Pilot Studies on Debugging Oracle Assistants

Eugene H. Spafford *Chonchanok Viravan*

Software Engineering Research Center
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907–1398

April 27, 1993

Abstract

A debugging oracle is a decision maker during a debugging process. Three major decisions during typical debugging sessions are on the identities, the locations, and the repairs of faults. A programmer usually acts as a debugging oracle. Our research objective is to help him in his decision-making process with a debugging oracle assistant.

To enhance our understanding of both the debugging oracle and the debugging oracle assistant, we studied how 14 expert programmers debug a C program with over 4300 executable lines of code including real faults of omission. Four different forms of debugging oracle assistance were tested. The outcome of the studies provides insight to programmers' needs and the forms of assistants which fulfill them.

We find that information alone does not improve debugging performance. The two assistants that helped programmers make more accurate decisions on faults observed when programmers needed help and provided unsolicited and customized assistance for each programmer. This customized assistance came in the form of hints, questions, confirmation, and/or explanation.

Our preliminary results are supported by research on Decision Support System (DSS) and Critic systems. The problems with debugging assistants we identified match the problems identified for DSS. The desirable features to improve DSSs match the desirable features of debugging oracle assistants. Because these features are also the characteristics of Critic systems, we have reason to believe that a desirable debugging oracle assistant is a debugging critic.

Contents

1 Introduction **3**
- 1.1 The Objective of the Study . . . 3
- 1.2 Initial Plan of Study . . . 3

2 Pilot Study 1 **7**
- 2.1 The Study . . . 7
- 2.2 The Results . . . 7
- 2.3 The Findings . . . 7
  - 2.3.1 About the programmers' needs . . . 8
  - 2.3.2 About the assistant . . . 8

3 Pilot Study 2 **9**
- 3.1 The Active Oracle Assistant for Nu . . . 9
- 3.2 The Study . . . 9
- 3.3 The Results . . . 9
- 3.4 The Findings . . . 10
  - 3.4.1 About the programmers' needs . . . 10
  - 3.4.2 About the assistant . . . 13

4 Pilot Study 3 **15**
- 4.1 The Information-only Assistant . . . 15
- 4.2 The Observation-Information-Question Assistant . . . 17
- 4.3 The Study . . . 18
- 4.4 The Results . . . 19
- 4.5 The Findings . . . 19
  - 4.5.1 About the programmers' needs . . . 19
  - 4.5.2 About the assistants . . . 20

5 Summary **21**

6 Related Work **23**

7 Future Work **25**

8 Acknowledgment **26**

A The Programmers **28**

B The Measurements **29**

C The Results **31**

1 Introduction

Debugging, a process of locating and fixing program faults, is one of the most serious bottlenecks in software development today [Agr91]. *Program faults* are physical evidence of *errors*; errors are inappropriate actions made during software development that may ultimately cause software to fail [IEE83]. Program testing is a conventional means of recognizing the presence of faults.

Our goal is to improve the decision-making process during debugging to reduce programmer debugging time and improve accuracy. We refer to the decision maker in a debugging process as a *debugging oracle*. Though three major decisions in debugging are on the identity, the location, and the repair of faults, we previously placed emphasis on supporting decisions on fault locations only [SV92].

In [SV92], we originally decided that a *debugging oracle assistant* should support two intermediate decisions within fault localization strategies (e.g., as in [Sha83, PS93]). They are the decisions on correctness of program parts or program states (e.g., data values). We proposed to find the information that helps programmers decide on correctness in the absence of a formal program specification.

The debate on whether our proposed role of a debugging oracle assistant is adequate to improve the debugging time and accuracy prompted us to conduct pilot studies. We tested four different assistants and used statistical analysis to suggest whether the assistant improved debugging performance. Though our findings did not match our expectations, they match the findings in the area of Decision Support Systems and Critic systems. As a result, we have revised the roles of a debugging oracle assistant and redefined our research hypothesis.

This paper covers, in chronological order, our hypotheses, studies, analyses, and findings of our pilot studies. After we summarize our findings, we describe evidence that supports our results from research in decision support systems and critic systems. In the conclusion, we suggest a list of possible directions to pursue in debugging research.

1.1 The Objective of the Study

The objective is to test whether our original research hypothesis is worth pursuing. Our original hypothesis was *the presence of appropriate information helps programmers judge the correctness status of hypothesized fault locations or program states significantly faster or more accurately.* We wanted to conduct studies to see:

1. how many times programmers asked for help on the above two tasks,
2. whether the high demand on both tasks are related to improved debugging performance, and
3. whether information alone can assist these tasks.

If we found evidence against these three, we hoped that the results of the studies would point out other desirable features for a debugging oracle assistant that are worth pursuing.

1.2 Initial Plan of Study

Instead of studying how novice programmers debug a toy-size program with seeded faults, we want to study how a debugging oracle assistant helps expert programmers debug a large program with real faults. Because we do not yet know what information can help, we need an assistant that can provide practically any information about the program.

The **debugging oracle assistant** that qualifies is the debugging oracle of the program. This is the person who wrote or maintains the program and can answer almost any question about it.

The **program** we used is *Nu*, a locally-developed Unix system administrator program for maintaining a user database. Nu is a screen-oriented program for adding new users, deleting old users and modifying information about existing users on departmental hosts. Information is maintained for all hosts, including diskless clients.

Nu's C source code consists of one header file and 16 source files. Version 1 of Nu has 6795 lines – 4320 of which are executable. Version 2 of Nu has 6771 lines – 4303 of which are executable. Blank lines constitute 12.6% of total lines. Comment lines constitute 6.9% of the non-blank lines. Approximately 17.6% of the non-blank lines use #define's values. Nu consists of 167 routines with an average length of 40 non-blank lines. The average identifier is 8.9 characters long. It uses 18 goto's (13 of them are in the mail.c file), 261 unique operators, and 1460 unique operands. Nu maintains five database files that amount to approximately 1600 lines of data.

Two **faults** under study are faults of omission. For fault 1, a data definition is missing. During the start up phase, the pointer *pde* is not defined when one of the fields in the *chfnadd* file is blank. The reference to this undefined variable causes the program to dump core. For fault 2, a data handling task is missing. The pointer $Home_dir_list \rightarrow pde$ is left pointing to a copy of a password database entry. Because it does not get reset to the original entry before the program frees the copy, the same space is written over. Ultimately, the program produces the wrong output. Figure 1 describes the characteristics of both faults in more detail.

One **error-revealing test case** per fault is given. For fault 1, the error-revealing data is in the 4-line file, *chfnadd*. Chfnadd contains finger information and password changes. Its format is as follows:

```
uid:optional_new_passwd:option_new_GECOS:time_of_change:
```

The empty GECOS field in the first line of chfnadd, "105:::714362704:", causes Nu to dump core. This file format is not given to the programmers.

For fault 2, the error-revealing test data comes from an interactive session of Nu which adds new home directories for multiple users. A sample test data (in the README file) contains two transactions. They are as follows:

$\wedge F$ *trinkle* $RETURN$ $\wedge E \wedge E$ *ah escher* $RETURN$ */u/u35* $RETURN$ $\wedge T \wedge T$ $\wedge F$
norman $RETURN$ $\wedge E \wedge E$ *ah arthur* $RETURN$ */u/u1* $RETURN$ $\wedge T \wedge T \wedge T$

The $\wedge$ represents a control key. The $\wedge F$ is a request to find the specified user. The first $\wedge E$ is a request for editing the entry. The second $\wedge E$ is a request for editing the home map entry. The ah is a request for adding a host specific entry. The first $\wedge T$ saves the map entry. The second $\wedge T$ saves the user entry. After all transactions the last $\wedge T$ saves all the changes and exits nu. The erroneous output is as follows:

```
/usr/ucb/rsh arthur ./install /u/u1 norman 147 147
/usr/ucb/rsh escher ./install /u/u35 norman 147 147
/var/amd/updatemaps
Saving database ... done
```

The correct output should have listed *trinkle 143 143* in the second line instead of *norman 147 147*.

The **programmers** must satisfy six requirements to be considered expert programmers for our studies. First, they must have at least six years of programming

Faults	**#1**	**#2**
Type	Missing data definition	Missing data handling task
Failure	Core dump	Wrong output
Error-revealing test data	Empty GECOS field in chfnadd file	Add home directories for multiple users
Chain of Causes of Failures	Reference bad pde pointers in mark_ml() ↑ Pass undefined pde to mark_ml() from get_chfn_info() ↑ Missing definition for pde when GECOS field is empty	Overwrite the same memory location in add_home_dir() ↑ Dangling pointers in home_dir_list in edit() ↑ Forget to reset pde in home_dir_list to point to the original entry before freeing the memory of copy of pde
Faulty routine	get_chfn_info()	edit()
Calling level	1	3
Correct fixes	- Initialize pde before calling mark_ml() - Do not call mark_ml() when pde is uninitialized	- Call change_home_dir() before freeing pde copy

Figure 1: The characteristics of the faults used in our pilot studies

experience. Second, they must have known C for at least five years. Third, they must have spent at least three years in the graduate school in the Computer Science department at Purdue University. Fourth, they must have taken at least three classes that used C language. Fifth, they must have coded C programs larger than one thousand lines before. Sixth, they must know how to use dbx.

