In Proc. 6th Joint Workshop on System Development
(JWSD2003), CD-ROM Edition, Nagoya, Japan, April 2003.

Analysis of Program Reading Process in Software Debugging Based on Multiple-view Analysis Model

Shinji Uchida[†], Akito Monden[††], Hajimu Iida[††], Ken-ichi Matsumoto[††] and Hideo Kudoh[†††]
[†]Electrical and Information Engineering, Kinki University Technical College,
[††]Graduate School of Information Science, Nara Institute of Science and Technology,
[†††]Faculty of Modern Management Information, Osaka Seikei University
uchida@ktc.ac.jp,{akito-m@is,iida@itc,matumoto@is}.aist-nara.ac.jp, kudoh@osaka-seikei.ac.jp

Abstract

This paper proposes a model for analyzing the reading process in software debugging. The model provides quantitative and objective visions to human's debugging activity, and provides framework for clarifying good- and/or bad-process for program reading. We have conducted experiments to observe debugging processes and collected process data based on the model. Some hypotheses such as "Expert subjects spend more time to read the faulty module than novice subjects do." were actually rejected; and, some other hypotheses such as "Reading the modules that have no dependence with faulty module degrade the debug efficiency." were accepted. The result of experiment suggested that explicit and quantitative evaluation of program reading process becomes possible by using the proposed model.

1. Introduction

Recently, software maintenance groups often need to debug programs that are not originally developed by them because today's many engineers move rapidly between companies and job assignments [8]. If a critical failure has occurred in a program after its release, engineers must quickly find a bug (fault) that has caused the failure even in case the program is new to the engineers. In addition, reuse activities in modern software development also force engineers to read and find bugs in reused code that is unfamiliar to them. Therefore, it is getting more and more important for software engineers to be able to quickly find and remove bugs in unfamiliar programs [11].

Unlike debugging familiar programs, debugging unfamiliar programs requires engineers to use some special expertise in program reading. In order to start finding bugs in an unfamiliar program, engineers must read and comprehend the program, however, engineers usually do not have enough time to comprehend the entire program because they must detect bugs within the scheduled time. Hence, engineers should somehow select the area that seems to lead to the bug detection, and, engineers should not read the area that is unrelated to the bug detection. Here, the strategies of area selection seem to greatly affect the efficiency of debugging. Therefore, engineers should somehow decide the area that seems to lead to the bug detection.

The goal of our research is to clarify both good- and bad- reading process for debugging unfamiliar programs. Our approach is to analyze debugging tasks under the controlled environments, and to find and/or to verify patterns peculiar to experts' (and novices') reading processes. Yet, observing subjects' external reading processes (such as "Which area did they read?" and "When did they read?") is not sufficient. We need to investigate subjects' intentions – "Why did they read?" In our previous researches [11][12], we have observed that strategies to select a module (= an area that should be comprehended) strongly relate to engineers' impressions of each module – either the module is faulty, not faulty or uncertain. Therefore, in order to clarify reading process, we need to follow up engineer's cognitive image of a program that represents faulty area, not faulty area and uncertain area. Moreover, structural properties of the target program should be taken in account as well. Some past researches say that the static slice of a program should be considered in debugging [3][15][7].

We propose a new modeling schema: Multiple-View Analysis Model of Debugging Process[12][13]. This

model provides two kinds of view to program reading in debugging process: Product view and Cognitive view. Product view represents target program's module structure and properties, while Cognitive view presents the property of human's activity. In order to analyze unclear part of human's activities in clear and well-defined way, target program structure is set as a basement of analysis framework. Human activities such as movement of reading target or module classification are, then, cast over the Product view. Cognitive view actually consists of following two views:

-Decision view: Human's cognitive image of the possible location of the bug.

- Behavior view: Externally observed human's action.

By using above three views (Product view, Decision view, and Behavior view), analysis of debugging processes may be done in clear way, by examining interactions and relationships among these views.

In order to get some insights concerning good- and bad-reading process for debugging, we actually carried out an experiment to observe debugging processes using video recordings and periodical interviews; and, we analyzed collected data based on our model.

In the rest of this paper, Section 2 describes the detail of our analysis model. Section 3 describes experiments to analyze debugging processes based on the proposed model. Section 4 describes the result and discussion of the experiments. Finally, Section 5 summarizes this paper.

2. Multi-View Analysis Model of Debugging Process

Fig. 1 shows the overview of the model. In this model, we set the recognition granularity of the target program to the module-level. E.g., the target program is abstracted as a set of multiple modules . Consider we have n modules, and the target program P is represented as a set of modules {m1 … mn} .

