# A Multiple-View Analysis Model of Debugging Processes

Shinji Uchida[†], Akito Monden[††], Hajimu Iida[††], Ken-ichi Matsumoto[††] and Hideo Kudo[†††]
[†]Electrical and Information Engineering, Kinki University Technical College,
[††]Graduate School of Information Science, Nara Institute of Science and Technology,
[†††]Information Engineering, Nara National College of Technology
uchida@ktc.ac.jp,{akito-m@is,iida@itc,matumoto@is}.aist-nara.ac.jp,kudoh@info.nara-k.ac.jp

## Abstract

This paper proposes a model for analyzing the reading strategies in software debugging. The model provides quantitative and objective visions to a human's debugging activity, and provides the framework for clarifying good- and/or bad-strategies for program reading. We have conducted a case study to observe the debugging processes under a controlled environment. The observation includes: Both novice debugger and expert debugger could correctly locate an area that seems to have a bug, however, only the expert subject could quickly narrow down that area, reading the faulty (or most suspicious) module only will not generally lead to a shorter debugging time, and the most well-performed subjects read the module that seems to be a key to find a fault. This case study suggested that explicit and quantitative evaluation of the debugging process becomes possible by using the proposed model.

## 1. Introduction

Recently, software maintenance groups often need to debug a program that was not originally developed by them because many of today's engineers move rapidly between companies and job assignments [12]. 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 a case where the program is new to the engineers. In addition, reuse activities in modern software development also force engineers to read and find bugs in a 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 [13].

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 the 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 the good- and bad- reading strategies for debugging unfamiliar programs. Our approach is to analyze debugging tasks under 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 the subjects' intentions – "Why did they read?" In our previous researches [13][14], we have observed that strategies to select a module (= an area that should be comprehended) strongly relates to the engineers' impressions of each module – either the module is faulty, not faulty, or uncertain. Therefore, in order to clarify reading strategies, we need to follow up on the engineer's cognitive image of a program that represents a faulty area, unfaulty area, and uncertain area. Moreover, the structural properties of the target program should be taken in account as well. Some past research states that the static slice of a program should be considered in debugging [3][16][8].

We propose a new modeling schema: Multiple-View Analysis Model of Debugging Process. This model provides two kinds of view to program reading in the debugging process: Product view and Cognitive view. The Product view represents a target program's module structure and properties, while the Cognitive view presents the property of a human's activity. In order to analyze the unclear part of a human's activities in a clear and well-defined way, the target program structure is set as a basement of analysis framework. Human activities such as movement of a reading target or module classification are, then, cast over the Product view. The Cognitive view actually consists of the 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 the three views (Product view, Decision view, and Behavior view), the analysis of the debugging processes may be done in clear way, by examining the interactions and relationships among these views.

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

In the rest of this paper, Section 2 describes the details of our analysis model. Section 3 describes a case study to analyze the debugging processes based on the proposed model. Section 4 describes the results 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} .

$$P = \{m1, m2, \dots, mn\}$$

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

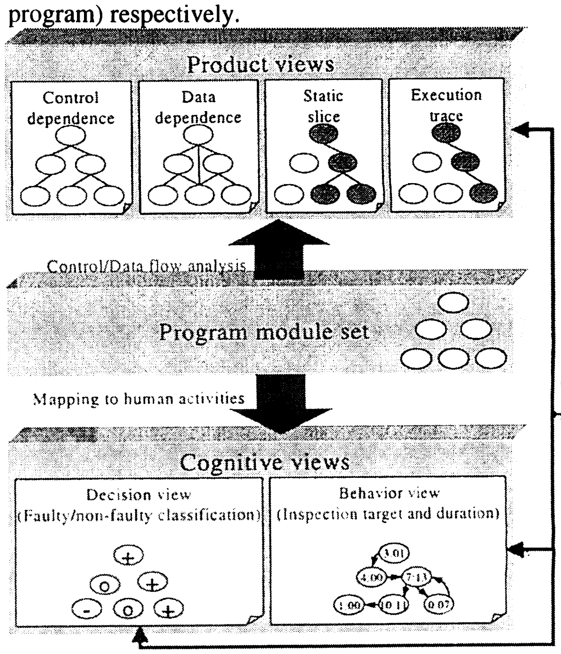

Product views
Control dependence
Data dependence
Static slice
Execution trace
Control/Data flow analysis
Program module set
Mapping to human activities
Cognitive views
Decision view (Faulty/non-faulty classification)
Behavior view (Inspection target and duration)
Find Co-relationships