The **working environment** requires the programmers to run Nu under SunOS version 4 in the X window environment. They can use only one debugger, dbx.

The **materials** given to the programmers include the source code listing of nu, nu's data files, a README file, and an instruction sheet. Our README file lists the purpose of the data files, the nature of program failure, and one error-revealing test case. Our Makefile is set up to compile the program in DEBUG mode. Lastly, our instruction sheet explains how to obtain these materials on-line. It also covers extra instructions for the programmers to report their progress.

The **monitoring process** monitors the accuracy, the time, the debugging process, and the oracle-programmer interaction. To monitor the accuracy, we asked the programmers to answer our *AboutBug* questionnaire at the end of each hour. The questionnaire asks (1) where they suspect the fault is, (2) where they think the fault cannot possibly be, (3) what the fault is, and (4) how the fault can be fixed. The answer to the first two questions must be as specific as possible: by file names, routine names, and line numbers. They must be mutually exclusive. Anything else falls into a region where programmers may look if they cannot find the bug in the suspected region. The answer to the third question is either the description of the fault or types of faults they suspect. The answer to the fourth question is either "do not know yet" or the code with correction.

To monitor the time and the debugging progress, we use script, tcsh, and RCS. The *tcsh* is a C shell that monitors the time of day that the commands were used. The history command in tcsh reports both the previous commands and their time stamps. *RCS* (Revision Control System) monitors changes made to the source code. The Makefile hides this operation from the programmers by automatically checking the modified files in and out of the RCS directory.

To monitor the oracle-programmer interactions, we taped their conversations. Our tape recorder also records the time of the conversation.

The **debugging performance measurements** we used include (1) the actual time (TIME), (2) the estimated time taken to fix the fault (ETIME), (3) the debugging speed (SPEED), (4) the accuracy (AC), and (5) the average accuracy in locating the fault (AACLOC). AC is calculated based on the accuracy in locating the fault (ACLOC), in identifying the fault (ACID), and in fixing the fault (ACFIX). AACLOC is the average of ACLOC at the end of each hour. Appendix B lists the definitions of these measurements.

The **statistical analysis** we used to interpret the results are Analysis of Variance (ANOVA) and contrast analysis. ANOVA helps us identify causes of variations in the debugging performance by testing the differences in the means of a measurement among two or more groups of experimental subjects. Contrast analysis helps us compare different groups of programmers.

For each performance measurement, the null hypothesis would claim that a given factor causes no variation in that measurement. The alternative hypothesis claims that it does. Two factors involved here are the faults and the assistant. Either factor or their combination can cause the variation. If the combination is the cause, the effectiveness of the assistant depends on the types of faults.

We consider the variation in the measurement significant if the p-value in the ANOVA result falls below our type I error-estimate (α) of 5%. To make this claim, we need an adequate sample size for the measurement under test. We consider our

sample size adequate if the power of the test $(1 - \beta)$ is more than 90%.[1] A small sample size can still yield a high power of test if the variation of the measurement within the group is very small, but variation between the groups is very large.

2 Pilot Study 1

In this study, the assistant is the oracle of the program under test. Our oracle is the Unix system administrator who maintains the program Nu. He can answer any questions from the programmers except "What is the fault?", "Where is the fault?", and "How can it be fixed?". We nevertheless refer to this feature as the *all-you-can-ask* feature.

2.1 The Study

We studied the programmers' abilities to find and fix the fault. This study compared groups of programmers with and without oracle access. Eight programmers who participated were called S_1 to S_8. We randomly assigned two programmers to each fault-assistant combination. S_1, S_2, S_3, and S_4 worked with fault 1; S_5, S_6, S_7, and S_8 worked with fault 2. Only S_3, S_4, S_7, and S_8 had oracle access. Variations among these programmers appear in Appendix A.

To prevent any eavesdropping, the oracle was in the room next to the programmers' room. Programmers with oracle access worked in different rooms from those without the access. We observed them to make sure they did not interact.

2.2 The Results

ANOVA suggests that the fault, not the Nu's oracle assistant, causes significant variations in all five measurements.[2] The p-values for AC, AACLOC, TIME, ETIME, and SPEED are 0.045, 0.014, 0.029, 0.038, and 0.034, respectively.

All programmers who debugged fault 1, except S_2, found the correct fix in about two hours. Though S_2 located the faulty routine the fastest, he settled for the fix that had a side-effect. Only one programmer, S_7, found the correct fix for fault 2. With oracle access, S_7 took about $3\frac{1}{2}$ hours. S_5 and S_6, who received no assistance, arrived at the fixes with side-effects in about 2-$2\frac{1}{2}$ hours. S_8, who had oracle access, could not identify and could not fix the fault. He did locate the faulty routine, however. The performance comparison is shown in Appendix C.

2.3 The Findings

This study provided evidence against rather than for our original hypothesis. We observed few demands for verifying fault locations and program states. The programmer who made the most requests to confirm fault locations, ironically, could not fix fault 2. We could not tell, however, whether information alone helped the

[1] The β denotes a type II error-estimate.

[2] Note that the results "suggest" instead of "conclude." The reason is not just because our sample size was inadequate to support the claim that the fault causes the differences. Our small sample size bars us from testing two of the three ANOVA assumptions. The first assumption that required the experiment to be repeatable is met. The other two are the homogeneity of error variances and the normal distribution of the population from which the samples are drawn. As we have to assume that they are true, we have to exercise caution before we make inferences from the results. Despite this drawback, suggestions with statistical analysis backup are still better than guessing.

programmers, mainly because they asked very few questions. S_3 and S_4 each asked only 2-3 questions. S_7 and S_8 only asked 19 and 11 questions, respectively. Though the evidence is insufficient to reject our original hypothesis, it provided an unexpected insight into the needs of the programmers.

2.3.1 About the programmers' needs

Instead of asking for help to formulate good decisions related to faults, the programmers made decisions on their own and merely asked the oracle for confirmation. The only two common requests made by all eight programmers were the confirmation of their proposed fault identities and fault repairs. The programmers with no assistance still asked for such confirmations, though we did not reply. To our surprise, only one programmer, S_8, requested confirmation on his proposed fault locations.

In the case of fault 1 where S_3 and S_4 decisions were already correct, they did not have to ask any more questions. The same did not hold true for S_7 and S_8 who debugged fault 2. They asked very few questions before they formulated their decisions on faults. Most of their questions were posed after the oracle rejected their decisions and they had to look for alternatives.

The programmers needed to make more requests for help to benefit from the oracle. Our speculation identified the following additional needs to which the oracle should respond.

1. *Programmers need suggestions on what can help.*

   The programmers claimed that they did not ask many questions because they did not know what to ask. They did not always know what information could save them time or improve the accuracy of their decisions. Instead of asking the oracle for a program overview, both S_7 and S_8 spent 1-1$\frac{1}{2}$ hours examining the code before they began to ask questions. Neither S_7 nor S_8 asked for the routine that performed the task they found missing – the information that could have improved the accuracy of the repair for fault 2.

2. *Programmers need an easy-to-access assistant.*

   All programmers with the oracle access felt that it was inconvenient to frequently leave their debugging environment to see the oracle. $S_3, S_4, S_7,$ and S_8 said they would have asked more questions had the oracle been seated next to them.

3. *Programmers need unsolicited help.*

   When the programmers, like S_1 and S_8, suspected the wrong fault locations, their confidence convinced them that a request for confirmation was totally unnecessary. S_1 considered such request a trait of a novice and would not ask for confirmation even if he could access the oracle. S_8 took about four hours before he realized he needed help and asked for it.

2.3.2 About the assistant

The oracle in this study was passive most of the time. He could provide information only after the programmer requested help. One unplanned action from the oracle was for him to ask programmers questions. However, according to S_7 and S_8, the oracle's questions helped them more than the information they asked for. The oracle's questions helped S_7 dismiss a wrong fix and S_8 locate the faulty routine. These oracle questions also inspired programmers to make more requests for help.

Unfortunately, this study provided insufficient evidence for or against our original hypothesis. We had to conduct a follow-up study to stimulate the programmers to ask more questions.

3 Pilot Study 2

We extended the role of the oracle for Nu to allow him to take the initiative. He can observe, question, warn, and give information without the programmers' request. We refer to this oracle as an *active oracle*. We refer to the oracle in the previous study as a *passive oracle*. The oracle is still the same person, however.

