Pre-publication draft of a paper which appeared in the Proceedings of the Fourth Australian Society for Computers in Learning in Tertiary Education Conference (ASCILITE '91), pages 581-590

DEBUGGING USING PARTIAL MODELS

Philip A Smith, Geoffrey I Webb
Department of Computing and Mathematics Deakin University

Abstract
We present initial work on an expert system that will help users to debug their Pascal programs. This system asks the user questions concerning attempts to build a 'partial model' of the program - a model of those aspects of the program likely to relate to the error. This contrasts with previous systems in which a complete model of the user's program is built and compared to templates of correct versions of the program. The advantages of this approach are greater flexibility, greater student involvement in the debugging process and lower computational overheads.

Keywords
program debugging, partial models.

1 Introduction

Bradman is a debugging system for use with Pascal programs. This system is highly interactive, relying on the user to supply relevant information at certain times. We believe that the techniques discussed herein make it feasible to build a relatively inexpensive system that is still capable of providing the user with assistance with non-trivial Pascal programs. Bradman is still in the early prototype stage. We describe the principles on which it is based and how we expect it perform.

2 Previous Program Debugging Assistants

Many other systems have been built to assist in the debugging process. They show markedly different approaches reflecting the richness of the problem that they are intended to solve

Three recent systems are examined that each display different approaches to finding bugs and which typify the main directions of previous research.

PROUST (Johnson and Soloway, 1984 and 1985) is a knowledge-based system which obtains deep understanding of a program by reconstructing the user's intentions in writing the programs. It uses libraries of goals and plans where a goal is an aim which must be satisfied by the program and a plan is a means of implementing a goal.

PROUST takes as input a "goal description" of the program. That is, a list of goals which a program must satisfy to be successful. It parses the users program and compares it to the goal description. It compares each plan in its library which might satisfy the goal to the user's code. If a corresponding plan is found for each goal then the program is deemed correct.

Often, however, a goal will have no plan which matches the code exactly. PROUST must still decide which plan the user intended implementing. This is not always simple because two or three possible plans might resemble the code equally. If PROUST chooses incorrectly then its idea of why the user made the error will also be incorrect and the wrong error message will be reported.

Sniffer (Shapiro 1981) is a debugging assistant for LISP. Sniffer expects more help from the user than PROUST. The user must isolate the area of code in which be thinks the error has occurred. He must also provide the system with a description of how the error manifests itself. The system then takes over and attempts to find the error. Sniffer is modularised into a series

of experts (sniffers), each of which is capable of finding a narrowly-defined error. In 1981 one sniffer had been implemented, the "cons bug" sniffer which is capable of finding errors involving the "cons" function. A number of extensions had been planned. The system is augmented with two other tools

1. The time-rover, which maintains a history of the execution of the program. The time-rover can be used by the user when isolating the piece of code that contains the error. It is also used making a diagnosis.
2. The cliché-finder, which recognises algorithms within the code. The algorithms are recognised no matter how they were coded. The cliché-finder is used by the sniffers.

Sniffer relies on the user to provide it with a description of the problem before it calls the sniffers.

GPSI (Harandi 1983) is a language which can be used to write systems to debug compilation errors and certain run-time errors which show sufficient explicit symptoms. Knowledge is maintained in a forest of inference trees, the roots of which represent hypotheses or sub-goals of other sub-goals and other nodes represent such things as pieces of evidence, connective and operative nodes etc. Weighting of the evidence is used to cut off fruitless searches early and to decide which tree to search next. Further information can be obtained interactively from the student. Only a shallow understanding of the program is achieved.

3 Description of Bradman

Bradman will be a rule based expert system that will attempt to gain deep understanding of a student's program without using large amounts of computing resources such as libraries of goals and plans, time-rovers etc. It will ask the student to supply details of the program's goals and the methods used to achieve these goals, as necessary.

Bradman will need many pieces of information to make a diagnosis. While all of this information could probably be obtained by interrogating the user this is not desirable. The questions that need to be asked could often appear repetitive test the user's patience.

To augment the information obtained from the user we are working on two simple tools - a line parser and a statement recogniser. The line parser will be used to pick out important information (identifiers, operators etc.) in a given line of the program. The statement recogniser will be used to give Bradman an idea of what each statement is. For example, a statement beginning with keywords 'write' or 'writeln' is noted as an output statement. Hence, if Bradman needs to know where a certain variable was output it can use the line parser on the statements indicated by the statement recogniser.

Facilities will also be included that will give fuller explanations of questions These will be of two types One will explain more fully what a question means. For example, a user might want to know what we mean when we ask if an error is a "compilation error" The other will explain why a question is asked. This will generally mean giving a summary of the inferences that have led Bradman to the current position and what the question is trying to establish. These faci1ities must be requested explicitly by the user

When Bradman eventually gives a diagnosis of the problem it will give the use chance to alter the program and run it within the session. This enables the user to continue the session if there are further problems or if the diagnosis does not so1ve the problem satisfactorily.