**Fig.1 Multi-View Analysis Model for Debugging Process**

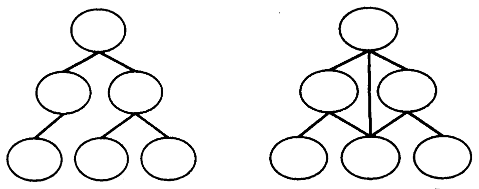
### 2.1. Product view

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

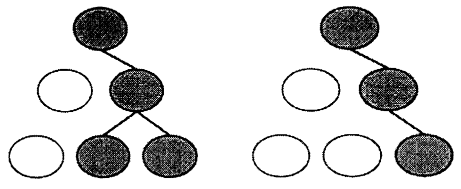
- Control dependence between modules (in this case, this is equivalent to a module call structure) (Fig. 2-(a)),
- Data dependence between modules (Fig. 2-(b)),
- Static slice [13][14] calculated from the wrong output value (Fig. 2-(c)), and
- Faulty execution path (Fig. 2-(d)).

The 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 a slice for the actual debugging. We use the static slice just for the process analysis.

As a summarization of the Product view, modules are categorized in the 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 the 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 the 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 actions: 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 have no bug. The third region is the neutral region (shown as B0), which is the set of modules that are uncertain whether they contain the bug or not (Fig 3). The 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 the transitions of the status of the human internal impression S(t), which is represented with actual assignments to B+, B0, and B-.

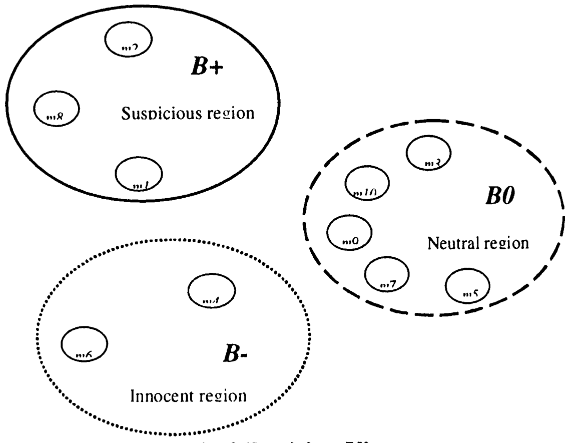


**Fig.3 Decision View**

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, the 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, the transition of the human decision can be formally expressed, and individual differences of the decisional transitions can be observed.

Since transitions are performed inside the debugger's mind, the actual decision movements cannot be observed externally. We use periodical simple interviews of the developer to get an impression of each module. During the experiment, the subjects had to answer the periodical interviews, answering a questionnaire every 5 minutes until the end of the experiment (Fig. 4). 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 thinks that a bug may be included in the module, he/she marks '+'(plus.) If the subject has no confidence of the bug existence, he/she marks '0'(zero.)

□[ 0 □{

*The bug may be in this module.* *The bug may not be in this module*

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

**Fig. 4 Interview sheet**

## 2.3. Behavior view

The behavior view can represent the external reading action of a debugger. In this view, the 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 5). 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 the 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 do not read all of the modules for debugging, and sometimes there is an implicit or explicit strategy for the module choice [5]. Moreover, a skilled programmer's patterns of movement over the modules might be significantly different from a novice programmer's patterns. This view is also capable of investigating such skills differences [4].

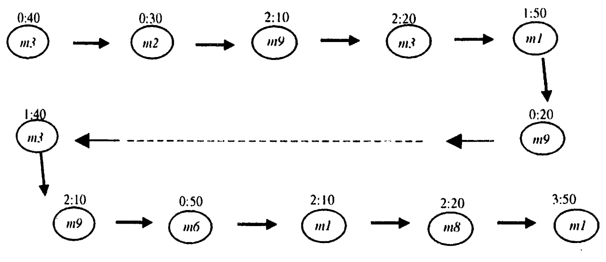


**Fig.5 Behavior View**

## 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 a 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 the resulting analysis is also highly subjective. For example, in Araki's model, the debugging process is explained as iterations of the debugger's hypothesis evolution, which cannot be observed quantitatively [1]. In Vessey's model, the debugging process is expressed based on the debugger's chunking ability [15]. However, these models cannot illustrate the objective activities of debuggers'.