Because we expected that the programmers would ask for more help and receive more information from an active oracle, we wanted to test whether they would debug faster or more accurately than the group with no assistant. We also anticipated that the group with the active oracle assistant would perform better than the group with the passive oracle assistant.

3.1 The Active Oracle Assistant for Nu

Our *active oracle assistant* for Nu has two features: the all-you-can-ask feature and the observation-and-action feature. The *all-you-can-ask* feature is the same as the one for the passive oracle. The oracle can answer any questions, except direct questions on the identity, the location, and the repair of the fault. The *observation-and-action* feature lets the oracle observe when the programmer needs help, then initiate the appropriate help. The oracle has the freedom to question programmers, issue warnings, give hints, or provide other means to improve the programmers' decisions on faults.

To stimulate the programmer to ask more questions, the oracle sat next to the programmer and gave a program overview and a failure overview right at the beginning. His location not only permitted him to observe the programmer's progress and take the initiative, it also made the oracle easier to access. His overview established a context for the programmer to ask questions. The program overview covered the general functionality of Nu, the input and output, the global data variables, and the data files. The failure overview reiterated the information in the README file (an error-revealing test case, the nature of the failure, and the description of other failing conditions). The oracle also went through a sample run of an error-revealing test case.

3.2 The Study

Because fault 1 was too easy to find and fix, we only studied fault 2. We studied two more programmers, S_9 and S_{10}. Both met the same requirements mentioned in Section 1.2. To prevent S_9 and S_{10} from getting clues from previous participants, we kept the identity of S_1-S_8 from them, and vice versa.

The active oracle worked with S_9 and S_{10} on a one-on-one basis. After the oracle's overview, the session was open for questions and answers both ways. Their performance was compared with that of S_5 and S_6 who received no assistance and that of S_7 and S_8 who received assistance from the passive oracle.

3.3 The Results

ANOVA suggests that the group with Nu's active oracle assistant debugged Nu more accurately than the group with no assistant. The difference is significant

because the p-values for AC and AACLOC are 0.033 and 0.001, respectively. Our sample size of two per group is adequate to support this claim because the power of test is greater than 97% for both measurements. Though S_9 and S_{10} were the fastest among those who debugged fault 2, we needed a sample size of five to confirm a significant variation in SPEED.

The comparison of the performances of all programmers who debugged fault 2 is shown in Appendix C. The contrast analysis suggests that the group with the active oracle performed better than the group with the passive oracle assistant. It yields significantly better AACLOC (p = 0.027), TIME (p = 0.008), and SPEED (p = 0.017). Unfortunately, the sample size is inadequate to support this claim with statistically significant confidence.

3.4 The Findings

This study still offered more evidence against our original hypothesis. The combined demands for verifying fault locations and program states is only 8%. The high demands in both tasks did not correlate to the improved performance.

This study did provide a wealth of information on effective debugging assistance, however. The active oracle succeeded not only in making programmers debug twice as fast as those with the passive oracle, but also in increasing the number of programmers' questions seven-fold. The following subsections discuss the types of demands programmers made and how the oracle responded to them.

3.4.1 About the programmers' needs

We categorized the programmers' requests into *confirmation* and *explanation* requests. Each confirmation request is either a hypothesis statement or a yes-or-no question. Each explanation request is one of the typical "what", "when", "where", "why" and "how" questions. We studied the number of requests made, what the requests asked for, and how many times the programmers presented wrong hypotheses or proposed wrong decisions.

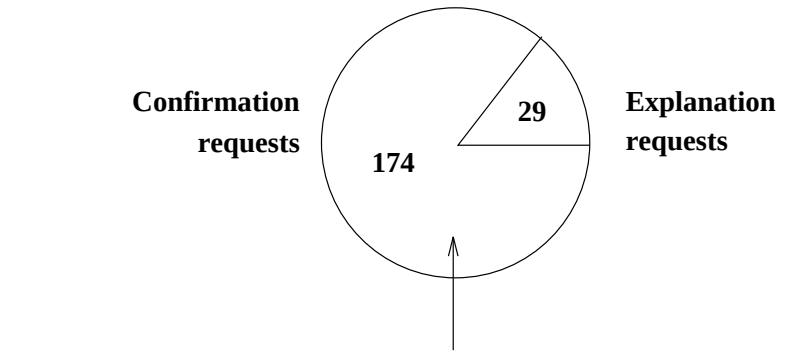
The results in Figure 2 show that 85% of $S_7 - S_{10}$'s requests were for confirmation. Their requests to confirm their understanding of the program were three times higher than their requests to confirm fault-related decisions and the intermediate correctness decisions combined. *Fault-related decisions* included the decisions on the identity, the location, and the repair of faults. The *intermediate correctness decisions* included the correctness of data value, program parts, program behavior, and conditions leading to failure.

Programmers needed help to formulate fault-related decisions. Two thirds of the fault-related decisions were wrong, but all of their intermediate correctness decisions were right. To improve the accuracy and speed in formulating fault-related decisions, the programmers needed two major things: help at the right time and adequate understanding of the problem.

1. *Assistance at the right time*

   Programmers needed both the information and the question at the right time. When the oracle questioned the programmer at the time the programmer misunderstood something, he cleared up the misunderstanding that could lead programmers to waste time.

   - *The right timing of information could save debugging time.*
     The right information at the right time can save time – as it can eliminate programmers asking irrelevant questions or using inefficient schemes to



total:wrong		total:wrong		total:wrong	
data	43: 12	fault repair	12: 10	data	11: 0
behavior	33: 6	fault identity	7: 3	behavior	6: 0
functionality	29: 7	fault location	5: 3	failing condition	5: 0
data handling	13: 3			code	1: 0
parameter	3: 1				
data dependency	3: 0				
design decision	1: 0				

Figure 2: The requests made by $S_7, S_8, S_9, and S_{10}$

derive the same information. A program slice was a good example. Had it been given at the beginning of the fault-finding phase, programmers could save debugging time.

A program slice is either static or dynamic. A static slice includes all statements that can affect the variable [Wei84]. A dynamic slice includes all statements that actually affect the variable with respect to the test case under investigation [Agr91]. If programmers look for program slices when they try to locate the faults, they would not waste time investigating irrelevant locations.

Not all programmers started off identifying erroneous variables and identifying their program slices, however. Some asked questions irrelevant to the code that the erroneous variable depended on. Those who looked for a program slice did not always use an accurate or an efficient method to define a slice either. To find routines leading to the failure, they usually relied on two Unix commands: *grep* and *ctags*. Unfortunately, the grep on variable names yielded a rough and inaccurate static slice of the variable. Their grep-slice was even more inaccurate if they were looking for a dynamic slice. S_{10} tried to identify the program slice with an inefficient method. He spent over one hour trying to define program slices by finding what did not belong in them. His quest ended when the oracle gave him the routine-level slice (similar to the one shown in Figure 3) as the hint. Had he ask for this hint earlier (as S_9 did), he might not have wasted time asking almost twice as many as questions as S_9 (94:54 questions).

- *The right timing of information could improve debugging accuracy.*
  The information is useful only when the programmers know how to use it. The timing of the information on an abort condition is a good example. When S_9 and S_{10} received this information in an overview, they did not even remember it. When the oracle told them again during the fixing phase, they considered it a vital clue to accurately repair fault 2.

2. *Adequate understanding of the problem*

   With or without an assistant, all programmers in our study who achieved 100% AC knew (1) the erroneous data variables, (2) the routines that affect them, (3) the location where their values first went wrong, (4) the correct values they should have, and (5) the consequence of their fixes. We consider these as a minimum set of facts required to find and fix fault 1 or fault 2 properly. All programmers who could not fix either fault, for example, did not know the consequence of their fixes. However, to fix fault 2, programmers needed one extra fact – the existing routine for the missing task. All programmers who did not know this fact failed to fix fault 2. Each of these facts can be viewed as a milestone in a model of a decision-making process for a debugging purpose.

What we learned about the types of information programmers wanted to understand was not a startling discovery. The top three types of information (see Figure 2) were the program data, program functionalities, program behavior.

Our observation also revealed preferences on the representation of information. The programmers liked to refer to data variables with respect to the semantic of its value. For example, the name-reference to a pointer pde varied from a copy of the original pde, an old pde, to a pointer to the password database entry. According to their answers to our AboutBug questionnaire, they initially defined the search space for fault as a list of .c files. As they searched for fault, this representation was refined to a list of routines. They did not define the search space in terms of statements until they thought they found the fault.

3.4.2 About the assistant

Because Nu's oracle helped the programmers more when he was active, the location of the active oracle, the observation-and-action feature, or both contributed toward improving debugging time and accuracy. Both factors stimulated a higher quality and quantity of questions, which made programmers use and benefit more from the all-you-can-ask feature.