P = {m1, m2, … , mn}

To make following discussion simple, we assume that there is only one bug in the program. This assumption is natural to present a situation that we have one failure observed and an engineer must find a bug (fault) concerning to that failure. We also assume that the location of the bug is module m1 in the following discussion.

The Product view provides structural characteristics of the target program, which is independent from actual debugging process. The Cognitive view provides the view of actual human activity in debugging process. The Cognitive view consists of the Decision view and the Behavior view, presenting internal and external action of a debugger (= a person who debugs a program) respectively.

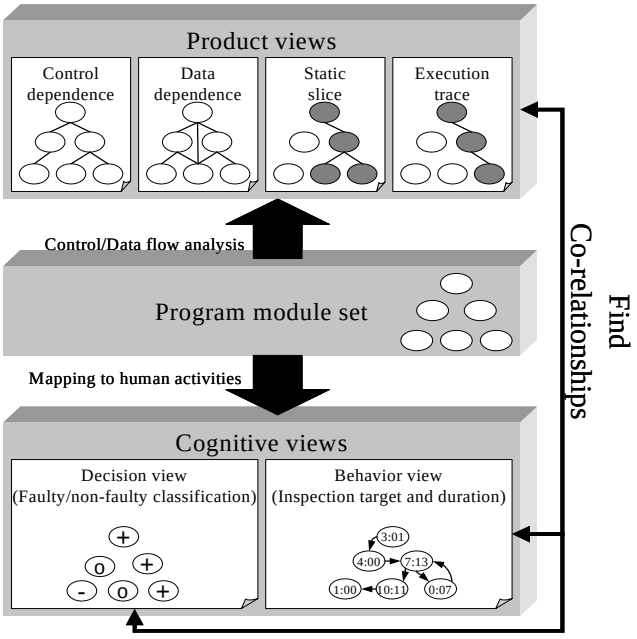


Fig.1 Multi-View Analysis Model for Debugging Process

2.1. Product view

The Product view expresses the abstract characteristics of the target program. Following four characteristics are presented:

·Control dependence between modules (in this case this is equivalent to module call structure) (Fig. 2-(a)),

·Data dependence between modules (Fig. 2-(b)),

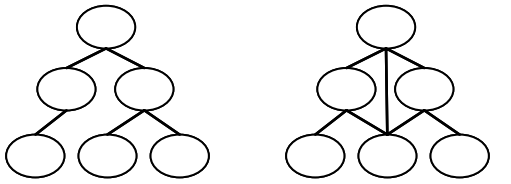
·Static slice [14][15] calculated from wrong output value (Fig. 2-(c)), and

·Faulty execution path (Fig. 2-(d)).

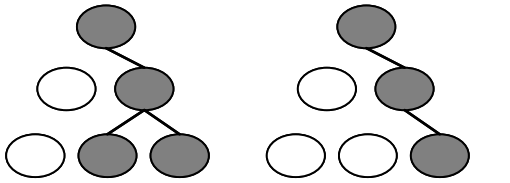
Program slice is the set of program statements possibly influencing the value of a particular variable in a particular statement (i.e. slicing point). [3] and [7] report that static slicing is useful for software maintenance. In our approach, we don't use slice for actual debugging. We use static slice just for process analysis.

As a summarization of the Product view, modules are categorized following four areas: S, E, SE and O. Area S is the set of modules included in the static slice. Area E is the set of modules included in the path of faulty execution (Fig. 2-(d)). Area E naturally contains the faulty module where the bug exists. Area SE is the set of

modules included in both of area S and area E. Area O is the set of the modules not belonging to any of above. In this area, there is naturally no faulty module.



(a)Control Dependence (b)Data Dependence



(c)Static Slice (d)Faulty execution path

Fig.2 Product View

2.2. Decision view

The Decision view presents internal action of the debugger. Here, we set a fundamental assumption that the debugging process is essentially expressed as a sequence of two kinds of action: expanding the trusted region of the program while narrowing down the suspicious region [1]. These actions are essentially influenced by the debugger's subjective suspicion about the bug's location.

In the Decision view, modules are continuously and subjectively classified into three regions (set of modules) during the debugging process. The first region is the suspicious region (shown as B+), which is the set of modules that might contain a bug. The second region is the trusted region (shown as B-), which is the set of modules that seems to be no bug. The third region is the neutral region (shown as B0), which is the set of modules uncertain whether containing the bug or not (Fig 3). Occasional movement of modules over these three regions can explicitly express the change of human internal impression of the bug location.

Hence, the debugging process in this view is observed as transitions of the status of the human internal impression S(t), which is represented with actual assignments to B+, B0, and B-.

