 5. Extension of Pattern Description Symbols

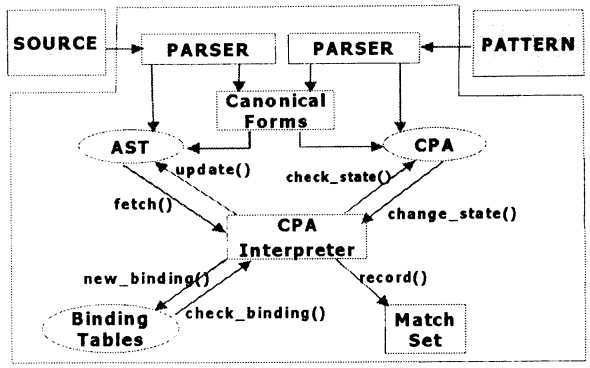

Figure 6. The Architecture of the SCRUPLE System. (from [13])

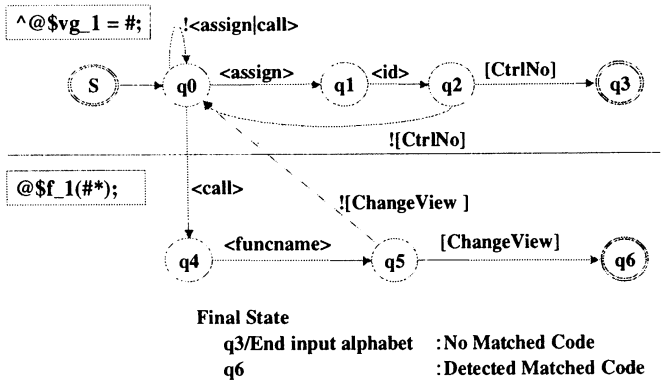

Figure 8. Example of transition diagram by Fig. 5

interpreter saves the code fragment into a match set. Also, the interpreter maintains the information about bindings of the named wildcards (e.g. $f_2 in Figure 5) in the binding tables.

We have implemented a system based on this architecture for C language. The system makes an AST

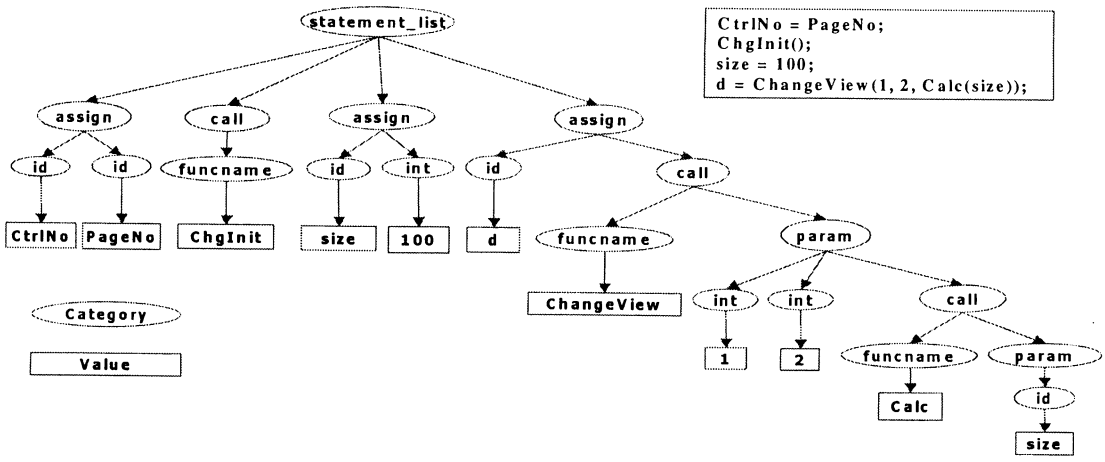

Figure 7. Example of AST(Attributed Syntax Trees)

for each function in the program code and matches the faulty code patterns. The system has some additional handlings in the AST parser for addressing the extension of the pattern description language as follows:

- The source code parser checks if the declaration of each variable in the statement exists in the same function or not, and adds the information of "local" or "global" to each variable on the AST.
- The AST parser normally analyzes in each step of code; however, the code fragments that matched the faulty code patterns sometimes exist while stepping over several functions called from a certain function. Therefore, when we extract an AST from a function in source code, we also go through the functions that are called from a target function. This function trace was done by the preceding depth search. In this search, it is necessary to make sure that a function that has been visited once must not be visited again.