Below is a discussion on the common information, questions, and observed events that aided the debugging decision-making processes. We discuss the *all-you-can-ask* feature under its two subfeatures: the explanation feature and the confirmation feature.

The explanation feature

Though answers to the explanation requests varied, the answers to requests concerning program behavior deserves attention. Because such requests were often phrased as a "What-If" question (e.g., "What happens when (specified condition) occurs?"), the answers were both actual and hypothetical. The behavior was often explained in terms of the consequence of the specified conditions. One sample question was "What would the program do if it received the abort command at the top-level?"

The confirmation feature

To respond to a confirmation request, the oracle indicated yes with reasons why or no with criticisms. To indirectly criticize, the oracle questioned a programmer to justify the decision. When a programmer gave his reasons, the oracle argued why they were wrong. In many cases, a programmer found his own flaws as he tried to explain. To directly criticize a programmer's decision, the oracle explained why he was against it. The information commonly used in his argument for fault-related decisions are as follows:

- *To criticize a decision on a fault location,* the oracle often used dependency information. The common argument is that the failure did not depend on the code at the specified locations. In another words, the oracle ruled out a location when it did not belong within a program slice of the erroneous variable.
- *To criticize a decision on a fault identity,* the oracle often explained why the failure was not the consequence of such fault.
- *To criticize a decision on a fault repair,* the oracle identified the undesirable consequence and the test conditions under which the program would fail.

Note that we use the term "criticize" instead of "verify" because the oracle's reason was not a proof of correctness. Rather, it gave the programmers reasonable doubt.

The observation-and-action feature

This feature helped to remedy or prevent potential problems that could affect debugging performance. The observation enabled the oracle to recognize events that called for his initiative. The action provided the appropriate help. We categorized the event-and-action pairs, or *rules* into three classes: remedial rules, preventive rules, and promotional rules.

1. *The remedial rules*

   When the oracle recognized events suggesting potential problems, he took remedial action. The events may suggest that programmers suspected a

wrong location, focused on an irrelevant code, settled with a repair with side-effect, or misunderstood the program.

The remedy often began with a question, followed by information. The oracle asked questions to confirm his suspicion of the problem, to find out the programmers' assumptions and justifications, and to enforce schemes to overcome their fixations. The information was provided later, as the oracle argued with their justifications, answered their questions to clear up misunderstandings, and suggested alternatives.

We observed two common problematic situations that caused the oracle to take remedial action.

- The programmer may suspect a wrong location.
  When the oracle observed S_9 inspecting a wrong location for a while, he asked why S_9 was looking at that part. The oracle left S_9 alone when S_9 replied that he just wanted to understand what the routine did. At another time when S_9 did suspect the wrong place, the oracle shifted S_9's attention. He made S_9 step backward through the routine-level trace while he repeatedly asked S_9 to verify the values of the erroneous variables after each routine call. He did not stop until S_9 found the routine where the value first went wrong.
- The programmer asked too many irrelevant questions.
  When the oracle got too many questions concerning code that the erroneous variables did not depend on, he began to question the programmers about the program slice. Sample questions included: "What is the role of the (specified) variable?", "What happens when the (specified) condition occurs?", and "What does this routine do?". When the answer was wrong, the oracle explained that aspect of the slice. When the programmers did not know the answer, they asked. If a programmer's questions remained off-focus, as in case of S_{10}, the oracle gave the routine-level dynamic slice (similar to the one in Figure 3) as a hint. This hint made S_{10} ask questions that focused more on the slice.

2. *The preventive rules*

   The oracle recognized an opportunity to prevent commonly occurring problems before they wasted time or lead to inaccurate decisions. His preventive action was to offer the appropriate assistance. Both hints and questions were used. Two situations that the oracle helped prevent were:

   - A programmer took too much time reviewing the program he had never seen before.
     The oracle prevented this situation by giving an overview of the program before a programmer began to debug it. The overview established the context of the program early. It made S_9 and S_{10} start asking questions at least an hour sooner than S_7 and S_8.
   - A programmer reimplemented an existing routine.
     When a programmer indicated which task was missing, the oracle gave him a *data abstraction of the erroneous global variable* hint – which happens to be the source file that contained the reusable routine for that task. Both programmers immediately recognized the routine they could reuse for fault 2 repair. This hint could have helped S_5 make the accurate repair, as he decided against the right solution because he did not want to implement a new routine.

3. *The promotional rules*

The oracle recognized the opportunity for a programmer to use certain tools or features to improve his debugging time or accuracy. His action was to stimulate them to use the features.

An example is when a programmer did not make any fault-related decisions for a while. The oracle asked questions to stimulate the programmers to make hypothetical decisions. This gave the oracle more opportunities to help them study the consequence of such decisions. Sample questions included: "What are the possible causes of failure?", "What is the correct value of variable?", "How could the problem be fixed?".

The findings from this study do not support our original hypothesis. In our hypothesis, we assume that information alone provides the necessary assistance. If this assumption holds, then information from the oracle should be sufficient. Perhaps S_9 and S_{10} performed better than $S_5 - S_8$ because they asked more and knew more. Perhaps the location of the active oracle alone, not his observation and assertive action, was responsible for the increase in the number of questions. In another words, the timing of the information and the customized assistant might not matter.

To settle this dispute, we devised two alternatives to an active oracle and tested them in our follow-up study. This test should also help us find an automatable replacement of an active oracle assistant (which is not always available in practice). Without an oracle, however, neither alternative could support the all-you-can-ask feature.

4 Pilot Study 3

In place of the all-you-can-ask feature, we summarized debugging information from previous studies into a set of hints. We made sure that the hints covered the information programmers asked, in the abstract representations that match their perceptions. These hints were intended to help the programmers understand the program, formulate better fault-related decisions, and self-criticize their decisions.

We envisioned two forms of assistant. The *Information-only (I-only) assistant* gave away all hints simultaneously. The *Observation-Information-Question (OIQ) assistant* gave away both hints and questions at the appropriate time.

We wanted to compare the performance among the groups of programmers that debugged fault 2. In particular, we wanted to test whether the group with the OIQ assistant debugged faster or more accurately than the group with the I-only assistant. We wanted to see which of the two assistants can improve the debugging performance just as well as the active oracle assistant. We also wanted to make sure that the programmers in each group debugged faster or more accurately than the group with no assistant.

4.1 The Information-only Assistant

The information-only (I-only) assistant gave all debugging hints on paper at the same time. Three categories of hints are the overview hints, the slices-related hints and the fault-related hints.

The **overview hints** include the verbal program overview and the failure overview. In our study, the *program overview* covered the overall functionality of the program, its input/output, its data files, and its global variables. The *failure overview* covered the tasks involved in the erroneous transaction and the nature of program failure.

The **slice-related hints** define and explain the search space for a fault. Our study included the following slice-related hints.

- *The statement that prints the erroneous output*
  This hint should help identify erroneous output variables. Note that this hint is suitable when the program terminates with the wrong output. Other forms of failure may need other hints to serve the same purpose.
- *The dynamic slice*
  This hint should help explain dependency between the code and the failure. It lists statements that affect the specified output variable in the given test case [Agr91]. We tag line numbers, routine names, and the file names to the slice for reference purposes. A programmer can extract from this hint: data dependency, control dependency, and statement dependency information.
- *The calling path*
  This hint should help explain the functionality on the dynamic slice. It lists all routines in a dynamic slice (indented according to the execution sequence) with two tags: the corresponding input command and the file name for each routine. Figure 3 displays this hint.
  The study by Soloway and Ehrlich [SE84] indicates that programmers understand the program functionality by recognizing program plans. A program plan relates the code pattern with its concept. Because a program slice is one possible form of a program plan [Ris92], it could help programmers understand the functionality. However, it should be easier to recognize a plan from a one-page routine-view of a slice than a multiple page statement-view of a slice.

```
cmds()  [main.c]

   case  CNTRL_F :
         find() [cmd.c]
              find_user() [cmd.c]

   case  CNTRL_E :
         edit( )  [cmd.c]
              copy_pde( )  [pde.c]

                   case  CNTRL_E:
                         edit_map( ) [map.c]

                              case 'a':  case 'h':
                                   add_host_item( )  [map.c]
                                        add_home_dir( )  [homes.c]

                              case CNTRL_T:

                   case CNTRL_T:
                         if  (delete)
                              copy_pde( )  [pde.c]
                              free_pde( )  [pde.c]

   case  CNTRL_T:
          save()  [cmd.c]
               build_home_dir()  [homes.c]
```

Figure 3: The calling path for $Home_dir_list \rightarrow pde$ in $build_home_dir()$

- *The routine-level trace*
  This hint helps explain the execution behavior and data on a dynamic slice. It lists the program states as if break points were set at the return of each routine call in a dynamic slice. The trace includes only the changes in values of the output variables and other variables that affected them.