At the beginning of the unfamiliar program debugging (t=0), the debugger does not understand any modules. Therefore, the initial state S(0) should be expressed as follows:

S(0) = [B+= f, B0= {m1, m2, … , mn}, B-=f].

As the bug localization activity proceeds, modules in B0 will be moved to either B+ or to B-. At the final stage of the debugging process (t=T), the bug has been localized, and the final state S(T) should always be as follows:

S(T)= [B+={m1 }, B0= f, B-={ m2, … , mn}].

The transition process from the initial state S(0) to the final state S(T) may greatly differ according to the debugger's capability and strategy. By using the Decision view, transition of the human decision can be formally expressed, and individual differences of the decisional transitions can be observed.

Since transitions are performed in the debugger's mind inside, actual decision movements cannot be observed externally. We use periodical simple interviews to the developer to get impression of each module.

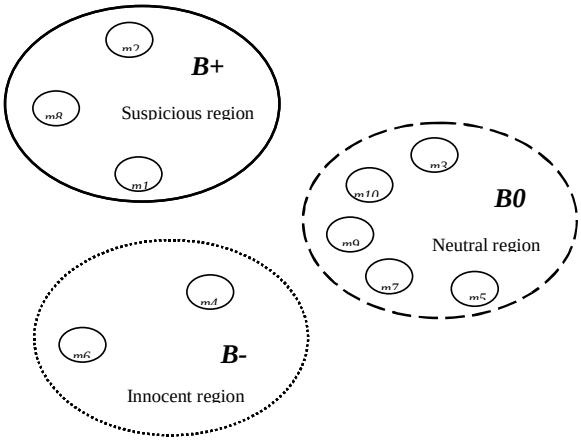


Fig.3 Decision View

2.3. Behavior view

Behavior view can express the external reading action of the debugger. In this view, human's behavior is represented as a sequence of module reading activity that is expressed as a pair of target module name and the reading duration, such as <m3,0:40>, <m2,0:30>, … , <m1,1:50> (Fig 4). This information can be captured in several ways such as video monitoring, command execution history, or eye gaze tracking. The order and the frequency of module readings clarify how the debugger read the program.

The Behavior view can be used to see each debugger's way of limiting the referenced module set. People usually don't read all of the modules for debugging, and sometimes there is implicit or explicit strategy for module choice [5]. Moreover, skilled programmer's patterns of movement over modules might be significantly different from a novice programmer's patterns. This view is also capable of investigating such skills differences [4].

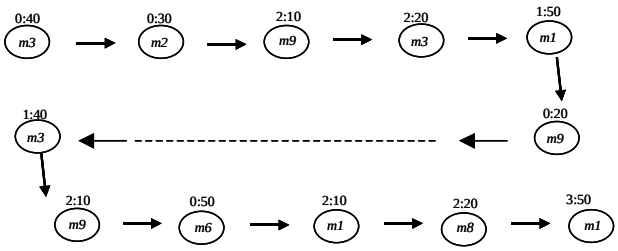


Fig.4 Behavior View

2.4. The advantage of the proposed model

The main advantage of this model is that the activities of the debugger, both internal and external, become clear, and quantitative and objective analysis can be applied to it. Generally speaking, observation and explanation of the debugger's internal activity are very hard. Many existing analyses mainly depend on subjective statements from debuggers, and resulting analysis is also highly subjective. For example, in Araki's model, the debugging process is explained as iterations of debugger's hypothesis evolution, which cannot be observed quantitatively [1]. In Vessey's model, the debugging process is expressed based on debugger's chunking ability [14]. However, these models cannot illustrate the objective activities of debuggers'.

In our model, multiple-view architecture is provided for analyzing debuggers' activities. Combining these three views over the module set, just like transparent sheets, enables various analyses. By placing the Decision view over the Product view, we can analyze the co-relation between debugger's internal recognition and the program structure. For example, we can evaluate the debugger's process better by knowing the characteristic of the module belonging to each region of B+, B-, and B0. This may lead to finding good- or bad-process for judgment of faultiness the module based on the program structure. By placing the Behavior view over the Product view, we can also analyze the debugger's program reading process better. For example, we can know whether the debugger's reading process is top-down or bottom-up, by examining the order of the module reading along program structure. This may lead to finding good- or bad-process for giving a priority of reading to each module.

For years, many researches have tried to understand how engineers comprehend programs during software maintenance [2][9][6]. In their studies, there is an assumption that engineers must comprehend the program wholly and in details. However, this assumption does not fit to usual debugging situations – debugging in the scheduled time. Comprehending the whole program can be regarded as the worst process for program reading in limited time. On the other hand, we focus on process of reading only a necessary part for the bug detection.