In our model, the multiple-view architecture is provided for analyzing the 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 the debugger's internal recognition and the program structure. For example, we can evaluate the debugger's strategy 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-strategies for judgment of the faultiness in 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 strategy is top-down or bottom-up, by examining the order of the module reading along the program structure. This may lead to finding good- or bad-strategies for giving a priority of reading to each module.

For years, many studies have tried to understand how engineers comprehend programs during software maintenance [2][6][7][12]. In their studies, there is an assumption that engineers must comprehend the program wholly and in detail. However, this assumption does not fit with usual debugging situations – debugging in the scheduled time. Comprehending the whole program can be regarded as the worst strategy for program reading in a limited time. On the other hand, we focus on strategies of reading only the necessary part for the bug detection. In addition, [9] reported that their experiment showed the programmers' skill level and existence of a line number in the Pascal program are the factors that affect the debugging performance, and programmers could almost always correct an error once it is located. While their approach uses a large number of subjects but does not analyze the debugging processes, our approach uses fewer subjects but analyzes the debugging processes in detail.

## 3. Case study

In order to get some insights concerning a good- and bad- reading strategy for debugging, we actually carried out a case study to observe debugging processes under a controlled environment. Based on the proposed model, we collected quantitative data for each model view (product, decision, behavior).

### 3.1. Environment

The case study was conducted using the Ginger2 CAESE environment [12]. This environment can record the debugger's various activities such as the eye-gaze point on the computer display, voice, key typing, and screen image. We also manually conducted periodical interviews in order to trace the debugger's cognitive impression on the potential location of the bug.

### 3.2. 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.

### 3.3. Target Programs and bugs

Two computer programs written in C language were 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.

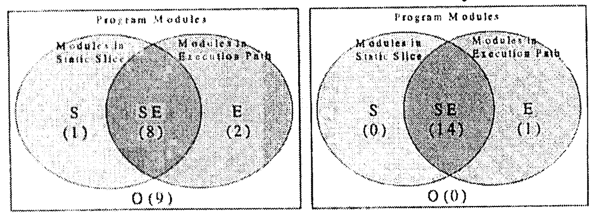
**- Program Y (Tick-Tack-Toe)**

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.6-(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 m odules 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.

### 3.4. Procedure and Collected Data

The subjects were given the documentation and source code. At first, they were shown the program execution with the 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 was located and actually corrected. We define some metric values by interview data. These metric values are supposed to have a relation to the debugging efficiency. Fig.7 summarizes the collected data from subject X1 through the entire debugging process. In the leftmost row of the table, the module names are enumerated. From the left to right, Product Views, Behavior View, and Decision View are indicated corresponding to each module.



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

**Fig. 6 Product view summary of target programs**

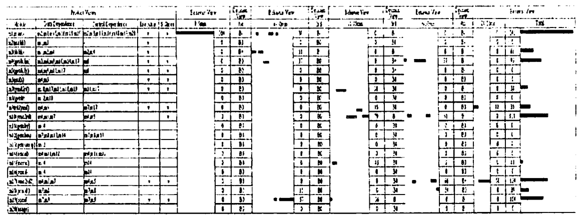


**Fig.7 Example of detailed debugging process**

## 4. Results and Discussion

### 4.1. Overview of result

#### 4.1.1. Debugging time

All the subjects successfully located the position of the bug and the correction was completed. Table 1 and 2 show the debugging time (the time required for finding a bug) of each subject and program. In program X, subject X5 required more than 4 times as much debugging time as X1. In Program Y, subject Y5 required about twice as much time as Y1. One of our concerns here is why X5 required so much time to locate the bug (and why X1 was so fast).

**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 |

#### 4.1.2. Decision View

Here we define some metrics concerning decision view. These metric values are supposed to have a relation to the debugging efficiency.

Nb:Time duration from the start time until the first time subject decided that the 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.