3.5 Presentation and application of result

The result of the detection of the faulty code patterns will be presented to the maintainer with the code fragment and the detailed information about the fault. Figure 9 shows an example of the resulting matched faulty code patterns in Ex.1 of Fig. 6. Since the fragments that match the faulty code patterns are not always the faults, the maintainers must check the fragments and find the faults with the detailed information of the matched faulty code patterns. If there is a fault found in a matched fragment, the maintainers can repair it at a low cost because they can refer to some additional detailed information and learn how to test and solve it.

3.6 Usage of the method

This proposed method is used in the following two situations:

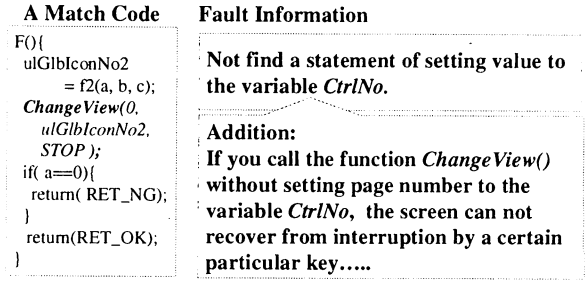

Figure 9. Example of Presentation of result from the Fault Detecting System.

- For code review: When developers and maintainers change program code, the proposed method can be used before a code review by a third person. They can check the changed code using all the faulty code patterns that have been found in the past. In this case, if a fault was found, the maintainers can correct it before going to next phase.
- When a new rule is found: When a maintainer finds a new implicit coding rule, the maintainer can check the whole program code using a faulty code pattern based on the newly found rule. In this case, the maintainer can find implicit faults injected in the past maintenance activities.

4. Case study

4.1 Objective

In this case study we will evaluate the effectiveness of the method for the detection of faulty code from two points of view.

[Usefulness of this method]

We will evaluate whether the detection of any faulty code fragments that match the faulty code patterns is

actually beneficial for the maintenance process. As mentioned in Section 2.3, the rate of failures caused by the violation of implicit coding rules is 32.7%. However, not all of the code fragments detected by the system are the faults, so if there are too few faults in the detected code fragments, it will only cause the increase of maintenance cost for checking the detected code fragments. Moreover, if these failures/faults can be easily detected by other methods, such as testing, it is not clear that our method is beneficial. From these points, we will investigate the following data:

- Ratio of faults to all of the detected code fragments
- Ratio of faults, which have not been reported as failures in the testing or running phase, to all of the detected faults.

[Performance of the system]

We will evaluate whether the fault detecting system with the pattern description language and pattern matching technique is useful for detecting code fragments that match the faulty code patterns. It is important to know how much ratio of the faulty code pattern can be descriptable in the proposed language because the patterns cannot be detected unless it is described. Moreover, it must be confirmed that the known faults can be definitely detected by the system. From these points, we will measure following data:

- Ratio of descriptable faulty code patterns in the original pattern description language to all of the faulty code patterns.
- Ratio of descriptable faulty code patterns in the extended pattern description language to all of the faulty code patterns.
- Ratio of detected faults to all known faults.

4.2 Material and procedure

In this case study we used a subsystem of large legacy embedded software, including hardware control modules and user interface modules. This subsystem is written in C language and the size is about 447,000 steps. It was developed in 1991, and it is still running (see Figure 10). The version we used for the case study was picked out of many different versions, which had been developed and maintained from April 1997 to May 1999.