3. Experiment

In order to get some insights concerning good- and bad-reading process for debugging, we actually carried out an experiment to observe debugging processes. Based on the proposed model, we corrected quantitative data for each model view (product, decision, behavior). We have also listed up several hypotheses about reading process; then, tried to verify using corrected data.

3.1. Hypotheses of program reading process

We put following five hypotheses concerning reading process to be verified in this experiment:

-Hypothesis 1: Reading the modules that have no dependence with faulty module degrade the debug efficiency.

-Hypothesis 2: There is a module that most of experts commonly read well but most of novices do not.

-Hypothesis 3: Expert subjects spend more time to read the faulty module than novice subjects do.

-Hypothesis 4: Expert subjects rapidly narrow down the focus of module examination.

-Hypothesis 5: Expert subjects read modules along with control dependence paths.

-Hypothesis 6:Expert subjects read modules along with data dependence paths.

Hypothesis 2 can be verified by examining Behavior view only. Hypothesis 1, 5, and 6 can be verified by examining Product view and Behavior view. Hypothesis 4 can be verified by Decision view and Product view. Hypothesis 3 can be verified by Decision view and Behavior view.

3.2. Environment

The experiment was conducted using the Ginger2 CAESE environment [10]. This environment can record debugger's various activities such as eye-gaze point on the computer display, voice, key typing, and screen image. We also manually conducted the periodical interviews in order to trace the debugger's cognitive movement about the possible location of the bug.

3.3. Subjects

Ten subjects participated in the experiment and were assigned to debug the same program independently. All subjects are graduate school students. They can use C programming language. They have 3~4 years experience of programming and at least 2 years experience of C programming.

3.4. Target Programs

Two computer programs written in C language was prepared for this experiment:

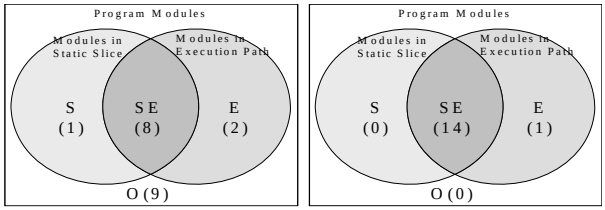
Program X(Calendar)

This program consists of about 300 lines/ 20 modules. This program is designed and coded to take the input of a date and to show a calendar of the date. There is a bug in module m17 (“ymd2rd2”) that produce the wrong out put of the date of a calendar.

Program Y (Tick-Tack-Toe)

This program consists of about 300lines/15modules. This program is designed and coded to play a game known as “tick-tack-toe”. This game uses 3x3 matrix where the computer and the user put marks by turns. The player, who succeeds to make three of his/her marks in a vertical/horizontal/slant line first, wins. There is a bug in module m9 (“check2moku3”) that the program fails to recognize two of opponent’s marks that are already in line, and therefore, it cannot prevent the opponent from winning.

As a summary of the Product view of each program, the modules of the target programs can be classified into four sets using static slicing as shown in Fig.5-(a),(b). In Program X, nine modules are contained in the slice (indicated as area S). Ten modules are contained in the execution trace (area E). Eight modules belong to both of the slice and the trace. Nine modules d o not belong to any of them (area O). The bug is located in one of the 8 modules in the “Slice & Execution” (area SE.) Program Y is classified into area SE and area E. Fourteen modules compose area SE. Area E consists of only one module.

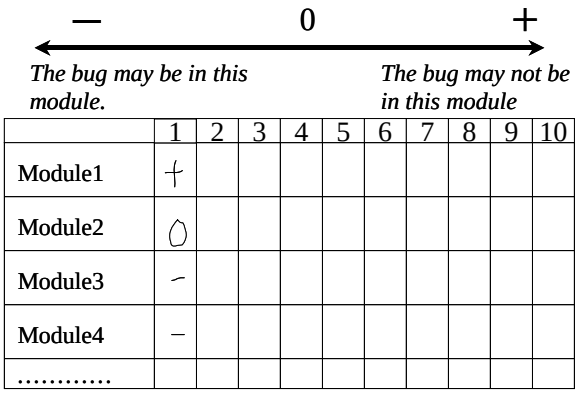


(a)Program X(calendar) (b)Program Y(tick-tack-toe)