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

Table3 and 4 summaries the decision view of the debugging process collected by interviews. Subjects who have a longer debugging time have a larger avg|B+|. Especially, the worst subjects in both Program X and Program Y have the largest values. This data suggests that novice debuggers take a longer time to narrow down the suspicious region (i.e. they have difficulty locating the bug position).

**Table3 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 |

**Table4 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 |

It is a little surprising to us that there is no tendency that the subjects who have a longer debugging time have large values of Nb. Indeed, the values of Nb are almost the same in Program X. This indicates that both the novice debugger and expert debugger can correctly locate an area that seems to have a bug, however, only the expert subject can narrow down that area.

### 4.1.3. Behavior View

Table 5 shows the reading time and its ratio to the total debugging time, summarized and based on the Product views. All the subjects spent more time in reading area SE than area S, E, and O. In Table 5, ymd2rd2 in program X and check2moku3 in program Y are the bug-located modules. Subject X5, who required the longest time to find a bug, 46% of his time in reading ymd2rd2 (containing a bug) while X1 spent only 14% of his time. Therefore, reading the faulty (or most suspicious) module only will not generally lead to the shorter debugging time.

In both of the programs (X and Y), the most well-performed subject (X1 and Y1) read the module that seems to be a key to finding a fault (ymd2rd1 and check3moku3). Actually, Subject X1 spent 131 sec. in reading ymd2rd1, while the other subjects spent at most 94 sec. in reading this module though they had more time in the total debugging time. Similarly, subject Y1 spent 209 sec. in reading check3moku3, while the other subjects spent at most 64 sec. in reading check3moku3. These key modules (ymd2rd1 and check3moku3) do not contain a bug, but both of them have similar characteristics as below:

- The name of these modules (functions) resembles that of the bug-included module.

Actually, ymd2rd1 and ymd2rd2 have a similar function. Also, check2moku3 and check3moku3 have a similar function.

- These modules refer to the global variable that is also referred in the bug-included module.

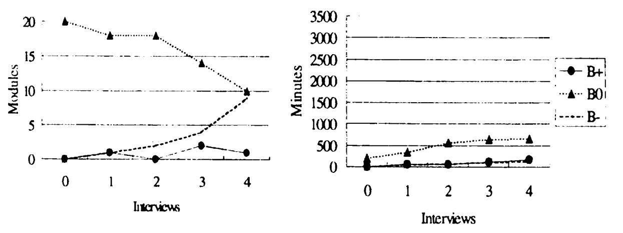
One possible interpretation is that reading these key modules helped in understanding the functionality of the bug-included modules.

**Table5 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%) |



**(a)Decision View (b)Behavior View**
**Fig. 8 Subject X1**

## 4.2. Individual processes

In this subsection, we pickup some subject and analyze their process in detail.

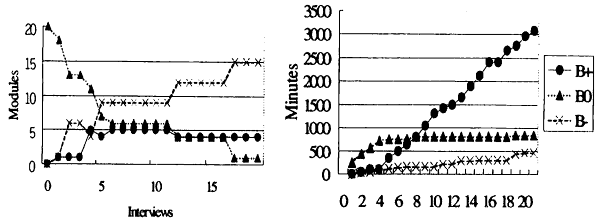
### 4.2.1. Program X

**Subject X1:**

This subject has the shortest debugging time in program X. Fig.8-(a) shows the Decision view of subject X1. The number of modules belong to B+(Suspicious region) is few in all the interviews. Moreover, the number of modules belong to B-(Innocent region) is increasing gradually. Fig.8-(b) shows the Behavior view (accumulated reading time) of subject X1. This subject reads the modules belonging to B0(Neutral region) in the first half of the debugging. Then, he reads the modules belonging to B+(Suspicious region) in the second half of the debugging. So, we may be able to say that this subject could narrow down the area containing a bug.

**Subject X5:**

This subject has the longest debugging time in program X. Fig.9-(a) shows the Decision view of subject X5. The number of the modules in B+(Suspicious region) could not be lessened. The number of the modules in B-(Certified region) is increasing gradually as the debugging process progresses. Fig.9-(b) shows the Behavior view of subject X5. This subject reads the modules belonging to B0(Neutral region) until the fifth interview. Then, he mostly reads the modules belonging to B+(Suspicious region) in the second half of the debugging. He persists in reading the modules belonging to B+(Suspicious region) too much.