In this case study, we used the system in the 2nd usage in Section 3.6, i.e. we detected faults from the whole program code by matching each of the faulty code patterns. We used the program code as the starting point of the maintenance (i.e. B=Dec.1, 1997). For making the faulty code patterns, we used the failure reports that were made between Dec.1, 1997 and May 1999 (B-C) because we could not get the reports between A and B. We could get the faulty code patterns, which have been created between A and C, then, we extracted the faulty code patterns which existed at point B for the matching in the source code in B.

Originally we must conduct the experiment with the source code at the point when each faulty code pattern had been found. However, we did not have the whole source code at each of the points, so we used only the source code at the point of B. Therefore, we could not get the faults which had been introduced between B and C.

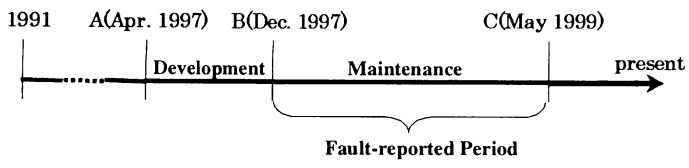

Figure 10. Schedule of the software used by case study.

4.3 Results

The results of the evaluation are shown in Table 2. Also, the appendix shows some typical faulty code patterns in the formal description.

We have manually counted the number of "failures" and "(implicit coding) rules". There were 39 rules at the starting point of the maintenance process, and only 17 of them were descriptable in the original pattern description language. However, by extending the language, 30 faulty code patterns became descriptable. Then, the system matched these 30 patterns and the causes of the 33 failures that were found in all of the detected code fragments against the 38 failures due to 30 faulty code patterns.

"Code fragments" is defined as the fragments of program code that is automatically detected by the fault detecting system. We manually checked 772 code fragments outputted from the system, and found 152 faults, which possibly cause failures (we could not confirm the failures happen on the running system). Moreover, we checked whether these possible faults have been reported as a failure or not and found that 111 of them were unreported, i.e. potential faults.

We could recognize some problems in the results; we could not describe 9 implicit coding rules in the pattern description language (D-F in Table 2), and the system could not detect 5 failures that have been caused by the faulty code patterns used in the matching process (G-H in Table 2). An example of the patterns, which were not descriptable, is that if function A would be changed, function B probably has to be changed. In this case, we have to get the difference between the pre-changed code and post-changed code and check whether a difference in the code exists in function B. Therefore, we have to extend not only the language but also the

Table 2. The Results of the Evaluation

	Item	Count	Ratio	Notes
A	All the Failures	165		Failures whose cause had been reported
B	Failures caused by a violation of the implicit coding rules in A	54	32.7%	= 54 / 165
C	Implicit coding rules extracted from B	45		10 failures were caused by the same rule
D	Rules that exist at the point of starting the maintenance	39		Point of Starting maintenance = 1/12/1997
E	Descriptable Rules in the original Pattern Language	17	43.6%	= 17 / 39
F	Descriptable Rules in the extended Pattern Language	30	76.9%	= 30 / 39
G	Failures caused by the violation of the implicit coding rules in F	38		Failures reported in the testing or running process
H	Failures detected by the Fault Detecting System in G	33	86.8%	= 33 / 38
I	Detected Code Fragments by the Fault Detecting System	772		Detected by using Patterns in E
J	Code Fragments whose faults were found in I	152	19.7%	= 152 / 772
K	Code Fragments that were not reported as faults in J	111	73.0%	= 111 / 152

matching system from a simple syntactic matching one. Also, one of the reasons for the undetected failures is the interruption of the matching process because of the lack of memory or stack.

According to the problems mentioned above, it is necessary to extend the pattern description language and improve the matching system with the extension of language. Moreover, it is important to improve the matching algorism to become a practical system.

4.4 Discussion

[Usefulness of this method]

19.7% of the detected code fragments were faults (J in Table 2), and only 27.0% of them had been found as failures (K). 73.0% of faults, which needed to be manually reviewed for detecting, could be automatically detected. Furthermore, only a part of the faults based on the implicit coding rules were found as a failure, like a tip of an iceberg, and these potential faults could be easily found by this method.