Fig. 5 Product view summary of target programs

3.4. Procedure

Subjects were given the documentation and source code. At first, they were shown the program execution with error symptom to be fully understood. Then, they started to debug the program, but no directions about the debugging method were given. The experiment was performed until the bug is located and actually corrected. During the experiment, subjects had to answer the periodical interviews answering questionnaire every 5 minutes until the end of experiment. In every interview, the probability of the bug existence for each module was scored. If the subject thought that no bug exists in the module, he/she marks ‘-’ (minus) in the questionnaire. If the subject think that a bug may be included in the module, he/she marks ‘+’(plus.) If the subject has no confidence of bug existence, he/she marks ‘0’(zero.)(Fig.6)



	1	2	3	4	5	6	7	8	9	10
Module1	+									
Module2	0									
Module3	-									
Module4	–									
……										

Fig. 6 Interview sheet

We define some metric values by interview data. These metric values are supposed to have relation to the debugging efficiency.

Nb:Time duration from start time until the first time subject decided that faulty module is actually suspicious to have a bug.

max|B+|:Maximum number of modules in B+ (suspicious region) through entire the process.

avg|B+|:Average number of modules belonging to B+ (suspicious region) per interval.

|B+|n:Final number of modules belonging to B+ (suspicious region) just before the bug is located and fixed.

max|B-|:Maximum number of module in B- (trusted region) through entire the process.

avg|B-|:Average number of modules belonging to B- (trusted region) per interval.

|B-|n:Final number of modules belonging to B- (trusted region) just before the bug is located and fixed.

m+→-:Total number of modules, which were judged to be suspicious at once, and then judged again to be innocent through the entire process.

avg|N+→-|:Average time duration of misjudgement of non-faulty module to be suspicious.

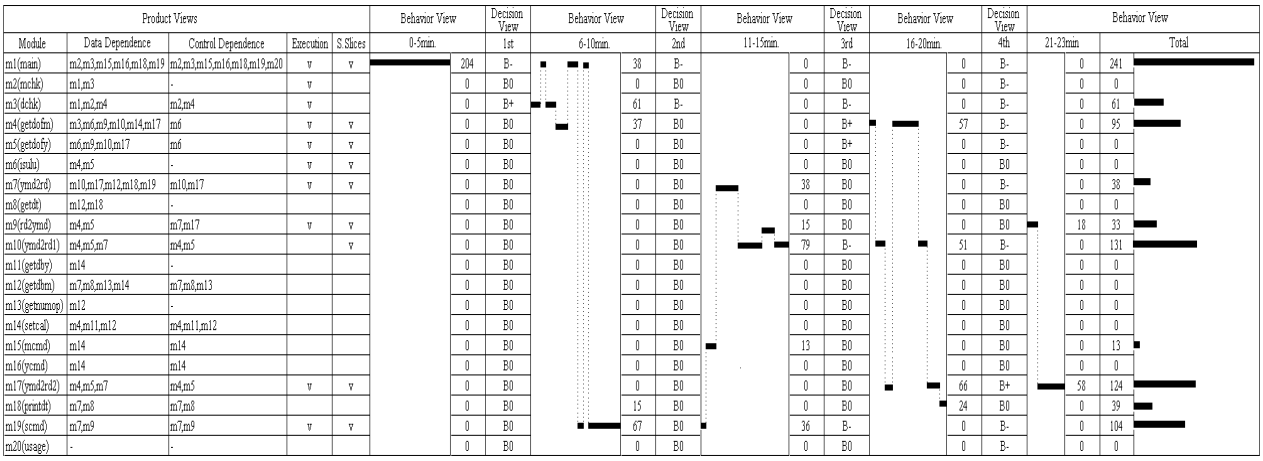


Fig.7 Example of detailed debugging process

m-→+:Total number of modules, which were judged to be innocent at once, and then judged again to be suspicious through the entire process.

All subjects successfully pointed the position of the bug and the correction was completed. Comparing the every subject's debugging time duration summarized in Table1, in case of Program X, subject X1 has the shortest debugging time as 23 minutes. In Table2, subject Y1 has the shortest debugging time as 21 minutes for Program Y.