It will be seen that Bradman uses a variety of techniques to home in on a particular problem. If, at a given stage of its investigations, there is only one option then it will accept this option as being true. If there is more than on possibility then it will look for further information to enable it to make a decision on which to pursue. It does this by consulting the program itself or by asking the user directly. If it s still unable to decide then it displays all of the possibilities to the user who is asked to decide for himself which option to follow.

Bradman attempts to construct a model of the user's intention when constructing a program, and to determine how the actual program differs from those intentions. However, unlike

PROUST, Bradman does not construct a complete model of either the user's intentions or the program's operation. Rather, it constructs incomplete or "partial" models, filling in only those aspects of a model that are needed to locate a bug. Further, the user is actively consulted during this process, helping him refine his own debugging skills while reducing the level of guesswork required by the system.

We'll now give an example of how we expect Bradman to work. We'll give a description of the programming problem to be solved, an example of a buggy program and an example of a dialogue that Bradman might have with the user regarding the error. An explanation of Bradman's reasoning behind each piece of dialogue will follow. The example problem and buggy program that we use is one used many times by the creators of Meno-II and PROUST. It should be noted that both of these systems found the bug using their own resources entirely.

3.1 Example

Problem: Read in a set of integers and print out their average. Stop reading numbers when the number 99999 is seen. Do NOT include 99999 in the average.

BUGGY STUDENT ATTEMPT

```
1/      program average (input, output);
2/
3/      var sum, num, count : integer;
4/            ave : real;
5/
6/      begin
7/            sum := 0; count := 0;
8/            read (num);
9/            while num<> 99999 do begin
10/                 sum := sum + num;
11/                 count := count + 1;
12/                 num := num + 1;
13/           end;
14/           ave := sum/count;
15/           writeln ('The average is', ave);
16/     end.
```

3.1.1 Dialogue

```
1/ What type of error does your program contain
   a. compilation error
   b. run-time system error
   c. logic error

   c

2/ In which of the following ways does the logic error manifest
   itself
      a. no output and no termination
      b. infinite output (no termination)
      c. wrong output given (program terminates)

   c
```

```
3/ As your program contains only one output statement there are
   two possibilities
      a. this output statement outputs an incorrect value
      b. another required output statement is missing from the
         program

         which of these possibilities would you like to explore
         first?

   a

4/ The only output statement is the writeln statement on line
   16. If it output wrong value then either
      a. its method for calculating the value must be wrong
      b. it must be executed at the wrong time

         which of these possibilities would you like to explore
         first?

   a

5/ The value output by the writeln statement at line 16 is
   obtained from the string 'The average is' and the variable
   'ave'.  Should these be printed by this statement?

   yes

6/ In that case, the variable 'ave' must contain the wrong
   value. 'ave' is assigned a value at line 15
           ave := sum/count;

      There are three possibilities
         a. That the expression 'sum / count' is incorrect,
         b. That 'sum' has been assigned the wrong value.
         c. That 'count' has been assigned the wrong value.

         which of these possibilities would you like to explore
         first?

   b

7/ 'sum' is assigned a value at line 12.
             sum := sum + num;
where 'sum' is being incremented by the amount 'num'.

Is this how 'sum' should be calculated?

   yes

8/ The variable 'num' has its value altered at two places
           line 8 / read (num);
           line 12/ num := num + 1;

Are these statements how you would expect 'num' to be altered?

   yes
```

```
9/ The second statement is contained within the while loop at
   line 9
      while num <> 99999 do

The condition 'num a 99999' suggests a sentinel test. A
sentinel test is where a number of values are read in from the
screen or from a file and are terminated by a predefined value
(usually a large integer).

Is
       num <> 99999
sentinel test?

   yes

10/  In that case a statement reading a value into num should
   be contained within the while loop. However, the only
   statement that alters num's value with the while statement
   is
      num := num + 1;

Was your intention that this statement read a new value into
'num' from standard input?

      yes

11/  Then that's your problem. You need to use a read
   statement every time you read a value from a file or
   standard input. Change line 12 to

      read (num).

12/  Does this solve your problem?

      yes

13/  Are there any other problems?
```

3.1.2 Explanation of Dialogue

Thee first two questions determine in what way the error manifests itself. Bradman is told that the program gives the wrong output. The questions 3 - 6 establish that the wrong output was given because the value in variable 'ave' was incorrect when the output statement at line 16 was executed.

We search through the program for places where the variable 'ave' has been altered. A variable can be altered in one of two possible places:

- In input statements
- In assignment statements with the variable as the left hand side.

The system finds one place where the variable 'ave' might receive a new value

ave sum / count; {on line 14}

The line-parser discovers that there are two further identifiers separated by a division sign. The variable 'ave' will have been given the wrong value either because the expression itself is

wrong (e.g. perhaps count / sum; was wanted) or because sum or count (or both) contain the wrong value.

The system has not got enough information to decide which of the above possibilities is correct. Hence it presents all three possibilities to the user who chooses what he believes to be the most promising. He decides that the expression 'sum/count' is indeed correct and decides to pursue the possibility that 'sum' might contain the wrong value.