From these results, this method makes it able to detect not only the faults that have caused failures but also potential faults that will cause failures in the future. Therefore, it can improve the reliability of the software. Moreover, the maintenance cost would also be lower because it is possible to repair the potential faults before becoming a failure in the testing or running process.

[Performance of the system]

The ratio of descriptable faulty code patterns was improved from 43.6% to 76.9% by the extension of the pattern description language (E and F in Table 2). This data shows that the selection and extension of the pattern language is appropriate because the ratio was widely improved by slight extension and it was not necessary to widely change the original architecture. It shows that the scalability of the original language is very good.

The ratio of detected known faults (H) shows that most of the faults, which we have expected to detect, could really be detected.

From this result, we confirmed that the procedure of detecting a violation code in our method is useful for detecting real faults and the performance is appropriate.

5. Related works

5.1 Checklists

This is a method for detecting mistakes using a list whose checking items are written for the coding and design. There are many popular checklists for detecting generally easy-mistaken items [5][6][12]. Also, there is research being conducted to select the necessary check items for each program [8].

These checklists have been created for applying themselves to the general software, so they do not include detailed check items for individual systems.

The faulty code patterns in our study can be included in the checklists; however, checking manually requires a lot of time and manpower if there are many patterns. Thus, an automatic detection system is needed.

5.2 Predicting fault-prone modules

This is a method for predicting fault-prone modules by using the software metrics data, e.g. lines of code, the number of calling functions, the number of loops, and so on. If the modules could be classified into a group of fault-prone and non fault-prone, maintainers can review or test the fault-prone modules intensively and find the faults efficiently.

Khoshgoftaar et al. researched predictors using the process data, e.g. number of new and changed lines of code, number of updates in designers' company careers and deployment usage [7]. Graves et al. presented their research that measures the number of changes and the age of code from the software change history improved the predicting accuracy [4]. Also, Andrews et al. researched the predicting method using the defect history and release data [1].

These methods can predict fault-prone code in a unit of module. In that, it is actually expensive to find the faults from the detected fault-prone modules.

6. Conclusion

This paper pointed out that the phenomenon of code decay in software evolution could be explained as an increase of the implicit coding rules. We found that a considerable number of failures are due to the violation of implicit coding rules (54 out of 165 failures). We executed an experiment applying this method for fault detection in the software. As a result, 76.9% of these rules were descriptable in the pattern language; and 86.8% of the faults, which have been reported and described, were extracted from the program code by the prototype matching system. Moreover, many potential faults based on some common faulty code patterns were automatically detected; we have found 152 faults based on 30 faulty code patterns. This result shows that the maintainers have repeatedly generated similar faults based on the same faulty code pattern. Moreover, 111 potential faults, which have not been reported, were also detected. From this result, we believe that the method is useful and practical in enhancing the reliability and reducing the maintenance cost.

The related works have mainly analyzed the complicated program code of legacy software at a viewpoint of the phenomenon, such as "code decay"[2][10]. However, it was difficult to give the maintainers useful feedback from the result of these works. In this paper, we analyzed the problem from a viewpoint of the maintainers, and proposed a method to directly solve it, i.e., we recognized the complexity of the program code as a generation of implicit coding rules, and summarized the significance and effectiveness of detecting faulty code violating implicit coding rules. Moreover, we proposed a concrete method for detecting the faulty code and evaluated the effectiveness and performance of the method through the case study.

In this paper, we have reported a case study for detecting faults from the entire program code at the starting point of the maintenance process; however, we must go further and evaluate our method by detecting from changed code during the maintenance process.

Finally, we have some future works for making the method more effective and useful.

- Improvement of the pattern description language.
- Development of a matching system that is quicker and more accurate.
- Make a guideline to describe the faulty code patterns.

7. Acknowledgement

This study was supported by the Industrial Technology Research Grant Program from the New Energy and Industrial Technology Development Organization (NEDO) of Japan.

8. References

[1] A. A. Andrews, M. C. Ohlsson, and C. Wohlin, "Deriving fault architectures from defect history," *J. of Software Maintenance: Research and Practice*, Vol. 12, No. 5, pp. 287 - 304, Sept.-Oct. 2000.