Table1 Debugging time of ProgramX

	X1	X2	X3	X4	X5
Debugging Time(T minutes)	23	27	84	86	106
Interviews(N)	4	5	16	16	19

Table2 Debugging time of ProgramY

	Y1	Y2	Y3	Y4	Y5
Debugging Time(T minutes)	21	29	32	34	41
Interviews(N)	4	6	6	6	8

4. Results and Discussion

Fig.7 summarizes the collected data from subject X1 through the entire debugging process. In the leftmost row of the table, module names are enumerated. From the, Product Views, Behavior View, and Decision View are indicated corresponding to each module. In the leftmost row of the table, module names are enumerated. From the, Product Views, Behavior View, and Decision View are indicated corresponding to each module. In the data dependence cell of a module, modules that are data-dependent to that module are enumerated. In the control dependence cell of a module, modules that are called by that module are enumerated. If a module is included in the faulty execution path (i.e. that module is executed when the program produces faulty out put), asterisk ('v') is indicated in the "Execution" area. If a module is included in the static slice calculated from the failure point (i.e. where the program produces wrong out put value), asterisk ('v') is indicated in the "S. Slice" area. In the Behavior view area, order and duration of module reading in the form of timing-chart, with numeric value. In the Decision view area, categorization result of the module, judged by the subject as one of B+, B0 or B-. Six hypotheses were verified through the analysis based on these collected data.

- Hypothesis 1: Reading the modules that have no dependence with faulty module degrade the debug efficiency.

Table 3 shows reading time and its ratio to total debugging time, summarised based on Product views. Two subjects (X3 and X4) of three, who were worst in debugging time, have spent relatively longer time (19% and 18% respectively) to read for the modules in area O (i.e. belonging to none of static slice and faulty execution path). In contrast, two better subjects (X1 and X2) didn't spend much time on these modules (6% and 5% respectively). This result shows that the hypothesis1 is to be accepted.

- Hypothesis 2: There is a module that most of experts commonly read well but most of novices do not.

Table3 Reading time of each area (Program X)

	X1	X2	X3	X4	X5	
S	131	55	26	94	28	-0.57
(Static Slice)	(15%)	(5%)	(1%)	(3%)	(1%)	-0.77
SE	635	1079	2810	2336	3799	0.96
Slice&Execution	(72%)	(90%)	(75%)	(67%)	(91%)	0.02
E	61	14	189	414	38	0.44
(Execution)	(7%)	(1%)	(5%)	(12%)	(1%)	0.09
O	52	58	699	633	305	0.74
(Other)	(6%)	(5%)	(19%)	(18%)	(7%)	0.58

Table4 and 5 shows reading time of every module, reading time ratio to total debugging time, and correlation coefficient between them, for Program X and Program Y, respectively. Coefficient shows that the better debugger, who completed debugging in shorter time, have spent more time to read a certain module (ymd2rd1,check3moku3). For Program X, subject X1 and X2 spent 15% and 5% of their reading time, respectively. For Program Y, subject Y1 spend 18% of the reading time. After this analysis, we have examined these modules, and we found that they are functionally

Table4 Reading time of each module (a)ProgramX

	X1	X2	X3	X4	X5
main	241 (27%)	110 (9%)	421 (11%)	412 (12%)	324 (8%)
mchk	0 (0%)	0 (0%)	49 (1%)	26 (1%)	13 (0%)
dchk	61 (7%)	14 (1%)	140 0.037684	388 0.111631	25 0.005876
getdofm	95 (11%)	15 (1%)	356 (10%)	172 (5%)	116 (3%)
getdofy	0 (0%)	0 (0%)	12 (0%)	27 (1%)	0 (0%)
isulu	0 (0%)	35 (3%)	22 (1%)	71 (2%)	187 (4%)
ymd2rd	38 (4%)	11 (1%)	18 (0%)	165 (5%)	304 (7%)
getdt	0 (0%)	0 (0%)	0 (0%)	45 (1%)	42 (1%)
rd2ymd	33 (4%)	526 (4%)	912 (24%)	243 (7%)	786 (19%)
ymd2rd1	**131 (15%)**	**55 (5%)**	**26 (1%)**	**94 (3%)**	**28 (1%)**
getdby	0 (0%)	0 (0%)	45 (1%)	174 (5%)	11 (0%)
getdbm	0 (0%)	0 (0%)	0 (0%)	0 (0%)	25 (1%)
getnumop	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
setcal	0 (0%)	11 (1%)	121 (3%)	69 (2%)	0 (0%)
mcmd	13 (2%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
ycmd	0 (0%)	0 (0%)	175 (5%)	229 (7%)	133 (3%)
ymd2rd2	**124 (14%)**	**268 (22%)**	**668 (18%)**	**779 (22%)**	**1910 (46%)**
printdt	39 (4%)	46 (4%)	358 (10%)	63 (2%)	94 (2%)
scmd	104 (12%)	113 (10%)	400 (11%)	467 (13%)	173 (4%)
usage	0 (0%)	0 (0%)	0 (0%)	54 0.015612	0 (0%)