To make the trace more readable, the raw values are replaced or accompanied by their interpretation. Aliases of the data are also presented. For example, a trace may list *"startpde (in cmds()) = an address of a password database entry", "pde (in edit()) = startpde", "Home_dir_list → pde contains a copy of pde."*

The **fault-related hints** help programmers formulate, confirm, or criticize their decisions on the identity, the location, and the repair of faults.

- *Test data set description*

  This hint is the next best thing to knowing the consequence of the fixes. It describes the combinations of conditions related to the erroneous transaction. The programmers can use this description to form test cases to test the program themselves.

- *Data abstraction of the erroneous global data*

  This hint should help the programmer find reusable code to repair the program. It includes the routine name, parameters, and its header comment. This hint includes the routine $Change_home_dir()$. The call to this routine is needed for fixing fault 2.

We intentionally did not question programmers in order to avoid the use of questions as form of assistance. Thus, we had no feedback on which variable was erroneous. We had to give the slice-related hints for both output variables: $Home_dir_list \rightarrow pde$ and $Home_dir_list \rightarrow fsl$. With no observation to determine timing for hints, the programmers were free to pick which hints to use and when to use them.

4.2 The Observation-Information-Question Assistant

This assistant is modeled after observation-and-action feature of the oracle. The OIQ assistant uses observation (O) to recognize the problems or opportunities to improve debugging performance. The action here is limited to providing debugging hints (I) and questions (Q). Though the same categories of hints as in Section 4.1 are given, the hints would be customized based on programmers' answers to our questions. The programmer's decisions on which output variables are erroneous, for example, determined which program slices he would receive.

The assistance is customized for each programmer. The observed events and the programmers' feedback determine which hints or questions each programmer will receive. We define four rules to observe and to act on the following four trigger events.

1. A programmer entered the fault-finding phase.

   First, we gave the overview verbally. Afterward, we asked "What are the erroneous output variables (if any)?" and gave the statement that prints the erroneous output. Once the programmer identified the erroneous variable(s), he received the calling paths, the dynamic slices, and the routine-level trace of that variable(s).

2. A programmer suspected a wrong location.

   If this event occurred after the programmer already had time to review the routine-level trace, this event would trigger us to suggest a strategic inspection of the trace. We made the programmer forward step through the routine-level trace while we repeatedly asked "Is this program state correct?". Once the programmer started to doubt, we switched to backward stepping. We

stopped when the programmer identified (what he thought was) the first erroneous program state.

This feedback enabled us to generate one extra hint on a fault location. Once a programmer identified the first erroneous state and the last correct state, we identified the routines and the statements that were trapped between these two states as our *guessed fault location*.

3. A programmer claimed that he fixed the fault.

   This event would trigger us to ask "Does your fix work with other test cases?" and give the test data description. This question was repeated for each combination in the description.

4. A programmer identified the missing task.

   This event would trigger us to ask "Do you know of existing code to fix the problem?" and to introduce the data abstraction of the erroneous global data that contains the reusable routine.

These rules are intended to make sure programmers can find most of the facts we identified as milestones in a debugging decision-making model in Section 3.4.1. The first rule is a preventive rule, intending to prevent programmers from wasting time investigating code irrelevant to the program failure. This rule forces a programmer to assimilate the information on the erroneous variables and the slice of the erroneous variables. The second rule is a remedial rule, intending to overcome a programmer's fixation and to suggest other possible locations for faults. This rule forces a programmer to assimilate the information on the first location where the data value first became erroneous. The third rule is also a remedial rule, intending to make a programmer self-criticize his own decision on fault repair. This rule forces a programmer to realize the consequence of his repair. The fourth rule is a preventive rule, intending to save time to rewrite the code and improve the accuracy of the repair. This rule forces a programmer to recognize the existence of reusable code.

4.3 The Study

We studied four more programmers: S_{11}, S_{12}, S_{13}, and S_{14}. S_{11} and S_{12} received the OIQ assistant. S_{13} and S_{14} received the I-only assistant.

Besides all requirements mentioned in section 1.2, the programmers must have met more group-requirements. One person in each group (S_{12} and S_{13}) must never have debugged others' programs and must not have written a C program for at least three months. The other person (S_{11} and S_{14}) must have debugged somebody else's programs before and must have written a C program within the same week of our study. We did this intentionally to see if our assistants are effective enough to overcome these differences.

All four programmers worked with the second faulty version of Nu. The material, the debugging environment, monitoring process, and performance measures remain the same as in the first two studies.

After the 15-minute verbal overview, we gave S_{13} and S_{14} the stack of hints and let them work on their own. We gave the first pair of questions and hints to S_{11} and S_{12} right after the overview. The rest followed when we observed the trigger event.

Originally, we planned to present just the questions and the hints on paper. The questionpreceded the hints because we listed it on a cover sheet of the hints. We had to change our plan when the programmers ignored our first pair of question-and-hints for identifying erroneous variables. Our immediate reaction was persuaded them to answer by telling them that their answers would allow us to give more

hints. This experience made us ask the questions verbally after we gave the hints for the rest of the session.

In the follow-up interview, we surveyed the programmers on the helpfulness of the hints and the assistants.

4.4 The Results

ANOVA suggests that the group with the OIQ assistant found and fixed the fault more accurately than the group with no assistant (S_5 and S_6). The p-values for AC and AACLOC are 0.033 and 0.003, respectively. Our sample size of two is adequate to support this claim because the power of test for AC and AACLOC are both greater than 99%.

Performance variations among other groups that debugged fault 2 (see Appendix C) have either no statistical significance or statistical significance with inadequate sample size. Nonetheless, some observable differences or similarities are worth mentioning.

- The group with the OIQ assistant performed almost as well as the group with the active oracle assistant.

  With the OIQ assistant, the programmers took about $2\frac{3}{4}$ to $3\frac{1}{4}$ hours to arrive at the right solution – about 15 - 45 minutes longer than the group with the active oracle assistant. This difference in TIME is not significant. In fact, the contrast analysis found no significant differences between the OIQ group and the active oracle's group in all five measurements

- The group with the I-only assistant did no better than the group with no assistant.

  Programmers in both groups settled with the solutions with side-effects. However, with the I-only assistant, the programmers took about $3\frac{1}{2}$ to 4 hours – about twice as long as the group with no assistant! This difference in TIME is significant (p = 0.009).

- The group with the OIQ assistant debugged Nu faster and located the fault more accurately than the group with the I-only assistant.

  These differences are statistically significant. The p-values for AACLOC and SPEED are 0.01 and 0.04 respectively. This result suggests that the timing of assistance, the ability to customize assistance for individual programmers, or both matters.

4.5 The Findings

We found more evidence against our original hypothesis. When debugging information is offered with little regard on its timing and no help to assimilate it – as the I-only assistant did – it could make some programmers debug slower than those with no assistant. The assistance in the form of observations and questions – as offered by the OIQ assistant – was needed to make programmers benefit from the information.

4.5.1 About the programmers' needs

Programmers with the I-only assistant did not know how to assimilate some of the hints. Both S_{13} and S_{14} initially ignored the hints and just examined the program directly. After approximately two hours, both began to pay some attention to the hints. Though they had no trouble discarding useless hints (those related to correct

output variables), they did not use two key hints: the test data description and the data abstraction. S_{13} and S_{14} also disliked the timing of the hints. They would prefer to receive hints dynamically as they debugged the program.

Programmers with the OIQ assistant still requested confirmation of fault identity and the repair, though we refused to answer. Instead, we gave S_{11} and S_{12} the test data description hint and asked them if their fixes worked with these data. Though the programmers needed help to identify the fault location, they did not ask for confirmation. The help that worked required the assistant to intervene when the programmers developed a fixation on a wrong location.

The need for support for decisions on correctness of data value surfaced when we enforced our fault-localization strategy. The programmers needed help to maintain details about these intermediate decisions and their inter-relationships. When S_{11} and S_{12} had to verify a lot of program states, they forgot their earlier decisions. Sometimes, they could not verify a data value that was defined by the variable they claimed erroneous earlier. Sometimes, they could not even verify the same variable values they considered to be erroneous earlier.

4.5.2 About the assistants

The OIQ assistant seems to satisfy the programmers' needs more than the I-only assistant. One piece of evidence to support this is in the performance of the programmer who had no recent C-experience and no foreign program debugging experience. With the OIQ assistant, S_{12} performed just as well as S_{11}. With the I-only assistant, S_{13} debugged twice as slow (in SPEED, not TIME) and half as accurately as S_{14}.

Below is the discussion on the three types of assistance featured in this study: observation, information, and question.