If 'sum' contains the wrong value then it is because it has been given the wrong value at some point in the program. Bradman finds that sum is given a value in the line

```
sum := sum + num;                {line 10}
```

A similar process is followed for the variable 'num'. It is altered in two places

```
read (num)                {line 8)
num := num + 1; {line 12}
```

Bradman is aware that incrementing a variable by one often means that the variable is being used as a counting variable. It looks to see if the statement is contained within a while loop. It finds that it is contained with the while loop at line 9. The condition of the while loop is examined. The condition consists of an integer being compared to a large integer. This sort of comparison is often a sentinel. The user is asked if this is true.

The affirmative response enables Bradman to close in on the error. The sentinel test requires an input statement involving 'num' within the while loop. It is not there but an assignment statement involving 'num' is. The student is told this and asked if an input statement is required within the while loop.

3.2 Comparison of Bradman with the other Systems

3.2.1 Comparison 1 - Variability

Programs for all but the most trivial problems display a great deal of variability. As Johnson and Soloway (1986) note - if 200 students are asked to write a program then it is likely that 200 different programs will result (excluding collusion).

This wide degree of variability generally causes great problems for debugging systems. PROUST which gave very good results for simple problems lost accuracy for more complex ones. Possible reasons for this include an increase of "implicit goals" in the problem description *(*Johnson 1990). Even for simple problems it was not feasible to include plans and goals for every possibility. Johnson was looking at ways to enable users to explicitly tell PROUST which plans they were trying to implement.

Sniffer attempts to reduce the problem of variability by making the user isolate an area of code in which the error is likely to be. The user must also give Sniffer a description of the problem. This puts a large responsibility on the user which is probably not suitable for novices. Sniffer is limited by the number of individual sniffers at its disposal. It's applicability can be increased by adding more sniffers.

Both PROUST and Sniffer would appear to be potentially capable of finding most bugs in any program. For example, if PROUST had goals and plans to cover each possible method of implementing a program then it would be capable of finding all possible errors. However, each additional goal adds to the complexity and slows response times. A trade-off between the two issues is inevitable. A similar situation exists for Sniffer. Perhaps Bradman, with less computational overhead, will be able to handle more situations.

GPSI handles the problem by limiting the domain to those errors for which ex licit symptoms exist. Bradman will attempt to obtain a deeper understanding of the program than systems created by GPSI. Hence it would be expected to handle a wider variety of errors.

3.2.2 Comparison 2 - Interactivity

PROUST is not interactive It contains all the knowledge that it needs to make a diagnosis. Sniffer allows the user to make use of the time rover to isolate the area of code that the system will search. It gives the user no guidance on how to proceed with this however. Sniffer also

expects the user to provide it with a description of the bug. It is possible for the sniffers to ask the user to provide more information but Shapiro does not expand on this aspect. Sniffer would appear to be designed for users with more than novice experience in LISP. GPSI maintains a more conversational dialogue with the user.

Interactivity brings with it the problem of reliability of input. Obviously if the user supp1ies incorrect information then a system cannot be expected to make a correct diagnosis. Bradman will not expect the user to answer questions that would be unreasonable for a novice to understand and will provide help facilities to give users fuller explanations if required.

Another aspect is that of which bugs are handled. PROUST is designed to find all bugs in a given program. Bradman, Sniffer and GPSI, which expect the user to describe the bug in some way, will only be called upon when the user realises that a bug has occurred. Hence bad programming practice such as failing to initialise counting variables will be picked up by PROUST but not by the other three systems.

3.2.3 Comparison 3 — Complexity

Bradman will be less complex than either Sniffer or PROUST. As a result its computational overhead will be lower and its response times should be quicker.

4 Conclusion

We believe that it is possible to debug non-trivial programs without using large computing resources. Bradman is an attempt at implementing such a system. It will construct partial models of the user's programming intentions and determine how these differ from the implementation provided by obtaining information from the user and augmented by program analysis performed by some simple tools. It is anticipated that this process will not only assist the user by helping to identify a particular bug in a particular program, but also will serve as a form of apprenticeship learning, with the student learning to debug their own programs through active participation in the debugging process.

Bibliography

[1] Harandi, M.T., "Knowledge-based Program Debugging: A Heuristic Model" in Proceedings 1983 Softfair, 1983.

[2] Johnson, W.L., "Understanding and Debugging Novice Programs" Artificial Intelligence 42, 1990, pp 51-97.

[3] Johnson, W.L., Soloway. E., "PROUST: An Automatic Debugger for Pascal Programs" Byte, April 1985, pp 179-190.

[4] Johnson, W.L., Soloway. E., "PROUST: Knowledge-Based Program Understanding" IEEE, 1984, pp 369-380.

[5] Johnson, W.L., Soloway. E., "Intention-Based Diagnosis of Programming Errors" Proceedings of the National Conference on Artificial Intelligence, Austin Texas, 1986, pp 162-168.

[6] Shapiro, D.G., "Sniffer: A System that Understands Bugs" Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1981.