Table5 Reading time of each module (b)ProgramY

	Y1	Y2	Y3	Y4	Y5
main	121 (11%)	139 (12%)	91 (7%)	85 (5%)	119 (6%)
start	69 (6%)	209 (18%)	35 (3%)	241 (13%)	270 (13%)
battle	42 (4%)	198 (17%)	47 (4%)	151 (8%)	249 (12%)
man	26 (2%)	50 (4%)	14 (1%)	99 (5%)	145 (7%)
computer	112 (10%)	157 (14%)	267 (20%)	144 (8%)	443 (22%)
check2moku	70 (6%)	55 (5%)	61 (5%)	96 (5%)	61 (3%)
check2moku1	71 (6%)	8 (1%)	318 (24%)	226 (12%)	284 (14%)
check2moku2	106 (9%)	62 (5%)	68 (5%)	46 (2%)	97 (5%)
check2moku3	**136 (12%)**	**182 (16%)**	**337 (3%)**	**326 (18%)**	**237 (12%)**
check3moku	7 (8%)	14 (1%)	11 (2%)	115 (6%)	12 (3%)
check3moku1	90 (1%)	13 (1%)	22 (1%)	111 (6%)	52 (1%)
check3moku2	48 (4%)	0 (0%)	0 (0%)	95 (5%)	5 (0%)
check3moku3	**209 (18%)**	**0 (0%)**	**63 (5%)**	**64 (3%)**	**10 (0%)**
printbord	27 (2%)	32 (2%)	4 (0%)	3 (0%)	23 (1%)
initialize	9 (1%)	30 (2%)	4 (0%)	42 (2%)	11 (1%)

similar to actual faulty module (ymd2rd2 and check2moku3). Their names are also similar and they are referring the same external variables. This result seems to support the hypothesis 2, however, further study will be needed to clarify the detailed characteristics of the key module.

- Hypothesis 3: Expert subjects spend more time to read the faulty module than novice subjects do Continuing with table4 and 5, now focusing to the actual faulty modules (ymd2rd2 and check2moku3), just reading longer these faulty modules does not directly related to shorter debugging time. E.g., subject X4 and X5 spends 22% and 24% of their reading time for ymd2rd2 in Program X respectively. Also, subject Y3 and Y4 spends 25% and 18% of their reading time for check2moku3 in Program Y respectively. This result seems to reject the hypothesis 3. I.e., reading the faulty (or most suspicious) module only won't generally lead to shorter debugging time.

- Hypothesis 4: Expert subjects rapidly narrow down the focus of module examination.

Table6 and 7 summaries the decision view of the debugging process collected by interviews. Subjects who took longer debugging time have larger avg|B+|. Especially, the worst subjects in both Program X and Program Y have the largest values. This data suggest that novice debuggers take longer time to narrow down the suspicious region (i.e. they have difficulty to locate the bug position).

Subjects who took longer debugging time, had larger difference between N and Nb: three subjects (X3, X4, X5) for Program X, and subject Y5 for Program Y, had N more than twice of Nb. I.e., they took longer time from getting suspicious to faulty module until actually removing the bug.

These results suggest, at least, the hypothesis 4 was not rejected because novice debuggers actually seem to have difficulty to locate the position of the bug.

Table6 Result of the Interviews(ProgramX)

	X1	X2	X3	X4	X5
Interviews(N)	4	5	16	16	19
Nb	4	4	5	5	4
max|B+|	2	3	4	3	5
avg|B+|	1	1.4	1.8	1.5	3.9
|B+|n	1	1	2	1	4
max|B-|	9	19	19	19	15
avg|B-|	4	9.6	9.6	9.5	9.7
|B-|n	9	19	18	9	15
m+→-	3	2	7	3	2
avg|N+→-|	1	3	5	7.9	11.6
m-→+	0	0	4	14	2

Table7 Result of the Interviews(ProgramY)

	Y1	Y2	Y3	Y4	Y5
Interviews(N)	4	5	6	6	8
Nb	3	5	6	5	4
max|B+|	2	1	1	5	7
avg|B+|	1.25	0.33	0.17	1.3	4.4
|B+|n	2	1	1	2	6
max|B-|	13	10	14	10	10
avg|B-|	9.5	6.8	9.5	4.7	7.5
|B-|n	13	10	14	10	9
m+→-	0	0	0	1	2
avg|N+→-|	-	-	-	1.2	5
m-→+	0	0	0	0	2

- Hypothesis 5: Expert subjects read modules along with control dependence paths.
- Hypothesis 6:Expert subjects read modules along with data dependence paths.

Table8 shows the ratio of existence of control/data dependence between current reading module and next reading module. Although subject X1 has high tendency to follow control/data dependence, other subjects don't show such tendency. Thus, we cannot accept this hypothesis for now.