[2] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? Assessing the evidence from change management data," *IEEE Trans. on Software Engineering*, Vol. 27, No. 1, pp. 1 - 12, Jan. 2001.

[3] M. Fowler, *Refactoring: Improving the design of existing code*, Addison-Wesley, 1999.

[4] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. on Software Engineering*, Vol. 26, No. 7, pp. 653 - 661, July 2000.

[5] C. P. Hollocker, *Software reviews and audit handbook*, p. 162, John Wiley & Sons, 1990.

[6] W. S. Humphrey, *A discipline for software engineering*, Addison-Wesley, 1995.

[7] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl, "Data mining for predictors of software quality," *Int'l J. of Software Engineering and Knowledge Engineering*, Vol. 9, No. 5, pp. 547 - 563, 1999.

[8] F. Macdonald, and J. Miller, "A comparison of tool-based and paper-based software inspection," *Empirical Software Engineering*, Vol. 3, No. 3, Autumn 1998.

[9] T. Matsumura, A. Monden, and K. Matsumoto, "A Method for Detecting Faulty Code Violating Implicit Coding Rules," *Proc. 5th International Workshop on Principles of Software Evolution (IWPSE2002)*, pp. 15-21, May 2002.

[10] A. Monden, S. Sato, K. Matsumoto, and K. Inoue, "Modeling and analysis of software aging process," F. Bomarius and M. Oivo (Eds), *Lecture Notes in Computer Science*, Vol. 1840, pp. 140 - 153, 2000.

[11] A. Monden, S. Sato, and K. Matsumoto, "Capturing industrial experiences of software maintenance using product metrics," *Proc. 5th World Multi-Conference on Systemics, Cybernetics and Informatics*, Vol. 2, pp. 394 - 399, July 2001.

[12] G. J. Myers, *The art of software testing*, John Wiley, New York, 1979.

[13] S. Paul, and A. Prakash, "A framework for source code search using program patterns," *IEEE Trans. on Software Engineering*, Vol. 20, No. 6, pp. 463 - 475, June 1994.

[14] E. Regelson and A. Anderson, "Debugging practices for complex legacy software systems," *Proc. International Conference on Software Maintenance*, pp. 137-143, Sept. 1994.

[15] N. F. Schneidewind and C. Ebert, "Preserve or redesign legacy systems?" *IEEE Software*, Vol. 15, No. 4, pp. 14 - 17, July/Aug. 1998.

Appendix

- *Pattern A*

```
$f_1 = 'GotoA'
$f_2 = ' GotoB '
$f_3 = ' GotoC '
$f_4 = ' GotoD'
$f_5 = ' GotoE'
$f_6 = ' GotoAll'
$m_1 = 'OPERATE_A'          *
$t_1 = 'EventTbl'
$m_2 = 'EVENTPROC'
$m_3 = 'SUBEVENT'
%%
struct $t_1 $v_1[] = {
        *#,
    $m_2( $m_1, #, # ),
        *#,
};
%
struct $t_1 $v[] = {
    *#,
    $m_3( #, $v_1 ),
    *#,
    $m_2( #, $f_7, # ),
    *#,
};
%
$f_7()
{
    *@;
    @[$f_1 | $f_2 | $f_3 | $f_4 | $f_5 | $f_6]($*v);
}
%
```

- *Pattern B*

```
$f_1 = 'SetKeyX_On'
$f_2 = 'ScreenInit'          *
%%
@$f_1($*v);
*@;
@$f_2;
%
```

- *Pattern C*

```
$f_1 = 'func_rstuv'  *
$f_2 = 'item__[a-z]*'
$f_3 = 'func_draw'
%%
@$f_1(#, #, #, $f_4 );
%
$f_4()
{
        *@;
        @[ $f_2 | $f_3 ](#);
        *@;
}
%
```

- *Pattern D*

```
$f_1 = 'disp_string'         *
$f_2 = 'set_disp'
%%
^@$f_2( *# );
@$f_1( *# );
%
```