Observation

We did not observe the programmers as closely as the oracle. First, the oracle kept programmers under constant supervision. We just checked on them every 10 minutes. Second, the oracle observed what programmers did on the screen, all the remarks they made, and even their body language. We just observed their verbal comments and their answers to the AboutBug questionnaire at the end of each hour.

The observation did help us recognize the need for additional assistance. When the programmers were unable to verify data values, we reminded them of their previous decisions, like "you said this was wrong earlier."

Information

Based on the rating and comments from $S_{11}, S_{12}, S_{13},$ and S_{14}, the three highest rated hints were the overview, the routine-level trace, and the calling path of the erroneous variable. Programmers said these hints saved them time because they no longer had to seek the same information on their own. They said the abstract representation of these hints promoted program understanding and improved the accuracy of their decisions. The format of the calling path (see Figure 3) received the most praise.

The common hints that the programmers used to support their fault-related decisions were as follows:

1. To locate the fault, all three views of a dynamic slice help. The order of preference is the behavior view (the routine-level trace) first, routine-view (the calling path) second, and the statement-view last.

2. To identify the fault, three hints that help are the failure overview, the trace, and the statement that printed the output.
3. To fix the fault, three hints that help are the test data description, the data abstraction, and the trace.

The lack of the timing in the I-only assistant caused some hints to lose their problem-prevention property. Though S_{14} needed and used a calling path, he spent an hour defining it himself before he realized the same hint was already available. Thus, this hint did not save him time.

The importance of the timing of the hints remained apparent with the OIQ assistant. Both S_{11} and S_{12} forgot that the fix must handle the abort conditions from the overview. We reminded them when we presented the test data description as they began to repair the program. The data abstraction was helpful when we brought it to the programmers' attention after they indicated which task was missing. This hint was ignored when we gave it as an extra hint during the fault-finding phase.

Questions

When we used the hints and questions in place of confirmation, the questions no longer served to identify the types of assistance needed. They succeeded, instead of preceded, the information, in order to help assimilate hints and generate more customized hints. With the questions, programmers with the OIQ assistant examined and used all our hints constructively. Without the questions, programmers with the I-only used very few hints. S_{14} used only one hint, the routine-level trace. The fact that S_{13} and S_{14} did not assimilate the test data description and the data abstraction may explain why their accuracy suffered.

One problem with a long series of questions was that they annoyed the programmers. Both S_{11} and S_{12} tried several times to make us stop our verification requests. They claimed that they already knew the fault location and perceived our questions as pointless. Though their feedback led them to the faulty location, both programmers discarded this information. They did not consider this hint until they used the test data description hint and realized that their repairs had side-effects.

5 Summary

We studied how expert programmers debugged a program with over 4300 executable lines with real faults of omission. In our attempt to find evidence to suggest that our original hypothesis is worth pursuing, we tested four types of assistants: Passive oracle, Active oracle, Information-only (I-only), and Observation-Information-Question (OIQ) assistants. The only two that helped improve debugging accuracy were the active oracle and the OIQ assistants.

Our results suggest that our original hypothesis is not worth pursuing because:

1. The demand to verify fault locations or data values was only 8% of programmers' requests to the oracle.
2. The programmer who made the most requests to verify fault locations was, ironically, the only one who could not fix fault 2 with the oracle assistant.
3. Information alone may not be an adequate form of debugging assistance. When debugging information is offered with little regard on its timing and with no assistant to help assimilate it – as the I-only assistant did – it could make some programmers debug slower than those with no assistant.

The major problems that deserve more attention are the *underuse* problems. Programmers did not benefit from our passive assistants – the passive oracle and the I-only assistants – because:

- Programmers underused the assistants.

  The programmers did not really know how to use an intelligent, but passive, assistant like an oracle. They did not always know what to ask for, when to ask for it – even when they needed help (e.g., when they were stuck at a wrong location). Programmers with the I-only assistant did not always know how or when the hints could help them debug. One of the programmers used only one hint and ignored the rest.

- The assistant underused the knowledge of the programmers.

  The passive oracle underused the knowledge of the programmers as he could not observe them. He would not know that the programmers misunderstood something until they asked for help. Thus, he could not clear up their misunderstandings early enough to prevent them from wasting their time. The I-only assistant did not ask for the programmer's belief on the correctness of the output variables. Without that feedback, we had to give the hints related to the slices of both the correct variable and the erroneous variable. Though both programmers successfully discarded the hints related to the slice of the correct variable, we still believe that the extra, irrelevant information could distract programmers.

The two active assistants that worked – the active oracle and the OIQ assistants – offered help to mitigate the underuse problems as well as respond to programmers' demands. Their features are as follows.

1. *Confirmation feature*

   The confirmation requests constituted 85% of the total requests that programmers who debugged fault 2 made to the oracle. We noticed that expert programmers liked to arrive at a tentative decision or understanding first. The hypothesis or a tentative decision in a confirmation request helped the oracle realize the flaw in a programmer's belief. The rationale behind the oracle's judgment corrected that misunderstanding. Once a programmer realized that he was wrong, he asked for more help.

2. *Explanation feature*

   The oracle provided the user-requested explanation. This feature is an important companion to the confirmation feature, as programmers did not always understand what the oracle told them.

   The OIQ assistant provided the system-initiated explanation – in terms of hints. All programmers who received the hints indicated that they could not have debugged as fast (if at all) without them. Yet, such explanation alone did not guarantee the improvement of accuracy – the evidence is the failure of the I-only group.

3. *The observation-and-action feature*

   This feature mitigates the underuse problems by providing unsolicited and customized help at the appropriate time. Both assistants that worked – the active oracle and the OIQ assistants – shared this feature.

   To provide the unsolicited help, the assistant first recognized the situations under which it can (1) prevent potential problems, (2) remedy current problems, or (3) promote its assistance. The appropriate help may take the forms of questions, explanation or hints.

According to programmers, the oracle's questions contributed more to their success than their own questions. To prevent or remedy problems, the oracle questions assessed each programmer's knowledge in order to design customized help for him. Some of the roles they played were surfacing wrong assumptions, eliciting and structuring the problem, and enforcing a scheme to overcome their fixation.

Questions were used to promote assistance in two different ways. The oracle used them to stimulate programmers to hypothesize about the program and the fault. The OIQ assistant used them to customize as well as assimilate the hints. The feedback on the correctness of output variables cut the number of slice-related hints in half. A string of questions helped programmers assimilate the alternatives for fault location and fault repair from the given hints.

To get some ideas on the benefits of these features, we compared the failure rate. We found that 83% of the programmers in groups with neither the observation-and-action assistance nor the confirmation assistance (that is, the group with no assistant or with the I-only assistant) failed to fix the fault properly. For the group with the confirmation and explanation assistance (that is, the passive oracle group), only 25% failed. The failure rate is 0% for the groups with the observation-and-action and explanation feature (that is, the active oracle and the OIQ groups). The active oracle groups debugged slightly faster than the OIQ group, as the programmers also received the confirmation assistance.

We must emphasize that our results only "suggest" the above findings. Because our original intention was to find if our research hypothesis is worth pursuing, our studies were too small to confirm them. For that, we need larger experiments with more programs, faults, and programmers.

6 Related Work

Several empirical studies in other disciplines obtained similar results as ours. Research findings in decision support systems support the same problems we recognized in debugging assistants. Research findings in critic systems indicated the need for similar types of assistance we found helpful.

A *decision support system (DSS)* is a computer-based system which has the objective of enhancing the overall effectiveness (e.g., by increasing reliability, accuracy and efficiency of obtaining relevant information) of decision makers [JWF87]. Conventional DSSs act as passive partners in decision-making. They are passive because they merely place a set of useful facilities at the disposal of a decision maker and expect that the decision maker will somehow exploit these facilities effectively for decision-making. They cannot take initiative — they can only respond to user requests [Rag91].

Passive assistants provide a weak form of support that does not exploit the full power and potential of computer-based support. Empirical studies in the problems users encounter in dealing with high functionality computer systems indicate that the users do not know:

- what tools can help [FLS85],
- when to use the tools to help [FKF+89],
- how to apply the results that the tools produce [BFN86],
- how to adapt the tools to their specific needs [Fis87], and

- how to improve their situation with assistance from knowledgeable agents [Ree90].

As a result, passive DSSs were underused [FM91]. At the opposite end, the expert systems, which aim to derive the decisions autonomously, often underuse the knowledge of their users [FM91]. Because they often lack the knowledge required to cover the complete problem domain, interaction with the human is inevitable. Yet, the systems de-skill the human by treating him as a mere supplier of data. Researchers in DSSs [FM91, Rag91] and our findings agree that merely providing more information would not solve these underuse problems.