Table8 Reading time of subject X1--X5

	X1	X2	X3	X4	X5	
Control Dependence	55%	35%	32%	26%	33%	-0.69
Data Dependence	50%	30%	25%	26%	26%	-0.72

5. Conclusion

In this paper we proposed a model for analyzing the reading process in debugging. The model provides three views for representing human activities: Product view for presenting structural properties of target program, Decision view for representing human's cognitive image of the possible location of the bug, and Behavior view for representing externally observed human's action. By using above three views, analysis of debugging processes can be done in clear way, by examining interactions and relationships among these views.

We have conducted experiments to observe debugging processes and collected process data based on our model. Several hypotheses were verified with collected data based on the model. Some hypotheses such as "Expert subjects spend more time to read the faulty module than novice subjects do." were actually rejected; and, some other hypotheses such as "Reading the modules that have no dependence with faulty module degrade the debug efficiency." were accepted. The result of experiment suggested that explicit and quantitative evaluation of debugging process become possible by using the proposed model.

Although some of our hypotheses suggested the candidates of good- and/or bad-reading process, we have not clarified the useful process for actual debugging yet. We need to employ more programs, bugs, and subjects in the future experiments to clarify more useful process. However, we believe our analysis model is a powerful tool for seeking for the quantitative and objective debugging strategies, which was very difficult in the past researches.

Acknowledgment

This study was financially supported by the Proposal-based New Industry Creative Type Technology R&D Promotion Program from the New Energy and Industrial Technology Development Organization (NEDO) of Japan.

References

[1] K. Araki, Z. Furukawa and J. Cheng, A general Framework for Debugging, IEEE Software, 18 (1991) 14-20.

[2] T. J. Bigerstaff, B. G. Mitbander and D. Webster, The Concept Assignment Problem in Software Understanding, Proceedings of 15th International Conference on Software Engineering (1993) 482-497.

[3] K. B. Gallagher and J. R. Lyle, Using Program Slicing in Software Maintenance, IEEE Transactions on Software Engineering, 17 (1991) 751-761.

[4] K. Iio, Y. Arai and T. Furuyama, Cognitive Process Analysis based on the Tendency to the Module Programmers View, Technical Report of JSAI, SIG-KBS-9402-2 (1994) 9-16 (in Japanese)

[5] J. Koenemann and S. P. Robertson, Expert problem solving strategies for program comprehension, Proceedings of Human Factors in Computing Systems (1992) 125-130.

[6] A. Von Mayrhauser and M. Vans, Program Comprehension During Software Maintenance and Evolution, Computer, 28 (1995) 44-55.

[7] A. Nishimatsu, K. Nishie, S. Kusumoto and K. Inoue, An Experimental Evaluation of Program Slicing on Fault Localization Process, IEICE Transactions, 582-D-I, (1999) 1336-1344.

[8] E. Regelson and A. Anderson, Debugging practices for complex legacy software systems. Proceedings of International Conference on Software Maintenance, (September 1994) 137-143.
[9] M. A. D. Storey, K. Wong and H. A. Muller, How Do Program Understanding Tools Affect How Programmers Understand Program, Proceedings of the Fourth Working Conference on Reverse Engineering (1997) 12- 21.
[10] K. Torii, K. Matsumoto, K. Nakakoji, Y. Takada, S. Takada and K. Shima, Ginger2: An Environment for Computer-Aided Empirical Software Engineering, IEEE Transactions on Software Engineering, 25 (July/August 1999) 474-492.
[11] S. Uchida, H. Kudo and A. Monden, An experiment and an Analysis of debugging process with periodic interviews, Proceedings of Software Symposium '98, (1998) 53-58 (in Japanese).
[12] S. Uchida, A. Monden, H. Iida, K. Matsumoto, K. Inoue and H. Kudo, Debugging process models based on changes in impressions of software modules, Proceedings of International Symposium on Future Software Technology 2000, Guiyang, China, (Aug. 2000), 57-62.
[13]S. Uchida, A. Monden, H. Iida, K. Matsumoto, and H. Kudo, A multiple-view analysis model of debugging processes, Proceeding of International Symposium on Empirical Software Engineering (ISESE2002), IEEE Computer Society Press, Nara, Japan, (Oct. 2002) 139-147.
[14] I. Vessey, Expertise in debugging computer programmers : A process analysis, International Journal of Man-Machine Studies, 23 (1985) 459-494.
[15] M. Weiser, Program slicing, Proceedings of 5th International Conference on Software Engineering, (1981) 439-449.
[16] M. Weiser, Programmers use slices when debugging, Communications of the ACM, 25, (1982) 446-452.