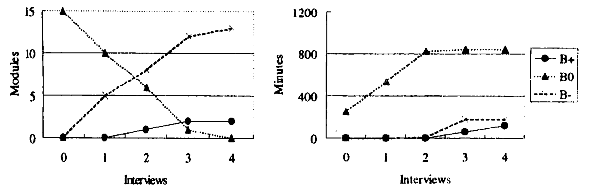
**(a)Decision View (b)Behavior View**

**Fig. 9 Subject X5**

### 4.2.2. Program Y

**Subject Y1:**

This subject has the shortest debugging in program Y. Fig.10-(a) shows the Decision view of subject Y1. The number of modules belong to B+(Suspicious region) is few in all interviews. Moreover, the number of modules belong to B-(Innocent region) is increasing gradually. Fig.10-(b) shows the Behavior view of subject Y1. This subject reads the modules belonging to B0(Neutral region) until the second interview. Then, he reads only the modules belonging to B+(Suspicious region) and B-(Innocent region) until the third interview. So, we may be able to say that this subject could narrow down the area containing a bug.

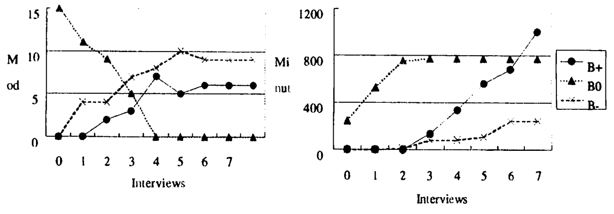


**(a)Decision View (b)Behavior View**

**Fig. 10 Subject Y1**

**Subject Y5:**

This subject has the longest debugging time in program Y. Fig.11-(a) shows the Decision view of subject Y5. He classifies the modules belonging to B0(Neutral region) into B+(Suspicious region) and B-(Certified region) at the fourth interview. Then, the number of the modules in B+(Suspicious region ) could not be made small. Fig.11-(b) shows the Behavior view of subject Y5. This subject reads the modules belonging to B0(Newtral region) until the second interview. Then, he reads only the modules belonging to B+(Suspicious region). He persists in reading the modules belonging to B+(Suspicious region) too much.



**(a)Decision View (b)Behavior View**

**Fig. 11 Subject Y5**

## 5. Conclusion

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

We have conducted a case study to observe the debugging processes and collected process data based on the model. The observation includes:

- Subjects who have a longer debugging time have a larger avg|B+|. This data suggests that novice debuggers take a longer time to narrow down the suspicious region.
- There is no tendency that subjects who have a longer debugging time have large values of Nb (Time duration from the start time until the first time the subject decided that the faulty module is actually suspicious to have a bug.) This indicates that both the novice debugger and expert debugger can correctly locate an area that seems to have a bug, however, only expert subject can narrow down that area.
- All the subjects spent more time in reading area SE than area S, E, and O. This follows previous research that program slicing is useful in debugging.
- Reading the faulty (or most suspicious) module only will not generally lead to the shorter debugging time.
- The most well-performed subject (X1 and Y1) read the module that seems to be a key to finding a fault (ymd2rd1 and check3moku3).

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

## 6. 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.

## 7. 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 Von Mayrhauser and M Vans, Program Understanding Behavior During Debugging of Large Scale Software, Empirical Studies of Programmers (1997) 157-179.

[8] 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.

[9] Paul W. Oman, Curtis, R. Cook, and Murthi Nanja, Effects of programming experience in debugging semantic errors, The Journal of Systems and Software 9, (1989), 197-207.

[10] E Regelson and A Anderson, Debugging practices for complex legacy software systems. Proceedings of International Conference on Software Maintenance, (September 1994) 137-143.

[11] 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.

[12] 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.

[13] 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).

[14] 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.

[15] I Vessey, Expertise in debugging computer programmers : A process analysis, International Journal of Man-Machine Studies, 23 (1985) 459-494.

[16] M Weiser, Program slicing, Proceedings of 5th International Conference on Software Engineering, (1981) 439-449.

[17] M Weiser, Programmers use slices when debugging, Communications of the ACM, 25, (1982) 446-452.