To use both the human and the machine knowledge to its full potential, recent research in DSS moved toward combining the conventional DSS with components of expert systems – making the assistant active [Rag91, JWF87]. *Active decision support systems (ADSS)* which emerged from this combination provide a cooperative problem solving environment. ADSSs provide tools to actively participate in the decision-making process, and decisions are made by fruitful collaboration between the human and the machine [Rag91]. Like our active oracle assistant for Nu, both the user and the system can take the initiative.

An ADSS can be considered a decision-making critic, as it shares the same characteristics and the same goals as the critic systems. *Critic Systems* support users in performing their own activities, provide the information only when it is relevant, and interfere only when the user's plan, action, or product is significantly inferior. Critics work under the assumptions that a user is competent enough to generate a product, determine a course of action, or make a decision by himself.

A traditional critic offers assistance similar to our confirmation feature. The psychological research by Lange and Harandi [LH85] suggests that expert users like to solve the problem on their own first before they consult an expert system. In response to such habit, a critic acts as a complement of an expert system. Instead of trying to solve the the problem autonomously, a traditional critic, like Miller's ATTENDING [Mil83], becomes operative only after the user has a tentative decision. After the system asks the user for the information on the details of the problem and for his justifications, the system reconstructs a plausible decision-making process using its knowledge base and internal models, and identifies potential problems and possible improvements. Silverman characterizes such a traditional critic as passive and after-task [Sil92].

According to Fischer and Mastaglio [FM88, FM91], a critic should also offer a user-directed explanation feature. Because the users may not fully understand the system's comments, the users must be able to question the system for further explanation.

A better ADSS or a critic system, according to several researchers [Sil92, FM91, Mil88, Rag91], should offer assistance like that of our observation-and-action feature. Such critic monitors a decision making process and provides unsolicited help or criticism when appropriate. A critic's advice is easier to understand if a system treats a user with respect to his knowledge and conceptual view [FM88].

Silverman's Principle 1 states that a critic should have a library of functions that serves as error-identification triggers, and influencer, debiaser, and director strategies. The *influencer* strategies help prevent biases before they occur. The *debiaser* strategies help correct the biases or errors after they occur. The *director* strategies help promote the use of a tool. All strategies could be activated before, during, and after tasks. This is an improvement over the traditional feature that offers an after-task debiaser alone.

Principle 1 is supported by the results from Silverman's experiments with over

one hundred participants on the largest real-world critic system, TIME[3] [Sil91], and his recent empirical studies with over fifty participants on sample critics for statistical problems [Sil92]. In the latter one, his results are strikingly similar to ours.

1. The users can be subjected to bias without a critic.
   Without a critic, 82% of the users (including statisticians and graduate students in statistics) failed to solve bias-prone statistical problems. In our studies, 83% of the programmers in the groups with no confirmation and no observation-and-action feature settled with the solutions with side-effects.
2. A critic that offers only a passive, after-task debiaser is not always adequate.
   With these traditional features of a critic, 31% of users still failed. In our studies, the failure rate is 25% in the groups that received only the confirmation and explanation assistance.
3. A critic that offers before-, during-, and after-task debiaser, influencer, and director can optimize the accuracy in the solution.
   The failure rate drops to 0%. We observe the same rate in groups that received observation-and-action assistance.

Current application domains of Critic Systems or ADSSs include the medical domain, business management, circuit design, ship design, kitchen design, knowledge acquisition, software specification, and programming in Lisp. The parallelism observed here suggests that debugging is a new application domain for a critic system.

7 Future Work

Our preliminary results, with supportive evidence from research in decision support systems and critic systems, contributes to the state-of-the-knowledge in debugging a list of suggestions and problems that deserve further investigation.

- One way to understand the complexity of a debugging problem is to view it as a decision-making process. The research in decision support systems and decision science could bring up currently overlooked issues.
- Problems of programmers underusing an assistant and vice versa hinder a debugging assistant's ability to improve debugging time or accuracy.
- A debugger is not likely to overcome the underuse problem by providing more information.
- A debugging assistant that criticizes debugging decisions as well as the decision making process has the potential to significantly improve debugging performance.
- To mitigate the underused-assistant problem, a debugging assistant should possess knowledge on some conditions under which it becomes helpful. Such knowledge would permit the assistant to provide unsolicited help when the timing is right.
- A debugging assistant can improve its effectiveness if it learns of the programmer's knowledge and customizes its assistance to augment that knowledge.

[3] TIME supports US Army personnel at 17 sites nationwide who must write hundreds of decision papers per year, one for each new piece of equipment the Army buys. It contains 1,500 rules, 2,000 note cards and 300 analogs to influence, debias, and direct the Army personnel to better balanced decision papers.

- The role of questions as a form of debugging assistance deserves further investigation.

Our future work is to enhance our prototype debugger, Spyder [Agr91], with a debugging oracle assistant to help programmers decide on fault locations based on our findings. To evaluate the quality of this assistant, we plan to conduct experiments on Spyder with more programs, more programmers, and more faults. The results of such experiments should yield a broader inference space to validate the effectiveness of our debugging assistant. Whether or not our automated assistant can be as effective as a human-oracle assistant remains to be seen.

8 Acknowledgment

This study was made possible by the generosity of several volunteers. We would like to thank Dan Trinkle for his help in preparing the experimental material and for acting as the oracle. We also thank Anupam Joshi, Vinod Anupam, Michael Beaven, Steve Chapin, Rajiv Choudhary, Jindon Chen, Mei-Hwa Chen, Ling-Yu Chuang, Sharon L. Decker, Patrick A. Muckelbauer, Victor T. Norman, Andrew Royappa, Maryjane E. Scharenberg, and HongHai Shen for their participation.[4] We also thank Sean X. Tang for answering some statistical questions and Ginny Atkinson for proofreading earlier drafts of this paper. We would like to thank Dr. H. E. Dunsmore for the analysis program to compute some software metrics for Nu. Lastly, we would like to thank Dr. Rich DeMillo for suggesting ideas for the initial study.

References

[Agr91] Hiralal Agrawal. *Towards Automatic Debugging of Computer Programs*. PhD thesis, Purdue University, West Lafayette, IN, 1991.

[BFN86] H.-D. Boecker, G. Fischer, and H. Nieper. The Enhancement of Understanding Through Visual Representations. In *Human Factors in Computing Systems, CHI'86 Conference Proceedings*, pages 44 – 50, Boston, MA, April 1986. ACM.

[Fis87] Gerhard Fischer. Cognitive View of Reuse and Redesign. *IEEE Software*, 4(4):60 – 72, July 1987.

[FKF+89] G. Fischer, W. Kintsch, P. W. Foltz, S. M. Mannes, H. Nieper-Lemke, and C. Steven. Theories, Methods, and Tools for the Design of User-Centered Systems. Technical report, Department of Computer Science, University of Colorado, Boulder, CO, March 1989.

[FLS85] G. Fischer, A. C. Lemke, and T. Schwab. Knowledge-Based Help Systems. In *Human Factors in Computing Systems, CHI'85 Conference Proceedings*, pages 161 – 167, San Francisco, CA, April 1985. ACM.

[FM88] Gerhard Fischer and Thomas Mastaglio. Computer-Based Critics. In *Proceedings of the 21th Hawaii International Conference on System Sciences*, pages 427 – 436, 1988.

[FM91] Gerhard Fischer and Thomas Mastaglio. A conceptual framework for knowledge-based critic systems. *Decision Support Systems*, 7:355 – 378, 1991.

[4] The names are in alphabetical order, not the order of the subject numbers.

[IEE83] IEEE Standard Glossary of Software Engineering Terminology, 1983. IEEE Std. 729-1983.

[JWF87] M. Tawfik Jelassi, Karen Williams, and Christine S. Fidler. The Emerging Role of DSS: From Passive to Active. *Decision Support Systems*, 3:299–307, 1987.

[LH85] Rense Lange and Mehdi T. Harandi. Human Engineering Aspects of a Program Debugging Expert System. In *The IEEE Computer Society's Ninth International Computer Software and Applications Conference*, Chicago, IL, October 1985.

[Mil83] P. Miller. ATTENDING: Critiquing a Physician's Managment Plan. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5(5):449 – 461, 1983.

[Mil88] Fatma Mili. Framework for a decision critic and advisor. In *Proceedings of the 21th Hawaii International Conference on System Sciences*, pages 381 – 386, 1988.

[PS93] Hsin Pan and E. H. Spafford. Fault Localization Methods for Software Debugging. *Journal of Computer and Software Engineering*, 1993. (to appear).

[Rag91] Sridhar A. Raghavan. JANUS A paradigm for active decision support. *Decision Support Systems*, 7:379–395, 1991.

[Ree90] B. Reeves. Finding and Choosing the Right Object in a Large Hardware Store – An Empirical Study of Cooperative Problem Solving among Humans. Technical report, Department of Computer Science, University of Colorado, Boulder, CO, March 1990.

[Ris92] Robert S. Rist. Plans in program design and understanding. In *AAAI-92 Workshop Program - AI and Automated Program Understanding*, pages 98 – 102, July 1992.

[SE84] Elliot Soloway and Kate Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, 10(5):595–609, September 1984.

[Sha83] E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, 1983.

[Sil91] Barry G. Silverman. Criticism-Based Knowledge Acquisition for Document Generation. In *Proceeding of Conference on Innovative Applications of Artificial Intelligence*, Cambridge, MA, 1991. AAAI Press/MIT Press.

[Sil92] Barry G. Silverman. Building a Better Critic Recent Empirical Results. *IEEE Expert*, 7(2):18 – 25, April 1992.

[SV92] Eugene H. Spafford and Chonchanok Viravan. Experimental Designs: Testing a Debugging Oracle Assistant. Technical Report SERC-TR-120-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, 1992.

[Wei84] Mark Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

A The Programmers

	Years of programming experience	Years knowing C	Weeks away from programming in C	Number of CS classes taken that used C	Years in the graduate school	The largest C program the programmer ever wrote	Debug somebody else program before	The largest program the programmer debugged, but did not write
S1	13	7	1	7	3.5	5K	yes	> 100K
S2	9	9	0	12	6	10-25K	yes	25-100K
S3	9	9	1	10	5	10-25K	yes	5K
S4	9	5	0	10	3.5	25-100K	yes	10K
S5	17	8	0	7	3	1-5K	yes	1K
S6	13	7	52	10	7	5-10K	yes	> 100K
S7	10	8	0	6	5	5-10K	yes	5K
S8	7	5	0	9	4	7K	yes	5K
S9	17	6	52	5	7	5-10K	yes	25-100K
S10	10	9	0	3	7	25-100K	yes	> 100K
S11	7	5	0	7	4	1-5K	yes	10-25K
S12	9	5	156	5	5	1-5K	no	0
S13	6	5	12	14	4	1-5K	no	0
S14	34	5	0	10	3	1-5K	yes	10-25K

B The Measurements

The accuracy:

1. Accuracy in locating faulty routine (ACLOC):

$$ACLOC = \begin{cases} 0, & \text{if search space omits faulty routine;} \\ \dfrac{total\ faulty\ routines}{total\ routines\ in\ search\ space} & \text{Otherwise.} \end{cases}$$

   The search space consists of program parts that the programmer intended to examine to find the fault. At the end of each hour, the programmer divides the program into three regions: *must-look, may-look-later*, and *will-not-look*. The *must-look* region is where he suspects the bug to be. The *may-look-later* region is where he may look for the bug if none is found in the must look region. The *will-not-look* region is where he thinks the bug could not possibly be. The search space is the *must-look* region only if the faulty routine is in there. Otherwise it also includes the *may-look-later* region.

2. Accuracy in identifying the fault (ACID):

$$ACID = \frac{number\ of\ causes\ of\ failure\ identified}{total\ causes\ of\ failure}$$

   We define the fault identity as a chain of causes of the program failure. The chain for fault 1 and 2 are shown in Figure 1.

3. Accuracy in fixing the fault (ACFIX):

$$ACFIX = \begin{cases} 1, & \text{if } correct\ solution; \\ .50, & \text{if } solution\ with\ side\ effect; \\ 0, & \text{Otherwise.} \end{cases}$$

   A solution is a repair made on one of the causes of failures (see Figure 1). Thus, any repair that merely avoids the failures does not count. A print statement that echos the known correct output, for example, is not considered a solution.

4. Overall accuracy (AC):

$$AC = 33 * ACLOC + 33 * ACID + 34 * ACFIX$$

   AC is computed based on the last ACLOC, ACID, and ACFIX.

5. The average accuracy in locating faulty routine (AACLOC):
   AACLOC is the sum of ACLOC reported at end of each hour divided by the number of hours.

The Accuracy gained per hour (SPEED)

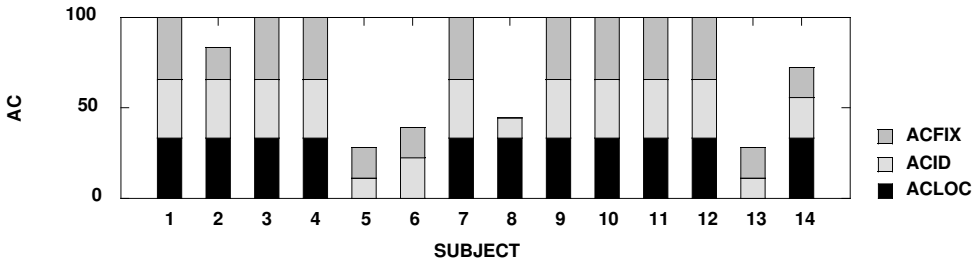
$$SPEED = 60 * \frac{AC}{TIME}$$

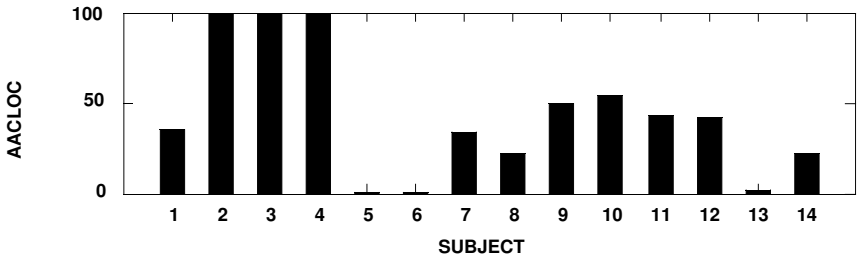
The time:

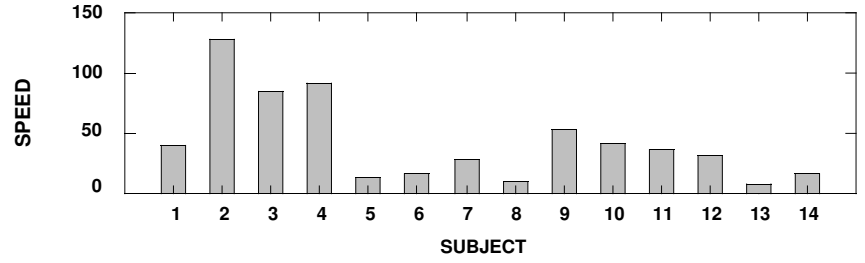
1. *The actual time (TIME):*
   TIME is measured in minutes. This excludes (1) the time to copy over the tar file, expand it, compile and run Nu for the first time, (2) the time to fill out AboutBug form and mail the script at the end of each hour, and (3) the break time. The consulting time with the oracle is included.
2. *The estimated time taken to fix the fault correctly (ETIME):*

$$ETIME = 100 * \frac{TIME}{AC}$$

C The Results

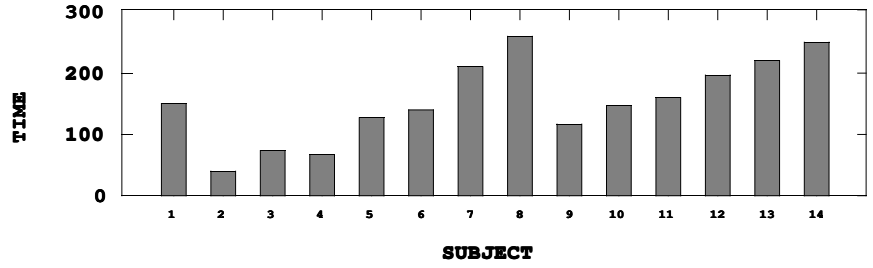


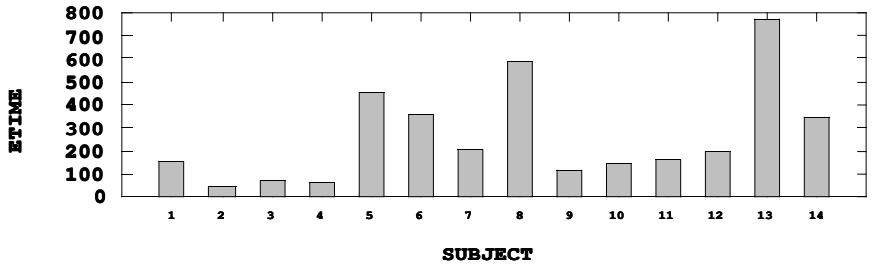




Fault	1				2									
Assistant	1		2		1		2		3		4		5	
Subjects	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Assistant: 1 = no assistant, 2 = passive oracle assistant, 3 = active oracle assistant
4 = OIQ assistant, 5 = I-only assistant





Fault	1				2									
Assistant	1		2		1		2		3		4		5	
Subjects	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Assistant: 1 = no assistant, 2 = passive oracle assistant, 3 = active oracle assistant
4 = OIQ assistant, 5 = I-only assistant